Oracle® Application Server Containers for J2EE

サービス・ガイド

10*g* リリース 2（10.1.2）

部品番号 : B15736-02

2005 年 10 月

Oracle Application Server Containers for J2EE サービス・ガイド , 10*g* リリース 2（10.1.2）

部品番号 : B15736-02

原本名 : Oracle Application Server Containers for J2EE Services Guide, 10*g* Release 2 (10.1.2) for Windows or UNIX

原本部品番号 : B14012-02

原本著者 : Alfred Franci

原本協力者 : Janis Greenberg, Mark Kennedy, Peter Purich, Elizabeth Hanes Perry, Sheryl Maring, Anirruddha Thakur, Anthony Lai, Ashok Banerjee, Brian Wright, Cheuk Chau, Debabrata Panda, Editor Ellen Siegal, Erik Bergenholtz, Gary Gilchrist, Irene Zhang, J.J. Snyder, Jon Currey, Jyotsna Laxminarayanan, Krishna Kunchithapadam, Kuassi Mensah, Lars Ewe, Lelia Yin, Mike Lehmann, Mike Sanko, Min-Hank Ho, Nickolas Kavantzas, Rachel Chan, Rajkumar Irudayaraj, Raymond Ng, Sastry Malladi, Stella Li, Sunil Kunisetty, Thomas Van Raalte.

Copyright © 2002, 2005 Oracle. All rights reserved.

制限付権利の説明

このプログラム（ソフトウェアおよびドキュメントを含む）には、オラクル社およびその関連会社に所有権のある情報が含まれています。このプログラムの使用または開示は、オラクル社およびその関連会社との契約に記された制約条件に従うものとします。著作権、特許権およびその他の知的財産権と工業所有権に関する法律により保護されています。

独立して作成された他のソフトウェアとの互換性を得るために必要な場合、もしくは法律によって規定される場合を除き、このプログラムのリバース・エンジニアリング、逆アセンブル、逆コンパイル等は禁止されています。

このドキュメントの情報は、予告なしに変更される場合があります。オラクル社およびその関連会社は、このドキュメントに誤りが無いことの保証は致し兼ねます。これらのプログラムのライセンス契約で許諾されている場合を除き、プログラムを形式、手段（電子的または機械的）、目的に関係なく、複製または転用することはできません。

このプログラムが米国政府機関、もしくは米国政府機関に代わってこのプログラムをライセンスまたは使用する者に提供される場合は、次の注意が適用されます。

U.S. GOVERNMENT RIGHTS

Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the Programs, including documentation and technical data, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement, and, to the extent applicable, the additional rights set forth in FAR 52.227-19, Commercial Computer Software--Restricted Rights (June 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

このプログラムは、核、航空産業、大量輸送、医療あるいはその他の危険が伴うアプリケーションへの用途を目的としておりません。このプログラムをかかる目的で使用する際、上述のアプリケーションを安全に使用するために、適切な安全装置、バックアップ、冗長性（redundancy）、その他の対策を講じることは使用者の責任となります。万一かかるプログラムの使用に起因して損害が発生いたしましても、オラクル社およびその関連会社は一切責任を負いかねます。

Oracle、JD Edwards、PeopleSoft、Retek は米国 Oracle Corporation およびその子会社、関連会社の登録商標です。その他の名称は、他社の商標の可能性があります。

このプログラムは、第三者の Web サイトへリンクし、第三者のコンテンツ、製品、サービスへアクセスすることがあります。オラクル社およびその関連会社は第三者の Web サイトで提供されるコンテンツについては、一切の責任を負いかねます。当該コンテンツの利用は、お客様の責任になります。第三者の製品またはサービスを購入する場合は、第三者と直接の取引となります。オラクル社およびその関連会社は、第三者の製品およびサービスの品質、契約の履行（製品またはサービスの提供、保証義務を含む）に関しては責任を負いかねます。また、第三者との取引により損失や損害が発生いたしましても、オラクル社およびその関連会社は一切の責任を負いかねます。

目次

はじめに ix

- 対象読者 x
- ドキュメントのアクセシビリティについて x
- このマニュアルの構成 x
- 関連ドキュメント xi
- 表記規則 xii
- サポートおよびサービス xiii

1 OC4J サービスの概要

- **Java Naming and Directory Interface（JNDI）** 1-2
- **Java Message Service（JMS）** 1-2
- **Remote Method Invocation（RMI）** 1-2
- **データ・ソース** 1-2
- **Java Transaction API（JTA）** 1-2
- **J2EE Connector Architecture（J2CA）** 1-3
- **Java Object Cache** 1-3

2 Java Naming and Directory Interface

- **概要** 2-2
  - 初期コンテキスト 2-2
- **JNDI コンテキストの構成** 2-3
- **JNDI 環境** 2-4
- **OC4J での初期コンテキストの作成** 2-5
  - J2EE アプリケーション・クライアントからの使用 2-5
    - 環境プロパティ 2-6
    - アプリケーション・クライアントからのオブジェクトへのアクセス 2-7
  - J2EE アプリケーション・コンポーネントからの使用 2-9
    - 同じアプリケーション内のオブジェクト 2-9
    - 同じアプリケーションにないオブジェクト 2-10
- **JNDI 状態レプリケーション** 2-11
  - JNDI 状態レプリケーションの有効化 2-12
  - JNDI 状態レプリケーションの制限事項 2-12
    - 特定のサブネット上の複数アイランド 2-12
    - クラスタ全体への変更の伝播 2-12
    - リモート・オブジェクトのバインド 2-12
    - 複数インスタンス環境における JNDI ルックアップ 2-12

3 Java Message Service（JMS）

概要 3-2
Oracle Application Server JMS 3-2
OracleAS JMS ポートの構成 3-3
OracleAS JMS の Destination オブジェクトの構成 3-3
デフォルトの Destination オブジェクト 3-4
デフォルトのコネクション・ファクトリ 3-5
メッセージの送信のステップ 3-5
メッセージの受信のステップ 3-7
OracleAS JMS ユーティリティ 3-8
OracleAS JMS のファイル・ベースの永続性 3-10
概要 3-10
永続性の有効化 3-11
リカバリ 3-11
異常終了 3-14
OracleAS JMS の事前定義済の例外キュー 3-14
メッセージの期限切れ 3-15
メッセージのページング 3-15
OracleAS JMS の jms.xml 構成ファイルの要素 3-16
例 3-20
OracleAS JMS のシステム・プロパティ 3-21
リソース・プロバイダ 3-23
カスタム・リソース・プロバイダの構成 3-23
Oracle JMS 3-24
OJMS をリソース・プロバイダとして使用する方法 3-24
JMS プロバイダのインストールと構成 3-24
ユーザーの作成と権限の割当て 3-25
JMS Destination オブジェクトの作成 3-25
OJMS リソース・プロバイダの定義 3-26
OJMS リソースへのアクセス 3-29
Oracle Application Server および Oracle データベースと OJMS の併用 3-31
aqapi.jar をコピーするときのエラー 3-31
OJMS の動作保証マトリックス 3-31
リソース参照内の論理名から JNDI 名へのマッピング 3-32
OracleAS JMS に対する JNDI ネーミング 3-33
OJMS に対する JNDI ネーミング 3-34
Java アプリケーション・クライアントに対する JNDI ネーミング・プロパティの設定 3-34
論理名を使用したクライアントからの JMS メッセージ送信 3-35
サード・パーティの JMS プロバイダ 3-36
WebSphere MQ をリソース・プロバイダとして使用する方法 3-36
WebSphere MQ の構成 3-36
SonicMQ をリソース・プロバイダとして使用する方法 3-37
SonicMQ の構成 3-37
SwiftMQ をリソース・プロバイダとして使用する方法 3-38
SwiftMQ の構成 3-38
Message-Driven Bean の使用 3-39
JMS の高可用性とクラスタリング 3-39
OracleAS JMS の高可用性構成 3-40

用語 3-40
OracleAS JMS サーバーの分散先 3-41
Cold Failover Cluster 3-42
OracleAS の専用 JMS サーバー 3-43
OPMN 構成の変更 3-44
OracleAS JMS の構成 3-44
キュー・コネクション・ファクトリ定義の例 3-45
アプリケーションのデプロイ 3-45
高可用性 3-45
OJMS の高可用性構成 3-46
RAC データベースを OJMS と併用する場合のフェイルオーバー 3-46
JMS と RAC ネットワーク・フェイルオーバーの併用 3-46
OJMS と透過的アプリケーション・フェイルオーバー（TAF）の併用 3-47
両方の JMS プロバイダに対するフェイルオーバーのサーバー・サイドのサンプル・コード 3-48
クラスタリングのベスト・プラクティス 3-49

4 データ・ソース

概要 4-2
データ・ソースのタイプ 4-2
エミュレートされたデータ・ソース 4-3
エミュレートされていないデータ・ソース 4-4
ネイティブ・データソース 4-5
データ・ソースの混在 4-6
データ・ソースの定義 4-7
構成ファイル 4-8
データ・ソース XML 構成ファイルの位置の定義 4-8
アプリケーション固有のデータ・ソースの XML 構成ファイル 4-8
データ・ソースの属性 4-9
Oracle Enterprise Manager 10*g* でのデータ・ソースの定義 4-11
XML 構成ファイルでのデータ・ソースの定義 4-12
致命的エラー・コードの拡張 4-12
パスワードの間接化 4-13
Oracle Enterprise Manager 10*g* を使用した間接パスワードの構成 4-13
手動による間接パスワードの構成 4-14
データベース・スキーマとデータ・ソースの関連付け 4-15
database-schema.xml ファイル 4-15
構成例 4-16
データ・ソースの使用方法 4-16
移植可能なデータ・ソース・ルックアップ 4-16
データ・ソースからの接続の取得 4-17
エミュレートされていないデータ・ソースによる接続の取得 4-18
グローバル・トランザクション外からの接続の取得 4-18
グローバル・トランザクション内からの接続の取得 4-18
接続取得のエラー条件 4-19
単一のデータ・ソースに対する 2 つの接続に異なるユーザー名を使用した場合 4-19
OCI JDBC ドライバが正しく構成されていない場合 4-19
2 フェーズ・コミットとデータ・ソースの使用 4-20
Oracle JDBC の拡張機能の使用方法 4-21
接続キャッシング・スキームの使用方法 4-22

OCI JDBC ドライバの使用方法 4-23
Oracle Application Server での Oracle JDBC-OCI ドライバのアップグレードに関する注意事項 4-23
DataDirect JDBC ドライバの使用方法 4-24
DataDirect JDBC ドライバのインストールと設定 4-24
DataDirect のデータ・ソース・エントリの例 4-25
SQLServer 4-25
DB2 4-25
Sybase 4-26
データ・ソースの高可用性のサポート 4-26
Oracle Maximum Availability Architecture（MAA） 4-26
Oracle Data Guard 4-26
Real Application Clusters（RAC） 4-26
ネットワーク・フェイルオーバー 4-27
TAF フェイルオーバー 4-27
OC4J における高可用性（HA）のサポート 4-28
OC4J でのネットワーク・フェイルオーバーの構成 4-28
OC4J での透過的アプリケーション・フェイルオーバー（TAF）の構成 4-29
TAF 記述子の構成（tnsnames.ora） 4-30
接続プーリング 4-31
TAF 例外の確認 4-31
SQL 例外処理 4-32

5 Oracle Remote Method Invocation

RMI/ORMI の概要 5-2
ORMI 拡張機能 5-2
RMI メッセージ・スループットの増大 5-2
スレッド化のサポートの拡張 5-2
同じ場所に配置されているオブジェクトのサポート 5-2
クライアント・サイドの要件 5-3
RMI 用の OC4J の構成 5-3
Oracle Enterprise Manager 10*g* を使用した RMI の構成 5-3
手動による RMI の構成 5-5
server.xml の編集 5-5
rmi.xml の編集 5-5
opmn.xml の編集 5-7
RMI 構成ファイル 5-7
RMI の JNDI プロパティ 5-8
プロバイダ URL のネーミング 5-8
コンテキスト・ファクトリの使用 5-10
ルックアップの例 5-10
OC4J スタンドアロン 5-10
Oracle Application Server 10*g*（9.0.4）より前のリリースの OC4J 5-11
Oracle Application Server 10*g*（9.0.4）以上の OC4J 5-11
HTTP を介した ORMI トンネリングの構成 5-12
OC4J マウント・ポイントの構成 5-13
クラスタ・アプリケーションでの HTTP を介した ORMI の構成 5-14

6 J2EE の相互運用性

RMI/IIOP の概要 6-2
トランスポート 6-2
ネーミング 6-2
セキュリティ 6-2
トランザクション 6-3
クライアント・サイドの要件 6-3
rmic.jar コンパイラ 6-3
相互運用可能トランスポートへの切替え 6-4
スタンドアロン環境での簡易相互運用性 6-4
スタンドアロン環境での拡張相互運用性 6-5
Oracle Application Server 環境での簡易相互運用性 6-6
Oracle Enterprise Manager 10*g* を使用した相互運用性のための構成 6-6
相互運用性のための手動による構成 6-8
Oracle Application Server 環境での拡張相互運用性 6-9
Oracle Enterprise Manager 10*g* を使用した相互運用性のための構成 6-9
相互運用性のための手動による構成 6-10
corbaname の URL 6-11
OPMN の URL 6-12
例外マッピング 6-12
OC4J ホスティング Bean の非 OC4J コンテナからの起動 6-12
相互運用性のための OC4J の構成 6-13
相互運用性 OC4J フラグ 6-13
相互運用性構成ファイル 6-13
相互運用に関する JNDI プロパティ（jndi.properties） 6-14
コンテキスト・ファクトリの使用 6-14
OC4J での IIOP の有効化 6-14

7 Java Transaction API

概要 7-2
トランザクションの境界設定 7-2
リソースの登録 7-2
1 フェーズ・コミット 7-3
単一リソースの登録 7-3
データ・ソースの構成 7-3
データ・ソースの接続の取得 7-3
JNDI ルックアップの実行 7-3
接続の取得 7-4
トランザクション境界設定 7-5
コンテナ管理のトランザクション境界設定 7-5
Bean 管理のトランザクション 7-7
JTA トランザクション 7-7
JDBC トランザクション 7-7
2 フェーズ・コミット 7-7
2 フェーズ・コミット・エンジンの構成 7-8
データベース構成手順 7-8
OC4J 構成手順 7-9

2 フェーズ・コミット・エンジンの制限事項 7-11
タイムアウトの構成 7-11
データベース・インスタンス障害が発生した場合の CMP Bean のリカバリ 7-12
コンテナ管理のトランザクションを使用する CMP Bean の接続のリカバリ 7-12
Bean 管理のトランザクションを使用する CMP Bean の接続のリカバリ 7-12
MDB でのトランザクションの使用 7-13
OC4J JMS を使用した MDB に対するトランザクションの動作 7-13
Oracle JMS を使用した MDB に対するトランザクションの動作 7-13
コンテナ管理のトランザクションを使用する MDB 7-13
Bean 管理のトランザクションを使用する MDB と JMS クライアント 7-14

8 J2EE Connector Architecture（J2CA）

概要 8-2
リソース・アダプタ 8-2
スタンドアロン・リソース・アダプタ 8-2
埋込みリソース・アダプタ 8-2
RAR ファイル構造の例 8-2
ra.xml ディスクリプタ 8-3
アプリケーション・インタフェース 8-3
Quality of Service に関する規約 8-3
リソース・アダプタのデプロイとアンデプロイ 8-4
デプロイメント・ディスクリプタ 8-4
oc4j-ra.xml ディスクリプタ 8-4
oc4j-connectors.xml ディスクリプタ 8-6
スタンドアロン・リソース・アダプタ 8-7
デプロイ 8-7
埋込みリソース・アダプタ 8-9
デプロイ 8-9
関連ファイルの位置 8-9
Quality of Service に関する規約の指定 8-10
接続プーリングの構成 8-10
EIS のサインオンの管理 8-11
コンポーネント管理のサインオン 8-12
コンテナ管理のサインオン 8-13
宣言的なコンテナ管理のサインオン 8-14
プログラム的なコンテナ管理のサインオン 8-15
OC4J 固有の認証クラス 8-15
JAAS 交換可能認証クラス 8-18
プログラム・インタフェースを介してアクセス可能な特殊機能 8-19

9 Java Object Cache

Java Object Cache の概念 9-2
Java Object Cache の基本アーキテクチャ 9-3
分散オブジェクト管理 9-4
Java Object Cache の動作 9-4
キャッシュの編成 9-5
Java Object Cache の機能 9-6
Java Object Cache のオブジェクト・タイプ 9-6

メモリー・オブジェクト 9-7
ディスク・オブジェクト 9-7
StreamAccess オブジェクト 9-7
プール・オブジェクト 9-8
Java Object Cache 環境 9-8
キャッシュ・リージョン 9-8
キャッシュ・サブリージョン 9-9
キャッシュ・グループ 9-9
リージョンとグループのサイズ制御 9-9
キャッシュ・オブジェクトの属性 9-10
オブジェクトのロード前に定義する属性の使用方法 9-11
オブジェクトのロード前およびロード後に定義する属性の使用方法 9-13
Java Object Cache を使用したアプリケーションの開発 9-15
Java Object Cache のインポート 9-15
キャッシュ・リージョンの定義 9-16
キャッシュ・グループの定義 9-16
キャッシュ・サブリージョンの定義 9-17
キャッシュ・オブジェクトの定義と使用 9-17
CacheLoader オブジェクトの実装 9-18
CacheLoader のヘルパー・メソッドの使用方法 9-19
キャッシュ・オブジェクトの無効化 9-20
キャッシュ・オブジェクトの破棄 9-20
複数のオブジェクトのロードおよび無効化 9-21
Java Object Cache の構成 9-22
宣言的なキャッシュ 9-27
宣言的なキャッシュ・ファイルの例 9-28
宣言的なキャッシュ・ファイルの形式 9-29
宣言可能なユーザー定義オブジェクト 9-32
宣言可能な CacheLoader、CacheEventListener および CapacityPolicy 9-33
非 OC4J コンテナでの Java Object Cache の初期化 9-33
容量制御 9-34
キャッシュ・イベント・リスナーの実装 9-36
制限事項およびプログラミングに関する注意点 9-38
ディスク・オブジェクトの操作 9-39
ローカルおよび分散ディスク・キャッシュ・オブジェクト 9-39
ローカル・オブジェクト 9-39
分散オブジェクト 9-39
ディスク・キャッシュへのオブジェクトの追加 9-40
オブジェクトの自動的な追加 9-40
オブジェクトの明示的な追加 9-40
ディスク・キャッシュにのみ存在するオブジェクトの使用方法 9-40
StreamAccess オブジェクトの操作 9-42
StreamAccess オブジェクトの作成 9-42
プール・オブジェクトの操作 9-43
プール・オブジェクトの作成 9-43
プール・オブジェクトの使用方法 9-44
プール・オブジェクトのインスタンス・ファクトリの実装 9-44
プール・オブジェクトのアフィニティ 9-45

ローカル・モードでの実行 9-46
分散モードでの実行 9-46
分散モード用のプロパティの構成 9-46
distribute 構成プロパティの設定 9-46
discoveryAddress 構成プロパティの設定 9-46
分散オブジェクト、リージョン、サブリージョンおよびグループの使用方法 9-47
分散オブジェクトでの REPLY 属性の使用方法 9-47
SYNCHRONIZE および SYNCHRONIZE_DEFAULT の使用方法 9-48
キャッシュされたオブジェクトの整合性レベル 9-50
ローカル・オブジェクトの使用 9-51
応答待機なしの変更の伝播 9-51
変更の伝播および応答の待機 9-51
複数のキャッシュ間にわたる変更のシリアライズ 9-51
OC4J サーブレットでのキャッシュ・オブジェクトの共有 9-52
ユーザー定義のクラス・ローダーの使用 9-52
分散キャッシュの HTTP およびセキュリティ 9-53
HTTP 9-53
SSL 9-53
ファイアウォール 9-54
着信接続要求の制限 9-55
監視およびデバッグ 9-56
キャッシュ構成用の XML Schema 9-59
属性の宣言用の XML Schema 9-60

索引

はじめに

Oracle Application Server 10*g* リリース 2（10.1.2）には、Oracle Application Server Containers for J2EE（OC4J）と呼ばれる J2EE 環境が組み込まれています。このマニュアルでは、OC4J によって提供されるサービスについて説明します。

この章には、次の項目が含まれます。

- 対象読者
- ドキュメントのアクセシビリティについて
- このマニュアルの構成
- 関連ドキュメント
- 表記規則
- サポートおよびサービス

対象読者

このマニュアルは、J2EE アーキテクチャに関する知識があり、Oracle J2EE サービスを理解する必要がある開発者を対象としています。

ドキュメントのアクセシビリティについて

オラクル社は、障害のあるお客様にもオラクル社の製品、サービスおよびサポート・ドキュメントを簡単にご利用いただけることを目標としています。オラクル社のドキュメントには、ユーザーが障害支援技術を使用して情報を利用できる機能が組み込まれています。HTML 形式のドキュメントで用意されており、障害のあるお客様が簡単にアクセスできるようにマークアップされています。標準規格は改善されつつあります。オラクル社はドキュメントをすべてのお客様がご利用できるように、市場をリードする他の技術ベンダーと積極的に連携して技術的な問題に対応しています。オラクル社のアクセシビリティについての詳細情報は、Oracle Accessibility Program の Web サイト `http://www.oracle.com/accessibility/` を参照してください。

ドキュメント内のサンプル・コードのアクセシビリティについて

スクリーン・リーダーは、ドキュメント内のサンプル・コードを正確に読めない場合があります。コード表記規則では閉じ括弧だけを行に記述する必要があります。しかし JAWS は括弧だけの行を読まない場合があります。

外部 Web サイトのドキュメントのアクセシビリティについて

このドキュメントにはオラクル社およびその関連会社が所有または管理しない Web サイトへのリンクが含まれている場合があります。オラクル社およびその関連会社は、それらの Web サイトのアクセシビリティに関しての評価や言及は行っておりません。

このマニュアルの構成

このマニュアルは、次の章で構成されています。

第 1 章「OC4J サービスの概要」

OC4J に含まれるサービス・テクノロジの概要を説明します。

第 2 章「Java Naming and Directory Interface」

Java Naming and Directory Interface（JNDI）を使用してオブジェクトをルックアップする方法について説明します。

第 3 章「Java Message Service（JMS）」

Java Message Service（JMS）と Oracle で提供される 2 つの Oracle JMS（OJMS）プロバイダにリソース・プロバイダをプラグインする方法について説明します。

第 4 章「データ・ソース」

データベース・サーバーへの接続がベンダーに依存しないでカプセル化されたデータ・ソースについて説明します。

第 5 章「Oracle Remote Method Invocation」

独自の Oracle RMI（ORMI）プロトコルを介した Remote Method Invocation（RMI）に関する OC4J サポートについて説明します。

第 6 章「J2EE の相互運用性」

標準の Internet Inter-ORB Protocol（IIOP）プロトコル経由の RMI を使用した EJB 2.0 の相互運用に関する OC4J サポートについて説明します。

第 7 章「Java Transaction API」

Java Transaction API（JTA）の Oracle における実装について説明します。

第 8 章「J2EE Connector Architecture（J2CA）」

J2EE Connector Architecture を OC4J アプリケーションで使用する方法について説明します。

第 9 章「Java Object Cache」

OC4J の Java Object Cache について、そのアーキテクチャとプログラミング機能も含めて説明します。

関連ドキュメント

詳細は、Oracle Java Platform グループから入手可能な、OC4J に関する次のマニュアルを参照してください。

- 『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』

  このマニュアルは、OC4J の概要および一般情報を提供します。サーブレット、JSP ページおよび EJB に関する初歩的な章が含まれ、一般的な構成とデプロイについて説明します。

- 『Oracle Application Server Containers for J2EE JavaServer Pages 開発者ガイド』

  このマニュアルは、OC4J で独自のページを実行する JSP 開発者向けの情報を提供します。JSP 標準の一般的な概要とプログラミングに関する考慮事項も含まれます。また、OC4J 環境を初めて使用する方のために、Oracle の付加価値機能および手順についても説明します。

- 『Oracle Application Server Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』

  このマニュアルは、タグ・ライブラリ、JavaBeans および他の OC4J Java ユーティリティに関する概念的な情報、詳細な構文および使用に関する情報を提供します。

- 『Oracle Application Server Containers for J2EE サーブレット開発者ガイド』

  このマニュアルは、サーブレット開発者向けに、OC4J でのサーブレットおよびサーブレット・コンテナの使用方法に関する情報を提供します。

- 『Oracle Application Server Containers for J2EE Enterprise JavaBeans 開発者ガイド』

  このマニュアルは、OC4J での EJB 実装および EJB コンテナに関する情報を提供します。

Oracle Application Server グループからは、次のマニュアルを入手できます。

- 『Oracle Application Server 管理者ガイド』
- 『Oracle Enterprise Manager 管理者ガイド』
- 『Oracle HTTP Server 管理者ガイド』
- 『Oracle Application Server パフォーマンス・ガイド』
- 『Oracle Application Server グローバリゼーション・サポート・ガイド』
- 『Oracle Application Server Web Cache 管理者ガイド』

JDeveloper グループからは、次のドキュメントが入手できます。

- Oracle JDeveloper オンライン・ヘルプ
- OTN（Oracle Technology Network）上の Oracle JDeveloper マニュアル

  `http://www.oracle.com/technology/products/jdev/index.html`

OC4J の詳細情報は、次の OTN リソースを参照してください。

- OC4J に関する OTN Web サイト

  `http://www.oracle.com/technology/tech/java/oc4j/index.html`

- OTN の OC4J ディスカッション・フォーラムは、次の URL よりアクセスできます。

  `http://otn.oracle.com/forums/forum.jsp?id=486963`

リリース・ノート、インストール関連ドキュメント、ホワイト・ペーパーまたはその他の関連ドキュメントは、OTN-J（Oracle Technology Network Japan）から、無償でダウンロードできます。OTN-J を使用するには、オンラインでの登録が必要です。登録は、次の Web サイトから無償で行えます。

`http://otn.oracle.co.jp/membership/`

すでに OTN-J のユーザー名およびパスワードを取得している場合は、次の URL で OTN-J Web サイトのドキュメントのセクションに直接接続できます。

`http://otn.oracle.co.jp/document/`

表記規則

このマニュアルでは次の表記規則を使用します。

規則	意味
. . .	文またはコマンドの中に、水平の省略記号がある場合は、例に直接関連しないコードの一部が省略されていることを示します。
太字	太字は、本文中で定義されている用語および用語集に記載されている用語を示します。
固定幅フォント	本文中の固定幅フォントは、実行可能ファイル、ファイル名、ディレクトリ名、Java クラス名、Java メソッド名、変数名、その他のプログラム要素（JSP タグや属性または XML 要素や属性など）、またはデータベースの SQL コマンドや要素（スキーマ名、表名または列名など）の項目を示します。
固定幅フォントのイタリック	固定幅フォントのイタリックは、プレースホルダまたは変数を示します。
< >	山カッコは、ユーザーが入力する名前を示します。
[]	大カッコは、カッコ内の項目を任意に選択することを表します。
|	縦線は、複数の選択項目の区切りに使用します。項目のうちの 1 つを入力します。縦線は、入力しないでください。

サポートおよびサービス

次の各項に、各サービスに接続するための URL を記載します。

オラクル社カスタマ・サポート・センター

オラクル製品サポートの購入方法、およびオラクル社カスタマ・サポート・センターへの連絡方法の詳細は、次の URL を参照してください。

http://www.oracle.co.jp/support/

製品マニュアル

製品のマニュアルは、次の URL にあります。

http://otn.oracle.co.jp/document/

研修およびトレーニング

研修に関する情報とスケジュールは、次の URL で入手できます。

http://www.oracle.co.jp/education/

その他の情報

オラクル製品やサービスに関するその他の情報については、次の URL から参照してください。

http://www.oracle.co.jp
http://otn.oracle.co.jp

注意： ドキュメント内に記載されている URL や参照ドキュメントには、Oracle Corporation が提供する英語の情報も含まれています。日本語版の情報については、前述の URL を参照してください。

1

OC4J サービスの概要

Oracle Application Server Containers for J2EE（OC4J）は、次のテクノロジをサポートします。このマニュアルには各テクノロジに関する章が含まれています。

- Java Naming and Directory Interface（JNDI）
- Java Message Service（JMS）
- Remote Method Invocation（RMI）
- データ・ソース
- Java Transaction API（JTA）
- J2EE Connector Architecture（J2CA）
- Java Object Cache

この章では、各テクノロジについて簡単に説明します。

注意： これらのテクノロジの他に、OC4J は JavaMail API、JavaBeans Activation Framework（JAF）および Java API for XML Processing（JAXP）をサポートします。これらのテクノロジの詳細は、Sun 社の J2EE ドキュメントを参照してください。

Java Naming and Directory Interface（JNDI）

OC4J により実装される Java Naming and Directory Interface（JNDI）サービスは、Java アプリケーションにネーミングおよびディレクトリ機能を提供します。JNDI は、特定のネーミングまたはディレクトリ・サービス実装とは関係なく定義されます。このため、JNDI を使用すると、Java アプリケーションは単一の API を使用して異なる（場合によっては複数の）ネーミングおよびディレクトリ・サービスにアクセスできます。この共通 API の背後にネーミングとディレクトリの異なるサービス・プロバイダ・インタフェース（SPI）をプラグインすると、様々なネーミング・サービスを処理できます。

詳細は、第 2 章「Java Naming and Directory Interface」を参照してください。

Java Message Service（JMS）

Java Message Service（JMS）は、Java プログラムに、エンタープライズ・メッセージ製品にアクセスする共通の方法を提供します。JMS は、JMS クライアントがエンタープライズ・メッセージ製品の機能にアクセスする方法を定義するインタフェースと関連セマンティックの集合です。

詳細は、第 3 章「Java Message Service（JMS）」を参照してください。

Remote Method Invocation（RMI）

Remote Method Invocation（RMI）は、リモート・プロシージャ・コール・パラダイムの Java 実装の 1 つです。この実装では、分散アプリケーションは、プロシージャ・コールを起動し、戻り値を解析して通信を行います。

OC4J は、Oracle Remote Method Invocation（ORMI）プロトコルを介した RMI と、Internet Inter-ORB Protocol（IIOP）を介した RMI の両方をサポートします。

OC4J は、デフォルトで RMI/ORMI を使用します。RMI/ORMI は、RMI/IIOP によるメリットに加えて、HTTP を介して RMI/ORMI を起動する「RMI トンネリング」と呼ばれる技術などの機能も提供します。

RMI/ORMI の詳細は、第 5 章「Oracle Remote Method Invocation」を参照してください。

バージョン 2.0 の Enterprise JavaBeans（EJB）の仕様では、Internet Inter-ORB Protocol（IIOP）を介して RMI を使用して、EJB ベースのアプリケーションが、異なるコンテナ間で別のアプリケーションを簡単に起動できるようにします。既存の EJB を、コード行を変更せずに、Bean のプロパティを編集して再デプロイするのみで相互運用可能にできます。J2EE は RMI を使用して、異なるコンテナで実行されている EJB 間の相互運用性を提供します。

相互運用性（RMI/IIOP）の詳細は、第 6 章「J2EE の相互運用性」を参照してください。

データ・ソース

データ・ソース（`javax.sql.DataSource` インタフェースを実装するオブジェクトのインスタンス化）を使用すると、データベース・サーバーへの接続を取得できます。

詳細は、第 4 章「データ・ソース」を参照してください。

Java Transaction API（JTA）

EJB では、トランザクションの管理に Java Transaction API（JTA）1.0.1 が使用されます。これらのトランザクションには、1 フェーズ・コミットと 2 フェーズ・コミットが関連します。

詳細は、第 7 章「Java Transaction API」を参照してください。

J2EE Connector Architecture（J2CA）

J2EE Connector Architecture（J2CA）は、J2EE プラットフォームを異種エンタープライズ情報システム（EIS）に接続するための標準アーキテクチャを定義します。EIS の例には、ERP、メインフレーム・トランザクション処理、データベース・システムおよび Java プログラミング言語で記述されていないレガシー・アプリケーションなどがあります。

詳細は、第 8 章「J2EE Connector Architecture（J2CA）」を参照してください。

Java Object Cache

Java Object Cache（以前の OCS4J）は、プロセス内、プロセス間およびローカル・ディスク上で Java オブジェクトを管理する Java クラスの集合です。Java Object Cache の主な目的は、取得や作成にコストがかかるオブジェクトのローカル・コピーを管理することによってサーバーのパフォーマンスを大幅に向上させる、強力で柔軟性のある使いやすいサービスを提供することです。キャッシュできるオブジェクトの型やオブジェクトの元のソースに制限はありません。キャッシュ内の各オブジェクトの管理は容易にカスタマイズできます。各オブジェクトには一連の属性が関連付けられており、キャッシュへのロード方法、格納場所（メモリーまたはディスク、あるいはその両方）、無効化の方法（時間ベースまたは明示的なリクエスト）、無効化されたときの通知先などが制御されます。オブジェクトは、グループ単位または個別に無効化できます。詳細は、第 9 章「Java Object Cache」を参照してください。

2 Java Naming and Directory Interface

この章では、Oracle Application Server Containers for J2EE（OC4J）のアプリケーションによって実装される Java Naming and Directory Interface（JNDI）サービスについて説明します。この章には、次の項目が含まれます。

- 概要
- JNDI コンテキストの構成
- JNDI 環境
- OC4J での初期コンテキストの作成

概要

JNDI は J2EE 仕様の一部であり、Java アプリケーションにネーミングおよびディレクトリ機能を提供します。JNDI は、特定のネーミングまたはディレクトリ・サービス実装とは関係なく定義されるため、JNDI を使用すると、Java アプリケーションは単一の API を使用して異なるネーミングおよびディレクトリ・サービスにアクセスできます。この共通 API の背後にネーミングとディレクトリの異なるサービス・プロバイダ・インタフェース（SPI）をプラグインすると、様々なネーミング・サービスを処理できます。

この章を読むには、JNDI と JNDI API に関する基本的な知識が必要です。チュートリアルや API ドキュメントなど、JNDI に関する基本情報は、Sun 社の次の Web サイトを参照してください。

`http://java.sun.com/products/jndi/index.html`

JNDI を実装する JAR ファイル `jndi.jar` は、OC4J で使用可能です。アプリケーションでは、他のライブラリや JAR ファイルを用意せずに JNDI API を利用できます。J2EE 互換のアプリケーションでは、JNDI を使用してネーミング・コンテキストを取得します。このネーミング・コンテキストによって、アプリケーションは、データ・ソースなどのオブジェクト、Java Message Service（JMS）サービス、ローカル Enterprise JavaBeans（EJB）とリモート EJB およびその他多数の J2EE オブジェクトやサービスを検出して取得できます。

注意： JNDI ネームスペースへのアクセスの制御方法の詳細は、『Oracle Application Server セキュリティ・ガイド』を参照してください。

初期コンテキスト

初期コンテキストの概念は、JNDI の基本です。J2EE アプリケーションで最も頻繁に行われる JNDI 操作は次の 2 つです。

- 新規 `InitialContext` オブジェクトの作成（`javax.naming` パッケージ内）
- J2EE またはその他のリソースをルックアップするための `InitialContext` の使用

OC4J は、起動時に各アプリケーションの構成 XML ファイルのリソース参照を読み取ることによって、各アプリケーションの JNDI 初期コンテキストを構成します。

注意： 初期構成後の各アプリケーションの JNDI ツリーは、完全にメモリーベースです。コンテキストに対して実行時に行われる追加は維持されません。OC4J を再起動すると、アプリケーション・コードでの `Context.bind` API コールなど、アプリケーション・コンポーネントにより JNDI ネームスペースに対して追加作成されたバインドは使用できなくなります。ただし、各種の XML ファイルを介して宣言的にバインドされた場合は、起動時に再構成されます。

例

次の例は、一般的な Web または EJB アプリケーションで、サーバー・サイドで使用される 2 行の Java コードです。

```
Context ctx = new InitialContext();
myEJBHome myhome =
        (HelloHome) ctx.lookup("java:comp/env/ejb/myEJB");
```

最初の文は、デフォルト環境を使用して新規の初期コンテキスト・オブジェクトを作成します。2 番目の文は、アプリケーションの JNDI ツリーで EJB ホーム・インタフェース参照をルックアップします。この場合、myEJB は、web.xml（または orion-web.xml）構成ファイルで <ejb-ref> タグに宣言されている Session Bean 名です。次に例を示します。

```
<ejb-ref>
  <ejb-ref-name>ejb/myEJB</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>myEjb.HelloHome</home>
  <remote>myEjb.HelloRemote</remote>
</ejb-ref>
```

この章では主に、JNDI を使用するための初期コンテキストの設定方法、および OC4J による JNDI ルックアップの実行方法について説明します。他の JNDI クラスおよびメソッドの詳細は、次の Javadoc を参照してください。

http://java.sun.com/products/jndi/1.2/javadoc/index.html

JNDI コンテキストの構成

OC4J は起動時に、サーバーにデプロイされた各アプリケーション用の JNDI コンテキストを構成します。OC4J サーバーには、少なくとも 1 つのアプリケーション（グローバル・アプリケーション）があります。このアプリケーションは、サーバー・インスタンス内の各アプリケーションに対するデフォルトの親です。ユーザー・アプリケーションは、グローバル・アプリケーションからプロパティを継承します。ユーザー・アプリケーションでは、グローバル・アプリケーションで定義されたプロパティ値のオーバーライド、プロパティに対する新しい値の定義、および必要に応じた新しいプロパティの定義が可能です。

OC4J サーバーとそれに含まれているアプリケーションの構成方法の詳細は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』を参照してください。

注意： OC4J に EJB をデプロイする際、Bean クラスをロードしてそのメソッドを検出し、EJB ラッパーを生成できるようにします。静的ブロックのコードは、クラスがロードされているときに実行されるため、JNDI 環境のコンテキストはまだ設定されていません。実行時でさえ、Bean は存在しないステージにあります。ライフ・サイクルのこのステージでは、JNDI 環境のコンテキストは未定義であるため、コンテキストに基づいて Bean プロバイダを使用することができません。

この問題を解決するには、ejbCreate() メソッドまたは setSessionContext() メソッドのいずれかで、Bean の構成時にコンテキストを設定し、キャッシュします。

OC4J が JNDI 初期コンテキストの構成に使用する環境は、次の 3 つの場所にあります。

- システム・プロパティ値。OC4J サーバーまたはアプリケーション・コンテナによって設定されます。
- jndi.properties ファイル。（application-client.jar の一部として）アプリケーションの EAR ファイルに含まれます。
- java.util.Hashtable インスタンスで明示的に指定された環境。JNDI 初期コンテキスト・コンストラクタに渡されます（このコンストラクタのコード例は、2-7 ページの「アプリケーション・クライアントからのオブジェクトへのアクセス」を参照してください）。

JNDI 環境

JNDI の InitialContext には、次の 2 つのコンストラクタがあります。

```
InitialContext()
InitialContext(Hashtable env)
```

1 番目のコンストラクタでは、デフォルトのコンテキスト環境を使用して Context オブジェクトが作成されます。OC4J のサーバー・サイド・アプリケーションでこのコンストラクタを使用すると、OC4J は、そのアプリケーション用のデフォルト環境を使用して、サーバーの起動時に初期コンテキストを作成します。このコンストラクタは通常、JSP、EJB またはサーブレットなど、サーバー・サイドで実行するコードで使用されます。

2 番目のコンストラクタでは、環境パラメータが使用されます。この形式の InitialContext コンストラクタは通常、JNDI 環境を指定する必要があるクライアント・アプリケーションで使用されます。このコンストラクタの env パラメータは java.util.Hashtable で、JNDI に必要なプロパティが含まれます。表 2-1 に、javax.naming.Context インタフェースで定義されるこれらのプロパティを示します。

表 2-1 InitialContext のプロパティ

プロパティ	意味
INITIAL_CONTEXT_FACTORY	java.naming.factory.initial プロパティの値。新規の初期コンテキスト・オブジェクトの作成時にどの初期コンテキスト・ファクトリを使用するかを指定します。
PROVIDER_URL	java.naming.provider.url プロパティの値。サーバー上のオブジェクトをルックアップするためにアプリケーション・クライアント・コードで使用される URL を指定します。別のアプリケーションのオブジェクトを検索するために、RMIInitialContextFactory および ApplicationClientInitialContextFactory でも使用されます。詳細は、2-6 ページの表 2-2「JNDI 関連の環境プロパティ」を参照してください。
SECURITY_PRINCIPAL	java.naming.security.principal プロパティの値。ユーザー名を指定します。アプリケーション・クライアント・コードでクライアントを認証するために必要です。サーバー・サイドのコードでは、認証はすでに実行されているため必要ありません。
SECURITY_CREDENTIAL	java.naming.security.credential プロパティの値。パスワードを指定します。アプリケーション・クライアント・コードでクライアントを認証するために必要です。サーバー・サイドのコードでは、認証はすでに実行されているため必要ありません。

これらのプロパティを設定し、新規 JNDI 初期コンテキストを取得するコード例は、2-7 ページの「アプリケーション・クライアントからのオブジェクトへのアクセス」を参照してください。

OC4J での初期コンテキストの作成

J2EE 1.3 仕様の 9.1 項では、アプリケーション・クライアントを次のように定義しています。

「... 独自の Java 仮想マシンで実行される第 1 層のクライアント・プログラム。Java テクノロジ・ベースのアプリケーション・モデルに従い、それぞれのメイン・メソッドによって起動し、仮想マシンが終了するまで実行されます。ただし、他の J2EE アプリケーション・コンポーネントと同様に、アプリケーション・クライアントはコンテナに依存してシステム・サービスを提供します。アプリケーション・クライアントのコンテナは、他の J2EE コンテナに比べてきわめて軽量であり、(この仕様に)記述されているセキュリティ・サービスとデプロイメント・サービスのみを提供します。」

次の項では、JNDI 初期コンテキストの使用方法について説明します。

- J2EE アプリケーション・クライアントからの使用
- J2EE アプリケーション・コンポーネントからの使用

J2EE アプリケーション・クライアントからの使用

J2EE サーバー・アプリケーションで使用可能なリソースをアプリケーション・クライアントでルックアップする必要がある場合、クライアントは初期コンテキストの構成に com.evermind.server パッケージの ApplicationClientInitialContextFactory を使用します。

注意: アプリケーションが J2EE クライアントの場合(すなわち、application-client.xml ファイルが存在する場合)は、クライアント・アプリケーションで使用されるプロトコル(ORMI または IIOP)に関係なく、常に ApplicationClientInitialContextFactory を使用する必要があります。プロトコル自体は、JNDI プロパティ java.naming.provider.url で指定します。詳細は、2-6 ページの表 2-2「JNDI 関連の環境プロパティ」を参照してください。

OC4J サーバーの外部で実行される Java コードで構成され、バンドルされた J2EE アプリケーションの一部でもあるアプリケーション・クライアントについて考えてみます。たとえば、ワークステーションで実行されるクライアント・コードで、EJB などのサーバー・オブジェクトに接続して、一部のアプリケーション・タスクを実行するとします。この場合、JNDI にアクセス可能な環境では、プロパティ java.naming.factory.initial の値に ApplicationClientInitialContextFactory を指定する必要があります。この値は、クライアント・コードで指定できます。または、EAR ファイルに含まれる application-client.jar ファイルの一部である jndi.properties ファイルで指定することもできます。

アプリケーションの一部であるリモート・オブジェクトにアクセスするために、ApplicationClientInitialContextFactory は、application-client.jar ファイル内の META-INF/application-client.xml ファイルおよび META-INF/orion-application-client.xml ファイルを読み取ります。

クライアントは、ApplicationClientInitialContextFactory を使用して JNDI 初期コンテキストを構成すると、java:comp/env 機能と RMIInitialContextFactory を使用して、ローカル・オブジェクト(そのアプリケーション自身またはその親アプリケーションに含まれるオブジェクト)をルックアップできます。ORMI または IIOP を使用すると、これらのオブジェクトでメソッドを起動できます。オブジェクトとリソースをアプリケーションの JNDI コンテキストにバインドするには、デプロイメント・ディスクリプタで定義する必要があることに注意してください。

環境プロパティ

ORMI プロトコルが使用されている場合、ApplicationClientInitialContextFactory は表 2-2 に示すプロパティを環境から読み取ります。

表 2-2 JNDI 関連の環境プロパティ

プロパティ	意味
dedicated.rmicontext	廃止された dedicated.connection 設定にかわるプロパティです。同じプロセスの複数のクライアントが InitialContext を取得する場合、OC4J はキャッシュされたコンテキストを戻します。そのため、各クライアントはプロセスに割り当てられている同じ InitialContext を受け取ります。サーバーのロード・バランシングを発生させるサーバー・ルックアップは、クライアントが固有の InitialContext を取得する場合にのみ発生します。dedicated.rmicontext=true を設定すると、各クライアントは共有コンテキストのかわりに固有の InitialContext を受け取ります。各クライアントに固有の InitialContext があれば、クライアントのロード・バランシングが可能です。 dedicated.rmicontext プロパティはデフォルトで false に設定されます。
java.naming.provider.url	ローカル・オブジェクトまたはリモート・オブジェクトの検索時に使用する URL を指定します。書式は次のいずれかです。 [http: | https:]ormi://*hostname*/*appname* または *corbaname*:*hostname*:*port*。*corbaname* の URL の詳細は、6-11 ページの「corbaname の URL」を参照してください。 カンマ区切りのリストで複数のホスト（フェイスオーバー用）を指定できます。
java.naming.factory.url.pkgs	一部のプラットフォームでは、JDK のバージョンによって、システム・プロパティ java.naming.factory.url.pkgs が自動設定され、com.sun.java.* が組み込まれることがあります。このプロパティをチェックし、com.sun.java.* が存在している場合は削除してください。
http.tunnel.path	RMIHttpTunnelServlet の代替パスを指定します。デフォルト・パスは、ターゲット・サイトの Web アプリケーションにバインドされている /servlet/rmi です。詳細は、5-12 ページの「HTTP を介した ORMI トンネリングの構成」を参照してください。
context.SECURITY_PRINCIPAL	ユーザー名を指定します。このプロパティは、クライアント・サイドのコードでクライアントを認証するために必要です。サーバー・サイドのコードでは、認証はすでに実行されているため必要ありません。このプロパティ名は、java.naming.security.principal としても定義されます。
context.SECURITY_CREDENTIAL	パスワードを指定します。このプロパティは、クライアント・サイドのコードでクライアントを認証するために必要です。サーバー・サイドのコードでは、認証はすでに実行されているため必要ありません。このプロパティ名は、java.naming.security.credentials としても定義されます。

アプリケーション・クライアントからのオブジェクトへのアクセス

この項では、同じロケーションの OC4J インスタンス内で実行されている EJB にアクセスするための、アプリケーション・クライアントの構成例を示します。

最初に、EJB が OC4J にデプロイされます。この例は、EJB のデプロイメント・ディスクリプタの抜粋です。

EJB は `EmployeeBean` という名前でデプロイされます。この名前は `ejb-jar.xml` で次のように定義されます。

```
<ejb-jar>
  <display-name>bmpapp</display-name>
  <description>
     An EJB app containing only one Bean Managed Persistence Entity Bean
  </description>
  <enterprise-beans>
     <entity>
        <description>no description</description>
        <display-name>EmployeeBean</display-name>
        <ejb-name>EmployeeBean</ejb-name>
        <home>bmpapp.EmployeeHome</home>
        <remote>bmpapp.Employee</remote>
        <ejb-class>bmpapp.EmployeeBean</ejb-class>
        <persistence-type>Bean</persistence-type>
        ...
     </entity>
  </enterprise-beans>
..
</ejb-jar>
```

EJB EmployeeBean は、`orion-ejb-jar.xml` 内の JNDI ロケーション `java:comp/env/bmpapp/EmployeeBean` にバインドされます。

orion-ejb-jar.xml ファイル

```
<orion-ejb-jar>
   <enterprise-beans>
       <entity-deployment name="EmployeeBean"
           location="bmpapp/EmployeeBean" table="EMP">
                        ...
       </entity-deployment>
                        ...
   </enterprise-beans>
           ...
</orion-ejb-jar>
```

アプリケーション・クライアント・プログラムは、`EmployeeBean` EJB を使用し、それを `EmployeeBean` として参照します。このアプリケーション・クライアント・プログラムからの抜粋を次に示します。

```
public static void main (String args[])
{
 ...
 Context context = new InitialContext();
 /**
  * Look up the EmployeeHome object. The reference is retrieved from the
  * application-local context (java:comp/env). The variable is
  * specified in the assembly descriptor (META-INF/application-client.xml).
  */
 Object homeObject =
     context.lookup("java:comp/env/EmployeeBean");
 // Narrow the reference to an EmployeeHome.
 EmployeeHome home =
     (EmployeeHome) PortableRemoteObject.narrow(homeObject,
                                                EmployeeHome.class);
```

```
    // Create a new record and narrow the reference.
    Employee rec =
        (Employee) PortableRemoteObject.narrow(home.create(empNo,
                                                           empName,
                                                           salary),
                                               Employee.class);
    // call method on the EJB
    rec.doSomething();
    ...
}
```

次の行でコンテキストを作成するときに、ハッシュ表を渡していないことに注意してください。

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

これは、コンテキストが jndi.properties ファイルから読み取られる値を使用して作成されるためです。この例では、次の内容が含まれています。

```
java.naming.factory.initial=com.evermind.server.ApplicationClientInitialContextFactory
java.naming.provider.url=ormi://localhost/bmpapp
java.naming.security.principal=SCOTT
java.naming.security.credentials=TIGER
```

または、jndi.properties ファイルを提供するかわりに、InitialContext のコンストラクタにハッシュ表を渡すこともできます。その場合、コードは次のようになります。

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
"com.evermind.server.ApplicationClientInitialContextFactory");
env.put("java.naming.factory.initial",
"com.evermind.server.ApplicationClientInitialContextFactory");
env.put("java.naming.provider.url","ormi://localhost/bmpapp");
env.put("java.naming.security.principal","SCOTT");
env.put("java.naming.security.credentials","TIGER");
Context initial = new InitialContext(env);
```

アプリケーション・クライアントのコードは EmployeeBean EJB を参照するため、この EJB を application-client.xml ファイルの <ejb-ref> 要素で宣言する必要があります。

```
<application-client>
   <display-name>EmployeeBean</display-name>
   <ejb-ref>
       <ejb-ref-name>EmployeeBean</ejb-ref-name>
       <ejb-ref-type>Entity</ejb-ref-type>
       <home>bmpapp.EmployeeHome</home>
       <remote>bmpapp.Employee</remote>
   </ejb-ref>
</application-client>
```

EmployeeBean EJB が orion-ejb-jar.xml ファイル内で構成されたとおり JNDI ロケーション java:comp/env/bmpapp/EmployeeBean にバインドされることに注意してください。アプリケーション・クライアント・プログラムで使用する EJB 名は、EJB が実際にバインドされている JNDI ロケーションにマップする必要があります。このマッピングは、次のように orion-application-client.xml ファイルで指定します。

```
orion-application-client.xml file:
<orion-application-client>
   <ejb-ref-mapping name="EmployeeBean" location="bmpapp/EmployeeBean" />
</orion-application-client>
```

J2EE アプリケーション・コンポーネントからの使用

OC4J で初期コンテキスト・ファクトリを使用して、J2EE アプリケーション・コンポーネントから次のオブジェクトにアクセスできます。

- 同じアプリケーション内のオブジェクト
- 同じアプリケーションにないオブジェクト

同じアプリケーション内のオブジェクト

J2EE アプリケーション・コンポーネントは、サーブレット、JSP ページおよび EJB から同じアプリケーション内のオブジェクトにアクセスするために使用できます。

サーバーで実行されているコードは、アプリケーションの一部として定義されます。このため、OC4J は、JNDI が使用するプロパティのデフォルトを設定できます。JNDI の InitialContext オブジェクトの構成時には、アプリケーション・コードでプロパティ値を提供する必要はありません。

このコンテキスト・ファクトリが使用されている場合、ApplicationContext は現行のアプリケーションに固有であるため、そのアプリケーションの web.xml、orion-web.xml または ejb-jar.xml などのファイルで指定されたすべての参照が使用可能です。つまり、java:comp/env を使用するルックアップは、アプリケーションが指定したすべてのリソースに対して機能します。このファクトリを使用するルックアップは、同じ Java 仮想マシン（JVM）でローカルに実行されます。

アプリケーションでリモート参照（同じ JVM 内の別の J2EE アプリケーションのリソースや J2EE アプリケーションの外部のリソースなど）をルックアップする必要がある場合は、RMIInitialContextFactory または IIOPInitialContextFactory を使用する必要があります。2-10 ページの「同じアプリケーションにないオブジェクト」を参照してください。

例 具体的な例として、データベース上で JDBC 操作を行うためにデータ・ソースを取得する必要があるサーブレットについて考えます。

データ・ソースの位置は、data-sources.xml で次のように指定します。

```
<data-source
    class="oracle.jdbc.pool.OracleConnectionCacheImpl"
    location="jdbc/pool/OracleCache"
    username="hr"
    password="hr"
    url="jdbc:oracle:thin:@//<hostname>:<TTC port>/<DB ID>"
/>
```

データ・ソースの位置の詳細は、第 4 章「データ・ソース」を参照してください。

サーブレットの web.xml ファイルでは、次のリソースが定義されます。

```
<resource-ref>
   <description>
      A data source for the database in which
      the EmployeeService enterprise bean will
      record a log of all transactions.
   </description>
   <res-ref-name>jdbc/EmployeeAppDB</res-ref-name>
   <res-type>javax.sql.DataSource</res-type>
   <res-auth>Container</res-auth>
   <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

対応する orion-web.xml のマッピングは次のとおりです。

```
<resource-ref-mapping name="jdbc/EmployeeAppDB" location="jdbc/pool/OracleCache" />
```

name 値は、web.xml の <res-ref-name> 要素で指定されている値と同じです。location 値は、data-sources.xml の <data-source> 要素の location または ejb-location です。

この場合、サーブレットの次のコードによって、データ・ソース・オブジェクトへの正しい参照が戻されます。

```
...
try {
  InitialContext ic = new InitialContext();
  ds = (DataSource) ic.lookup("java:comp/env/jdbc/EmployeeAppDB");
  ...
}
catch (NamingException ne) {
  throw new ServletException(ne);
}
...
```

初期コンテキスト・ファクトリを指定する必要はありません。これは、アプリケーションの起動時に、システム・プロパティ `java.naming.factory.initial` のデフォルト値として、OC4J が `ApplicationInitialContextFactory` を設定するためです。

この場合、同じアプリケーション内または `java:comp/` の下に含まれているオブジェクトのルックアップに URL が必要ないため、プロバイダ URL を指定する必要はありません。

注意： 一部のプラットフォームでは、JDK のバージョンによって、システム・プロパティ `java.naming.factory.url.pkgs` が自動設定され、`com.sun.java.*` が組み込まれることがあります。このプロパティをチェックし、`com.sun.java.*` が存在している場合は削除してください。

アプリケーションは、java:comp/env 機能を使用すると、固有のネームスペース内のみでなく、宣言されている親アプリケーションのネームスペース内、あるいはグローバル・アプリケーション（特定の親アプリケーションが宣言されなかった場合のデフォルトの親）内で指定されたリソースもルックアップできます。

同じアプリケーションにないオブジェクト

同じアプリケーションにないオブジェクトには、次のコンテキスト・ファクトリのいずれかを使用してアクセスします。

- RMIInitialContextFactory
- IIOPInitialContextFactory

RMIInitialContextFactory ほとんどのアプリケーションについては、デフォルトのサーバー・サイド `ApplicationInitialContextFactory` または `ApplicationClientInitialContextFactory` のいずれかを使用できます。ただし、次のような場合には、追加のコンテキスト・ファクトリを使用する必要があります。

- クライアント・アプリケーションに `application-client.xml` ファイルがない場合は、`ApplicationClientInitialContextFactory` プロパティではなく、`RMIInitialContextFactory` プロパティを使用する必要があります。
- クライアント・アプリケーションが JNDI ネームスペースに（特定のアプリケーションのコンテキスト内ではなく）リモートでアクセスする場合は、`RMIInitialContextFactory` を使用する必要があります。

`RMIInitialContextFactory` は、次の環境プロパティを使用します。これらのプロパティは、`ApplicationClientInitialContextFactory` でも使用されます。2-6 ページの表 2-2 を参照してください。

- `java.naming.provider.url`
- `http.tunnel.path`
- `context.SECURITY_PRINCIPAL`
- `Context.SECURITY_CREDENTIALS`

ここでは、異なるマシン上の別の OC4J インスタンスで実行されている EJB にアクセスするサーブレットの例を示します。この例の EJB は、2-7 ページの「アプリケーション・クライアントからのオブジェクトへのアクセス」で使用した EmployeeBean です。

次のコードは、サーブレット・コードから抜粋したものです。

```
Hashtable env = new Hashtable();
env.put("java.naming.factory.initial",
"com.evermind.server.rmi.RMIInitialContextFactory");
env.put("java.naming.provider.url","ormi://remotehost/bmpapp");
env.put("java.naming.security.principal","SCOTT");
env.put("java.naming.security.credentials","TIGER");
Context context = new InitialContext(env);
Object homeObject =
context.lookup("java:comp/env/EmployeeBean");
```

アプリケーション・クライアントの場合と同様に、このサーブレットの web.xml ファイルで <ejb-ref> 要素を宣言する必要があります。

```
<ejb-ref>
<ejb-ref-name>EmployeeBean</ejb-ref-name>
<ejb-ref-type>Entity</ejb-ref-type>
<home>bmpapp.EmployeeHome</home>
<remote>bmpapp.Employee</remote>
</ejb-ref>
```

また、orion-web.xml には、次の例に示すように、論理名 EmployeeBean から、EJB がバインドされている実際の JNDI 名へのマッピングが指定されている必要があります。

```
<ejb-ref-mapping name="EmployeeBean" location="bmpapp/EmployeeBean" />
```

IIOPInitialContextFactory このファクトリの使用条件は RMIInitialContextFactory の場合と同じですが、プロトコルには ORMI ではなく IIOP を使用します。

注意： このファクトリは EJB のルックアップ専用です。

JNDI 状態レプリケーション

JNDI 状態レプリケーションにより、OC4J クラスタの 1 つの OC4J インスタンスでコンテキストに対して行われた変更が、他のすべての OC4J インスタンスのネームスペースに確実にレプリケートされます。

JNDI 状態レプリケーションが有効化されている場合は、1 つのサーバー上でシリアライズ可能な値をアプリケーション・コンテキストに（リモート・クライアント、EJB またはサーブレットを使用して）バインドし、それを別のサーバー上で読み取ることができます。また、この方法でサブコンテキストを作成したり破棄することもできます。

この項には、次の項目が含まれます。

- JNDI 状態レプリケーションの有効化
- JNDI 状態レプリケーションの制限事項

JNDI 状態レプリケーションの有効化

EJB クラスタリングが有効化されると、JNDI 状態レプリケーションが有効化されます。

JNDI 状態レプリケーションを利用するには、特に EJB クラスタリングを必要としない場合（起動クラスやデータ・ソースの検索に JNDI を使用する場合など）でも、EJB クラスタリングを有効化する必要があります。

EJB クラスタリングの有効化の詳細は、『Oracle Application Server Containers for J2EE Enterprise JavaBeans 開発者ガイド』の「EJB のクラスタリング」を参照してください。

OC4J のクラスタリングの概要は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』の「OC4J のクラスタリング」を参照してください。

JNDI 状態レプリケーションの制限事項

JNDI 状態レプリケーションに依存する場合は、次の制限事項を考慮してください。

- 特定のサブネット上の複数アイランド
- クラスタ全体への変更の伝播
- リモート・オブジェクトのバインド

特定のサブネット上の複数アイランド

状態レプリケーションのパフォーマンスを改善するために OC4J プロセスをグループ（アイランド）単位で編成することができますが、EJB アプリケーションでは、OC4J インスタンスのすべての OC4J プロセス間の状態がレプリケートされ、アイランドのサブグループ化は使用されません。この詳細は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』の「OC4J のクラスタリング」を参照してください。

そのため、JNDI クラスタリングは単一アイランドのサブネットに制限されません。1 つのサブネットに複数のアイランドが存在する場合は、そのサブネットのアイランドすべてでグローバルな JNDI コンテキストが共有されます。

クラスタ全体への変更の伝播

リバインド（名前の変更）とアンバインドは伝播しません。ローカルに適用されますが、クラスタ全体では共有されません。

シリアライズ不可の値へのバインドも、クラスタ全体に伝播しません。

リモート・オブジェクトのバインド

アプリケーション・コンテキスト内でリモート・オブジェクト（通常はホームまたは EJB オブジェクト）をバインドすると、その JNDI オブジェクトはクラスタ全体で共有されますが、バインドされている最初のサーバーに障害が発生すると、シングル・ポイント障害が発生します。

複数インスタンス環境における JNDI ルックアップ

複数の OC4J インスタンスが存在する環境では、JNDI ルックアップに次の追加情報が必要です。

- opmn 接頭辞
- ons ホスト名
- ons ポート番号

これらの情報は EJB および JMS にも適用されます。

3

Java Message Service（JMS）

この章には、次の項目が含まれます。

- 概要
- Oracle Application Server JMS
- リソース・プロバイダ
- Oracle JMS
- リソース参照内の論理名から JNDI 名へのマッピング
- サード・パーティの JMS プロバイダ
- Message-Driven Bean の使用
- JMS の高可用性とクラスタリング

この章で使用している JMS の例は、次の URL の OTN Web サイトの OC4J サンプル・コード・ページからダウンロードしてください。

```
http://www.oracle.com/technology/tech/java/oc4j/demos/index.html
```

概要

Java クライアントおよび Java 中間層サービスでは、エンタープライズ・メッセージ・システムを使用できる必要があります。Java Message Service（JMS）は、Java プログラムに、これらのシステムにアクセスする共通の方法を提供します。JMS は、アプリケーション・コンポーネント間でデータを渡すための標準のメッセージ API で、異機種種間環境およびレガシー環境でのビジネス統合を可能にします。

JMS には、次の 2 つのプログラミング・モデルがあります。

- Point-to-Point: メッセージは JMS キューを使用してシングル・コンシューマに送信されます。
- パブリッシュ / サブスクライブ : メッセージは登録されているすべてのリスナーに JMS トピックを介して配布されます。

JMS のキューおよびトピックは JNDI 環境にバインドされ、J2EE アプリケーションで使用可能になります。

次のように、統合要件と Quality of Service（QOS）要件に応じて複数の JMS プロバイダの中から選択できます。

- Oracle Application Server JMS: OC4J とともにインストールされ、メモリー内で実行される JMS プロバイダ。
- Oracle JMS（OJMS）: Oracle データベースの機能であり、Streams Advanced Queuing メッセージ・システムをベースとする JMS プロバイダ。
- サード・パーティの JMS プロバイダ : サード・パーティの JMS プロバイダ WebSphere MQ、SonicMQ、SwiftMQ と統合できます。

Oracle Application Server JMS

OracleAS JMS は、次の機能を提供する Java Message Service です。

- JMS 1.0.2b 仕様への準拠
- メモリー内またはファイル・ベースのメッセージの永続性における選択肢の提供
- 配信できないメッセージ用の例外キューの提供

この項には、次の項目が含まれます。

- OracleAS JMS ポートの構成
- OracleAS JMS の Destination オブジェクトの構成
- メッセージの送信のステップ
- OracleAS JMS ユーティリティ
- OracleAS JMS のファイル・ベースの永続性
- 異常終了
- OracleAS JMS の事前定義済の例外キュー
- メッセージのページング
- OracleAS JMS の jms.xml 構成ファイルの要素
- OracleAS JMS のシステム・プロパティ

OracleAS JMS ポートの構成

Oracle Enterprise Manager 10*g* を使用して OracleAS JMS のポート範囲を構成できます。デフォルトの範囲は 3201 〜 3300 です。

OC4J ホームページで「管理」ページを選択し、「インスタンス・プロパティ」列、「サーバー・プロパティ」の順に選択します。「複数仮想マシン構成」セクションにスクロールします。

OracleAS JMS の Destination オブジェクトの構成

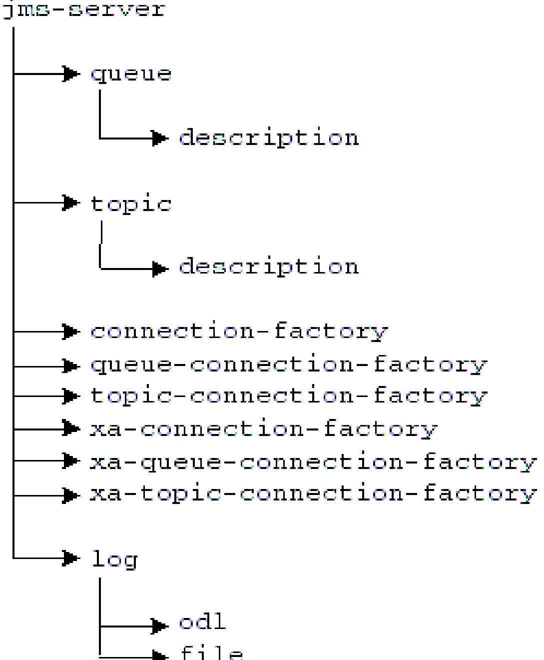
OracleAS JMS `Destination` オブジェクトは、`jms.xml` ファイル内で構成します。OracleAS JMS `Destination` オブジェクトは、キューまたはトピックです。OracleAS JMS はすでに OC4J とともにインストールされているため、構成する必要があるのはアプリケーションで使用するキュー、トピックおよびそれぞれのコネクション・ファクトリのみです。

- Oracle Enterprise Manager 10*g* での構成 : Oracle Enterprise Manager 10*g* を介して `jms.xml` ファイルを直接編集するには、「管理」ページの「インスタンス・プロパティ」列で「拡張プロパティ」を選択します。このセクションで `jms.xml` を選択し、XML ファイルそのものを変更します。
- スタンドアロン OC4J での構成 : `J2EE_HOME/config/jms.xml` にあるデフォルトの `jms.xml` ファイルを構成できます。必要な場合は、このファイルの名前と位置を変更できます。JMS 構成ファイルの名前と位置を変更するには、OC4J サーバー構成ファイル（`J2EE_HOME/config/server.xml`）の新しい名前と位置を指定します。`server.xml` ファイルでは、`<jms-config>` 要素を使用して JMS 構成ファイルの名前と位置を指定します。

> **注意 :** OracleAS JMS に対する（`jms.xml` の変更による）構成変更を有効にするには、OC4J を再起動（停止して起動）する必要があります。

図 3-1 に、`jms.xml` ファイルの各要素の構成順序を示します。`jms.xml` ファイルの全要素とその属性の詳細は、3-16 ページの「OracleAS JMS の jms.xml 構成ファイルの要素」を参照してください。

図 3-1 構成要素の階層



jms.xml ファイルでは、使用するトピックとキューを定義します。Destination オブジェクト（キューまたはトピック）ごとに、jms.xml ファイルで名前（位置とも呼ばれます）とコネクション・ファクトリを指定する必要があります。次の jms.xml ファイルの構成例では、Oc4jjmsDemo デモで使用されるキューが定義されています。

キューは次のように定義されます。

- キューの名前（位置）は jms/demoQueue です。
- キュー・コネクション・ファクトリは、jms/QueueConnectionFactory として定義されます。

トピックは次のように定義されます。

- トピックの名前（位置）は jms/demoTopic です。
- トピック・コネクション・ファクトリは、jms/TopicConnectionFactory として定義されます。

```
<?xml version="1.0" ?>
<!DOCTYPE jms-server PUBLIC "OracleAS JMS server" "http://xmlns.oracle.com/ias/dtds
/jms-server.dtd">

<jms-server port="9127">
   <queue location="jms/demoQueue"> </queue>
   <queue-connection-factory location="jms/QueueConnectionFactory">
   </queue-connection-factory>

   <topic location="jms/demoTopic"> </topic>
   <topic-connection-factory location="jms/TopicConnectionFactory">
   </topic-connection-factory>

    <!-- path to the log-file where JMS-events/errors are stored -->
    <log>
        <file path="../log/jms.log" />
    </log>
</jms-server>
```

注意： これらの値はすべてデフォルトであるため、構成する必要はありません。ただし、例では、独自の Destination オブジェクトおよびコネクション・ファクトリの構成方法を理解できるように、キュー、トピックおよびそれぞれのコネクション・ファクトリの構成を示してあります。

jms.xml ファイルの各要素については、3-16 ページの「OracleAS JMS の jms.xml 構成ファイルの要素」を参照してください。

デフォルトの Destination オブジェクト

OracleAS JMS では、2 つのデフォルトの Destination オブジェクトが次のように作成されます。

- デフォルト・キューは、jms/demoQueue として定義されます。
- デフォルト・トピックは、jms/demoTopic として定義されます。

この 2 つの Destination オブジェクトは、jms.xml 構成ファイルに追加しなくてもコード内で使用できます。

これらのオブジェクトには、次のデフォルト・コネクション・ファクトリが自動的に関連付けられます。

- jms/QueueConnectionFactory
- jms/TopicConnectionFactory

デフォルトのコネクション・ファクトリ

OracleAS JMS では、XA/ 非 XA および各種 JMS ドメインに対してデフォルトのコネクション・ファクトリが 6 つ作成されます。新規コネクション・ファクトリを定義するのではなく、これらのコネクション・ファクトリをコード内で使用できます。`jms.xml` 構成ファイルに追加する必要はありません。`jms.xml` ファイルで新規コネクション・ファクトリを定義するのは、`connection-factory` 要素の 1 つ以上のオプション属性にデフォルト以外の値を指定する必要がある場合のみです。

デフォルトのコネクション・ファクトリは次のとおりです。

- `jms/ConnectionFactory`
- `jms/QueueConnectionFactory`
- `jms/TopicConnectionFactory`
- `jms/XAConnectionFactory`
- `jms/XAQueueConnectionFactory`
- `jms/XATopicConnectionFactory`

そのため、デフォルトのコネクション・ファクトリのみを使用する場合は、必要なトピックとキューを `jms.xml` ファイル内で定義するだけですみます。次の例では、`jms/demoQueue` および `jms/demoTopic` を定義しています。この 2 つのオブジェクトはどちらも、それぞれのデフォルト・コネクション・ファクトリを使用します。

```
<?xml version="1.0" ?>
<!DOCTYPE jms-server PUBLIC "OracleAS JMS server" "http://xmlns.oracle.com/ias/dtds
/jms-server.dtd">

<jms-server port="9127">
   <queue location="jms/demoQueue"> </queue>
   <topic location="jms/demoTopic"> </topic>
    <!-- path to the log-file where JMS-events/errors are stored -->
    <log>
        <file path="../log/jms.log" />
    </log>
</jms-server>
```

OracleAS JMS により内部的にデフォルトのコネクション・ファクトリ・オブジェクトが作成され、JMS コネクションが作成される OC4J サーバー内のデフォルト名にバインドされます。

ただし、デフォルトのコネクション・ファクトリを `jms.xml` ファイル内で構成し、特定の属性を持つように再定義することもできます。

メッセージの送信のステップ

JMS クライアントは、次のステップに従って JMS メッセージを送信します。

1. JNDI ルックアップを使用して、構成済の JMS `Destination` オブジェクト（キューまたはトピック）とそのコネクション・ファクトリを取得します。
2. コネクション・ファクトリから接続を作成します。
3. メッセージを受信する場合は、接続が開始されます。
4. 接続を使用してセッションを作成します。
5. 取得した JMS `Destination` を指定し、キューの場合はセンダー、トピックの場合はパブリッシャを作成します。
6. メッセージを作成します。
7. キュー・センダーまたはトピック・パブリッシャを使用してメッセージを送信します。
8. キュー・セッションをクローズします。
9. JMS `Destination` タイプに応じて接続をクローズします。

例 3-1 に JMS メッセージの送信ステップを示します。完全な例は、次の URL の OTN Web サイトの OC4J サンプル・コード・ページからこの章で使用している JMS の例をダウンロードしてください。

```
http://www.oracle.com/technology/tech/java/oc4j/demos/index.html
```

注意： 例 3-1 では、簡潔にするためにほとんどのエラー・コードを削除してあります。エラー処理を確認するには、OTN の Web サイトで入手可能なサンプル・コードを参照してください。

例 3-1 メッセージをキューに送信する OracleAS JMS クライアント

OracleAS JMS の JNDI ルックアップでは、`jms.xml` ファイル内で OracleAS JMS `Destination` およびコネクション・ファクトリの先頭に `java:comp/env/` 接頭辞を付けて定義する必要があります。

注意： または、JNDI ルックアップで論理名を使用することもできます。詳細は、3-32 ページの「リソース参照内の論理名から JNDI 名へのマッピング」を参照してください。OracleAS JMS クライアントと OJMS クライアントの相違点は、JNDI ルックアップで提供する名前のみです。クライアントを JMS プロバイダに依存させない場合は、実装に論理名を使用し、OC4J 固有のデプロイメント・ディスクリプタのみを変更します。

次の例で示す `dosend` メソッドは、メッセージを送信するためのキューを設定します。この例では、キュー・センダーの作成後に複数のメッセージを送信します。

```
public static void dosend(int nmsgs)
{
   // 1a. Retrieve the queue connection factory.
    QueueConnectionFactory qcf = (QueueConnectionFactory)
            ctx.lookup("java:comp/env/jms/QueueConnectionFactory");
   // 1b. Retrieve the queue.
    Queue q = (Queue) ctx.lookup("java:comp/env/jms/demoQueue");

   // 2. Create the JMS connection.
    QueueConnection qc  = qcf.createQueueConnection();
   // 3. Start the queue connection.
    qc.start();
   // 4. Create the JMS session over the JMS connection.
    QueueSession    qs  = qc.createQueueSession(false,
                                    Session.AUTO_ACKNOWLEDGE);
    //5. Create a sender on the JMS session to send messages.
    QueueSender     snd = qs.createSender(q);

    // Send out messages...
    for (int i = 0; i < nmsgs; ++i)
    {
     //6. Create the message using the createMessage method of the
     // JMS session.
     Message msg = qs.createMessage();
     //7. Send the message out over the sender (snd) using the
     // send method.
     snd.send(msg);
     System.out.println("msg:" + " id=" + msg.getJMSMessageID());
    }
```

```
   //8 & 9 Close the sender, the JMS session and the JMS connection.
    snd.close();
    qs.close();
    qc.close();

}
```

メッセージの受信のステップ

JMS クライアントは、次のステップに従って JMS メッセージを受信します。

1. JNDI ルックアップを使用して、構成済の JMS `Destination` オブジェクト（キューまたはトピック）とそのコネクション・ファクトリを取得します。
2. コネクション・ファクトリから接続を作成します。
3. メッセージを受信する場合は、接続が開始されます。
4. 接続を使用してセッションを作成します。
5. 取得した JMS `Destination` を指定し、キューの場合はレシーバ、トピックの場合はサブスクライバを作成します。
6. キュー・レシーバまたはトピック・サブスクライバを使用してメッセージを受信します。
7. キュー・セッションをクローズします。
8. JMS `Destination` タイプに応じて接続をクローズします。

例 3-2 に JMS メッセージの受信ステップを示します。完全な例は、次の URL の OTN Web サイトの OC4J サンプル・コード・ページからこの章で使用している JMS の例をダウンロードしてください。

http://www.oracle.com/technology/tech/java/oc4j/demos/index.html

> **注意：** 例 3-2 では、簡潔にするためにほとんどのエラー・コードを削除してあります。エラー処理を確認するには、OTN の Web サイトで入手可能なサンプル・コードを参照してください。

例 3-2 キューからメッセージを受信する OracleAS JMS クライアント

次の例で示す `dorcv` メソッドは、受信するメッセージを取り出すキューを設定します。キュー・レシーバの作成後に、ループしてキューからすべてのメッセージを受信し、予想されたメッセージの数と比較します。

```
public static void dorcv(int nmsgs)
{
   Context ctx = new InitialContext();

   // 1a. Retrieve the queue connection factory.
    QueueConnectionFactory qcf = (QueueConnectionFactory)
            ctx.lookup("java:comp/env/jms/QueueConnectionFactory");
   // 1b. Retrieve the queue.
    Queue q = (Queue) ctx.lookup("java:comp/env/jms/demoQueue");

   // 2. Create the JMS connection.
    QueueConnection qc  = qcf.createQueueConnection();
   // 3. Start the queue connection.
    qc.start();
   // 4. Create the JMS session over the JMS connection.
    QueueSession    qs  = qc.createQueueSession(false,
                                    Session.AUTO_ACKNOWLEDGE);
   // 5. Create a receiver, as we are receiving off of the queue.
    QueueReceiver   rcv = qs.createReceiver(q);
```

```
      // 6. Receive the messages.
       int count = 0;
       while (true)
       {
         Message msg = rcv.receiveNoWait();
         System.out.println("msg:" + " id=" + msg.getJMSMessageID());
         ++count;
       }

       if (nmsgs != count)
       {
           System.out.println("expected: " + nmsgs + " found: " + count);
       }

      // 7 & 8 Close the receiver, the JMS session and the JMS connection.
       rcv.close();
       qs.close();
       qc.close();
   }
```

OracleAS JMS ユーティリティ

OC4J JMS には、デバッグおよび情報へのアクセス用に、OC4J 固有のコマンドライン・ユーティリティ `com.evermind.server.jms.JMSUtils` が付属しています。

パス *J2EE_HOME*/`oc4j.jar` を CLASSPATH 変数に指定する必要があります。その後、次のように `JMSUtils` を実行します。

```
java com.evermind.server.jms.JMSUtils [gen_options] [command]
                     [command_options]
```

OracleAS JMS サーバーが稼働している必要があります。`JMSUtils` を使用できるのは管理者のみです。ユーザーは、セキュリティの「ユーザー・マネージャ」で管理者ロール内で定義します。セキュリティ・ロール内でユーザーを定義する方法は、『Oracle Application Server Containers for J2EE セキュリティ・ガイド』を参照してください。

`JMSUtils` の汎用オプションを使用すると、OracleAS JMS サーバーに接続できます。表 3-1 にこれらのオプションを示します。

表 3-1 JMSUtils のオプション

オプション	説明
`-host <hostname>`	OracleAS JMS サーバーがインストールされている（リモート）ホスト。クライアントが OracleAS JMS サーバーと同じノードに存在する場合、このオプションは不要です。
`-port <port>`	OracleAS JMS サーバーへのアクセスに使用される（リモート）ポート。デフォルトの JMS ポート番号は 9127 です。
`-username <username>`	JMS コネクションを作成するために OracleAS JMS サーバーへのアクセスに使用するユーザー名。このユーザーは、管理ロールを使用して「ユーザー・マネージャ」セキュリティ構成で定義します。
`-password <password>`	JMS コネクションを作成するために OracleAS JMS サーバーへのアクセスに使用するパスワード。このパスワードは、管理ロールを使用して「ユーザー・マネージャ」セキュリティ構成で定義します。
`-clientID <ID>`	この ID をすべての JMS コネクションに使用します。このオプションは、トピックの永続サブスクリプションを識別する場合にのみ必要です。

各コマンドでは、実行するアクションを記述します。表 3-2 を参照してください。これらのコマンドの一部には、必要なアクションの詳細を記述する固有のオプション（command_options）があります。

構文の使用方法を表示するには、JMSUtils コマンドを引数なしで発行します。使用可能なコマンド・セット、引数のオプションおよび各コマンドの動作の詳細情報を表示するには、次のコマンドを発行します。

```
java com.evermind.server.jms.JMSUtils help
```

表 3-2 OC4J JMS ユーティリティ

ユーティリティ	コマンドの説明
help	すべてのユーティリティ・コマンドの詳細ヘルプを出力します。
check [<other-selector>]	-selector コマンド・オプションで識別された JMS メッセージ・セレクタの妥当性をチェックします。オプションで、指定された 2 つのセレクタが等価として処理されるかどうかをチェックします（永続サブスクリプションを再アクティブ化する場合に役立ちます）。この場合、2 番目のセレクタはオプションの <other-selector> で識別します。
knobs	使用可能なすべてのシステム・プロパティ（表 3-6 を参照）と OC4J JMS サーバー上での現行の設定を表示します。
stats	OC4J JMS サーバーに関して使用可能な DMS 統計をすべて表示します（JMS 以外の統計も含まれます）。（DMS の詳細は、『Oracle Application Server パフォーマンス・ガイド』を参照してください。）
destinations	OC4J JMS に認識される永続 Destination オブジェクトの全リストを出力します。
durables	OC4J JMS に認識される永続サブスクリプションの全リストを出力します。
subscribe <topic>	<topic> で新規の永続サブスクリプションを作成します。名前、メッセージ・セレクタ、ローカルかどうか、および永続サブスクリプション・クライアント ID を指定します。これにより、既存のアクティブでない永続サブスクリプションが置き換えられます。名前は、-name コマンド・オプションを使用して指定します。メッセージ・セレクタは、-selector コマンド・オプションを使用して指定します。永続サブスクリプションがローカルかどうかは、-noLocal コマンド・オプションを使用して指定します。クライアント ID は、-clientID 汎用オプションを使用して定義します。
unsubscribe	既存のアクティブでない永続サブスクリプションを削除します。永続サブスクリプションは、名前（-name コマンド・オプション）とクライアント ID（-clientID 汎用オプション）で識別します。
browse <destination>	指定された宛先（jms.xml で定義されているキューまたはトピックの永続サブスクリプション）のメッセージを参照します。
drain <destination>	指定された宛先（キューまたはトピックの永続サブスクリプション）のメッセージをデキューします。
copy <from-destination> <to-destination>	ある宛先（キューまたはトピックの永続サブスクリプション）から別の宛先にメッセージをコピーします。ソースとシンクの宛先が同一の場合、コマンドは実行されず、かわりにエラーが生成されます。
move <from-destination> <to-destination>	ある宛先（キューまたはトピックの永続サブスクリプション）から別の宛先にメッセージを移動します。ソースとシンクの宛先が同一の場合、コマンドは実行されず、かわりにエラーが生成されます。

表 3-3 にコマンド・オプションを示します。

表 3-3 JMSUtils コマンド・オプション

コマンド・オプション	説明
`-selector <selector>`	指定された JMS メッセージ・セレクタを使用して、キュー・レシーバと永続サブスクライバを作成します。
`-noLocal [true|false]`	`true` に設定すると、サブスクライバには同じ接続でパブリッシュされたメッセージは表示されません。永続サブスクライバの作成時に使用します。デフォルト値は `false` です。
`-name <name>`	トピックを操作する永続サブスクリプション名を定義します。このオプションは、トピックを読み取るコマンドには必須で、キューを読み取るコマンドの場合は無視されます。
`-silent`	処理中はメッセージを出力しません。処理済メッセージの合計数が保持され、標準エラーに出力されます。
-count <count>	現行の操作中に、指定された数のメッセージのみを処理します。count が負の値または 0（ゼロ）の場合は、選択したメッセージがすべて処理されます。

`JMSUtils` を使用して例外キューを参照する例を次に示します。

```
java com.evermind.server.jms.JMSUtils -username admin -password welcome
        browse jms/Oc4jJmsExceptionQueue
```

OracleAS JMS のファイル・ベースの永続性

OC4J JMS は、JMS `Destination` オブジェクト（キューおよびトピック）に関してファイル・ベースの永続性をサポートしています。次の各項では、ファイル・ベースの永続性について詳しく説明します。

- 概要
- 永続性の有効化
- リカバリ

概要

永続性が有効化されている場合、OC4J は次の操作を自動的に実行します。

- 永続性ファイルが存在しない場合、OC4J は自動的にファイルを作成して該当するデータで初期化します。
- 永続性ファイルが存在していても空の場合は、該当するデータで初期化されます。

> **注意：** OC4J サーバーがアクティブである場合は、永続性ファイルを移動、削除、変更しないでください。このような操作を実行すると、データが破損してメッセージが消失する可能性があります。OC4J がアクティブでない場合は、永続性ファイルを削除すると、そのファイルに関連付けられている宛先からすべてのメッセージと永続サブスクリプションが削除されます。OC4J が再起動すると、JMS サーバーによりファイルが通常の方法で再び初期化されます。

永続性が有効化されていても、ファイルに残るのは特定のメッセージのみです。ファイルに残るメッセージについては、次の条件がすべて満たされている必要があります。

- `jms.xml` ファイルの `persistence-file` 属性で、`Destination` オブジェクトが永続と定義されていること。

- メッセージに永続配信モードが設定されていること。これはデフォルト・モードです。非永続配信モード（DeliveryMode.NON_PERSISTENT で定義）を指定して定義されている永続的な宛先に送信されたメッセージは、永続的ではありません。

どのメッセージが永続的であるかのセマンティクスの詳細は、JMS 仕様を参照してください。

永続性の有効化

Destination オブジェクトについてファイル・ベースの永続性を有効化するには、jms.xml ファイルで persistent-file 属性を指定します。スタンドアロン OC4J 内の JMS オブジェクトの場合は、この属性を指定するのみで永続性が有効化されます。次の XML 構成に、persistence-file 属性でファイル名を pers として定義する方法を示します。persistence-file 属性のパスと命名規則については、3-16 ページの「OracleAS JMS の jms.xml 構成ファイルの要素」を参照してください。

```
<queue name="foo" location="jms/persist" persistence-file="pers">
</queue>
```

persistence-file 属性のパスは、ファイルの絶対パス、または application.xml に定義されている persistence ディレクトリに対する相対パスです。

JMS サーバーは、永続性ファイル用のディレクトリを一切作成しません。そのため、永続性ファイルを jms.xml に定義する場合、次のように既存の絶対ディレクトリ内を指定する必要があります。

```
persistence-file="/this/dir/exists/PersistenceFile"
```

または、次のようにファイル名のみを指定します。

```
persistence-file="PersistenceFile"
```

後者の場合、デフォルトでは、永続性ファイルは $J2EE_HOME/persistence（スタンドアロン・インスタンスの場合）または $J2EE_HOME/persistence/<island_name>（*i*AS 環境の場合）に作成されます。

Oracle Application Server では、複数の OC4J インスタンスが同じファイル・ディレクトリに書き込む場合があり、その際に同じ永続性ファイル名を指定する場合もあります。この属性を設定するとファイル・ベースの永続性が有効化されますが、永続性ファイルが他の OC4J インスタンスにより上書きされる可能性もあります。

リカバリ

OC4J JMS のファイル・ベースの永続性により、メッセージのリカバリ可能で永続的なストレージが提供されます。各 OC4J JMS Destination（キューまたはトピック）は、相対パス名または絶対パス名に関連付けることができます。このパスは、Destination オブジェクトに送信されたメッセージを格納するファイルを指します。ファイルは、ファイル・システムのどこにでも（必ずしも *J2EE_HOME* ディレクトリ内でなくても）置くことができます。複数の永続性ファイルを同じディレクトリに置いたり、リモート・ネットワーク・ファイル・システムに置いたり、ローカル・ファイル・システムの一部にすることができます。

次の項では、OracleAS JMS の永続性リカバリの様々な側面について説明します。

- リカバリ可能性の有効範囲
- 永続性ファイルの管理
- JMS クライアントへのエラーのレポート
- OracleAS JMS のリカバリ・ステップ

リカバリ可能性の有効範囲 OC4J JMS は、可能性のある障害タイプすべてからリカバリできるわけではありません。次のいずれかの障害が発生した場合、OC4J JMS では永続性ファイルのリカバリ可能性は保証されません。

- メディアの破損 : 永続性ファイルを保持しているディスク・システムが異常終了したか破損した場合。
- 外部の破損 : 永続性ファイルが削除、編集、変更されるか、または（ソフトウェアにより）他の方法で破壊された場合。永続性ファイルに書き込むのは OC4J JMS サーバーのみにする必要があります。
- エラーを戻さない失敗または破損 : JDK の I/O メソッドがエラーを戻さずに失敗した場合、あるいは読取りまたは書込み中のデータが破損し、エラーが戻されない場合。
- `java.io.FileDescriptor.sync()` の失敗 : `sync()` コールが、指定のディスクリプタに関連付けられているファイル・バッファすべてを、基礎となるファイル・システムに正常かつ完全にフラッシュしない場合。

永続性ファイルの管理 OC4J JMS サーバーの稼働中は、現在使用中の永続性ファイルのコピー、削除または名前変更を実行しないでください。OC4J JMS で使用中の永続性ファイルに対して前述のアクションを実行すると、リカバリ不可能なエラーが発生します。

ただし、OC4J JMS サーバーで永続性ファイルが使用されていない場合は、これらのファイルに対して次の管理およびメンテナンス操作を実行できます。

- 削除 : 永続性ファイルを削除すると、すべてのメッセージが削除され、トピックの場合はすべての永続サブスクリプションが削除されます。OC4J JMS の起動時に、新規（空）の永続性ファイルが初期化されます。
- コピー : 既存の永続性ファイルをアーカイブまたはバックアップのためにコピーできます。既存の永続性ファイルが破損した場合は、（OC4J JMS `Destination` 名とファイルの関連付けが維持されている場合は）適切なパス名が指す以前のバージョンを使用して、JMS `Destination` の以前の内容に戻すことができます。

永続性ファイルの連結、分割、並替え、マージはできません。このような操作を実行すると、永続性ファイル内のデータが破損し、リカバリ不可能になります。

OC4J JMS では、内部の構成およびトランザクション状態管理に、ユーザー指定の永続性ファイルとロック・ファイルに加えて特殊ファイル `jms.state` が使用されます。OC4J JMS は、通常の操作中にこのファイルとその内容をクリーン・アップします。アーカイブを目的とする場合でも、このファイルは削除、移動、コピーまたは変更しないでください。`jms.state` ファイルを操作すると、メッセージとトランザクションが消失する可能性があります。

注意 : `jms.state` ファイルの位置は、OC4J の操作モード（スタンドアロンまたは Oracle Application Server）に応じて次のように異なります。

- スタンドアロン : *J2EE_HOME*/`persistence` ディレクトリ
- Oracle Application Server: *J2EE_HOME*/`persistence/<island_name>` ディレクトリ

`persistence` ディレクトリの位置は、`application.xml` ファイルで定義されます。

JMS クライアントへのエラーのレポート JMS クライアントがメッセージをエンキューまたはデキューしたり、トランザクションをコミットまたはロールバックするときの操作順序は、次のとおりです。

- クライアントがファンクション・コールを実行します。
  - 事前永続性操作
    - 永続性が発生します。
  - 事後永続性操作
- クライアントのファンクション・コールが戻ります。

事前永続性フェーズまたは永続性フェーズで障害が発生すると、クライアントは `JMSException` または他のなんらかのタイプのエラーを受け取りますが、永続性ファイルは変更されません。

事後永続性フェーズで障害が発生すると、クライアントは `JMSException` または他のなんらかのタイプのエラーを受け取ります。ただし、永続性ファイルはそのまま更新され、OC4J JMS は操作が成功した場合と同様にリカバリします。

OracleAS JMS のリカバリ・ステップ ロック・ファイルは、複数の OC4J プロセスが同じ永続性ファイルに書き込まないようにします。複数の OC4J JVM が同じ `persistence-file` 位置にある同じファイルを指すように構成されていると、各 JVM により相互のデータが上書きされ、永続的な JMS メッセージが破損または消失する可能性があります。この種の共有違反を防止するために、OracleAS JMS は各永続性ファイルをロック・ファイルに関連付けます。そのため、各永続性ファイル（`/path/to/persistenceFile` など）は、`/path/to/persistenceFile.lock` というロック・ファイルに関連付けられます。永続性ファイルの詳細は、3-11 ページの「永続性の有効化」を参照してください。

OC4J には、ロック・ファイルを作成および削除するための適切なパーミッションが必要です。OC4J が正常終了すると、ロック・ファイルが自動的にクリーン・アップされます。ただし、OC4J が異常終了すると、ロック・ファイルはファイル・システムに残ります。OC4J は残っているロック・ファイルを共有違反と区別できないため、ユーザーは異常終了後に OC4J を再起動する前に、すべてのロック・ファイルを手動で削除する必要があります。OracleAS JMS では、既存のロック・ファイルが検出されると、関連する永続的な JMS 宛先は作成されません。

OC4J JMS では、ロック・ファイルが自動的に削除されることはありません。OC4J JMS で特定の永続性ファイルを使用するには、ロック・ファイルを手動で削除する必要があります。以降の説明は、問題のロック・ファイルがすべて削除されていることを前提としていることに注意してください。

注意 : この手動による介入が必要になるのは、異常停止した場合のみです。3-14 ページの「異常終了」を参照してください。

OC4J JMS は、そこで構成されている永続性ファイルすべてに対して、異常終了時にリカバリ操作を実行します。つまり、OC4J が異常終了した場合に、ユーザーが JMS サーバー構成を変更して OC4J を再起動すると、JMS サーバーは引き続き元の構成にあった永続性ファイルをすべてリカバリしようとします。その後このリカバリに成功すると、指定の新規構成の使用に移行します。

古い構成のリカバリに失敗すると、OC4J JMS サーバーは起動しません。サーバーは停止されるか、またはリカバリに成功するまで繰り返し再起動されます。

OC4J JMS は、`jms.state` ファイル内の現行の永続性構成をキャッシュします。このファイルは、トランザクション状態の維持にも使用されます。現行の構成のリカバリをすべて省略する場合は、`jms.state` ファイルを削除し、すべてのロック・ファイルを削除し、場合によっては OC4J JMS サーバー構成を変更して、サーバーを白紙の状態で起動できます（この方法はお薦めしません）。`jms.state` ファイルが見つからない場合は、OC4J JMS サーバーにより新規作成されます。

なんらかの理由で jms.state ファイル自体が破損した場合は、そのファイルを削除する必要があります。これに付随して、保留トランザクションはすべて消失します。つまり、コミットされたが、そのトランザクションに関係する個々の Destination オブジェクトすべてがコミットを実行していないトランザクションは消失します。

異常終了時にメッセージ・アクティビティが進行中だった場合、OC4J JMS は永続性ファイルのリカバリを試行します。データの（前述したタイプの）破損は、破損データのクリーン・アウトにより処理されますが、これによりメッセージとトランザクションが消失する可能性があります。

永続性ファイルのヘッダーが破損すると、この種の破損ファイルはユーザー構成エラーと区別できないことが多いため、OC4J JMS はファイルをリカバリできなくなる場合があります。oc4j.jms.forceRecovery 管理プロパティ（表 3-6 を参照）は、メッセージが消失したりユーザー構成エラーがマスクされる可能性があっても無効なデータをすべて除去してリカバリを進行するように、OC4J JMS サーバーに指示します。

異常終了

OC4J が正常終了すると、ロック・ファイルが自動的にクリーン・アップされます。ただし、OC4J が異常終了すると（kill -9 コマンドなど）、ロック・ファイルはファイル・システムに残ります。OC4J は残っているロック・ファイルを共有違反と区別できないため、異常終了後は OC4J を再起動する前にロック・ファイルすべてを手動で削除する必要があります。OC4J JMS は既存のロック・ファイルを検出すると、関連する永続的な JMS Destination オブジェクトを作成しません。

ロック・ファイルのデフォルト位置は persistence ディレクトリ（*J2EE_HOME*/persistence）です（persistence ディレクトリは、application.xml ファイルで定義されます）。他の位置は、Destination オブジェクトの persistence-file 属性内に設定できます。

OracleAS JMS の事前定義済の例外キュー

OC4J JMS には、JMS 仕様の拡張機能として、配信できないメッセージを処理するための事前定義済の例外キューが付属しています。これは 1 つの永続的なグローバル例外キューであり、すべての Destination オブジェクト内で配信できないメッセージが格納されます。この例外キューの名前（jms/Oc4jJmsExceptionQueue）、JNDI ロケーション（jms/Oc4jJmsExceptionQueue）および永続性ファイル（Oc4jJmsExceptionQueue）はすべて固定です。

注意： Oc4jJmsExceptionQueue 永続性ファイルの位置は、OC4J の操作モード（スタンドアロンまたは Oracle Application Server）に応じて次のように異なります。

- スタンドアロン：J2EE_HOME/persistence ディレクトリ
- Oracle Application Server: J2EE_HOME/persistence/<island_name> ディレクトリ

persistence ディレクトリの位置は、application.xml ファイルで定義されます。

例外キューは OC4J JMS とそのクライアントに常時使用可能であり、jms.xml 構成ファイルでは明示的に定義しません。明示的に定義しようとするとエラーが発生します。例外キューの名前、JNDI ロケーションおよび persistence ディレクトリへのパス名は、それぞれのネームスペースの予約語とみなす必要があり、これらの名前で他のエンティティを定義しようとするとエラーが発生します。

メッセージは、期限切れまたはリスナー・エラーが原因で配信できなくなることがあります。次の項では、期限切れで配信できないメッセージに対して実行される操作について説明します。

メッセージの期限切れ

デフォルトでは、永続的な Destination に送信されたメッセージが期限切れになると、OC4J JMS はそのメッセージを例外キューに移動します。期限切れになったメッセージの JMSXState は値 3（EXPIRED）に設定されますが、メッセージ・ヘッダー、プロパティおよび本文は特に変更されません。このメッセージは ObjectMessage にラップされ（この章の他の項で説明したように該当するプロパティ名と値のコピーが実行され）、ラッピング・メッセージが例外キューに送られます。

例外キューにメッセージを送る動作を指定するには、oc4j.jms.saveAllExpired プロパティ（表 3-6 を参照）を使用します。

ラップしている ObjectMessage の DeliveryMode は、元のメッセージと同じです。

デフォルトでは、非永続的または一時的な Destination オブジェクトで期限切れになっているメッセージは、例外キューに移動しません。これらの Destination オブジェクトに送信されたメッセージは永続的ではなく、期限切れバージョンはありません。

送信先の Destination オブジェクトが永続的であるか、非永続的つまり一時的であるかに関係なく、期限切れのメッセージがすべて OC4J JMS 例外キューに送信されるように指定できます。そのためには、OC4J サーバーの起動時に oc4j.jms.saveAllExpired 管理プロパティ（表 3-6 を参照）を true に設定します。この場合、期限切れのメッセージはすべて例外キューに移動します。

メッセージのページング

OracleAS JMS サーバーでは、次の場合にメッセージ本文のページング・インとページング・アウトがサポートされます。

- メッセージが永続的な配信モードに設定されている場合。
- メッセージが永続的な Destination オブジェクトに送信される場合（3-10 ページの「OracleAS JMS のファイル・ベースの永続性」を参照）。
- OC4J サーバーの JVM がメモリー不足の場合。

ページングされるのはメッセージ本文のみです。メッセージ・ヘッダーとプロパティはページング対象とみなされません。ページングしきい値は、OracleAS JMS のシステム・プロパティ oc4j.jms.pagingThreshold を使用して設定します。この DOUBLE 値（範囲 [0,1] に限定）は、OracleAS JMS サーバーがメッセージ本文をページング対象とみなすメモリー使用率の下限を表します。この値は、Java 仮想マシン（JVM）で使用中のメモリー量の見積です。この値の範囲は 0.0（プログラムではメモリーがまったく使用されていない）～ 1.0（プログラムで使用可能メモリーがすべて使用されている）です。

値の範囲は 0.0 ～ 1.0 です。JVM メモリーを使用しない Java プログラムを記述するのはほとんど不可能であり、JVM ヒープがいっぱいになる前にメモリーすべてを使用するとプログラムはほぼ常に終了します。

たとえば、ページングしきい値が 0.5 の場合に JVM のメモリー使用率が 0.6 になると、OracleAS JMS はメモリー使用率がしきい値を下回るまで、または、それ以上はメッセージ本文をページング・アウトできなくなるまで、できるだけ多数のメッセージ本文をページング・アウトしようとします。

ページング・アウトされたメッセージを JMS クライアントが要求すると、OracleAS JMS サーバーはそのメッセージ本文を（JVM のメモリー使用率に関係なく）自動的にページング・インし、正しいメッセージ・ヘッダーと本文をクライアントに配信します。クライアントに配信されたメッセージは、サーバー JVM でのメモリー使用率に応じて再びページング・アウトの対象とみなすことができます。

メモリー使用率がページングしきい値を下回ると、OracleAS JMS サーバーはメッセージ本文のページング・アウトを停止します。すでにページング・アウトされていたメッセージ本文が自動的にページング・インされることはありません。メッセージ本文のページング・インは、必要な場合（つまり、メッセージがデキューされるかクライアントにより参照される場合）にのみ発生します。

デフォルトでは、OracleAS JMS サーバーのページングしきい値は 1.0 に設定されます。つまり、デフォルトでは、OracleAS JMS サーバーはメッセージ本文をページングしません。

JMS アプリケーション、そこで送受信されるメッセージのサイズ、実際の使用例での経験とメモリー使用率の監視結果に応じて、ページングしきい値に適切な値を選択する必要があります。

ページングしきい値には、正しい値を指定する必要があります。JMS のセマンティクスは、ページングが有効化されているかどうかに関係なく常に保たれます。ページングしきい値の制御により、OracleAS JMS サーバーはメモリー内でページングしない場合よりも多数のメッセージを処理できます。

OracleAS JMS の jms.xml 構成ファイルの要素

この項では、`jms.xml` で OC4J JMS 構成に使用する XML 要素について説明します。XML ファイル内の要素の順序構成は次のとおりです。

```
<jms-server>
   <queue>
      <description></description>
   </queue>
   <topic>
     <description></description>
   </topic>
   <connection-factory></connection-factory>
   <queue-connection-factory></queue-connection-factory>
   <topic-connection-factory></topic-connection-factory>
   <xa-connection-factory></xa-connection-factory>
   <xa-queue-connection-factory></xa-queue-connection-factory>
   <xa-topic-connection-factory></xa-topic-connection-factory>
   <log>
     <file></file>
   </log>
</jms-server>
```

表 3-4 に、JMS の構成要素を示します。

表 3-4 JMS の構成要素

要素	説明	属性
`jms-server`	OC4J JMS サーバー構成のルート要素です。	`host`: この OC4J JMS サーバーのバインド先として `String` で定義されているホスト名（DNS またはドット表記法によるホスト名）です。デフォルトでは、JMS サーバーは `0.0.0.0`（構成ファイルでは `[ALL]`）にバインドします。この属性はオプションです。 `port`: この OC4J JMS サーバーのバインド先 `int`（有効な TCP/IP ポート番号）として定義されているポートです。デフォルト設定は `9127` です。この設定は、OC4J のスタンドアロン構成にのみ適用されます。Oracle Application Server では、構成ファイル内のポート設定は、OC4J サーバーの起動時に（OPMN、DCM などにより）使用されるコマンドライン引数で上書きされます。この属性はオプションです。

表 3-4 JMS の構成要素（続き）

要素	説明	属性
queue	この要素ではOracleAS JMS のキューを構成します。キューは OC4J JMS の起動時に使用可能であり、サーバーが再起動または再構成されるまで使用可能です。0（ゼロ）個以上のキューを任意の順序で構成できます。新規に構成したキューは、OC4J を再起動するまで使用できません。	name: この必須属性は、OC4J JMS キューのプロバイダ固有名（String）です。この名前には、空でなく有効な任意の文字列を使用できます（空白や他の特殊文字は使用できますがお薦めしません）。この属性で指定した名前を Session.createQueue() で使用すると、プロバイダ固有名を JMS キューに変換できます。キューとトピックの両方に同じ名前を指定することはできません。ただし、複数のキューに同じ名前と異なるロケーションを指定することはできます。デフォルト名はありません。 location: この必須属性では、キューのバインド先となる JNDI ロケーション（String）を記述します。この値は、有効な名前に関する JNDI ルールに従う必要があります。OC4J JMS コンテナ内では、ロケーションはそのままバインドされ、アクセスできます。アプリケーション・クライアントでは、名前は java:comp/env/ JNDI ネームスペースの一部であり、関連するデプロイメント・ディスクリプタで適切に宣言する必要があります。関連デプロイメント・ディスクリプタが適切に指定されていれば、java:comp/env/ 名もコンテナ内で使用できます。ロケーションは、すべての Destination オブジェクト間および jms.xml 内のコネクション・ファクトリ要素間で一意である必要があります。デフォルトはありません。 persistence-file: オプションのパスとファイル名（String）です。persistence-file 属性のパスは、ファイルの絶対パス、または application.xml に定義されている persistence ディレクトリに対する相対パスです。デフォルト・パスは、Oracle Application Server 環境の場合は *J2EE_HOME*/persistence/<island>、スタンドアロン環境の場合は *J2EE_HOME*/persistence です。 persistence-file 属性の意味の詳細は、3-11 ページの「リカバリ」を参照してください。jms.xml 内で名前が同じで位置が異なる複数の queue 要素を宣言する場合は、そのすべてで persistence-file に同じ値を指定するか、またはまったく値を指定しません。複数の宣言のうち 1 つ以上で persistence-file を指定すると、このキューにはその値が使用されます。

表 3-4 JMS の構成要素（続き）

要素	説明	属性
topic	この要素ではOracleAS JMS のトピックを構成します。トピックは OC4J JMS の起動時に使用可能であり、サーバーが再起動または再構成されるまで使用可能です。0（ゼロ）個以上のトピックを任意の順序で構成できます。新規に構成したトピックは、OC4J を再起動するまで使用できません。	name: この必須属性は、OC4J JMS トピックのプロバイダ固有名（String）です。この名前には、空でなく有効な任意の文字列を使用できます（空白や他の特殊文字は使用できますがお薦めしません）。この属性で指定した名前を Session.createTopic() で使用すると、プロバイダ固有名を JMS トピックに変換できます。キューとトピックの両方に同じ名前を指定することはできません。ただし、複数のトピックに同じ名前と異なるロケーションを指定することはできます。デフォルト名はありません。 location: この必須属性では、トピックのバインド先となる JNDI ロケーション（String）を記述します。この値は、有効な名前に関する JNDI ルールに従う必要があります。OC4J JMS コンテナ内では、ロケーションはそのままバインドされ、アクセスできます。アプリケーション・クライアントでは、名前は java:comp/env/ JNDI ネームスペースの一部であり、関連するデプロイメント・ディスクリプタで適切に宣言する必要があります。 関連デプロイメント・ディスクリプタが適切に指定されていれば、java:comp/env/ 名もコンテナ内で使用できます。ロケーションは、jms.xml 内のすべてのトピックおよびコネクション・ファクトリ要素間で一意である必要があります。デフォルトはありません。 persistence-file: オプションのパスとファイル名（String）です。persistence-file 属性のパスは、ファイルの絶対パス、または application.xml に定義されている persistence ディレクトリに対する相対パスです。デフォルト・パスは、Oracle Application Server 環境の場合は *J2EE_HOME*/persistence/<island>、スタンドアロン環境の場合は *J2EE_HOME*/persistence です。 persistence-file 属性の意味の詳細は、3-11 ページの「リカバリ」を参照してください。jms.xml 内で名前が同じで位置が異なる複数の queue 要素または topic 要素を宣言する場合は、そのすべてで persistence-file に同じ値を指定するか、またはまったく値を指定しません。複数の宣言のうち 1 つ以上で persistence-file を指定すると、このトピックにはその値が使用されます。
description	キューまたはトピックの用途をユーザーに知らせるユーザー定義文字列です。	
connection-factory	JMS ドメインのコネクション・ファクトリ構成です。	この要素の属性は、表 3-5 を参照してください。
queue-connection-factory	JMS ドメインのコネクション・ファクトリ構成です。	この要素の属性は、表 3-5 を参照してください。
topic-connection-factory	JMS ドメインのコネクション・ファクトリ構成です。	この要素の属性は、表 3-5 を参照してください。
xa-connection-factory	コネクション・ファクトリ構成の XA バリアントです。	この要素の属性は、表 3-5 を参照してください。
xa-queue-connection-factory	コネクション・ファクトリ構成の XA バリアントです。	この要素の属性は、表 3-5 を参照してください。
xa-topic-connection-factory	コネクション・ファクトリ構成の XA バリアントです。	この要素の属性は、表 3-5 を参照してください。

表 3-4 JMS の構成要素（続き）

要素	説明	属性
log	ファイルまたは ODL 形式による JMS アクティビティのロギングを有効化します。ロギングの詳細は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』の「OC4J ロギングの有効化」を参照してください。	

表 3-5 に、コネクション・ファクトリ定義の属性を示します。

表 3-5 コネクション・ファクトリ構成の属性

属性	型	必須 / オプション	デフォルト	説明
location	String	必須	（該当なし）	コネクション・ファクトリのバインド先となる JNDI ロケーション。この値は、有効な名前に関する JNDI ルールに従う必要があります。OC4J JMS コンテナ内では、ロケーションはそのままバインドされ、アクセスできます。アプリケーション・クライアントでは、名前は java:comp/env/ JNDI ネームスペースの一部であり、関連するデプロイメント・ディスクリプタで適切に宣言する必要があります。関連デプロイメント・ディスクリプタが適切に指定されていれば、java:comp/env/ 名もコンテナ内で使用できます。ロケーションは、すべての Destination および jms.xml 内のコネクション・ファクトリ要素間で一意である必要があります。
host	String（DNS またはドット表記法によるホスト名）	オプション	[ALL]	このコネクション・ファクトリの接続先となる固定の OC4J JMS ホスト。デフォルトでは、コネクション・ファクトリは jms-server 要素に構成されているのと同じホストを使用します。デフォルト以外の値を使用すると、ローカルで使用可能な OC4J JMS サーバーや他の Oracle Application Server またはクラスタ構成を省略し、すべての JMS 操作を特定の OC4J JVM に送ることができます。
port	int（有効な TCP/IP ポート番号）	オプション	9127	このコネクション・ファクトリの接続先となる固定の OC4J JMS ポート。デフォルトでは、コネクション・ファクトリは jms-server 要素に構成されているのと同じポート（あるいは、コマンドラインで Oracle Application Server またはクラスタ構成に対して指定したポートの値）を使用します。デフォルト以外の値を使用すると、ローカルで使用可能な OC4J JMS サーバーや他の Oracle Application Server またはクラスタ構成を省略し、すべての JMS 操作を特定の OC4J JVM に送ることができます。
username	String	オプション	空の文字列	このコネクション・ファクトリから作成される JMS デフォルト接続の認証に使用するユーザー名。ユーザー名自体は、他の OC4J 機能を使用して正しく作成して構成する必要があります。
password	String	オプション	空の文字列	このコネクション・ファクトリから作成される JMS デフォルト接続の認証に使用するパスワード。パスワード自体は、他の OC4J 機能を使用して正しく作成して構成する必要があります。

表 3-5 コネクション・ファクトリ構成の属性（続き）

属性	型	必須 / オプション	デフォルト	説明
clientID	String	オプション	空の文字列	このコネクション・ファクトリから作成される接続用に、管理上構成されている固定の JMS clientID。clientID を指定しない場合、デフォルトは空の文字列です。この値は、JMS 仕様に基づいて、クライアント・プログラムで上書きすることもできます。clientID が使用されるのはトピックの永続サブスクリプションの場合のみで、この値はキューと非永続トピックの操作には関係しません。

注意： 表 3-5 のプロパティ password は、パスワードの間接化をサポートしています。詳細は、『Oracle Application Server Containers for J2EE セキュリティ・ガイド』を参照してください。

例

この項では、コネクション・ファクトリ構成の一部分であるコード例を示します。

次のコード例は、コネクション・ファクトリ jms/Cf、キュー・コネクション・ファクトリ jms/Qcf、XA トピック・コネクション・ファクトリ jmx/xaTcf を構成しています。

```
<connection-factory location="jms/Cf">
</connection-factory>

<queue-connection-factory location="jms/Qcf">
</queue-connection-factory>

<xa-topic-connection-factory location="jms/xaTcf"
    username="foo" password="bar" clientID="baz">
</xa-topic-connection-factory>
```

トピック・コネクション・ファクトリを追加する場合は、一意の名前を使用する必要があります。たとえば、コネクション・ファクトリ jms/Cf（前述）と同じ名前を使用することはできません。したがって、次の指定は無効です。

```
<!-- Invalid: cannot reuse "location"  -->
<topic-connection-factory location="jms/Cf">
</topic-connection-factory>
```

次のコード例は、キューおよびトピック構成の一部分です。このセグメントでは、キュー foo とトピック bar を作成しています。

```
<queue name="foo" location="jms/foo">
</queue>

<topic name="bar" location="jms/bar">
</topic>
```

特定の位置は予約済であり、jms.xml 構成ファイルでは再定義できません。次のコード例では、予約済の位置であるため、キューの位置を定義するときに jms/Oc4jJmsExceptionQueue が使用できないことを示しています。

```
<!-- Invalid: cannot use a reserved "location" -->
<queue name="fubar" location="jms/Oc4jJmsExceptionQueue">
</queue>
```

キューおよびトピックの永続性ファイルを定義するときに、位置とファイル名を定義できます。また、ファイル名が同じであれば、複数の永続性ファイルを指定できます。そのため、永続性ファイルは同じキューについて 2 つの位置に書き込まれます。

```
<queue name="foo" location="jms/persist" persistence-file="pers">
</queue>

<!-- OK: multiple persistence file specification ok if consistent -->
<queue name="foo" location="jms/file" persistence-file="pers">
</queue>

<!-- Invalid: multiple persistence file specifications should be consistent -->
<queue name="foo" location="jms/file1" persistence-file="notpers">
</queue>
```

また、同じ永続性ファイルに書き込む 2 つのオブジェクトを使用することはできません。位置が異なる場合でも、キューまたはトピックにはそれぞれ固有の永続性ファイル名が必要です。

```
<topic name="demoTopic" location="jms/dada" persistence-file="/tmp/abcd">
</topic>

<!-- Invalid: cannot reuse persistence-file for multiple destinations -->
<topic name="demoTopic1" location="jms/dada1" persistence-file="/tmp/abcd">
</topic>
```

OracleAS JMS のシステム・プロパティ

OC4J JMS では、JVM システム・プロパティを介して OC4J JMS サーバーと JMS クライアントを実行時に構成できます。これらのプロパティは、JMS の基本機能には影響しません。OC4J JMS 固有の機能、拡張機能およびパフォーマンスの最適化に関係します。

表 3-6 に、これらの管理プロパティの概要を示します。

表 3-6 OC4J JMS の管理プロパティ

JVM システム・プロパティ	プロパティの型	デフォルト値	サーバー / クライアント	用途
`oc4j.jms.serverPoll`	`long`	`15000`	JMS クライアント	JMS コネクションが OC4J サーバーを ping し、通信例外を例外リスナーにレポートする間隔（ミリ秒数）。
`oc4j.jms.messagePoll`	`long`	`1000`	JMS クライアント	JMS 非同期コンシューマが OC4J JMS サーバーで新規メッセージの有無をチェックするまで待機する最大間隔（ミリ秒数）。
`oc4j.jms.listenerAttempts`	`int`	`5`	JMS クライアント	メッセージが配信不可能と宣言されるまでの、リスナーによる配信試行回数。
`oc4j.jms.maxOpenFiles`	`int`	`64`	OC4J サーバー	OC4J JMS サーバーでの（永続性ファイルの）オープン・ファイル記述子の最大数。サーバー構成で指定されている永続的な `Destination` オブジェクトの数が、オペレーティング・システムで許可されるオープン・ファイル記述子の最大数よりも多い場合に関連します。
`oc4j.jms.saveAllExpired`	`boolean`	`false`	OC4J サーバー	すべての `Destination` オブジェクト（永続的、非永続的および一時的）の、期限切れになったメッセージすべてを、OC4J JMS の例外キューに保存します。

表 3-6 OC4J JMS の管理プロパティ（続き）

JVM システム・プロパティ	プロパティの型	デフォルト値	サーバー / クライアント	用途
`oc4j.jms.socketBufsize`	`int`	`64 * 1024`	JMS クライアント	クライアント / サーバー通信に TCP/IP ソケットを使用する場合は、ソケットの入出力ストリームに指定されたバッファ・サイズを使用します。最小バッファ・サイズの 8KB が規定されます。クライアントとサーバーの間で送信されるメッセージのサイズが大きいほど、バッファ・サイズを大きくすると妥当なパフォーマンスが得られます。
`oc4j.jms.debug`	`boolean`	`false`	JMS クライアント	true の場合は、JMS クライアントと OC4J JMS サーバーで `NORMAL` イベントのトレースが有効化されます。すべてのログ・イベント（`NORMAL`、`WARNING`、`ERROR` および `CRITICAL`）が両方の `stderr` に送られ、可能な場合は *`J2EE_HOME`*`/log/server.log` または *`J2EE_HOME`*`/log/jms.log` にも送られます。このプロパティを `true` に設定すると、通常は大量のトレース情報が生成されます。
`oc4j.jms.noDms`	`boolean`	`false`	JMS クライアント	`true` の場合は、インスツルメント処理が無効化されます。
`oc4j.jms.forceRecovery`	`boolean`	`false`	OC4J サーバー	`true` の場合は、破損した永続性ファイルが強制的にリカバリされます。デフォルトでは、OC4J JMS サーバーはヘッダーが破損した永続性ファイルのリカバリを実行しません（通常は、この状態を構成エラーと区別できないため）。強制リカバリにより、OC4J JMS サーバーはほぼ常に正常に起動し、永続性ファイルと `Destination` オブジェクトを使用可能にします。
`oc4j.jms.pagingThreshold`	`double`	`1.0`	OC4J サーバー	OracleAS JMS サーバーがメッセージ本文をページング対象とみなし始めるメモリー使用率の下限を表します。この値は、JVM で使用中のメモリー量の見積です。この値の範囲は 0.0（プログラムではメモリーがまったく使用されていない）～ 1.0（プログラムで使用可能メモリーがすべて使用されている）です。 詳細は、3-15 ページの「メッセージのページング」を参照してください。
`oc4j.jms.usePersistenceLockFiles`	`boolean`	`true`	OC4J サーバー	ロック・ファイルを使用して、OracleAS JMS の永続性ファイルを複数の OC4J プロセスによる上書きから保護する必要があるかどうかを制御します。デフォルトでは、ロック・ファイルは複数の OC4J プロセスによる意図しない上書きを防止するために使用されます。ただし、そのためには、ユーザーは OC4J が異常終了した場合にロック・ファイルを手動で削除する必要があります。このシステム・プロパティを `false` に設定すると、永続的な宛先用のロック・ファイルは作成されません。各永続性ファイルにアクセスするアクティブ・プロセスが 1 つしかないことを保証できる場合にのみ、このプロパティを `false` に設定してください。このプロパティは OC4J サーバーの起動時に設定します。停止するまでは、すべての JMS クライアントに対して有効になっています。

リソース・プロバイダ

OC4J には、JMS プロバイダを透過的にプラグインするための ResourceProvider インタフェースが用意されています。

OC4J の ResourceProvider インタフェースを使用すると、EJB、サーブレットおよび OC4J クライアントは多数の異なる JMS プロバイダにアクセスできます。リソースは、java:comp/resource/ の下で使用可能です。Oracle JMS には、ResourceProvider インタフェースを使用してアクセスします。Oracle JMS の詳細は、3-24 ページの「Oracle JMS」を参照してください。

カスタム・リソース・プロバイダの構成

カスタム・リソース・プロバイダは、次のいずれかの方法で構成できます。

- すべてのアプリケーション（グローバル）に対するリソース・プロバイダである場合は、グローバルの application.xml ファイルを構成します。
- 単一のアプリケーション（ローカル）に対するリソース・プロバイダである場合は、アプリケーションの orion-application.xml ファイルを構成します。

カスタム・リソース・プロバイダを追加するには、選択した XML ファイル（前述）に次のコードを追加します。

```
<resource-provider class="providerClassName" name="JNDIname">
     <description>description </description>
     <property name="name" value="value" />
</resource-provider>
```

<resource-provider> 属性は、次のように構成します。

- class: リソース・プロバイダ・クラス名。
- name: リソース・プロバイダの識別名。この名前は、次のように、アプリケーションの JNDI でリソース・プロバイダを検索するときに使用されます。

  ```
  java:comp/resource/JNDIname/
  ```

<resource-provider> のサブ要素は、次のように構成します。

- description サブ要素 : 特定のリソース・プロバイダの記述。
- property サブ要素 : *name* および *value* 属性を使用して、リソース・プロバイダに与えるパラメータを識別します。name 属性ではパラメータ名を識別し、その値を value 属性で指定します。

リソース・プロバイダを取得するには、JNDI ルックアップで次の構文を使用します。

```
java:comp/resource/JNDIname/resourceName
```

JNDIname はリソース・プロバイダ名（<resource-provider> 要素の name 属性で指定）、*resourceName* はアプリケーション実装で定義されているリソース名です。リソース・プロバイダとして定義されている Oracle JMS の例は、3-24 ページの「OJMS をリソース・プロバイダとして使用する方法」を参照してください。

Oracle JMS

Oracle JMS（OJMS）は、Oracle データベースの Oracle Streams Advanced Queuing（AQ）機能への JMS インタフェースです。OJMS は JMS 1.0.2b 仕様を実装し、J2EE 1.3 仕様との互換性があります。OC4J での OJMS アクセスは、リソース・プロバイダ・インタフェースを介して発生します。AQ と OJMS の詳細は、『Oracle Streams アドバンスト・キューイング・ユーザーズ・ガイドおよびリファレンス』を参照してください。

この項の内容は次のとおりです。

- OJMS をリソース・プロバイダとして使用する方法
- Oracle Application Server および Oracle データベースと OJMS の併用

OJMS をリソース・プロバイダとして使用する方法

OJMS キューにアクセスする手順は、次のとおりです。

1. データベースに OJMS をインストールして構成します。3-24 ページの「JMS プロバイダのインストールと構成」を参照してください。
2. データベース上で RDBMS ユーザーを作成して権限を割り当てます。JMS アプリケーションは、このユーザーを使用してバックエンド・データベースに接続します。ユーザーには、OJMS 操作を実行するための権限が必要です。OJMS を使用すると、データベース・ユーザーは任意のスキーマのキューにアクセスできます。ただし、ユーザーに適切なアクセス権限がある必要があります。3-25 ページの「ユーザーの作成と権限の割当て」を参照してください。
3. OJMS で JMS `Destination` オブジェクトを作成します。3-25 ページの「JMS Destination オブジェクトの作成」を参照してください。
4. OC4J の XML 構成で、バックエンド・データベースに関する情報を使用して、`<resource-provider>` 要素で OJMS リソース・プロバイダを定義します。必要に応じて、データ・ソースまたは LDAP ディレクトリ・エントリを作成します。3-26 ページの「OJMS リソース・プロバイダの定義」を参照してください。
5. JNDI ルックアップを介して実装内のリソースにアクセスします。3-29 ページの「OJMS リソースへのアクセス」を参照してください。

JMS プロバイダのインストールと構成

ユーザーまたは DBA が、『Oracle Streams アドバンスト・キューイング・ユーザーズ・ガイドおよびリファレンス』と汎用データベース・マニュアルに従って OJMS をインストールする必要があります。この JMS プロバイダをインストールして構成した後に、追加構成を適用する必要があります。これには、次が含まれます。

1. ユーザーまたは DBA が、JMS クライアントからデータベースへの接続に使用する RDBMS ユーザーを作成する必要があります。このユーザーに、OJMS 操作を実行するための適切なアクセス権限を付与します。3-25 ページの「ユーザーの作成と権限の割当て」を参照してください。
2. ユーザーまたは DBA が、JMS `Destination` オブジェクトをサポートするための表とキューを作成する必要があります。3-25 ページの「JMS Destination オブジェクトの作成」を参照してください。

注意： 次の項では、SQL を使用してキュー、トピック、それぞれの表を作成し、JMS デモで提供されている権限を割り当てます。JMS デモについては、次の URL の OTN Web サイトの OC4J サンプル・コード・ページを参照してください。

```
http://www.oracle.com/technology/tech/java/oc4j/demos/
index.html
```

ユーザーの作成と権限の割当て

JMS クライアントからデータベースへの接続に使用する RDBMS ユーザーを作成します。このユーザーに、OJMS 操作を実行するためのアクセス権限を付与します。必要な権限は、必要な機能に応じて異なります。各種機能に必要な権限の詳細は、『Oracle Streams アドバンスト・キューイング・ユーザーズ・ガイドおよびリファレンス』を参照してください。

次の例では、`jmsuser` を作成します。このユーザーを固有のスキーマ内で作成し、OJMS 操作に必要な権限を付与する必要があります。次の文を実行するには、`SYS DBA` であることが必要です。

```
DROP USER jmsuser CASCADE ;

GRANT connect,resource,AQ_ADMINISTRATOR_ROLE TO jmsuser
   IDENTIFIED BY jmsuser ;
GRANT execute ON sys.dbms_aqadm  TO  jmsuser;
GRANT execute ON sys.dbms_aq     TO  jmsuser;
GRANT execute ON sys.dbms_aqin   TO  jmsuser;
GRANT execute ON sys.dbms_aqjms  TO  jmsuser;

connect jmsuser/jmsuser;
```

ユーザーの必要に応じて、2 フェーズ・コミット権限やシステム管理権限など、他の権限の付与が必要になる場合があります。2 フェーズ・コミット権限の詳細は、第 7 章の「Java Transaction API」を参照してください。

JMS Destination オブジェクトの作成

各 JMS プロバイダには、JMS Destination オブジェクトを作成するための固有のメソッドが必要です。`DBMS_AQADM` パッケージと OJMS メッセージ・タイプの詳細は、『Oracle Streams アドバンスト・キューイング・ユーザーズ・ガイドおよびリファレンス』を参照してください。この例では、OJMS には次のメソッドが必要です。

注意 : OJMS の例で表の作成に使用する SQL は、次の URL の OTN Web サイトの OC4J サンプル・コード・ページから入手できる JMS の例に含まれています。

`http://www.oracle.com/technology/tech/java/oc4j/demos/index.html`

---

1. JMS `Destination`（キューまたはトピック）を処理する表を作成します。

   OJMS では、トピックとキューの両方にキュー表が使用されます。JMS の例では、キュー用に 1 つの表 `demoTestQTab` が作成されます。

   このキュー表を作成するには、次の SQL を実行します。

   ```
   DBMS_AQADM.CREATE_QUEUE_TABLE(
           Queue_table           => 'demoTestQTab',
           Queue_payload_type    => 'SYS.AQ$_JMS_MESSAGE',
           sort_list => 'PRIORITY,ENQ_TIME',
           multiple_consumers  => false,
           compatible            => '8.1.5');
   ```

   `multiple_consumers` パラメータは、複数のコンシューマが存在するかどうかを指定します。そのため、キューの場合は常に `false`、トピックの場合は常に `true` です。

2. JMS `Destination` を作成します。トピックを作成する場合は、そのトピックの各サブスクライバを追加する必要があります。JMS の例では、1 つのキュー `demoQueue` が必要です。

次のコマンドでは、キュー表 demoTestQTab にキュー demoQueue が作成されます。作成後に、このキューが起動されます。

```
DBMS_AQADM.CREATE_QUEUE(
      Queue_name          => 'demoQueue',
      Queue_table         => 'demoTestQTab');

DBMS_AQADM.START_QUEUE(
      queue_name         => 'demoQueue');
```

トピックを追加する場合は、次の例を参照してください。この例は、トピック表 demoTestTTab にトピック demoTopic を作成する方法を示しています。作成後は、2 つの永続サブスクライバがトピックに追加されます。最後にトピックが起動され、ユーザーにそのトピックに対する権限が付与されます。

注意： Oracle AQ では、DBMS_AQADM.CREATE_QUEUE メソッドを使用してキューとトピックの両方が作成されます。

```
DBMS_AQADM.CREATE_QUEUE_TABLE(
        Queue_table            => 'demoTestTTab',
        Queue_payload_type     => 'SYS.AQ$_JMS_MESSAGE',
        multiple_consumers  => true,
        compatible             => '8.1.5');
DBMS_AQADM.CREATE_QUEUE( 'demoTopic', 'demoTestTTab');
DBMS_AQADM.ADD_SUBSCRIBER('demoTopic',
                         sys.aq$_agent('MDSUB', null, null));
DBMS_AQADM.ADD_SUBSCRIBER('demoTopic',
                         sys.aq$_agent('MDSUB2', null, null));
DBMS_AQADM.START_QUEUE('demoTopic');
```

注意： ここで定義する名前は、アプリケーションのデプロイメント・ディスクリプタでキューまたはトピックの定義に使用したのと同じ名前にする必要があります。

OJMS リソース・プロバイダの定義

OJMS リソース・プロバイダを定義するには、Oracle Enterprise Manager 10*g* を使用する方法と、XML ファイルを手動で編集する方法があります。次の各項では、それぞれの方法について説明します。

- Oracle Enterprise Manager 10g を介した OJMS プロバイダの構成
- OC4J XML ファイルでの OJMS プロバイダの構成

Oracle Enterprise Manager 10*g* を介した OJMS プロバイダの構成　OJMS プロバイダは、JMS セクションで Application Server Control を使用して構成できます。OJMS プロバイダを追加するには、「管理」ページの「アプリケーション・デフォルト」列で「JMS プロバイダ」を選択します。

「**新規 JMS プロバイダの追加**」ボタンをクリックし、各 JMS プロバイダを構成します。

OJMS またはサード・パーティの JMS プロバイダを構成できます。OC4J インストールでは、トピックとキューを除き、事前構成済の OracleAS JMS が常に提供されます。

注意： OJMS プロバイダとサード・パーティのプロバイダはどちらも構成方法が同じであるため、この項の説明にはサード・パーティの JMS プロバイダの構成手順も含まれています。

JMS プロバイダのタイプを選択した後、次の情報を指定する必要があります。

- OJMS: OJMS をインストールして構成するデータベースのデータ・ソース名と JNDI ロケーションを指定します。
- サード・パーティの JMS プロバイダ : サード・パーティのプロバイダの名前、JNDI 初期コンテキスト・ファクトリ・クラスおよび JNDI URL を指定します。この JMS プロバイダについて `java.naming.factory.initial` および `java.naming.provider.url` などの JNDI プロパティを追加するには、「**プロパティの追加**」をクリックします。行が追加され、各 JNDI プロパティの名前と値を追加できます。

ここで構成するのはプロバイダのみで、`Destination` オブジェクト（トピック、キューおよびサブスクリプション）は構成しません。

特定のアプリケーション専用の JMS プロバイダを構成するには、「アプリケーション」ページでアプリケーションを選択し、「リソース」列にスクロールして「JMS プロバイダ」を選択します。デフォルトの JMS プロバイダの場合と同じ画面が表示されます。

OC4J XML ファイルでの OJMS プロバイダの構成 `<resource-provider>` 要素内で OJMS プロバイダを構成します。

- すべてのアプリケーション（グローバル）に対する JMS プロバイダである場合は、グローバルの `application.xml` ファイルを構成します。
- 単一のアプリケーション（ローカル）に対する JMS プロバイダである場合は、アプリケーションの `orion-application.xml` ファイルを構成します。

次のコード例に、OJMS に XML 構文を使用して JMS プロバイダを構成する方法を示します。

```
<resource-provider class="oracle.jms.OjmsContext" name="ojmsdemo">
   <description> OJMS/AQ </description>
   <property name="datasource" value="jdbc/emulatedDS"></property>
</resource-provider>
```

`<resource-provider>` 要素の属性は、次のとおりです。

- `class` 属性 : OJMS プロバイダは `class` 属性で構成されている `oracle.jms.OjmsContext` クラスにより実装されます。
- `name` 属性 : OJMS リソース・プロバイダの名前は `ojmsdemo` です。

また、`<property>` 要素の `name/value` 属性では、OJMS で使用されるデータ・ソースが識別されます。トピックまたはキューはこのデータ・ソースに接続して、メッセージ操作を容易にする表とキューにアクセスします。この例では、データ・ソースは `jdbc/emulatedDS` として識別されます。

リソース・プロバイダ構成で `<property>` 要素の属性を構成する方法は、アプリケーションの実行場所に応じて異なります。OJMS を使用し、データベース内の AQ にアクセスする場合は、データ・ソース・プロパティまたは URL プロパティを使用してリソース・プロバイダを構成する必要があります。ここでは、それぞれの方法について説明します。

- データ・ソース・プロパティを使用したリソース・プロバイダの構成
- URL プロパティを使用したリソース・プロバイダの構成

データ・ソース・プロパティを使用したリソース・プロバイダの構成

アプリケーションが OC4J 内で実行される場合は、データ・ソースを使用します。データ・ソースを使用するには、最初に、OJMS プロバイダがインストールされている `data-sources.xml` ファイル内で構成する必要があります。JMS トピックおよびキューでは、メッセージ操作を容易にするためにデータベース表とキューが使用されます。使用するデータ・ソースのタイプは、必要な機能に応じて異なります。

注意： トランザクションがない場合や、1 フェーズ・トランザクションの場合は、エミュレートされたデータ・ソースまたはエミュレートされていないデータ・ソースのどちらでも使用できます。2 フェーズ・コミット・トランザクションをサポートする場合に使用できるのは、エミュレートされていないデータ・ソースのみです。詳細は、第 7 章を参照してください。

例 3-3 シン JDBC ドライバでエミュレートされたデータ・ソース

次の例には、シン JDBC ドライバを使用してエミュレートされたデータ・ソースが含まれています。2 フェーズ・コミット・トランザクションをサポートするには、エミュレートされていないデータ・ソースを使用します。エミュレートされたデータ・ソースとエミュレートされていないデータ・ソースの違いについては、4-7 ページの「データ・ソースの定義」を参照してください。

この例は、XML 定義の形式で示されています。Oracle Enterprise Manager 10*g* を介して新規データ・ソースを構成に追加する手順については、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』を参照してください。

```
<data-source
  class="com.evermind.sql.DriverManagerDataSource"
  name="OracleDS"
  location="jdbc/emulatedOracleCoreDS"
  xa-location="jdbc/xa/emulatedOracleXADS"
  ejb-location="jdbc/emulatedDS"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="jmsuser"
  password="jmsuser"
  url="jdbc:oracle:thin:@//localhost:5521/oracle.regress.rdbms.dev.us.oracle.com"
/>
```

このデータ・ソースを環境にあわせてカスタマイズします。たとえば、`myhost:1521:orcl` をデータベースのホスト名、ポートおよび SID に置き換えます。

注意： パスワードを明確に指定するかわりに、パスワードの間接化を使用できます。詳細は、『Oracle Application Server Containers for J2EE サービス・ガイド』を参照してください。

次に、データ・ソース名を使用してリソース・プロバイダを構成します。ここでは、データ・ソース `jdbc/emulatedDS` を使用した OJMS のリソース・プロバイダの構成例を示します。

```
<resource-provider class="oracle.jms.OjmsContext" name="ojmsdemo">
  <description> OJMS/AQ </description>
  <property name="datasource" value="jdbc/emulatedDS"></property>
</resource-provider>
```

データ・ソースの構成の詳細は、4-7 ページの「データ・ソースの定義」を参照してください。

URL プロパティを使用したリソース・プロバイダの構成

このリリースでは、データ・ソースはシリアライズ可能ではありません。そのため、アプリケーション・クライアントは URL 定義を使用して OJMS リソースにアクセスする必要があります。アプリケーションがスタンドアロン・クライアントの場合（つまり、OC4J の外部で実行される場合）は、OJMS がインストールされているデータベースの URL を持つ URL プロパティを使用して `<resource-provider>` 要素を構成します。また、必要な場合は、そのデータベース用のユーザー名とパスワードを指定します。次の例に URL 構成を示します。

```
<resource-provider class="oracle.jms.OjmsContext" name="ojmsdemo">
  <description> OJMS/AQ </description>
  <property name="url"
value="jdbc:oracle:thin:@//localhost:5521/oracle.regress.rdbms.dev.us.oracle.com">
  </property>
```

```
<property name="username" value="user"></property>
<property name="password" value="passwd"></property>
```

OJMS リソースへのアクセス

OJMS リソースにアクセスするステップは、OracleAS JMS リソースの場合と同じです。3-5 ページの「メッセージの送信のステップ」を参照してください。唯一の違いは、JNDI ルックアップで提供されるリソースの名前です。

- コネクション・ファクトリの OJMS 構文は、"java:comp/resource" + JMS プロバイダ名 + "TopicConnectionFactories" または "QueueConnectionFactories" + ユーザー定義名です。ユーザー定義名には、他の構成と一致しない任意の名前を指定できます。xxxConnectionFactories には、定義するファクトリのタイプの詳細を記述します。この例では、JMS プロバイダ名は <resource-provider> 要素で ojmsdemo として定義されています。
  - キュー・コネクション・ファクトリの場合 : JMS プロバイダ名は ojmsdemo で、名前に myQCF を使用することにしたため、コネクション・ファクトリ名は java:comp/resource/ojmsdemo/QueueConnectionFactories/myQCF となります。
  - トピック・コネクション・ファクトリの場合 : JMS プロバイダ名は ojmsdemo で、名前に myTCF を使用することにしたため、コネクション・ファクトリ名は java:comp/resource/ojmsdemo/TopicConnectionFactories/myTCF となります。

  前述の myQCF と myTCF が示すユーザー定義名は、ロジックの他の場所では使用されません。そのため、任意の名前を選択できます。

- すべての Destination の OJMS 構文は、"java:comp/resource" + JMS プロバイダ名 + "Topics" または "Queues" + Destination 名です。Topic または Queue には、定義する Destination のタイプの詳細を記述します。Destination 名は、データベースで定義されている実際のキュー名またはトピック名です。

  この例では、JMS プロバイダ名は <resource-provider> 要素で ojmsdemo として定義されています。データベースでは、キュー名は demoQueue です。
  - キューの場合 : JMS プロバイダ名が ojmsdemo で、キュー名が demoQueue であれば、トピックの JNDI 名は java:comp/resource/ojmsdemo/Queues/demoQueue となります。
  - トピックの場合 : JMS プロバイダ名が ojmsdemo で、トピック名が demoTopic であれば、トピックの JNDI 名は java:comp/resource/ojmsdemo/Topics/demoTopic となります。

例 3-4 に JMS メッセージの送信ステップ、例 3-5 に JMS メッセージの受信ステップを示します。完全な例は、次の URL の OTN Web サイトの OC4J サンプル・コード・ページからこの章で使用している JMS の例をダウンロードしてください。

http://www.oracle.com/technology/tech/java/oc4j/demos/index.html

> **注意 :** 例 3-4 および例 3-5 では、簡潔にするためにほとんどのエラー処理を削除してあります。エラー処理を確認するには、OTN の Web サイトで入手可能なサンプル・コードを参照してください。

例 3-4 OJMS キューにメッセージを送信する OJMS クライアント

次の例で示す dosend メソッドは、メッセージを送信するためのキューを設定します。この例では、キュー・センダーの作成後に複数のメッセージを送信します。キューの設定とメッセージの送信に必要なステップの詳細は、3-5 ページの「メッセージの送信のステップ」を参照してください。

```
public static void dosend(int nmsgs)
{
   // 1a. Retrieve the queue connection factory
```

```
     QueueConnectionFactory qcf = (QueueConnectionFactory)
     ctx.lookup(
       "java:comp/resource/ojmsdemo/QueueConnectionFactories/myQCF");
    // 1b. Retrieve the queue
     Queue q = (Queue)
        ctx.lookup("java:comp/resource/ojmsdemo/Queues/demoQueue");

    // 2. Create the JMS connection
     QueueConnection qc  = qcf.createQueueConnection();
    // 3. Start the queue connection.
     qc.start();
    // 4. Create the JMS session over the JMS connection
     QueueSession    qs  = qc.createQueueSession(false,
                                    Session.AUTO_ACKNOWLEDGE);
     // Create a sender on the JMS session to send messages.
     QueueSender     snd = qs.createSender(q);

     // Send out messages...
     for (int i = 0; i < nmsgs; ++i)
     {
      //Create the message using the createMessage method
     // of the JMS session
      Message msg = qs.createMessage();
      // Send the message out over the sender (snd) using the send method
      snd.send(msg);
      System.out.println("msg:" + " id=" + msg.getJMSMessageID());
     }

    // Close the sender, the JMS session and the JMS connection.
     snd.close();
     qs.close();
     qc.close();
}
```

例 3-5 キューからメッセージを受信する OJMS クライアント

次の例で示す dorcv メソッドは、受信するメッセージを取り出すキューを設定します。キュー・レシーバの作成後に、ループしてキューからすべてのメッセージを受信し、予想されたメッセージの数と比較します。キューの設定とメッセージの受信に必要なステップの詳細は、3-5 ページの「メッセージの送信のステップ」を参照してください。

```
public static void dorcv(int nmsgs)
{
   Context ctx = new InitialContext();

   // 1a. Retrieve the queue connection factory
    QueueConnectionFactory qcf = (QueueConnectionFactory) ctx.lookup(
          "java:comp/resource/ojmsdemo/QueueConnectionFactories/myQCF");
   // 1b. Retrieve the queue
    Queue q = (Queue)
      ctx.lookup("java:comp/resource/ojmsdemo/Queues/demoQueue");

   // 2. Create the JMS connection
    QueueConnection qc  = qcf.createQueueConnection();
   // 3. Start the queue connection.
    qc.start();
   // 4. Create the JMS session over the JMS connection
    QueueSession    qs  = qc.createQueueSession(false,
                                   Session.AUTO_ACKNOWLEDGE);
   // Create a receiver, as we are receiving off of the queue.
    QueueReceiver   rcv = qs.createReceiver(q);

   // Receive the messages
```

```
        int count = 0;
        while (true)
        {
          Message msg = rcv.receiveNoWait();
          System.out.println("msg:" + " id=" + msg.getJMSMessageID());
          ++count;
        }

        if (nmsgs != count)
        {
            System.out.println("expected: " + nmsgs + " found: " + count);
        }

       // Close the receiver, the JMS session and the JMS connection.
        rcv.close();
        qs.close();
        qc.close();
    }
```

Oracle Application Server および Oracle データベースと OJMS の併用

この項では、OJMS（AQ/JMS）を Oracle Application Server と併用する場合に発生する一般的な問題について説明します。

- aqapi.jar をコピーするときのエラー
- OJMS の動作保証マトリックス

aqapi.jar をコピーするときのエラー

OJMS を Oracle Application Server と併用する場合に発生する一般的なエラーは、次のとおりです。

```
PLS-00306 "wrong number or types of arguments"
```

このメッセージが表示される場合は、Oracle Application Server で使用されている `aqapi.jar` ファイルに、AQ に使用されている Oracle データベースのリリースとの互換性がありません。一般的な間違いは、Oracle データベース・インストールと Oracle Application Server インストールに互換性があるという誤った想定のもとに、`aqapi.jar` ファイルを Oracle データベース・インストールと Oracle Application Server インストールの間でコピーすることです。このような混同は、Oracle Application Server と Oracle データベースの両方に OJMS クライアントの JAR ファイルが付属していることが原因で起こります。このファイルをコピーしないでください。表 3-7 のマトリックスを使用して、データベースと Oracle Application Server の正しいリリースを確認し、Oracle Application Server に付属する `aqapi.jar` ファイルを使用してください。

Oracle Application Server インストールでは、OJMS クライアントの JAR ファイルは `ORACLE_HOME/rdbms/jlib/aqapi.jar` にあり、これを `CLASSPATH` に挿入する必要があります。

OJMS の動作保証マトリックス

表 3-7 は、OJMS クライアントが OC4J で実行されている場合に Oracle Application Server と併用する、Oracle データベースのリリースを示しています。○は、そのセルで交差する Oracle データベースのリリースと Oracle Application Server のリリースが、併用した場合に正常に動作することが保証されていることを示します。交差部に○が示されていない場合、対応する Oracle データベースおよび Oracle Application Server のリリースは併用できません。

> **注意：** これは Oracle Application Server と Oracle データベース全般の動作保証マトリックスではありません。あくまでも、Oracle Application Server で OJMS を併用する場合のみを対象としています。

表 3-7 OJMS の動作保証マトリックス

OracleAS/Oracle データベース	リリース 9.0.1	リリース 9.0.1.3	リリース 9.0.1.4	リリース 9.2.0.1	リリース 9.2.0.2 以上	リリース 10.1.0 以上
9.0.2	○	○		○		
9.0.3			○			○
9.0.4			○		○	
9.0.4.1						○
10.1.2			○		○	○

リソース参照内の論理名から JNDI 名へのマッピング

クライアントは、JMS `Destination` オブジェクトを介してメッセージを送受信します。また、クライアントは明示的な名前または論理名を使用して、JMS `Destination` オブジェクトとコネクション・ファクトリを取得できます。3-2 ページの「Oracle Application Server JMS」および 3-24 ページの「Oracle JMS」の例では、JNDI ルックアップのコールに明示的な名前を使用しています。この項では、クライアント・アプリケーションで論理名を使用して、OC4J 固有のデプロイメント・ディスクリプタ内で JMS プロバイダの JNDI 名を制限する方法について説明します。この間接化を使用すると、任意の JMS プロバイダにあわせてクライアント実装を汎用化できます。

クライアント・アプリケーションのコードに論理名を使用する場合は、その論理名を次のいずれかの XML ファイル内で定義します。

- スタンドアロン Java クライアントの場合 : `application-client.xml` ファイル
- クライアントとして機能する EJB の場合 : `ejb-jar.xml` ファイル
- クライアントとして機能する JSP およびサーブレットの場合 : `web.xml` ファイル

論理名を、OC4J デプロイメント・ディスクリプタ内のトピック名またはキュー名の実際の名前にマップします。

論理名の作成

コネクション・ファクトリと `Destination` オブジェクトの論理名は、次のように作成できます。

- コネクション・ファクトリの論理名を `<resource-ref>` 要素にあるクライアントの XML デプロイメント・ディスクリプタに指定します。
  - コネクション・ファクトリの論理名を `<resource-ref>` 要素に指定します。
  - コネクション・ファクトリのクラス・タイプを `<res-type>` 要素に `javax.jms.QueueConnectionFactory` または `javax.jms.TopicConnectionFactory` として指定します。
  - 認証での役割（`Container` または `Bean`）を `<res-auth>` 要素に指定します。
  - 共有範囲（`Shareable` または `Unshareable`）を `<res-sharing-scope>` 要素に指定します。
- JMS `Destination`（トピックまたはキュー）の論理名を `<resource-env-ref>` 要素に指定します。
  - トピックまたはキューの論理名を `<resource-env-ref-name>` 要素に指定します。
  - Destination のクラス・タイプを `<resource-env-ref-type>` 要素に `javax.jms.Queue` または `javax.jms.Topic` として指定します。

例

次の例では、キューの論理名を指定する方法を示します。

```
<resource-ref>
  <res-ref-name>myQCF</res-ref-name>
  <res-type>javax.jms.QueueConnectionFactory</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
<resource-env-ref>
  <resource-env-ref-name>myQueue</resource-env-ref-name>
  <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
</resource-env-ref>
```

論理名から実際の名前へのマッピング

作成した論理名を、OC4J デプロイメント・ディスクリプタ内の実際の名前にマップします。OJMS と OracleAS JMS では、実際の名前（JNDI 名）が異なります。ただし、マッピングは次のいずれかのファイルで定義します。

- スタンドアロン Java クライアントの場合 : `orion-application-client.xml` ファイルに定義します。
- クライアントとして機能する EJB の場合 : `orion-ejb-jar.xml` ファイルに定義します。
- クライアントとして機能する JSP およびサーブレットの場合 : `orion-web.xml` ファイルに定義します。

クライアントのデプロイメント・ディスクリプタ内の論理名は、次のようにマップされます。

- `<resource-ref>` 要素で定義されているコネクション・ファクトリの論理名は、`<resource-ref-mapping>` 要素内の JNDI 名にマップされます。
- `<resource-env-ref>` 要素で定義されている JMS `Destination` の論理名は、`<resource-env-ref-mapping>` 要素内の JNDI 名にマップされます。

これ以降は、OracleAS JMS と OJMS でのマッピングと、この命名規則のクライアントでの使用方法について説明します。

- OracleAS JMS に対する JNDI ネーミング
- OJMS に対する JNDI ネーミング
- Java アプリケーション・クライアントに対する JNDI ネーミング・プロパティの設定
- 論理名を使用したクライアントからの JMS メッセージ送信

OracleAS JMS に対する JNDI ネーミング

OracleAS JMS の `Destination` およびコネクション・ファクトリの JNDI 名は、`jms.xml` ファイル内で定義されます。例 3-1 に示すように、キューとキュー・コネクション・ファクトリの JNDI 名は次のとおりです。

- トピックの JNDI 名は `jms/demoQueue` です。
- トピック・コネクション・ファクトリの JNDI 名は `jms/QueueConnectionFactory` です。

この 2 つの名前のどちらにも `java:comp/env/` が先頭に付加されます。次の例は、`orion-ejb-jar.xml` ファイル内のマッピングを示しています。

```
<resource-ref-mapping
    name="myQCF"
   location="java:comp/env/jms/QueueConnectionFactory">
</resource-ref-mapping>

<resource-env-ref-mapping
    name="myQueue"
```

```
    location="java:comp/env/jms/demoQueue">
</resource-env-ref-mapping>
```

OJMS に対する JNDI ネーミング

OJMS の Destination オブジェクトおよびコネクション・ファクトリ・オブジェクトに対する JNDI ネーミングは、orion-ejb-jar.xml ファイルで指定されている名前と同じです。3-29 ページの「OJMS リソースへのアクセス」を参照してください。

次の例では、コネクション・ファクトリとキューの論理名を実際の JNDI 名にマップしています。特に、application-client.xml ファイル内で myQueue として論理的に定義されたキューは、JNDI 名である java:comp/resource/ojmsdemo/Queues/demoQueue にマップされます。

```
<resource-ref-mapping
  name="myQCF"
  location="java:comp/resource/ojmsdemo/QueueConnectionFactories/myQF">
</resource-ref-mapping>

<resource-env-ref-mapping
  name="myQueue"
  location="java:comp/resource/ojmsdemo/Queues/demoQueue">
</resource-env-ref-mapping>
```

Java アプリケーション・クライアントに対する JNDI ネーミング・プロパティの設定

Oracle Application Server では、Java アプリケーション・クライアントは JNDI プロパティに次のように指定して JMS Destination オブジェクトにアクセスします。

```
java.naming.factory.initial=
   com.evermind.server.ApplicationClientInitialContextFactory
java.naming.provider.url=opmn:ormi://$HOST:$OPMN_REQUEST_PORT:$OC4J_INSTANCE/
java.naming.security.principal=admin
java.naming.security.credentials=welcome
```

次の操作が必要です。

- ApplicationClientInitialContextFactory を初期コンテキストのファクトリ・オブジェクトとして使用します。
- プロバイダ URL で OPMN のホストとポートおよび OC4J インスタンスを指定します。

OC4J スタンドアロン環境では、Java アプリケーション・クライアントは JNDI プロパティに次のように指定して JMS Destination オブジェクトにアクセスします。

```
java.naming.factory.initial=
   com.evermind.server.ApplicationClientInitialContextFactory
java.naming.provider.url=ormi://myhost/
java.naming.security.principal=admin
java.naming.security.credentials=welcome
```

次の操作が必要です。

- ApplicationClientInitialContextFactory を初期コンテキストのファクトリ・オブジェクトとして使用します。
- プロバイダ URL でスタンドアロン OC4J のホストとポートを指定します。

論理名を使用したクライアントからの JMS メッセージ送信

リソースを定義して JNDI プロパティを構成した後、クライアントでは、次の手順に従って JMS メッセージを送信する必要があります。この手順は、例 3-6 に詳しく示した手順をまとめたものです。

1. JNDI ルックアップを使用して、構成済の JMS `Destination` とそのコネクション・ファクトリを取得します。
2. コネクション・ファクトリから接続を作成します。キューについてメッセージを受信する場合は、接続が開始されます。
3. 接続を使用してセッションを作成します。
4. 取得した JMS `Destination` を提供し、キューの場合はセンダー、トピックの場合はパブリッシャを作成します。
5. メッセージを作成します。
6. キュー・センダーまたはトピック・パブリッシャを使用してメッセージを送信します。
7. キュー・セッションをクローズします。
8. JMS `Destination` タイプに応じて接続をクローズします。

例 3-6 JSP クライアントからトピックへのメッセージ送信

メッセージをトピックに送信する方法は、メッセージをキューに送信する方法とほぼ同じです。キューを作成するかわりにトピックを作成します。センダーを作成するかわりにパブリッシャを作成します。

次の JSP クライアント・コードは、メッセージをトピックに送信します。このコードでは、OC4J デプロイメント・ディスクリプタにマップされている論理名を使用しています。例の番号付きコメントは、この項の最初の部分に示した手順に対応しています。

```
<%@ page import="javax.jms.*, javax.naming.*, java.util.*" %>
<%

//1a. Look up the topic.
jndiContext = new InitialContext();
topic = (Topic)jndiContext.lookup("demoTopic");

//1b. Look up the Connection factory.
topicConnectionFactory = (TopicConnectionFactory)
   jndiContext.lookup("myTCF");

//2 & 3. Retrieve a connection and a session on top of the connection.
topicConnection = topicConnectionFactory.createTopicConnection();
topicSession = topicConnection.createTopicSession(true,
                                     Session.AUTO_ACKNOWLEDGE);

//4. Create the publisher for any messages destined for the topic.
topicPublisher = topicSession.createPublisher(topic);

//5 & 6. Create and send out the message.
for (int  ii = 0; ii < numMsgs; ii++)
{
  message = topicSession.createBytesMessage();
  String  sndstr = "1::This is message " + (ii + 1) + " " + item;
  byte[]  msgdata = sndstr.getBytes();
  message.writeBytes(msgdata);

  topicPublisher.publish(message);
  System.out.println("--->Sent message: " + sndstr);
}

//7 & 8. Close publisher, session, and connection for topic.
```

```
topicPublisher.close();
topicSession.close();
topicConnection.close();
%>
```

サード・パーティの JMS プロバイダ

この項では、次のサード・パーティの JMS プロバイダと、リソース・プロバイダ・インタフェースを使用して各プロバイダを OC4J と統合する方法について説明します。

- WebSphere MQ をリソース・プロバイダとして使用する方法
- SonicMQ をリソース・プロバイダとして使用する方法
- SwiftMQ をリソース・プロバイダとして使用する方法

次に、リソース・プロバイダ・インタフェースがサポートする操作を示します。

- 次の指定を使用したキューとトピックのルックアップ

  java:comp/resource/*providerName*/*resourceName*

- EJB でのメッセージの送信
- EJB でのメッセージの同期受信

注意： Oracle がサポートするのは、OJMS 以外のリソース・プロバイダに対する 1 フェーズ・コミットのセマンティックのみです。

コンテキスト・スキャン・リソース・プロバイダ・クラスは、汎用のリソース・プロバイダ・クラスです。このクラスは、サード・パーティのメッセージ・プロバイダで使用するために OCJ に付属しています。

WebSphere MQ をリソース・プロバイダとして使用する方法

WebSphere MQ は、IBM 社のメッセージ・プロバイダです。この例では、WebSphere MQ を JMS コネクションのデフォルトのリソース・プロバイダにする方法について説明します。WebSphere MQ リソースは、OC4J では `java:comp/resource/MQSeries/` の下で使用可能です。

WebSphere MQ の構成

WebSphere MQ を構成する手順は、次のとおりです。

1. システムに WebSphere MQ をインストールして構成します。次に、ベンダーから提供されている例またはツールを実行してインストールが成功していることを確認します。（これらの方法については、ソフトウェアに付属のドキュメントを参照してください。）
2. リソース・プロバイダを構成します。リソース・プロバイダを構成するには、Oracle Enterprise Manager 10*g* を使用する方法（3-26 ページの「OJMS リソース・プロバイダの定義」を参照）と、`orion-application.xml` の `<resource-provider>` 要素を使用する方法の 2 つがあります。どちらかの方法を使用し、WebSphere MQ をカスタム・リソース・プロバイダとして追加します。次の例では、`<resource-provider>` 要素を使用した WebSphere MQ の構成方法を示します。Oracle Enterprise Manager 10*g* を使用して構成する場合も、これと同じ情報を使用できます。

```
<resource-provider
    class="com.evermind.server.deployment.ContextScanningResourceProvider"
    name="MQSeries">
  <description> MQSeries resource provider </description>
  <property
      name="java.naming.factory.initial"
      value="com.sun.jndi.fscontext.RefFSContextFactory">
```

```
  </property>
  <property
      name="java.naming.provider.url"
      value="file:/var/mqm/JNDI-Directory">
  </property>
</resource-provider>
```

3. 次の WebSphere MQ JMS クライアント JAR ファイルを、*J2EE_HOME*/lib に追加します。

```
com.ibm.mq.jar
com.ibm.mqbind.jar
com.ibm.mqjms.jar
mqji.properties
```

4. ファイル・システム JNDI の JAR ファイル fscontext.jar および providerutil.jar を *J2EE_HOME*/lib に追加します。

SonicMQ をリソース・プロバイダとして使用する方法

SonicMQ は、Sonic Software 社のメッセージ・プロバイダです。リソース・プロバイダ・インタフェースは、サード・パーティの JMS 実装をプラグインするためのサポートを提供します。この例では、SonicMQ を JMS コネクションのデフォルトのリソース・プロバイダにする方法について説明します。SonicMQ リソースは、OC4J では java:comp/resource/SonicMQ の下で使用可能です。

注意： SonicMQ ブローカには、JNDI サービスが埋め込まれていません。かわりに、外部のディレクトリ・サーバーを使用して管理オブジェクトを登録します。キューなどの管理オブジェクトは、管理者が（SonicMQ Explorer を使用して）作成するか、（Sonic Management API を使用して）プログラムで作成します。Oracle は、ファイル・システム JNDI を使用して、SonicMQ Explorer から管理オブジェクトを登録します。

SonicMQ の構成

SonicMQ を構成する手順は、次のとおりです。

1. システムに SonicMQ をインストールして構成します。次に、ベンダーから提供されている例またはツールを実行してインストールが成功していることを確認します。（これらの方法については、ソフトウェアに付属のドキュメントを参照してください。）

2. リソース・プロバイダを構成します。リソース・プロバイダを構成するには、Oracle Enterprise Manager 10*g* を使用する方法（3-26 ページの「OJMS リソース・プロバイダの定義」を参照）と、orion-application.xml の <resource-provider> 要素を使用する方法の 2 つがあります。どちらかの方法を使用し、SonicMQ をカスタム・リソース・プロバイダとして追加します。次の例では、<resource-provider> 要素を使用した SonicMQ の構成方法を示します。Oracle Enterprise Manager 10*g* を使用して構成する場合も、これと同じ情報を使用できます。

```
<resource-provider
  class="com.evermind.server.deloyment.ContextScanningResourceProvider"
  name="SonicJMS">
   <description>
      SonicJMS resource provider.
   </description>
   <property name="java.naming.factory.initial"
         value="com.sun.jndi.fscontext.RefFSContextFactory">
   <property name="java.naming.provider.url"
         value="file:/private/jndi-directory/">
</resource-provider>
```

3. 次の SonicMQ JMS クライアント JAR ファイルを、*J2EE_HOME*/lib に追加します。

```
Sonic_client.jar
Sonic_XA.jar
```

SwiftMQ をリソース・プロバイダとして使用する方法

SwiftMQ は、IIT Software 社のメッセージ・プロバイダです。この例では、SwiftMQ を JMS コネクションのデフォルトのリソース・プロバイダにする方法について説明します。SwiftMQ リソースは、OC4J では `java:comp/resource/SwiftMQ` の下で使用可能です。

SwiftMQ の構成

SwiftMQ を構成する手順は、次のとおりです。

1. システムに SwiftMQ をインストールして構成します。次に、ベンダーから提供されている例またはツールを実行してインストールが成功していることを確認します。（これらの方法については、ソフトウェアに付属のドキュメントを参照してください。）

2. リソース・プロバイダを構成します。リソース・プロバイダを構成するには、Oracle Enterprise Manager 10*g* を使用する方法（3-26 ページの「OJMS リソース・プロバイダの定義」を参照）と、orion-application.xml の <resource-provider> 要素を使用する方法の 2 つがあります。どちらかの方法を使用し、SwiftMQ をカスタム・リソース・プロバイダとして追加します。次の例では、<resource-provider> 要素を使用した SwiftMQ の構成方法を示します。Oracle Enterprise Manager 10*g* を使用して構成する場合も、これと同じ情報を使用できます。

```
<resource-provider
  class="com.evermind.server.deloyment.ContextScanningResourceProvider"
  name="SwiftMQ">
   <description>
      SwiftMQ resource provider.
   </description>
   <property name="java.naming.factory.initial"
      value="com.swiftmq.jndi.InitialContextFactoryImpl">
   <property name="java.naming.provider.url"
      value="smqp://localhost:4001">
</resource-provider>
```

3. 次の SwiftMQ JMS クライアント JAR ファイルを、*J2EE_HOME*/lib に追加します。

```
swiftmq.jar
```

Message-Driven Bean の使用

OracleAS JMS または OJMS にアクセスする MDB のデプロイの詳細は、『Oracle Application Server Containers for J2EE Enterprise JavaBeans 開発者ガイド』の「Message-Driven Bean」を参照してください。

注意： orion-ejb-jar.xml ファイルの transaction-timeout 属性に定義されている Message-Driven Bean（MDB）トランザクション・タイムアウトは、オプションのパラメータです。この属性は、Oracle JMS を使用するコンテナ管理のトランザクションの MDB に対するトランザクション・タイムアウト間隔（秒数）を制御します。デフォルトは 1 日（86,400 秒）です。MDB の transaction-timeout 属性は、Oracle JMS を JMS プロバイダとして使用する CMT MDB にのみ適用されます。この属性の設定は、BMT MDB または OC4J JMS を使用する MDB には影響を与えません。（バグ 3079322）

- Oracle Application Server での JMS の動作 : この時間内に完了しないトランザクションはロールバックされます。メッセージは Destination オブジェクトに再配信されます。Oracle JMS によるメッセージの再配信の試行が終わると（デフォルトの試行回数は 5）、メッセージは例外キューに移動します。詳細は、『Oracle Streams アドバンスト・キューイング・ユーザーズ・ガイドおよびリファレンス』を参照してください。
- OC4J での JMS の動作 : transaction-timeout 設定は OC4J JMS を使用する CMT MDB に対しては機能しません。タイムアウトは常に 1 日で、変更できません。タイムアウトが発生すると、OC4J JMS は、配信が成功するまでメッセージをいつまでも再配信します。再試行の制限は設定できません。

また、server.xml ファイルに定義されているグローバルの transaction-timeout 属性は、MDB には一切影響を与えません。

JMS の高可用性とクラスタリング

可用性の高い JMS サーバーは、JMS 要求がソフトウェアまたはハードウェア障害による中断なしで処理されるという保証を提供します。高可用性を実現する方法の 1 つはフェイルオーバー機能を使用することです。サーバー・インスタンスの 1 つに障害が発生すると、ソフトウェア、ハードウェアおよびインフラストラクチャ・メカニズムの組合せにより、別のサーバー・インスタンスが要求の処理を引き継ぎます。

表 3-8 に、OracleAS JMS と OJMS の高可用性のサポートを示します。

表 3-8 高可用性の概要

機能	OJMS	OracleAS JMS
高可用性	RAC + OPMN	OPMN
構成	RAC 構成、リソース・プロバイダ構成	専用 JMS サーバー、`jms.xml` 構成、opmn.xml 構成
メッセージ・ストア	RAC データベース上	専用 JMS サーバー / 永続性ファイル内
フェイルオーバー	同一または異なるマシン（RAC による）	Oracle Application Server Cold Failover Cluster（中間層）内の同一または異なるマシン

JMS クラスタリングにより提供される環境では、この種の環境にデプロイされた JMS アプリケーションが、複数の OC4J インスタンスまたはプロセスにまたがって JMS 要求のロード・バランシングを実行できます。ステートレス・アプリケーションのクラスタリングは通常のことで、アプリケーションは複数のサーバーにデプロイされ、ユーザー要求はその 1 つにルーティングされます。

JMS はステートフル API です。JMS クライアントと JMS サーバーの両方に、接続、セッションおよび永続サブスクリプションに関する情報など、相互に関する状態が含まれます。ユーザーは、その環境を構成し、クラスタ対応アプリケーションを記述するときに少数の単純なテクニックを使用できます。

ここでは、OJMS と OracleAS JMS で高可用性とクラスタリングを使用する方法について説明します。

- OracleAS JMS の高可用性構成
- OJMS の高可用性構成
- RAC データベースを OJMS と併用する場合のフェイルオーバー
- 両方の JMS プロバイダに対するフェイルオーバーのサーバー・サイドのサンプル・コード
- クラスタリングのベスト・プラクティス

OracleAS JMS の高可用性構成

OracleAS JMS のクラスタリングは、通常、この種の環境にデプロイされたアプリケーションにより、OC4J の複数インスタンスにまたがってメッセージのロード・バランシングを実行できることを意味します。また、この環境では、コンテナ・プロセスを複数のノード間で分散できるため、ある程度の高可用性が実現します。いずれかのプロセスまたはノードが停止した場合は、代替ノード上のプロセスがメッセージの処理を続行します。

この項では、JMS クラスタリングの 2 つの使用例について説明します。

- OracleAS JMS サーバーの分散先

  この構成では、すべてのファクトリと宛先がすべての OC4J インスタンス上で定義されます。各 OC4J インスタンスには、それぞれの宛先の個別コピーがあります。宛先のコピーはそれぞれ一意で、OC4J インスタンス間ではレプリケートまたは同期化されません。宛先は、永続的でもメモリー内でもかまいません。1 つの OC4J インスタンスにエンキューされたメッセージは、その OC4J インスタンスからでなければデキューできません。

  この構成は、OC4J インスタンス間で要求のロード・バランシングを必要とする高スループットのアプリケーションに最適です。この使用例の場合、構成変更は不要です。

- Cold Failover Cluster

  この構成は 2 ノード・クラスタです。常にアクティブなノードは 1 つのみです。2 つ目のノードは、1 つ目のノードに障害が発生した場合にアクティブになります。

- OracleAS の専用 JMS サーバー

  この構成では、1 つの OC4J インスタンス内の 1 つの JVM が専用 JMS サーバーとなります。JMS クライアントをホスティングする他のすべての OC4J インスタンスは、JMS 要求を専用 JMS サーバーに転送します。

  この構成はメンテナンスとトラブルシューティングが最も容易であり、大多数の OracleAS JMS アプリケーション、特にメッセージの順序付けを必要とする場合に適しています。

用語

ここで紹介する用語の詳細は、『Oracle Application Server 高可用性ガイド』および『Oracle Process Manager and Notification Server 管理者ガイド』を参照してください。

- OHS: Oracle HTTP Server。
- OracleAS クラスタ : 類似する構成を持つ Oracle Application Server インスタンスのグループ。
- Oracle Application Server インスタンス : Oracle Application Server のインストール環境（つまり *ORACLE_HOME*）。
- OC4J インスタンス : Oracle Application Server インスタンス内に複数の OC4J インスタンスが存在でき、各 OC4J インスタンスには同一構成を持つ 1 ～ n 個の JVM があります。

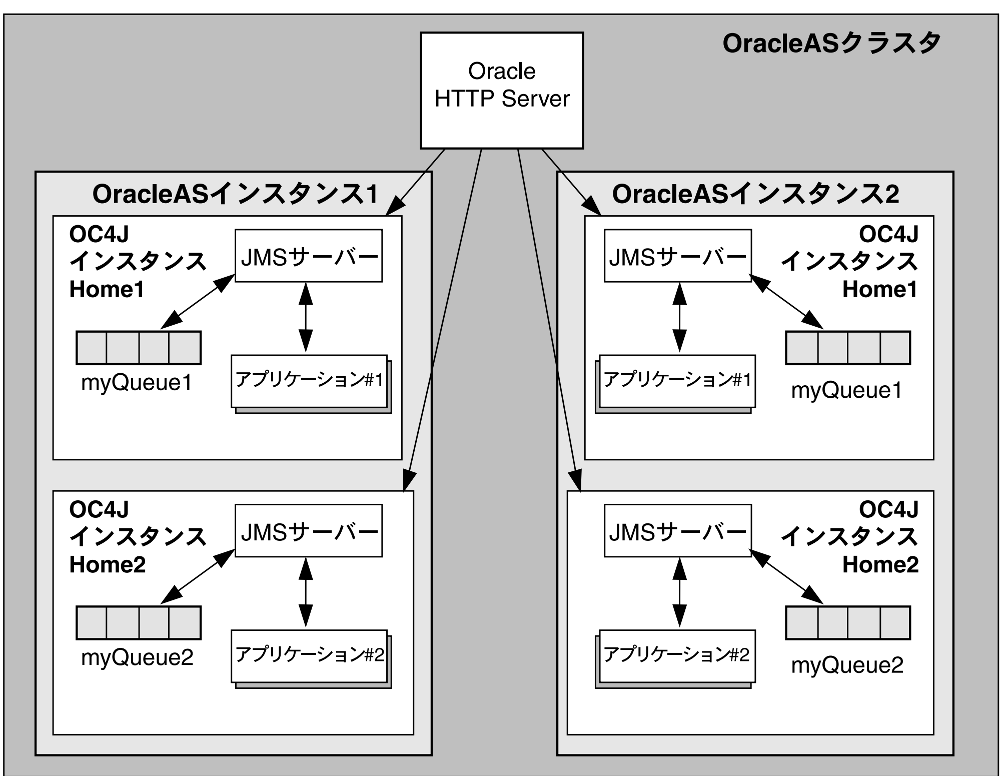
- ファクトリ : JMS コネクション・ファクトリ。
- 宛先 : JMS 宛先。

OracleAS JMS サーバーの分散先

この構成では、OHS は HTTP リクエストを処理し、Oracle Application Server クラスタ内の 2 つの Oracle Application Server インスタンス間でリクエストのロード・バランシングを実行します。この構成は、3 つ以上の Oracle Application Server インスタンスへと拡張できます。この種のデプロイメントには、次のような多数のメリットがあります。

- アプリケーションと JMS サーバーの両方が同じ JVM 内で実行され、プロセス間通信が不要であるため、高スループットが実現します。
- ロード・バランシングにより、高スループットと高可用性が促進されます。
- シングル・ポイント障害がなく、OC4J プロセスが 1 つでも使用可能であれば、リクエストを処理できます。
- JMS 操作に影響を与えずに Oracle Application Server インスタンスをクラスタリングできます。
- Destination オブジェクトは、永続的でもメモリー内でもかまいません。

図 3-2 OracleAS JMS サーバーの分散先



各 Oracle Application Server インスタンス内に、2 つの OC4J インスタンスが定義されています。これらの OC4J インスタンスでは、それぞれ別のアプリケーションが実行されます。つまり、OC4J インスタンス #1（`Home1`）はアプリケーション #1 を実行し、OC4J インスタンス #2（`Home2`）はアプリケーション #2 を実行します。各 OC4J インスタンスで複数の JVM を実行するように構成し、この複数の JVM 間でアプリケーションを拡張できることに注意してください。

Oracle Application Server クラスタ内では、各 Oracle Application Server インスタンスの構成情報は（ホスト名やポート番号など、インスタンス固有の情報を除いて）同一です。つまり、Oracle Application Server インスタンス #1 の OC4J インスタンス #1 にデプロイされたアプリケーション #1 は、Oracle Application Server インスタンス #2 の OC4J インスタンス #1 にもデプロイされます。このタイプのアーキテクチャでは、複数の Oracle Application Server インスタンス間でメッセージのロード・バランシングを実行できます。また、特にハードウェア障害から保護するために Oracle Application Server インスタンス #2 が別のノードにデプロイされている場合には、JMS アプリケーションの高可用性が実現します。

各アプリケーションのセンダーとレシーバは、1 つの OC4J インスタンスにともにデプロイする必要があります。つまり、ある OC4J プロセスで JMS サーバーにエンキューされたメッセージは、その OC4J プロセスからでなければデキューできません。

すべてのファクトリと宛先は、すべての OC4J プロセス上で定義されます。各 OC4J プロセスには、それぞれの宛先の個別コピーがあります。宛先のコピーは、レプリケートまたは同期化されません。そのため、この図では、アプリケーション #1 は宛先 `myQueue1` に書き込んでいます。この宛先は 2 つのロケーション（Oracle Application Server インスタンス #1 および #2）に物理的に存在し、各 OC4J インスタンス内でそれぞれの JMS サーバーにより管理されます。

この種の JMS デプロイメントは、特定のタイプの JMS アプリケーションにのみ適していることに注意してください。メッセージの順序が重要でない場合、メッセージは同じ名前を持つ分散キューにエンキューされます。JMS の Point-to-Point メッセージングのセマンティクスでは、メッセージは複数のキュー間で複製できません。前述の例では、メッセージはロード・バランシング・アルゴリズムにより決定されたキューに送信され、着信時に MDB によりデキューされます。

Cold Failover Cluster

この構成は 2 ノード・クラスタです。常にアクティブなノードは 1 つのみです。2 つ目のノードは、1 つ目のノードに障害が発生した場合にアクティブになります。Cold Failover については、『Oracle Application Server 高可用性ガイド』を参照してください。

構成

次の例に示すように、両方のノードを同じ構成にします。両方の OC4J インスタンスの `jms.xml` を変更します。`jms-server` の host パラメータを次のように設定します。

```
<jms-server host=vmt.us.oracle.com port="9127">
….
….
</jms-server>
```

ファイル・ベースのメッセージの永続性をキューに使用する場合、両方のノードからアクセスできる共有ディスク上にファイルを置く必要があります。共有ディスクは、一方のノードから他方のノードにフェイルオーバーする際に、仮想 IP を使用する必要があります。次のように `persistence-file` を構成します。

```
<queue name="Demo Queue" location="jms/demoQueue"
persistence-file="/path/to/shared_file_system/demoQueueFile">
        <description>A dummy queue</description>
</queue>
```

更新、停止および起動

各ノードでは、次のコマンドを使用して、構成の更新、停止および起動を行います。

```
$ORACLE_HOME/dcm/bin/dcmctl updateConfig -ct  oc4j
$ORACLE_HOME/opmn/bin/opmnctl stopall
$ORACLE_HOME/opmn/bin/opmnctl startall
```

OracleAS の専用 JMS サーバー

この構成では、Oracle Application Server クラスタ環境で 1 つの OC4J インスタンスが専用 JMS サーバーとして構成されます。この OC4J インスタンスはすべてのメッセージを処理するため、常にメッセージの順序付けが維持されます。すべての JMS アプリケーションは、この専用サーバーを使用してコネクション・ファクトリと宛先をホスティングし、エンキューおよびデキューのリクエストを処理します。

専用 JMS プロバイダとして機能する OC4J JVM は、クラスタ内のすべての JMS アプリケーションに対して 1 つのみです。そのために、`opmn.xml` ファイル内の JMS ポート範囲は専用 OC4J インスタンス用の 1 つのポートのみに制限されます。

この図は OC4J の `Home` インスタンス内のアクティブな JMS サーバーを示していますが、JMS プロバイダは専用 OC4J インスタンス内でホスティングすることをお薦めします。たとえば、`Home` は Oracle Application Server のインストール後に実行されるデフォルトの OC4J インスタンスですが、Oracle Enterprise Manager 10*g* を使用して 2 つ目の OC4J インスタンスを作成する必要があります。次の `opmn.xml` ファイルの例では、OC4J インスタンス `JMSserver` が作成されます。

図 3-3 OracleAS の専用 JMS サーバー

OracleASクラスタ
Oracle HTTP Server
OracleASインスタンス1
OC4J インスタンス Home
JMSサーバー
App1 キュー
App2 キュー
アプリケーション#1
アプリケーション#2
OracleASインスタンス2
JMSサーバー（使用不可）
OC4J インスタンス Home
アプリケーション#1
アプリケーション#2

OC4J インスタンス `JMSserver` を作成した後、この Oracle Application Server インスタンスについて `opmn.xml` ファイルに次の 2 つの変更を加える必要があります。

1. この OC4J インスタンス（`JMSserver`）用に起動される JVM が 1 つのみであることを確認します。
2. このインスタンスの JMS ポート範囲の値を 1 つのみ指定します。

OC4J インスタンス内の JVM を 1 つに限定することで、他の JVM が同じ永続性ファイル・セットを使用しないことが保証されます。

ポート値を 1 つにすると、OPMN では、常にこの値が専用 JMS サーバーに割り当てられます。このポート値を使用して、`jms.xml` ファイル内でコネクション・ファクトリを定義できます。他の OC4J インスタンスは、これを専用 JMS サーバーへの接続に使用します。

OPMN と動的ポート割当ての詳細は、『Oracle Process Manager and Notification Server 管理者ガイド』を参照してください。

OPMN 構成の変更

注意： 構成ファイルを手動で（つまり Oracle Enterprise Manager 10*g* を使用せずに）編集する場合は、次の Distributed Configuration Management（DCM）コマンドを実行します。

```
dcmctl updateConfig
```

詳細は、『Distributed Configuration Management 管理者ガイド』を参照してください。

次の XML は opmn.xml ファイルから抜粋したもので、必要な変更内容と、変更する箇所を見つける方法を示しています。

- Oracle Enterprise Manager 10*g* を使用して OC4J インスタンス JMSserver が作成されており、(1) で示されている行は JMSserver 定義の開始位置を示しているとします。
- (2) で示されている行は、OC4J JVM に JMS ポートを割り当てるときに OPMN で使用される JMS ポート範囲です。JMS プロバイダとして機能させる必要のある専用 OC4J インスタンスについて、この範囲を 1 つの値に限定します。この例では、元の範囲は 3701 ～ 3800 です。コネクション・ファクトリ定義では、この値を 3701 ～ 3701 として構成し、使用ポートを確認します。
- (3) で示されている行では、JMSserver のデフォルト・アイランドに含まれる JVM の数を定義します。デフォルトでは、この値は 1 に設定されます。この値は常に 1 にする必要があります。

```
<ias-component id="OC4J">
  (1) <process-type id="JMSserver" module-id="OC4J" status="enabled">
     <module-data>
       <category id="start-parameters">
         <data id="java-options" value="-server
           -Djava.security.policy=$ORACLE_HOME/j2ee/home/config/java2.policy
           -Djava.awt.headless=true
         "/>
       </category>
       <category id="stop-parameters">
         <data id="java-options"
           value="-Djava.security.policy=
             $ORACLE_HOME/j2ee/home/config/java2.policy
           -Djava.awt.headless=true"/>
       </category>
    </module-data>
    <start timeout="600" retry="2"/>
    <stop timeout="120"/>
    <restart timeout="720" retry="2"/>
    <port id="ajp" range="3000-3100"/>
    <port id="rmi" range="3201-3300"/>
    (2) <port id="jms" range="3701-3701"/>
    (3) <process-set id="default_island" numprocs="1"/>
  </process-type>
</ias-component>
```

OracleAS JMS の構成

この使用例で前述したように、OC4J インスタンスの 1 つは JMS サーバー専用です。他の OC4J インスタンスおよび OC4J 外部で実行されるスタンドアロン JMS クライアントは、JMS 要求を専用 JMS サーバーに転送するように設定する必要があります。すべてのコネクション・ファクトリと宛先は、JMS サーバー・インスタンスの jms.xml ファイル内で定義されます。この jms.xml ファイルを、JMS サーバーと通信する他のすべての OC4J インスタンスにコピーする必要があります。

専用 JMS サーバー上の jms.xml ファイル内で構成するコネクション・ファクトリでは、サーバーのホスト名とポート番号を明示的に指定する必要があります。これらの値（特にポート番号）には、前述のとおり専用サーバー用に OPMN で定義された 1 つのポート番号を使用する必要があります。これと同じコネクション・ファクトリの構成を他のすべての OC4J インスタンスにも使用します。これにより、OC4J インスタンスすべてがその操作について専用 JMS サーバーを指すことになります。

そのため、専用 JMS サーバーが host1、ポート 3701 上で実行される場合、クラスタ内の各 OC4J インスタンス用の jms.xml ファイル内で定義されたコネクション・ファクトリはすべて、host1、ポート 3701 を指す必要があります。このポートは、専用 OC4J インスタンス（この例では JMSserver）内で専用 JMS サーバー用に opmn.xml ファイル内で使用可能な単一ポートです。

専用 JMS サーバー上の jms.xml ファイル内で構成されている宛先は、他のすべての OC4J インスタンス上でも構成する必要があります。ただし、これらの宛先の物理ストアは専用 JMS サーバー上にあります。

キュー・コネクション・ファクトリ定義の例

専用 OracleAS JMS サーバーの jms.xml ファイル内でキュー・コネクション・ファクトリを定義する例を次に示します。

```
<!-- Queue connection factory -->
<queue-connection-factory name="jms/MyQueueConnectionFactory"
          host="host1" port="3701"
          location="jms/MyQueueConnectionFactory"/>
```

専用 JMS サーバーの jms.xml ファイルには、管理上の変更（つまり、新規 Destination オブジェクトの追加）を行う必要があります。次に、JMS アプリケーションを実行する他のすべての OC4J インスタンスの jms.xml ファイル内で、同じ変更を実行します。この変更には、手動で実行する方法と、専用 JMS サーバーの jms.xml ファイルを他の OC4J インスタンスにコピーする方法があります。

アプリケーションのデプロイ

JMS アプリケーションが実際にデプロイされる場所は、ユーザーが決定します。専用 JMS サーバーは、JMS 要求を処理する一方で、デプロイされた JMS アプリケーションも実行できます。また、JMS アプリケーションは他の OC4J インスタンス（つまり Home）にもデプロイできます。

専用 JMS サーバーの jms.xml ファイルは、JMS アプリケーションがデプロイされる OC4J インスタンスすべてに伝播させる必要があることに注意してください。JMS アプリケーションは、別の JVM で実行中のスタンドアロン JMS クライアントにデプロイすることもできます。

高可用性

OPMN には、専用 JMS サーバーの稼働を維持するために、フェイルオーバー・メカニズムが用意されています。なんらかの理由で JMS サーバーに障害が発生すると、OPMN はそれを検出して JVM を再起動します。ハードウェア障害が発生した場合、メッセージをリカバリする唯一の方法は、永続する宛先がネットワーク・ファイル・システム上でホスティングされるようにすることです。OC4J インスタンスを起動し、これらの永続するファイルを指すように構成できます。

OPMN による Oracle Application Server プロセスの管理の詳細は、『Oracle Process Manager and Notification Server 管理者ガイド』を参照してください。

OJMS の高可用性構成

OJMS で高可用性を実現するには、次のようにします。

- AQ キューおよびトピックを含む Oracle データベースを RAC モードで実行します。これにより、データベースの高可用性が保証されます。
- Oracle Application Server を OPMN モードで実行します。これにより、アプリケーション・サーバー（およびそこにデプロイされたアプリケーション）の高可用性が保証されます。

Oracle Application Server クラスタ内の各アプリケーション・インスタンスは、OC4J リソース・プロバイダを使用して、RAC モードで稼働しているバックエンド Oracle データベースを指します。これらのリソース・プロバイダから導出されたオブジェクトで起動される JMS 操作は、Real Application Clusters（RAC）データベースに送られます。

アプリケーション障害が発生すると、そのアプリケーション内の状態情報は失われます（つまり、接続、セッションおよびメッセージの状態はコミットされていません）。アプリケーション・サーバーが再起動されると、アプリケーションは JMS 状態を適切に再作成して操作を再開する必要があります。

バックエンド・データベース（非 RAC データベース）のネットワーク・フェイルオーバーが発生すると、サーバー内の状態情報は失われます（つまり、トランザクションの状態はコミットされていません）。また、アプリケーション内の JMS オブジェクト（コネクション・ファクトリ、`Destination` オブジェクト、接続、セッションなど）も無効になることがあります。データベース障害の発生後にこれらのオブジェクトを使用する場合は、アプリケーション・コードで例外を示すことができます。JNDI を介してすべての JMS 管理オブジェクトをルックアップできるポイントに達するまで、コードは `JMSException` をスローし、そのポイントから続行する必要があります。

RAC データベースを OJMS と併用する場合のフェイルオーバー

RAC データベースを使用するアプリケーションでは、データベース・フェイルオーバーを使用する必要があります。第 4 章「データ・ソース」で示すように、フェイルオーバーには 2 つのタイプがあります。次の各項では、各フェイルオーバーの使用方法について説明します。

- JMS と RAC ネットワーク・フェイルオーバーの併用
- OJMS と透過的アプリケーション・フェイルオーバー（TAF）の併用

注意： データ・ソースの `RAC-enabled` 属性については、第 4 章「データ・ソース」を参照してください。このフラグをインフラストラクチャ・データベースとともに使用する方法の詳細は、『Oracle Application Server 高可用性ガイド』を参照してください。

JMS と RAC ネットワーク・フェイルオーバーの併用

RAC データベースに対して実行するスタンドアロン OJMS クライアントでは、API `com.evermind.sql.DbUtil.oracleFatalError()` を起動して接続オブジェクトが無効かどうかを判断し、接続を再び取得するためのコードを記述する必要があります。その後、必要に応じてデータベース接続を再確立します。`oracleFatalError()` メソッドは、ネットワーク・フェイルオーバー中にデータベースによりスローされた SQL エラーが致命的エラーかどうかを検出します。このメソッドは SQL エラーとデータベース接続を使用して、致命的エラーの場合は `true` を戻します。`true` の場合は、トランザクションを積極的にロールバックし、JMS の状態（失われた接続、セッションおよびメッセージなど）を再作成する必要があります。

次の例にこのロジックの概略を示します。

```
getMessage(QueueSesssion session)
{
    try
    {
        QueueReceiver rcvr;
         Message msgRec = null;
        QueueReceiver rcvr = session.createReceiver(rcvrQueue);
         msgRec = rcvr.receive();
    }
    catch(Exception e )
    {
        if (exc instanceof JMSException)
        {
           JMSException  jmsexc = (JMSException) exc;
           sql_ex = (SQLException)(jmsexc.getLinkedException());

           db_conn =
             (oracle.jms.AQjmsSession)session.getDBConnection();

           if ((DbUtil.oracleFatalError(sql_ex, db_conn))
           {
               // failover logic
            }
          }
     }
}
```

OJMS と透過的アプリケーション・フェイルオーバー（TAF）の併用

注意： 透過的アプリケーション・フェイルオーバー（TAF）の詳細は、第 4 章「データ・ソース」を参照してください。

TAF が構成されているほとんどの場合、アプリケーションは他のデータベース・インスタンスへのフェイルオーバーが発生したことを認識しません。そのため、障害からリカバリするために何かする必要はありません。

ただし、障害の発生時に OC4J から ORA エラーがスローされる場合があります。OJMS は、これらのエラーを関連する SQL 例外とともに JMSException としてユーザーに渡します。この場合は、次の 1 つ以上の操作を実行してください。

- エラーが致命的エラーかどうかは、DbUtil.oracleFatalError メソッドを使用して判断できます。3-46 ページの「JMS と RAC ネットワーク・フェイルオーバーの併用」を参照してください。致命的エラーでない場合、クライアントは短時間だけスリープした後、現行の操作を再試行してリカバリします。
- 少し待ってから JMS コネクションの使用を試行することで、不完全なフェイルオーバーにより発生したフェイルバックと一時的なエラーからリカバリできます。待っている間に、データベースはフェイルオーバーして障害からリカバリし、データベース自体を元の状態に戻します。
- トランザクション例外（「トランザクションをロールバックしてください。」（ORA-25402）、「トランザクションのステータスが不明です。」（ORA-25405）など）の場合は、現行の操作をロールバックし、前回のコミット後の操作をすべて再試行する必要があります。例外の原因が解消されるまで、接続は使用できません。この再試行に失敗した場合は、すべての接続をクローズして再作成し、コミットされていない操作をすべて再試行します。

両方の JMS プロバイダに対するフェイルオーバーのサーバー・サイドのサンプル・コード

次のコードに、サーバー・サイド・フェイルオーバーに対してトレラントなキューの JMS アプリケーション・コードを示します。この例は、OJMS と OracleAS JMS の両方に有効です。

```
while (notShutdown)
{
 Context ctx = new InitialContext();

 /* create the queue connection factory */
 QueueConnectionFactory qcf = (QueueConnectionFactory)
    ctx.lookup(QCF_NAME);
 /* create the queue */
 Queue q   = (Queue) ctx.lookup(Q_NAME);
 ctx.close();

 try
 {
  /*Create a queue connection, session, sender and receiver */
  QueueConnection qc = qcf.createQueueConnection();
  QueueSession qs = qc.createQueueSession(true, 0);
  QueueSender snd = qs.createSender(q);
  QueueReceiver rcv = qs.createReceiver(q);

  /* start the queue */
  qc.start();

  /* receive requests on the queue receiver and send out
     replies on the queue sender.
  while (notDone)
  {
    Message request = rcv.receive();
    Message reply   = qs.createMessage();

   /* put code here to process request and construct reply */

   snd.send(reply);
   qs.commit();
  }
  /* stop the queue */
  qc.stop();
 }
 catch (JMSException ex)
 {
  if (transientServerFailure)
 { // retry }
 else {
  notShutdown = false;
 }
}
```

クラスタリングのベスト・プラクティス

- JMS クライアント・サイドの状態を最小限に抑えます。
  - 処理をトランザクション済セッションで実行します。
  - 完全なリカバリ可能性を得るために、中間的なプログラムの状態を JMS キューまたはトピックに保存するか、チェックポイントを設定します。
  - J2EE アプリケーションの状態が JVM 境界にまたがってシリアライズ可能またはリカバリ可能であるかどうかに依存しません。JMS オブジェクトには常に一時的なメンバー変数を使用し、JMS の状態を適切に保存してリカバリする受動 / 能動およびシリアライズ / デシリアライズ関数を記述します。
- トピックへの非永続サブスクリプションを使用しません。
  - トピックへの非永続サブスクリプションでは、アクティブなサブスクライバごとにメッセージが複製されます。クラスタリングやロード・バランシングにより、複数のアプリケーション・インスタンスが作成されます。アプリケーションで非永続サブスクライバが作成されると、各メッセージの複製がトピックにパブリッシュされます。これは非効率的であるか、セマンティック的に無効です。
  - トピックには永続サブスクリプションのみを使用します。できるだけキューを使用してください。
- 永続サブスクリプションの存続期間を延長しません。
  - 常にアクティブであることが可能な永続サブスクリプションのインスタンスは 1 つのみです。クラスタリングやロード・バランシングにより、複数のアプリケーション・インスタンスが作成されます。アプリケーションで永続サブスクリプションが作成されると、クラスタ内のアプリケーションのインスタンスは 1 つのみ成功し、他のすべては `JMSException` が発生して失敗します。
  - 永続サブスクリプションは短時間または少数のコード範囲で作成、使用してクローズし、サブスクリプションがアクティブになっている期間を最短に抑えます。
  - クラスタリング（クラスタ内で実行されているアプリケーションの他のインスタンスが現在同じコード・ブロック内にあること）による永続サブスクリプションの作成失敗に対応するアプリケーション・コードを記述し、適切なバックオフ方法をプログラミングします。永続サブスクリプションの作成失敗は、常に致命的エラーとして処理しないでください。

4 データ・ソース

この章では、Oracle Application Server Containers for J2EE（OC4J）アプリケーションでのデータ・ソースの構成方法と使用方法について説明します。データ・ソースは、データベース・サーバーへの接続をベンダーに依存しないでカプセル化したものです。データ・ソースは、`javax.sql.DataSource` インタフェースを実装するオブジェクトをインスタンス化します。

この章には、次の項目が含まれます。

- 概要
- データ・ソースの定義
- データ・ソースの使用方法
- 2 フェーズ・コミットとデータ・ソースの使用
- Oracle JDBC の拡張機能の使用方法
- 接続キャッシング・スキームの使用方法
- OCI JDBC ドライバの使用方法
- DataDirect JDBC ドライバの使用方法
- データ・ソースの高可用性のサポート

概要

データ・ソースは、javax.sql.DataSource インタフェースを実装する Java オブジェクトです。データ・ソースは、JDBC 接続を作成するための、ベンダーに依存しない移植可能なメソッドを提供します。データ・ソースは、データベースへの JDBC 接続を戻すファクトリです。J2EE アプリケーションは、JNDI を使用して DataSource オブジェクトをルックアップします。各 JDBC 2.0 ドライバは、JNDI ネームスペースにバインド可能な DataSource オブジェクトの独自の実装を提供します。このデータ・ソース・オブジェクトがバインドされた後は、JNDI ルックアップによって取得できます。データ・ソースはベンダーに依存しないため、J2EE アプリケーションでは、データ・ソースを使用してデータ・サーバーへの接続を取得することをお薦めします。

データ・ソースのタイプ

OC4J では、データ・ソースが次のように分類されます。

- エミュレートされたデータ・ソース
- エミュレートされていないデータ・ソース
- ネイティブ・データソース

表 4-1 に、データ・ソース・タイプ間の主な違いを示します。

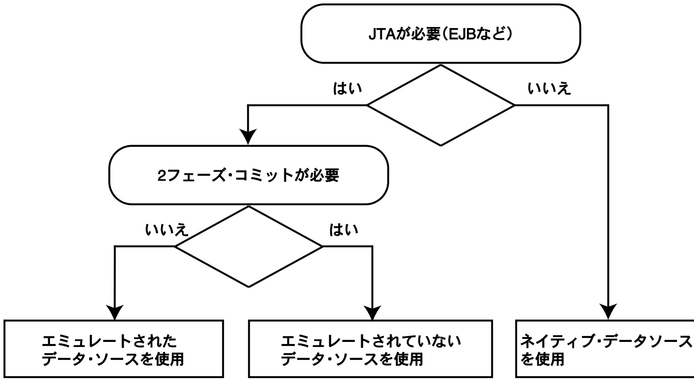
表 4-1 エミュレートされたデータ・ソース、エミュレートされていないデータ・ソース、ネイティブ・データソースの特徴

	エミュレートされている	エミュレートされていない
JTA なし		ネイティブ・データソース ■ ベンダーの拡張機能 ■ ベンダーの JDBC プール / キャッシュ ■ JTA なし
JTA	エミュレートされたデータ・ソース ■ 軽量トランザクション ■ 1 フェーズ・コミット ■ OC4J プール / キャッシュ	エミュレートされていないデータ・ソース ■ 完全トランザクション ■ 2 フェーズ・コミット ■ Oracle JDBC プール / キャッシュ

注意： ejb-location によりエミュレートされていないデータ・ソースにアクセスする場合は、OC4J のプールとキャッシュを使用します。OracleConnectionCacheImpl を使用すると、OC4J および Oracle JDBC のプールとキャッシュの両方にアクセスできます。

図 4-1 に、データ・ソースのタイプを選択する際に参考となる意志決定ツリーを示します。

図 4-1 データ・ソースのタイプの選択



次の項では、各データ・ソース・タイプの詳細を説明します。

エミュレートされたデータ・ソース

エミュレートされたデータ・ソースとは、JTA トランザクション用に XA プロトコルをエミュレートするデータ・ソースです。このタイプのデータ・ソースは、Oracle データ・ソース用の OC4J のキャッシュ、プーリングおよび Oracle JDBC の拡張機能を提供します。従来は、JDBC ドライバに XA 機能がなかったため、エミュレートされたデータ・ソースが必要でした。現在は、ほとんどの JDBC ドライバが XA 機能を備えていますが、エミュレートされた XA が引き続き優先される場合（2 フェーズ・コミットを必要としないトランザクションなど）があります。

エミュレートされたデータ・ソースから取得した接続は、XA グローバル・トランザクションの完全なサポートを提供せずに XA API をエミュレートするため、非常に高速になります。特に、エミュレートされたデータ・ソースは、2 フェーズ・コミットをサポートしません。ローカル・トランザクションに対して、あるいはアプリケーションで 2 フェーズ・コミットを必要としないグローバル・トランザクションを使用するときは、エミュレートされたデータ・ソースの使用をお薦めします。2 フェーズ・コミットの制限事項の詳細は、第 7 章「Java Transaction API」を参照してください。

エミュレートされたデータ・ソースの `data-sources.xml` 構成エントリを次に示します。

```
<data-source
    class="com.evermind.sql.DriverManagerDataSource"
    name="OracleDS"
    location="jdbc/OracleCoreDS"
    xa-location="OracleDS"
    ejb-location="jdbc/OracleDS"
    connection-driver="oracle.jdbc.driver.OracleDriver"
    username="scott"
    password="tiger"
    url="jdbc:oracle:thin:@//localhost:5521/oracle.regress.rdbms.dev.us.oracle.com"
    inactivity-timeout="30"
/>
```

エミュレートされたデータ・ソースを `data-sources.xml` 内で定義するときには、`location`、`ejb-location` および `xa-location` 属性の値を指定する必要があります。ただし、エミュレートされたデータ・ソースを JNDI 経由でルックアップするときには、`ejb-location` 属性で指定した値を使用します。次に例を示します。

```
Context ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup("jdbc/OracleDS");
// This lookup could also be done as
// DataSource ds = (DataSource) ic.lookup("java:comp/env/jdbc/OracleDS");
Connection con = ds.getConnection();
```

この接続によって、`scott/tiger` 用のデータベース・セッションがオープンします。

注意： 以前のリリースでは、データ・ソース・オブジェクトの取得に `location` 属性および `xa-location` 属性がサポートされていました。これらの属性は現在廃止されています。アプリケーション、EJB、サーブレットおよび JSP では、データ・ソースの取得に、エミュレートされたデータ・ソース定義で `ejb-location` の JNDI 名のみを使用する必要があります。エミュレートされたデータ・ソースの場合は 3 つの値をすべて指定する必要がありますが、実際に使用されるのは `ejb-location` のみです。

エミュレートされたデータ・ソースをグローバル・トランザクション内で使用する場合は、注意する必要があります。トランザクション・マネージャに登録する `XAResource` はエミュレートされた `XAResource` であるため、トランザクションは実際には 2 フェーズ・コミット・トランザクションになりません。グローバル・トランザクションに本当の 2 フェーズ・コミット・セマンティクスが必要な場合は、エミュレートされていないデータ・ソースを使用する必要があります。2 フェーズ・コミットの制限事項の詳細は、第 7 章「Java Transaction API」を参照してください。

単一のグローバル・トランザクション内で同じユーザー名とパスワードを使用して複数の接続をデータ・ソースから取得すると、複数の論理接続は単一の物理接続を共有します。次のコードは、単一の物理接続を共有する 2 つの接続 `conn1` と `conn2` を示しています。これらの接続は、いずれも同じデータ・ソース・オブジェクトから取得されます。また、同じユーザー名とパスワードを使用して認証を行います。

```
Context ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup("jdbc/OracleCMTDS1");
Connection conn1 = ds.getConnection("scott", "tiger");
Connection conn2 = ds.getConnection("scott", "tiger");
```

エミュレートされていないデータ・ソース

エミュレートされていないデータ・ソースは、グローバル・トランザクションの 2 フェーズ・コミット機能を含め、完全な（エミュレートされていない）JTA サービスを提供します。このタイプのデータ・ソースは、プーリング、キャッシュ、分散トランザクション機能およびベンダーの JDBC 拡張機能（現在は Oracle JDBC の拡張機能のみ）を提供します。2 フェーズ・コミットの制限事項の詳細は、第 7 章「Java Transaction API」を参照してください。

分散データベースの通信、リカバリおよび信頼性に関しては、エミュレートされていないデータ・ソースの使用をお薦めします。エミュレートされていないデータ・ソースは、同じデータベースへの同一ユーザー用の論理接続に対して物理接続を共有します。

注意： エミュレートされていないデータ・ソースの実行には、Java 対応データベースを使用する必要があります。エミュレートされていないデータ・ソースと非 Java 対応データベースを使用している場合、MDB アプリケーション（AQJMS）をデプロイすると例外が生成され、OC4J の stdout にスローされます。エミュレートされたデータ・ソースまたは Java 対応データベースに切り替えれば、デプロイは正しく行われます。

注意： エミュレートされていないデータ・ソースを使用するアプリケーションがアンデプロイされると、物理データベース接続は、OC4J が再起動されるまで、OC4J から削除されません。

エミュレートされていないデータ・ソースの data-sources.xml 構成エントリを次に示します。

```
<data-source
  class="com.evermind.sql.OrionCMTDataSource"
  location="jdbc/OracleDS"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="scott"
  password="tiger"
  url="jdbc:oracle:thin:@//localhost:5521/oracle.regress.rdbms.dev.us.oracle.com"
</data-source>
```

location 属性の値を使用して JNDI ルックアップを実行する必要があります。

次に、必要な属性定義を示します。

- location は、JNDI ネームスペース内でこのデータ・ソースがバインドされる JNDI 名です。このデータ・ソースを取得するには、JNDI ルックアップで location を使用します。
- url、username および password では、このデータ・ソースとの接続を取得するときに使用するデータベース、デフォルトのユーザー名およびパスワードを識別します。
- class は、ネームスペース内でバインドするデータ・ソース・クラスのタイプを定義します。

ネイティブ・データソース

ネイティブ・データソースとは、JDBC ベンダーが提供する DataSource 実装です。この種のデータ・ソースは、キャッシュ、プーリングおよびベンダー固有の拡張機能など、ベンダーの JDBC ドライバ機能を公開します。OC4J はネイティブ・データソースをグローバル・トランザクション内で登録できません。また、グローバル・トランザクションのセマンティクスを必要とする EJB や他のコンポーネントでは、これらを使用する可能性があります。そのため、この種のデータ・ソースを使用する場合は注意する必要があります。

ネイティブ・データソース実装はエミュレータを使用せずに直接使用できます。OC4J はネイティブ・データソースの直接使用をサポートしており、ベンダー固有のプーリング、キャッシュ、拡張機能およびプロパティによるメリットが得られます。ただし、ネイティブ・データソースは JTA サービス（開始、コミット、ロールバックなど）を提供しません。

ネイティブ・データソースの data-sources.xml 構成エントリを次に示します。

```
<data-source
    class="com.my.DataSourceImplementationClass"
    name="NativeDS"
    location="jdbc/NativeDS"
    username="user"
    password="pswd"
    url="jdbc:myDataSourceURL"
</data-source>
```

JNDI ルックアップを実行するには、location 属性の値を使用する必要があります。

データ・ソースの混在

単一のアプリケーションで、異なるタイプの複数のデータ・ソースを使用する場合があります。

アプリケーションでデータ・ソースを混在させている場合は、次の問題点に注意してください。

- JTA トランザクションをサポートするのは、エミュレートされたデータ・ソースとエミュレートされていないデータ・ソースのみです。

  ネイティブ・データソースから取得した接続は JTA トランザクションに登録できません。

- 本当の 2 フェーズ・コミットをサポートするのは、エミュレートされていないデータ・ソースのみです（エミュレートされたデータ・ソースは 2 フェーズ・コミットをエミュレートします）。

  2 フェーズ・コミット・トランザクションに複数の接続を登録するには、すべての接続でエミュレートされていないデータ・ソースを使用する必要があります。2 フェーズ・コミットの制限事項の詳細は、第 7 章「Java Transaction API」を参照してください。

- JTA トランザクションをオープンし、そのトランザクションに参加しない接続を取得する場合は、ネイティブ・データソースを使用します。

- アプリケーションで JTA トランザクションを使用していない場合は、どのデータ・ソースからでも接続を取得できます。

- アプリケーションで javax.transaction.UserTransaction をオープンしている場合、その後のすべてのトランザクション作業は、そのオブジェクトを介して実行する必要があります。

  接続の rollback() メソッドまたは commit() メソッドを起動しようとすると、次のような SQLException を受け取ります。

  ```
  calling commit() [or rollback()] is not allowed on a container-managed transactions
  Connection
  ```

  次の例に、実行される操作を示します。

```
Context ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup("JDBC/OracleCMTDS1"); // Using JTA DataSources
Connection conn1 = ds.getConnection("scott", "tiger");
javax.transaction.UserTransaction ut =
    (javax.transaction.UserTransaction)ic.lookup("java:comp/UserTransaction");
ut.begin();
conn1.query();
conn1.commit(); // not allowed, returns error: calling commit[or rollback] is not
allowed
                // on a container-managed transaction connection
```

データ・ソースの定義

OC4J データ・ソースは、`data-sources.xml` ファイルに定義します。

OC4J とともにインストールされる `data-sources.xml` ファイルには、事前定義済のデフォルト・データ・ソースが含まれており、通常は、このデータ・ソースで十分です。ただし、不十分な場合は、独自にデータ・ソースを定義する必要があります。

表 4-2 に、各タイプのデータ・ソースの構成要件を示します。

表 4-2 データ・ソース構成の概要

構成	エミュレートされていないデータ・ソース	エミュレートされたデータ・ソース	ネイティブ・データソース
データ・ソースのクラス	`OrionCMTDataSource`	`DriverManagerDataSource`	`OracleConnection-CacheImpl`
接続ドライバ	該当なし	ベンダー固有 Oracle の拡張機能用 OracleDriver	該当なし

JNDI コンテキスト仕様	location	location
		`ejb-location`
		`xa-location`
			location

JNDI コンテキスト・ルックアップ	location	`ejb-location`	location
URL	Oracle ドライバの URL	ベンダー固有 Oracle: シンまたは OCI（OCI 付き TAF）	ベンダー固有 Oracle: シンまたは OCI（OCI 付き TAF）
追加構成	Oracle データベース・コミット・コーディネータ 2 フェーズ・コミット・コーディネータ用データベース・リンク	なし	キャッシュ・スキーム

注意： `OrionCMTDataSource` およびその要素の詳細は、`http://www.orionserver.com/docs/api/orion/com/evermind/sql/OrionCMTDataSource.html` を参照してください。

表 4-3 に、各タイプのデータ・ソースの特性を示します。

表 4-3 データ・ソースの特性

特性	エミュレートされていないデータ・ソース	エミュレートされたデータ・ソース	ネイティブ・データソース
プールとキャッシュのサポート	Oracle JDBC ドライバ・プール	OC4J 接続プール	ベンダー固有 Oracle
ベンダーの拡張機能のサポート	Oracle のみ	Oracle のみ	ベンダー固有 Oracle
JTA サポート	完全 XA（1 または 2 フェーズ・コミット）	エミュレートされた XA（1 フェーズ・コミット）	サポート外
J2CA サポート	なし	あり	あり

注意： ejb-location によりエミュレートされていないデータ・ソースにアクセスする場合は、OC4J のプールとキャッシュを使用します。OracleConnectionCacheImpl を使用すると、OC4J および Oracle JDBC のプールとキャッシュの両方にアクセスできます。

新規データ・ソース・オブジェクトを定義する手順は、次のとおりです。

1. data-sources.xml ファイルの位置を選択します（4-8 ページの「構成ファイル」を参照）。
2. データ・ソースの属性を理解します（4-9 ページの「データ・ソースの属性」を参照）。
3. データ・ソースを定義します。これには、Oracle Enterprise Manager 10*g* を使用する方法（4-11 ページの「Oracle Enterprise Manager 10g でのデータ・ソースの定義」を参照）と、構成ファイルを手動で編集する方法（4-12 ページの「XML 構成ファイルでのデータ・ソースの定義」を参照）があります。

構成ファイル

1 つの主要な構成ファイル J2EE_HOME/config/data-sources.xml により、データ・ソースは OC4J サーバー・レベルで設定されます。

各アプリケーションには、別々の JNDI ネームスペースもあります。ファイル web.xml、ejb-jar.xml、orion-ejb-jar.xml および orion-web.xml には、アプリケーションの JNDI 名をデータ・ソースにマップするために使用できるエントリが含まれています。これらのエントリについて、次の項で説明します。

データ・ソース XML 構成ファイルの位置の定義

アプリケーションは、application.xml ファイルでこのファイルが認識されている場合にのみ、このファイルに定義されているデータ・ソースを認識できます。application.xml ファイルの <data-sources> タグの path 属性には、data-sources.xml ファイルの名前とパスが含まれている必要があります。次に例を示します。

```
<data-sources path="data-sources.xml"/>
```

<data-sources> タグの path 属性には、data-sources.xml ファイルのフルパス名が含まれます。パスには、絶対パス、または application.xml の位置からの相対パスのどちらでも指定できます。application.xml ファイルと data-sources.xml ファイルは、いずれも *J2EE_HOME*/config/application.xml ディレクトリに格納されています。したがって、このパスには data-sources.xml ファイル名のみが含まれています。

アプリケーション固有のデータ・ソースの XML 構成ファイル

各アプリケーションでは、EAR ファイルに、独自の data-sources.xml ファイルを定義できます。そのためには、EAR ファイルにパッケージされている orion-application.xml ファイル内に、data-sources.xml ファイルへの参照を含めます。

構成手順は、次のとおりです。

1. アプリケーションの META-INF ディレクトリ内で、data-sources.xml ファイルと orion-application.xml ファイルを検索します。
2. orion-application.xml ファイルを編集して、次のように <data-sources> タグを追加します。

```
<orion-application>
    <data-sources path="./data-sources.xml"/>
</orion-application>
```

データ・ソースの属性

データ・ソースには、多数の属性を設定できます。一部の属性は必須ですが、ほとんどはオプションです。後述の表では、属性が必須の場合はその旨を示してあります。属性は、<data-source> タグで指定されます。

表 4-4 に、データ・ソースの属性とその説明を示します。

表 4-4 に示す data-source の属性の他に、data-source には property サブノードを追加することもできます。これらのサブノードは、データ・ソース・オブジェクトの汎用プロパティの構成に使用されます（JavaBeans の規則に従います)。property ノードには name および value 属性があり、データ・ソースの Bean プロパティの名前と値を指定するために使用されます。

OC4J のデータ・ソースの属性はすべて、インフラストラクチャ・データベースにも適用できます。インフラストラクチャ・データベースの詳細は、『Oracle 高可用性アーキテクチャおよびベスト・プラクティス』を参照してください。

表 4-4 データ・ソースの属性

属性名	説明	デフォルト値
class	データ・ソースを実装するクラスに名前を付けます。 エミュレートされていないデータ・ソースの場合は、com.evermind.sql.OrionCMTDataSource を使用してもかまいません。 エミュレートされたデータ・ソースの場合は、com.evermind.sql.DriverManagerDataSource を使用する必要があります。 （この値は必須です。）	該当なし
location	データ・ソース・オブジェクトの JNDI 論理名。OC4J は、この名前でクラス・インスタンスをアプリケーションの JNDI ネームスペースにバインドします。この JNDI ルックアップ名は、エミュレートされていないデータ・ソースに使用されます。4-7 ページの表 4-2「データ・ソース構成の概要」も参照してください。	該当なし
name	データ・ソース名。アプリケーション内で一意であることが必要です。	なし
connection-driver	このデータ・ソースの JDBC ドライバ・クラス名。java.sql.Connection を処理する一部のデータ・ソースで使用されます。 ほとんどのデータ・ソースの場合、ドライバは oracle.jdbc.driver.OracleDriver です。この属性は、class 属性が com.evermind.sql.DriverManagerDataSource に設定されている、エミュレートされたデータ・ソースにのみ適用されます。	なし
username	データ・ソース接続の取得時に使用されるデフォルトのユーザー名。	なし
password	データ・ソース接続の取得時に使用されるデフォルトのパスワード。4-13 ページの「パスワードの間接化」も参照してください。	なし
URL	データベース接続用の URL。	なし
xa-location	XA データ・ソースの論理名。エミュレートされたデータ・ソースにのみ適用されます。4-7 ページの表 4-2「データ・ソース構成の概要」も参照してください。	なし
ejb-location	この属性は、JTA の 1 フェーズ・コミット・トランザクション、またはエミュレートされたデータ・ソースのルックアップに使用します。この属性を使用してデータ・ソースを取得すると、戻された接続を oracle.jdbc.OracleConnection にマップできます。4-7 ページの表 4-2「データ・ソース構成の概要」も参照してください。	なし

表 4-4 データ・ソースの属性(続き)

属性名	説明	デフォルト値
stmt-cache-size	JDBC の文キャッシュを有効化し、キャッシュされる文の最大数を定義するために 0(ゼロ)以外の値に設定されるパフォーマンス・チューニング属性。反復的なカーソル作成と文の解析および作成によるオーバーヘッドを回避するために有効化されます。connection-driver が oracle.jdbc.driver.OracleDriver に、class が com.evermind.sql.DriverManagerDataSource に設定されている、エミュレートされたデータ・ソースにのみ適用されます。	0(無効化)
inactivity-timeout	未使用の接続をクローズするまでのキャッシュ時間(秒)。	60 秒
connection-retry-interval	失敗した接続を再試行するまでの待機間隔(秒)。	1 秒
max-connections	プールされたデータ・ソースのオープン接続の最大数。	データ・ソースのタイプによって異なります。
min-connections	プールされたデータ・ソースのオープン接続の最小数。OC4J は、DataSource.getConnection メソッドが起動されるまでこれらの接続をオープンしません。	0
wait-timeout	プールが max-connections に到達した場合に、接続が解放されるのを待機する時間(秒)。	60
max-connect-attempts	接続の再試行回数。なんらかの理由でネットワークや環境が不安定であるために接続に失敗する場合に有効です。	3
clean-available-connections-threshold	このオプションの属性では、使用可能な接続のクリーン・アップが発生する時期のしきい値(秒)を指定します。たとえば、ある接続が不良な場合は、使用可能な接続がクリーン・アップされます。別の接続が不良な場合(つまり、例外をスローする場合)に、しきい値で指定した時間が経過すると、使用可能な接続が再びクリーン・アップされます。しきい値で指定した時間が経過しなければ、使用可能な接続が再びクリーン・アップされることはありません。	30
rac-enabled	このオプションの属性では、システムが Real Application Clusters (RAC)対応かどうかを指定します。このフラグをインフラストラクチャ・データベースで使用する方法については、『Oracle 高可用性アーキテクチャおよびベスト・プラクティス』を参照してください。このフラグをユーザー・データベースで使用する方法については、4-24 ページの「DataDirect JDBC ドライバの使用方法」および 4-26 ページの「データ・ソースの高可用性のサポート」を参照してください。 データ・ソースが RAC データベースを指す場合は、このプロパティを true に設定します。これにより、OC4J は RAC インスタンス障害の発生時のパフォーマンスが改善される方法で接続プールを管理できるようになります。	false
schema	このオプションの属性では、データ・ソースに関連付ける database-schema を指定します。特に、追加のデータ型やサード・パーティ・データベースとともに CMP を使用する場合に有効です。この属性の使用方法については、4-15 ページの「データベース・スキーマとデータ・ソースの関連付け」を参照してください。	なし

次に、clean-available-connections-threshold および rac-enabled 属性の使用例を示します。

```
<data-source
    class="com.evermind.sql.OrionCMTDataSource"
    name="NEDS1"
    location="jdbc/NELoc1"
    connection-driver="oracle.jdbc.driver.OracleDriver"
    min-connections="5"
    max-connections="10"
    clean-available-connections-threshold="35"
    rac-enabled="true"
    username="scott"
    password="tiger"
    url="jdbc:oracle:thin:@//localhost:5521/oracle.regress.rdbms.dev.us.oracle.com"
    inactivity-timeout="30"
    max-connect-attempts="5"
/>
```

OC4J では、定義するデータ・ソースごとに、location、ejb-location、xa-location および pool-location に対して 1 つずつ、最大 4 つのデータ・ソースを作成して JNDI 内でバインドできます。選択するデータ・ソースのタイプは、data-sources.xml 属性の class、connection-driver および url に関連付けられている値と、データ・ソース・オブジェクトが作成されルックアップされる JNDI コンテキストにより決定されます。データ・ソースのタイプの詳細は、4-2 ページの「データ・ソースのタイプ」を参照してください。

Oracle Enterprise Manager 10*g* でのデータ・ソースの定義

どのタイプのデータ・ソースも、Oracle Enterprise Manager 10*g* を使用して定義できます。

管理ツールの使用方法は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』を参照してください。Oracle Enterprise Manager 10*g* の詳細は、『Oracle Enterprise Manager 管理者ガイド』を参照してください。

この項では、これらの手順の概略を説明します。

Oracle Enterprise Manager 10*g* を使用して「データ・ソース」ページにドリルダウンします。OC4J では、起動時に data-sources.xml ファイルが解析され、データ・ソース・オブジェクトがインスタンス化されてサーバーの JNDI ネームスペースにバインドされます。新規データ・ソースの指定を追加した場合は、OC4J サーバーを再起動して新規データ・ソースをルックアップに使用できるようにする必要があります。

エミュレートされたデータ・ソースを定義するには、エミュレートされていないデータ・ソースの定義と同じ手順を、JNDI ロケーションを定義する手順の 1 つ前まで実行します。エミュレートされていないデータ・ソースの定義手順のスクリーン・ショットには、「**ロケーション**」入力フィールドがあります。エミュレートされたデータ・ソースを定義する場合は、3 つのフィールド「**ロケーション**」、「**トランザクション (XA) ロケーション**」および「**EJB ロケーション**」に入力します。

> **注意：** 以前のリリースでは、データ・ソース・オブジェクトの取得に location 属性および xa-location 属性がサポートされていました。これらの属性はすでに廃止されています。アプリケーション、EJB、サーブレットおよび JSP では、データ・ソースの取得に、エミュレートされたデータ・ソース定義の ejb-location の JNDI 名のみを使用する必要があります。エミュレートされたデータ・ソースの場合は 3 つの値をすべて指定する必要がありますが、実際に使用されるのは ejb-location のみです。

XML 構成ファイルでのデータ・ソースの定義

$J2EE_HOME/config/data-sources.xml ファイルは、デフォルトのデータ・ソースとともに事前にインストールされます。ほとんどの場合、このデフォルトのデータ・ソースで十分です。ただし、カスタマイズした独自のデータ・ソース定義を追加することもできます。

デフォルトのデータ・ソースは、エミュレートされたデータ・ソースです。

データ・ソースのタイプの詳細は、4-2 ページの「データ・ソースのタイプ」を参照してください。

次は、ほぼすべてのアプリケーション用に変更できる、エミュレートされたデータ・ソースの簡単な定義です。

```
<data-source
  class="com.evermind.sql.DriverManagerDataSource"
  name="OracleDS"
  location="jdbc/OracleCoreDS"
  xa-location="OracleDS"
  ejb-location="jdbc/OracleDS"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="scott"
  password="tiger"
  url="jdbc:oracle:thin:@//localhost:5521/oracle.regress.rdbms.dev.us.oracle.com"
  inactivity-timeout="30"
/>
```

すべてのデータ・ソース属性の詳細は、4-9 ページの「データ・ソースの属性」を参照してください。

致命的エラー・コードの拡張

data-sources.xml で定義したデータ・ソースごとに、そのデータ・ソースの通信対象であるバックエンド・データベースにアクセスできないことを示す致命的エラー・コードを定義できます。OC4J でこれらのエラー・コードのいずれかが検出されると（JDBC ドライバによって SQLException がスローされた場合）、その接続プールは削除されます。つまり、接続プールのすべての接続がクローズされます。Oracle で事前定義されている致命的エラー・コードは、3113、3114、1033、1034、1089 および 1090 です。

Oracle にその他の致命的エラー・コードを追加するには、次の手順を実行します。

<data-source> 要素のサブタグである <fatal-error-codes> 要素を使用します。<fatal-error-codes> 要素で、子要素の <error-code> を使用して 1 個の致命的エラー・コードを定義します。1 つの <fatal-error-codes> 要素に対して、0 〜 n 個の <error-code> 要素を定義できます。たとえば、致命的エラー・コードとして 10、20、30 を設定する場合、データ・ソース定義は次のようになります。

```
<data-source
class="com.evermind.sql.DriverManagerDataSource"
name="ds"
location="jdbc/ds"
xa-location="jdbc/xa/ds"
ejb-location="jdbc/ejb/ds"
@ connection-driver="oracle.jdbc.driver.OracleDriver"
username="scott"
@ password="tiger"

@ url="jdbc:oracle:thin:@//localhost:1521/oracle.regress.rdbms.dev.us.oracle.com">

         <fatal-error-codes>
            <error-code code='10'/>
            <error-code code='20'/>
            <error-code code='30'/>
         </fatal-error-codes>

</data-source>
```

パスワードの間接化

data-sources.xml ファイルには、認証用のパスワードが必要です。これらのパスワードをデプロイメント・ファイルおよび構成ファイルに埋め込むと、特にこのファイルのパーミッションにより誰でも読み取ることができる場合には、セキュリティ・リスクを招きます。この問題を回避するために、OC4J はパスワードの間接化をサポートしています。

間接パスワードは、特殊な間接化記号（->）とユーザー名（またはユーザー名とレルム）で構成されます。OC4J は間接パスワードを検出すると、付与されているアクセス権限を使用して、ユーザー・マネージャが提供するセキュリティ・ストアから、指定のユーザーに関連付けられているパスワードを取得します。

ユーザーとパスワードの作成およびユーザー・マネージャの操作の詳細は、『Oracle Application Server Containers for J2EE セキュリティ・ガイド』の「パスワード管理」を参照してください。

たとえば、4-3 ページの「エミュレートされたデータ・ソース」のサンプル・コードには次の行が含まれています。

```
password="tiger"
```

この行を、次のように間接化記号（->）とユーザー名（scott）で置き換えることができます。

```
password="->scott"
```

これは、パスワード tiger を持つユーザー scott がユーザー・マネージャで作成されている場合です。

OC4J はセキュリティ・ストアへのアクセス権限を持つため、このユーザー（scott）に関連付けられているパスワード（tiger）を取得できます。

パスワードの間接化を構成するには、次の 2 つの方法があります。

- Oracle Enterprise Manager 10g を使用した間接パスワードの構成
- 手動による間接パスワードの構成

Oracle Enterprise Manager 10*g* を使用した間接パスワードの構成

Oracle Enterprise Manager 10*g* を使用して間接パスワードを構成する手順は、次のとおりです。

1. Oracle Enterprise Manager 10*g* にログインします。
2. タイプ OC4J のターゲットを選択します。
3. 「**管理**」を選択します。Oracle Application Server の Oracle Enterprise Manager 10*g* のホームページが表示されます。
4. 「**管理**」を選択します。
5. 「**データ・ソース**」を選択します。データ・ソースのリストが表示されます。
6. 「**選択**」列でクリックしてデータ・ソースを選択します。
7. 「**編集**」をクリックします。図 4-2 に示す「データ・ソースの編集」ページが表示されます。

図 4-2「データ・ソースの編集」ページ

Datasource Username and Password

Cleartext passwords may pose a security risk, especially if the permissions on the data-sources.xml configuration file allows it to be read by any user. You can specify an indirect password to avoid this risk. An indirect password is used to do a look up in the User Manager to get the password.

Username scott

Use Cleartext Password

Password

Use Indirect Password

Indirect Password scott

example: Scott, customers/Scott

8. **ユーザー名およびパスワード**領域で「**間接パスワードの使用**」をクリックし、「**間接パスワード**」フィールドに適切な値を入力します。
9. 「**適用**」をクリックします。

手動による間接パスワードの構成

データ・ソースの間接パスワードを手動で構成する手順は、次のとおりです。

1. 該当する OC4J XML 構成ファイルまたはデプロイメント・ファイルを編集します。
   - `data-sources.xml`: `<data-source>` 要素の `password` 属性
   - `ra.xml`: `<res-password>` 要素
   - `rmi.xml`: `<cluster>` 要素の `password` 属性
   - `application.xml`: `<resource-provider>` 要素と `<commit-coordinator>` 要素の `password` 属性
   - `jms.xml`: `<password>` 要素
   - `internal-settings.xml`: `<sep-property>` 要素、`attributes name=" keystore-password"` および `name=" truststore-password"`
2. 前述のパスワードのいずれかを間接パスワードにするには、リテラルのパスワード文字列を、「->」に続けてユーザー名またはレルムとユーザー名をスラッシュ（/）で区切った文字列で置き換えます。

例： `<data-source password="->Scott">`

これにより、ユーザー・マネージャはユーザー名 Scott をルックアップし、そのユーザー用に格納されているパスワードを使用します。

データベース・スキーマとデータ・ソースの関連付け

データ・ソースは、データベース・インスタンスを識別します。データ・ソースの schema 属性を使用すると、特定のデータベース用にカスタマイズできる database-schema.xml ファイルにデータ・ソースを関連付けることができます。

コンテナ管理の永続性（CMP）を使用する場合、コンテナは Bean を保持するために必要なデータベース・スキーマの作成を受け持ちます。データ・ソースを database-schema.xml ファイルに関連付けると、コンテナにより最終的に生成される SQL に反映させることができます。これにより、アプリケーションではサポートされてもデータベースではサポートされない追加のデータ型（java.math.BigDecimal など）への対応などの問題を解決できます。

database-schema.xml ファイル

例 4-1 に示すように、database-schema.xml ファイルには database-schema 要素が含まれています。この要素は、表 4-5 に示す属性で構成されます。

例 4-1 database-schema 要素

```
<database-schema case-sensitive="true" max-table-name-length="30" name="MyDatabase"
not-null="not null" null="null" primary-key="primary key">
    <type-mapping type="java.math.BigDecimal" name="number(20,8)" />
    <disallowed-field name="order" />
</database-schema>
```

表 4-5 database-schema.xml ファイルの属性

属性	説明
case-sensitive	このデータベースで名前が大 / 小文字を区別して処理されるかどうか（true または false）を指定します。この指定は、disallowed-field サブ要素で指定する名前に適用されます。
max-table-name-length	このオプションの属性では、このデータベース用の表名の最大長を指定します。この値よりも長い名前は切り捨てられます。
name	このデータベースの名前です。
not-null	このデータベースで NOT NULL 制約を示すためのキーワードを指定します。
null	このデータベースで NULL 制約を示すためのキーワードを指定します。
primary-key	このデータベースで主キー制約を示すためのキーワードを指定します。

<database-schema> 要素には、次のサブ要素を必要な数だけ含めることができます。

- <type-mapping>
- <disallowed-field>

<type-mapping> このサブ要素は、Java の型をこのデータベース・インスタンスに適切な対応する型にマップするために使用されます。このサブ要素には次の 2 つの属性があります。

- name: データベース型の名前
- type: Java の型の名前

<disallowed-field> このサブ要素は、このデータベース・インスタンスの予約語であるために使用できない名前を指定します。このサブ要素の属性は 1 つです。

- name: 予約語の名前

構成例

この例では、アプリケーションでサポートされているデータ型（java.math.BigDecimal）を、基礎となるデータベースでサポートされているデータ型にマップする方法を示します。

1. database-schemas/oracle.xml ファイル内で、java.math.BigDecimal のマッピングを次のように定義します。

```
<type-mapping type="java.math.BigDecimal" name="number(20,8)" />
```

2. このスキーマを <data-source> 要素内で次のように使用します。

```
<data-source
    class="com.evermind.sql.DriverManagerDataSource"
    name="OracleDS"
    ejb-location="jdbc/OracleDS"
    schema="database-schemas/oracle.xml"
    connection-driver="oracle.jdbc.driver.OracleDriver"
    username="scott"
    password="tiger"
    url="jdbc:oracle:thin:@//localhost:5521/oracle.regress.rdbms.dev.us.oracle.com"
    clean-available-connections-threshold="30"
    rac-enabled="false"
    inactivity-timeout="30"
/>
```

3. 次のように、この <data-source> を EJB に使用します。

```
<orion-ejb-jar>
   <enterprise-beans>
     <entity-deployment name="BigDecimalTest" data-source="jdbc/OracleDS" />
   </enterprise-beans>
```

4. EJB をデプロイすると、OC4J により適切な表が作成されます。

データ・ソースの使用方法

この項では、アプリケーションでのデータ・ソースの使用方法について説明します。

- 移植可能なデータ・ソース・ルックアップ
- データ・ソースからの接続の取得
- エミュレートされていないデータ・ソースによる接続の取得
- 接続取得のエラー条件

データ・ソースのメソッドについては、J2EE API のドキュメントを参照してください。

移植可能なデータ・ソース・ルックアップ

OC4J サーバーの起動時に、j2ee/home/config ディレクトリの data-sources.xml ファイル内のデータ・ソースが OC4J の JNDI ツリーに追加されます。JNDI を使用してデータ・ソースをルックアップするときは、JNDI ルックアップを次のように指定します。

```
DataSource ds = ic.lookup("jdbc/OracleCMTDS1");
```

OC4J サーバーは、独自の内部 JNDI ツリーでこのデータ・ソースを検索します。

ただし、アプリケーションでは、移植可能な java:comp/env 機能を使用して、アプリケーションの JNDI ツリーでデータ・ソースをルックアップする、より移植性のある方法をお薦めします。<resource-ref> タグを使用して、アプリケーションの web.xml ファイルまたは ejb-jar.xml ファイルに、データ・ソースを指すエントリを格納します。次に例を示します。

```
<resource-ref>
     <res-ref-name>jdbc/OracleDS</res-ref-name>
     <res-type>javax.sql.DataSource</res-type>
```

```
     <res-auth>Container</res-auth>
</resource-ref>
```

<res-ref-name> には次のいずれかを指定します。

- data-sources.xml ファイルに定義されている実際の JNDI 名（jdbc/OracleDS など）。この場合、マッピングは必要ありません。前述のコード例を参照してください。<res-ref-name> は、data-sources.xml ファイルにバインドされている JNDI 名と同じです。

  次の JNDI ルックアップのように、java:comp/env を使用せずにこのデータ・ソースを取得します。

  ```
  InitialContext ic = new InitialContext();
  DataSource ds = ic.lookup("jdbc/OracleDS");
  ```

- OC4J 固有のファイル orion-web.xml または orion-ejb-jar.xml で実際の JNDI 名にマップされている論理名。この場合、OC4J 固有の XML ファイルによって、web.xml ファイルまたは ejb-jar.xml ファイルの論理名から、data-sources.xml ファイルに定義されている実際の JNDI 名へのマッピングが定義されます。

例 4-2 論理 JNDI 名の実際の JNDI 名へのマッピング

次のコードは、前述の 2 つのうち 2 番目のオプションを示しています。JNDI 取得用のコードで論理名 jdbc/OracleMappedDS を使用するように選択する場合は、web.xml または ejb-jar.xml ファイル内で次のように記述します。

```
<resource-ref>
     <res-ref-name>jdbc/OracleMappedDS</res-ref-name>
     <res-type>javax.sql.DataSource</res-type>
     <res-auth>Container</res-auth>
</resource-ref>
```

実際の JNDI 名が検索されるように、jdbc/OracleMappedDS を data-sources.xml ファイル内の実際の JNDI 名にマップする、<resource-ref-mapping> 要素を記述する必要があります。デフォルトのエミュレートされたデータ・ソースを使用している場合、ejb-location は実際の JNDI 名の jdbc/OracleDS で定義されます。次に例を示します。

```
<resource-ref-mapping name="jdbc/OracleMappedDS" location="jdbc/OracleDS" />
```

この結果、次の Java 文を使用して、アプリケーションの JNDI ネームスペースでデータ・ソースをルックアップできます。

```
InitialContext ic = new InitialContext();
DataSource ds = ic.lookup("jdbc/OracleMappedDS");
```

データ・ソースからの接続の取得

データベースのデータを変更する 1 つの方法は、JDBC 接続を取得し、JDBC 文を使用することです。JDBC 操作では、かわりにデータ・ソース・オブジェクトの使用をお薦めします。

> **注意 :** データ・ソースは常に、論理接続を戻します。

データベース内のデータを変更する手順は、次のとおりです。

1. data-sources.xml ファイルのデータ・ソース定義の JNDI ルックアップを使用して DataSource オブジェクトを取得します。

   ルックアップは、デフォルト・データ・ソースの論理名で実行されます。デフォルト・データ・ソースは、data-sources.xml ファイルの ejb-location タグに定義されているエミュレートされたデータ・ソースです。

   JNDI の lookup() メソッドは、Java の object を戻すため、JNDI が DataSource に戻すオブジェクトは、必ずキャストまたはナローイングする必要があります。

2. DataSource オブジェクトで示されたデータベースへの接続を作成します。

接続の作成後、データ・ソースによって指定されたこのデータベースに対して JDBC 文を構成し、実行できます。

次のコードは、これらの手順を示しています。

```
Context ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup("jdbc/OracleDS");
Connection conn = ds.getConnection();
```

アプリケーション・コードで、次の DataSource オブジェクトのメソッドを使用して、データベースへの接続を取得します。

- getConnection();

  ユーザー名とパスワードは、データ・ソース定義で定義されているユーザー名とパスワードです。

- getConnection(String username, String password);

  このユーザー名とパスワードは、データ・ソース定義で定義されているユーザー名とパスワードに優先します。

データ・ソースが Oracle データベースを参照している場合は、getConnection メソッドで戻された接続オブジェクトを oracle.jdbc.OracleConnection にキャストして、すべての Oracle 拡張機能を使用できます。詳細は、4-21 ページの「Oracle JDBC の拡張機能の使用方法」を参照してください。

次の例にこの操作を示します。

```
oracle.jdbc.OracleConnection conn =
  (oracle.jdbc.OracleConnection) ds.getConnection();
```

接続の取得後は、JDBC を介して、データベースに対して SQL 文を実行できます。

一般的な接続取得エラーを処理する方法については、4-18 ページの「エミュレートされていないデータ・ソースによる接続の取得」を参照してください。

エミュレートされていないデータ・ソースによる接続の取得

エミュレートされていないデータ・ソース・オブジェクトを変更する場合の物理的な動作は、接続をグローバル・トランザクションの外部と内部のどちらのデータ・ソースから取得しているかによって異なります。次の項では、これらの相違点について説明します。

- グローバル・トランザクション外からの接続の取得
- グローバル・トランザクション内からの接続の取得

グローバル・トランザクション外からの接続の取得

ユーザーがエミュレートされていないデータ・ソースから接続を取得し、そのユーザーがグローバル・トランザクションに含まれていない場合は、getConnection メソッドごとに論理ハンドルが戻されます。接続が作業に使用されるとき、作成された各接続ごとに物理接続が作成されます。したがって、グローバル・トランザクション外で 2 つの接続を作成した場合は、両方の接続が別々の物理接続を使用します。各接続をクローズすると、次の接続の取得で使用されるように、接続はプールに戻されます。

グローバル・トランザクション内からの接続の取得

ユーザーがエミュレートされていないデータ・ソースから接続を取得し、そのユーザーがグローバル JTA トランザクションに含まれている場合は、そのトランザクション内で同一ユーザーが同じ DataSource オブジェクトから取得した物理接続はすべて、同じ物理接続を共有します。

たとえば、トランザクションを開始し、scott ユーザーで jdbc/OracleCMTDS1 DataSource から 2 つの接続を取得した場合は、両方の接続が物理接続を共有します。次の例では、conn1 と conn2 が同じ物理接続を共有します。

```
Context ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup("jdbc/OracleCMTDS1");
txn.begin(); //start txn
Connection conn1 = ds.getConnection("scott", "tiger");
Connection conn2 = ds.getConnection("scott", "tiger");
```

ただし、異なる DataSource オブジェクトから取得された接続の場合は、それぞれに物理接続が取得されます。次の例では、conn1 と conn2 が異なる DataSource オブジェクト（jdbc/OracleCMTDS1 と jdbc/OracleCMTDS2）から取得されます。conn1 と conn2 は、別個の物理接続上に存在します。

```
Context ic = new InitialContext();
DataSource ds1 = (DataSource) ic.lookup("jdbc/OracleCMTDS1");
DataSource ds2 = (DataSource) ic.lookup("jdbc/OracleCMTDS2");
txn.begin; //start txn
Connection conn1 = ds1.getConnection();
Connection conn2 = ds2.getConnection();
```

接続取得のエラー条件

次の場合にはエラーが発生する可能性があります。

- 単一のデータ・ソースに対する 2 つの接続に異なるユーザー名を使用した場合
- OCI JDBC ドライバが正しく構成されていない場合

単一のデータ・ソースに対する 2 つの接続に異なるユーザー名を使用した場合

ユーザー名とパスワードを指定して DataSource オブジェクトから接続を取得したとき、このユーザー名とパスワードは、同じトランザクション内のその後のすべての接続の取得で使用されます。これは、すべてのタイプのデータ・ソースに当てはまります。

たとえば、アプリケーションがユーザー名 scott で jdbc/OracleCMTDS1 データ・ソースから接続を取得したとします。同じデータ・ソースから 2 番目の接続を、adams などの別のユーザー名で取得すると、2 番目のユーザー名（adams）は無視されます。かわりに、元のユーザー名（scott）が使用されます。

```
Context ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup("jdbc/OracleCMTDS1");
txn.begin(); //start txn
Connection conn1 = ds.getConnection("scott", "tiger"); //uses scott/tiger
Connection conn2 = ds.getConnection("adams", "woods"); //uses scott/tiger also
```

このように、同じデータ・ソースに対して 2 つの異なるユーザーを使用して認証を行うことはできません。adams/woods で表にアクセスしようとすると、エラーが発生します。

OCI JDBC ドライバが正しく構成されていない場合

OCI JDBC ドライバを使用している場合は、4-23 ページの「OCI JDBC ドライバの使用方法」の推奨事項に従って構成していることを確認してください。

2 フェーズ・コミットとデータ・ソースの使用

Oracle の 2 フェーズ・コミット・コーディネータは、適切なリカバリを使用して 2 フェーズ・コミットを実行する DTC（分散トランザクション・コーディネータ）エンジンです。この 2 フェーズ・コミット・エンジンは、トランザクションの終了時に、確実に全データベースに対するすべての変更をまとめてコミットするか、または完全にロールバックします。2 フェーズ・コミット・エンジンは、グローバル・トランザクションに関係するデータベースの 1 つでも、別のデータベースでもかまいません。複数のデータベースまたは同一データベース内の複数のセッションが 1 つのトランザクションに関係している場合は、2 フェーズ・コミット・コーディネータを指定する必要があります。指定しない場合は、トランザクションをコミットできません。

コミット・コーディネータを指定するには、次の 2 つの方法があります。

- `J2EE_HOME/config` ディレクトリのグローバルな `application.xml` を使用し、すべてのアプリケーションに対して 1 つのコミット・コーディネータを指定します。
- このコミット・コーディネータをアプリケーションごとに、そのアプリケーションの `orion-application.xml` ファイルで上書きします。

次に例を示します。

```
<commit-coordinator>
  <commit-class class="com.evermind.server.OracleTwoPhaseCommitDriver" />
  <property name="datasource" value="jdbc/OracleCommitDS" />
  <property name="username" value="system" />
  <property name="password" value="manager" />
</commit-coordinator>
```

注意：

- `<commit-coordinator>` 要素の password 属性では、パスワードの間接化がサポートされます。詳細は、『Oracle Application Server Containers for J2EE セキュリティ・ガイド』の「パスワード管理」を参照してください。
- 2 フェーズ・コミットを構成できるのは、エミュレートされていないデータ・ソースの場合のみです。データ・ソースのタイプの詳細は、4-2 ページの「データ・ソースのタイプ」を参照してください。

グローバルな `application.xml` ファイルでユーザー名とパスワードを指定すると、これらの値は、`datasource.xml` ファイルの値に優先します。これらの値が NULL の場合は、コミット・コーディネータとの接続に `datasource.xml` ファイル内のユーザー名とパスワードが使用されます。

コミット・コーディネータ（例 : System）との接続に使用されるユーザー名とパスワードには、すべてのトランザクションを実施する権限が必要です。デフォルトでは、`commit-coordinator` は、インストール時に、NULL のユーザー名とパスワードでグローバルな `application.xml` ファイルに指定されます。

2 フェーズ・コミットに関係する各データ・ソースは、`OrionCMTDatasource` データ・ソース・ファイル内に `dblink` 情報を指定している必要があります。この `dblink` は、このデータベースに接続するためにコミット・コーディネータ・データベースで作成された `dblink` の名前です。

たとえば、`db1` がコミット・コーディネータ用のデータベースで、`db2` と `db3` がグローバル・トランザクションに関係している場合は、次の例のように、`db1` データベースに `link2` および `link3` を作成します。

```
connect commit_user/commit_user
create database link link2 using "inst1_db2"; // link from db1 to db2
create database link link3 using "inst1_db3"; // link from db1 to db3;
```

次は、application.xml ファイルに、jdbc/OracleCommitDS というデータ・ソースを定義します。

```
<data-source
    class="com.evermind.sql.OrionCMTDataSource"
    name="OracleCommitDS"
    location="jdbc/OracleCommitDS"
    connection-driver="oracle.jdbc.driver.OracleDriver"
    username="system"
    password="manager"
    url="jdbc:oracle:thin:@//localhost:5521/db1.regress.rdbms.dev.us.oracle.com"
   inactivity-timeout="30"/>
```

次は、グローバル・トランザクションに関係する db2 のデータ・ソース記述です。db1 で作成された link2 が、ここではプロパティとして指定されていることに注意してください。

```
<data-source
     class="com.evermind.sql.OrionCMTDataSource"
     name="OracleDB2"
     location="jdbc/OracleDB2"
     connection-driver="oracle.jdbc.driver.OracleDriver"
     username="scott"
     password="tiger"
     url="jdbc:oracle:thin:@//localhost:5521/db2.regress.rdbms.dev.us.oracle.com"
     inactivity-timeout="30">
     <property name="dblink"
               value="LINK2.REGRESS.RDBMS.EXAMPLE.COM"/>
</data-source>
```

次は、グローバル・トランザクションに関係する db3 のデータ・ソース記述です。db1 で作成された link3 が、ここではプロパティとして指定されていることに注意してください。

```
<data-source
     class="com.evermind.sql.OrionCMTDataSource"
     name="OracleDB3"
     location="jdbc/OracleDB3"
     connection-driver="oracle.jdbc.driver.OracleDriver"
     username="scott"
     password="tiger"
     url="jdbc:oracle:thin:@//localhost:5521/db3.regress.rdbms.dev.us.oracle.com"
     inactivity-timeout="30">
     <property name="dblink"
               value="LINK3.REGRESS.RDBMS.EXAMPLE.COM"/>
</data-source>
```

2 フェーズ・コミットの制限事項の詳細は、第 7 章「Java Transaction API」を参照してください。

Oracle JDBC の拡張機能の使用方法

Oracle JDBC の拡張機能を使用するには、次のように、戻された接続を oracle.jdbc.OracleConnection にキャストします。

```
Context ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup("jdbc/OracleCMTDS1");
oracle.jdbc.OracleConnection conn =
  (oracle.jdbc.OracleConnection) ds.getConnection();
```

戻された接続 conn で Oracle の拡張機能を使用できます。

```
// you can create oracle.jdbc.* objects using this connection
oracle.jdbc.Statement orclStmt =
  (oracle.jdbc.OracleStatement)conn.createStatement();
// assume table is varray_table
```

```
oracle.jdbc.OracleResultSet rs =
  orclStmt.executeQuery("SELECT * FROM " + tableName);
while (rs.next())
{
  oracle.sql.ARRAY array = rs.getARRAY(1);
...
}
```

接続キャッシング・スキームの使用方法

データ・ソース定義内で使用する接続キャッシング・スキームを定義できます。接続キャッシング・スキームには、DYNAMIC_SCHEME、FIXED_WAIT_SCHEME および FIXED_RETURN_NULL_SCHEME の 3 つのタイプがあります。

キャッシング・スキームを指定するには、cacheScheme という名前の <property> 要素に整数値または文字列値を指定します。表 4-6 に、サポートされる値を示します。

表 4-6 接続キャッシング・スキーム

値	キャッシュ・スキーム
1	DYNAMIC_SCHEME
2	FIXED_WAIT_SCHEME
3	FIXED_RETURN_NULL_SCHEME

注意： この項で説明する接続キャッシュ・スキームは、ネイティブ・データソースにのみ適用されます。他のデータ・ソースには適用されません。

次の例は、DYNAMIC_SCHEME を使用したデータ・ソースです。

```
<data-source
  class="oracle.jdbc.pool.OracleConnectionCacheImpl"
  name="OracleDS"
  location="jdbc/pool/OracleCache"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="scott"
  password="tiger"
  url="jdbc:oracle:thin:@//hostname:TTC port number/SERVICE
  inactivity-timeout="30">
  <property name="cacheScheme" value="1" />
</data-source>
```

この例では、<property name> 要素に **value="DYNAMIC_SCHEME"** を指定することもできます。

data-sources.xml でデータ・ソースを作成する場合は、次の点に注意してください。すなわち、class が oracle.jdbc.pool.OracleConnectionCacheImpl に設定されているとき、ejb-location、xa-location および pooled-location の各属性を指定しないでください。location 属性のみを指定します。JNDI で他の属性を使用してデータ・ソースにアクセスすると、データベースが停止した場合にキャッシュされていた接続の予期しないクリーン・アップが発生します。

OCI JDBC ドライバの使用方法

この章の Oracle データ・ソース定義の例では、Oracle JDBC シン・ドライバが使用されています。ただし、Oracle JDBC OCI ドライバを使用することもできます。OC4J サーバーを起動する前に、次の作業を実行します。

- OC4J がインストールされているシステムに Oracle Client をインストールします。
- ORACLE_HOME 変数を設定します。
- LD_LIBRARY_PATH（または使用している OS の対応する環境変数）に $ORACLE_HOME/lib を設定します。
- TNS_ADMIN に、有効な tnsnames.ora ファイルが含まれている有効な Oracle 管理ディレクトリを設定します。

<data-source> 要素定義の url 属性で使用する URL は、次のいずれかの形式です。

- jdbc:oracle:oci:@

  この TNS エントリは、クライアントと同じシステム上にあるデータベース用で、クライアントは IPC モードでデータベースに接続します。

- jdbc:oracle:oci:@*TNS_service_name*

  TNS サービス名は、インスタンス tnsnames.ora ファイルのエントリです。

- jdbc:oracle:oci:@*full_TNS_listener_description*

  TNS の詳細は、『Oracle Net Services 管理者ガイド』を参照してください。

Oracle Application Server での Oracle JDBC-OCI ドライバのアップグレードに関する注意事項

クライアント・ライブラリの互換性に制限があることから、任意のバージョンの Oracle JDBC-OCI ドライバにアップグレードすることはできません。サポートされるのは、Oracle Application Server 10*g* リリース 2（10.1.2）内にインストールされている Oracle Client ライブラリと一致する OCI ドライバ・バージョンへのアップグレードです。たとえば、Oracle JDBC 10.1.x ドライバはサポートされますが、Oracle JDBC 9.2.x ドライバはサポートされません。

Oracle Application Server で JDBC-OCI の使用をサポートする場合、ORACLE_HOME およびライブラリ・パスの適切な値を起動環境に伝播するため、各 OC4J インスタンス用の opmn.xml エントリも必要です。

環境変数 ORACLE_HOME は、すべてのプラットフォームに共通ですが、ライブラリ・パスを指定する環境変数の名前は、次のようにオペレーティング・システムごとに異なります。

- LD_LIBRARY_PATH（Solaris）
- SLIB_PATH（AIX）
- SHLIB_PATH（HP-UX）
- PATH（Windows）

opmn.xml でライブラリ・パスを指定する一般的な構文は、次のとおりです。

```
<prop name="<LIB_PATH_VARIABLE>" value="<LIB_PATH_VARIABLE_VALUE>"/>
```

ここで、<LIB_PATH_VARIABLE> は、ライブラリ・パスを指定するプラットフォーム固有の適切な変数名で置き換えます。また、<LIB_PATH_VARIABLE_VALUE> は、その変数の値で置き換えます。

次に、Solaris OS の場合の例を示します。

```
    <process-type id="OC4J_SECURITY" module-id="OC4J">
      <environment>
        <variable id="ORACLE_HOME"
  value="/u01/app/oracle/product/inf10120"/>
        <variable
          id="LD_LIBRARY_PATH"
          value="/u01/app/oracle/product/inf10120/lib"
        />
      </environment>
      ...
```

DataDirect JDBC ドライバの使用方法

アプリケーションで異機種間データベースに接続する必要がある場合は、DataDirect JDBC ドライバを使用します。DataDirect JDBC ドライバは、Oracle データベース用ではありませんが、Microsoft、SQLServer、Sybase および DB2 などの Oracle 以外のデータベースに接続する場合に使用できます。OC4J で DataDirect ドライバを使用する場合は、`data-sources.xml` ファイルに、各データベースに対応するエントリを追加します。

DataDirect JDBC ドライバのインストールと設定

『DataDirect Connect for JDBC User's Guide and Reference』の説明に従って DataDirect JDBC ドライバをインストールします。

ドライバのインストール後に、後述の指示に従って設定します。

注意 : 後述の指示では、次の定義に注意してください。

- `OC4J_INSTALL`: スタンドアロン OC4J 環境では、`oc4j_extended.zip` ファイルの解凍先ディレクトリです。Oracle Application Server では、`OC4J_INSTALL` は `ORACLE_HOME` です。
- スタンドアロン OC4J 環境でも Oracle Application Server でも、`DDJD_INSTALL` は DataDirect JDBC ドライバの内容の解凍先ディレクトリです。
- スタンドアロン OC4J 環境では、`INSTANCE_NAME` は `home` です。
- Oracle Application Server では、INSTANCE_NAME は DataDirect JDBC ドライバをインストールする OC4J インスタンスです。

---

1. DataDirect JDBC ドライバの内容を *`DDJD_INSTALL`* ディレクトリに解凍します。
2. *`OC4J_INSTALL`*`/j2ee/`*`INSTANCE_NAME`*`/applib` ディレクトリが存在しない場合は作成します。
3. *`DDJD_INSTALL`*`/lib` にある DataDirect JDBC ドライバを、*`OC4J_INSTALL`*`/j2ee/`*`INSTANCE_NAME`*`/applib` ディレクトリにコピーします。
4. 次のように、`application.xml` ファイルに `j2ee/home/applib` の位置を参照するライブラリ・エントリが含まれていることを確認します。

   ```
   <library path="../../INSTANCE_NAME/applib" />
   ```

5. 4-25 ページの「DataDirect のデータ・ソース・エントリの例」の説明に従って、データ・ソースを `data-source.xml` ファイルに追加します。

DataDirect のデータ・ソース・エントリの例

この項では、次の Oracle 以外の各データベースについてデータ・ソース・エントリの例を示します。

- SQLServer
- DB2
- Sybase

ベンダー固有のデータ・ソースをクラス属性で直接使用することもできます。つまり、クラス属性で OC4J 固有のデータ・ソースを使用する必要はありません。

詳細は、『DataDirect Connect for JDBC User's Guide and Reference』を参照してください。

注意： OC4J は、エミュレートされていない非 Oracle データ・ソースでは動作しません。つまり、2 フェーズ・コミット・トランザクションには非 Oracle データ・ソースを使用できません。

SQLServer

次は、SQLServer のデータ・ソース・エントリの例です。

```
<data-source
   class="com.evermind.sql.DriverManagerDataSource"
   name="MerantDS"
   location="jdbc/MerantCoreSSDS"
   xa-location="jdbc/xa/MerantSSXADS"
   ejb-location="jdbc/MerantSSDS"
   connection-driver="com.oracle.ias.jdbc.sqlserver.SQLServerDriver"
   username="test"
   password="secret"
   url="jdbc:oracle:sqlserver://hostname:port;User=test;Password=secret"
   inactivity-timeout="30"
 />
```

DB2

次は、DB2 データベースのデータ・ソース構成の例です。

```
<data-source
  class="com.evermind.sql.DriverManagerDataSource"
  name="MerantDS"
  location="jdbc/MerantDB2DS"
  xa-location="jdbc/xa/MerantDB2XADS"
  ejb-location="jdbc/MerantDB2DS"
  connection-driver="com.oracle.ias.jdbc.db2.DB2Driver"
  username="test"
  password="secret"
  url="jdbc:oracle:db2://hostname:port;LocationName=jdbc;CollectionId=default;"
  inactivity-timeout="30"
/>
```

Sybase

次は、Sybase データベースのデータ・ソース構成の例です。

```
<data-source
  class="com.evermind.sql.DriverManagerDataSource"
  name="MerantDS"
  location="jdbc/MerantCoreSybaseDS"
  xa-location="jdbc/xa/MerantSybaseXADS"
  ejb-location="jdbc/MerantSybaseDS"
  connection-driver="com.oracle.ias.jdbc.sybase.SybaseDriver"
  username="test"
  password="secret"
  url="jdbc:oracle:sybase://hostname:port;User=test;Password=secret"
  inactivity-timeout="30"
/>
```

データ・ソースの高可用性のサポート

この項では、データ・ソースの高可用性（HA）のサポートについて説明します。

高可用性（HA）アーキテクチャに要求されるのは、すべてのコンポーネントにわたって冗長性を包含し、どのような種類の停止に対しても高速なクライアント・フェイルオーバーを達成し、一貫した高いパフォーマンスを提供し、ユーザー・エラー、破損およびサイトの障害から保護する機能を提供する一方で、容易にデプロイ、管理および拡張ができることです。

Oracle Maximum Availability Architecture（MAA）

Oracle Maximum Availability Architecture（MAA）は、推奨事項と構成指示を提供します。これにより、可用性の要件に最適の Oracle プラットフォーム可用性アーキテクチャを選択して実装できます。

MAA の主要な推奨事項は、次のとおりです。

- 冗長な中間層またはアプリケーション層（Oracle Application Server）、ネットワークおよびストレージ・インフラストラクチャを使用します。
- 人為的エラーやデータ障害から保護し、サイト障害からリカバリするために Oracle Data Guard を使用します。
- ホストおよびインスタンス障害から保護するために、各サイトで Real Application Clusters（RAC）を使用します。
- 安定した運用上のベスト・プラクティス（インスタンス障害からのリカバリ所要時間を制御するファスト・スタート・チェックポイントなど）を使用します。

MMA の詳細は、次の URL を参照してください。

http://www.oracle.com/technology/deploy/availability/htdocs/maa.htm

Oracle Data Guard

Oracle Data Guard は Oracle データベースと統合されたソフトウェアで、スタンバイ・データベースと呼ばれる本番データベースのリアルタイム・コピーを管理し、このインスタンスと冗長データベースとの同期状態を維持します。Oracle Data Guard は、ログ転送サービス、管理されたリカバリ、スイッチオーバーおよびフェイルオーバーの各機能を使用して、この 2 つのデータベースを管理します。

Real Application Clusters（RAC）

RAC は複数のノードまたはコンピュータを使用し、それぞれのノードまたはコンピュータでは、Oracle インスタンスが実行され、共有ディスク記憶域に常駐する 1 つのデータベースにアクセスします。RAC 環境では、すべてのアクティブ・インスタンスが共有データベースに対して同時にトランザクションを実行できます。RAC では、共有データへの各インスタンスのアクセスが自動的に調整され、データ一貫性とデータ整合性が提供されます。

RAC は、次の 2 つのタイプのフェイルオーバー・メカニズムに依存します。

- ネットワーク・フェイルオーバー。ネットワーク層に実装されます。
- 透過的アプリケーション・フェイルオーバー（TAF）。ネットワーク層の最上位に実装されます。

ネットワーク・フェイルオーバー

ネットワーク・フェイルオーバーはデフォルト・フェイルオーバーであり、JDBC シン・ドライバの使用時に使用できるのは、このタイプのフェイルオーバーのみです。ネットワーク・フェイルオーバーにより、RAC クラスタ内のデータベース・インスタンスが停止した後に作成される新しいデータベース接続は、その作成に停止したデータベース・インスタンスの tns 別名が使用されても、同じクラスタ内のバックアップまたは残存データベース・インスタンスに対して作成されることが保証されます。使用可能なフェイルオーバー・メカニズムがネットワーク・フェイルオーバーのみの場合、既存の接続が残存する RAC インスタンスに自動的に再接続されることはありません。既存の接続は使用できなくなり、使用しようとすると ORA-03113 例外が戻されます。ネットワーク・フェイルオーバーのみを実行するように構成されている RAC クラスタでフェイルオーバーが発生すると、進行中のデータベース操作（AQ 操作など）は様々な例外が発生して失敗する可能性があります。

TAF フェイルオーバー

TAF フェイルオーバーを使用できるのは、JDBC OCI ドライバを使用している場合のみです。このタイプのフェイルオーバーを有効化するには、JDBC 接続の作成に使用する TNS 別名の CONNECT_DATA の一部として FAILOVER_MODE を設定する必要があります。

TAF は、RAC や Data Guard などの高可用性環境を対象としたランタイム・フェイルオーバーであり、アプリケーションからサービスへの接続をフェイルオーバーして再確立することを指します。これにより、クライアント・アプリケーションは接続に失敗した場合にデータベースに自動的に再接続でき、進行中だった SELECT 文を必要に応じて再開できます。この再接続は、Oracle Call Interface（OCI）ライブラリ内で自動的に発生します。

TAF は、RAC クラスタに属するデータベース・インスタンスに対して作成されたデータベース接続で進行中の操作について、ベスト・エフォート・フェイルオーバー・メカニズムを提供します。また、既存の接続（フェイルオーバー時に使用中でない接続）をバックアップまたは残存データベース・インスタンスに再接続することを保証しようとします。ただし、最後にコミットされたトランザクションより後に発生したトランザクション操作の場合、TAF が常に再実行できるわけではありません。この動作が発生すると、通常は ORA-25408（「安全にコールを再実行することはできません。」）エラーがスローされます。この場合、データベース接続を再び使用するには、アプリケーションが現行のトランザクションを明示的にロールバックする必要があります。また、アプリケーションは、最後にコミットしたトランザクションより後の操作をすべて再実行して、フェイルオーバー発生前と同じ状態に戻す必要があります。

TAF の保護またはフェイルオーバーの対象となるのは、次のとおりです。

- データベース接続
- ユーザー・セッションの状態
- プリコンパイルされた SQL 文
- 障害時に結果を戻し始めていたアクティブ・カーソル（SELECT 文）

TAF の保護またはフェイルオーバーの対象外となるのは、次のとおりです。

- OCI8 以上を使用していないアプリケーション
- PL/SQL パッケージの状態など、サーバー・サイド・プログラム変数
- アクティブな更新トランザクション（4-31 ページの「TAF 例外の確認」を参照）

OC4J における高可用性（HA）のサポート

OC4J は、HA アーキテクチャの一部として RAC、Data Guard および TAF と統合できます。

この項では、HA に直接関連する OC4J 固有の構成上の問題について説明します。この情報を MAA の推奨事項および手順とともに参考にしてください。

OC4J の HA 構成には、次のような問題点があります。

- OC4J でのネットワーク・フェイルオーバーの構成
- OC4J での透過的アプリケーション・フェイルオーバー（TAF）の構成
- 接続プーリング
- TAF 例外の確認
- SQL 例外処理

OC4J でのネットワーク・フェイルオーバーの構成

ネットワーク・フェイルオーバーを使用するように OC4J を構成する手順は、次のとおりです。

1. `data-sources.xml` でネットワーク・フェイルオーバー対応のデータ・ソースを構成します。

   次に例を示します。

```
<data-source
    class="com.evermind.sql.DriverManagerDataSource"
    name="OracleDS"
    location="jdbc/OracleCoreDS"
    xa-location="jdbc/xa/OracleXADS"
    ejb-location="jdbc/OracleDS"
    connection-driver="oracle.jdbc.driver.OracleDriver"
    username="scott"
    password="tiger"
    url="jdbc:oracle:thin:@(DESCRIPTION=
        (LOAD_BALANCE=on)
        (ADDRESS=(PROTOCOL=TCP) (HOST=host1) (PORT=1521))
        (ADDRESS=(PROTOCOL=TCP) (HOST=host2) (PORT=1521))
        (CONNECT_DATA=(SERVICE_NAME=service_name)))"
    inactivity-timeout="300"
    connection-retry-interval="2"
    max-connect-attempts="60"
    max-connections="60"
    min-connections="12"
/>
```

   この例では、`url` 要素に注意してください。複数のホストが指定されている場合、JDBC クライアントは現行のホストにアクセスできなければ代替ホストの 1 つをランダムに選択します。

   データ・ソース構成の詳細は、4-7 ページの「データ・ソースの定義」を参照してください。

OC4J での透過的アプリケーション・フェイルオーバー（TAF）の構成

TAF を使用するように OC4J を構成する手順は、次のとおりです。

1. 4-30 ページの「TAF 記述子の構成（tnsnames.ora）」の説明に従って TAF 記述子を構成します。

2. `data-sources.xml` で TAF 対応のデータ・ソースを構成します。次に例を示します。

```
<data-source
    class="com.evermind.sql.DriverManagerDataSource"
    name="OracleDS"
    location="jdbc/OracleCoreDS"
    xa-location="jdbc/xa/OracleXADS"
    ejb-location="jdbc/OracleDS"
    connection-driver="oracle.jdbc.driver.OracleDriver"
    username="scott"
    password="tiger"
    url="jdbc:oracle:oci8:@(description=(load_balance=on)(failover=on)
      (address=(protocol=tcp)(host=db-node1)(port=1521))
      (address=(protocol=tcp)(host=db-node2)(port=1521))
      (address=(protocol=tcp)(host=db-node3)(port=1521))
      (address=(protocol=tcp)(host=db-node4)(port=1521))
      (connect_data=
        (service_name=db.us.oracle.com)
        (failover_mode=(type=select)(method=basic)(retries=20)(delay=15))))"
    rac-enabled="true"
    inactivity-timeout="300"
    connection-retry-interval="2"
    max-connect-attempts="60"
    max-connections="60"
    min-connections="12"
/>
```

   この例では、`url` 要素の `failover` が on で、`failover_mode` が定義されていることに注意してください。複数のホストが指定されている場合、JDBC クライアントは現行のホストにアクセスできなければ代替ホストの 1 つをランダムに選択します。`failover_mode` オプションの詳細は、4-30 ページの表 4-7「TAF 構成オプション」を参照してください。

   データ・ソース構成の詳細は、4-7 ページの「データ・ソースの定義」を参照してください。

3. `orion-application.xml` ファイルで、Oracle JMS を JMS 用リソース・プロバイダとして構成します。次に例を示します。

```
<resource-provider
    class="oracle.jms.OjmsContext" name="cartojms1">
    <description> OJMS/AQ </description>
    <property name="datasource" value="jdbc/CartEmulatedDS"></property>
</resource-provider>
```

> **注意：** TAF を使用するように構成できるのは、JDBC OCI クライアントを使用するように構成されているデータ・ソースのみです。

TAF 記述子の構成（tnsnames.ora）

TAF は、`tnsnames.ora` ファイルの Net8 パラメータを使用して構成されます。

接続記述子の CONNECT_DATA セクションに FAILOVER_MODE パラメータを使用して TAF を構成できます。表 4-7 に、TAF がサポートしているサブパラメータを示します。

表 4-7 TAF 構成オプション

サブパラメータ	説明
BACKUP	バックアップ接続用に別のネット・サービス名を指定します。PRECONNECT METHOD を使用して接続を事前確立する場合は、バックアップを指定する必要があります。
TYPE	フェイルオーバーのタイプを指定します。Oracle Call Interface（OCI）アプリケーションには、デフォルトで次の 3 つのタイプの Oracle Net フェイルオーバー機能を使用できます。 ■ SESSION: 設定すると、セッションがフェイルオーバーされます。ユーザーの接続が失われると、バックアップ上でユーザー用に新規セッションが自動的に作成されます。このタイプのフェイルオーバーは、SELECT のリカバリを試行しません。 ■ SELECT: 設定すると、オープン・カーソルを使用するユーザーは障害後も引き続きフェッチできます。ただし、このモードでは、通常の SELECT 操作中にクライアント・サイドにオーバーヘッドが発生します。 ■ NONE: これはデフォルト・タイプです。フェイルオーバー機能は使用されません。フェイルオーバーが発生しないように明示的に指定することもできます。
METHOD	プライマリ・ノードからバックアップ・ノードへのファスト・フェイルオーバーの実行方法を決定します。 ■ BASIC: 設定すると、フェイルオーバー時に接続が確立されます。このオプションでは、フェイルオーバー時までバックアップ・サーバー上ではほとんど処理を必要としません。 ■ PRECONNECT: 設定すると、接続が事前確立されます。この場合、フェイルオーバーは高速になりますが、バックアップ・インスタンスでは、すべてのサポート対象インスタンスからの全接続をサポートできる必要があります。
RETRIES	フェイルオーバー後に接続を試行する回数を指定します。DELAY を指定すると、RETRIES での再試行回数はデフォルトで 5 回になります。 **注意 :** コールバック関数が登録されている場合、このサブパラメータは無視されます。
DELAY	接続試行間で待機する秒数を指定します。RETRIES を指定すると、DELAY はデフォルトで 1 秒になります。 **注意 :** コールバック関数が登録されている場合、このサブパラメータは無視されます。

次の例では、Oracle Net は `sales1-server` または `sales2-server` 上のプロトコル・アドレスの 1 つにランダムに接続します。接続後にインスタンス障害が発生すると、TAF アプリケーションは別のノード上のリスナーにフェイルオーバーします。

```
sales.us.acme.com=
    (DESCRIPTION=
        (LOAD_BALANCE=on)
        (FAILOVER=on)
        (ADDRESS=(PROTOCOL=tcp)(HOST=sales1-server)(PORT=1521))
        (ADDRESS=(PROTOCOL=tcp)(HOST=sales2-server)(PORT=1521))
        (CONNECT_DATA=
            (SERVICE_NAME=sales.us.acme.com)
            (FAILOVER_MODE=
```

```
(TYPE=session)
(METHOD=basic)
(RETRIES=20)
(DELAY=15))))
```

TAF の構成方法の詳細は、『Oracle Net Services 管理者ガイド』を参照してください。

接続プーリング

トランザクションが 2 つの Bean にまたがっている場合に、各 Bean が同じデータベースの異なるインスタンスへの JDBC 接続を取得すると、コミット時に、OC4J は（2 フェーズ・コミットのかわりに）単純コミットを発行するため、そのトランザクションは疑わしいものになります。アプリケーションがこのようなトランザクションを検出した場合、TAF または接続プーリングのどちらか一方を使用します。ただし、両方は使用しません。

インスタンス障害が発生すると、休止接続が OC4J の接続プールと JDBC タイプ 2 の接続プールの両方から消去されます。

データベースが停止して `getConnection()` がコールされると、接続プーリングが使用されていれば、プールがクリーン・アップされます。コール元は `getConnection()` コールでの例外を捕捉して再試行する必要があります。OC4J コンテナがこの再試行を実行する場合があります。

接続に問題があることが検出されると、OC4J は接続プールをクリーン・アップします。つまり、`getConnection()` がエラー・コード 3113 または 3114 とともに `SQLException` をスローする場合です。

ユーザー接続ハンドルを使用中に例外が発生したときに、OC4J で例外の原因がデータベース接続エラーであるかデータベース操作エラーであるかを検出できると役立ちます。接続エラーの発生時にデータベースからスローされる最も一般的なエラー・コードは、3113 および 3114 です。これらのエラー・コードは、通常、削除処理中の接続に関して戻されます。また、新規の接続試行の場合は、エラー・コード 1033、1034、1089 および 1090 を受け取ることがあります。

ファスト接続クリーン・アップは、非 RAC 環境と RAC 環境の両方に実装されます。

非 RAC 環境では、`java.sql.SQLException` がスローされると、未割当ての接続がすべてプールから削除されます。

RAC 環境では、`java.sql.SQLException` がスローされると、最初に未割当ての接続すべての状態がチェックされます。稼働中の接続のみが残ります。それ以外の接続はプールから削除されます。

TAF 例外の確認

TAF はアクティブなトランザクションをフェイルオーバー後は保持できないため、アクティブな更新トランザクションは障害時にロールバックされます。TAF は、障害が発生したアプリケーションからの確認を、ROLLBACK コマンドを介して要求します。つまり、アプリケーションは ROLLBACK を発行するまでエラー・メッセージを受け取ります。

一般的な障害の例を次に示します。

1. JDBC 接続が失敗するか、TAF によりスイッチオーバーされます。
2. TAF が例外を発行します。
3. TAF が、アプリケーションからの ROLLBACK 形式の確認を待機します。
4. アプリケーションがトランザクションをロールバックして再実行します。

Oracle Call Interface（OCI）のコールバックおよびフェイルオーバー・イベントを使用すると、アプリケーションで TAF 操作をカスタマイズして、必要な確認を自動的に提供できます。

アプリケーション（J2EE コンポーネント）では、関数を指定して Oracle インスタンスの障害ステータスを取得し、TAF をカスタマイズできます。OCI ライブラリは、フェイルオーバー中に OCI コールバック機能を使用して、この関数を自動的にコールします。表 4-8 に、OCI API に定義されているフェイルオーバー・イベントを示します。

表 4-8 OCI API のフェイルオーバー・イベント

記号	値	意味
FO_BEGIN	1	失われた接続が検出され、フェイルオーバーが開始されています。
FO_END	2	フェイルオーバーが正常終了しました。
FO_ABORT	3	再試行オプションは指定されておらず、フェイルオーバーに失敗しました。
FO_REAUTH	4	ユーザー・ハンドルが再認証されました。
FO_ERROR	5	フェイルオーバーは一時的に失敗しましたが、アプリケーションはエラーを処理して再試行できる可能性があります。
FO_RETRY	6	フェイルオーバーを再試行します。
FO_EVENT_UNKNOWN	7	不良または不明なフェイルオーバー・イベントです。

詳細は、『Oracle Call Interface プログラマーズ・ガイド』を参照してください。

SQL 例外処理

使用するドライバのタイプに応じて SQL 例外のエラー・コードが異なり、トランザクションの再実行がサポートされる場合とサポートされない場合があります。

これらのエラー・コードは、コール元にスローされる `java.sql.SQLException` で `getErrorCode()` をコールして取得されます。

表 4-9 に、これらの問題をドライバ・タイプ別に示します。

表 4-9 SQL 例外とドライバ・タイプ

ドライバ	エラー・コード	サーブレット層	Session Bean（CMT、BMT）	Entity Bean（CMP）
シン JDBC	17410	再実行が機能します。	再実行が機能します（No activetransaction エラーは無視されます）。	再実行はサポートされません。
OCI	3113、3114	再実行が機能します。	再実行はサポートされません。	再実行はサポートされません。
OCI/TAF		アプリケーションが TAF に確認を送信した後（4-31 ページの「TAF 例外の確認」を参照）、残存ノード上で再実行が機能します。	アプリケーションが TAF に確認を送信した後（4-31 ページの「TAF 例外の確認」を参照）、残存ノード上で再実行が機能します。	アプリケーションが TAF に確認を送信した後（4-31 ページの「TAF 例外の確認」を参照）、OC4J が透過的に続行します。

5 Oracle Remote Method Invocation

この章では、独自の Remote Method Invocation（RMI）/Oracle RMI（ORMI）プロトコルを使用した、OC4J コンテナ間での EJB の相互起動を可能にする、Oracle Application Server Containers for J2EE（OC4J）サポートについて説明します。

この章には、次の項目が含まれます。

- RMI/ORMI の概要
- RMI 用の OC4J の構成
- HTTP を介した ORMI トンネリングの構成

RMI/ORMI の概要

Java Remote Method Invocation（RMI）を使用すると、分散 Java ベース間アプリケーションを作成できます。この種のアプリケーションでは、リモート Java オブジェクトのメソッドを、異なるホスト上にあるものも含め、他の Java 仮想マシン（JVM）から起動できます。

デフォルトでは、OC4J の EJB は Oracle Remote Method Invocation（ORMI）プロトコルを介して RMI コールをやり取りします。ORMI は、OC4J 向けに最適化された Oracle 独自のプロトコルです。

RMI/IIOP を使用するように EJB を変換し、様々な EJB コンテナ間で EJB が相互に起動できるようにすることもできます。第 6 章「J2EE の相互運用性」を参照してください。

> **注意：** OC4J 10*g* リリース 2（10.1.2）の実装では、ロード・バランシングとフェイルオーバーがサポートされるのは ORMI の場合のみで、IIOP の場合はサポートされません。

ORMI 拡張機能

ORMI は OC4J 向けに拡張されており、次の機能が用意されています。

- RMI メッセージ・スループットの増大
- スレッド化のサポートの拡張
- 同じ場所に配置されているオブジェクトのサポート

RMI メッセージ・スループットの増大

ORMI を使用すると、OC4J は高トランザクション・レートで処理できます。これは、Oracle の SpecJ Application Server ベンチマークに反映されています。次の URL を参照してください。

```
http://www.spec.org/
```

一方向 ORMI は、IIOP メッセージよりも小さいメッセージを使用して、このパフォーマンスを達成します。メッセージが小さいため、送受信用の帯域幅が小さく、エンコードとデコードの処理時間が短縮されます。ORMI のメッセージ・サイズは、クライアントとサーバーの間でやり取りされる状態情報の量を最適化することで、さらに小さくなります。ORMI を使用すると、一部の状態はサーバー上でキャッシュされるため、RMI コールのたびに送信する必要がありません。フェイルオーバー時には、クライアント・コードにより新規サーバーに必要な状態情報がすべて再送されるため、これはステートレスであるという RMI 要件に違反しません。

スレッド化のサポートの拡張

ORMI は OC4J のスレッド化モデルと密結合され、そのキューイング、プーリングおよびステージング機能を最大限に利用します。

ORMI はクライアントごとに 1 つずつスレッドを使用します。マルチスレッド化されたクライアントの場合、OC4J は 1 つの接続を介して各コールを多重化します。ただし、OC4J は各コールをシリアライズしないため、複数のスレッドが相互をブロックすることはありません。

この機能により、各クライアント（単一スレッドまたはマルチスレッド）からリモート・サーバーに対する接続は必ず 1 つになります。

同じ場所に配置されているオブジェクトのサポート

同じ場所に配置されているオブジェクトの場合、RMI/ORMI は同じ場所に配置された使用例を検出し、余分で不要なソケット・コールを回避します。

これは、JNDI レジストリが同じ場所に配置されている場合にも当てはまります。

クライアント・サイドの要件

EJB にアクセスするには、クライアント・サイドで次の手順を実行します。

1. `oc4j_client.zip` ファイルを次の URL からダウンロードします。

   `http://www.oracle.com/technology/software/products/ias/devuse.html`

2. クライアント・サイド・ディレクトリ（`d:¥oc4jclient` など）に解凍します。
3. CLASSPATH に `d:¥oc4jclient¥oc4jclient.jar` を追加します。

`oc4j_client.zip` ファイルには、クライアントに必要な JAR ファイル（`oc4jclient.jar` および `optic.jar` など）がすべて含まれています。これらの JAR ファイルには、クライアントの対話に必要なクラスが格納されています。クライアントに必要な他の JAR ファイルはすべて `oc4jclient.jar` のマニフェスト・クラスパスで参照されるため、`oc4jclient.jar` のみを CLASSPATH に追加する必要があります。

このファイルをブラウザにダウンロードする場合は、特定のパーミッションを付与する必要があります。『Oracle Application Server Containers for J2EE Enterprise JavaBeans 開発者ガイド』の「EJB アプリケーションのセキュリティの構成」の「ブラウザにおける権限の付与」を参照してください。

RMI 用の OC4J の構成

OC4J を RMI 用に構成するには、次の 2 つの方法があります。

- Oracle Enterprise Manager 10g を使用した RMI の構成
- 手動による RMI の構成

OC4J の構成には Oracle Enterprise Manager 10*g* を使用することをお薦めします。

OC4J を RMI 用に構成した後、5-7 ページの「RMI 構成ファイル」の説明に従って RMI のプロパティを指定する必要があります。

Oracle Enterprise Manager 10*g* を使用した RMI の構成

次のように Oracle Enterprise Manager 10*g* を使用して、RMI を使用するように OC4J を構成することをお薦めします。

1. RMI を介してアプリケーションにアクセスできるようにする OC4J インスタンスにナビゲートします。

   図 5-1 に、home という OC4J インスタンスを示します。

図 5-1 Oracle Enterprise Manager 10*g* のシステム・コンポーネント

System Components

Enable/Disable Components | Create OC4J Instance

Start | Stop | Restart | Delete OC4J Instance

Select All | Select None

Select	Name	Status	Start Time	CPU Usage (%)	Memory Usage (MB)
☐	home	⇧	May 29, 2003 7:03:58 PM	Unavailable	94.92
☐	HTTP_Server	⇩			
☐	JServ	⇩			
☐	OC4J_EM	⇧	May 28, 2003 1:15:25 PM	0.05	157.68
☐	OID	⇩			
☐	WebCache	⇩			

TIP This table contains only the enabled components of the application server. Only components that have the checkbox enabled can be started or stopped.

Related Links Process Management

2. OC4J インスタンス名をクリックします。
3. 「**管理**」タブをクリックします。
4. 「**サーバー・プロパティ**」をクリックします。
5. Oracle Application Server 環境では、デフォルトで RMI は無効化されています。RMI を有効化するには、図 5-2 のように「RMI ポート」フィールドに値を入力して、OC4J インスタンスごとに一意の RMI ポート（またはポート範囲）を設定します。

図 5-2 Oracle Enterprise Manager 10*g* のサーバー・プロパティのポート構成

6. 「**適用**」をクリックします。
7. ブラウザの「**戻る**」ボタンをクリックします。
8. 「**レプリケーション・プロパティ**」をクリックします。
9. 図 5-3 に示すように「**レプリケート状態**」チェック・ボックスを選択します。

   「**レプリケート状態**」チェック・ボックスの選択を解除すると、「**EJB アプリケーション**」画面の他の属性は無視されます。

図 5-3 Oracle Enterprise Manager 10*g* のレプリケーション・プロパティ

10. 図 5-3 に示すように「**RMI サーバー・ホスト**」フィールドを構成します。

    サーバーが RMI リクエストを受け入れる特定のホスト名または IP アドレスを入力します。OC4J サーバーは、この特定のホストからの RMI リクエストのみを受け入れます。

> **注意：**「レプリケーション・プロパティ」ウィンドウの他の属性は、EJB クラスタリングにのみ適用されます。詳細は、『Oracle Application Server Containers for J2EE Enterprise JavaBeans 開発者ガイド』の「EJB クラスタリング用のマルチキャスト・アドレスの構成」を参照してください。

11. 「**適用**」をクリックします。

手動による RMI の構成

OC4J の構成には Oracle Enterprise Manager 10*g* を使用することをお薦めします。5-3 ページの「Oracle Enterprise Manager 10g を使用した RMI の構成」を参照してください。RMI を手動で構成する方法を選択した場合は、次の手順を実行する必要があります。

1. プロパティ・ファイル `server.xml` を編集します。5-5 ページの「server.xml の編集」を参照してください。
2. 環境に該当する構成ファイルを選択します。
   - OC4J スタンドアロン環境の場合は、`rmi.xml` ファイルのみを編集します（5-5 ページの「rmi.xml の編集」を参照）。
   - Oracle Application Server 環境の場合は、`rmi.xml` ファイル（5-5 ページの「rmi.xml の編集」を参照）および `opmn.xml` ファイル（5-7 ページの「opmn.xml の編集」を参照）の両方を編集します。

注意： Oracle Application Server 環境では、opmn は `opmn.xml` ファイル（5-7 ページの「opmn.xml の編集」を参照）に定義されている RMI ポートの範囲から、OC4J インスタンスごとに RMI ポートを選択します。`rmi.xml` ファイルの `rmi-server` 要素の `port` 属性は無視されます。

Oracle Application Server 環境で構成ファイルを手動で変更した結果は、Oracle Application Server のコマンドラインで `dcmctl updateConfig` コマンドを実行して構成リポジトリを同期化するまで適用されません。

---

server.xml の編集

`server.xml` ファイルの `<rmi-config>` 要素で、RMI 構成ファイルのパス名を指定する必要があります。構文は次のとおりです。

```
<rmi-config path="RMI_PATH" />
```

通常、*`RMI_PATH`* は `./rmi.xml` です。ファイル名は自由に設定できます。

Oracle Application Server 環境の場合にのみ、Oracle Application Server のコマンドラインで次のコマンドを実行して変更を適用します。

```
dcmctl updateConfig
```

rmi.xml の編集

`rmi.xml` ファイルを編集して `rmi-server` 要素を構成し、リモート RMI サーバーへの接続（および RMI サーバーからの接続の受入れ）に使用するホスト、ポート、ユーザー名およびパスワードを指定します。

`rmi.xml` ファイルを構成する手順は、次のとおりです。

1. このローカル RMI サーバー用の `rmi-server` 要素を追加します。

   次に例を示します。

   ```
   <rmi-server host="hostname" port="port">
   </rmi-server>
   ```

   `<rmi-server>` 要素のユーザーが置き換えることができる属性は次のとおりです。

   - *`hostname`*: RMI サーバーが RMI リクエストを受け入れるホスト名または IP アドレス。この属性を省略すると、RMI サーバーはすべてのホストからの RMI リクエストを受け入れます。
   - *`port`*: RMI サーバーが RMI リクエストをリスニングするポート番号。

注意： OC4J スタンドアロン環境では、この属性を省略すると、デフォルトで 23791 に設定されます。

Oracle Application Server 環境では、opmn は `opmn.xml` ファイル（5-7 ページの「opmn.xml の編集」を参照）に定義されている RMI ポートの範囲から、OC4J インスタンスごとに RMI ポートを選択します。`rmi-server` 要素の `port` 属性は無視されます。

---

2. 0（ゼロ）個以上の `server` 要素で、それぞれアプリケーションから RMI 経由で接続できるリモート（Point-to-Point）RMI サーバーを指定して、`rmi-server` 要素を構成します。

   次に例を示します。

```
<rmi-server host="hostname" port="port">
    <server host="serverhostname" username="username" port="serverport"
    password="password"/>
</rmi-server>
```

   `host` 属性は必須、その他の属性はオプションです。`server` 要素のユーザーが置き換えることができる属性は次のとおりです。

   - *`serverhostname`*: RMI サーバーが RMI リクエストをリスニングするホスト名または IP アドレス。
   - *`username`*: リモート RMI サーバー上の有効なプリンシパルのユーザー名。
   - *`serverport`*: リモート RMI サーバーが RMI リクエストをリスニングするポート番号。
   - *`password`*: プリンシパル *username* が使用するパスワード。

3. 0（ゼロ）個以上の `log` 要素で、それぞれ RMI 固有の通知が書き込まれるファイルを指定して、`rmi-server` 要素を構成します。

   たとえば、`file` 要素を使用して次のように記述します。

```
<rmi-server host="hostname" port="port">
    <log>
        <file path="logfilepathname" />
    </log>
</rmi-server>
```

   または、`odl` 要素を使用して次のように記述します。

```
<rmi-server host="hostname" port="port">
    <log>
        <odl path="odlpathname" max-file-size="size" max-num-files="num"/>
    </log>
</rmi-server>
```

   `file` 要素または `odl` 要素のどちらか一方を使用できます（両方は使用できません）。

   `log` 要素のユーザーが置き換えることができる属性は次のとおりです。

   - *`odlpathname`*: この領域に使用するログ・フォルダのパスとフォルダ名。絶対パス、または *`J2EE_HOME`*`/config` ディレクトリへの相対パスを使用できます。このパスは、RMI ログ・ファイルの場所を示します。
   - *`size`*: 各ログ・ファイルの最大バイト数。
   - *`num`*: ログ・ファイルの最大数。
   - *`logfilepathname`*: サーバーがすべての RMI リクエストを書き込むログ・ファイルのパス名（*`logfilepathname`*）。

   `<odl>` 要素は、OC4J 10*g* リリース 2（10.1.2）実装での新規要素です。各 ODL ログ・エントリは、それぞれのログ・ファイルに XML 形式で書き込まれます。ログ・ファイル数には上限があります。この上限に達すると上書きされます。

ODL ロギングを有効化すると、各メッセージはそれぞれのログ・ファイル `logN.xml` に書き込まれます。*N* は 1 から始まる番号です。最初のログ・メッセージはログ・ファイル `log1.xml` に書き込まれます。ログ・ファイルが最大サイズに達すると、第 2 のログ・ファイル `log2.xml` が開いてロギングは続行されます。最後のログ・ファイルがいっぱいになると、最初のログ・ファイル `log1.xml` が消去され、新規メッセージ用に新規のログ・ファイルが開きます。そのため、ログ・ファイルは絶えずロール・オーバーされ、ディスク領域全体を使用することはありません。

ODL ロギングの詳細は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』を参照してください。

4. Oracle Application Server 環境の場合にのみ、Oracle Application Server のコマンドラインで次のコマンドを実行して変更を適用します。

```
dcmctl updateConfig
```

opmn.xml の編集

Oracle Application Server 環境では、`opmn.xml` ファイルを編集し、このローカル RMI サーバーが RMI リクエストをリスニングするポート範囲を指定します。

`opmn.xml` ファイルを構成する手順は、次のとおりです。

1. `opmn.xml` ファイルから抜粋した次の例のように、`port id="rmi"` 要素を使用して `rmi` のポート範囲を構成します。

```
<ias-component id="OC4J">
    <process-type id="home" module-id="OC4J">
        <port id="ajp" range="3301-3400" />
        <port id="rmi" range="3101-3200" />
        <port id="jms" range="3201-3300" />
        <process-set id="default-island" numprocs="1"/>
    </process-type>
</ias-component>
```

`opmn.xml` ファイルの構成方法の詳細は、『Oracle Application Server 管理者ガイド』を参照してください。

2. Oracle Application Server のコマンドラインで次のコマンドを実行して変更を適用します。

```
dcmctl updateConfig
```

RMI 構成ファイル

EJB による通信を可能にするには、表 5-1 に示す構成ファイルのパラメータを構成する必要があります。

表 5-1 RMI 構成ファイル

コンテキスト	ファイル	説明
サーバー	`server.xml`	このファイルの `<sep-config>` 要素は、サーバー拡張プロバイダのプロパティに対するパス名（通常は `internal-settings.xml`）を指定します。 **例：** `<sep-config path="./internal-settings.xml">`
アプリケーション	`jndi.properties`	このファイルでは、クライアントが使用する初期ネーミング・コンテキストの URL を指定します。5-8 ページの「RMI の JNDI プロパティ」を参照してください。

RMI の JNDI プロパティ

この項では、RMI/ORMI に固有の JNDI プロパティについて説明します。詳細は、『Oracle Application Server Containers for J2EE Enterprise JavaBeans 開発者ガイド』の「EJB へのアクセス方法」の「EJB へのアクセス手順」を参照してください。

次の RMI/ORMI プロパティは、jndi.properties ファイルで制御されます。

- java.naming.provider.url（5-8 ページの「プロバイダ URL のネーミング」を参照）
- java.naming.factory.initial（5-10 ページの「コンテキスト・ファクトリの使用」を参照）

プロバイダ URL のネーミング

次の構文を使用して、java.naming.provider.url を設定します。

<prefix>://<host>:<port>:<oc4j_instance>/<application-name>

表 5-2 に、この構文で使用する引数を示します。

表 5-2 プロバイダ URL のネーミング

変数	説明
prefix	Oracle Application Server アプリケーションには opmn:ormi を使用します。 スタンドアロン OC4J アプリケーションには ormi を使用します。 HTTP トンネリングを使用するアプリケーションには http:ormi を使用します（5-12 ページの「HTTP を介した ORMI トンネリングの構成」を参照）。 非 OC4J コンテナと相互運用する必要のあるアプリケーションには corbaname を使用します（6-11 ページの「corbaname の URL」を参照）。
host	Oracle Application Server アプリケーションの場合は、opmn.xml ファイルに定義されている OPMN ホストの名前です。通常、OPMN は OC4J インスタンスと同じノードにありますが、別のマシンにある場合はホスト名を指定します。 スタンドアロン OC4J アプリケーションの場合は、rmi.xml ファイルの rmi-server 要素の host 属性に定義されているポート番号です。
port	Oracle Application Server 10*g* リリース 2（10.1.2）では、opmn:ormi 接頭辞を使用する場合は、request ポートを指定します。opmn プロセスはこのポートでリスニングし、RMI リクエストを該当する OC4J インスタンス用に選択した RMI ポートに転送します（5-9 ページの「opmn リクエスト・ポートの使用」を参照）。この引数を省略すると、デフォルトの request ポート値 6003 が使用されます。 10*g* リリース 2（10.1.2）より前の Oracle Application Server では、ormi 接頭辞を使用する場合は、opmn により OC4J インスタンス用に選択された RMI ポートを指定する必要があります（5-9 ページの「opmnctl を使用した選択済 RMI ポートの表示」を参照）。 スタンドアロン OC4J アプリケーションでは、ormi 接頭辞を使用する場合は、rmi.xml ファイルの rmi-server 要素の port 属性に定義されているポート番号を指定する必要があります。 HTTP トンネリングを使用し、http:ormi 接頭辞を使用するアプリケーションの場合に指定するポートについては、5-12 ページの「HTTP を介した ORMI トンネリングの構成」を参照してください。 非 OC4J コンテナと相互運用する必要があり、corbaname 接頭辞を使用するアプリケーションの場合に指定するポートについては、6-11 ページの「corbaname の URL」を参照してください。

表 5-2 プロバイダ URL のネーミング（続き）

変数	説明
`oc4j_instance`	Oracle Application Server アプリケーションの場合は、Enterprise Manager で定義した OC4J インスタンスの名前です。 スタンドアロン OC4J アプリケーションの場合、この引数は該当しません。
`application-name`	アプリケーション名です。

次に例を示します。

```
java.naming.provider.url=opmn:ormi://localhost:oc4j_inst1/ejbsamples
```

opmn リクエスト・ポートの使用

Oracle Application Server 10*g* リリース 2（10.1.2）では、`opmn.xml` ファイル内で構成されている `notification-server` 要素の `port` 要素の `request` 属性に定義されているポートを指定できます（デフォルト : 6003）。opmn が `request` ポートで RMI リクエストを受信すると、それを該当する OC4J インスタンス用に選択した RMI ポートに転送します。

たとえば、次の `opmn.xml` ファイルの抜粋を考えてみます。

```
<notification-server>
    <port local="6100" remote="6200" request="6004"/>
    <log-file path="$ORACLE_HOME/opmn/logs/ons.log" level="4"
        rotation-size="1500000"/>
    <ssl enabled="true" wallet-file="$ORACLE_HOME/opmn/conf/ssl.wlt/default"/>
</notification-server>
```

この例では、`notification-server` 要素の `port` 要素の `request` 属性に定義されているポートが 6004 であるため、JNDI ネーミング・プロバイダ URL ではポート 6004 が使用されます。

この URL の使用例は、5-11 ページの「Oracle Application Server 10g（9.0.4）以上の OC4J」を参照してください。

opmnctl を使用した選択済 RMI ポートの表示 各 OC4J インスタンス用に opmn で選択された RMI ポートを確認するには、opmn を実行中のホストで次のコマンドを使用します。

```
opmnctl status -l
```

これにより、OC4J インスタンスごとに 1 行を含むデータ表が出力されます。

次に例を示します（読みやすいように一部の列は省略されています）。

```
Processes in Instance: core817.dsunrdb22.us.oracle.com
-------------------+--------------------+-------+ ... +------
ias-component      | process-type       |   pid | ... | ports
-------------------+--------------------+-------+ ... +------
WebCache           | WebCacheAdmin      | 28821 | ... | administration:4000
WebCache           | WebCache           | 28820 | ... |
statistics:4002,invalidation:4001,http:7777
OC4J               | home               |  2012 | ... | iiop:3401,jms:3701,rmi:3201,ajp:3000
HTTP_Server        | HTTP_Server        | 28818 | ... | http2:7200,http1:7778,http:7200
dcm-daemon         | dcm-daemon         | 28811 | ... | N/A
LogLoader          | logloaderd         |   N/A | ... | N/A
```

この表の `ports` 列は、opmn により選択されたポートを示します。次に例を示します。

```
iiop:3401,jms:3701,rmi:3201,ajp:3000
```

この例では、opmn はプロセス ID が 2012 の OC4J インスタンス上の RMI 用にポート 3201 を選択しています。そのため、この OC4J インスタンスには JNDI ネーミング・プロバイダ URL で 3201 をポートとして使用します。

コンテキスト・ファクトリの使用

初期コンテキスト・ファクトリは、クライアント用の初期コンテキスト・クラスを作成します。

`java.naming.factory.initial` プロパティを次のいずれかに設定してください。

- `com.evermind.server.ApplicationClientInitialContextFactory`
- `com.evermind.server.ApplicationInitialContextFactory`
- `com.evermind.server.RMIInitialContextFactory`

`ApplicationClientInitialContextFactory` は、スタンドアロン・アプリケーション・クライアントからリモート・オブジェクトをルックアップするときに使用されます。このコンテキスト・ファクトリは、`application-client.xml` および `orion-application-client.xml` で検出した `refs` および `ref-mappings` を使用します。これは、初期コンテキストが Java アプリケーションでインスタンス化される場合のデフォルトの初期コンテキスト・ファクトリです。

> **注意：** あるアプリケーションの EJB に別のアプリケーションの EJB からアクセスする場合は、RMIInitialContextFactory オブジェクトを使用できません。この場合、これらのアプリケーション間の親と子の関係を使用し、デフォルトの初期コンテキスト・ファクトリ・オブジェクトを使用する必要があります。

`RMIInitialContextFactory` は、ORMI プロトコルを使用して異なるコンテナ間でリモート・オブジェクトをルックアップするときに使用されます。

使用する初期コンテキスト・ファクトリのタイプは、クライアントに応じて異なります。

- クライアントが OC4J コンテナ外部の Pure Java クライアントの場合は、`ApplicationClientInitialContextFactory` クラスを使用します。
- クライアントが OC4J コンテナ内の EJB またはサーブレット・クライアントの場合は、`ApplicationInitialContextFactory` クラスを使用します。これはデフォルト・クラスであるため、初期コンテキスト・ファクトリ・クラスを指定せずに新規 `InitialContext` を作成するたびに、クライアントでは `ApplicationInitialContextFactory` クラスが使用されます。
- クライアントが JNDI ツリーを操作または横断する管理クラスの場合は、`RMIInitialContextFactory` クラスを使用します。
- クライアントが DNS ロード・バランシングを使用する場合は、`RMIInitialContextFactory` クラスを使用します。

ルックアップの例

この項では、EJB のルックアップ方法の例を示します。

- OC4J スタンドアロン
- Oracle Application Server 10g（9.0.4）より前のリリースの OC4J
- Oracle Application Server 10g（9.0.4）以上の OC4J

OC4J スタンドアロン

次の例に、スタンドアロン OC4J インスタンスにデプロイされている J2EE アプリケーション `ejbsamples` 内で `MyCart` という EJB をルックアップする方法を示します。アプリケーションは、RMI ポート 23792 でリスニングするように構成されているノード `localhost` にあります。

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_
FACTORY,"com.evermind.server.rmi.RMIInitialContextFactory");
env.put(Context.SECURITY_PRINCIPAL, "admin");
env.put(Context.SECURITY_CREDENTIALS, "welcome");
```

```
env.put(Context.PROVIDER_URL, "ormi://localhost:23792/ejbsamples");

Context context = new InitialContext(env);
Object homeObject = context.lookup("MyCart");
CartHome home = (CartHome)PortableRemoteObject.narrow(homeObject,CartHome.class);
```

Oracle Application Server 10*g*（9.0.4）より前のリリースの OC4J

Oracle Application Server 環境の OC4J インスタンスの場合、RMI ポートは動的に割り当てられ、`JAZNUserManager` がデフォルトのユーザー・マネージャです。

10*g* リリース 2（10.1.2）より前の Oracle Application Server では、Oracle Application Server 内の EJB にアクセスする場合は、opmn により割り当てられた RMI ポートを把握する必要があります。OC4J インスタンス用の JVM が 1 つのみの場合は、RMI のポート範囲を特定の番号（3101 ～ 3101 など）に限定する必要があります。

次の例に、10*g* リリース 2（10.1.2）より前の Oracle Application Server 環境で J2EE アプリケーション `ejbsamples` 内で `MyCart` という EJB をルックアップする方法を示します。アプリケーションは、RMI ポート 3101 でリスニングするように構成されているノード `localhost` にあります。

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_
FACTORY,"com.evermind.server.rmi.RMIInitialContextFactory");
env.put(Context.SECURITY_PRINCIPAL, "jazn.com/admin");
env.put(Context.SECURITY_CREDENTIALS, "welcome");
env.put(Context.PROVIDER_URL, "ormi://localhost:3101/ejbsamples");

Context context = new InitialContext(env);
Object homeObject = context.lookup("MyCart");
CartHome home = (CartHome)PortableRemoteObject.narrow(homeObject,CartHome.class);
```

Oracle Application Server 10*g*（9.0.4）以上の OC4J

URL に次のタイプのルックアップを使用して Oracle Application Server 環境で EJB をルックアップできます。OC4J インスタンスに割り当てられた RMI ポートを認識する必要はありません。

次の例に、Oracle Application Server 10*g* リリース 2（10.1.2）環境で、J2EE アプリケーション `ejbsamples` 内で `MyCart` という EJB をルックアップする方法を示します。EJB アプリケーションは、ノード `localhost` にあります。この起動とスタンドアロンの起動の違いは、`ormi` に対する opmn 接頭辞、EJB アプリケーションがデプロイされている OC4J インスタンス名 `oc4j_inst1` の指定、および RMI ポートの指定が不要という点です。

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_
FACTORY,"com.evermind.server.rmi.RMIInitialContextFactory");
env.put(Context.SECURITY_PRINCIPAL, "jazn.com/admin");
env.put(Context.SECURITY_CREDENTIALS, "welcome");
env.put(Context.PROVIDER_URL,"opmn:ormi://localhost:oc4j_inst1/ejbsamples");

Context context = new InitialContext(env);
Object homeObject = context.lookup("MyCart");
CartHome home = (CartHome)PortableRemoteObject.narrow(homeObject,CartHome.class);
```

HTTP を介した ORMI トンネリングの構成

EJB では、ファイアウォールを越えて通信するとき、HTTP を介して RMI を送信するトンネリングを使用できます。このトンネリングは、RMI/ORMI でのみサポートされます。RMI/IIOP を使用した HTTP トンネリングは実行できません。

RMI トンネリングをサポートするように OC4J を構成する手順は、次のとおりです。

1. `global-web-application.xml` に次のエントリが存在することを確認します（デフォルト・インストールでは、これらのエントリが事前構成済です）。

```
<servlet>
    <servlet-name>rmi</servlet-name>
    <servlet-class>com.evermind.server.rmi.RMIHttpTunnelServlet
    </servlet-class>
</servlet>
<servlet>
    <servlet-name>rmip</servlet-name>
    <servlet-class>com.evermind.server.rmi.RMIHttpTunnelProxyServlet
    </servlet-class>
</servlet>
```

2. JNDI プロバイダ URL を変更します（5-8 ページの「RMI の JNDI プロパティ」を参照）。OC4J EJB サーバーにアクセスするための JNDI プロバイダの URL の書式は、次のとおりです。

   ormi://*hostname*:*ormi_port*/*appName*

   - リクエストを Oracle Application Server またはスタンドアロン環境で OC4J のホーム・インスタンスに直接トンネリングするには、URL を次のように設定します。

     http:ormi://*hostname*:*http_port*/*appName*

   - Oracle Application Server 環境でのみ OC4J のマウント・ポイントにマップされているインスタンスにリクエストを直接トンネリングするには、*oc4j_mount* を構成し（5-13 ページの「OC4J マウント・ポイントの構成」を参照）、URL を次のように設定します。

     http:ormi://*hostname*:*http_port*/*appName*@*oc4j_mount*

     注意： *http_port* は ORMI ポートではなく HTTP ポートであり（省略した場合はデフォルトで 80 に設定）、*appName* は web-site.xml に定義されているアプリケーション・コンテキストではなくアプリケーション名です。

3. HTTP 通信がプロキシ・サーバーを経由する場合は、コマンドラインで proxyHost および（オプションで）proxyPort を指定し、EJB クライアントを起動します。

   -Dhttp.proxyHost=*proxy_host* -Dhttp.proxyPort=*proxy_port*

   注意： *proxy_port* を省略すると、デフォルトで 80 が使用されます。

OC4J マウント・ポイントの構成

OC4J のマウント・ポイントを使用して、リクエストが OC4J のホーム・インスタンス以外の OC4J インスタンスに直接トンネリングされます。受信リクエストをルーティングするため、Oracle HTTP Server（OHS）では、アプリケーション固有のマウント・ポイントのリストを利用します。これらのマウント・ポイントにより、リクエストで指定された URL が、リクエストを処理する OC4J インスタンスにマップされます。アプリケーション固有のマウント・ポイントのリストは、mod_oc4j 構成ファイルの `mod_oc4j.conf` で定義されます。マウント・ポイントは、通常、アプリケーションのデプロイ時に定義されます。

注意： この機能を使用するには、Web モジュールがデプロイされている必要があります。または、コンテキスト・インスタンスで有効なコンテキスト・ルートを使用する必要があります。プレースホルダ Web モジュールを使用すると、既存の Web モジュール（削除または変更するとクライアント接続に影響を与える可能性のあるモジュール）に対する依存性を除外できます。

次にこの種のマッピングの例を示します。

```
Oc4jMount /xyz inst1
Oc4jMount /xyz/* inst1
```

この例では、OC4J インスタンス `inst1` は、`/xyz` で始まる URL を持つリクエストをすべて受信します。

次の URL は、5-12 ページの「HTTP を介した ORMI トンネリングの構成」の最初の部分に示したものです。

http:ormi://*hostname*:*http_port*/*appName*@*oc4j_mount*

この URL の内容は、次のとおりです。

- *http_port* は、HTTP ポートです。ORMI ポートではありません。
- *appName* は、`default-web-site.xml` の属性 `name` で定義されるアプリケーションの名前です。`default-web-site.xml` の属性 `application` で定義されるアプリケーション・コンテキストではありません。

  `default-web-site.xml` ファイル内の *appName* のコンテキスト・ルートが、`mod-oc4j.conf` ファイル内のコンテキスト・ルートと同じであることを確認してください。この例では、アプリケーション名は demoApp、コンテキスト・ルートは `/xyz` です。このアプリケーションの場合、`default-web-site.xml` ファイル内の実際の行は次のようになります。

  ```
  <default-web-app application="default" name="demoApp" root="/xyz" />
  ```

- *oc4j_mount* は、リクエストのルーティング先となる OC4J インスタンスです。

したがって、この例で、リクエストを `defaultWebApp` から OC4J インスタンス `inst1` に直接トンネリングするには、URL は次のようになります。

http:ormi://*hostname*:*http_port*/defaultWebApp@xyz

`mod-oc4j.conf` ファイルは、Oracle HTTP Server のコンポーネントです。詳細は、『Oracle HTTP Server 管理者ガイド』を参照してください。

`default-web-site.xml` ファイルは OC4J の構成ファイルです。詳細は、『Oracle Application Server Containers for J2EE サーブレット開発者ガイド』を参照してください。

注意： これらの構成ファイルを手動で変更した場合、変更内容は OC4J インスタンスを再起動するまで有効になりません。Application Server Control コンソールで変更を行った場合、変更内容は即座に有効になります。この場合、OC4J インスタンスを再起動する必要はありません。

クラスタ・アプリケーションでの HTTP を介した ORMI の構成

ここでは、この章ですでに説明した情報を参考に、クラスタ・アプリケーションで HTTP を介した ORMI を構成する手順を示します。

1. プレースホルダ Web アプリケーションを作成します。この場合に必要なのは、基本的な web.xml ファイルを備えた `WEB-INF` ディレクトリを含む WAR ファイルのみです。実際の JSP やサーブレットなどのファイルは必要ありません。

2. `web.xml` ファイルに空の `distributable` タグを追加します。次に例を示します。

```
<description> A placeholder web.xml file to be used to implement HTTP tunnel ORMI
connection for the EJB application.</description>
      <distributable />
      ...
```

3. `application.xml` ファイルに Web モジュールを含めます。

```
      <module>
        <web>
          <web-uri>placeholder.war</web-uri>
         <context-root>/tunnel</context-root>
        </web.>
      </module>
```

   ここでは、WAR ファイルが `placeholder.war` であり、コンテキスト・ルートが `/tunnel` であると仮定します。

4. 『Oracle Application Server 高可用性ガイド』の説明に従って、レプリケーション用の OC4J インスタンスを構成します。

   Web アプリケーションと EJB アプリケーションの両セクションを構成します。

5. プレースホルダ WAR ファイルを使用して新しい EAR ファイルをパッケージおよびデプロイします。

   手動構成クラスタを使用する場合は、各 OC4J インスタンスにデプロイする必要があります。

6. クライアントが次の URL を使用することを確認します。

```
http:ormi://host.domain:http_port/<appname>@<contextroot>
```

   `<contextroot>` は、この例の場合は tunnel であり、`<appname>` は、このアプリケーションの EAR ファイルをデプロイしたときに指定した名前です。

これで、フェイルオーバーが適切に動作します。

注意 : アプリケーションにすでに Web モジュールが含まれている場合は、前述の手順に従って、そのモジュールを分散可能とし、そのコンテキスト・ルートを使用するよう構成できます。ただし、この方法の場合、Web モジュールに対する変更がクライアント構成に反映されません。たとえば、コンテキスト・ルートを変更する場合、クライアント接続情報にもその変更を実装する必要があります。

6

J2EE の相互運用性

この章では、標準の Remote Method Invocation（RMI）/Internet Inter-ORB Protocol（IIOP）を使用した、異なるコンテナ間での EJB の相互起動を可能にする、Oracle Application Server Containers for J2EE（OC4J）サポートについて説明します。

この章には、次の項目が含まれます。

- RMI/IIOP の概要
- 相互運用可能トランスポートへの切替え
- 相互運用性のための OC4J の構成

RMI/IIOP の概要

Java Remote Method Invocation（RMI）を使用すると、分散 Java ベース間アプリケーションを作成できます。この種のアプリケーションでは、リモート Java オブジェクトのメソッドを、異なるホスト上にあるものも含め、他の Java 仮想マシン（JVM）から起動できます。

バージョン 2.0 の EJB 仕様では、EJB ベースのアプリケーションが、異なるコンテナ間で別のアプリケーションを簡単に起動する機能が追加されています。既存の EJB を、コード行を変更せずに、Bean のプロパティを編集して再デプロイするのみで相互運用可能にできます。再デプロイの詳細は、6-4 ページの「相互運用可能トランスポートへの切替え」を参照してください。

EJB 相互運用性は、次のもので構成されています。

- トランスポート相互運用性 : CORBA IIOP（Internet Inter-ORB Protocol。ORB は Object Request Broker）を使用。
- ネーミング相互運用性 : CORBA CosNaming Service（OMG CORBA Object Service 仕様の一部である CORBA Object Service Naming）を使用。
- セキュリティ相互運用性 : Common Secure Interoperability Version 2（CSIv2）を使用。
- トランザクション相互運用性 : CORBA Transaction Service（OTS）を使用。

OC4J は、トランスポート相互運用性、ネーミング相互運用性およびセキュリティ相互運用性を提供します。

トランスポート

デフォルトでは、OC4J の EJB は独自のプロトコルである RMI/Oracle Remote Method Invocation（ORMI）を通信に使用します。第 5 章「Oracle Remote Method Invocation」を参照してください。

> **注意 :** OC4J 10*g* リリース 2（10.1.2）の実装では、ロード・バランシングとフェイルオーバーがサポートされるのは ORMI の場合のみで、IIOP の場合はサポートされません。

OC4J では、RMI/IIOP を使用するように EJB を簡単に変換し、様々な EJB コンテナ間で EJB を相互に起動できます。この章では、RMI/IIOP の構成方法と使用方法について説明します。

ネーミング

OC4J は CORBA CosNaming サービスをサポートします。OC4J では、`EJBHome` オブジェクト参照を CosNaming サービスで公開でき、アプリケーションが CORBA を使用して JNDI 名をルックアップできる JNDI CosNaming 実装を提供します。JNDI API または CosNaming API のいずれかを使用してアプリケーションを作成できます。

セキュリティ

OC4J は Common Secure Interoperability Version 2（CSIv2）をサポートします。CSIv2 は、様々な準拠レベルを指定します。OC4J は、準拠レベル 0（ゼロ）を必要とする EJB 仕様に準拠しています。

トランザクション

EJB2.0 仕様では、オプションのトランザクション相互運用性機能が規定されています。仕様に準拠した IQA シートの実装では、次のいずれかを選択する必要があります。

- トランザクション相互運用可能 : 異なる J2EE コンテナでホスティングされている Bean 間でトランザクションがサポートされます。
- トランザクション相互運用不可 : 同じコンテナ内の Bean 間でのみトランザクションがサポートされます。

このリリースの OC4J は、トランザクション相互運用不可です。つまり、トランザクションが複数の EJB コンテナにまたがると、指定の例外が発生します。

クライアント・サイドの要件

EJB にアクセスするには、クライアント・サイドで次の手順を実行する必要があります。

1. `oc4j_client.zip` ファイルを次の URL からダウンロードします。

   `http://www.oracle.com/technology/software/products/ias/devuse.html`

2. クライアント・サイド・ディレクトリ（`d:¥oc4jclient` など）に解凍します。
3. CLASSPATH に `d:¥oc4jclient¥oc4jclient.jar` を追加します。

`oc4j_client.zip` ファイルには、クライアントに必要な JAR ファイル（`oc4jclient.jar` および `optic.jar` など）がすべて含まれています。これらの JAR ファイルには、クライアントの対話に必要なクラスが格納されています。クライアントに必要な他の JAR ファイルはすべて `oc4jclient.jar` のマニフェスト・クラスパスで参照されるため、`oc4jclient.jar` のみを CLASSPATH に追加する必要があります。

このファイルをブラウザにダウンロードする場合は、特定のパーミッションを付与する必要があります。『Oracle Application Server Containers for J2EE Enterprise JavaBeans 開発者ガイド』の「EJB アプリケーションのセキュリティの構成」の「ブラウザにおける権限の付与」を参照してください。

rmic.jar コンパイラ

CORBA オブジェクトの起動または CORBA オブジェクトによる起動のためには、RMI オブジェクトに、対応するスタブ、スケルトンおよび Interface Description Language（IDL）が必要です。`rmic.jar` コンパイラを使用して、Java クラスからスタブとスケルトンを生成するか、IDL を生成します。5-3 ページの「RMI 用の OC4J の構成」を参照してください。

RMI/IIOP で使用する場合は、必ず `-iiop` オプションを使用してコンパイルしてください。

相互運用可能トランスポートへの切替え

OC4J では、EJB は独自のプロトコルである RMI/ORMI を使用して通信します（第 5 章「Oracle Remote Method Invocation」を参照）。RMI/IIOP を使用するように EJB を変換し、EJB コンテナ間で EJB を相互に起動できます。

注意： RMI/IIOP のサポートは CORBA 2.3.1 仕様に基づいています。以前のリリースの CORBA を使用してコンパイルされたアプリケーションは、正しく動作しない可能性があります。

次の 4 つの項では、この変換について詳しく説明します。

スタンドアロン環境での簡易相互運用性

スタンドアロン環境で RMI/IIOP を使用するように EJB を変換する手順は、次のとおりです。

1. `-DGenerateIIOP=true` フラグを指定して OC4J を再起動します。
2. `admin.jar` を使用してアプリケーションをデプロイします。`-iiopClientJar` スイッチを使用して、クライアントのスタブ JAR ファイルを取得する必要があります。次に例を示します。

```
java -jar $J2EE_HOME/admin.jar ormi://localhost admin welcome -deploy -file
filename
 -deploymentName application_name -iiopClientJar stub_jar_filename
```

注意： デプロイするアプリケーションで相互運用性（IIOP）を有効化するために、`-iiopClientJar` スイッチを使用する必要があります。OC4J では、相互運用性はアプリケーションごとに有効化されます。

---

3. `-iiopClientJar` スイッチを指定して `admin.jar` を実行し、クライアントの `classpath` を変更して、デプロイ中に取得されたスタブ JAR ファイルを組み込みます。

   OC4J により生成されたスタブ JAR ファイルのコピーは、次のようにサーバーのデプロイメント・ディレクトリに置かれている場合もあります。

```
application_deployment_directory/appname/ejb_module/module_iiopClient.jar
```

4. `ormi` の URL のかわりに `corbaname` の URL を使用するように、クライアントの JNDI プロパティ `java.naming.provider.url` を編集します。`corbaname` の URL の詳細は、6-11 ページの「corbaname の URL」を参照してください。

注意： 実行時に行われる ORMI スタブの生成とは異なり、IIOP スタブおよび Tie クラス・コードの生成はデプロイ時に行われます。このため、管理者自身が JAR ファイルを `classpath` に追加する必要があります。サーバーで実行している場合は、サーバーと IIOP スタブに必要な生成済クラスのリストが自動的に使用可能になります。

---

5. （オプション）Bean を CORBA アプリケーションでアクセスできるようにするには、`rmic.jar` を実行して、そのインタフェースを記述する IDL を生成します。コマンドライン・オプションの詳細は、6-13 ページの「相互運用性のための OC4J の構成」を参照してください。

スタンドアロン環境での拡張相互運用性

この項では、前項に続いて、スタンドアロン環境で RMI/IIOP を使用するように EJB を変換する方法について説明します。

1. `orion_ejb_jar.xml` および `internal_settings.xml` で、Bean に対する CSIv2 セキュリティ・ポリシーを指定します。これらのセキュリティ関連情報の詳細は、『Oracle Application Server Containers for J2EE セキュリティ・ガイド』を参照してください。
2. `-DGenerateIIOP=true` フラグを指定して OC4J を再起動します。
3. `admin.jar` を使用してアプリケーションをデプロイします。`-iiopClientJar` スイッチを使用して、クライアントのスタブ JAR ファイルを取得する必要があります。次に例を示します。

```
java -jar $J2EE_HOME/admin.jar ormi://localhost admin welcome -deploy -file
filename
 -deployment_name application_name -iiopClientJar stub_jar_filename
```

> **注意：** デプロイするアプリケーションで相互運用性（IIOP）を有効化するために、`-iiopClientJar` スイッチを使用する必要があります。OC4J では、相互運用性はアプリケーションごとに有効化されます。

4. `-iiopClientJar` スイッチを指定して `admin.jar` を実行し、クライアントの `classpath` を変更して、デプロイ中に取得されたスタブ JAR ファイルを組み込みます。

   OC4J により生成されたスタブ JAR ファイルのコピーは、次のようにサーバーのデプロイメント・ディレクトリに置かれている場合もあります。

   *application_deployment_directory/appname/ejb_module/module*_iiopClient.jar

5. `ormi` の URL のかわりに `corbaname` の URL を使用するように、クライアントの JNDI プロパティ `java.naming.provider.url` を編集します。`corbaname` の URL の詳細は、6-11 ページの「corbaname の URL」を参照してください。

> **注意：** 実行時に行われる ORMI スタブの生成とは異なり、IIOP スタブおよび Tie クラス・コードの生成はデプロイ時に行われます。このため、管理者自身が JAR ファイルを `classpath` に追加する必要があります。サーバーで実行している場合は、サーバーと IIOP スタブに必要な生成済クラスのリストが自動的に使用可能になります。

6. （オプション）Bean を CORBA アプリケーションでアクセスできるようにするには、`rmic.jar` を実行して、そのインタフェースを記述する IDL を生成します。コマンドライン・オプションの詳細は、6-13 ページの「相互運用性のための OC4J の構成」を参照してください。

Oracle Application Server 環境での簡易相互運用性

Oracle Application Server 環境で RMI/IIOP を使用して EJB にアクセスするには、次の 2 つの方法があります。

- Oracle Enterprise Manager 10g を使用した相互運用性のための構成
- 相互運用性のための手動による構成

Oracle Enterprise Manager 10*g* を使用した相互運用性のための構成

Oracle Enterprise Manager 10*g* を使用すると、Oracle Application Server 環境で RMI/IIOP を使用してアクセスできるように EJB を構成できます。次の手順を実行してください。

1. RMI/IIOP を介してアプリケーションにアクセスできるようにする OC4J インスタンスにナビゲートします。図 6-1 に、home という OC4J インスタンスを示します。

図 6-1 Oracle Enterprise Manager 10*g* のシステム・コンポーネント

System Components

Enable/Disable Components | Create OC4J Instance

Start | Stop | Restart | Delete OC4J Instance

Select All | Select None

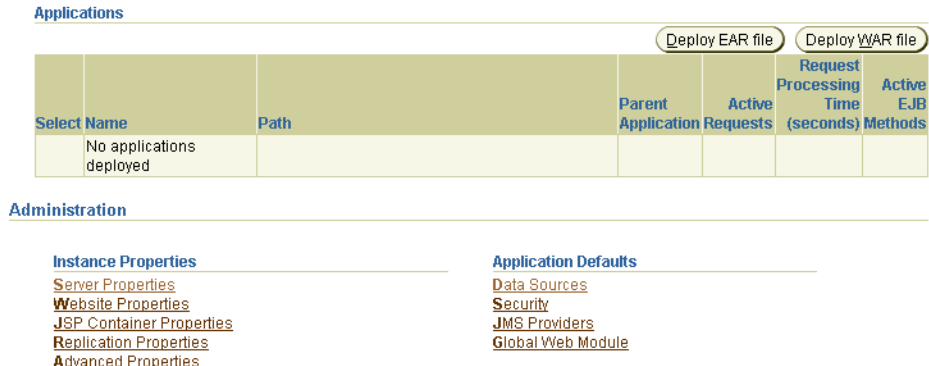
Select	Name	Status	Start Time	CPU Usage (%)	Memory Usage (MB)
☐	home	⇧	May 29, 2003 7:03:58 PM	Unavailable	94.92
☐	HTTP_Server	⇩			
☐	JServ	⇩			
☐	OC4J_EM	⇧	May 28, 2003 1:15:25 PM	0.05	157.68
☐	OID	⇩			
☐	WebCache	⇩			

TIP This table contains only the enabled components of the application server. Only components that have the checkbox enabled can be started or stopped.

Related Links Process Management

2. この OC4J インスタンスの「管理」セクションで「サーバー・プロパティ」をクリックします。図 6-2 に、これを示します。

図 6-2 Oracle Enterprise Manager 10*g* のサーバー・プロパティ



Oracle Application Server 環境では、デフォルトで RMI/IIOP は無効化されています。RMI/IIOP を有効化するには、図 6-3 に示すように「IIOP ポート」フィールドに値を入力して、OC4J インスタンスごとに一意の IIOP ポート（またはポート範囲）が必ず存在するようにします。次に、「適用」をクリックします。

図 6-3 Oracle Enterprise Manager 10*g* のポート構成

Islands

Island ID	Number of Processes
default_island	1

Add Another Row

Related Links: Virtual Machine Metrics

Ports

RMI Ports 3101-3200

JMS Ports 3201-3300

AJP Ports 3000-3100

RMI-IIOP Ports

IIOP Ports 3401-3500

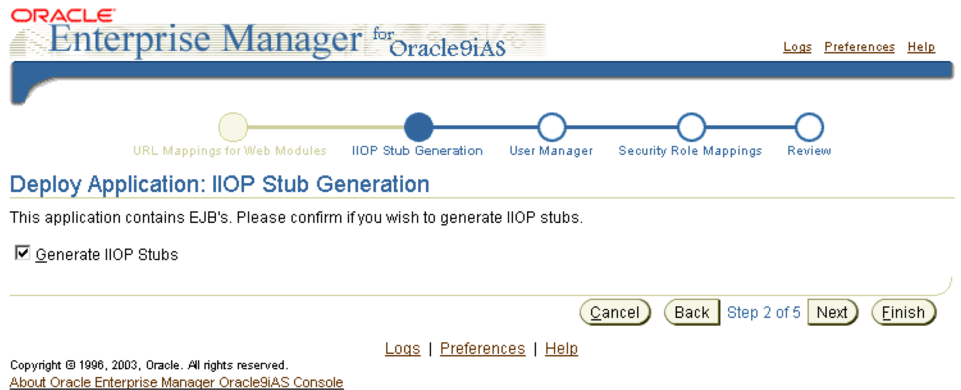
IIOP SSL (Server only)

IIOP SSL (Server and Client)

Oracle Enterprise Manager 10*g* のデプロイメント・ウィザードを使用して、アプリケーションをデプロイします。

図 6-4 に示すように、「IIOP スタブの生成」を選択し、このアプリケーションのクライアント IIOP スタブを生成可能にします。

図 6-4 Oracle Enterprise Manager 10*g* のスタブ生成



Oracle Enterprise Manager 10*g* のデプロイメント・ウィザードを使用して、アプリケーションのデプロイを完了します。

相互運用性のための手動による構成

Oracle Application Server 環境で、RMI/IIOP を使用してリモート・アクセスできるように EJB を手動で構成する手順は、次のとおりです。

1. Oracle Application Server 環境では、デフォルトで RMI/IIOP は無効化されています。RMI/IIOP を有効化するには、OPMN の構成ファイル *J2EE_HOME*/opmn/conf/opmn.xml で、OPMN により管理される OC4J インスタンスごとに一意の IIOP ポート（またはポート範囲）が存在することを確認します。

   注意： 相互運用性を有効化する OC4J インスタンスごとに、IIOP ポート（またはポート範囲）を指定する必要があります。これを指定しないと、OC4J では IIOP リスナーが構成されないため、OC4J の internal-settings.xml ファイルでの構成に関係なく、相互運用性は自動的に無効化されます。

   次に例を示します。

```
<ias-component id="OC4J">
          <process-type id="home" module-id="OC4J">
               <port id="ajp" range="3000-3100"/>
               <port id="rmi" range="23791-23799"/>
               <port id="jms" range="3201-3300"/>
               <port id="iiop" range="3401-3500"/>
               <process-set id="default_island" numprocs="1"/>
          </process-type>
 </ias-component>
```

2. いずれかの構成ファイルを手動で変更した場合は、dcmctl を使用して構成を更新する必要があります。次のコマンドを使用します。

```
dcmctl updateConfig
```

3. opmnctl または Oracle Enterprise Manager 10*g* を使用して、OPMN で管理される OC4J インスタンスをすべて再起動します。

   次のコマンドを使用すると、opmnctl の詳細を表示できます。

```
opmnctl help
```

   OPMN と OPMN で管理されるプロセスすべてを停止して再起動するには、最初に次のコマンドを使用します。

```
opmnctl stopall
```

   その後に、次のコマンドを使用します。

```
opmnctl startall
```

   Oracle Enterprise Manager 10*g* の詳細は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』を参照してください。

4. -enableIIOP オプションを指定して dcmctl を使用し、アプリケーションをデプロイします。次に例を示します。

```
dcmctl deployApplication -f filename -a application_name -enableIIOP
```

5. クライアントの classpath を変更して、OC4J によって生成されたスタブ JAR ファイルを組み込みます。通常、このファイルはサーバーのデプロイメント・ディレクトリにあります。

```
application_deployment_directory/appname/ejb_module/module_iiopClient.jar
```

6. ormi の URL のかわりに OPMN または corbaname の URL を使用するように、クライアントの JNDI プロパティ java.naming.provider.url を編集します。corbaname の URL の詳細は、6-11 ページの「corbaname の URL」を参照してください。OPMN の URL の詳細は、6-12 ページの「OPMN の URL」を参照してください。

> **注意：** 実行時に行われる ORMI スタブの生成とは異なり、IIOP スタブおよび Tie クラス・コードの生成はデプロイ時に行われます。このため、管理者自身が JAR ファイルを classpath に追加する必要があります。サーバーで実行している場合は、サーバーと IIOP スタブに必要な生成済クラスのリストが自動的に使用可能になります。

7. （オプション）Bean を CORBA アプリケーションでアクセスできるようにするには、rmic.jar を実行して、そのインタフェースを記述する IDL を生成します。コマンドライン・オプションの詳細は、6-13 ページの「相互運用性のための OC4J の構成」を参照してください。

Oracle Application Server 環境での拡張相互運用性

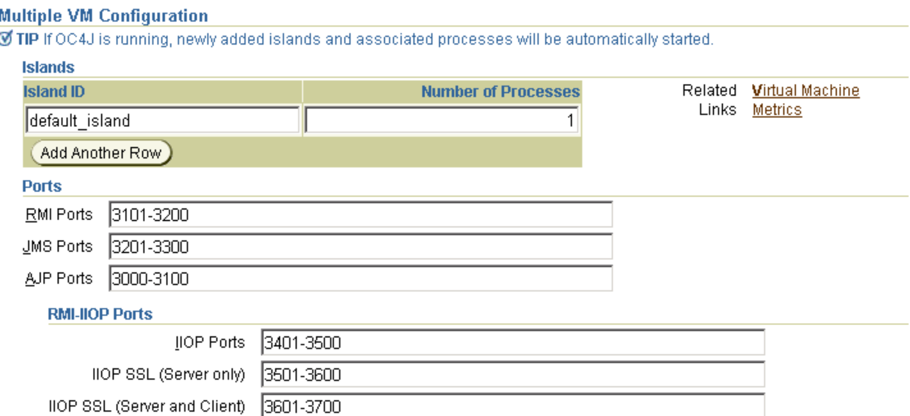
Oracle Application Server 環境で RMI/IIOP を使用して EJB にアクセスするには、次の 2 つの方法があります。

- Oracle Enterprise Manager 10g を使用した相互運用性のための構成
- 相互運用性のための手動による構成

Oracle Enterprise Manager 10*g* を使用した相互運用性のための構成

Oracle Enterprise Manager 10*g* を使用した相互運用性のための拡張構成方法と、6-6 ページの「Oracle Enterprise Manager 10g を使用した相互運用性のための構成」で説明した簡易構成の相違点は、ポート指定のみです。つまり、CSIv2 を使用した相互運用性を有効化する OC4J インスタンスごとに、iiop、iiops1 および iiops2 ポート（またはポート範囲）を指定する必要があります。これを指定しないと、OC4J では IIOP リスナーが構成されないため、OC4J の internal-settings.xml ファイルでの構成に関係なく、相互運用性は自動的に無効化されます。図 6-5 に、これを示します。

図 6-5 Oracle Enterprise Manager 10*g* のポート指定



相互運用性のための手動による構成

この項では、前項に続いて、Oracle Application Server 環境で RMI/IIOP を使用するように EJB を変換する方法について説明します。

1. orion_ejb_jar.xml および internal_settings.xml で、Bean に対する CSIv2 セキュリティ・ポリシーを指定します。これらのセキュリティ関連情報の詳細は、『Oracle Application Server Containers for J2EE セキュリティ・ガイド』を参照してください。

2. Oracle Application Server 環境では、デフォルトで RMI/IIOP は無効化されています。RMI/IIOP を有効化するには、OPMN の構成ファイル *J2EE_HOME*/opmn/conf/opmn.xml で、OPMN により管理される OC4J インスタンスごとに一意の iiop、iiops1 および iiops2 ポート（またはポート範囲）が存在することを確認します。これらは次のポートを意味します。

   iiop: 標準 IIOP ポート

   iiops1: サーバー・サイド認証専用の IIOP/SSL ポート

   iiops2: クライアント・サイドおよびサーバー・サイド認証用の IIOP/SSL ポート

   注意： CSIv2 を使用した相互運用性を有効化する OC4J インスタンスごとに、iiop、iiops1 および iiops2 ポート（またはポート範囲）を指定する必要があります。これを指定しないと、OC4J では IIOP リスナーが構成されないため、OC4J の internal-settings.xml ファイルでの構成に関係なく、相互運用性は自動的に無効化されます。

   次に例を示します。

```
<ias-component id="OC4J">
    <process-type id="home" module-id="OC4J">
        <port id="ajp" range="3000-3100"/>
        <port id="rmi" range="23791-23799"/>
        <port id="jms" range="3201-3300"/>
        <port id="iiop" range="3401-3500"/>
        <port id="iiops1" range="3501-3600"/>
        <port id="iiops2" range="3601-3700"/>
        <process-set id="default_island" numprocs="1"/>
    </process-type>
</ias-component>
```

   注意： OPMN の URL を使用するようにクライアントの JNDI プロパティ java.naming.provider.url を構成すると、OPMN に割り当てられたポートが OC4J にレポートされないため、クライアントは iiops1 または iiops2 ポートに接続できません。

   ---

3. opmnctl または Oracle Enterprise Manager 10*g* を使用して、OPMN で管理される OC4J インスタンスをすべて再起動します。

   次のコマンドを使用すると、opmnctl の詳細を表示できます。

```
opmnctl help
```

   OPMN と OPMN で管理されるプロセスすべてを停止して再起動するには、最初に次のコマンドを使用します。

```
opmnctl stopall
```

その後に、次のコマンドを使用します。

```
opmnctl startall
```

Oracle Enterprise Manager 10*g* の詳細は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』を参照してください。

4. `-enableIIOP` オプションを指定して `dcmctl` を使用し、アプリケーションをデプロイします。次に例を示します。

   ```
   dcmctl  deployApplication -f filename -a application_name -enableIIOP
   ```

5. クライアントの `classpath` を変更して、OC4J によって生成されたスタブ JAR ファイルを組み込みます。通常、このファイルはサーバーのデプロイメント・ディレクトリにあります。

   ```
   application_deployment_directory/appname/ejb_module/module_iiopClient.jar
   ```

6. `ormi` の URL のかわりに `OPMN` または `corbaname` の URL を使用するように、クライアントの JNDI プロパティ `java.naming.provider.url` を編集します。`corbaname` の URL の詳細は、6-11 ページの「corbaname の URL」を参照してください。OPMN の URL の詳細は、6-12 ページの「OPMN の URL」を参照してください。

   注意： 実行時に行われる ORMI スタブの生成とは異なり、IIOP スタブおよび Tie クラス・コードの生成はデプロイ時に行われます。このため、管理者自身が JAR ファイルを `classpath` に追加する必要があります。サーバーで実行している場合は、サーバーと IIOP スタブに必要な生成済クラスのリストが自動的に使用可能になります。

   ---

7. （オプション）Bean を CORBA アプリケーションでアクセスできるようにするには、`rmic.jar` を実行して、そのインタフェースを記述する IDL を生成します。コマンドライン・オプションの詳細は、6-13 ページの「相互運用性のための OC4J の構成」を参照してください。

corbaname の URL

相互運用するためには、EJB は `CosNaming` を使用して他の Bean をルックアップする必要があります。つまり、ルート `NamingContext` をルックアップするための URL には、`ormi` の URL スキームのかわりに、`corbaname` の URL スキームを使用する必要があります。この項では、EJB 開発者が一般的に使用する `corbaname` のサブセットについて説明します。`corbaname` スキームの詳細は、CORBA Naming Service 仕様の第 2.5.3 項を参照してください。`corbaname` スキームは、`corbaloc` スキームに基づいています。これについては、CORBA 仕様の第 13.6.10.1 項で説明されています。

`corbaname` の URL スキームの最も一般的な形式は、次のとおりです。

```
corbaname::host[:port]
```

この `corbaname` の URL によって、従来型の DNS ホスト名または IP アドレスとポート番号が指定されます。次に例を示します。

```
corbaname::example.com:8000
```

`corbaname` の URL では、ホストとポートの後に `#` および文字列表現による `NamingContext` を記述することによって、ネーミング・コンテキストを指定することもできます。指定したホストの `CosNaming` サービスが、ネーミング・コンテキストを解析します。

```
corbaname::host[:port]#namingcontext
```

次に例を示します。

```
corbaname::example.com:8000#Myapp
```

OPMN の URL

この項では、RMI/IIOP 固有の OPMN の URL の詳細を説明します。OPMN の URL の一般情報は、5-8 ページの「RMI の JNDI プロパティ」を参照してください。

Oracle Application Server 環境では、各 Oracle Application Server インスタンス内のすべての OC4J プロセスの IIOP ポートは、OPMN により動的に管理されます。このため、OC4J プロセスがアクティブに IIOP リクエストをリスニングしているポートをクライアントが認識することはできません。クライアントがアクティブな OC4J プロセスすべての IIOP ポートを認識せずに、Oracle Application Server 環境で RMI/IIOP リクエストを正常に発行できるようにするには、次の形式の URL を使用して `jndi.naming.provider.url` プロパティ（クライアントの `jndi.properties` ファイル内）を変更します。

```
opmn:corbaname::opmn_host[:opmn_port]:]:OC4J_instance_name#naming_context
```

次に例を示します。

```
opmn:corbaname::dlsun74:6003:home#stateless
```

注意：

- OC4J 10*g* リリース 2（10.1.2）の実装では、ロード・バランシングとフェイルオーバーがサポートされるのは ORMI の場合のみで、IIOP の場合はサポートされません。
- OPMN の URL を使用すると、クライアントは `iiops1` または `iiops2`（`ssl-port` または `ssl-client-server-auth-port`）ポートに接続できません。

例外マッピング

EJB が IIOP を介して起動されるとき、OC4J はシステム例外を CORBA 例外にマップする必要があります。表 6-1 に、例外マッピングを示します。

表 6-1 Java-CORBA の例外マッピング

OC4J システム例外	CORBA システム例外
`javax.transaction.TransactionRolledbackException`	`TRANSACTION_ROLLEDBACK`
`javax.transaction.TransactionRequiredException`	`TRANSACTION_REQUIRED`
`javax.transaction.InvalidTransactionException`	`INVALID_TRANSACTION`
`java.rmi.NoSuchObjectException`	`OBJECT_NOT_EXIST`
`java.rmi.AccessException`	`NO_PERMISSION`
`java.rmi.MarshalException`	`MARSHAL`
`java.rmi.RemoteException`	`UNKNOWN`

OC4J ホスティング Bean の非 OC4J コンテナからの起動

OC4J でホスティングされていない EJB は、OC4J ホスティング EJB を起動するために、`oc4j_interop.jar` ファイルを `classpath` に追加する必要があります。OC4J では、他のコンテナが、`java:comp/HandleDelegate` において `HandleDelegate` オブジェクトを JNDI ネームスペース内で使用可能にすることを前提としています。`oc4j_interop.jar` ファイルには、ホーム・ハンドル、リモート・ハンドルおよびメタデータ・オブジェクトの標準の移植可能な実装が含まれています。

相互運用性のための OC4J の構成

EJB に相互運用性サポートを追加するには、相互運用性プロパティを指定する必要があります。これらのプロパティの一部は OC4J の起動時に指定され、その他はデプロイメント・ファイルに指定されています。

次のセキュリティ関連の項目は、現在『Oracle Application Server Containers for J2EE セキュリティ・ガイド』に記載されています。

- EJB サーバー・セキュリティのプロパティ（internal-settings.xml）
- CSIv2 セキュリティのプロパティ
- CSIv2 セキュリティのプロパティ（internal-settings.xml）
- CSIv2 セキュリティのプロパティ（ejb_sec.properties）
- CSIv2 セキュリティのプロパティ（orion-ejb-jar.xml）
- EJB クライアント・セキュリティのプロパティ（ejb_sec.properties）

相互運用性 OC4J フラグ

次の OC4J 起動フラグによって、RMI 相互運用性がサポートされます。

- `-DGenerateIIOP=true`: アプリケーションが再デプロイされるたびに、新規のスタブとスケルトンを生成します。
- `-Diiop.debug=true`: デプロイ時のデバッグ・メッセージを生成します。その大部分はコード生成に関連しています。
- `-Diiop.runtime.debug=true`: 実行時のデバッグ・メッセージを生成します。

相互運用性構成ファイル

EJB による通信を可能にするには、表 6-2 に示す構成ファイルのパラメータを構成する必要があります。

表 6-2 相互運用性構成ファイル

コンテキスト	ファイル	説明
サーバー	server.xml	このファイルの <sep-config> 要素は、サーバー拡張プロバイダのプロパティに対するパス名（通常は internal-settings.xml）を指定します。次に例を示します。 <sep-config path="./internal-settings.xml">
	internal-settings.xml	このファイルでは、RMI/IIOP 固有のサーバー拡張プロバイダのプロパティを指定します。これらのセキュリティ関連情報の詳細は、『Oracle Application Server Containers for J2EE セキュリティ・ガイド』を参照してください。
アプリケーション	orion-ejb-jar.xml	<session-deployment> および <entity-deployment> エンティティの <ior-security-config> サブエンティティは、サーバーの Common Secure Interoperability Version 2（CSIv2）セキュリティのプロパティを指定します。 これらのセキュリティ関連情報の詳細は、『Oracle Application Server Containers for J2EE セキュリティ・ガイド』を参照してください。

表 6-2 相互運用性構成ファイル（続き）

コンテキスト	ファイル	説明
	ejb_sec.properties	このファイルでは、EJB に対するクライアント・サイド・セキュリティのプロパティを指定します。これらのセキュリティ関連情報の詳細は、『Oracle Application Server Containers for J2EE セキュリティ・ガイド』を参照してください。
	jndi.properties	このファイルでは、クライアントが使用する初期ネーミング・コンテキストの URL を指定します。詳細は、6-14 ページの「相互運用に関する JNDI プロパティ（jndi.properties）」を参照してください。

相互運用に関する JNDI プロパティ（jndi.properties）

次の RMI/IIOP プロパティは、クライアントの jndi.properties ファイルで制御されます。

- Bean を相互運用可能にするには、java.naming.provider.url に OPMN または corbaname の URL を使用できます。corbaname の URL の詳細は、6-11 ページの「corbaname の URL」を参照してください。OPMN の URL の詳細は、6-12 ページの「OPMN の URL」を参照してください。
- contextFactory は、ApplicationClientInitialContextFactory またはクラス IIOPInitialContextFactory です。

  アプリケーションに application-client.xml ファイルがある場合は、contextFactory を ApplicationClientInitialContextFactory に設定したままにします。アプリケーションに application-client.xml ファイルがない場合は、contextFactory を IIOPInitialContextFactory に変更します。

コンテキスト・ファクトリの使用

com.evermind.server.ApplicationClientInitialContextFactory は、スタンドアロン・アプリケーション・クライアントからリモート・オブジェクトをルックアップするときに使用されます。このコンテキスト・ファクトリは、application-client.xml および orion-application-client.xml で検出した refs および ref-mappings を使用します。これは、初期コンテキストが Java アプリケーションでインスタンス化される場合のデフォルトの初期コンテキスト・ファクトリです。

com.oracle.iiop.server.IIOPInitialContextFactory は、IIOP プロトコルを使用して異なるコンテナ間でリモート・オブジェクトをルックアップするときに使用されます。

OC4J での IIOP の有効化

このチュートリアルでは、OC4J で IIOP アプリケーションを有効化するために必要な手順について説明します。ここでの作業を完了すると、次のことを実行できるようになります。

- IIOP を介したリモート EJB へのアクセス
- IIOP over SSL を使用した EJB 起動の保護
- IIOP over SSL を使用したリモート・クライアントによる corbaname ルックアップの保護

デプロイおよび構成の変更を最小限に抑えるため、このチュートリアルでは、インストールを通じて使用可能になるデモ EJB アプリケーションの helloworld を使用します（このアプリケーションは次のオラクル社の OTN サイトからも入手できます）。

```
http://www.oracle.com/technology/sample_code/tech/java/ejb_corba/
index.html
```

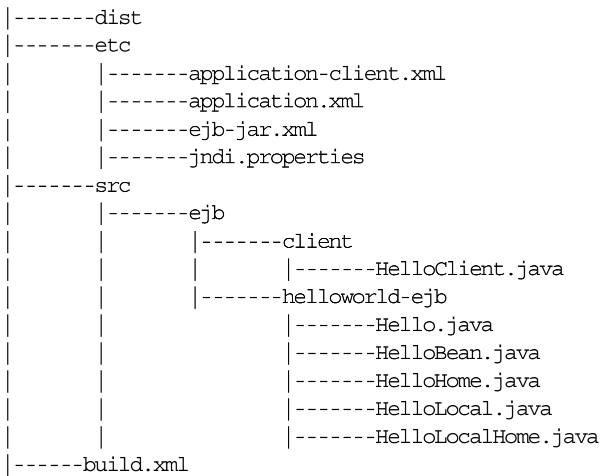
デフォルトの OC4J インストール環境で helloworld アプリケーションをビルドしてデプロイすると、ORMI を介してのみアクセス可能なアプリケーションが生成されます。特定のアプリケーションで IIOP を有効にするには、OC4J のサーバー構成とクライアント・アプリケーションに変更を加える必要があります。必要な変更は、次のとおりです。

- IIOPServerExtensionProvider を構成します。
- java.naming.provider.url を変更します。
- -iiopClientJar 引数を使用してアプリケーションをデプロイします。

次の項では、これらの手順の詳細を説明します。

はじめに

EJB デモを開発環境に展開します。helloworld アプリケーションは、<install-dir>/demo/ejb/helloworld の下に次の構成で存在している必要があります。



```
|-------dist
|-------etc
|       |-------application-client.xml
|       |-------application.xml
|       |-------ejb-jar.xml
|       |-------jndi.properties
|-------src
|       |-------ejb
|       |       |-------client
|       |       |       |-------HelloClient.java
|       |       |-------helloworld-ejb
|       |               |-------Hello.java
|       |               |-------HelloBean.java
|       |               |-------HelloHome.java
|       |               |-------HelloLocal.java
|       |               |-------HelloLocalHome.java
|------build.xml
```

例である helloworld 以外のアプリケーションは、これ以降のチュートリアルでは無視できますが、IIOP を有効化するために必要な変更によって影響を受けないようにする必要があります。このチュートリアルでは、デモがルート・パーティションにインストールされるため、アプリケーションは /demo/ejb/helloworld の下に配置されます。

付属の Ant ビルドファイルにより、src のコンパイル、jar と ear のビルド、およびクライアント・アプリケーション実行のためのターゲットが指定されます。このチュートリアルは、読者が Ant ビルドファイルを熟知していることを前提としています。Ant を熟知していない場合は、Apache の Ant ドキュメント・サイトの http://ant.apache.org/manual/index.html を参照してください。

OC4J での IIOP の構成

次のように server.xml ファイルを編集します。

```
<install-dir>j2ee/home/config/internal-settings.xml
```

server.xml ファイルに次の行が含まれていることを確認します。

```
<sep-config path="./internal-settings.xml" />
```

この行がコメント・アウトされている場合は、コメントを解除します。この行が存在しない場合は、次の行の下に追加してください。

```
<rmi-config path="./rmi.xml" />
```

これにより、OC4J 用の IIOPServerExtensionProvider を構成します。

ここで、次のように internal-settings.xml ファイルを編集し、IIOP 設定を構成します。

```
<install-dir>j2ee/home/config/internal-settings.xml
```

ファイルに次の設定が含まれていることを確認します。

```
<server-extension-provider name="IIOP"
   class="com.oracle.iiop.server.IIOPServerExtensionProvider">
   <sep-property name="port" value="5555" />
   <sep-property name="host" value="localhost" />
   <sep-property name="ssl" value="false" />
   <sep-property name="trusted-clients" value="*" />
</server-extension-provider>
```

必要に応じて、実際の環境に一致するようホストとポートを変更できます。ファイルに SSL のエントリが含まれる場合、一時的にそれらのエントリをコメント・アウトします。

```
<!--
   <sep-property name="ssl-port" value="5556" />
   <sep-property name="ssl-client-server-auth-port" value="5557" />
   <sep-property name="keystore" value="keystore.jks" />
   <sep-property name="keystore-password" value="->pwForSSL" />
   <sep-property name="truststore" value="truststore.jks" />
   <sep-property name="truststore-password" value="->pwForSSL" />
-->
```

これで、IIOP 用に OC4J が構成されました。サーバー・サイドで IIOP を有効化するための最後の手順として、JVM 引数 -DgenerateIIOP=true を使用して OC4J を起動します。この作業は、スタンドアロン OC4J の場合はコマンドラインを使用して、Oracle Application Server インストール環境の場合は ${ORACLE_HOME}/opmn/opmn.xml ファイルを使用して実行できます。

JNDI プロバイダ URL の構成

helloworld アプリケーション用として、次のように jndi.properties ファイルを編集します。

```
<install-dir>/demo/ejb/helloworld/etc/jndi.properties

java.naming.factory.initial=com.evermind.server.ApplicationClientInitialContextFactory
java.naming.provider.url=corbaname:iiop:localhost:5555#helloworld
#java.naming.provider.url=ormi://localhost:23791/helloworld
java.naming.security.principal=admin
java.naming.security.credentials=welcome
```

ORMI プロバイダ URL を含む行はコメント・アウトし、この例での corbaname プロバイダ URL に一致する行を追加します。

アプリケーションのビルドとデプロイ

<install-dir>/demo/ejb/helloworld ディレクトリでデフォルトの Ant ターゲットを実行し、アプリケーションをビルドします。

```
<install-dir>/demo/ejb/helloworld > ant
```

OC4J をまだ起動していない場合は起動し、次のデプロイ・コマンドを実行します。

```
java -jar ${J2EE_HOME}/admin.jar ormi://localhost:23791 admin welcome -deploy -file
dist/helloworld.ear -deploymentName helloworld -iiopClientJar dist/helloworld_iiop_
client.jar
```

これにより、helloworld アプリケーションがデプロイされ、クライアント IIOP スタブを含むクライアント EJB JAR が dest/helloworld_iiop_client.jar に生成されます。

アプリケーションの実行

`<install-dir>/demo/ejb/common.xml` ファイルを編集し、`ORACLE_HOME`、`JAVA_HOME` および `J2EE_HOME` の環境設定が、実際の環境に一致していることを確認します。

`>ant run` を実行します。

クライアント・アプリケーションから `"Hello ..."` という正常なレスポンスが返されます。IIOP を介した通信が動作していることを検証するには、次の JVM 引数をクライアントとサーバーの両方に設定します。

```
-Diiop.runtime.debug=true
```

サーバーでの IIOP over SSL の有効化

`internal-settings.xml` ファイルを編集し、SSL 設定に対するコメントを解除するか、SSL 設定を追加します（次の例に記載された**太字**の行を参照）。

```
<server-extension-provider name="IIOP"
   class="com.oracle.iiop.server.IIOPServerExtensionProvider">
   <sep-property name="port" value="5555" />
   <sep-property name="host" value="localhost" />
   <sep-property name="ssl" value="true" />
   <sep-property name="trusted-clients" value="*" />
   <sep-property name="ssl-port" value="5556" />
   <sep-property name="ssl-client-server-auth-port" value="5557" />
   <sep-property name="keystore" value="keystore.jks" />
   <sep-property name="keystore-password" value="yourPWD" />
   <sep-property name="truststore" value="truststore.jks" />
   <sep-property name="truststore-password" value=" yourPWD " />
</server-extension-provider>
```

必要に応じて、実際の環境に一致するようホストとポートを変更できます。キーストアおよびトラストストアのファイルは、同じ物理ファイルである場合があります。ここに記載されている名前は、例にすぎません。キーストア・ファイルがない場合、次の Sun 社の例に従って keytool を使用できます。

`http://java.sun.com/docs/books/tutorial/security1.2/summary/tools.html`

例の keystore および truststore プロパティに絶対パスとファイル名を追加します。

クライアントでの SSL の有効化

`helloworld` アプリケーション用として、次のように `jndi.properties` ファイルを編集します。

```
<install-dir>/demo/ejb/helloworld/etc/jndi.properties
java.naming.factory.initial=com.evermind.server.ApplicationClientInitialContextFactory
java.naming.provider.url=corbaname:iiop:localhost:5556#helloworld
java.naming.security.principal=admin
java.naming.security.credentials=welcome
```

プロバイダ URL のポートを `internal-settings.xml` の SSL ポートに変更します。

`helloworld` アプリケーション用の `ejb_sec.properties` というファイルを作成します。

```
oc4j.iiop.trustedServers=*
nameservice.useSSL=true
oc4j.iiop.trustStoreLoc=<path to server's keystore>
oc4j.iiop.trustStorePass=<password for server's keystore file>
```

このファイルによって、アプリケーションのセキュリティ要件が OC4J クライアントのブートストラップ・クラスに伝達されます。この例のプロパティでは、EJB ルックアップに SSL を使用し、SSL をサポートするすべてのサーバーを信頼することが示されています。トラストストア設定を使用すると、サーバー・キーストアの証明書をエクスポートして第 2 のトラストストア・ファイルにインポートすることなく、OC4J 用に構成された同一のキーストアを簡単に利用できます。

IIOP over SSL を使用したアプリケーションの実行

`>ant run` を実行します。

クライアント・アプリケーションから `"Hello ..."` という正常なレスポンスが返されます。IIOP over SSL を使用した通信が動作していることを検証するには、クライアントとサーバーの両方に `-Diiop.runtime.debug=true` を設定します。

7 Java Transaction API

この章では、Oracle Application Server Containers for J2EE（OC4J）の Java Transaction API（JTA）について説明します。この章には、次の項目が含まれます。

- 概要
- 1 フェーズ・コミット
- 2 フェーズ・コミット
- タイムアウトの構成
- データベース・インスタンス障害が発生した場合の CMP Bean のリカバリ
- MDB でのトランザクションの使用

概要

アプリケーション・サーバーにデプロイされたアプリケーションは、Java Transaction API（JTA）10.1 を使用してトランザクション境界を設定できます。

たとえば、OC4J コンテナにデプロイされている Bean 管理のトランザクション、サーブレットまたは Java オブジェクトを持つ Enterprise JavaBeans（EJB）は、トランザクションを開始および終了（境界を設定）できます。

この章では、OC4J で JTA を使用する方法について説明します。JTA の概念については説明しません。この章を読むには、グローバル・トランザクションの使用方法とプログラミング方法を理解している必要があります。詳細は、Sun 社の Web サイト（`http://java.sun.com/products/jta`）を参照してください。

コード例は、OTN の OC4J サンプル・コード・サイトからダウンロードできます。

```
http://www.oracle.com/technology/sample_code/tech/java/oc4j/htdocs/
oc4jsamplecode/oc4j-demo-ejb.html
```

JTA では、トランザクション境界設定とリソースの登録が行われます。

トランザクションの境界設定

アプリケーションでは、トランザクション境界が設定されます。Enterprise JavaBeans では、Bean 管理のトランザクションまたはコンテナ管理のトランザクションのいずれかを介したトランザクション管理に JTA 1.0.1 が使用されます。

- Bean 管理のトランザクションは、Bean 実装内でプログラムによって境界設定されます。トランザクション境界は、アプリケーションによって完全に制御されます。
- コンテナ管理のトランザクションは、コンテナによって制御されます。つまり、コンテナは、デプロイメント・ディスクリプタ内の定義に従って、既存のトランザクションと結合するか、アプリケーション用に新しいトランザクションを起動します。新規に作成されたトランザクションは、Bean メソッドが完了すると終了します。トランザクションを管理するために、実装でコードを提供する必要はありません。

注意： すべてのデータ・ソースが JTA トランザクションをサポートするわけではありません。（詳細は、4-16 ページの「データ・ソースの使用方法」を参照してください。）

リソースの登録

トランザクションの複雑さは、アプリケーションでトランザクションに登録しているリソースの数によって決まります。

- 1 フェーズ・コミット（1pc）：トランザクションに登録されているリソース（データベース）が 1 つのみの場合は、1 フェーズ・コミットを使用できます。
- 2 フェーズ・コミット（2pc）：登録されているリソースが 2 つ以上の場合は、構成が複雑な 2 フェーズ・コミットを使用する必要があります。

1 フェーズ・コミット

1 フェーズ・コミット（1pc）は、1 つのリソースのみが含まれるトランザクションです。JTA トランザクションは、リソースの登録およびトランザクション境界設定で構成されます。

単一リソースの登録

1 フェーズ・コミットで単一のリソースを登録するには、次の 2 つの手順を実行する必要があります。

- データ・ソースの構成
- データ・ソースの接続の取得

データ・ソースの構成

1 フェーズ・コミットの場合は、エミュレートされたデータ・ソースを使用します。エミュレートされたデータ・ソース・タイプとエミュレートされていないデータ・ソース・タイプの詳細は、第 4 章「データ・ソース」を参照してください。

可能な場合は、1 フェーズ・コミットの JTA トランザクションにデフォルトのデータ・ソース（`data-sources.xml`）を使用してください。このデータ・ソースは標準の OC4J インストールに付属しています。このデータ・ソースの `url` 属性をデータベース URL 情報で変更した後、`ejb-location` 属性で構成された JNDI 名を指定した JNDI ルックアップを使用して、コードでデータ・ソースを取得します。トランザクションに関係している各データベースに対してデータ・ソースを構成します。

```
<data-source
  class="com.evermind.sql.DriverManagerDataSource"
  name="OracleDS"
  location="jdbc/OracleCoreDS"
  xa-location="jdbc/xa/OracleXADS"
  ejb-location="jdbc/OracleDS"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="scott"
  password="tiger"
  url="jdbc:oracle:thin:@//localhost:1521/ORCL"
  inactivity-timeout="30"
/>
```

必要な属性定義の詳細は、第 4 章「データ・ソース」を参照してください。

データ・ソースの接続の取得

データベース内の表に対して SQL 文を実行する前に、そのデータベースへの接続を取得する必要があります。これらの更新を JTA トランザクションに含めるには、次の 2 つの手順を実行します。

- JNDI ルックアップの実行
- 接続の取得

JNDI ルックアップの実行

トランザクションの開始後に、JNDI ネームスペースからデータ・ソースをルックアップします。データ・ソースを取得するには、次の 2 つの方法があります。

- データ・ソース定義での JNDI ルックアップの実行
- 環境を使用した JNDI ルックアップの実行

データ・ソース定義での JNDI ルックアップの実行 data-sources.xml ファイルのデータ・ソース定義にバインドされている JNDI 名でルックアップを実行して、接続を取得できます。次に例を示します。

```
Context ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup("jdbc/OracleDS");
Connection conn = ds.getConnection();
```

環境を使用した JNDI ルックアップの実行 Bean コンテナの環境で定義された論理名でルックアップを実行できます。詳細は、第 4 章「データ・ソース」を参照してください。ejb-jar.xml または web.xml の J2EE デプロイメント・ディスクリプタで論理名を次のように定義します。

```
<resource-ref>
     <res-ref-name>jdbc/OracleMappedDS</res-ref-name>
     <res-type>javax.sql.DataSource</res-type>
     <res-auth>Container</res-auth>
</resource-ref>
```

OC4J 固有のデプロイメント・ディスクリプタ（orion-ejb-jar.xml など）の <res-ref-name> を、次のように data-sources.xml ファイルでバインドされている JNDI 名にマップします。jdbc/OracleDS は、data-sources.xml ファイルに定義されている JNDI 名です。

```
<resource-ref-mapping name="jdbc/OracleMappedDS" location="jdbc/OracleDS" />
```

その後、環境の JNDI ルックアップを使用してデータ・ソースを取得し、接続を作成します。次に例を示します。

```
InitialContext ic = new InitialContext();
DataSource ds = ic.lookup("java:comp/env/jdbc/OracleMappedDS");
Connection conn = ds.getConnection();
```

次に、データベースに対する JDBC 文の準備および実行を開始します。

接続の取得

getConnection メソッドを使用して、このデータ・ソース・オブジェクトから接続を取得します。この操作には次の 2 つの方法があります。

- ds.getConnection() を使用します。引数は指定しません。
- ds.getConnection(*username*, *password*) を使用します。ユーザー名およびパスワードを指定します。

このメソッドで引数を指定せずに使用するのは、データ・ソース定義に必要なユーザー名とパスワードが含まれている場合です。

データ・ソース定義にユーザー名とパスワードが含まれていない場合、またはデータ・ソースに指定されている以外のユーザー名とパスワードを使用する場合は、他のメソッドを使用します。

トランザクション境界設定

JTA では、Bean が Bean 管理のトランザクションであると指定してトランザクション境界をユーザー自身で設定するか、あるいは Bean がコンテナ管理のトランザクションであると指定して、コンテナでトランザクション境界を設定するように指定できます。コンテナ管理のトランザクションは、すべての EJB で使用できます。ただし、Bean 管理のトランザクションは、Session Bean と MDB に使用できます。

注意： クライアントではトランザクション境界を設定できません。トランザクション・コンテキストは、OC4J インスタンスの境界を超えて伝播できません。したがって、リモート・クライアントまたはリモート EJB のいずれも、トランザクションを開始または結合できません。

Bean デプロイメント・ディスクリプタで境界設定のタイプを指定します。例 7-1 に、`<transaction-type>` 要素を `Container` として定義して、コンテナ管理のトランザクションとして宣言される Session Bean を示します。Bean 管理のトランザクション境界設定を使用するように Bean を構成するには、この要素を `Bean` と定義します。

例 7-1 コンテナ管理のトランザクションとして宣言される Session Bean

```
</session>
  <description>no description</description>
  <ejb-name>myEmployee</ejb-name>
  <home>cmtxn.ejb.EmployeeHome</home>
  <remote>cmtxn.ejb.Employee</remote>
  <ejb-class>cmtxn.ejb.EmployeeBean</ejb-class>
  <session-type>Stateful</session-type>
  <transaction-type>Container</transaction-type>
  <resource-ref>
   <res-ref-name>jdbc/OracleMappedDS</res-ref-name>
   <res-type>javax.sql.DataSource</res-type>
   <res-auth>Application</res-auth>
  </resource-ref>
</session>
```

コンテナ管理のトランザクション境界設定

Bean で CMT を使用するように定義する場合は、デプロイメント・ディスクリプタの `<trans-attribute>` 要素で、この Bean に対する JTA トランザクションをコンテナでどのように管理するかを指定する必要があります（例 7-2 を参照）。表 7-1 に、デプロイメント・ディスクリプタで指定する必要があるトランザクション属性のタイプについて簡単に説明します。

表 7-1 トランザクション属性

トランザクション属性	説明
`NotSupported`	Bean はトランザクションに関連しません。 Bean の実行者が、トランザクションに関連しているときに Bean をコールすると、実行者のトランザクションが一時停止され、Bean が実行されます。Bean が戻ると、実行者のトランザクションが再開されます。
`Required`	Bean はトランザクションに関連している必要があります。 実行者がトランザクションに関連している場合は、実行者のトランザクションが使用されます。 実行者がトランザクションに関連していない場合は、コンテナによってその Bean 用の新規トランザクションが開始されます。これがデフォルト属性です。

表 7-1 トランザクション属性（続き）

トランザクション属性	説明
`Supports`	実行者が関連しているトランザクションの状態にかかわらず、そのトランザクションが Bean に使用されます。 実行者がトランザクションを開始している場合は、実行者のトランザクション・コンテキストが使用されます。 実行者がトランザクションに関連していない場合は、Bean も関連しません。
`RequiresNew`	実行者がトランザクションに関連しているかどうかに関係なく、その Bean に対してのみ存在する新規トランザクションが開始されます。 実行者が、トランザクションに関連しているときに Bean をコールすると、実行者のトランザクションは Bean が完了するまで一時停止されます。
`Mandatory`	この Bean を起動する前に、実行者がトランザクションに関連している必要があります。Bean には、実行者のトランザクション・コンテキストが使用されます。
`Never`	Bean はトランザクションに関連しません。さらに、Bean コール中、実行者がトランザクションに関連することはありません。 実行者がトランザクションに関連している場合は、`RemoteException` がスローされます。

注意： 各種 Entity Bean のデフォルトのトランザクション属性（`<trans-attribute>` 要素）は、次のとおりです。

- CMP 2.0 の Entity Bean の場合、デフォルトは `Required` です。
- MDB の場合、デフォルトは `NotSupported` です。
- 他のすべての Entity Bean の場合、デフォルトは `Supports` です。

例 7-2 に、デプロイメント・ディスクリプタの `<container-transaction>` 部分を示します。この例は、この Bean で、`myEmployee` EJB のすべての（`*`）メソッドに対して `RequiresNew` トランザクション属性を指定する方法を示しています。

例 7-2 デプロイメント・ディスクリプタの <container-transaction>

```
<assembly-descriptor>
   <container-transaction>
      <description>no description</description>
      <method>
         <ejb-name>myEmployee</ejb-name>
         <method-name>*</method-name>
      </method>
     <trans-attribute>RequiresNew</trans-attribute>
   </container-transaction>
</assembly-descriptor>
```

トランザクションの開始、コミットまたはロールバックに Bean 実装は必要ありません。これらの機能はすべて、デプロイメント・ディスクリプタで指定したトランザクション属性に基づいて、コンテナによって処理されます。

Bean 管理のトランザクション

Bean を、<transaction-type> 内で Bean 管理のトランザクション（BMT）として宣言した場合は、Bean 実装によって、グローバル・トランザクションに対する開始、コミットまたはロールバックの境界を設定する必要があります。また、トランザクションの開始前ではなく開始後に、データ・ソース接続を取得する必要があることに注意してください。

プログラムによるトランザクション境界 プログラムによるトランザクション境界の場合、Bean 開発者は、JTA のユーザー・トランザクション・インタフェースまたは JDBC の接続インタフェース・メソッドのいずれかを使用できます。Bean 開発者は、タイムアウト時間内でトランザクションを明示的に開始し、コミットまたはロールバックする必要があります。

Web コンポーネント（JSP、サーブレット）では、プログラムによるトランザクション境界を使用できます。ステートレスおよびステートフル Session Bean ではこの種のトランザクション境界を使用できますが、Entity Bean では使用できないため、宣言的なトランザクション境界を使用する必要があります。

クライアント・サイドのトランザクション境界 J2EE 仕様は、この形式のトランザクション境界を要求していません。また、パフォーマンスおよび待機時間の理由からお薦めできません。OC4J は、クライアント・サイドのトランザクション境界をサポートしていません。

JTA トランザクション

Web コンポーネントまたは Bean 作成者は、UserTransaction インタフェースの開始、コミットおよびロールバックのメソッドを明示的に発行する必要があります。次に例を示します。

```
Context initCtx = new Initial Context();
ut = (UserTransaction)  initCtx.lookup("java:comp/UserTransaction");
…
ut.begin();
// Commit the transaction started in ejbCreate.
Try {
        ut.commit();
} catch (Exception ex) { …..}
```

JDBC トランザクション

java.sql.Connection クラスは、コミットおよびロールバックのメソッドを提供します。JDBC トランザクションは、最新のコミット、ロールバックまたは接続文に続く最初の SQL 文で暗黙的に開始されます。

2 フェーズ・コミット

JTA の主な機能は、単純なトランザクションおよびグローバル・トランザクションを、宣言的にまたはプログラムによって開始および終了することです。グローバル・トランザクションが完了するとき、すべての変更がコミットまたはロールバックされます。2 フェーズ・コミットの実装が難しいのは、構成が細かいためです。2 フェーズ・コミットの場合は、エミュレートされていないデータ・ソースのみを使用する必要があります。エミュレートされていないデータ・ソースの詳細は、4-4 ページの「エミュレートされていないデータ・ソース」を参照してください。

図 7-1 に、2 フェーズ・コミット・エンジン jdbc/OracleCommitDS の例を示します。このエンジンは、グローバル・トランザクションで、2 つのデータベース jdbc/OracleDS1 および jdbc/OracleDS2 を調整します。JTA の 2 フェーズ・コミット環境の構成時に、この例を参照してください。

2 フェーズ・コミット・エンジンの構成

グローバル・トランザクションに複数のデータベースが関連している場合、これらのリソースに対する変更は、同時にすべてコミットまたはロールバックされる必要があります。つまり、トランザクションの終了時に、トランザクション・マネージャはコーディネータ（2 フェーズ・コミット・エンジンとも呼ばれます）と通信し、トランザクションに関連する全データベースに対するすべての変更をコミットまたはロールバックします。2 フェーズ・コミット・エンジンは、次の要素で構成する必要のある Oracle9*i* Database Server データベースです。

- このデータベース（2 フェーズ・コミット・エンジン）から、トランザクションに関連する各データベースへの完全修飾データベース・リンク。トランザクションの終了時に、2 フェーズ・コミット・エンジンは、完全修飾データベース・リンクを介して、関連するデータベースと通信します。
- 関連する各データベースに対するセッションの作成と、コミットまたはロールバックを実行する責任が与えられたユーザー。通信を実行するユーザーは、関連するすべてのデータベース上に作成され、適切な権限を与えられる必要があります。

この調整を容易にするには、データベースと OC4J に対して次の 2 つの項で説明する構成手順を実行します。

データベース構成手順

次の手順に従って、Oracle9*i* Database Server データベースを 2 フェーズ・コミット・エンジンとして指定して構成します。

1. 2 フェーズ・コミット・エンジンでユーザー（`COORDUSR` など）を作成してトランザクションを簡素化し、次の 3 つの操作を実行します。

   a. ユーザーは、2 フェーズ・コミット・エンジンから、関連する各データベースへのセッションをオープンする必要があります。

   b. これらの各データベースに接続できるように、ユーザーに `CONNECT`、`RESOURCE`、`CREATE SESSION` 権限を付与します。また、`FORCE ANY TRANSACTION` 権限を付与すると、ユーザーによるトランザクションのコミットまたはロールバックが可能になります。

   c. トランザクションに関連するすべてのデータベース上にこのユーザーを作成し、これらのパーミッションを付与します。

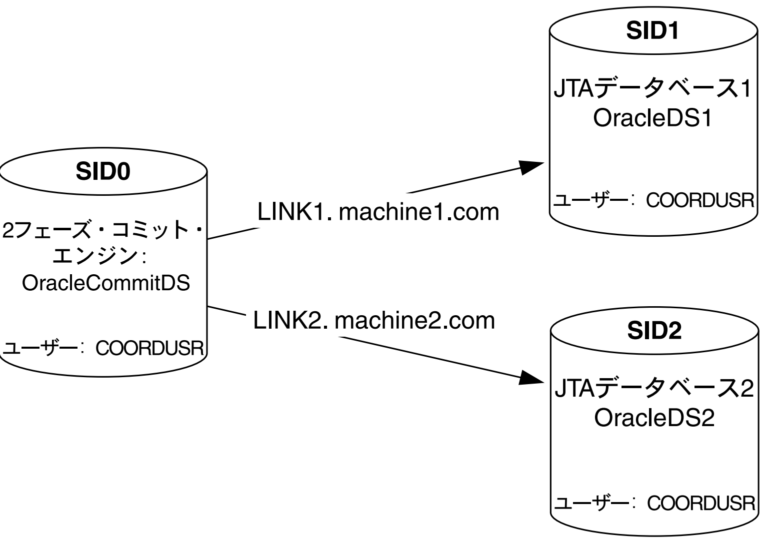
   たとえば、トランザクションの完了に必要なユーザーが `COORDUSR` の場合は、2 フェーズ・コミット・エンジンおよびそのトランザクションに関連する各データベースで次のコマンドを実行します。

   ```
   CONNECT SYSTEM/MANAGER;
   CREATE USER COORDUSR IDENTIFIED BY COORDUSR;
   GRANT CONNECT, RESOURCE, CREATE SESSION TO COORDUSR;
   GRANT FORCE ANY TRANSACTION TO COORDUSR;
   ```

2. 2 フェーズ・コミット・エンジンから、グローバル・トランザクションに関連する各データベースへの完全修飾のパブリック・データベース・リンクを、（`CREATE PUBLIC DATABASE LINK` コマンドを使用して）構成します。この手順は、トランザクション終了時に、2 フェーズ・コミット・エンジンが各データベースと通信するために必要です。`COORDUSR` は、これらのリンクを使用して、関連するすべてのデータベースに接続できる必要があります。

図 7-1 に、トランザクションに関連する 2 つのデータベースを示します。2 フェーズ・コミット・エンジンから各データベースへのデータベース・リンクは、data-sources.xml ファイルの <property> 要素の各 OrionCMTDataSource 定義で提供されます。dblink の <property> 要素については、次の手順を参照してください。

図 7-1 2 フェーズ・コミットの図



OC4J 構成手順

1. 2 フェーズ・コミットの調整を構成するには、まず、2 フェーズ・コミット・エンジンとして機能するデータベースを定義した後、次のように構成します。

data-sources.xml ファイルで、OrionCMTDataSource を使用して、2 フェーズ・コミット・エンジン・データベース用のエミュレートされていないデータ・ソースを定義します。次のコードは、data-sources.xml ファイルに、2 フェーズ・コミット・エンジンの OrionCMTDataSource を定義しています。

```
<data-source
  class="com.evermind.sql.OrionCMTDataSource"
  name="OracleCommitDS"
  location="jdbc/OracleCommitDS"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="coordusr"
  password="coordpwd"
  url="jdbc:oracle:thin:@//localhost:1521/ORCL"
  inactivity-timeout="30"
/>
```

グローバルの application.xml ファイルで、2 フェーズ・コミット・エンジンのデータ・ソースを参照します。このファイルは config ディレクトリにあります。

次のように、2 フェーズ・コミット・エンジンを構成します。

```
<commit-coordinator>
 <commit-class class="com.evermind.server.OracleTwoPhaseCommitDriver" />
 <property name="datasource" value="jdbc/OracleCommitDS" />
 <property name="username" value="coordusr" />
 <property name="password" value="coordpwd" />
</commit-coordinator>
```

注意： `<commit-coordinator>` 要素の password 属性では、パスワードの間接化がサポートされます。詳細は、『Oracle Application Server Containers for J2EE セキュリティ・ガイド』を参照してください。

パラメータは次のように指定します。

`data-sources.xml` ファイルに定義されている `OrionCMTDataSource` に対して、JNDI 名 `"jdbc/OracleCommitDS"` を指定します。この結果、データ・ソースが 2 フェーズ・コミット・エンジンとして使用されるように識別されます。

2 フェーズ・コミット・エンジンのユーザー名とパスワードを指定します。これは、データ・ソース構成でも指定できるオプションの手順です。これらは、2 フェーズ・コミット・エンジンに対するログイン認可として使用するユーザー名とパスワードです。このユーザーに `FORCE ANY TRANSACTION` データベース権限を付与するか、またはすべてのセッション・ユーザーをコミット・コーディネータであるユーザーと同じにする必要があります。

`<commit-class>` を指定します。2 フェーズ・コミット・エンジンの場合、このクラスは常に `OracleTwoPhaseCommitDriver` です。

`OrionCMTDataSource` の JNDI 名は、`name` が `datasource` の `<property>` 要素で識別されます。

ユーザー名は、`<property>` 要素の `username` で識別されます。

パスワードは、`<property>` 要素の `password` で識別されます。

2. グローバル・トランザクションに関係するデータベースを構成するには、データベースごとに次の情報を使用して、タイプ `OrionCMTDataSource` のエミュレートされていないデータ・ソース・オブジェクトを構成します。

   a. オブジェクトにバインドされている JNDI 名。

   b. データベースとの接続を作成するための URL。

   c. 2 フェーズ・コミット・エンジンからこのデータベースへの完全修飾のデータベース・リンク（`LINK1.`*`machine1`*`.COM` など）。このリンクは、`data-sources.xml` ファイルのデータ・ソース定義内の `<property>` 要素で提供されます。

次の `OrionCMTDataSource` オブジェクトは、グローバル・トランザクションに関連する 2 つのデータベースを指定します。各データベースに、2 フェーズ・コミット・エンジンからそのデータベース自体へのデータベース・リンクを示す `dblink` という `<property>` 要素があります。

```
<data-source
  class="com.evermind.sql.OrionCMTDataSource"
  name="OracleCMTDS1"
  location="jdbc/OracleDS1"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="scott"
  password="tiger"
    url="jdbc:oracle:thin:@//localhost:1521/db1.ORCL"
  inactivity-timeout="30">
  <property name="dblink"
   value="LINK1.machine1.COM"/>
</data-source>

<data-source
  class="com.evermind.sql.OrionCMTDataSource"
  name="OracleCMTDS2"
  location="jdbc/OracleDS2"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  username="scott"
  password="tiger"
     url="jdbc:oracle:thin:@//localhost:1521/db2.ORCL"
```

```
    inactivity-timeout="30">
    <property name="dblink"
     value="LINK2.machine2.COM"/>
  </data-source>
```

> **注意：** 2 フェーズ・コミット・エンジンを変更する場合は、すべてのデータベース・リンク（新しい 2 フェーズ・コミット・エンジン内および OrionCMTDataSource の <property> 定義内の両方）を更新する必要があります。

2 フェーズ・コミット・エンジンおよびトランザクションに関連するすべてのデータベースを構成すると、1 フェーズ・コミットと同じ方法でトランザクションを開始および停止できます。詳細は、7-3 ページの「1 フェーズ・コミット」を参照してください。

2 フェーズ・コミット・エンジンの制限事項

OC4J リリースの 2 フェーズ・コミットでは、次の data-sources.xml 構成がサポートされます。

```
<data-source
       class="com.evermind.sql.OrionCMTDataSource"
       location="jdbc/OracleDS"
       connection-driver="oracle.jdbc.driver.OracleDriver"
       username="scott"
       password="tiger"
       url="jdbc:oracle:thin:@//localhost:1521/ORCL
/>
```

2 フェーズ・コミットは、前述のコード例のようにエミュレートされていないデータ・ソース構成でのみ動作します。関連するエミュレートされていないデータ・ソースの URL は、すべて Oracle データベース・インスタンスを指す必要があります。コミット後も ACID（原子性、一貫性、独立性、永続性）のセマンティクスを持つのは、グローバル・トランザクションに関連する複数の Oracle リソースのみです。つまり、2 フェーズ・コミットは Oracle データベース・リソースでのみサポートされますが、完全リカバリは常にサポートされます。

エミュレートされた構成では、2 フェーズ・コミットが動作するように見えますが、リカバリ機能がないためサポートされません。トランザクションの ACID プロパティは保証されず、アプリケーションに問題が発生する可能性があります。

JTA の 2 フェーズ・コミット（2pc）機能は、Oracle9*i* リリース 2（9.2）では動作しません。かわりに、Oracle9*i* リリース 2（9.2.0.4）以上を使用すると、2pc 機能を有効化できます。

タイムアウトの構成

server.xml ファイル内で、timeout 属性を持つ <transaction-config> 要素を使用してタイムアウトを構成できます。この属性では、タイムアウトによりロールバックされる前にトランザクションに許容される完了までの最大時間（ミリ秒）を指定します。デフォルト値は 30000 です。これは、OC4J で開始されるすべてのトランザクションのデフォルト・タイムアウトです。この値は、動的 API の UserTransaction.setTransactionTimeout(*milliseconds*) を使用して変更できます。

サーバーの DTD では、<transaction-config> 要素が次のように定義されます。

```
<!ELEMENT transaction-config (#PCDATA)>
<!ATTLIST transaction-config timeout CDATA #IMPLIED>
```

データベース・インスタンス障害が発生した場合の CMP Bean のリカバリ

バックエンド・データベースに障害が発生した場合は、それを知る必要があります。特に、CMP Bean がトランザクション内で機能している場合は、これが重要になります。データベース・インスタンスに障害が発生すると、障害時に実行しようとしていた操作の再試行が必要になる場合があります。次の項では、CMP Bean がコンテナ管理のトランザクション内にあるか、Bean 管理のトランザクション内にあるかに関係なく、リカバリを実装する方法について説明します。

- コンテナ管理のトランザクションを使用する CMP Bean の接続のリカバリ
- Bean 管理のトランザクションを使用する CMP Bean の接続のリカバリ

コンテナ管理のトランザクションを使用する CMP Bean の接続のリカバリ

コンテナ管理のトランザクションを使用して CMP Bean を定義する場合は、トランザクションの再確立について再試行の回数と間隔を設定できます。これにより、EJB コンテナは、データベース・インスタンスに障害が発生し、トランザクション内でやり取りしている際に接続が停止すると、指定の回数に達するまでデータベースへの新規接続を（指定の間隔で）自動的に取得し、障害が発生した TRY ブロック内で操作を再実行します。

自動再試行回数および間隔を設定するには、CMP Bean の `orion-ejb-jar.xml` ファイルの `<entity-deployment>` 要素内で次のオプション属性を設定します。

- `max-tx-retries`: このパラメータでは、システム・レベルの障害によりロールバックされたトランザクションの再試行回数を指定します。デフォルトは 0（ゼロ）です。
- `tx-retry-wait`: このパラメータでは、トランザクションの再試行間隔を秒単位で指定します。デフォルトは 60 秒です。

Bean 管理のトランザクションを使用する CMP Bean の接続のリカバリ

EJB コンテナは、Bean 管理のトランザクションの CMP Bean または EJB クライアントを管理しません。そのため、JDBC 接続エラーを示す例外を受け取った場合は、それぞれがトランザクション内のメソッドを再試行できるかどうかを認識する必要があります。

再試行できるかどうかを判断するには、データベース接続と SQL 例外を `DbUtil.oracleFatalError()` メソッドのパラメータとして指定します。これにより、新規接続を取得して操作を再試行できるかどうかが判断されます。このメソッドが true を戻したら、トランザクションを続行するために新規接続を作成します。

次のコードに、`DbUtil.oracleFatalError()` メソッドの実行方法を示します。

```
if ((DbUtil.oracleFatalError(sql_ex, db_conn))
{
 //retrieve the database connection again.
 //re-execute operations in the try block where the failure occurred.
}
```

MDB でのトランザクションの使用

BMT トランザクションと CMT トランザクションは、どちらも MDB 内でサポートされます。MDB のデフォルトのトランザクション属性（`trans-attribute`）は `NOT_SUPPORTED` です。

仕様では、MDB は `REQUIRED` 属性および `NOT_SUPPORTED` 属性のみをサポートします。`SUPPORTS` などの他の属性を指定すると、デフォルトの属性 `NOT_SUPPORTED` が使用されます。この場合、エラーはスローされません。

`ejb-jar.xml` ファイルの `<message-driven-deployment>` 要素内には、`transaction-timeout` 属性に定義するように、トランザクションのタイムアウトを定義できます。この属性は、コンテナ管理のトランザクションの MDB のトランザクション・タイムアウト間隔（秒数）を制御します。デフォルトは 1 日（86,400 秒）です。この時間内に完了しないトランザクションはロールバックされます。

OC4J JMS を使用した MDB に対するトランザクションの動作

1 つのトランザクションに異機種または複数のリソースが関連する場合、2 フェーズ・コミットはサポートされません。たとえば、MDB が永続性にデータベースを使用する CMP Bean に通信し、OC4J JMS を介してクライアントからメッセージを受信する場合、この MDB にはデータベースと OC4J JMS という 2 つのリソースが含まれます。この場合、2 フェーズ・コミットはサポートされません。

2 フェーズ・コミットがサポートされない場合、トランザクションのコミット時にすべてのシステムが正常にコミットされる保証はありません。これはロールバックの場合も同様です。2 フェーズ・コミット・エンジンを使用しない場合、ACID 品質のグローバル・トランザクションは保証されません。

Oracle JMS を使用した MDB に対するトランザクションの動作

Oracle JMS は、バックエンドの Oracle データベースをキューおよびトピックのファシリテータとして使用します。Oracle JMS はキューとトピックにデータベース表を使用するため、ユーザーに対する 2 フェーズ・コミットのデータベース権限の付与が必要になる場合があります。

OC4J では 1 フェーズ・コミットが最適化されるため、トランザクションに 2 つのデータベース（または複数のデータ・ソース）が関連する場合以外は、2 フェーズ・コミットを使用する必要はありません。2 フェーズ・コミットの使用は、Oracle JMS 内で完全にサポートされます。

バックエンド・データベースに障害が発生した場合は、それを知る必要があります。特に、MDB Bean がトランザクション内で機能している場合は、これが重要になります。データベース・インスタンスに障害が発生すると、障害時に実行しようとしていた操作の再試行が必要になる場合があります。

次の項では、MDB Bean がコンテナ管理のトランザクション内にあるか、Bean 管理のトランザクション内にあるかに関係なく、リカバリを実装する方法について説明します。

- コンテナ管理のトランザクションを使用する MDB
- Bean 管理のトランザクションを使用する MDB と JMS クライアント

コンテナ管理のトランザクションを使用する MDB

コンテナ管理のトランザクションを使用して MDB を定義する場合は、JMS セッションの再確立について再試行の回数と間隔を設定できます。これにより、データベースとやり取りしている際にトランザクションが失敗すると、コンテナは指定の回数に達するまで（指定の間隔で）自動的に再試行します。自動再試行回数および間隔を設定するには、MDB の `orion-ejb-jar.xml` ファイルの `<message-driven-deployment>` 要素内で次のオプション属性を設定します。

- `dequeue-retry-count`: データベース・フェイルオーバーの発生後に、リスナー・スレッドが新規データベース接続で JMS セッションを再取得しようとする頻度を指定します。デフォルトは 0（ゼロ）です。
- `dequeue-retry-interval`: 再試行間隔を指定します。デフォルトは 60 秒です。

Bean 管理のトランザクションを使用する MDB と JMS クライアント

コンテナは、Bean 管理のトランザクションの MDB または JMS クライアントを管理しません。そのため、JDBC 接続エラーを示す例外を受け取った場合は、それぞれがトランザクション内のメソッドを再試行できるかどうかを認識する必要があります。再試行できるかどうかを判断するには、データベース接続と SQL 例外を `DbUtil.oracleFatalError()` メソッドのパラメータとして入力します。

次のように、JMS セッション・オブジェクトからデータベース接続を取得する必要や、戻された JMS 例外から SQL 例外を取得する必要があります。

1. JMS 例外から基礎となる SQL 例外を取得します。
2. JMS セッションから基礎となるデータベース接続を取得します。
3. `DbUtil.oracleFatalError()` メソッドを実行し、例外が再試行可能なエラーを示しているかどうかを判断します。このメソッドが true を戻したら、JMS アクティビティを続行するために新規 JMS コネクション、セッションおよび可能な場合はセンダーを作成します。

次のコードに、JMS 例外 `jmsexc` を処理して SQL 例外 `sql_ex` を取得する方法を示します。また、データベース接続 `db_conn` が、JMS セッション `session` から取得されます。SQL 例外とデータベース接続は、`DbUtil.oracleFatalError` メソッドの入力パラメータです。

```
try
{
 ..
}
catch(Exception e )
{
 if (exc instanceof JMSException)
 {
  JMSException  jmsexc = (JMSException) exc;
  sql_ex = (SQLException)(jmsexc.getLinkedException());
  db_conn = (oracle.jms.AQjmsSession)session.getDBConnection();

  if ((DbUtil.oracleFatalError(sql_ex, db_conn))
  {
    // Since the DBUtil function returned true, regain the JMS objects
    // Look up the Queue Connection Factory.
    QueueConnectionFactory qcf = (QueueConnectionFactory)
              ctx.lookup ("java:comp/resource/" + resProvider +
              "/QueueConnectionFactories/myQCF");

    // Lookup the Queue.
    Queue queue = (Queue) ctx.lookup ("java:comp/resource/" + resProvider +
                                "/Queues/rpTestQueue");

    // Retrieve a connection and a session on top of the connection.
    // Create queue connection using the connection factory.
    QueueConnection qconn = qcf.createQueueConnection();

    // We're receiving msgs, so start the connection.
    qconn.start();

 // Create a session over the queue connection.
    QueueSession qsess = qconn.createQueueSession(false,
                                               Session.AUTO_ACKNOWLEDGE);

    //Since this is for a queue, create a sender on top of the session.
    //This is used to send out the message over the queue.
    QueueSender snd = sess.createSender (q);

  }
 }
}
```

8 J2EE Connector Architecture（J2CA）

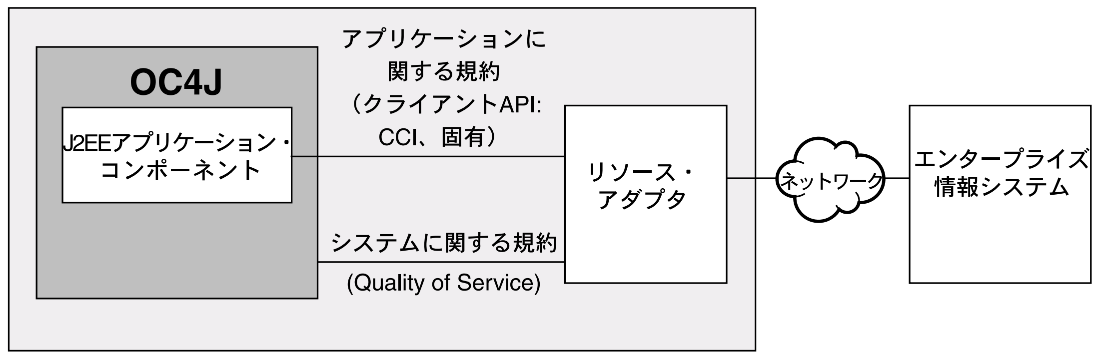
この章では、Oracle Application Server Containers for J2EE（OC4J）アプリケーションにおける J2EE Connector Architecture（J2CA）1.0 の使用方法について説明します。この章には、次の項目が含まれます。

- 概要
- リソース・アダプタのデプロイとアンデプロイ
- Quality of Service に関する規約

概要

J2EE Connector Architecture は、J2EE プラットフォームを異種エンタープライズ情報システム（EIS）に接続するための標準アーキテクチャを定義します。一般的な EIS には、ERP、データベース・システム、メインフレーム・トランザクション処理および Java プログラミング言語で記述されていないレガシー・アプリケーションなどがあります。図 8-1 に、J2EE Connector Architecture を示します。

図 8-1 J2EE Connector Architecture



リソース・アダプタ

リソース・アダプタは、アプリケーション・サーバーまたはアプリケーション・クライアントが、特定の EIS に接続するために使用するドライバです。リソース・アダプタの例としては、リレーショナル・データベース接続用の JDBC ドライバ、ERP システム接続用の ERP リソース・アダプタ、およびトランザクション処理（TP）モニター接続用の TP リソース・アダプタなどがあります。J2EE 1.3 仕様では、アプリケーション・サーバーによってスタンドアロン・リソース・アダプタと埋込みリソース・アダプタの両方がサポートされることが必要です。

スタンドアロン・リソース・アダプタ

他のアプリケーションから独立してアプリケーション・サーバーに直接デプロイできるリソース・アダプタ・モジュールを、スタンドアロン・リソース・アダプタと呼びます。この種のアダプタは、スタンドアロンの Resource Adapter Archive（RAR）ファイルに格納されており、アプリケーション・サーバー・インスタンスにデプロイされているすべてのアプリケーションで使用できます。RAR アーカイブの内容と構造の例は、8-2 ページの「RAR ファイル構造の例」を参照してください。

埋込みリソース・アダプタ

1 つ以上の J2EE モジュールも含む J2EE アプリケーションの一部としてデプロイされているリソース・アダプタ・モジュールを、埋込みリソース・アダプタと呼びます。この種のアダプタは、Enterprise Application Archive（EAR）ファイル内にバンドルされている J2EE アプリケーションでのみ使用できます。

RAR ファイル構造の例

RAR アーカイブの内容と構造の例を次に示します。

```
/META-INF/ra.xml
/META-INF/oc4j-ra.xml
/howto.html
/images/icon.jpg
/ra.jar
/cci.jar
/win.dll
/solaris.so
```

注意： RAR ファイル内で参照される JAR ファイルは、アーカイブ内の任意のディレクトリに格納できます。

注意： 一般に、`/META-INF/oc4j-ra.xml` ファイルは RAR ベンダーから提供される RAR アーカイブの一部ではなく、通常はデプロイ中に OC4J により生成されます。ただし、デプロイヤは、デプロイ前に RAR アーカイブに `oc4j-ra.xml` ファイルを追加するように選択できます。または、生成されたファイルを編集することもできます。

リソース・アダプタによっては、アプリケーションまたはアプリケーション・モジュールから、RAR とバンドルされているアダプタ固有のクラスへのアクセスが必要となる場合があります。スタンドアロン・リソース・アダプタの場合、これらのカスタム・クラスは OC4J 内にデプロイ済のすべてのアプリケーションで使用できます。埋込みリソース・アダプタを使用できるのは、そのアダプタと同じアプリケーションに属するモジュールのみです。

ra.xml ディスクリプタ

`ra.xml` ディスクリプタは、リソース・アダプタ用の標準 J2EE デプロイメント・ディスクリプタです。詳細は、J2EE Connector Architecture 1.0 仕様を参照してください。

アプリケーション・インタフェース

リソース・アダプタによって提供されるクライアント API は、リソース・アダプタのタイプや基礎となる EIS に固有のクライアント API であるか、標準の Common Client Interface（CCI）です。CCI の詳細は、J2EE Connector 1.0 仕様を参照してください。たとえば、JDBC はリレーショナル・データベース・アクセス固有のクライアント API です。

リソース・アダプタでサポートされるクライアント・インタフェースを判断できます。クライアント・インタフェースは、RAR アーカイブにバンドルされている `ra.xml` ファイルの `<connection-interface>` 要素内で指定されています。

Quality of Service に関する規約

J2EE Connector Architecture は、アプリケーション・サーバーと EIS 間の Quality of Service（QoS）に関する次の 3 つの規約を定義しています。

- 接続管理。アプリケーション・コンポーネントが EIS に接続し、アプリケーション・サーバーが提供する接続プーリングを使用できます。8-10 ページの「接続プーリングの構成」も参照してください。

注意： J2EE Connector の接続プーリング・インタフェースは、JDBC インタフェースとは異なります。J2EE Connector の接続プールは JDBC の接続プールと共有されず、一方の接続プールに対して設定されたプロパティが他方に影響を与えることはありません。

- トランザクション管理。アプリケーション・サーバーが、トランザクション・マネージャを使用して複数のリソース・マネージャ間のトランザクションを管理できます。

トランザクション管理には、デプロイ時の構成は必要ありません。詳細は、J2EE Connector 1.0 仕様を参照してください。

オプション機能のサポート

- OC4J は、オプションの接続共有機能（J2EE Connector Architecture 1.0 仕様の第 6.9 項）およびローカル・トランザクションの最適化機能（第 6.12 項）をサポートしていません。
- OC4J は、J2EE Connector Architecture のリソース・アダプタに対する 2 フェーズ・コミットをサポートしていません（2 フェーズ・コミットの制限事項については、第 7 章「Java Transaction API」を参照してください）。
- セキュリティ管理。J2EE サーバーと EIS 間の認証、認可およびセキュアな通信を提供します。8-11 ページの「EIS のサインオンの管理」も参照してください。

すべてのリソース・アダプタが、アプリケーション・サーバーに交換可能となるように、その QoS 規約をサポートする必要があります。

リソース・アダプタのデプロイとアンデプロイ

この項では、リソース・アダプタのデプロイとアンデプロイについて説明します。

デプロイメント・ディスクリプタ

OC4J は、ra.xml、oc4j-ra.xml および oc4j-connectors.xml の 3 つのデプロイメント・ディスクリプタをサポートします。ra.xml ディスクリプタは、常にリソース・アダプタとともに提供されます。oc4j-ra.xml がアーカイブに存在しない場合は、リソース・アダプタをデプロイするたびに OC4J により生成されます。また、埋込みリソース・アダプタの場合は、oc4j-connectors.xml がアーカイブに存在しなければ、OC4J により生成されます。

oc4j-ra.xml ディスクリプタ

oc4j-ra.xml ディスクリプタは、リソース・アダプタに関して OC4J 固有のデプロイメント情報を提供します。このファイルには、1 つ以上の <connector-factory> 要素が含まれています。

oc4j-ra.xml を使用すると、次の操作を実行できます。

- コネクション・ファクトリのインスタンスの構成とバインド

  コネクション・ファクトリは、アプリケーション・コンポーネントで EIS への接続を取得するために使用されます。コネクション・ファクトリ・クラス名は、ra.xml に定義されている connectionfactory-impl-class 要素内で指定されます。OC4J では、デプロイヤがこのクラスのインスタンスを構成し、Java Naming and Directory Interface（JNDI）ネームスペースにバインドできます。

  そのためには、<connector-factory> 要素を作成し、各要素に location 属性を使用して JNDI ロケーションを割り当てます。また、デプロイヤは <config-property> 要素を使用して各インスタンスを構成することもできます。

  構成可能なプロパティのリストは、ra.xml 内で <config-property> 要素として指定されます。デプロイヤは、oc4j-ra.xml 内で <config-property> 要素を使用して、これらのプロパティの値を指定または上書きできます。

  例 : com.example.eis.ConnectionFactoryImpl のコネクション・ファクトリ実装を持つリソース・アダプタを考えます。このアダプタがスタンドアロンでデプロイされ、1 つのコネクション・ファクトリが構成されており、その JNDI ロケーションが myEIS/connFctry1 であるとします。<connector-factory> は、ポート 1999 でホスト myMc123 に接続するように構成されています。また、このコネクション・ファクトリを論理名 eis/myEIS でルックアップして使用する EJB アプリケーションがあるとします。

  この例に関連するファイルを次に示します。

  ra.xml: コネクション・ファクトリ実装の指定です（リソース・アダプタ・ベンダーが提供）。

```
<resourceadapter>
    . . .
    <config-property>
        <config-property-name>HostName</config-property-name>
```

```
        <config-property-type>java.lang.String</config-property-type>
    </config-property>
    <config-property>
        <config-property-name>Port</config-property-name>
        <config-property-type>java.lang.Integer</config-property-type>
        <config-property-value>2345</config-property-value>
    </config-property>
    <connectionfactory-impl-class>
        com.example.eis.ConnectionFactoryImpl
    </connectionfactory-impl-class>
    ...
</resourceadapter>
```

oc4j-ra.xml: プロパティ myMc123（ホスト）と 1999（ポート）が JNDI ロケーション myEIS/connFctry1 にバインドされる、コネクション・ファクトリ実装の指定です（OC4J により生成され、デプロイヤが編集）。

```
<connector-factory location="myEIS/connFctry1">
    ...
    <config-property>
        <config-property-name>HostName</config-property-name>
        <config-property-value>myMc123</config-property-value>
    </config-property>
    <config-property>
        <config-property-name>Port</config-property-name>
        <config-property-value>1999</config-property-value>
    </config-property>
    ...
</connector-factory>
```

注意： `<config-property-type>` 要素は、型を変更できないため、`oc4j-ra.xml` ファイルには表示されません。

---

ejb-jar.xml: EJB によりアクセスされるリソース参照（コネクション・ファクトリ）の指定です（アプリケーション・ベンダーが提供）。

```
<resource-ref>
  <res-ref-name>eis/myEIS</res-ref-name>
  <res-type>javax.resource.cci.ConnectionFactory</res-type>
  <res-auth>Application</res-auth>
</resource-ref>
```

orion-ejb-jar.xml: 論理参照名から実際の JNDI 名へのマッピングです（OC4J により生成され、デプロイヤが編集）。

```
<resource-ref-mapping name ="eis/myEIS" location ="myEIS/connFctry1"/>
```

EJB クラス : コネクション・ファクトリの使用方法です（デプロイヤが作成）。

```
try
{
    Context ic = new InitialContext();
    cf = (ConnectionFactory) ic.lookup("java:comp/env/eis/myEIS");
} catch (NamingException ex) {
    ex.printStackTrace();
}
```

- 接続プーリングのカスタマイズ

  デプロイヤは、<connection-pooling> 要素を使用してコネクション・ファクトリの各インスタンス用の接続プーリングを構成できます。この要素については、8-10 ページの「接続プーリングの構成」を参照してください。

- 認証の管理

  デプロイヤは、<security-config> 要素を使用して、コネクション・ファクトリの各インスタンスの認証方式を構成できます。この要素を適用できるのは、アプリケーション・コンポーネントでコンテナ管理のサインオンが使用される場合のみです。8-11 ページの「EIS のサインオンの管理」も参照してください。

- ロギングの設定

  デプロイヤは、<log> 要素を使用してコネクション・ファクトリ・インスタンスごとにロギングを設定できます。次に例を示します。

```
<connector-factory location="myEIS/connFctry1">
        <log>
                <file path="./logConnFctry1.log" />
        </log>
</connector-factory>
```

  パス名を指定しないか、ディレクトリが存在しないと、ロギングは有効化されず、OC4J は警告メッセージを出力します。ディレクトリが存在していてもファイルが存在しない場合、OC4J はファイルを作成してロギングを有効化します。ログ・ファイルにはデフォルトの位置がないため、<log> 要素を指定しないとロギングは有効化されません。

  また、デプロイヤは、各 <connector-factory> 要素に <description> 要素を追加することもできます。この要素にはコネクション・ファクトリの記述が含まれており、OC4J では解析されません。

oc4j-connectors.xml ディスクリプタ

OC4J にデプロイされているリソース・アダプタは、oc4j-connectors.xml ディスクリプタを介して構成できます。oc4j-connectors.xml ファイルは、すべてのスタンドアロン・アダプタ用に(グループとして)1 つと、アプリケーションごとに 1 つずつ存在します。

注意: 一般に、/META-INF/oc4j-connectors.xml ファイルは EAR ベンダーから提供される EAR アーカイブの一部ではなく、通常はデプロイ中に OC4J により生成されます。ただし、デプロイヤは、デプロイ前に EAR アーカイブに oc4j-connectors.xml ファイルを追加するように選択できます。または、生成されたファイルを編集することもできます。

ルート要素は <oc4j-connectors> です。各コネクタは、そのコネクタの名前とパス名を指定する <connector> 要素によって表されます。各 <connector> 要素には、次の要素が含まれます。

- <description>: コネクタのテキスト説明。OC4J では解析されません。この要素はオプションです。

- <native-library path="pathname">: ネイティブ・ライブラリが含まれるディレクトリ。この要素を指定しない場合、ライブラリは、解凍された RAR のディレクトリを含むディレクトリにあるとみなされます。OC4J は、解凍された RAR のディレクトリと相対させて pathname 属性を解析します。この要素はオプションです。

- <security-permission enabled="booleanvalue">: 各リソース・アダプタに付与されるパーミッション。各 <security-permission> には、Java 2 セキュリティ・ポリシー・ファイルの構文に準拠した <security-permission-spec> が含まれます。

  OC4J は、ra.xml の各 <security-permission> 要素について、oc4j-connectors.xml の <security-permission> 要素を自動的に生成します。生成された各要素では、enabled 属性が false に設定されています。enabled 属性を true に設定すると、名前付きパーミッションが付与されます。つまり、デプロイヤは、リソース・アダプタからリクエストされるパーミッションを明示的に付与する必要があります。OC4J のデフォルト動作では、これらのパーミッションはデプロイ時に付与されません。

例：

```
<oc4j-connectors>
  <connector name="myEIS" path="eis.rar">
    <native-library> path="lib"</native-library>
    <security-permission>
      <security-permission-spec enabled="false">
        grant {permission java.lang.RuntimePermission "LoadLibrary.*"};
      </security-permission-spec>
    </security-permission>
  </connector>
</oc4j-connectors>
```

注意： <native-library> 要素の path 属性は、.dll または .so ファイルがあるディレクトリを指す必要があります。前述の例では、次のような RAR 構造が可能です。

```
/META-INF/ra.xml
/ra.jar
/lib/win.dll
/lib/solaris.so
```

スタンドアロン・リソース・アダプタ

デプロイ時に、各スタンドアロン・リソース・アダプタには、リソース・アダプタのアンデプロイなどの今後の操作用に、一意の名前を付ける必要があります。OC4J では、同じ名前を持つ 2 つのスタンドアロン・リソース・アダプタのデプロイは許可されません。

デプロイメント・ディスクリプタと解凍された RAR ファイルの位置は、表 8-2 を参照してください。

デプロイ

デプロイ時に、OC4J は RAR ファイルを解凍し、OC4J 固有のデプロイメント・ディスクリプタ・ファイルが存在しない場合は作成します。デプロイメント・プロセスでは、oc4j-connectors.xml ファイルに <connector> エントリが自動的に追加されます。<connector-factory> 要素のスケルトン・エントリは oc4j-ra.xml 内でも作成されます。デプロイヤは、この 2 つのファイルを編集して詳細に構成できます。詳細は、『Oracle Application Server 管理者ガイド』を参照してください。

スタンドアロン・リソース・アダプタのデプロイは、次のいずれかの方法で実行します。

- dcmctl を使用したデプロイとアンデプロイ
- admin.jar を使用したデプロイとアンデプロイ

dcmctl を使用したデプロイとアンデプロイ スタンドアロン・リソース・アダプタを Oracle Application Server インスタンスにデプロイするには、deployApplication オプションを指定してコマンドライン・ツール dcmctl を使用します。構文は次のとおりです。

```
dcmctl deployApplication -f example.rar -a example
```

deployApplication スイッチは、次に示す追加のコマンドライン・スイッチによってサポートされます。

- -f myRA.rar: リソース・アダプタの RAR ファイルのパス名。このスイッチは必須です。
- -a myRA: リソース・アダプタ名。このスイッチは必須です。

デプロイ済のリソース・アダプタを削除するには、undeployApplication オプションを指定して dcmctl を使用します。構文は次のとおりです。

```
dcmctl undeployApplication -a example
```

必須の -a 引数によって、削除するアダプタを指定します。

dcmctl は、RAR、WAR および EAR ファイルをサポートします。詳細は、『Oracle Application Server 管理者ガイド』を参照してください。

admin.jar を使用したデプロイとアンデプロイ スタンドアロン・リソース・アダプタを OC4J スタンドアロン・インスタンスにデプロイするには、-deployconnector スイッチを指定してコマンドライン・ツール admin.jar を使用します。構文は次のとおりです。

```
-deployconnector -file mypath.rar -name myname -nativeLibPath libpathname
-grantAllPermissions
```

-deployconnector スイッチは、次に示す追加のコマンドライン・スイッチによってサポートされます。

- -file myRA.rar: リソース・アダプタの RAR ファイルのパス名。このスイッチは必須です。
- -name myRA: リソース・アダプタ名。このスイッチは必須です。
- -nativeLibPath libpathname: RAR ファイル内のネイティブ・ライブラリのパス名（8-6 ページの「oc4j-connectors.xml ディスクリプタ」の <native-library> 要素も参照）。
- -grantAllPermissions: RAR 内でリクエストされたすべてのランタイム・パーミッションを付与します（8-6 ページの「oc4j-connectors.xml ディスクリプタ」の <security-permission> 要素も参照）。

例：

```
java -jar admin.jar ormi://localhost admin welcome -deployconnector -file ./myRA.rar
-name myRA
```

注意： `admin.jar` の詳細は、『Oracle Application Server Containers for J2EE スタンドアロン・ユーザーズ・ガイド』を参照してください。このマニュアルは、OTN-J から OC4J スタンドアロン製品とともにダウンロードできます。

手動によるデプロイ コネクタを手動でデプロイする場合は、次の手順を実行します。

1. `OC4J_HOME/connectors_dir` の下に connectorname ディレクトリを作成します。
2. `OC4J_HOME/connectors_dir/connectorname` にコネクタの RAR ファイルをコピーします。
3. 新規リソース・アダプタ用として `connectors_dir` に `oc4j-connectors.xml` ファイルを作成するか、ファイルがすでに存在する場合は、そのファイルに `<connector>` 要素を追加します。
4. OC4J を再起動します。OC4J によって、新しい `oc4j-ra.xml` ファイルがアダプタ用の `OC4J_HOME/application-deployments/default/connectorname` に生成されます。生成されたファイルは、使用するコネクタに適した `<connector_factory>` 要素を含むように変更する必要があります。

注意： `oc4j-connectors.xml` と `oc4j-ra.xml` の詳細は、8-4 ページの「oc4j-ra.xml ディスクリプタ」および 8-6 ページの「oc4j-connectors.xml ディスクリプタ」を参照してください。

デプロイ済のリソース・アダプタを削除するには、`admin.jar` の `-undeployconnector` スイッチを使用します。構文は次のとおりです。

```
-undeployconnector -name myname
```

必須の -name 引数によって、削除するアダプタを指定します。このコマンドによって、指定したリソース・アダプタを使用しているすべての <connector> エントリが oc4j-connectors.xml から削除され、デプロイ中に作成されたディレクトリとファイルが削除されます。

埋込みリソース・アダプタ

埋込みリソース・アダプタを、付属しているアプリケーションから独立してデプロイまたはアンデプロイすることはできません。アダプタ名は oc4j-connectors.xml ファイルで指定できます。このファイルで指定されていない場合は、RAR アーカイブ名が使用されます。

デプロイメント・ディスクリプタと解凍された RAR ファイルの位置は、表 8-2 を参照してください。

デプロイ

OC4J では、埋込みリソース・アダプタを含む EAR ファイルのデプロイ時に、RAR ファイルが解凍され、OC4J 固有のデプロイメント・ディスクリプタ・ファイルが存在しない場合は作成されます。デプロイメント・プロセスでは、oc4j-connectors.xml ファイルに <connector> エントリが自動的に追加されます。<connector-factory> 要素のスケルトン・エントリは oc4j-ra.xml 内でも作成されます。デプロイヤは、この 2 つのファイルを編集して詳細に構成できます。

埋込みリソース・アダプタを含むアプリケーションは、次のいずれかの方法でデプロイします。

- dcmctl を使用したデプロイ
- admin.jar によるデプロイ

dcmctl を使用したデプロイ dcmctl の使用方法については、『Oracle Application Server 管理者ガイド』を参照してください。

admin.jar によるデプロイ admin.jar の詳細は、『Oracle Application Server Containers for J2EE スタンドアロン・ユーザーズ・ガイド』を参照してください。

関連ファイルの位置

表 8-1 に、デプロイ時に OC4J により作成され、このマニュアルで言及している各種デプロイメント・ディレクトリへのパスを示します。各パスは、OC4J インストールのルート・ディレクトリへの相対パスです。デプロイメント・ディレクトリは、server.xml 内で表に示す属性を設定することによりカスタマイズできます。これらの属性は、<application-server> 要素に属します。

表 8-1 ディレクトリの位置

	属性	属性の説明	デフォルト値
connectors_dir	connector-directory	すべてのスタンドアロン・リソース・アダプタのルート・ディレクトリ	connectors
applications_dir	applications-directory	すべてのアプリケーションのルート・ディレクトリ	applications
application_ deployments_dir	deployment-directory	デプロイ時に生成されるファイルすべてのルート・ディレクトリ	application-deployments

表 8-2 に、デプロイ時に生成され、このマニュアルで言及している各種ファイルへのパスを示します。各パスは、OC4J インストールのルート・ディレクトリへの相対パスです。表 8-2 では、appname はアプリケーションのデプロイに使用される名前です。

表 8-2 ファイルの位置

	スタンドアロン・リソース・アダプタ	埋込みリソース・アダプタ
解凍された RAR アーカイブの位置	connectors_dir/deployment_name	applications_dir/appname/rar_name
oc4j-connectors.xml	config または application.xml の <connectors> タグに定義されている位置	application_deployments_dir/appname/META-INF または orion-application.xml の <connectors> タグに定義されている位置
oc4j-ra.xml	application_deployments_dir/default/deployment_name	application_deployments_dir/appname/rar_name または oc4j-connectors.xml ファイルが EAR に含まれており、name 属性で connector 要素を指定している場合は、application_deployments_dir/appname/connector_name

Quality of Service に関する規約の指定

デプロイ時に、接続ごとに接続プーリングおよび認証メカニズムを構成できます。この項では、様々な構成方法について説明します。

接続プーリングの構成

接続プーリングは、一連の接続をアプリケーション内で再使用できるようにする J2EE 1.3 の機能です。J2EE Connector 1.0 の仕様は、データベース固有ではなく汎用を目的としているため、J2EE Connector の接続プーリング・インタフェースは、JDBC のインタフェースとはかなり異なります。

oc4j-ra.xml 内で接続プーリングのプロパティを設定するには、オプションの <connection-pooling> 要素内に <property> 要素を指定します。この要素を指定しないと、アプリケーションが接続をリクエストするたびに新規接続が作成されます。構文は次のとおりです。

```
<property name="propname" value="propvalue" />
```

propname の値は次のいずれかです。

- maxConnections: プール内の許容最大接続数。値を指定しないと、接続数は無限となります。
- minConnections: 最小接続数。minConnections が 0（ゼロ）より大きい場合は、指定した接続数が OC4J の初期化時にオープンします。初期化時に必要な情報がない場合は、接続をオープンできない場合があります。たとえば、接続に JNDI ルックアップが必要な場合、初期化が完了するまで JNDI 情報は使用できないため、接続は作成できません。デフォルトは 0（ゼロ）です。
- scheme: 許容最大接続数に達した後の接続リクエストを、OC4J がどのように処理するかを指定します。次のいずれかの値を指定する必要があります。
  - dynamic: OC4J は、最大制限に違反する場合でも、常に新規接続を作成し、アプリケーションに戻します。クローズした制限違反の接続は、接続プールには戻らないで破棄されます。

注意： プール・サイズが maxConnections プロパティに指定した最大値を超えないかぎり、OC4J はクローズ時にプール接続を破棄しません。

- fixed: OC4J は、アプリケーションによる接続リクエストが最大制限に達した場合に例外を呼び出します。
- fixed_wait: OC4J は、使用中の接続がプールに戻されるまで、アプリケーションの接続リクエストをブロックします。waitTimeout が指定されている場合は、指定した時間制限内に接続が使用可能にならなかったときに、例外がスローされます。

- waitTimeout: maxConnections を超過していて fixed_wait スキームが有効な場合に、OC4J が使用可能な接続を待機する最大秒数。他のすべての場合、このプロパティは無視されます。

注意： waitTimeout を指定しないと、デフォルト動作ではタイムアウトなしとなります。

<connection-pooling> 要素の構成例を次に示します。

```
<connection-pooling>
    <description>my pooling configuration </description>
    <property name="waitTimeout" value="60" />
    <property name="scheme" value="fixed_wait" />
    <property name="maxConnections" value="3" />
    <property name="minConnections" value="1" />
</connection-pooling>
```

この例では、接続プールは最小接続数 1（OC4J は起動時に 1 つの接続の作成を試行）と最大接続数 3 を指定して定義されています。3 つの接続がすべて使用中のときに接続リクエストが発行されると、fixed_wait スキームを持つプールは接続が戻されるまで最大 60 秒待機します。60 秒後も使用可能な接続がない場合は、新規接続をリクエストした API のコール元に例外がスローされます。

EIS のサインオンの管理

J2EE Connector Architecture では、EIS への統合に対処するために J2EE モードのエンドツーエンド・セキュリティ拡張の一部として、アプリケーション・コンポーネントで EIS に対して確立済の接続にセキュリティ・コンテキストを関連付けることができます。

アプリケーション・コンポーネントは、それ自体が EIS にサインオンするか、OC4J によりサインオンを管理させることができます。コンポーネント管理のサインオンはプログラム的に実装する必要がありますが、コンテナ管理のサインオンは宣言的またはプログラム的に指定できます。サインオンのタイプは、EJB または Web コンポーネントの <res-auth> デプロイメント・ディスクリプタ要素を使用して指定します。

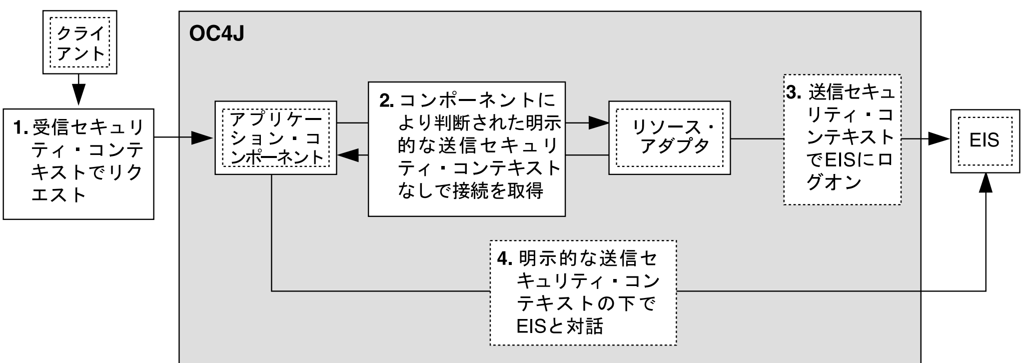
注意： これ以降の説明は、J2EE Connector Architecture 1.0 の仕様の第 7 章を十分に理解していることを前提としています。この仕様では、開始プリンシパル、コール元プリンシパルおよびリソース・プリンシパルという用語を使用しています。この項で使用しているとおり、受信セキュリティ・コンテキストは開始プリンシパルまたはコール元プリンシパルを指し、送信セキュリティ・コンテキストはリソース・プリンシパルを指します。

コンポーネント管理のサインオン

EIS のサインオンを自身で管理するアプリケーションのデプロイ時には、<res-auth> を Application に設定します。サインオンに関する明示的なセキュリティ情報は、アプリケーション・コンポーネントによって提供される必要があります。

図 8-2 に、コンポーネント管理のサインオンに関連するステップを示します。図に続いて各ステップを詳しく説明します。

図 8-2 コンポーネント管理のサインオン



1. クライアントが、受信セキュリティ・コンテキストに関連するリクエストを送ります。
2. リクエスト処理の一部として、アプリケーション・コンポーネントが受信セキュリティ・コンテキストを送信セキュリティ・コンテキストにマップし、送信セキュリティ・コンテキストを使用して EIS への接続をリクエストします。
3. 接続取得の一部として、リソース・アダプタが、アプリケーション・コンポーネントから提供された送信セキュリティ・コンテキストを使用して EIS にログオンします。
4. 接続が取得されると、アプリケーション・コンポーネントは確立された送信セキュリティ・コンテキストの下で EIS と対話できます。

次の例は、コンポーネント管理のサインオンを実行するアプリケーションから抜粋したものです。

例 :

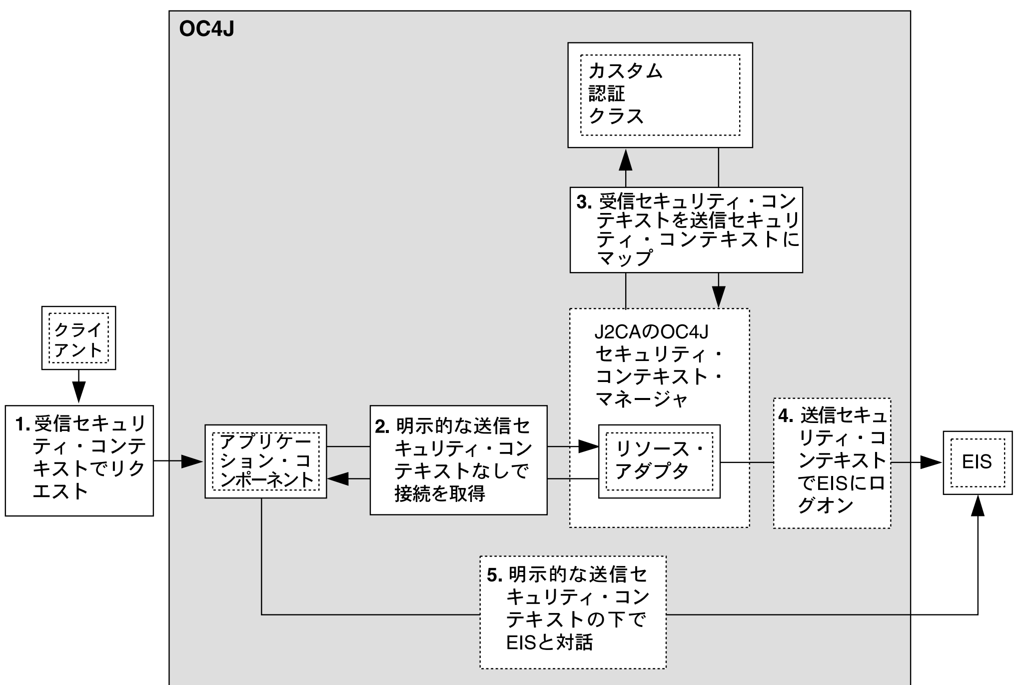
```
Context initctx = new InitialContext();
// perform JNDI lookup to obtain a connection factory
javax.resource.cci.ConnectionFactory cxf =
        (javax.resource.cci.ConnectionFactory)initctx.lookup("java:com/env/eis/MyEIS");

// If component-managed sign-on is specified, the code
// should instead provide explicit security
// information in the getConnection call

// We need to get a new ConnectionSpec implementation
// instance for setting login attributes
com.myeis.ConnectionSpecImpl connSpec = ...
connSpec.setUserName("EISuser");
connSpec.setPassword("EISpassword");
javax.resource.cci.Connection cx = cxf.getConnection(connSpec);
```

コンテナ管理のサインオン

コンテナに EIS サインオンを依存するアプリケーションのデプロイ時には、<res-auth> を Container に設定します。サインオンに関するセキュリティ情報は、コンテナによって提供される必要があります。また、コンテナはデプロイメント・ディスクリプタまたは交換可能認証クラスを使用して、送信セキュリティ・コンテキストを判断します。

図 8-3 に、コンテナ管理のサインオンに関連するステップを示します。図に続いて各ステップを詳しく説明します。

図 8-3 コンテナ管理のサインオン



1. クライアントが、受信セキュリティ・コンテキストに関連するリクエストを送ります。
2. リクエスト処理の一部として、アプリケーション・コンポーネントが EIS への接続をリクエストします。
3. 接続取得の一部として、コンテナ（図 8-3 に示す OC4J のセキュリティ・コンテキスト・マネージャ）は、提供されたデプロイメント・ディスクリプタの要素（図では省略）または認証クラスに基づいて、受信セキュリティ・コンテキストを送信セキュリティ・コンテキストにマップします。
4. リソース・アダプタは、コンテナから提供された送信セキュリティ・コンテキストを使用して EIS にログオンします。
5. 接続が取得されると、アプリケーション・コンポーネントは確立された送信セキュリティ・コンテキストの下で EIS と対話できます。

次の例は、コンテナ管理のサインオンに依存するアプリケーションから抜粋したものです。

例：

```
Context initctx = new InitialContext();

// perform JNDI lookup to obtain a connection factory
javax.resource.cci.ConnectionFactory cxf =
        (javax.resource.cci.ConnectionFactory)initctx.lookup("java:com/env/eis/MyEIS");

// For container-managed sign-on, no security information is passed in the
// getConnection call
javax.resource.cci.Connection cx = cxf.getConnection();
```

宣言的なコンテナ管理のサインオン

oc4j-ra.xml ファイル内でプリンシパルのマッピングを作成できます。プリンシパル・マッピング方式を使用するには、<security-config> 要素の下の <principal-mapping-entries> サブ要素を使用します。

各 <principal-mapping-entry> 要素には、開始プリンシパルからリソース・プリンシパルとパスワードへのマッピングが含まれます。

<default-mapping> 要素を使用して、デフォルトのリソース・プリンシパルのユーザー名とパスワードを指定します。現在の開始プリンシパルに対応する開始ユーザーを持つ <principal-mapping-entry> 要素がない場合は、このプリンシパルを使用して EIS にログインします。<principal-mapping-entries> 要素を指定しないと、OC4J は EIS にログインできない場合があります。

たとえば、OC4J プリンシパル scott が、EIS にユーザー名 scott とパスワード tiger でログインし、他のすべての OC4J ユーザーは、その EIS にユーザー名 guest とパスワード guestpw でログインする場合、oc4j-ra.xml の <connector-factory> 要素は次のようになります。

```
<connector-factory name="..." location="...">
    ...
  <security-config>
    <principal-mapping-entries>
      <default-mapping>
        <res-user>guest</res-user>
        <res-password>guestpw</res-password>
      </default-mapping>
      <principal-mapping-entry>
        <initiating-user>scott</initiating-user>
        <res-user>scott</res-user>
        <res-password>tiger</res-password>
      </principal-mapping-entry>
    </principal-mapping-entries>
  </security-config>
    ...
</connector-factory>
```

注意： <res-password> 要素はパスワードの間接化をサポートします。詳細は、『Oracle Application Server Containers for J2EE セキュリティ・ガイド』を参照してください。

プログラム的なコンテナ管理のサインオン

OC4J は、プログラム的な認証の使用をサポートします。この認証には、OC4J 固有のメカニズムを使用する方法と、Java Authentication and Authorization Service（JAAS）のような標準メカニズムを使用する方法があります。詳細は、Sun 社の JAAS 仕様を参照してください。

OC4J 固有の認証クラス

OC4J は、プリンシパル・マッピング用に `oracle.j2ee.connector.PrincipalMapping` インタフェースを提供します。このインタフェースの各メソッドを表 8-3 に示します。

OC4J 固有のプログラム的なコンテナ管理のサインオンを使用するには、このインタフェースの実装を提供する必要があります。

表 8-3 oracle.j2ee.connector.PrincipalMapping インタフェースのメソッドの説明

メソッドのシグネチャ	説明
`public void init(java.util.Properties prop)`	OC4J によりコールされ、`PrincipalMapping` 実装クラスの設定を初期化します。OC4J は、このメソッドに `oc4j-ra.xml` の `<config-property>` 要素内で指定されているプロパティを渡します。この実装クラスでは、デフォルトのユーザー名とパスワード、LDAP 接続情報またはデフォルト・マッピングの設定にプロパティを使用できます。
`public void setManagedConnectionFactory (ManagedConnectionFactory mcf)`	OC4J で、`PasswordCredential` の作成に必要な `ManagedConnectionFactory` インスタンスに実装クラスを提供するために使用されます。
`public void setAuthenticationMechanisms (java.util.Map authMechanisms)`	OC4J によりコールされ、リソース・アダプタでサポートされている認証方式を `PrincipalMapping` 実装クラスに渡します。渡されるマッピングのキーは、`BasicPassword` または `Kerbv5` など、サポートされている方式のタイプを含む文字列です。値は、`javax.resource.spi.security.PasswordCredential` など、`ra.xml` 内で宣言されている、対応する資格証明インタフェースを含む文字列です。リソース・アダプタで複数の認証方式がサポートされる場合は、マッピングに複数の要素を含めることができます。
`public javax.security.auth.Subject mapping (javax.security.auth.Subject initiatingSubject)`	OC4J により使用され、実装クラスでプリンシパル・マッピングを実行できるようにします。アプリケーション・ユーザーのサブジェクトが渡されます。J2CA 1.0 仕様に従い、このメソッドの実装はリソース・アダプタで EIS リソースへのログインに使用されるサブジェクトを戻す必要があります。正しいリソース・プリンシパルを判断できない場合、実装は `NULL` を戻すことができます。

EIS への接続が作成されると、OC4J は開始ユーザーに `initiatingPrincipal` を指定して mapping メソッドを起動します。mapping メソッドは、リソース・プリンシパルと資格証明を含んだ Subject を戻す必要があります。戻された Subject は、Connector Architecture 1.0 仕様の第 8.2.6 項のオプション A またはオプション B のいずれかに準拠している必要があります。

OC4J は、抽象クラス `oracle.j2ee.connector.AbstractPrincipalMapping` も提供します。このクラスは、`setManagedConnectionFactory()` メソッドおよび `setAuthenticationMechanism()` メソッドのデフォルト実装を提供する他に、リソース・アダプタが BasicPassword または Kerberos バージョン 5（Kerbv5）認証方式をサポートするかどうかを判断するためのユーティリティ・メソッド、およびアプリケーション・サーバー・ユーザー Subject から Principal を抽出するためのメソッドも提供します。`oracle.j2ee.connector.AbstractPrincipalMapping` クラスを拡張することによって、開発者が実装するのは、init メソッドと mapping メソッドのみとなります。

表 8-4 に、oracle.j2ee.connector.AbstractPrincipalMapping クラスで公開されるメソッドを示します。

表 8-4 oracle.j2ee.connector.AbstractPrincipalMapping クラスのメソッドの説明

メソッドのシグネチャ	説明
public abstract void init (java.util.Properties prop)	このメソッドはサブクラスで実装する必要があります。詳細は、表 8-3 の PrincipalMapping インタフェースを参照してください。
public void setManagedConnectionFactory (ManagedConnectionFactory mcf)	渡される ManagedConnectionFactory インスタンスを格納します。サブクラスにこのメソッドを実装する必要はなく、このメソッドにより保存される getManagedConnectionFactory オブジェクトを使用できるようにします。
public void setAuthenticationMechanisms (java.util.Map authMechanisms)	認証方式のマッピングを格納します。この方式をサブクラスで実装する必要はありません。かわりに、isBasicPasswordSupported または isKerbv5Supported メソッドを使用して、リソース・アダプタでサポートされている認証方式を判断できるようにします。getAuthenticationMechanisms メソッドを使用して認証方式を取得することもできます。
public javax.security.auth.Subject mapping (javax.security.auth. Subject initiatingSubject)	OC4J により使用され、実装クラスでプリンシパル・マッピングを実行できるようにします。アプリケーション・ユーザーのサブジェクトが渡されます。J2EE Connector Architecture 仕様に従い、このメソッドの実装はリソース・アダプタで EIS リソースへのログインに使用されるサブジェクトを戻す必要があります。正しいリソース・プリンシパルを判断できない場合、実装は NULL を戻すことができます。
public abstract javax.security.auth.Subject mapping (javax.security.auth. Subject initiatingSubject)	このメソッドはサブクラスで実装する必要があります。詳細は、表 8-3 の PrincipalMapping インタフェースを参照してください。
public ManagedConnectionFactory getManagedConnectionFactory()	PasswordCredentials オブジェクトの作成に必要な ManagedConnectionFactory インスタンスを戻すために、この抽象クラスにより提供されるユーティリティ・メソッド。
public java.util.Map getAuthenticationMechanisms()	OC4J により提供され、このリソース・アダプタでサポートされている認証方式すべてのマッピングを戻すためのユーティリティ・メソッド。戻されるマッピングのキーは、BasicPassword または Kerbv5 など、サポートされている方式のタイプを含む文字列です。値は、javax.resource.spi.security.PasswordCredential など、ra.xml 内で宣言されている、対応する資格証明インタフェースを含む文字列です。
public boolean isBasicPasswordSupported()	このリソース・アダプタで BasicPassword 認証方式がサポートされるかどうかを、サブクラスで判断できるようにするユーティリティ・メソッド。
public boolean isKerbv5Supported()	このリソース・アダプタで Kerbv5 認証方式がサポートされるかどうかを、サブクラスで判断できるようにするユーティリティ・メソッド。
public java.security.Principal getPrincipal (javax.security.auth. Subject subject)	OC4J から渡される特定のアプリケーション・サーバー・ユーザー・サブジェクトから Principal オブジェクトを抽出するために提供されるユーティリティ・メソッド。

AbstractPrincipalMapping の拡張 この簡単な例は、oracle.j2ee.connector.AbstractPrincipalMapping 抽象クラスを拡張して、ユーザーを常にデフォルトのユーザーとパスワードにマップするプリンシパル・マッピングを提供する方法を示しています。デフォルトのユーザーとパスワードの指定には、oc4j-ra.xml の <principal-mapping-interface> 要素の下のプロパティを使用します。

PrincipalMapping クラスは MyMapping という名前です。次のように定義されます。

```
package com.acme.app;

import java.util.*;
import javax.resource.spi.*;
import javax.resource.spi.security.*;
import oracle.j2ee.connector.AbstractPrincipalMapping;
import javax.security.auth.*;
import java.security.*;

public class MyMapping extends AbstractPrincipalMapping
{
  String m_defaultUser;
  String m_defaultPassword;

  public void init(Properties prop)
  {
    if (prop != null)
    {
      // Retrieves the default user and password from the properties
      m_defaultUser = prop.getProperty("user");
      m_defaultPassword = prop.getProperty("password");
    }
  }
  public Subject mapping(Subject initiatingSubject)
  {
    // This implementation only supports BasicPassword authentication
    // mechanism. Return if the resource adapter does not support it.
    if (!isBasicPasswordSupported())
      return null;
    // Use the utility method to retrieve the Principal from the
    // OC4J user. This code is included here only as an example.
    // The principal obtained is not being used in this method.
    Principal principal = getPrincipal(initiatingSubject);
    char[] resPasswordArray = null;
    if (m_defaultPassword != null)
      resPasswordArray = m_defaultPassword.toCharArray();
    // Create a PasswordCredential using the default user name and
    // password, and add it to the Subject per option A in section
    // 8.2.6 in the Connector 1.0 spec.
    PasswordCredential cred =
      new PasswordCredential(m_defaultUser, resPasswordArray);
    cred.setManagedConnectionFactory(getManagedConnectionFactory());
    initiatingSubject.getPrivateCredentials().add(cred);
    return initiatingSubject;
  }
}
```

実装クラスを作成した後は、そのクラスが含まれている JAR ファイルを、解凍済 RAR ファイルが含まれているディレクトリにコピーします。RAR ファイルの位置については、表 8-2 を参照してください。ファイルをコピーした後は、oc4j-ra.xml を編集して、新規クラスに対する <principal-mapping-interface> 要素を組み込みます。

次に例を示します。

```
<connector-factory name="..." location="...">
  ...
  <security-config>
    <principal-mapping-interface>
      <impl-class>com.acme.app.MyMapping</impl-class>
      <property name="user" value="scott" />
      <property name="password" value="tiger" />
    </principal-mapping-interface>
  </security-config>
  ...
</connector-factory>
```

JAAS 交換可能認証クラス

EIS へのサインオンは、JAAS を使用してプログラム的に管理することもできます。OC4J は、Connector Architecture 1.0 仕様の付録 C に準拠する JAAS 交換可能認証フレームワークを提供します。このフレームワークを使用すると、アプリケーション・サーバーとその基礎となる認証サービスは互いに独立した状態となり、アプリケーション・サーバーを変更せずに新規認証サービスをプラグインできます。

認証モジュールの例を次に示します。

- プリンシパル・マッピング JAAS モジュール
- 資格証明マッピング JAAS モジュール
- Kerberos JAAS モジュール（コール元の偽装用）

JAAS ログイン・モジュールは、顧客、EIS のベンダーまたはリソース・アダプタのベンダーによって提供されます。ログイン・モジュールは、Sun 社の JAAS 仕様に記述されているとおりに、javax.security.auth.spi.LoginModule インタフェースを実装する必要があります。

OC4J は、パブリック証明書を含んだ javax.security.auth.Subject のインスタンス、および OC4J ユーザーを表す java.security.Principal の実装のインスタンスを渡すことで、開始ユーザーのサブジェクトをログイン・モジュールに提供します。認証済のユーザーが存在しない（つまり、匿名ユーザーの）場合は、NULL の Subject が渡される可能性があります。開始ユーザーのサブジェクトは、JAAS ログイン・モジュールの initialize メソッドに渡されます。

JAAS ログイン・モジュールの login メソッドでは、開始ユーザーに基づいて、対応するリソース・プリンシパルを検索し、リソース・プリンシパルに対する新しい PasswordCredential または GenericCredential インスタンスを作成する必要があります。次に、リソース・プリンシパルと資格証明オブジェクトが、commit メソッドの開始 Subject に追加されます。リソースの資格証明は、リソース・アダプタが提供する javax.resource.spi.ManagedConnectionFactory 実装の createManagedConnection メソッドに渡されます。

NULL の Subject が渡されると、JAAS ログイン・モジュールは、リソース・プリンシパルと適切な資格証明を含んだ新しい javax.security.auth.Subject を作成します。

JAAS と <connector-factory> 要素 oc4j-ra.xml の各 <connector-factory> 要素は、異なる JAAS ログイン・モジュールを指定できます。<jaas-module> 要素で、コネクタ・ファクトリ構成の名前を指定します。次は、コンテナ管理のサインオンに JAAS ログイン・モジュールを使用している oc4j-ra.xml の <connector-factory> 要素の例です。

```
<connector-factory connector-name="myBlackbox" location="eis/myEIS1">
  <description>Connection to my EIS</description>
  <config-property name="connectionURL"
      value="jdbc:oracle:thin:@localhost:5521:orcl" />
  <security-config>
    <jaas-module>
      <jaas-application-name>JAASModuleDemo</jaas-application-name>
    </jaas-module>
  </security-config>
</connector-factory>
```

JAAS では、特定のアプリケーションに使用する LoginModule と LoginModule の起動順序を指定する必要があります。JAAS は、`<jaas-application-name>` 要素に指定された値を使用して、LoginModule をルックアップします。詳細は、『Oracle Application Server Containers for J2EE セキュリティ・ガイド』を参照してください。

プログラム・インタフェースを介してアクセス可能な特殊機能

ログイン・モジュールと OC4J 固有の認証クラスでは、OC4J ユーザーから EIS ユーザーへのマッピングに加えて、OC4J グループから EIS ユーザーへもマップできます。

`oracle.j2ee.connector` パッケージには、OC4J ユーザーを表す `InitiatingPrincipal` クラスと OC4J グループを表す `InitiatingGroup` クラスが含まれています。OC4J は `InitiatingPrincipal` のインスタンスを作成し、それをログイン・モジュールの initialize メソッドおよび OC4J 固有の認証クラスの mapping メソッドに渡される Subject に取り込みます。

`oracle.j2ee.connector` パッケージには、`java.security.Principal` インタフェースを実装して `getGroups()` メソッドを追加する `InitiatingPrincipal` クラスも含まれています。`getGroups` メソッドは、この OC4J ユーザーが属す OC4J グループまたは JAZN ロールを表す `oracle.j2ee.connector.InitiatingGroup` オブジェクトの `java.util.Set` を戻します。グループのメンバーシップは、ユーザー・マネージャに応じて `principals.xml` または `jazn-data.xml` などの OC4J 固有のディスクリプタ・ファイルで定義されます。`oracle.j2ee.connector.InitiatingGroup` クラスは `java.security.Principal` インタフェースを実装しますが、その機能は拡張しません。

9 Java Object Cache

この章では、Oracle Application Server Containers for J2EE（OC4J）の Java Object Cache について、そのアーキテクチャとプログラミング機能も含めて説明します。この章には、次の項目が含まれます。

- Java Object Cache の概念
- Java Object Cache のオブジェクト・タイプ
- Java Object Cache 環境
- Java Object Cache を使用したアプリケーションの開発
- ディスク・オブジェクトの操作
- StreamAccess オブジェクトの操作
- プール・オブジェクトの操作
- ローカル・モードでの実行
- 分散モードでの実行

Java Object Cache の概念

Oracle Application Server 10*g* は、動的に生成されるコンテンツに関する Web サイトのパフォーマンスの問題を管理する E-Business を支援するために、Java Object Cache を提供しています。Java Object Cache により、Oracle Application Server 10*g* で動作する Web サイトのパフォーマンス、スケーラビリティおよび可用性が向上します。

Java Object Cache では、頻繁にアクセスするオブジェクトや作成にコストがかかるオブジェクトをメモリーまたはディスクに格納することによって、Java プログラム内で情報を繰り返し作成したりロードする必要がなくなります。Java Object Cache は、コンテンツをすばやく取得するため、アプリケーション・サーバーの負荷が大幅に軽減されます。

Oracle Application Server 10*g* のキャッシュ・アーキテクチャには、次のキャッシュ・コンポーネントが含まれています。

- **Oracle Application Server Web Cache**。Web Cache は、アプリケーション・サーバー（Web サーバー）の手前に位置しており、サーバーのコンテンツをキャッシュし、そのコンテンツをリクエストしている Web ブラウザに提供します。Web サイトへのアクセス時に、ブラウザは、HTTP リクエストを Web Cache に送信します。Web Cache は、アプリケーション・サーバーに対する仮想サーバーとして動作します。リクエストしたコンテンツが変更された場合、Web Cache はアプリケーション・サーバーから新しいコンテンツを取得します。

  Web Cache は、アプリケーションの外部でメンテナンスされる HTTP レベルのキャッシュで、高速なキャッシュ操作を行います。純粋なコンテンツ・ベースのキャッシュであるため、静的データ（HTML、GIF、JPEG ファイルなど）または動的データ（サーブレットや JSP の結果など）をキャッシュできます。このキャッシュが、アプリケーション外部にフラットなコンテンツ・ベースのキャッシュとして存在する場合は、オブジェクト（Java オブジェクトや XML DOM（Document Object Model）オブジェクトなど）を構造化された形式でキャッシュすることはできません。さらに、キャッシュされたデータに対する後処理機能がかなり制限されます。

- **Java Object Cache**。Java Object Cache は、アプリケーション・サーバーが Java プログラムを使用してコンテンツを提供する場合に、コストがかかる、あるいは頻繁に使用される Java オブジェクトのキャッシングを提供します。キャッシュされた Java オブジェクトは、生成されたページを含んでいたり、新規コンテンツの作成に役立つプログラム内のサポート・オブジェクトを提供する場合もあります。Java Object Cache は、Java アプリケーションの指定どおりに、オブジェクトを自動的にロードおよび更新します。

- **Web Object Cache**。Web Object Cache は、Web アプリケーション・レベルのキャッシング機能です。アプリケーション・レベルのキャッシュは、Java の Web アプリケーション内に埋め込まれ、メンテナンスされます。Web Object Cache は、Web ベースとオブジェクト・ベースの両方を組み合せたハイブリッド・キャッシュです。Web Object Cache を使用すると、アプリケーションは、Application Program Interface（API）コール（サーブレットの場合）またはカスタム・タグ・ライブラリ（JSP の場合）を使用してプログラムでキャッシュできます。Web Object Cache は通常、Web Cache の補足機能として使用されます。デフォルトでは、Web Object Cache は、そのリポジトリとして Java Object Cache を使用します。

  カスタム・タグ・ライブラリまたは API を使用すると、ユーザーはページ・フラグメントの境界を定義でき、JSP ページやサーブレットの実行の中間結果と部分的な結果を、キャッシュされたオブジェクトとして取得、格納、再使用、処理および管理できます。ブロックごとに独自の結果に基づいたキャッシュ・オブジェクトを作成できます。キャッシュされたオブジェクトは、HTML または XML のテキスト・フラグメント、XML DOM オブジェクトまたは Java のシリアライズ可能オブジェクトなどです。これらのオブジェクトは、HTTP セマンティックと関連付けると簡単にキャッシュできます。また、HTTP 外部で再利用できます。たとえば、キャッシュ内の XML オブジェクトを Simple Mail Transfer Protocol（SMTP）、Java Message Service（JMS）、Advanced Queueing（AQ）または Simple Object Access Protocol（SOAP）を介して出力する場合に再利用できます。

注意： この章では、主に Java Object Cache について説明します。3 種類のキャッシュとその相違点については、『Oracle Application Server Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。

Java Object Cache の基本アーキテクチャ

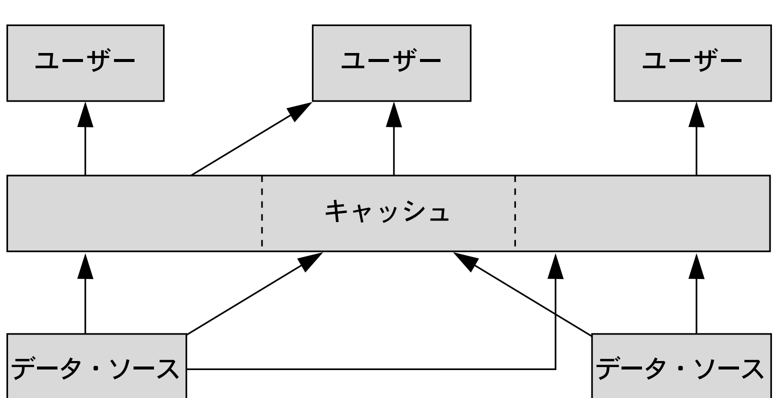
図 9-1 は、Java Object Cache の基本アーキテクチャを示しています。キャッシュは、ユーザー・プロセスに情報を配信します。プロセスとは、HTML ページを生成するサーブレット・アプリケーションまたはその他の Java アプリケーションです。

Java Object Cache は、一般的なアプリケーションを対象とするインプロセスのプロセス全体のキャッシング・サービスです。つまり、オブジェクトはプロセスのメモリー領域内にキャッシュされ、Java Object Cache は単一のサービスとして、そのプロセス内で実行されるすべてのスレッドにより共有されます。これは、他のプロセス内で実行されるサービスとは異なります。Java Object Cache では、すべての Java オブジェクトを管理できます。キャッシュされたオブジェクトを共有しやすいように、キャッシュ内のオブジェクトはすべて名前でアクセスされます。キャッシング・サービスでは、キャッシュされるオブジェクトの構造に制限はありません。オブジェクトの名前、構造、タイプおよび元のソースは、すべてアプリケーションにより定義されます。

システム・リソースを最大限に活用するために、キャッシュ内のオブジェクトはすべて共有されます。共有されていても、キャッシュされたオブジェクトへのアクセスがアクセス・ロックによりシリアライズされることはなく、高水準の同時アクセスが可能です。オブジェクトが無効化または更新されると、そのオブジェクトの無効バージョンは、そのバージョンへの参照が存在するかぎりキャッシュに残っています。そのため、キャッシュにはオブジェクトの複数バージョンが同時に存在する可能性がありますが、有効なバージョンが複数存在することはありません。古いバージョンまたは無効なバージョンのオブジェクトは、無効化される前にそのバージョンを参照していたアプリケーションでのみ参照可能です。オブジェクトが更新されると、そのオブジェクトの新規コピーがキャッシュ内で作成され、古いバージョンには無効を示すマークが付けられます。

オブジェクトは、ユーザー提供の `CacheLoader` オブジェクトを使用してキャッシュにロードされます。このローダー・オブジェクトは、ユーザー・アプリケーションがキャッシュに存在しないオブジェクトをリクエストした時点で Java Object Cache によりコールされます。図 9-1 に、このアーキテクチャを示します。アプリケーションはキャッシュとやり取りしてオブジェクトを取得し、キャッシュは CacheLoader を介してデータ・ソースとやり取りします。このプロセスにより、オブジェクトの作成と使用が明確に分割されます。

図 9-1 Java Object Cache の基本アーキテクチャ



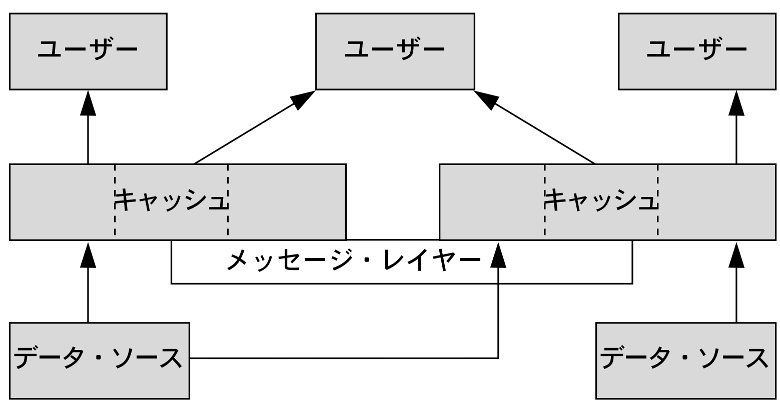
分散オブジェクト管理

Java Object Cache は、複数の Java プロセスが同じアプリケーションを実行したり、同じアプリケーションのかわりに動作している環境で使用できます。この環境の場合、同じオブジェクトを異なるプロセスでキャッシュできると便利です。簡易性、可用性およびパフォーマンスのために、Java Object Cache は各プロセス固有のものです。プロセスへのオブジェクトのロードは、集中管理されません。ただし、Java Object Cache では、プロセス間でオブジェクトの更新および無効化が調整されます。あるプロセス内でオブジェクトが更新または無効化された場合は、他のすべての関連プロセスでもそのオブジェクトが更新または無効化されます。この分散管理によって、集中管理によるオーバーヘッドを発生させることなく複数プロセスのシステムの同期が維持されます。

図 9-2 に、次の動作を示します。

- アプリケーションが Java Object Cache とのやり取りによりオブジェクトを取得する方法
- Java Object Cache がデータ・ソースとやり取りする方法
- Java Object Cache のキャッシュがキャッシュ・メッセージ・システムを介してキャッシュ・イベントを調整する方法

図 9-2 Java Object Cache の分散アーキテクチャ



Java Object Cache の動作

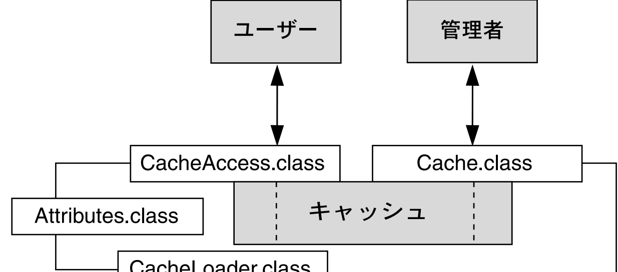
Java Object Cache は、プロセス内、プロセス間およびローカル・ディスク上の Java オブジェクトを管理します。Java Object Cache は、Java オブジェクトのローカル・コピーを管理することによって、Java のパフォーマンスを大幅に向上させる、強力で柔軟な使いやすいサービスを提供します。キャッシュできる Java オブジェクトの型や、オブジェクトの元のソースに関する制限はほとんどありません。プログラマは、Java Object Cache を使用して、取得や作成にコストがかかるオブジェクトを、キャッシュ・アクセスなしに管理します。

Java Object Cache は、新規および既存のアプリケーションに簡単に統合できます。オブジェクトは、ユーザー定義オブジェクト `CacheLoader` を使用してオブジェクト・キャッシュにロードでき、`CacheAccess` オブジェクトを使用してアクセスできます。`CacheAccess` オブジェクトは、ローカル・オブジェクト管理および分散オブジェクト管理をサポートします。Java Object Cache のほとんどの機能が、管理または構成を必要としません。拡張機能によって、`Cache` クラスの管理 API を使用した構成がサポートされます。管理には、ローカル・ディスク領域のネーミングやネットワーク・ポートの定義など、構成オプションの設定が含まれます。管理機能を使用すると、アプリケーションと Java Object Cache を完全に統合できます。

キャッシュされた各 Java オブジェクトには一連の属性が関連付けられており、オブジェクトをキャッシュにロードする方法、オブジェクトの格納場所、およびオブジェクトを無効化する方法が制御されます。キャッシュされたオブジェクトは、時間制御または明示的なリクエストに基づいて無効化されます（オブジェクトが無効化されたときに通知を行うことができます）。オブジェクトは、グループ単位または個別に無効化できます。

図 9-3 は、Java Object Cache の基本 API を示しています。図 9-3 は、分散キャッシュ管理については示していません。

図 9-3 Java Object Cache の基本 API



キャッシュの編成

Java Object Cache は、次のように編成されます。

- **キャッシュ環境**。キャッシュ環境には、キャッシュ・リージョン、サブリージョン、グループおよび属性が含まれます。キャッシュ・リージョン、サブリージョンおよびグループは、オブジェクトとオブジェクトの集合を関連付けます。属性は、キャッシュ・リージョン、サブリージョン、グループおよび個々のオブジェクトに関連付けられます。属性は、Java Object Cache によるオブジェクトの管理方法に影響を与えます。
- **キャッシュ・オブジェクト・タイプ**。キャッシュ・オブジェクト・タイプには、メモリー・オブジェクト、ディスク・オブジェクト、プール・オブジェクトおよび `StreamAccess` オブジェクトがあります。

表 9-1 に、キャッシュ環境およびキャッシュ・オブジェクト・タイプの構成メンバーの要約を示します。

表 9-1 キャッシュの組織化された構成メンバー

キャッシュの構成メンバー	説明
属性	キャッシュ・リージョン、グループおよび個々のオブジェクトに関連付けられる機能。属性は、Java Object Cache によるオブジェクトの管理方法に影響を与えます。
キャッシュ・リージョン	Java Object Cache 内のキャッシュ・オブジェクトの集合を保持する組織化されたネームスペース。
キャッシュ・サブリージョン	親リージョン、サブリージョンまたはグループ内のキャッシュ・オブジェクトの集合を保持する組織化されたネームスペース。
キャッシュ・グループ	オブジェクト間の関連付けを定義するために使用される組織化された構成メンバー。リージョン内のオブジェクトは、グループ単位で無効化できます。グループ内のオブジェクトには、共通の属性を関連付けることができます。
メモリー・オブジェクト	メモリーに格納され、メモリーからアクセスされるオブジェクト。
ディスク・オブジェクト	ディスクに格納され、ディスクからアクセスされるオブジェクト。
プール・オブジェクト	Java Object Cache が管理する同一オブジェクトのセット。オブジェクトは、プールからチェックアウトされ、使用された後に戻されます。

表 9-1 キャッシュの組織化された構成メンバー（続き）

キャッシュの構成メンバー	説明
StreamAccess オブジェクト	Java の OutputStream を使用してロードされ、Java の InputStream を使用してアクセスされるオブジェクト。オブジェクトのサイズおよびキャッシュ容量に応じて、メモリーまたはディスクからアクセスします。

Java Object Cache の機能

Java Object Cache には、次の機能があります。

- オブジェクトを更新または無効化できます。
- オブジェクトを明示的に無効化するか、有効時間またはアイドル時間を指定する属性を使用して無効化できます。
- オブジェクトをプロセス間で調整できます。
- オブジェクトのロードと作成を自動化できます。
- オブジェクトのロードをプロセス間で調整できます。
- オブジェクトを、類似する特性によってキャッシュ・リージョンまたはグループで関連付けることができます。
- イベント処理や特別な処理に対してキャッシュ・イベント通知を提供します。
- キャッシュ管理属性を、オブジェクトごとに指定するか、キャッシュ・リージョンまたはグループに適用できます。

Java Object Cache のオブジェクト・タイプ

この項では、Java Object Cache で管理されるオブジェクト・タイプについて説明します。

- メモリー・オブジェクト
- ディスク・オブジェクト
- StreamAccess オブジェクト
- プール・オブジェクト

注意： オブジェクトは、任意の Java オブジェクトの名前で識別されます。名前の識別に使用される Java オブジェクトは、デフォルト Java オブジェクトの equals メソッドと hashcode メソッドをオーバーライドします。オブジェクトが分散されていて、更新されたりディスクに保存される可能性がある場合は、Serializable インタフェースを実装する必要があります。

メモリー・オブジェクト

メモリー・オブジェクトは、Java Object Cache で管理される Java オブジェクトです。メモリー・オブジェクトは、Java 仮想マシン（JVM）のヒープ領域に Java オブジェクトとして格納されます。メモリー・オブジェクトには、HTML ページ、データベース問合せの結果または Java オブジェクトとして格納できる、任意の情報を格納できます。

メモリー・オブジェクトは通常、アプリケーションが提供するローダーを使用して Java Object Cache にロードされます。メモリー・オブジェクトのソースは、外部にある場合があります（Oracle9*i* Database Server の表のデータを使用する場合など）。アプリケーションが提供するローダーは、ソースにアクセスしてメモリー・オブジェクトを作成または更新します。Java Object Cache がない場合、アプリケーションは、ローダーを使用せずにソースに直接アクセスする必要があります。

メモリー・オブジェクトを更新するには、メモリー・オブジェクトのプライベート・コピーを取得し、コピーに変更を加えた後、更新したオブジェクトを（`CacheAccess.replace()` メソッドを使用して）キャッシュに戻します。これにより、元のメモリー・オブジェクトが置き換えられます。

`CacheAccess.defineObject()` メソッドは、属性をオブジェクトに関連付けます。属性が定義されていない場合、オブジェクトは、その関連リージョン、サブリージョンまたはグループからデフォルト属性を継承します。

アプリケーションは、メモリー・オブジェクトをローカル・ディスクにスプールするようにリクエストできます（`SPOOL` 属性を使用）。この属性を設定すると、サイズが大きい、あるいは再作成のコストが高いためほとんど更新しないメモリー・オブジェクトを Java Object Cache で処理できます。ディスク・キャッシュがメモリー・キャッシュよりかなり大きいサイズに設定されている場合、ディスク上のオブジェクトはメモリー内のオブジェクトより長い時間ディスク・キャッシュに存在します。

ローカル・ディスクにスプールされたメモリー・オブジェクトを `DISTRIBUTE` 属性の分散機能と組み合せると、オブジェクトの永続性が提供されます（Java Object Cache が分散モードで動作している場合）。オブジェクトの永続性により、JVM の再起動後もオブジェクトが存続できます。

ディスク・オブジェクト

ディスク・オブジェクトは、ローカル・ディスクに格納され、Java Object Cache を使用してアプリケーションによってディスクから直接アクセスされます。ディスク・オブジェクトは、指定のディスク位置と `DISTRIBUTE` 属性の設定（および Java Object Cache が分散モードとローカル・モードのどちらで動作しているか）に応じて、Java Object Cache プロセス間で共有される場合と、特定のプロセスに限定される場合があります。

ディスク・オブジェクトは、明示的に無効化するか、`TimeToLive` 属性または `IdleTime` 属性を設定して無効化できます。Java Object Cache で追加領域が要求されたとき、参照されていないディスク・オブジェクトはキャッシュから削除される場合があります。

StreamAccess オブジェクト

StreamAccess オブジェクトは、Java の `InputStream` および `OutputStream` クラスを使用してアクセスされるように設定されている特殊なキャッシュ・オブジェクトです。`StreamAccess` オブジェクトへのアクセス方法は、オブジェクトのサイズとキャッシュの容量に基づいて、Java Object Cache によって決定されます。サイズの小さいオブジェクトはメモリーからアクセスされ、サイズの大きいオブジェクトはディスクから直接ストリームされます。`streamAccess` オブジェクトはすべてディスクに格納されます。

`StreamAccess` オブジェクトに対するキャッシュ・ユーザーのアクセスは、`InputStream` を使用して行われます。メモリー・オブジェクトおよびディスク・オブジェクトに適用される属性はすべて、`StreamAccess` オブジェクトにも適用されます。`StreamAccess` オブジェクトは、ストリームを管理する機能を提供しません。たとえば、`StreamAccess` オブジェクトはソケットのエンドポイントを管理できません。`InputStream` オブジェクトと `OutputStream` オブジェクトは、固定サイズの、潜在的に非常に大きいオブジェクトへのアクセスに使用できます。

プール・オブジェクト

プール・オブジェクトは、Java Object Cache で管理される特別なクラスのオブジェクトです。プール・オブジェクトには、同一オブジェクト・インスタンスのセットが含まれます。プール・オブジェクト自体は共有オブジェクトですが、プール内のオブジェクトはプライベート・オブジェクトです。プール内の個々のオブジェクトは、チェックアウトして使用した後、不要になるとプールに戻すことができます。

`TimeToLive` や `IdleTime` などの属性をプール・オブジェクトに関連付けることができます。これらの属性はプール・オブジェクト全体に適用されます。

Java Object Cache は、アプリケーション定義のファクトリ・オブジェクトを使用してプール内のオブジェクトをインスタンス化します。プールのサイズは、必要に応じて、および `TimeToLive` 属性または `IdleTime` 属性の値に基づいて増減します。プールの最小サイズは、プールの作成時に指定されます。最小サイズ値は、最小保証値ではなく、リクエストとして解釈されます。プール・オブジェクト内のオブジェクトは、領域不足によりキャッシュから削除されるため、プールのサイズはリクエストした最小値より小さくなる場合があります。プールの最大サイズの値を設定すると、プールで使用可能なオブジェクト数に関して強い制限を設けることができます。

Java Object Cache 環境

Java Object Cache 環境には、次のものがあります。

- キャッシュ・リージョン
- キャッシュ・サブリージョン
- キャッシュ・グループ
- リージョンとグループのサイズ制御
- キャッシュ・オブジェクトの属性

この項では、これらの Java Object Cache 環境の構成メンバーについて説明します。

キャッシュ・リージョン

Java Object Cache は、キャッシュ・リージョン内のオブジェクトを管理します。キャッシュ・リージョンは、キャッシュ内のネームスペースを定義します。キャッシュ・リージョン内の各オブジェクトには一意の名前を付ける必要があり、キャッシュ・リージョン名とオブジェクト名の組合せで、オブジェクトが一意に識別される必要があります。したがって、キャッシュ・リージョン名は他のリージョン名と異なる必要があり、リージョン内のすべてのオブジェクトに、リージョンに関して一意の名前を付ける必要があります(異なるリージョンまたはサブリージョン内に存在する場合は、複数のオブジェクトに同じ名前を付けることができます)。

アプリケーションのサポートに必要な数のリージョンを定義できます。ただし、ほとんどのアプリケーションで、必要となるリージョンは 1 つのみです。Java Object Cache は、デフォルト・リージョンを提供しています。リージョンが指定されない場合、オブジェクトはデフォルト・リージョンに配置されます。

リージョンに対して属性を定義できます。属性は、リージョン内のオブジェクト、サブリージョンおよびグループによって継承されます。

キャッシュ・サブリージョン

Java Object Cache は、キャッシュ・リージョン内のオブジェクトを管理します。キャッシュ・リージョン内にサブリージョンを指定すると、子の階層が定義されます。キャッシュ・サブリージョンは、キャッシュ・リージョンまたは上位のキャッシュ・サブリージョン内のネームスペースを定義します。キャッシュ・サブリージョン内の各オブジェクトには一意の名前を付ける必要があり、キャッシュ・リージョン名、キャッシュ・サブリージョン名およびオブジェクト名の組合せで、オブジェクトが一意に識別される必要があります。

アプリケーションのサポートに必要な数のサブリージョンを定義できます。

サブリージョンの定義時に属性が定義されない場合、サブリージョンは、その親であるリージョンまたはサブリージョンから属性を継承します。サブリージョンの属性は、サブリージョン内のオブジェクトによって継承されます。サブリージョンの親リージョンが無効化または破棄されると、サブリージョンも無効化または破棄されます。

キャッシュ・グループ

キャッシュ・グループは、リージョン内のオブジェクト間の関連を作成します。キャッシュ・グループによって、関連オブジェクトをまとめて操作できます。オブジェクトは通常、まとめて無効化する必要があったり、共通の属性を使用するため、キャッシュ・グループで関連付けられます。同じリージョンまたはサブリージョン内のキャッシュ・オブジェクトのセットは、キャッシュ・グループを使用して関連付けることができ、キャッシュ・グループの中に別のキャッシュ・グループを含めることもできます。

Java Object Cache オブジェクトは、ある時点で 1 つのグループにのみ属することができます。オブジェクトをグループに関連付ける前に、グループを明示的に作成する必要があります。グループは名前で定義されます。グループには独自の属性を設定できます。その親であるリージョン、サブリージョンまたはグループから属性を継承することもできます。

グループ名は、個々のオブジェクトの識別には使用されませんが、なんらかの共通点があるオブジェクトのセット（集合）を定義するために使用されます。グループは、階層ネームスペースを定義しません。オブジェクト・タイプでは、名前付けの目的でオブジェクトが区別されることはありません。したがって、リージョン内に同じ名前のグループとメモリー・オブジェクトを含めることはできません。リージョン内に階層なネームスペースを定義するには、サブリージョンを使用する必要があります。

グループの中にグループを含めることができ、親と子の関係を設定できます。子グループは、親グループから属性を継承します。

リージョンとグループのサイズ制御

Java Object Cache では、リージョンまたはグループの最大サイズを、そこに含まれるオブジェクトの数または最大許容バイト数として指定できます。リージョンの容量をバイト数で制御する場合は、リージョン内のすべてのオブジェクトのサイズ属性を設定します。この属性は、オブジェクトの作成時にユーザーが直接設定できます。また、`Attributes.MEASURE` 属性フラグを設定すると、自動的に設定されます。リージョンまたはグループのサイズは、リージョン・レベルとサブリージョン・レベル、リージョンまたは別のグループ内のグループ・レベルなど、ネーミング階層の複数のレベルで設定できます。

リージョンまたはグループの容量に達した場合、そのリージョンまたはグループに関連付けられている `CapacityPolicy` オブジェクトが定義されていれば、そのオブジェクトがコールされます。容量ポリシーが指定されていない場合は、デフォルトのポリシーが使用されます。デフォルトのポリシーでは、同等以下の優先順位を持ち参照されていないオブジェクトが見つかると、そのオブジェクトが新規オブジェクトのために無効化されます。オブジェクトの優先順位属性が設定されていない場合、優先順位は `Integer.MAX_VALUE` とみなされます。削除するオブジェクトの検索時には、そのリージョンまたはグループ内とすべてのサブリージョンおよびサブグループ内のオブジェクトがすべて検索されます。容量ポリシーに基づいて削除可能な最初のオブジェクトが削除されます。そのため、検索領域で最下位の優先順位を持つオブジェクトが削除されるとはかぎらない場合があります。

図 9-4 と図 9-5 に例を示します。どちらの図でも、グレーの部分は検索領域を示しています。

リージョン A の容量はオブジェクト 50 個に設定され、サブリージョン B とサブリージョン C の容量はそれぞれ 20 個に設定されています。リージョン A のオブジェクト数が 50 個に達し、そのうちの 10 個がリージョン A にあり、サブリージョン B および C にそれぞれ 20 個ずつある場合は、リージョン A に対する容量ポリシーがコールされます。削除されるオブジェクトは、リージョン A に含まれている場合と、サブリージョンのどちらかに含まれている場合があります。図 9-4 にこの状況を示します。

リージョン A の容量に達する前にサブリージョン B のオブジェクト数が 20 個になると、サブリージョン B に対する容量ポリシーがコールされ、サブリージョン B のオブジェクトのみが削除対象とみなされます。図 9-5 にこの状況を示します。

図 9-4 容量ポリシーの例 1

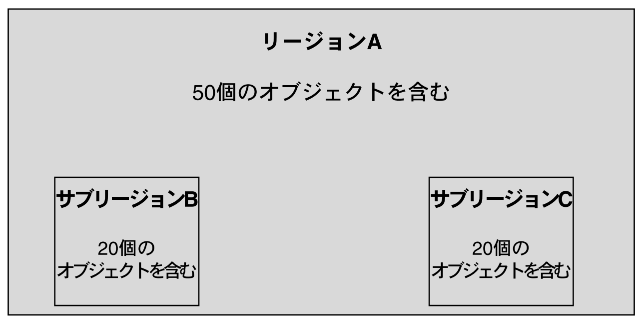
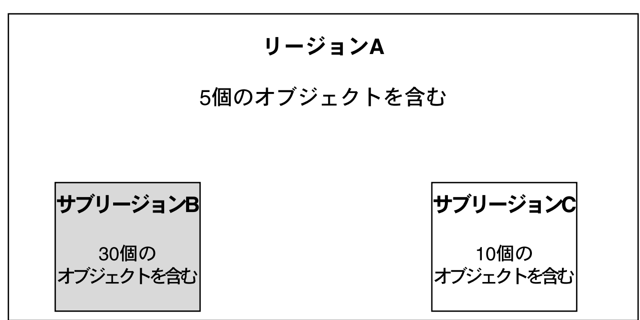


図 9-5 容量ポリシーの例 2



キャッシュ・オブジェクトの属性

キャッシュ・オブジェクトの属性は、Java Object Cache によるオブジェクトの管理方法に影響を与えます。各オブジェクト、リージョン、サブリージョンおよびグループには、一連の属性が関連付けられます。オブジェクトに適用可能な属性は、デフォルトの属性値、オブジェクトの親であるリージョン、サブリージョンまたはグループから継承した属性値、あるいはそのオブジェクトに対してユーザーが選択した属性値のいずれかです。

属性は、次の 2 つのカテゴリに分けられます。

- 最初のカテゴリは、オブジェクトがキャッシュにロードされる前に定義する必要のある属性です。表 9-2 に、これらの属性をまとめます。表 9-2 に示されている各属性には、対応する `set` メソッドまたは `get` メソッドはありません（`LOADER` 属性を除く）。これらの属性を設定するには、`Attributes.setFlags()` メソッドを使用します。
- 第 2 のカテゴリは、オブジェクトがキャッシュに格納された後に変更できる属性です。表 9-3 に、これらの属性をまとめます。

注意： 特定のタイプのオブジェクトには適用されない属性もあります。表 9-2 および表 9-3 の説明の「オブジェクト・タイプ」を参照してください。

オブジェクトのロード前に定義する属性の使用方法

表 9-2 に示した属性は、オブジェクトのロード前に定義する必要があります。これらの属性は、オブジェクトの基本的な管理特性を決定します。

次のリストは、表 9-2 に示した属性の設定に使用できるメソッドです（`Attributes` オブジェクト引数の値を設定します）。

- `CacheAccess.defineRegion()`
- `CacheAccess.defineSubRegion()`
- `CacheAccess.defineGroup()`
- `CacheAccess.defineObject()`
- `CacheAccess.put()`
- `CacheAccess.createPool()`
- `CacheLoader.createDiskObject()`
- `CacheLoader.createStream()`
- `CacheLoader.SetAttributes()`

注意： 表 9-2 に示した属性は、`CacheAccess.resetAttributes()` メソッドを使用してリセットすることはできません。

表 9-2 Java Object Cache の属性（オブジェクト作成時に設定）

属性名	説明
`DISTRIBUTE`	オブジェクトがローカル・オブジェクトであるか分散オブジェクトであるかを指定します。Java Object Cache の分散キャッシング機能を使用している場合は、オブジェクトはローカル・オブジェクトとして設定されるため、更新および無効化はサイト内の他のキャッシュに伝播されません。 **オブジェクト・タイプ :** リージョン、サブリージョンまたはグループに設定されると、オブジェクトに固有の `DISTRIBUTE` 属性が明示的に設定されないかぎり、この属性は、リージョン、サブリージョンまたはグループ内のオブジェクトに対して `DISTRIBUTE` 属性のデフォルト値を設定します。プール・オブジェクトは常にローカルであるため、この属性は適用されません。 **デフォルト値 :** すべてのオブジェクトがローカルです。
`GROUP_TTL_DESTROY`	`TimeToLive` が期限切れになったときに、関連するオブジェクト、グループまたはリージョンを破棄することを示します。 **オブジェクト・タイプ :** リージョンまたはグループに設定されると、`TimeToLive` が期限切れになったときに、リージョンまたはグループ内のすべてのオブジェクト、およびリージョン、サブリージョンまたはグループ自体が破棄されます。 **デフォルト値 :** `TimeToLive` が期限切れになったとき、グループ・メンバー・オブジェクトのみが無効化されます。
`LOADER`	オブジェクトに関連付けられる `CacheLoader` を指定します。 **オブジェクト・タイプ :** リージョンまたはグループに設定されると、指定した `CacheLoader` が、そのリージョン、サブリージョンまたはグループのデフォルト・ローダーになります。`LOADER` 属性は、リージョンまたはグループ内のオブジェクトごとに指定します。 **デフォルト値 :** 設定されません。

表 9-2 Java Object Cache の属性（オブジェクト作成時に設定）（続き）

属性名	説明
ORIGINAL	オブジェクトが外部ソースからロードされたのではなく、キャッシュ内で作成されたことを示します。ORIGINAL オブジェクトは、参照件数が 0（ゼロ）になっても、キャッシュから削除されません。ORIGINAL オブジェクトは、不要になったときに明示的に無効化する必要があります。 **オブジェクト・タイプ :** リージョンまたはグループに設定されると、オブジェクトに固有の ORIGINAL 属性が設定されないかぎり、この属性は、リージョン、サブリージョンまたはグループ内のオブジェクトに対して ORIGINAL 属性のデフォルト値を設定します。 **デフォルト値 :** 設定されません。
REPLY	あるオブジェクトの更新または無効化のリクエストが完了した後、リモート・キャッシュから応答メッセージが送信されるように指定します。この属性は、キャッシュ間で高水準の整合性が必要な場合に設定してください。DISTRIBUTE 属性が設定されていない場合、あるいはキャッシュが非分散モードで開始された場合、REPLY は無視されます。 **オブジェクト・タイプ :** リージョンまたはグループに設定されると、オブジェクトに固有の REPLY 属性が明示的に設定されないかぎり、この属性は、リージョン、サブリージョンまたはグループ内のオブジェクトに対して REPLY 属性のデフォルト値を設定します。メモリー・オブジェクト、StreamAccess オブジェクトおよびディスク・オブジェクトの場合、この属性は、DISTRIBUTE 属性の値が DISTRIBUTE に設定されている場合にのみ適用されます。プール・オブジェクトは常にローカルであるため、この属性は適用されません。 **デフォルト値 :** 応答は送信されません。DISTRIBUTE がローカルに設定されている場合、REPLY 属性は無視されます。
SPOOL	領域を回復するために、キャッシュ・システムによってメモリーからメモリー・オブジェクトが削除されるときに、そのメモリー・オブジェクトを消去せずにディスクに格納するように指定します。この属性はメモリー・オブジェクトにのみ適用されます。オブジェクトが分散オブジェクトでもある場合、オブジェクトは、それをスプールしたプロセスの終了後も存続します。ローカル・オブジェクトにアクセスできるのは、それをスプールしているプロセスのみであるため、Java Object Cache が分散モードで動作していない場合、スプール・オブジェクトは、プロセスの終了時に失われます。 **注意 :** オブジェクトをスプールするには、シリアライズ可能であることが必要です。 **オブジェクト・タイプ :** リージョン、サブリージョンまたはグループに設定されると、オブジェクトに固有の SPOOL 属性が設定されないかぎり、この属性は、リージョン、サブリージョンまたはグループ内のオブジェクトに対して SPOOL 属性のデフォルト値を設定します。 **デフォルト値 :** メモリー・オブジェクトはディスクに格納されません。
SYNCHRONIZE	このオブジェクトに対する更新を同期化する必要があることを示します。このフラグが設定されている場合、オブジェクトをロードまたは置換できるのは、そのオブジェクトの所有者のみとなります。所有権を取得するには、CacheAccess.getOwnership() メソッドを使用します。オブジェクトの所有者は CacheAccess オブジェクトです。SYNCHRONIZE 属性を設定することによって、ユーザーがオブジェクトの読取りまたは無効化を実行できなくなることはありません。 **オブジェクト・タイプ :** リージョン、サブリージョンまたはグループに設定すると、所有権の制限は、そのリージョン、サブリージョンまたはグループ全体に適用されます。プール・オブジェクトはこの属性を使用しません。 **デフォルト値 :** 更新は同期化されません。

表 9-2 Java Object Cache の属性（オブジェクト作成時に設定）（続き）

属性名	説明
SYNCHRONIZE_DEFAULT	リージョン、サブリージョンまたはグループ内のすべてのオブジェクトが同期化されることを示します。リージョン、サブリージョンまたはグループ内の各ユーザー・オブジェクトは、SYNCHRONIZE 属性でマーク付けされます。オブジェクトの所有権は、オブジェクトがロードまたは更新される前に取得する必要があります。 SYNCHRONIZE_DEFAULT 属性を設定することによって、ユーザーがオブジェクトの読取りまたは無効化を実行できなくなることはありません。したがって、SYNCHRONIZE 属性が設定されているオブジェクトの読取りまたは無効化には、所有権は必要ありません。 **オブジェクト・タイプ :** リージョン、サブリージョンまたはグループに設定すると、所有権は、そのリージョン、サブリージョンまたはグループ内の個々のオブジェクトに適用されます。プール・オブジェクトはこの属性を使用しません。 **デフォルト値 :** 更新は同期化されません。
ALLOWNULL	キャッシュが影響を受けるオブジェクトの有効な値として NULL を受け入れるように指定します。cacheLoader オブジェクトから戻される NULL オブジェクトは、ObjectNotFoundException を生成するのではなくキャッシュされます。 **オブジェクト・タイプ :** リージョン、サブリージョン、グループまたはプールに設定されると、オブジェクトに明示的に設定されないかぎり、この属性は、リージョン、サブリージョン、グループまたはプール内の各オブジェクトに個別に適用されます。 **デフォルト値 :** OFF（NULL は使用できません）。
MEASURE	オブジェクトがキャッシュにロードまたは置換されるときに、キャッシュされるオブジェクトのサイズ属性を自動的に計算することを指定します。キャッシュまたはリージョンの容量を、オブジェクトの数ではなくサイズに基づいて正確に制御できます。 **オブジェクト・タイプ :** リージョン、サブリージョン、グループまたはプールに設定されると、オブジェクトに明示的に設定されないかぎり、この属性は、リージョン、サブリージョン、グループまたはプール内の各オブジェクトに個別に適用されます。 **デフォルト値 :** OFF（オブジェクトのサイズは自動的には計算されません）。
CapacityPolicy	リージョンまたはグループのサイズ制御に CapacityPolicy オブジェクトを使用するように指定します。この属性は、個々のオブジェクトに対して設定すると無視されます。 **オブジェクト・タイプ :** リージョン、サブリージョンまたはグループに設定すると、この属性はそのリージョンまたはグループ全体に適用されます。この属性は、個々のオブジェクトやプールには適用できません。 **デフォルト値 :** OFF（リージョンまたはグループに対する容量ポリシーは定義されません。リージョンまたはグループが容量に達すると、そのリージョンまたはグループ内の参照されていない最初のオブジェクトが無効化されます）。

オブジェクトのロード前およびロード後に定義する属性の使用方法

一連の Java Object Cache の属性は、オブジェクトのロード前またはロード後に更新できます。表 9-3 に、これらの属性を示します。これらの属性は、9-11 ページの「オブジェクトのロード前に定義する属性の使用方法」に示したリストのメソッドを使用して設定でき、CacheAccess.resetAttributes() メソッドを使用してリセットできます。

表 9-3 Java Object Cache の属性

属性名	説明
DefaultTimeToLive	リージョン、サブリージョンまたはグループ内のすべてのオブジェクトに対して個別に適用される TimeToLive 属性のデフォルト値を設定します。この属性は、リージョン、サブリージョンおよびグループにのみ適用されます。この値は、個々のオブジェクトに TimeToLive を設定すると上書きできます。 **オブジェクト・タイプ :** リージョン、サブリージョン、グループまたはプールに設定されると、オブジェクトに固有の TimeToLive が明示的に設定されないかぎり、この属性は、リージョン、サブリージョン、グループまたはプール内のすべてのオブジェクトに適用されます。 **デフォルト値 :** 自動的な無効化は行われません。

表 9-3 Java Object Cache の属性（続き）

属性名	説明
IdleTime	オブジェクトが無効化されるまでの、キャッシュに（参照件数 0（ゼロ）で）アイドル状態で存在する時間を指定します。TimeToLive 属性または DefaultTimeToLive 属性が設定されている場合、IdleTime 属性は無視されます。 **オブジェクト・タイプ :** リージョン、サブリージョン、グループまたはプールに設定されると、オブジェクトに IdleTime が明示的に設定されないかぎり、この属性は、リージョン、サブリージョン、グループまたはプール内の各オブジェクトに個別に適用されます。 **デフォルト値 :** IdleTime による自動的な無効化は行われません。
CacheEventListener	オブジェクトに関連付けられる CacheEventListener を指定します。 **オブジェクト・タイプ :** リージョン、サブリージョンまたはグループに設定されると、CacheEventListener がリージョン、サブリージョンまたはグループ内のオブジェクトに個別に指定されないかぎり、指定した CacheEventListener は、そのリージョン、サブリージョンまたはグループのデフォルトの CacheEventListener になります。 **デフォルト値 :** CacheEventListener は設定されません。
TimeToLive	オブジェクトが無効化されるまでにキャッシュに存在する最大時間を設定します。リージョン、サブリージョンまたはグループに関連付けられると、期限切れになった場合に、そのリージョン、サブリージョンまたはグループ内のすべてのオブジェクトが無効化されます。リージョン、サブリージョンまたはグループが破棄されない場合（つまり、GROUP_TTL_DESTROY が設定されていない場合）、TimeToLive の値はリセットされます。 **オブジェクト・タイプ :** リージョン、サブリージョン、グループまたはプールに設定されると、オブジェクトに固有の TimeToLive が明示的に設定されないかぎり、この属性は、リージョン、サブリージョン、グループまたはプール全体に適用されます。 **デフォルト値 :** 自動的な無効化は行われません。
Version	アプリケーションで、キャッシュ内のオブジェクトの各インスタンスに対して Version が設定される場合があります。Version は、アプリケーションの利便性および確認に利用できます。キャッシング・システムではこの属性は使用されません。 **オブジェクト・タイプ :** リージョン、サブリージョン、グループまたはプールに設定されると、オブジェクトに固有の Version が明示的に設定されないかぎり、この属性は、リージョン、サブリージョン、グループまたはプール内のすべてのオブジェクトに適用されます。 **デフォルト値 :** デフォルトの Version は 0 です。
Priority	容量に達した時点で、どのオブジェクトがキャッシュまたはリージョンから削除されるかを制御します。この属性は整数で、キャッシュ、リージョンまたはグループのサイズ制御に使用する CapacityPolicy オブジェクトで使用できます。数が大きいほど優先順位が高くなります。リージョンとグループの容量制御では、空き容量（特に他のオブジェクト用）を増やすためにオブジェクトが削除される場合、優先順位の低いオブジェクトをキャッシュに入れるために優先順位の高いオブジェクトが削除されることはありません。キャッシュの容量制御では、優先順位の高いオブジェクトをキャッシュに入れるために優先順位の低いオブジェクトが削除対象として選択されます。 **オブジェクト・タイプ :** リージョン、サブリージョン、グループまたはプールに設定されると、オブジェクトに明示的に設定されないかぎり、この属性は、リージョン、サブリージョン、グループまたはプール内の各オブジェクトに個別に適用されます。 **デフォルト値 :** integer.MAX_VALUE。
MaxSize	リージョンまたはグループに使用可能な最大バイト数を指定します。この属性をオブジェクトに対して指定すると無視されます。 **オブジェクト・タイプ :** リージョン、サブリージョンまたはグループに設定すると、この属性はそのリージョンまたはグループ全体に適用されます。この属性は、個々のオブジェクトやプールには適用できません。 **デフォルト値 :** 無制限。

表 9-3 Java Object Cache の属性（続き）

属性名	説明
MaxCount	リージョンまたはグループに格納できるオブジェクトの最大数を指定します。この属性をオブジェクトに対して指定すると無視されます。 **オブジェクト・タイプ :** リージョン、サブリージョンまたはグループに設定すると、この属性はそのリージョンまたはグループ全体に適用されます。この属性は、個々のオブジェクトやプールには適用できません。 **デフォルト値 :** 無制限。
ユーザー定義属性	属性をユーザーが定義できます。この種の属性は、オブジェクト、グループまたはリージョンに関連付けられる名前 / 値ペアです。`CapacityPolicy` オブジェクトとの併用を意図した属性ですが、必要に応じてキャッシュ・ユーザーが定義できます。 **オブジェクト・タイプ :** リージョン、サブリージョン、グループまたはプールに設定されると、オブジェクトに対して明示的にリセットされないかぎり、これらの属性は、リージョン、サブリージョン、グループまたはプール内の各オブジェクトに使用できます。 **デフォルト値 :** デフォルトでは、ユーザー定義属性は設定されません。

Java Object Cache を使用したアプリケーションの開発

この項では、Java Object Cache を使用するアプリケーションの開発方法を説明します。この項には、次の項目が含まれます。

- キャッシュ・オブジェクトの属性
- Java Object Cache のインポート
- キャッシュ・グループの定義
- キャッシュ・サブリージョンの定義
- キャッシュ・オブジェクトの定義と使用
- CacheLoader オブジェクトの実装
- キャッシュ・オブジェクトの無効化
- キャッシュ・オブジェクトの破棄
- 複数のオブジェクトのロードおよび無効化
- Java Object Cache の構成
- 宣言的なキャッシュ
- 容量制御
- キャッシュ・イベント・リスナーの実装
- 制限事項およびプログラミングに関する注意点

Java Object Cache のインポート

Oracle Installer によって、Java Object Cache の JAR ファイル `cache.jar` が、`$ORACLE_HOME/javacache/lib` ディレクトリ（UNIX の場合）または `%ORACLE_HOME%¥javacache¥lib` ディレクトリ（Windows の場合）にインストールされます。

Java Object Cache を使用するには、次のように `oracle.ias.cache` をインポートします。

```
import oracle.ias.cache.*;
```

キャッシュ・リージョンの定義

Java Object Cache へのアクセスはすべて、キャッシュ・リージョンに関連付けられている CacheAccess オブジェクトを使用して行われます。キャッシュ・リージョンは通常、static メソッド CacheAccess.defineRegion() を使用して、アプリケーション名に関連付けて定義します。キャッシュが初期化されていない場合は、defineRegion() によって Java Object Cache が初期化されます。

リージョンを定義するときに属性も設定できます。属性は、Java Object Cache によるオブジェクトの管理方法を指定します。Attributes.setLoader() メソッドは、キャッシュ・ローダーの名前を設定します。例 9-1 にこの設定を示します。

例 9-1 CacheLoader の名前の設定

```
Attributes attr = new Attributes();
MyLoader mloader = new MyLoader;
attr.setLoader(mloader);
attr.setDefaultTimeToLive(10);

final static String APP_NAME_ = "Test Application";
CacheAccess.defineRegion(APP_NAME_, attr);
```

defineRegion の最初の引数は、String を使用してリージョン名を設定します。この static メソッドは、Java Object Cache 内にプライベート・リージョン名を作成します。2 番目の引数は、デフォルトのキャッシュ属性を使用して、新規リージョンの属性を定義します。

キャッシュ・グループの定義

キャッシュ内の複数のオブジェクト間の関連を作成する場合は、キャッシュ・グループを作成します。オブジェクトは通常、まとめて無効化する必要があったり、共通の属性セットを使用するため、キャッシュ・グループで関連付けられます。

同じリージョンまたはサブリージョン内のキャッシュ・オブジェクトのセットは、キャッシュ・グループを使用して関連付けることができ、このキャッシュ・グループには別のキャッシュ・グループを含めることもできます。オブジェクトをキャッシュ・グループに関連付ける前に、キャッシュ・グループを定義する必要があります。キャッシュ・グループは名前で定義されます。また、固有の属性を使用するか、あるいはその親であるキャッシュ・グループ、サブリージョンまたはリージョンから属性を継承できます。例 9-2 のコードは、Test Application という名前のリージョン内にキャッシュ・グループを定義します。

例 9-2 キャッシュ・グループの定義

```
final static String APP_NAME_ = "Test Application";
final static String GROUP_NAME_ = "Test Group";
// obtain an instance of CacheAccess object to a named region
CacheAccess caccess = CacheAccess.getAccess(APP_NAME_);
// Create a group
caccess.defineGroup(GROUP_NAME_);
// Close the CacheAccess object
caccess.close();
```

キャッシュ・サブリージョンの定義

リージョン内またはすでに定義されているサブリージョン内にプライベート・ネームスペースを作成する場合は、サブリージョンを定義します。サブリージョンのネームスペースは、親のネームスペースから独立しています。リージョン内では、異なるサブリージョン内に存在する場合は、2 つのオブジェクトに同じ名前を付けることができます。

サブリージョンには、キャッシュ・オブジェクト、グループまたは別のサブリージョンなど、リージョンに含めることができるすべてのものを含めることができます。オブジェクトをサブリージョンに関連付ける前に、サブリージョンを作成する必要があります。キャッシュ・サブリージョンは名前で定義されます。また、固有の属性を使用するか、あるいはその親であるキャッシュ・リージョンまたはサブリージョンから属性を継承できます。サブリージョンの親を取得するには、getParent() メソッドを使用します。

例 9-3 のコードは、Test Application という名前のリージョン内にキャッシュ・サブリージョンを定義します。

例 9-3 キャッシュ・サブリージョンの定義

```
final static String APP_NAME_ = "Test Application";
final static String SUBREGION_NAME_ = "Test SubRegion";
// obtain an instance of CacheAccess object to a named region
CacheAccess caccess = CacheAccess.getAccess(APP_NAME_);
// Create a SubRegion
caccess.defineSubRegion(SUBREGION_NAME_);
// Close the CacheAccess object
caccess.close();
```

キャッシュ・オブジェクトの定義と使用

オブジェクトのロード前に、キャッシュ内での個々のオブジェクトの管理方法を、Java Object Cache に対して示すこともできます。管理オプションは、オブジェクトのロード時に、CacheLoader.load() メソッド内で属性を設定することで指定できます。ただし、CacheAccess.defineObject() メソッドを使用して、属性をオブジェクトに関連付けることもできます。オブジェクトに対する属性が定義されていない場合、Java Object Cache は、そのオブジェクトが関連付けられているリージョン、サブリージョンまたはグループのデフォルトの属性セットを使用します。

例 9-4 は、キャッシュ・オブジェクトの属性の設定方法を示しています。この例は、リージョン APP_NAME_ がすでに定義されていることを前提としています。

例 9-4 キャッシュ属性の設定

```
import oracle.ias.cache.*;
final static String APP_NAME_ = "Test Application";
CacheAccess cacc = null;
try
{
   cacc = CacheAccess.getAccess(APP_NAME_);
// set the default IdleTime for an object using attributes
   Attributes attr = new Attributes();
// set IdleTime to 2 minutes
   attr.setIdleTime(120);

// define an object and set its attributes
   cacc.defineObject("Test Object", attr);

// object is loaded using the loader previously defined on the region
// if not already in the cache.
   result = (String)cacc.get("Test Object");
}  catch (CacheException ex){
     // handle exception
   } finally {
```

```
        if (cacc!= null)
           cacc.close();
}
```

CacheLoader オブジェクトの実装

Java Object Cache には、オブジェクトをキャッシュにロードするためのメカニズムが 2 つ用意されています。

- アプリケーションで `CacheAccess.put()` メソッドを使用してオブジェクトをキャッシュに直接入れることができます。
- `CacheLoader` オブジェクトを実装できます。

ほとんどの場合は、`CacheLoader` を実装する方法をお薦めします。キャッシュ・ローダーを使用すると、オブジェクトのリクエスト時に、オブジェクトをキャッシュにロードする必要があるかどうかが Java Object Cache によって自動的に判断されます。また、Java Object Cache では、オブジェクトが同時に複数のユーザーからリクエストされた場合に、ロードが調整されます。

`CacheLoader` オブジェクトをリージョン、サブリージョン、グループまたはオブジェクトに関連付けることができます。`CacheLoader` を使用すると、Java Object Cache でオブジェクトのロードをスケジュールおよび管理し、「オブジェクトがキャッシュ内に存在しない場合はロードする」というロジックを処理できます。

オブジェクトがキャッシュ内に存在しない場合、アプリケーションで `CacheAccess.get()` または `CacheAccess.preLoad()` メソッドがコールされると、キャッシュによって `CacheLoader.load` メソッドが実行されます。`load` メソッドが戻されると、Java Object Cache は戻されたオブジェクトをキャッシュに挿入します。`CacheAccess.get()` の使用時にキャッシュがいっぱいの場合、オブジェクトはローダーから戻された後、すぐにキャッシュ内で無効化されます（したがって、キャッシュがいっぱいの状態で `CacheAccess.get()` メソッドを使用した場合、`CacheFullException` は生成されません）。

リージョン、サブリージョンまたはグループに対して定義されている場合、`CacheLoader` は、そのリージョン、サブリージョンまたはグループに関連付けられているすべてのオブジェクトのデフォルト・ローダーとみなされます。個々のオブジェクトに対して定義されている `CacheLoader` オブジェクトは、そのオブジェクトのロードにのみ使用されます。

注意： リージョン、サブリージョンまたはグループ、あるいは複数のキャッシュ・オブジェクトに対して定義されている `CacheLoader` オブジェクトは、同時アクセスを考慮して記述する必要があります。`CacheLoader` オブジェクトが共有されるため、実装はスレッド・セーフであることが必要です。

CacheLoader のヘルパー・メソッドの使用方法

CacheLoader は、load() メソッドの実装内で使用できるヘルパー・メソッドをいくつか提供しています。表 9-4 に、使用可能な CacheLoader メソッドをまとめます。

表 9-4 load() で使用される CacheLoader メソッド

メソッド	説明
setAttributes()	ロードされるオブジェクトの属性を設定します。
netSearch()	使用可能な他のキャッシュを検索して、ロード対象のオブジェクトを探します。オブジェクトは、リージョン名、サブリージョン名およびオブジェクト名で一意に識別されます。
getName()	ロードされるオブジェクトの名前を戻します。
getRegion()	ロードされるオブジェクトに関連付けられたリージョンの名前を戻します。
createStream()	StreamAccess オブジェクトを作成します。
createDiskObject()	ディスク・オブジェクトを作成します。
exceptionHandler()	非キャッシュ例外を CacheExceptions に変換し、そのベースを元の例外に設定します。
log()	キャッシュ・サービスのログにメッセージを記録します。

例 9-5 は、ロードされるオブジェクトが分散 Java Object Cache キャッシュで使用可能かどうかを、cacheLoader.netSearch() メソッドを使用してチェックする CacheLoader オブジェクトを示しています。netSearch() によってオブジェクトが見つからない場合、ロード・メソッドは、よりコストがかかるコールを使用してオブジェクトを取得します（コストがかかるコールには、リモート Web サイトへの HTTP 接続や Oracle9*i* Database Server への接続などがあります)。この例の場合、Java Object Cache は、結果を String として格納します。

例 9-5 CacheLoader の実装

```
import oracle.ias.cache.*;
class YourObjectLoader extends CacheLoader{
      public YourObjectLoader () {
      }
      public Object load(Object handle, Object args) throws CacheException
     {
         String contents;
         // check if this object is loaded in another cache
         try {
            contents = (String)netSearch(handle, 5000);// wait for up to 5 scnds
            return new String(contents);
         } catch(ObjectNotFoundException ex){}

         try {
            contents =  expensiveCall(args);
            return new String(contents);
         } catch (Exception ex) {throw exceptionHandler("Loadfailed", ex);}
           }

    private String expensiveCall(Object args) {
        String str = null;
        // your implementation to retrieve the information.
        // str = ...
        return str;
    }
 }
```

キャッシュ・オブジェクトの無効化

オブジェクトをキャッシュから削除するには、オブジェクト、グループ、サブリージョンまたはリージョンに TimeToLive 属性を設定するか、オブジェクトを明示的に無効化または破棄します。

オブジェクトを無効化すると、そのオブジェクトに、キャッシュから削除されたことを示すマークが付けられます。リージョン、サブリージョンまたはグループを無効化すると、リージョン、サブリージョンまたはグループ内の個々のオブジェクトすべてが無効化されますが、すべてのグループ、ローダーおよび属性などの環境は使用可能なままキャッシュに残ります。オブジェクトを無効化してもオブジェクトの定義は解除されません。オブジェクト・ローダーはその名前に関連付けられたままになります。オブジェクトをキャッシュから完全に削除するには、CacheAccess.destroy() メソッドを使用します。

オブジェクトは、TimeToLive 属性または IdleTime 属性に基づいて自動的に無効化することもできます。TimeToLive または IdleTime が期限切れになったとき、デフォルトではオブジェクトは無効化され、破棄されません。

オブジェクト、グループ、サブリージョンまたはリージョンが分散として定義されている場合、無効化リクエストは分散環境内のすべてのキャッシュに伝播されます。

オブジェクト、グループ、サブリージョンまたはリージョンを無効化するには、次のように CacheAccess.invalidate() メソッドを使用します。

```
CacheAccess cacc = CacheAccess.getAccess("Test Application");
cacc.invalidate("Test Object");  // invalidate an individual object
cacc.invalidate("Test Group"); // invalidate all objects associated with a group
cacc.invalidate();     // invalidate all objects associated with the region cacc
cacc.close();          // close the CacheAccess handle
```

キャッシュ・オブジェクトの破棄

オブジェクトをキャッシュから削除するには、オブジェクト、グループ、サブリージョンまたはリージョンに TimeToLive 属性を設定するか、オブジェクトを明示的に無効化または破棄します。

オブジェクトを破棄すると、オブジェクトとその関連環境（関連するすべてのローダー、イベント・ハンドラおよび属性など）に、キャッシュから削除されたことを示すマークが付けられます。リージョン、サブリージョンまたはグループを破棄すると、リージョン、サブリージョンまたはグループに関連付けられているすべてのオブジェクト（関連環境も含む）に、削除されたことを示すマークが付けられます。

オブジェクトは、TimeToLive 属性または IdleTime 属性に基づいて自動的に破棄することもできます。デフォルトでは、オブジェクトは無効化され、破棄はされません。オブジェクトを破棄する必要がある場合は、属性 GROUP_TTL_DESTROY を設定します。リージョンを破棄すると、リージョンのアクセスに使用された CacheAccess オブジェクトもクローズされます。

オブジェクト、グループ、サブリージョンまたはリージョンを破棄するには、次のように CacheAccess.destroy() メソッドを使用します。

```
CacheAccess cacc = CacheAccess.getAccess("Test Application");
cacc.destroy("Test Object"); // destroy an individual object
cacc.destroy("Test Group");  // destroy all objects associated with
                             // the group "Test Group"

cacc.destroy();       // destroy all objects associated with the region
                      // including groups and loaders
```

複数のオブジェクトのロードおよび無効化

ほとんどの場合、オブジェクトはキャッシュに個別にロードされますが、複数のオブジェクトが 1 セットとしてキャッシュにロードされる場合があります。その主な例は、データベースからの 1 回の読取りでキャッシュされたオブジェクトが複数作成される場合です。この場合は、CacheLoader.load メソッドの 1 回のコールで複数のオブジェクトを作成する方が効率的です。

この使用例をサポートするために、抽象クラス CacheListLoader と CacheAccess.loadList メソッドが追加されています。CacheListLoader オブジェクトは、抽象メソッド loadList とヘルパー・メソッド getNextObject、getList、getNamedObject および saveObject を定義して CacheLoader オブジェクトを拡張します。キャッシュ・ユーザーは CacheListLoader.loadList メソッドを実装します。ヘルパー・メソッドを使用すると、ユーザーはオブジェクト・リストを反復してオブジェクトを個別に作成し、キャッシュに保存できます。CacheLoader に定義されているヘルパー・メソッドが CacheListLoader メソッドから使用される場合は、最初に getNextObject または getNamedObject をコールして正しいコンテキストを設定する必要があります。

CacheAccess.loadList メソッドは、ロードされるオブジェクト名の配列を引数として取ります。キャッシュは、このオブジェクト配列を処理します。現在キャッシュ内に存在しないオブジェクトは、キャッシュされたオブジェクトに対して定義されている CacheListLoader オブジェクトに渡されるリストに追加されます。オブジェクトに対して CacheListLoader オブジェクトが定義されていない場合、または異なる CacheListLoader オブジェクトが定義されている場合、各オブジェクトは定義済の CacheLoader.load メソッドを使用して個別にロードされます。

最善の方法は、CacheListLoader.loadList メソッドと CacheListLoader.load メソッドの両方を実装することです。どちらのメソッドがコールされるかは、ユーザーがキャッシュに対してリクエストする順序によって決定されます。たとえば、CacheAccess.loadList メソッドの前に CacheAccess.get メソッドがコールされると、CacheAccess.loadList メソッドではなく CacheListLoader.load メソッドが使用されます。

利便性を考慮して、無効化メソッドと破棄メソッドはオブジェクト配列も処理するようにオーバーロードされています。

例 9-6 にサンプル CacheListLoader、例 9-7 にその使用例を示します。

例 9-6 サンプル CacheListLoader

```
Public class ListLoader extends CacheListLoader
{
   public void loadList(Object handle, Object args) throws CacheException
   {
      while(getNextObject(handle) != null)
      {
         // create the cached object based on the name of the object
         Object  cacheObject = myCreateObject(getName(handle));
         saveObject(handle, cacheObject);
      }
   }

   public Object load(Object handle, Object args) throws CacheException
   {
      return myCreateObject(getName(handle));
   }

   private Object myCreateObject(Object name)
   {
      // do whatever to create the object
   }
}
```

例 9-7 使用例

```
// Assumes the cache has already been initialized

CacheAccess   cacc;
Attributes    attr;
ListLoader    loader = new
ListLoader();
String        objList[];
Object        obj;

// set the CacheListLoader for the region
attr  = new Attributes();
attr.setLoader(loader);

//define the region and get access to the cache
CacheAccess.defineRegion("region name", attr);
cacc = CacheAccess.getAccess("region name");

// create the array of object names
objList = new String[3];
for (int j = 0; j < 3; j++)
      objList[j] = "object " + j;

// load the objects in the cache via the CacheListLoader.loadList method
cacc.loadList(objList);

// retrieve the already loaded object from the cache
obj = cacc.get(objList[0]);

// do something useful with the object

// load an object using the CacheListLoader.load method
obj = cache.get("another object")

// do something useful with the object
```

Java Object Cache の構成

Java Object Cache は、OC4J の起動時に自動的に初期化されません。次の例に示すように、opmn.xml で -Doracle.ias.jcache=true を使用して、OC4J に jcache を初期化させることができます。

```
   <ias-component id="OC4J">
            <process-type id="home" module-id="OC4J" status="enabled">
              <module-data>
                  <category id="start-parameters">
                    <data id="java-options" value="-server
-Djava.security.policy=$ORACLE_HOME/j2ee/home/config/java2.policy
 -Djava.awt.headless=true
-DApplicationServerDebug=true
-Ddatasource.verbose=true
-Djdbc.debug=true -Doracle.ias.jcache=true"/>
                  </category>
                  <category id="stop-parameters">
                    <data id="java-options"
value="-Djava.security.policy=$ORACLE_HOME/j2ee/home/config/java2.
policy -Djava.awt.headless=true"/>
                  </category>
              </module-data>
              <start timeout="600" retry="2"/>
              <stop timeout="120"/>
              <restart timeout="720" retry="2"/>
              <port id="ajp" range="3301-3400"/>
```

```
          <port id="rmi" range="3201-3300"/>
          <port id="jms" range="3701-3800"/>
          <process-set id="default_island" numprocs="1"/>
        </process-type>
    </ias-component>
```

OC4J ランタイムは、javacache.xml ファイルに定義されている構成設定を使用して、Java Object Cache を初期化します。ファイルのパスは、OC4J の server.xml ファイルの <javacache-config> タグに指定されます。server.xml では、javacache.xml の相対パスのデフォルト値は次のとおりです。

```
<javacache-config path="../../../javacache/admin/javacache.xml"/>
```

javacache.xml の記述ルールとデフォルトの構成値は、XML Schema 内で指定されます。XML Schema ファイル ora-cache.xsd とデフォルトの javacache.xml は、$ORACLE_HOME/javacache/admin ディレクトリ（UNIX の場合）および %ORACLE_HOME%¥javacache¥admin ディレクトリ（Windows の場合）にあります。

以前のリリース（リリース 10*g*（9.0.4）より前）の Java Object Cache では、構成には javacache.properties ファイルを使用していました。リリース 10*g*（9.0.4）以上は、Java Object Cache の構成に javacache.xml を使用します。

注意： javacache.properties を使用するリリース（リリース 10*g*（9.0.4）より前）と javacache.xml を使用するリリース（リリース 10*g*（9.0.4）以上）の両方を同じホストにインストールする場合は、javacache.xml の discovery-port 属性と javacache.properties の coordinatorAddress 属性が同じポートに構成されていないことを確認する必要があります。同じポートに構成されている場合は、どちらか一方の値を異なるポート番号に手動で変更してください。デフォルトの範囲は 7000 ～ 7099 です。

構成例を次に示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<cache-configuration
xmlns=http://www.oracle.com/oracle/ias/cache/configuration
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.oracle.com/oracle/ias/cache/configuration
ora-cache.xsd">
  <logging>
    <location>javacache.log</location>
    <level>ERROR</level>
  </logging>
  <communication>
    <isDistributed>true</isDistributed>
    <discoverer discovery-port="7000"/>
  </communication>
  <persistence>
    <location>diskcache</location>
    <disksize>32</disksize>
  </persistence>
  <max-objects>1000</max-objects>
  <max-size>48</max-size>
  <clean-interval>30</clean-interval>
</cache-configuration>
```

表 9-5 に、有効なプロパティ名と、各プロパティの有効な型を示します。

表 9-5 Java Object Cache の構成プロパティ

構成の XML 要素	説明	型
`clean-interval`	各キャッシュのクリーンの間隔を秒で指定します。Java Object Cache は、キャッシュのクリーンの間隔で、オブジェクトに関連付けられている `TimeToLive` 属性または `IdleTime` 属性によって無効化されたオブジェクトがないかチェックします（これらの属性については、表 9-3 を参照してください）。 **デフォルト値：**`60`	正の整数
`ping-interval`	リモート・キャッシュ・システムの可用性を判断するために、各キャッシュの終了が検出される間隔を秒単位で指定します。 **デフォルト値：**`60`	正の整数
`max-size`	Java Object Cache で使用可能なメモリーの最大サイズを MB で指定します。 **デフォルト値：**`10`	正の整数
`max-objects`	キャッシュで許可されるメモリー内オブジェクトの最大数を指定します。この件数には、グループ・オブジェクト、ディスクにスプールされているオブジェクトまたは現在メモリーにないオブジェクトは含まれません。 **デフォルト値：**`5000`	正の整数
`region-name-separator`	親リージョン名と子リージョン名を区切るセパレータを指定します。9-25 ページの「例」を参照してください。 **デフォルト値：**`/`	`String`
`preload-file`	宣言的なキャッシュ構成ファイルへのフルパスを指定します。このファイルのフォーマットは、宣言的なキャッシュ・スキーマ（`cache.xsd`）に準拠する必要があります。宣言的なキャッシュ構成では、システムは Java Object Cache サービスの初期化時に、キャッシュのリージョン、グループ、オブジェクト、属性およびポリシーを事前に定義できます。宣言的なキャッシュの詳細は、9-27 ページの「宣言的なキャッシュ」を参照してください。9-25 ページの「例」も参照してください。 **注意：**宣言的なキャッシュの XML Schema のファイル・パスは、*`ORACLE_HOME`*`/javacache/admin/cache.xsd` です。宣言的なキャッシュ・ファイルを記述する場合は、XML Schema を参照してください。 **デフォルト値：**宣言的なキャッシュは使用されません。	`String`

表 9-5 Java Object Cache の構成プロパティ（続き）

構成の XML 要素	説明	型
communication	キャッシュが分散されているかどうかを示します。分散キャッシングの使用時に、Java Object Cache がキャッシング・システムに加わるために最初に接続する IP アドレスとポートを指定します。 distribute プロパティがオブジェクトに設定されている場合、そのオブジェクトの更新および無効化は、Java Object Cache で認識されている他のキャッシュに伝播されます。 communication 要素の isDistributed サブ要素が false に設定されていると、オブジェクトの属性セットが分散に設定されている場合でも、すべてのオブジェクトがローカルとして処理されます。9-25 ページの「例」を参照してください。 **デフォルト値 :** キャッシュは分散されません（isDistributed サブ要素は false に設定されます）。	複合（サブ要素あり）
logging	ログ・ファイル名やログ・レベルなどのログ出力属性を指定します。ログ・レベルに使用可能なオプションは、OFF、FATAL、ERROR、DEFAULT、WARNING、TRACE、INFO および DEBUG です。9-25 ページの「例」を参照してください。 **デフォルトのログ・ファイル名 :** UNIX の場合 : $ORACLE_HOME/javacache/admin/logs/javacache.log Windows の場合 : %ORACLE_HOME%¥javacache¥admin¥logs¥javacache.log **デフォルトのログ・レベル :** DEFAULT	複合（サブ要素あり）
persistence	ディスク・キャッシュのルートへの絶対パスやディスク・キャッシュの最大サイズなど、ディスク・キャッシュ構成を指定します。ルート・パスを指定すると、ディスク・キャッシュのデフォルト最大サイズは 10MB となります。ディスク・キャッシュのサイズは MB 単位です。9-25 ページの「例」を参照してください。 **デフォルト値 :** ディスク・キャッシングは使用できません。	複合（サブ要素あり）

注意 : 構成プロパティは、Attributes クラスを使用して指定する Java Object Cache の属性とは異なります。

例

次の例に、<preload-file> 要素の使用方法を示します。

■ 宣言的なキャッシュ構成ファイルを指定します。

```
<preload-file>/app/oracle/javacache/admin/decl.xml</preload-file>
```

次の例に、<communication> 要素の使用方法を示します。

■ 分散キャッシュをオフにします。

```
<communication>
  <isDistributed>false</isDistributed>
</communication>
```

■ ローカル・マシンの複数の JVM 間でキャッシュを分散します。

```
<communication>
  <isDistributed>true</isDistributed>
</communication>
```

- Java Object Cache がローカル・ノードのキャッシング・システムに加わるために最初に接続する初期検出ポートを指定します。

```
<communication>
  <isDistributed>true</isDistributed>
  <discoverer discovery-port="7000">
</communication>
```

- Java Object Cache がキャッシング・システムに加わるために最初に接続する IP アドレスと初期検出ポートを指定します。

```
<communication>
<isDistributed>true</isDistributed>
<discoverer ip="192.10.10.10" discovery-port="7000">
</communication>
```

- Java Object Cache がキャッシング・システムに加わるために最初に接続する、複数の IP アドレスと初期検出ポートを指定します。最初に指定したアドレスにアクセスできない場合は、次に指定したアドレスに接続されます。

```
<communication>
  <isDistributed>true</isDistributed>
  <discoverer ip="192.10.10.10" discovery-port="7000">
  <discoverer ip="192.11.11.11" discovery-port="7000">
  <discoverer ip="192.22.22.22" discovery-port="7000">
  <discoverer ip="192.22.22.22" discovery-port="8000">
</communication>
```

次の例に、`<persistence>` 要素の使用方法を示します。

- デフォルトのディスク・サイズを使用してディスク・キャッシュのルート・パスを指定します。

```
<persistence>
  <location>/app/9iAS/javacache/admin/diskcache</location>
</persistence>
```

- ディスク・サイズが 20MB のディスク・キャッシュのルート・パスを指定します。

```
<persistence>
  <location>/app/9iAS/javacache/admin/diskcache</location>
  <disksize>20</disksize>
</persistence>
```

次の例に、`<logging>` 要素の使用方法を示します。

- ログ・ファイル名を指定します。

```
<logging>
<location>/app/9iAS/javacache/admin/logs/my_javacache.log</location>
</logging>
```

- ログ・レベルとして `INFO` を指定します。

```
<logging>
<location>/app/9iAS/javacache/admin/logs/my_javacache.log</location>
<level>INFO</level>
</logging>
```

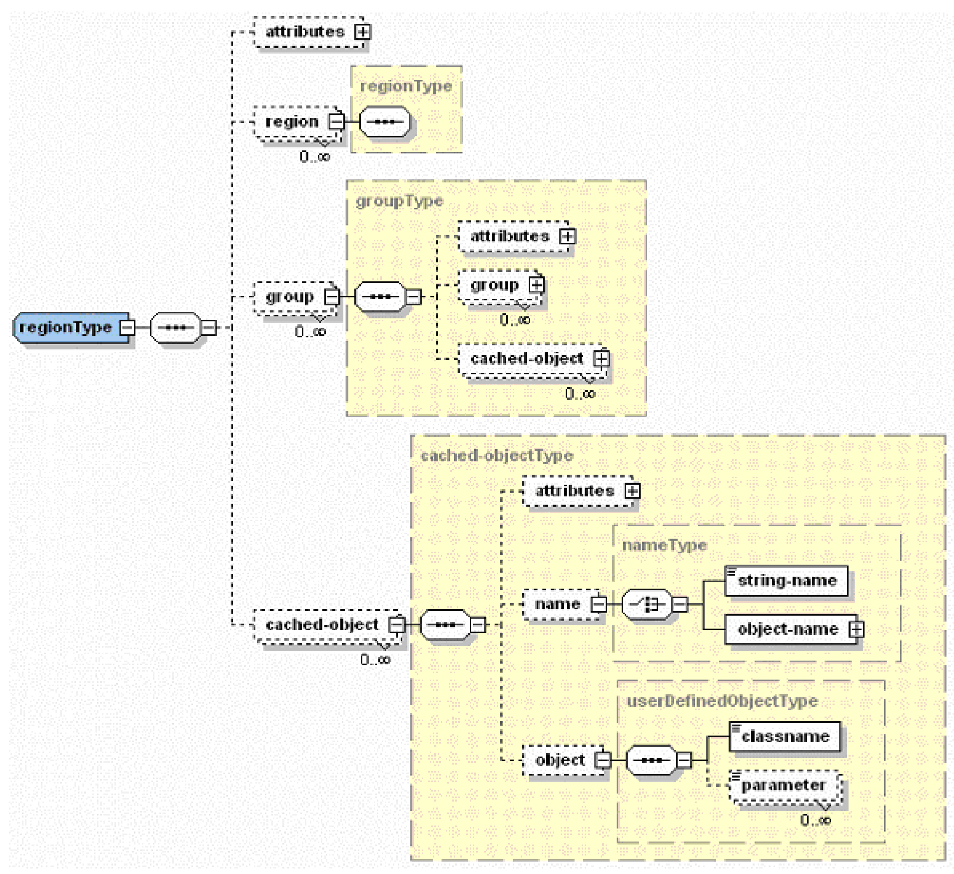
宣言的なキャッシュ

Java Object Cache 10*g* リリース 2（10.1.2）では、オブジェクト、グループ、リージョン、キャッシュ属性を宣言的に定義できます。宣言的キャッシュを使用する場合、アプリケーションでキャッシュ・オブジェクトと属性を定義するために Java コードを記述する必要はありません。

宣言的なキャッシュ・ファイルは、Java Object Cache の初期化時に自動的に読み取ることができます。宣言的なキャッシュ・ファイルの位置は、キャッシュ構成ファイルの `<preload-file>` 要素内で指定します（キャッシュ構成ファイルの構文は、9-52 ページの「OC4J サーブレットでのキャッシュ・オブジェクトの共有」を参照してください）。また、宣言的なキャッシュ・ファイルは、プログラムによってロードするか、`oracle.ias.cache.Configurator.class` のパブリック・メソッドを使用して明示的にロードできます。宣言的なキャッシュ・ファイルを複数使用することもできます。

図 9-6 に、宣言的なキャッシュを示します。

図 9-6 宣言的なキャッシュのアーキテクチャ



システムの初期化時に宣言的なキャッシュ・ファイルが自動的にロードされるように、Java Object Cache を設定できます。例 9-8 にこの設定を示します。例 9-9 には、宣言的なキャッシュ・ファイルをプログラムで読み取る方法を示します。

例 9-8 宣言的なキャッシュを自動的にロードする方法

```
<!-- Specify declarative cache file:my_decl.xml in javacache.xml -->
<cache-configuration>
  …
<preload-file>/app/9iAS/javacache/admin/my_decl.xml</preload-file>
  …
</cache-configuration>
```

例 9-9 宣言的なキャッシュ・ファイルをプログラムで読み取る方法

```
try {
  String filename = "/app/9iAS/javacache/admin/my_decl.xml";
Configurator config = new Configurator(filename);
Config.defineDeclarable();
} catch (Exception ex) {
}
```

宣言的なキャッシュ・ファイルの例

```
<?xml version="1.0" encoding="UTF-8"?>
<cache xmlns="http://www.javasoft.com/javax/cache"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.oracle.com/javax/cache">
  <region name="fruit">
    <attributes>
      <time-to-live>3000</time-to-live>
      <max-count>200</max-count>
      <capacity-policy>
        <classname>com.acme.MyPolicy</classname>
      </capacity-policy>
    </attributes>
    <group name="apple">
      <attributes>
        <flag>spool</flag>
        <flag>distribute</flag>
        <cache-loader>
          <classname>com.acme.MyLoader</classname>
          <parameter name="color">red</parameter>
        </cache-loader>
      </attributes>
    </group>
    <cached-object>
      <name>
        <string-name>theme</string-name>
      </name>
      <object>
        <classname>com.acme.DialogHandler</classname>
        <parameter name="prompt">Welcome</parameter>
      </object>
    </cached-object>
  </region>
</cache>
```

宣言的なキャッシュ・ファイルの形式

宣言的なキャッシュ・ファイルは XML 形式です。このファイルの内容は、Oracle Application Server 10*g* に付属する宣言的なキャッシュの XML Schema に準拠する必要があります。この XML Schema のファイル・パスは、`ORACLE_HOME/javacache/admin/cache.xsd` です。

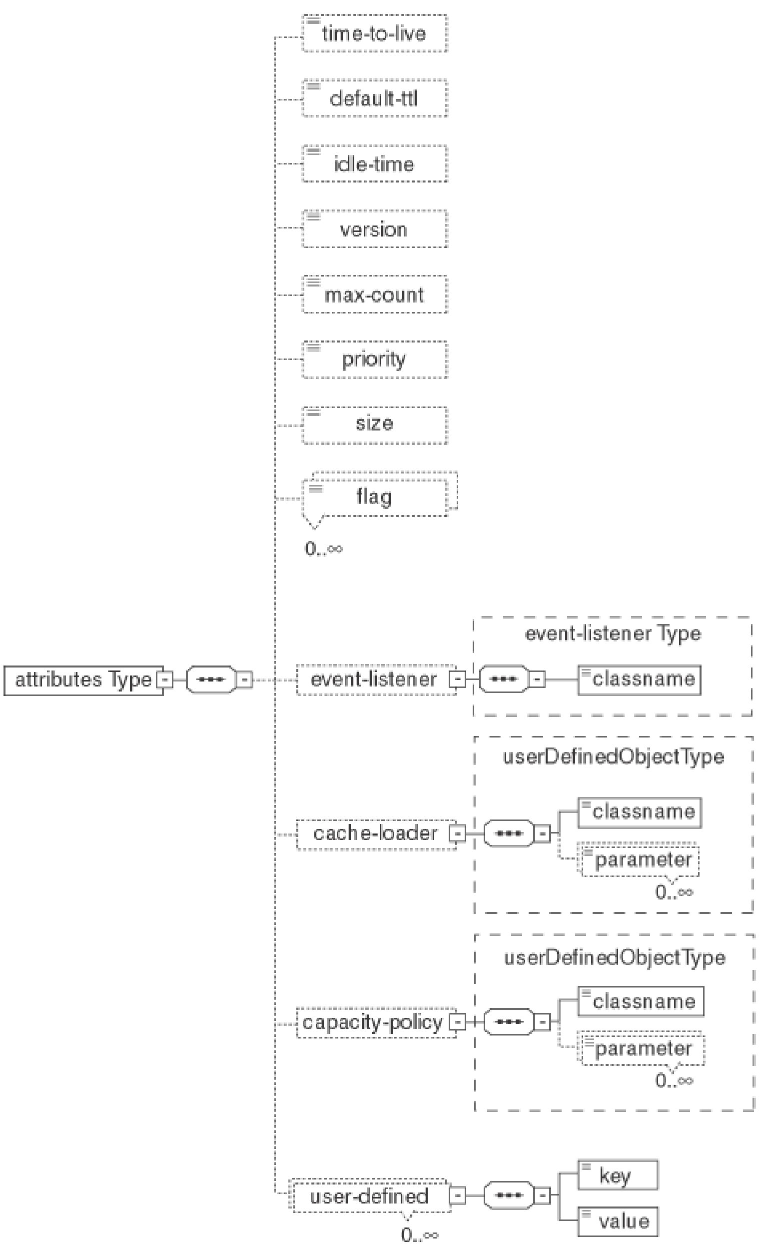
表 9-6 に、宣言的なキャッシュ・スキーマの要素、その子および各要素の有効な型を示します。ほとんどの要素の使用方法を示すコードについては、9-31 ページの「例」を参照してください。

表 9-6 宣言的なキャッシュのスキーマの説明（cache.xsd）

要素	説明	子	型
region	キャッシュのリージョンまたはサブリージョンを宣言します。	<attributes> <region> <group> <cached-object>	regionType
group	キャッシュのグループまたはサブグループを宣言します。	<attributes> <group> <cached-object>	groupType
cached-object	キャッシュ・オブジェクトを宣言します。	<attributes> <name> <object>	objectType
name	キャッシュされるオブジェクトの名前を宣言します。この名前には、単純な文字列型または指定した Java オブジェクトの型を使用できます。	<string-name> <object-name>	nameType
object	ユーザー定義の Java オブジェクトを宣言します。指定するオブジェクトのクラスでは、`oracle.ias.cache` パッケージの宣言可能なインタフェースを実装する必要があります。	<classname> <parameter>	userDefinedObjectType
attributes	キャッシュ・リージョン、グループまたはキャッシュ・オブジェクトの attributes オブジェクトを宣言します。子要素は、それぞれ `oracle.ias.cache` パッケージの attributes クラスの各フィールドに対応します。詳細は、`Attributes.class` の Javadoc を参照してください。	<time-to-live> <default-ttl> <idle-time> <version> <max-count> <priority> <size> <flag> <event-listener> <cache-loader> <capacity-policy> <user-defined>	attributesType
event-listener	`CacheEventListener` オブジェクトを宣言します。	<classname>	event-listenerType
cache-loader	`CacheLoader` オブジェクトを宣言します。	<classname> <parameter>	userDefinedObjectType
capacity-policy	`CapacityPolicy` オブジェクトを宣言します。	<classname> <parameter>	userDefinedObjectType
user-defined	ユーザー定義の文字列型属性を宣言します。	<key> <value>	element

図 9-7 に、宣言的なキャッシュ・スキーマの属性を示します。

図 9-7 宣言的なキャッシュ・スキーマの属性



例

次の例に、表 9-6 に示した要素の使用方法を示します。

- <region> 要素を使用してキャッシュのリージョンとサブリージョンを宣言します。

```
<region name="themes">
    <region name="cartoon">
      <!-- sub region definition -->
    </region>
  <group name="colors">
    <!-- group definition -->
  </group>
</region>
```

- <group> 要素を使用してキャッシュのグループとサブグループを宣言します。

```
<group name="colors">
  <group name="dark">
    <!-- sub group definition -->
  </group>
</group>
```

- <cached-object> 要素を使用して、キャッシュされるオブジェクトを宣言します。

```
<cached-object>
  <name>
    <string-name>DialogHandler</string-name>
  </name>
  <object>
    <classname>com.acme.ConfigManager</classname>
    <parameter name="color">blue</parameter>
  </object>
</cached-object>
```

- <name> 要素で文字列を使用して、キャッシュされるオブジェクトの名前を宣言します。

```
<name>
  <string-name>DialogHandler</string-name>
</name>
```

<name> 要素でオブジェクトを使用して、キャッシュされるオブジェクトの名前を宣言します。

```
<name>
  <object-name>
    <classname>DialogHandler</classname>
    <parameter name="color">green</parameter>
  </object-name>
</name>
```

- <object> 要素を使用して、ユーザー定義の Java オブジェクトを宣言します。

```
<object>
  <classname>com.acme.CustomConfigManager</classname>
  <parameter name="color">blue</parameter>
</object>

// Implementation of CustomConfigManager.java
package com.acme;
import oracle.ias.cache.Declarable;
public class CustomConfigManager implements Declarable {
}
```

- `<attributes>` 要素を使用して、キャッシュ・リージョン、グループまたはキャッシュ・オブジェクトの attributes オブジェクトを宣言します。

```
<attributes>
  <time-to-live>4500</time-to-live>
  <default-ttl>6000</default-ttl>
  <version>99</version>
  <max-count>8000</max-count>
  <priority>50</priority>
  <flag>spool</flag>
  <flag>allownull</flag>
  <flag>distribute</flag>
  <flag>reply</flag>
  <cache-loader>
    <classname>MyLoader</classname>
    <parameter name="debug">false</parameter>
  </cache-loader>
</attributes>
```

- `<user-defined>` 要素を使用して、ユーザー定義の文字列型の属性を宣言します。

```
<attributes>
  <user-defined>
    <key>color</key>
    <value>red</value>
  </user-defined>
</attributes>
```

宣言可能なユーザー定義オブジェクト

キャッシュ・オブジェクト、オブジェクト属性およびユーザー定義オブジェクトのトポロジは、すべて宣言的なキャッシュ・ファイルに記述できます。宣言的なキャッシュ・ファイルで宣言されているユーザー定義の Java オブジェクト（`CacheLoader`、`CacheEventListener` および `CapacityPolicy` など）がシステムでロードおよびインスタンス化されるようにするには、そのオブジェクトを `oracle.ias.cache.Declarable` インタフェースのインスタンスにする必要があります。つまり、宣言的なキャッシュ・ファイルで宣言されているすべての Java オブジェクトについて、`oracle.ias.cache.Declarable` インタフェースを実装します。すべてのユーザー定義の Java オブジェクトは、アプリケーションのクラス・ローダーではなく JVM のデフォルト・クラス・ローダーによってロードされることに注意してください。宣言可能なオブジェクトがインスタンス化されると、システムにより、その `init(Properties props)` メソッドが暗黙的に起動されます。このメソッドは、宣言的なキャッシュ・ファイルに定義されているユーザー指定のパラメータ（名前 / 値ペア）を使用して、必要な初期化タスクを実行します。例 9-10 に、パラメータ（color = yellow）で宣言的に渡すことでオブジェクトを定義する方法を示します。

例 9-10 パラメータで宣言的に渡すことによるオブジェクトの定義

宣言的な XML ファイル内で次のように記述します。

```
<cached-object>
  <name>
    <string-name>Foo</string-name>
  </name>
  <object>
    <classname>com.acme.MyCacheObject</classname>
    <parameter name="color">yellow</parameter>
  </object>
</cached-object>
```

宣言可能なオブジェクトの実装は次のとおりです。

```
package com.acme;

import oracle.ias.cache.*;
```

```
import java.util.Properties;

public class MyCacheObject implements Declarable {

  private String color_;

  /**
    * Object initialization
    */
  public void init(Properties prop) {
         color_ = prop.getProperty("color");
  }
}
```

宣言可能な CacheLoader、CacheEventListener および CapacityPolicy

宣言的なキャッシュ・ファイル内で CacheLoader、CacheEventListener または CapacityPolicy オブジェクトを指定する場合は、そのオブジェクト自体が oracle.ias.cache.Declarable のインスタンスでもあることが必要です。この要件は、ユーザー定義オブジェクトの要件と同様です。必要な抽象クラスを拡張する他に、指定のオブジェクトごとに宣言可能なインタフェースを実装する必要があります。例 9-11 に、宣言可能な CacheLoader の実装を示します。

例 9-11 宣言可能な CacheLoader の実装

```
import oracle.ias.cache.*;
import java.util.Properties;

public class MyCacheLoader extends CacheLoader implements Declarable {

  public Object load(Object handle, Object argument) {
    // should return meaningful object based on argument
    return null;
  }

  public void init(Properties prop) {
  }
}
```

非 OC4J コンテナでの Java Object Cache の初期化

Java アプリケーション内で Java Object Cache を使用して非 OC4J ランタイムで実行するには、アプリケーション（Java クラス）が初期化される場所に次の参照を挿入する必要があります。

```
Cache.open(/path-to-ocnfig-file/javacache.xml);
```

コードでパラメータを指定せずに Cache.open() を起動すると、Java Object Cache では内部のデフォルト構成パラメータが使用されます。また、Cache.init(CacheAttributes) を起動して Java Object Cache を初期化することもできます。これにより、独自の構成ファイルから構成パラメータを導出するか、プログラム的に生成できます。

OC4J ランタイムで Java Object Cache が使用されない場合は、JVM が起動される CLASSPATH に cache.jar を組み込む必要があります。また、Cache.open(String *config_filename*) を起動するか（*config_filename* は有効な javacache.xml ファイルへのフルパス）、Cache.init(CacheAttributes) を起動して、Java Object Cache を明示的に初期化します。

次のいずれかのメソッド起動を使用して、非 OC4J コンテナ内で Java Object Cache を明示的に初期化します。

- `Cache.open();`

  `cache.jar` ファイルに格納されているデフォルトの Java Object Cache 構成を使用します。

- `Cache.open(`*`/path-to-oracle-home/javacache/admin/javacache.xml`*`);`

  `javacache.xml` ファイルに定義されている構成を使用します。

- `Cache.open(`*`/path-to-user's-own-javacache.xml`*`);`

  特定の `javacache.xml` ファイルに定義されている構成を使用します。

- `Cache.init(CacheAttributes);`

  `CacheAttributes` オブジェクト内で設定されている構成を使用します。

OC4J コンテナ内で実行される J2EE アプリケーションの場合、`javacache.xml` ファイルへのパスは OC4J の `server.xml` 構成ファイル内で構成できます。キャッシュは、OC4J プロセスの起動時に自動的に初期化できます。詳細は、OC4J の構成を参照してください。

非 OC4J コンテナでは、前述のメソッド起動を使用しない場合、`Cache.getAccess()` または `Cache.defineRegion()` を起動すると、Java Object Cache が（`cache.jar` に格納されているデフォルトの構成設定を使用して）暗黙的に初期化されます。

容量制御

新しい容量制御機能を使用すると、キャッシュ・ユーザーは、キャッシュ、リージョンまたはグループの容量に達したときに、どのオブジェクトをキャッシュから削除するかを決定するためのポリシーを指定できます。ポリシーを指定するには、抽象クラス `CapacityPolicy` を拡張し、キャッシュ、リージョンまたはグループの属性としてインタンス化されたオブジェクトを設定します。

リージョンおよびグループの場合は、そのリージョンまたはグループが容量に達し、新規オブジェクトがロードされるときに、`CapacityPolicy` オブジェクトがコールされます。リージョンまたはグループ内で無効化するオブジェクトが見つからないと、新規オブジェクトはキャッシュに保存されません（ユーザーには戻されますが、即時に無効化されます）。

キャッシュの容量がなんらかの最高水位標（構成された最大使用率）に達した場合、キャッシュ全体に関連付けられている `CapacityPolicy` オブジェクトがコールされます。最高水位標に達すると、キャッシュはオブジェクトを削除してキャッシュ内のロードを最高水位標より 3% 下げようとします。この最高水位標は、`capacityBuffer` キャッシュ属性で指定されます。`capacityBuffer` が 5 に設定されている場合、キャッシュは使用率が 95%（100% - 5%）になるとオブジェクトの削除を開始し、使用率が 92%（95% - 3%）になるまで削除を続行します。`capacityBuffer` のデフォルト値は 15 です。

キャッシュには、特定のリージョンまたはグループに使用するものとは異なる容量ポリシーを使用できます。

デフォルトで、グループおよびリージョンに対する容量ポリシーでは、新規オブジェクトが追加されるときに容量に達している場合は、同等以下の優先順位を持つ参照されていないオブジェクトが削除されます。キャッシュの場合、デフォルト・ポリシーでは、オブジェクトの優先順位に従って過去 2 回のクリーン間隔中に参照されていないオブジェクトが削除されます。つまり、優先順位が低く最近参照されていないオブジェクトから順番に削除されます。

容量ポリシーを作成しやすいように、キャッシュ内のオブジェクトに関して多数の統計が保持され、キャッシュ、リージョンおよびグループ間で集計されます。この統計は `CapacityPolicy` オブジェクトで使用できます。キャッシュ・オブジェクトの場合は、次の統計が保持されます。

- 優先順位。
- アクセス回数 : オブジェクトが参照された回数。
- サイズ : オブジェクトのバイト数（使用可能な場合）。
- 最終アクセス時間 : オブジェクトが最後にアクセスされた時間（ミリ秒）。

- 作成時間 : オブジェクトが作成された時間（ミリ秒）。
- ロード時間 : オブジェクトのミリ秒単位のロード所要時間（オブジェクトが CacheAccess.put でキャッシュに追加された場合、この値は 0（ゼロ）です）。

これらの統計とともに、オブジェクトに関連付けられている属性すべてを CapacityPolicy オブジェクトで使用できます。

キャッシュ、リージョンおよびグループの場合は、次の集計統計が保持されます。これらの統計ごとに、下限、上限および平均値が保持されます。これらの統計は、クリーン間隔ごと、または Cache.updateStats() のコール時に再計算されます。

- 優先順位。
- アクセス回数 : オブジェクトが参照された回数。
- サイズ : オブジェクトのバイト数（使用可能な場合）。
- 最終アクセス時間 : オブジェクトが最後にアクセスされた時間（ミリ秒）。
- ロード時間 : オブジェクトのミリ秒単位のロード所要時間（オブジェクトが CacheAccess.put でキャッシュに追加された場合、この値は 0（ゼロ）です）。

例 9-12 に、オブジェクト・サイズに基づくリージョンのサンプル CapacityPolicy オブジェクトを示します。

例 9-12 オブジェクト・サイズに基づくサンプル CapacityPolicy

```
class SizePolicy extends CapacityPolicy
{
   public boolean policy (Object victimHandle, AggregateStatus aggStatus,
    long currentTime , Object newObjectHandle) throws CacheException
   {
      int             newSize;
      int             oldSize;

      oldSize = getAttributes(victimHandle).getSize();
      newSize = getAttributes(newObjectHandle).getSize();
      if (newSize >= oldSize)
         return true;
      return false;
   }
```

例 9-13 に、アクセス時間と参照回数に基づくキャッシュのサンプル CapacityPolicy を示します。オブジェクトの参照回数が平均値を下回っており、過去 30 秒アクセスされていないと、キャッシュから削除されます。

例 9-13 アクセス時間と参照回数に基づくサンプル CapacityPolicy

```
class SizePolicy extends CapacityPolicy
{
public boolean policy (Object victimHandle, AggregateStatus aggStatus, long
 currentTime , Object newObjectHandle) throws CacheException
{
   long          lastAccess;
   int           accessCount;
   int           avgAccCnt;

   lastAccess    = getStatus(victimHandle).getLastAccess();
   accessCount   = getStatus(victimHandle).getAccessCount();
   avgAccCnt     = aggStatus.getAccessCount(AggregateStatus.AVG);

   if (lastAccess + 30000 < currentTime && accessCount < avgAccCnt)
      return true;
   }

}
```

キャッシュ・イベント・リスナーの実装

キャッシュ内のオブジェクトのライフ・サイクルでは、オブジェクトの作成やオブジェクトの無効化など、多数のイベントが発生する可能性があります。この項では、キャッシュ・イベントが発生したときのアプリケーションへの通知方法について説明します。

オブジェクトの作成の通知を受信するには、cacheLoader の一部としてイベント通知を実装します。無効化または更新の通知の場合は、CacheEventListener を実装し、Attributes.setCacheEventListener() を使用して CacheEventListener をオブジェクト、グループ、リージョンまたはサブリージョンに関連付けます。

CacheEventListener は、java.util.EventListener を拡張するインタフェースです。キャッシュ・イベント・リスナーには、登録済のコールバック・メソッドを設定する機能があり、イベント発生時に実行されます。Java Object Cache では、イベント・リスナーは、キャッシュ内のオブジェクトが無効化または更新された場合に実行されます。

イベント・リスナーは、キャッシュ内のオブジェクト、グループ、リージョンまたはサブリージョンに関連付けられます。イベント・リスナーがグループ、リージョンまたはサブリージョンに関連付けられた場合、リスナーは、デフォルトでそのグループ、リージョンまたはサブリージョン自体が無効化されたときにのみ実行されます。その中のメンバーが無効化された場合、イベントはトリガーされません。Attributes.setCacheEventListener() のコールは Boolean 引数を使用します。この値が true の場合、イベント・リスナーは、リージョン、サブリージョンまたはグループ自体ではなく、リージョン、サブリージョンまたはグループの各メンバーに適用されます。この場合、リージョン、サブリージョンまたはグループ内のオブジェクトが無効化されると、イベントがトリガーされます。

CacheEventListener インタフェースには、1 つのメソッド handleEvent() があります。このメソッドは、1 つの引数 CacheEvent オブジェクト（java.util.EventObject を拡張します）を使用します。このオブジェクトには 2 つのメソッドがあり、getID() はイベントのタイプ（OBJECT_INVALIDATION または OBJECT_UPDATED）を戻し、getSource() は無効化されるオブジェクトを戻します。グループおよびリージョンの場合、getSource() メソッドはグループ名またはリージョン名を戻します。

handleEvent() メソッドは、Java Object Cache が管理するバックグラウンド・スレッドのコンテキスト内で実行されます。必要なスレッド・コンテキストが使用可能でない場合があるため、このメソッドでは Java ネイティブ・インタフェース（JNI）コードを使用しないでください。

例 9-14 に、CacheEventListener を実装し、オブジェクトまたはグループに関連付ける方法を示します。

例 9-14 CacheEventListener の実装

```
import oracle.ias.cache.*;
   // A CacheEventListener for a cache object
   class MyEventListener implements
   CacheEventListener  {

       public void handleEvent(CacheEvent ev)
       {
          MyObject obj = (MyObject)ev.getSource();
          obj.cleanup();
        }

       // A CacheEventListener for a group object
       class MyGroupEventListener implements CacheEventListener {
       public void handleEvent(CacheEvent ev)
       {
          String groupName = (String)ev.getSource();
          app.notify("group " + groupName + " has been invalidated");

       }
   }
```

Attributes.listener 属性を使用して、リージョン、サブリージョン、グループまたはオブジェクトの CacheEventListener を指定します。

例 9-15 に、オブジェクトにキャッシュ・イベント・リスナーを設定する方法を示します。また、例 9-16 に、グループにキャッシュ・イベント・リスナーを設定する方法を示します。

例 9-15 オブジェクトのキャッシュ・イベント・リスナーの設定

```
import oracle.ias.cache.*;

   class YourObjectLoader extends CacheLoader
   {
      public YourObjectLoader () {
      }

      public Object load(Object handle, Object args) {
         Object obj = null;
         Attributes attr = new Attributes();
         MyEventListener el = new MyEventListener();
         attr.setCacheEventListener(CacheEvent.OBJECT_INVALIDATED, el);

         // your implementation to retrieve or create your object

         setAttributes(handle, attr);
         return obj;
     }
}
```

例 9-16 グループのキャッシュ・イベント・リスナーの設定

```
import oracle.ias.cache.*;
try
{
   CacheAccess cacc = CacheAccess.getAccess(myRegion);
   Attributes attr = new Attributes ();

   MyGroupEventListener listener = new MyGroupEventListener();
   attr.setCacheEventListener(CacheEvent.OBJECT_INVALIDATED, listener);

   cacc.defineGroup("myGroup", attr);
   //....
   cacc.close();

}catch(CacheException ex)
{
   // handle exception
}
```

制限事項およびプログラミングに関する注意点

この項では、Java Object Cache を使用するときの制限事項およびプログラミングに関する注意点について説明します。

- CacheAccess オブジェクトは、スレッド間で共有しないでください。このオブジェクトは、キャッシング・システムに対するユーザーを表します。CacheAccess オブジェクトには、キャッシュに対するユーザー・アクセスの現在の状態（現在アクセスされているオブジェクト、現在所有されているオブジェクトなど）が含まれています。CacheAccess オブジェクトを共有する必要はなく、共有した場合の結果は予測できません。
- CacheAccess オブジェクトは、同時に 1 つのキャッシュ済オブジェクトに対する参照のみを保持します。複数のキャッシュ済オブジェクトが同時にアクセスされる場合は、複数の CacheAccess オブジェクトを使用します。メモリーに格納されるオブジェクトについては、この作業は重要ではありません。これは、Java では、キャッシュが参照されていない場合でも、キャッシュ済オブジェクトはガベージ・コレクションの対象とならないためです。ディスク・オブジェクトについては、キャッシュ参照がメンテナンスされない場合は、基礎となるファイルが別のユーザーや時間ベースの無効化によって削除され、予期しない例外が発生する可能性があります。リソース管理を最適化するには、キャッシュ済オブジェクトが使用されている間、キャッシュ参照をオープンのままにします。
- CacheAccess オブジェクトは、使用しなくなったときは必ずクローズしてください。CacheAccess オブジェクトはプールされます。これらは、ユーザーのかわりにキャッシュ・リソースを取得します。アクセス・オブジェクトが使用されなくなったときにクローズされないと、これらのリソースがプールに戻されず、JVM によってガベージ・コレクションの対象となるまでクリーン・アップされません。CacheAccess オブジェクトが絶えず割り当てられ、クローズされない場合は、パフォーマンスが低下することがあります。
- ローカル・オブジェクト（Attributes.DISTRIBUTE 属性が設定されていないオブジェクト）が CacheAccess.save() メソッドを使用してディスクに保存された場合は、プロセスの終了後、このオブジェクトは存続しません。定義により、ローカル・オブジェクトを参照できるのは、そのオブジェクトがロードされたキャッシュ・インスタンスのみです。そのキャッシュ・インスタンスがなんらかの理由で消失した場合、管理対象のオブジェクトは、ディスクに存在している場合でも失われます。プロセスの終了後もオブジェクトが存続する必要がある場合は、オブジェクトとキャッシュの両方を DISTRIBUTE として定義する必要があります。
- キャッシュ構成（キャッシュ環境とも呼ばれます）はキャッシュに固有で、リージョン、サブリージョン、グループおよびオブジェクトの定義が含まれます。キャッシュ構成はディスクには保存されず、他のキャッシュには伝播されません。アプリケーションの初期化時に、キャッシュ構成を定義します。
- CacheAccess.waitForResponse() または CacheAccess.releaseOwnership() メソッドのコールがタイムアウトになった場合は、正常に戻されるまでコールを繰り返す必要があります。CacheAccess.waitForResponse() に失敗した場合は、CacheAccess.cancelResponse をコールしてリソースを解放します。CacheAccess.releaseOwnership() に失敗した場合は、タイムアウト値に -1 を指定して CacheAccess.releaseOwnership をコールし、リソースを解放します。
- グループまたはリージョンが破棄または無効化されたときは、分散定義がローカル定義より優先されます。つまり、グループが分散されている場合、個々のオブジェクトまたは関連グループがローカルとして定義されている場合でも、グループまたはリージョン内のすべてのオブジェクトが、キャッシュ・システム全体で無効化または破棄されます。グループまたはリージョンがローカルとして定義されている場合、グループ内のローカル・オブジェクトはローカルで無効化され、分散オブジェクトはキャッシュ・システム全体で無効化されます。
- オブジェクトまたはグループが SYNCHRONIZE 属性を設定して定義されている場合、オブジェクトをロードまたは置換するには所有権が必要です。ただし、オブジェクトへの一般的なアクセスまたはオブジェクトの無効化には所有権は必要ありません。

- 通常、キャッシュに格納されるオブジェクトは、ユーザー定義のクラス・ローダーではなく、JVM の初期化時に `classpath` で定義されているシステム・クラス・ローダーでロードする必要があります。特に、アプリケーション間で共有しているオブジェクト、あるいはディスクに保存またはスプールされる可能性があるオブジェクトは、システムの `classpath` で定義する必要があります。この定義を行わない場合、`ClassNotFoundException` または `ClassCastException` が発生する場合があります。
- 一部のシステムでは、オープン・ファイル記述子がデフォルトで制限されている場合があります。これらのシステムでは、システム・パラメータを変更してパフォーマンスを向上させる必要があります。たとえば、UNIX システムでは、オープン・ファイル記述子の数の適切な値は、`1024` 以上です。
- ローカル・モードまたは分散モードで構成されている場合は、起動時に、1 つのアクティブな Java Object Cache キャッシュが JVM プロセス（Java Object Cache API を使用する JVM で動作しているプログラム）に作成されます。

ディスク・オブジェクトの操作

Java Object Cache は、メモリー内のオブジェクトのみでなく、ディスク上のオブジェクトも管理できます。

この項には、次の項目が含まれます。

- ローカルおよび分散ディスク・キャッシュ・オブジェクト
- ディスク・キャッシュへのオブジェクトの追加

ローカルおよび分散ディスク・キャッシュ・オブジェクト

この項には、次の項目が含まれます。

- ローカル・オブジェクト
- 分散オブジェクト

ローカル・オブジェクト

ローカル・モードで作業しているときは、オブジェクトに `DISTRIBUTE` 属性が設定されている場合でも、キャッシュ属性 `isDistributed` は設定されず、すべてのオブジェクトがローカル・オブジェクトとして処理されます。ローカル・モードでは、ディスク・キャッシュのオブジェクトはすべて、そのオブジェクトをロードした Java Object Cache キャッシュでのみ参照でき、プロセスの終了後は存続しません。ローカル・モードでは、ディスク・キャッシュに格納されたオブジェクトは、そのキャッシュを使用しているプロセスが終了すると失われます。

分散オブジェクト

キャッシュ属性 `isDistributed` が true に設定されている場合、キャッシュは分散モードで動作します。ディスク・キャッシュ・オブジェクトは、そのディスク・キャッシュをホスティングしているファイル・システムにアクセスできるすべてのキャッシュで共有できます。これは、構成されているディスク・キャッシュの位置により決定されます。この構成では、ディスク・リソースの使用率が高くなり、ディスク・オブジェクトは、Java Object Cache プロセスの終了後も存続します。

ディスク・キャッシュに格納されているオブジェクトは、`diskPath` 構成プロパティに指定されたパスと、ファイルの残りのパスを表す内部的に生成された `String` の組合せで識別されます。したがって、`diskPath` が物理ディスク上の同一ディレクトリを表し、Java Object Cache プロセスでアクセスできるかぎり、ディスク・キャッシュを共有するキャッシュのディレクトリ構造は異なっていてもかまいません。

ディスクに保存されているメモリー・オブジェクトが分散オブジェクトでもある場合、メモリー・オブジェクトは、それをスプールしたプロセスの終了後も存続できます。

ディスク・キャッシュへのオブジェクトの追加

Java Object Cache でディスク・キャッシュを使用するには、次のようにいくつかの方法があります。

- オブジェクトの自動的な追加
- オブジェクトの明示的な追加
- ディスク・キャッシュにのみ存在するオブジェクトの使用方法

オブジェクトの自動的な追加

Java Object Cache は、特定のオブジェクトをディスク・キャッシュに自動的に追加します。これらのオブジェクトは、メモリー・キャッシュまたはディスク・キャッシュに常駐します。ディスク・キャッシュ内のオブジェクトが必要な場合は、コピーされてメモリー・キャッシュに戻されます。ディスクへのスプーリングの操作は、Java Object Cache によって、メモリー・キャッシュに空き領域が必要であると判断された場合に行われます。オブジェクトのスプーリングが発生するのは、そのオブジェクトに対して SPOOL 属性が設定されている場合のみです。

オブジェクトの明示的な追加

1 つ以上のオブジェクトを Java Object Cache のディスク・キャッシュに強制的に書き込むこともできます。CacheAccess.save() メソッドを使用すると、リージョン、サブリージョン、グループまたはオブジェクトが、ディスク・キャッシュに書き込まれます（オブジェクトがディスク・キャッシュ内にすでに存在している場合は、再度書き込まれることはありません）。

注意： CacheAccess.save() を使用すると、オブジェクトに SPOOL 属性が設定されていない場合でも、オブジェクトがディスクに保存されます。

リージョン、サブリージョンまたはグループで CacheAccess.save() をコールすると、そのリージョン、サブリージョンまたはグループ内のすべてのオブジェクトがディスク・キャッシュに保存されます。CacheAccess.save() メソッドのコール時に、オブジェクトがシリアライズ可能ではない、または他の理由でディスクに書き込むことができない場合は、Java Object Cache ログにイベントが記録され、保存操作は次のオブジェクトで続行されます。各オブジェクトがディスクに書き込まれる場合、書込みは同期的に実行されます。グループまたはリージョンが保存される場合、書込みは非同期のバックグラウンド・タスクとして実行されます。

ディスク・キャッシュにのみ存在するオブジェクトの使用方法

ディスク・キャッシュからのみ直接アクセスされるオブジェクトは、CacheLoader.load() メソッドから CacheLoader.createDiskObject() をコールすると、ディスク・キャッシュにロードされます。createDiskObject() メソッドは、アプリケーションがディスク・オブジェクトのロードに使用できる File オブジェクトを戻します。そのディスク・オブジェクトに対してディスク・オブジェクトの属性が定義されていない場合は、createDiskObject() メソッドを使用して設定します。ローカル・ディスク・オブジェクトと分散ディスク・オブジェクトでは、管理方法が異なります。ローカルまたは分散のいずれであるかは、オブジェクトの作成時に、指定した属性に基づいて決定されます。

注意： 同じキャッシュ・システム内の分散キャッシュ間でディスク・キャッシュ・オブジェクトを共有する場合は、ディスク・キャッシュ・オブジェクトの作成時に DISTRIBUTE 属性を定義する必要があります。この属性は、オブジェクトの作成後は変更できません。

CacheAccess.get() がディスク・オブジェクトでコールされると、ファイルへのフルパス名が戻され、アプリケーションで必要に応じてファイルをオープンできます。

ディスク・オブジェクトはローカル・ディスクに格納され、Java Object Cache を使用してアプリケーションによってディスクから直接アクセスされます。ディスク・オブジェクトは、DISTRIBUTE 属性の設定（および Java Object Cache が分散モードとローカル・モードのどちらで動作しているか）に応じて、すべての Java Object Cache プロセスによって共有される場合と、特定のプロセスに限定される場合があります。

例 9-17 は、ディスク・オブジェクトをキャッシュにロードするローダー・オブジェクトを示しています。

例 9-17 CacheLoader でのディスク・オブジェクトの作成

```
import oracle.ias.cache.*;

class YourObjectLoader extends CacheLoader
{
   public Object load(Object handle, Object args) {
      File file;
      FileOutputStream = out;
      Attributes attr = new Attributes();

      attr.setFlags(Attributes.DISTRIBUTE);
      try
      // The distribute attribute must be set on the createDiskObject method
      {
         file = createDiskObject(handle, attr);
         out = new FileOutputStream(file);

         out.write((byte[])getInfofromsomewhere());
         out.close();
     }
     catch (Exception ex) {
       // translate exception to CacheException, and log exception
         throw exceptionHandler("exception in file handling", ex)
      }
      return file;
      }
   }
```

例 9-18 に、Java Object Cache のディスク・オブジェクトを使用するアプリケーション・コードを示します。この例では、Stock-Market というリージョンが、例 9-17 でそのリージョンのデフォルト・ローダーとして設定された YourObjectLoader ローダーによって、すでに定義されていると仮定しています。

例 9-18 ディスク・オブジェクトを使用するアプリケーション・コード

```
import oracle.ias.cache.*;

try
{
   FileInputStream in;
   File file;
   String filePath;
   CacheAccess cacc = CacheAccess.getAccess("Stock-Market");

   filePath = (String)cacc.get("file object");
   file = new File(filePath);
   in = new FileInputStream(filePath);
   in.read(buf);

// do something interesting with the data
   in.close();
   cacc.close();
}
```

```
catch (Exception ex)
{
// handle exception
}
```

StreamAccess オブジェクトの操作

StreamAccess オブジェクトはストリームとしてアクセスされ、ディスク・キャッシュに自動的にロードされます。オブジェクトは OutputStream としてロードされ、InputStream として読み取られます。サイズの小さい StreamAccess オブジェクトにはメモリーまたはディスク・キャッシュからアクセスでき、サイズの大きい StreamAccess オブジェクトはディスクから直接ストリームされます。StreamAccess オブジェクトのアクセス方法は、オブジェクトのサイズおよびキャッシュの容量に基づいて、Java Object Cache によって自動的に決定されます。

ユーザーには常に、オブジェクトがファイル内にあるか、メモリー内にあるかに関係なく、ストリーム・オブジェクト（読取りの場合は InputStream、書込みの場合は OutputStream）が示されます。StreamAccess オブジェクトを使用すると、オブジェクトのサイズやリソースの可用性に関係なく、Java Object Cache ユーザーは常に同じ方法でオブジェクトにアクセスできます。

StreamAccess オブジェクトの作成

StreamAccess オブジェクトを作成するには、オブジェクトがキャッシュにロードされるときに、CacheLoader.load() メソッドから CacheLoader.createStream() メソッドをコールします。createStream() メソッドによって OutputStream オブジェクトが戻されます。OutputStream オブジェクトを使用すると、キャッシュにオブジェクトをロードできます。

オブジェクトに対する属性がまだ定義されていない場合は、createStream() メソッドを使用して設定します。ローカル・ディスク・オブジェクトと分散ディスク・オブジェクトでは、管理方法が異なります。ローカルまたは分散のいずれであるかは、オブジェクトの作成時に属性に基づいて決定されます。

注意： 同じキャッシュ・システム内の分散キャッシュ間で StreamAccess オブジェクトを共有する場合は、StreamAccess オブジェクトの作成時に DISTRIBUTE 属性を定義する必要があります。この属性は、オブジェクトの作成後は変更できません。

例 9-19 に、StreamAccess オブジェクトをキャッシュにロードするローダー・オブジェクトを示します。

例 9-19 キャッシュ・ローダーでの StreamAccess オブジェクトの作成

```
import oracle.ias.cache.*;

class YourObjectLoader extends CacheLoader
{
   public Object load(Object handle, Object args) {
     OutputStream = out;
     Attributes attr = new Attributes();
     attr.setFlags(Attributes.DISTRIBUTE);

     try
     {
        out = createStream(handle, attr);
        out.write((byte[])getInfofromsomewhere());
     }
     catch (Exception ex) {
        // translate exception to CacheException, and log exception
        throw exceptionHandler("exception in write", ex)
```

```
        }
        return out;
        }
}
```

プール・オブジェクトの操作

プール・オブジェクトは、Java Object Cache で管理される特別なキャッシュ・オブジェクトです。プール・オブジェクトには、同一オブジェクト・インスタンスのセットが含まれます。プール・オブジェクト自体は共有オブジェクトですが、プール内のオブジェクトは Java Object Cache により管理されるプライベート・オブジェクトです。ユーザーは、プール・アクセス・オブジェクトを使用して、プール内の個々のオブジェクトをチェックアウトしてアクセスし、その後不要になるとオブジェクトをプールに戻します。

この項には、次の項目が含まれます。

- プール・オブジェクトの作成
- プール・オブジェクトの使用方法
- プール・オブジェクトのインスタンス・ファクトリの実装
- プール・オブジェクトのアフィニティ

プール・オブジェクトの作成

プール・オブジェクトを作成するには、`CacheAccess.createPool()` を使用します。`CreatePool()` メソッドは、次の引数を使用します。

- `PoolInstanceFactory`
- `Attributes` オブジェクト
- 2 つの整数引数

整数引数では、最大プール・サイズと最小プール・サイズを指定します。`CreatePool()` に引数としてグループ名を指定すると、プール・オブジェクトがグループに関連付けられます。

`TimeToLive` や `IdleTime` などの属性をプール・オブジェクトに関連付けることができます。これらの属性は、`CacheAccess.createPool()` を使用して属性セットで指定されると、プール・オブジェクト自体に適用できます。またはプール内のオブジェクトに個別に適用することもできます。

`CacheAccess.createPool()` を使用して、整数引数で最小サイズと最大サイズを指定します。最初に最小サイズを指定してください。これにより、プール内に作成されるオブジェクトの最小数を設定します。最小サイズは、最小保証ではなく、リクエストとして解釈されます。プール・オブジェクト内のオブジェクトはリソースが不足した場合にキャッシュから削除されるため、プールのオブジェクト数がリクエストした最小値よりも小さくなる場合があります。最大プール・サイズは、プールで使用可能なオブジェクト数に関して強い制限を設けます。

注意： プール・オブジェクトおよびプール・オブジェクト内のオブジェクトは、常にローカル・オブジェクトとして処理されます。

例 9-20 に、プール・オブジェクトの作成方法を示します。

例 9-20 プール・オブジェクトの作成

```
import oracle.ias.cache.*;

   try
   {
      CacheAccess cacc = CacheAccess.getAccess("Stock-Market");
      Attributes  attr = new Attributes();
      QuoteFactory poolFac = new QuoteFactory();

      // set IdleTime for an object in the pool to three minutes
      attr.setIdleTime(180);
      // create a pool in the "Stock-Market" region with a minimum of
      // 5 and a maximum of 10 object instances in the pool
      cacc.createPool("get Quote", poolFac, attr, 5, 10);
      cacc.close();
   }
   catch(CacheException ex)
   {
           // handle exception
   }
}
```

プール・オブジェクトの使用方法

プール内のオブジェクトにアクセスするには、`PoolAccess` オブジェクトを使用します。static メソッド `PoolAccess.getPool()` は、指定したプールへのハンドルを戻します。`PoolAccess.get()` メソッドは、プール内からオブジェクトのインスタンスを戻します（これによって、オブジェクトはプールからチェックアウトされます)。オブジェクトが不要になったときには、`PoolAccess.returnToPool()` メソッドを使用してプールに戻します。これによって、オブジェクトはプールにチェックインされます。最後に、プール・ハンドルが不要になったときに `PoolAccess.close()` メソッドをコールします。

例 9-21 に、`PoolAccess` オブジェクトを作成し、オブジェクトをプールからチェックアウトした後、プールにオブジェクトをチェックインし、`PoolAccess` オブジェクトをクローズするために必要なコールを示します。

例 9-21 PoolAccess オブジェクトの使用方法

```
PoolAccess pacc = PoolAccess.getPool("Stock-Market", "get Quote");
//get an object from the pool
GetQuote  gq = (GetQuote)pacc.get();
// do something useful with the gq object
// return the object to the pool
pacc.returnToPool(gq);
pacc.close();
```

プール・オブジェクトのインスタンス・ファクトリの実装

Java Object Cache は、アプリケーション定義のファクトリ・オブジェクト `PoolInstanceFactory` を使用して、プール内のオブジェクトをインスタンス化および削除します。`PoolInstanceFactory` は、実装が必要な 2 つのメソッド `createInstance()` および `destroyInstance()` を持つ抽象クラスです。

Java Object Cache は、プール内に蓄積されるオブジェクトのインスタンスを作成する場合に、`createInstance()` をコールします。Java Object Cache は、オブジェクトのインスタンスがプールから削除されるときに `destroyInstance()` をコールします（プール内のオブジェクト・インスタンスが `destroyInstance()` に渡されます)。

プール・オブジェクトのサイズ（プール内のオブジェクト数）は、これらの PoolInstanceFactory() メソッドを使用して管理されます。プール内のサイズおよびオブジェクト数は、必要に応じて、および TimeToLive 属性または IdleTime 属性の値に基づいて、自動的に増減します。

例 9-22 に、PoolInstanceFactory の実装時に必要なコールを示します。

例 9-22 プール・インスタンスのファクトリ・メソッドの実装

```
import oracle.ias.cache.*;
   public class MyPoolFactory implements PoolInstanceFactory
   {
       public Object createInstance()
      {
         MyObject obj = new MyObject();
         obj.init();
         return obj;
       }
       public void destroyInstance(Object obj)
       {
           ((MyObject)obj).cleanup();
       }
   }
```

プール・オブジェクトのアフィニティ

オブジェクト・プールは、シリアルで再利用可能なオブジェクトの集合です。ユーザーはプールからオブジェクトをチェックアウトして機能を実行し、完了後にオブジェクトを元のプールにチェックインします。オブジェクトがチェックアウトされている間は、ユーザーがそのオブジェクト・インスタンスを排他的に使用することになります。オブジェクトのチェックイン後は、ユーザーはそのオブジェクトへのアクセスをすべて放棄します。オブジェクトのチェックアウト中に、ユーザーは現行のタスクを実行できるようにプール・オブジェクトに一時的な変更を適用（状態を追加）できます。このような変更を追加するにはある程度のコストが発生するため、ユーザーができるだけ変更を適用済のプールから同じオブジェクトを取得できるようにするとメリットが得られます。リリース 9.0.2 以降の Java Object Cache では、そのための唯一の方法はオブジェクトをチェックインしないことでしたが、これではプールの目的にそぐわないことになります。ここで説明したプール要件をサポートするために、Java Object Cache のプール管理には次の 2 つの段落で説明する機能が追加されています。

PoolAccess オブジェクトの returnToPool メソッドを使用してプールにチェックインされたオブジェクトは、そのオブジェクトを参照した最後の PoolAccess オブジェクトとの関連を維持します。PoolAccess ハンドルがオブジェクト・インスタンスをリクエストすると、前回と同じオブジェクトが戻されます。この関連は、PoolAccess ハンドルがクローズされるか、PoolAccess.release メソッドがコールされるか、またはオブジェクトが別のユーザーに与えられる時点で終了します。オブジェクトが別のユーザーに与えられる前にコールバックが実行され、ユーザーがオブジェクトとの関連を放棄するかどうかが判断されます。ユーザーが関連を解除しない場合は、新規ユーザーにはそのオブジェクトへのアクセスが与えられません。PoolAffinityFactory インタフェースは、PoolInstanceFactory インタフェースを拡張してコールバック・メソッド affinityRelease を追加します。このメソッドは、関連を解除できる場合は true、それ以外の場合は false を戻します。

プール全体が無効化される場合、affinityRelease メソッドはコールされません。その場合は、PoolInstanceFactory.instanceDestroy メソッドを使用してオブジェクト・インスタンスのクリーン・アップが実行されます。

ローカル・モードでの実行

ローカル・モードで実行中の Java Object Cache は、オブジェクトを共有せず、同じシステムでローカルに実行されているか、あるいはネットワークを介してリモートで実行されている他のキャッシュとは通信しません。ローカル・モードで実行されているときは、システムのシャットダウンやプログラム・エラー発生時のオブジェクトの永続性はサポートされません。

デフォルトでは、Java Object Cache はローカル・モードで実行され、キャッシュ内のすべてのオブジェクトがローカル・オブジェクトとして処理されます。Java Object Cache がローカル・モードで構成されている場合、キャッシュはすべてのオブジェクトの `DISTRIBUTE` 属性を無視します。

分散モードでの実行

分散モードで実行中の Java Object Cache は、オブジェクトを共有でき、同じシステムでローカルに実行されているか、あるいはネットワークを介してリモートで実行されている他のキャッシュと通信できます。オブジェクトの更新および無効化は、通信を行っているキャッシュ間で伝播されます。分散モードでは、システムのシャットダウンやプログラム・エラー発生時のオブジェクトの永続性がサポートされます。

この項には、次の項目が含まれます。

- 分散モード用のプロパティの構成
- 分散オブジェクト、リージョン、サブリージョンおよびグループの使用方法
- キャッシュされたオブジェクトの整合性レベル
- OC4J サーブレットでのキャッシュ・オブジェクトの共有

分散モード用のプロパティの構成

Java Object Cache を分散モードで実行するように構成するには、`javacache.xml` ファイルの `distribute` および `discoveryAddress` 構成プロパティの値を設定します。

distribute 構成プロパティの設定

Java Object Cache を分散モードで起動するには、構成ファイルで `isDistributed` 属性を `true` に設定します。設定方法は、9-22 ページの「Java Object Cache の構成」を参照してください。

discoveryAddress 構成プロパティの設定

分散モードでは、無効化、破棄および置換は、キャッシュのメッセージ・システムを介して伝播されます。メッセージ・システムでは、キャッシュが最初の初期化時にキャッシュ・システムに加わることができるように、既知のホスト名およびポート・アドレスが必要です。`javacache.xml` ファイルの communication セクションで `discoverer` 属性を使用すると、ホスト名とポート・アドレスのリストを指定できます。

デフォルトでは、Java Object Cache は、`discoverer` の値を `:12345` に設定します（これは、`localhost:12345` と同じです）。サイト上の他のソフトウェアと競合しないように、システム管理者に `discoveryAddress` の設定を依頼してください。

Java Object Cache が複数のシステムにまたがる場合は、ノードごとに *`hostname:port`* ペアを 1 つ指定して、複数の discoverer エントリを構成します。この結果、使用可能な特定のシステムまたはプロセスの起動順序に対する依存性が回避されます。9-22 ページの「Java Object Cache の構成」も参照してください。

注意： 同じキャッシュ・システムで同時に動作するすべてのキャッシュが、同じホスト名とポート・アドレスのセットを指定している必要があります。discoverer 属性で設定されるアドレス・リストは、特定のキャッシュ・システムを構成するキャッシュを定義します。アドレス・リストが異なると、キャッシュ・システムは別々のグループに分割され、キャッシュ間で不整合が発生する可能性があります。

分散オブジェクト、リージョン、サブリージョンおよびグループの使用方法

Java Object Cache が分散モードで実行されているとき、個々のリージョン、サブリージョン、グループおよびオブジェクトは、ローカルまたは分散のいずれかです。デフォルトでは、オブジェクト、リージョン、サブリージョンおよびグループはローカルとして定義されます。デフォルトのローカルの値を変更するには、オブジェクト、リージョンまたはグループの定義時に DISTRIBUTE 属性を設定します。

分散キャッシュには、ローカル・オブジェクトと分散オブジェクトの両方が含まれる場合があります。

Java Object Cache のいくつかの属性およびメソッドを使用すると、分散オブジェクトを処理でき、キャッシュ間のオブジェクト・データの整合性レベルを制御できます。9-50 ページの「キャッシュされたオブジェクトの整合性レベル」も参照してください。

分散オブジェクトでの REPLY 属性の使用方法

複数のキャッシュにわたるオブジェクトを更新、無効化または破棄するときは、関係するすべてのサイトで操作がいつ完了したかを認識することが役立つ場合があります。REPLY 属性を設定すると、関係するすべてのキャッシュが、オブジェクトに対してリクエストした操作が完了したときに、送信者に応答を送信します。CacheAccess.waitForResponse() メソッドを使用すると、ユーザーはすべてのリモート操作が完了するまでブロックできます。

分散操作が複数のキャッシュ全体で完了するのを待機するには、CacheAccess.waitForResponse() を使用します。レスポンスを無視するには、レスポンスの収集に使用されるキャッシュ・リソースを解放する CacheAccess.cancelResponse() メソッドを使用します。

CacheAccess.waitForResponse() および CacheAccess.cancelResponse() は、いずれも CacheAccess オブジェクトがアクセスするすべてのオブジェクトに適用されます。この機能により、アプリケーションは、複数のオブジェクトを更新した後で、そのすべての応答を待機できます。

例 9-23 は、オブジェクトを分散として設定する方法、および REPLY 属性が設定されているときの応答の処理方法を示しています。この例では、属性はリージョン全体にも設定できます。また属性は、アプリケーションにあわせて、グループまたは個々のオブジェクトに設定できます。

例 9-23 応答を使用した分散キャッシング

```
import oracle.ias.cache.*;

CacheAccess cacc;
String      obj;
Attributes attr = new Attributes ();
MyLoader    loader = new MyLoader();

// mark the object for distribution and have a reply generated
// by the remote caches when the change is completed

attr.setFlags(Attributes.DISTRIBUTE|Attributes.REPLY);
attr.setLoader(loader);

CacheAccess.defineRegion("testRegion",attr);
cacc = CacheAccess.getAccess("testRegion"); // create region with
```

```
  //distributed attributes

obj = (String)cacc.get("testObject");
cacc.replace("testObject", obj + "new version"); // change will be
  // propagated to other caches

cacc.invalidate("invalidObject"); // invalidation is propagated to other caches

try
{
// wait for up to a second,1000 milliseconds, for both the update
// and the invalidate to complete
    cacc.waitForResponse(1000);

catch (TimeoutException ex)
{
   // tired of waiting so cancel the response
   cacc.cancelResponse();
}
cacc.close();
}
```

SYNCHRONIZE および SYNCHRONIZE_DEFAULT の使用方法

複数のキャッシュにわたるオブジェクトを更新するとき、あるいは複数スレッドで単一のオブジェクトにアクセスするときは、更新操作を調整する場合があります。SYNCHRONIZE 属性を設定すると、同期化された更新が可能になり、アプリケーションは、オブジェクトをロードまたは更新する前にオブジェクトの所有権を取得する必要があります。

SYNCHRONIZE 属性は、リージョン、サブリージョンおよびグループにも適用されます。SYNCHRONIZE 属性がリージョン、サブリージョンまたはグループに適用される場合、そのリージョン、サブリージョンまたはグループ内のオブジェクトをロードまたは置換するには、リージョン、サブリージョンまたはグループの所有権を取得する必要があります。

リージョン、サブリージョンまたはグループに SYNCHRONIZE_DEFAULT を設定すると、そのリージョン、サブリージョンまたはグループ内のすべてのオブジェクトに SYNCHRONIZE 属性が適用されます。リージョン、サブリージョンまたはグループ内の個々のオブジェクトをロードまたは置換するには、そのオブジェクトの所有権を取得する必要があります。

オブジェクトの所有権を取得するには、CacheAccess.getOwnership() を使用します。所有権を取得すると、他の CacheAccess インスタンスはそのオブジェクトのロードまたは置換を許可されません。オブジェクトの読取りおよび無効化は同期による影響を受けません。

所有権を取得し、オブジェクトの変更が完了した後、CacheAccess.releaseOwnership() をコールしてオブジェクトを解放してください。CacheAccess.releaseOwnership() は、リモート・キャッシュでの更新の完了を、指定した時間まで待機します。指定した時間内に更新が完了した場合は所有権が解放されます。指定した時間内に完了しない場合は TimeoutException がスローされます。メソッドがタイムアウトになった場合は、CacheAccess.releaseOwnership() を再度コールしてください。所有権が解放されるように、CacheAccess.releaseOwnership() は必ず正常に戻される必要があります。タイムアウト値が -1 の場合、所有権は他のキャッシュからのレスポンスを待機せずにただちに解放されます。

例 9-24 に、SYNCHRONIZE および SYNCHRONIZE_DEFAULT を使用した分散キャッシングを示します。

例 9-24 SYNCHRONIZE および SYNCHRONIZE_DEFAULT を使用した分散キャッシング

```
import oracle.ias.cache.*;

CacheAccess cacc;
String     obj;
Attributes attr = new Attributes ();
MyLoader   loader = new MyLoader();

// mark the object for distribution and set synchronize attribute
attr.setFlags(Attributes.DISTRIBUTE|Attributes.SYNCHRONIZE);
attr.setLoader(loader);

//create region
CacheAccess.defineRegion("testRegion");
cacc = CacheAccess.getAccess("testRegion");
cacc.defineGroup("syncGroup", attr); //define a distributed synchronized group
cacc.defineObject("syncObject", attr); // define a distributed synchronized object
attr.setFlagsToDefaults()  // reset attribute flags

// define a group where SYNCHRONIZE is the default for all objects in the group
attr.setFlags(Attributes.DISTRIBUTE|Attributes.SYNCHRONIZE_DEFAULT);
cacc.defineGroup("syncGroup2", attr);
try
{
// try to get the ownership for the group don't wait more than 5 seconds
   cacc.getOwnership("syncGroup", 5000);
   obj = (String)cacc.get("testObject", "syncGroup"); // get latest object
   // replace the object with a new version
   cacc.replace("testObject", "syncGroup", obj + "new version");
   obj = (String)cacc.get("testObject2", "syncGroup"); // get a second object
   // replace the object with a new version
   cacc.replace("testObject2", "syncGroup", obj + "new version");
}

catch (TimeoutException ex)
{
   System.out.println("unable to acquire ownership for group");
   cacc.close();
   return;
}
try
{
   cacc.releaseOwnership("syncGroup",5000);
}
catch (TimeoutException ex)
{
   // tired of waiting so just release ownership
   cacc.releaseOwnership("syncGroup", -1));
}
try
{
   cacc.getOwnership("syncObject", 5000); // try to get the ownership for the object
   // don't wait more than 5 seconds
   obj = (String)cacc.get("syncObject");  // get latest object
   cacc.replace("syncObject", obj + "new version"); // replace the object with a new
version
}
catch (TimeoutException ex)
{
```

```
      System.out.println("unable to acquire ownership for object");
      cacc.close();
      return;
   }
   try
   {
      cacc.releaseOwnership("syncObject", 5000);
   }
   catch (TimeoutException ex)
   {
      cacc.releaseOwnership("syncObject", -1)); // tired of waiting so just release
   ownership
   }
   try
   {
      cacc.getOwnership("Object2", "syncGroup2", 5000); // try to get the ownership for
   the object
      // where the ownership is defined as the default for the group don't wait more than
   5 seconds
      obj = (String)cacc.get("Object2", "syncGroup2"); // get latest object
      // replace the object with new version
      cacc.replace("Object2", "syncGroup2", obj + "new version");
   }

   catch (TimeoutException ex)
   {
      System.out.println("unable to acquire ownership for object");
      cacc.close();
      return;
   }
   try
   {
      cacc.releaseOwnership("Object2", 5000);
   }
   catch (TimeoutException ex)
   {
      cacc.releaseOwnership("Object2", -1)); // tired of waiting so just release ownership
   }
      cacc.close();
   }
```

キャッシュされたオブジェクトの整合性レベル

Java Object Cache 内で、各キャッシュはその JVM プロセス内で所有オブジェクトをローカルで管理します。分散モードでは、複数プロセスを使用しているとき、あるいはシステムが複数のサイトで実行されているときは、オブジェクトのコピーが複数のキャッシュに存在する可能性があります。

Java Object Cache では、複数のキャッシュで使用可能なオブジェクトのコピー間で必要な整合性レベルを指定できます。指定する整合性レベルは、アプリケーションおよびキャッシュされるオブジェクトによって決まります。サポートされる整合性レベルは、指定しない場合から、オブジェクトのすべてのコピーの整合性をすべての通信キャッシュにわたって保つように指定する場合まで多数あります。

オブジェクト属性を設定すると、整合性のレベルが指定されます。異なるキャッシュに存在するオブジェクト間の整合性は、次の 4 つのレベルに分類されます。

- ローカル・オブジェクトの使用（整合性要件なし）
- 応答待機なしの変更の伝播
- 変更の伝播および応答の待機
- 複数のキャッシュ間にわたる変更のシリアライズ

ローカル・オブジェクトの使用

分散キャッシュ内のオブジェクト間の整合性が必要でない場合、オブジェクトはローカル・オブジェクトとして定義してください（Attributes.DISTRIBUTE が未設定の場合は、ローカル・オブジェクトであることを示しています)。ローカルは、オブジェクトのデフォルト設定です。ローカル・オブジェクトの場合は、すべての更新および無効化がローカル・キャッシュでのみ参照できます。

応答待機なしの変更の伝播

分散キャッシュ間でオブジェクトの更新を配布するには、DISTRIBUTE 属性を設定してオブジェクトを分散として定義します。分散オブジェクトに対するすべての変更が、システム内の他のキャッシュに配布されます。このレベルの整合性を使用した場合、オブジェクトがキャッシュにロードされたり更新されるタイミングは制御または指定されず、すべてのキャッシュにおける変更の完了に関する通知は行われません。

変更の伝播および応答の待機

分散キャッシュ間でオブジェクトの更新を配布し、次の処理に進む前に変更の完了を待機するには、オブジェクトの DISTRIBUTE 属性および REPLY 属性を設定します。これらの属性を設定すると、すべてのキャッシュで変更が完了したときに通知が行われます。オブジェクトに対して Attributes.REPLY を設定すると、リモート・サイトで変更が完了したときに、変更元のキャッシュに応答が戻されます。これらの応答は非同期に戻されるため、CacheAccess.replace() メソッドおよび CacheAccess.invalidate() メソッドはブロックしません。応答を待機してブロックするには、CacheAccess.waitForResponse() を使用します。

複数のキャッシュ間にわたる変更のシリアライズ

Java Object Cache の最も高いレベルの整合性を使用するには、リージョン、サブリージョン、グループまたはオブジェクトに適切な属性を設定して、オブジェクトが同期化オブジェクトとして動作するようにします。

リージョン、サブリージョンまたはグループに Attributes.SYNCHRONIZE_DEFAULT を設定すると、そのリージョン、サブリージョンまたはグループ内のすべてのオブジェクトに SYNCHRONIZE 属性が設定されます。

オブジェクトに Attributes.SYNCHRONIZE を設定すると、アプリケーションは、オブジェクトをロードまたは変更する前にそのオブジェクトの所有権を取得する必要があります。この属性を設定すると、オブジェクトへの書込みアクセスが効果的にシリアライズされます。オブジェクトの所有権を取得するには、CacheAccess.getOwnership() メソッドを使用します。Attributes.SYNCHRONIZE 属性を設定すると、更新が完了したときに、所有者に通知が送信されます。未処理の更新が完了し、応答が受信されるまでブロックするには、CacheAccess.releaseOwnership() を使用します。このメソッドは、他のキャッシュがオブジェクトを更新またはロードできるようにオブジェクトの所有権を解放します。

注意 : オブジェクトに対して Attributes.SYNCHRONIZE を設定することによる効果は、Java メソッドで synchronized を設定した場合とは異なります。Attributes.SYNCHRONIZE が設定されていると、Java Object Cache は、キャッシュでオブジェクトの作成と更新がシリアライズされるようにしますが、Java プログラマは、オブジェクトの参照を取得し、そのオブジェクトを変更できます。

Attributes.SYNCHRONIZE でこのレベルの整合性を使用すると、オブジェクトを外部ソースからロードする前に、CacheLoader.load() メソッドで CacheLoader.netSearch() をコールします。ロード・メソッドで CacheLoader.netSearch() をコールすると、Java Object Cache は、他のすべてのキャッシュを検索してオブジェクトのコピーを探します。このプロセスにより、異なるバージョンのオブジェクトが外部ソースからキャッシュにロードされることがなくなります。SYNCHRONIZE 属性を REPLY 属性および無効化メソッドとともに正しく使用することで、キャッシュ・システム全体でオブジェクトの整合性が保証されます。

OC4J サーブレットでのキャッシュ・オブジェクトの共有

Java Object Cache の配布機能を利用したり、キャッシュ・オブジェクトをサーブレット間で共有するには、アプリケーションのデプロイを多少変更する必要があります。サーブレット間で共有される、または JVM 間に配布されるユーザー定義オブジェクトは、システム・クラス・ローダーによってロードされる必要があります。デフォルトでは、サーブレットによってロードされたオブジェクトは、コンテキスト・クラス・ローダーによってロードされます。これらのオブジェクトは、ロードしたコンテキスト内のサーブレットでのみ参照可能です。オブジェクト定義は、他のサーブレットまたは別の JVM のキャッシュでは使用できません。オブジェクトがシステム・クラス・ローダーによってロードされた場合、そのオブジェクト定義は、他のサーブレットや他の JVM のキャッシュで使用できるようになります。

> **注意：** 廃止 : Apache JServ サーブレット環境（JServ）では、前述の機能は、JServ プロセスの開始時に使用可能な `classpath` 定義にキャッシュ・オブジェクトを含めることで実現していました。

OC4J では、システムの `classpath` は、`oc4j.jar` ファイルと関連 JAR ファイル（`cache.jar` など）のマニフェストから導出されます。環境内の `classpath` は無視されます。キャッシュ・オブジェクトを OC4J の `classpath` に組み込むには、クラス・ファイルを *`ORACLE_HOME`*`/javacache/sharedobjects/classes` にコピーするか、JAR ファイル *`ORACLE_HOME`*`/javacache/cachedobjects/share.jar` に追加します。`classes` ディレクトリと `share.jar` ファイルは両方とも、`cache.jar` のマニフェストに組み込まれています。

ユーザー定義のクラス・ローダーの使用

オブジェクトは、ユーザー定義のクラス・ローダーを必要とするキャッシュに配置できます。Cache Service では、2 つのメソッド `Attributes.setClassLoader()` および `Attributes.getClassLoader()` を提供して、ユーザー定義のクラス・ローダーをサポートします。ユーザー定義のクラス・ローダーを使用するようにリージョンまたはグループを設定すると、リージョンまたはグループ下のすべてのオブジェクトは、そのクラス・ローダーを使用して（属性を継承して）ロードされます。この属性は、リージョンまたはグループに属さないオブジェクトには適用されず、設定されている場合は無視されます。

次の例は、ユーザー定義のクラス・ローダーの設定方法を示しています。オブジェクト A は、MyClassLoader を使用してロードされます。

```
import oracle.ias.cache.*;
import java.lang.ClassLoader;
Classloader loader = this.getClass().getClassLoader();
CacheAttributes cAttr = new CacheAttributes();
Attributes attr = new Attributes();
CacheAccess cacc;
Cache.init(cAttr);
attr.setClassLoader(loader);
CacheAccess.defineRegion("region A", attr);
cacc = CacheAccess.getAccess("region A");
cacc.get("object A")
```

ユーザー定義のクラス・ローダーを使用できるのは、オブジェクト・コンテンツに対してのみで、オブジェクト名には使用できないことに注意してください。次の行を例に説明します。

```
cacc.put(name, object);
```

`object` はユーザー定義のクラス・ローダーを必要とすることができますが、`name` はできません。

分散キャッシュの HTTP およびセキュリティ

この項では、分散キャッシュの HTTP およびセキュリティについて説明します。

HTTP

デフォルトでは、Cache Service は、TCP の最上部で構築される独自のプロトコルを使用してキャッシュ間の通信を行います。独自のプロトコルの他に、Cache Service では、キャッシュ間の通信に HTTP を使用することもサポートします。互換性の理由により独自のプロトコルは保持されますが、より新しい機能のいくつかは、HTTP のためだけに実装されます。特に分散キャッシュ・システムでは、リモート・キャッシュからディスク・オブジェクトまたはストリーム・オブジェクトを取得する際に、HTTP モードが必要です。たとえば、ディスク・オブジェクトおよびストリーム・オブジェクトの処理に HTTP モードが必要とされる操作には `CacheAccess.getAllCached()`、`CacheLoader.getFromRemote()` および `CacheLoader.netSearch()` の 3 つがあります。HTTP モードを有効化するには、`CacheAttributes.transport` を設定する必要があります。

HTTP を使用するには、分散システム内のすべてのキャッシュで有効化する必要があります。すでに独自のプロトコルを使用している機能でも、有効化された HTTP で動作することができます。

次の例に、HTTP モードを有効化する 2 つの方法を示します。

例 1: `cache_attributes.xml` での HTTP の有効化

```
--------------------
<?xml version="1.0" encoding="UTF-8"?>
<cache-configuration xmlns="http://www.oracle.com/oracle/ias/cache/configuration"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <communication>
      <isDistributed>true</isDistributed>
      <transport>HTTP</transport>
   </communication>
</cache-configuration>

import oracle.ias.cache.*;
Cache.open("cache_attributes.xml");
```

例 2: コードでの HTTP の有効化

```
import oracle.ias.cache.*;
CacheAttributes cAttr = new CacheAttributes();
cAttr.distribute = true;
cAttr.transport = CacheAttributes.HTTP;
Cache.init(cAttr);
```

SSL

キャッシュ間の通信を保護するため、Cache Service では SSL プロトコルをサポートします。JDK keytool プログラムを使用し、証明書を生成してキーストアを設定できます。詳細は、Sun 社の J2SE 1.4.2 Key and Certificate Management Tool の Web ページを参照してください。OC4J に使用されるのと同じ鍵のペアおよび証明書を Cache Service に使用することができます。

SSL を使用するには、分散システム内のすべてのキャッシュで有効化する必要があります。

キーストアの設定後、次のコマンドを使用してキーストアの場所をキャッシュに指定する必要があります。

```
java -jar $ORACLE_HOME/javacache/lib/cache.jar sslconfig <cache_attributes.xml>
<keystore_file> <password>
```

指定する内容は次のとおりです。

- `$ORACLE_HOME` は、Oracle *i*AS インスタンスのホーム・ディレクトリです。
- `cache_attributes.xml` は、キャッシュ構成ファイルです。
- `keystore_file` は、keytool によって生成されるキーストア・ファイルへのフルパスです。
- `password` は、鍵のペアの生成に keytool で使用されるパスワードです。

これにより、キャッシュで使用する SSL 構成ファイルが生成されます。ファイルの名前は、cache_attributes.xml に指定された名前です。また、CacheAttributes.isSSLEnabled を true に設定する必要があります。

次の例に、SSL を有効化する 2 つの方法を示します。

例 1: `cache_attributes.xml` での SSL の有効化

```
<?xml version="1.0" encoding="UTF-8"?>
<cache-configuration
 xmlns="http://www.oracle.com/oracle/ias/cache/configuration"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <communication>
      <isDistributed>true</isDistributed>
      <useSSL>true</useSSL>
      <keyStore>.keyStore</keyStore>
      <sslConfigFile>.sslConfig</sslConfigFile>
   </communication>
</cache-configuration>

import oracle.ias.cache.*;
Cache.open("cache_attributes.xml");
```

例 2: コードでの SSL の有効化

```
import oracle.ias.cache.*;
CacheAttributes cAttr = new CacheAttributes();
cAttr.distribute = true;
cAttr.isSSLEnabled = true;
cAttr.keyStoreLocation = ".keyStore";
cAttr.sslConfigFilePath = ".sslConfig";
Cache.init(cAttr);
```

2 つのキャッシュは、相互に通信するために同じ鍵のセットを使用する必要があります。システム内のキャッシュが複数のマシン上にある場合は、キーストア・ファイルをすべてのマシンにコピーし、システム内の各キャッシュ構成ファイルに対して `java -jar` ... コマンドを実行する必要があります。

ファイアウォール

ファイアウォールを越えて分散キャッシュ・システムを動作させるための現在の対応策は、一連のアウトバウンドの TCP ポートをファイアウォールで有効化し、それらを `cache_attributes.xml` に定義することです。

たとえば、ポートの範囲が `7100` ～ `7199` の場合、`cache_attributes.xml` は次のようになります。

```
cache_attributes.xml
--------------------
<?xml version = '1.0' encoding = 'UTF-8'?>
<cache-configuration xmlns="http://www.oracle.com/oracle/ias/cache/configuration"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <communication>
      <isDistributed>true</isDistributed>
      <useSSL>false</useSSL>
      <sslConfigFile>.sslConfig</sslConfigFile>
```

```
      <port lower="7100" upper="7199"/>
      <discoverer discovery-port="7100" original="true" xmlns=""/>
   </communication>
</cache-configuration>
```

`discovery-port` が指定した範囲内にあることを確認してください。

着信接続要求の制限

複数のアドレスを指定して構成されているシステムで、複数のネットワーク・サブネット（プライベートおよびパブリックなど）をサポートするために、構成要素 `localAddress` を `cache_attributes.xml` に指定し、着信接続要求を、指定したローカル・アドレスに制限できます。デフォルトでは、分散キャッシュ・システムはリスナー・ソケットをオペレーティング・システムから戻されるプライマリ・ホスト・アドレスにバインドします。ただし、`localAddress` が指定されている場合、キャッシュはリスナー・ソケットを指定されたアドレスにバインドします。`localAddress` に指定された値は、完全修飾されたホスト名または IP アドレスであることが必要です。次に例を示します。

```
cache_attributes.xml
--------------------
<?xml version = '1.0' encoding = 'UTF-8'?>
<cache-configuration xmlns="http://www.oracle.com/oracle/ias/cache/configuration"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <communication>
      <isDistributed>true</isDistributed>
      <localAddress>123.456.78.90</localAddress>
      <discoverer discovery-port="7100" original="true" xmlns=""/>
   </communication>
</cache-configuration>
```

または

```
cache_attributes.xml
--------------------
<?xml version = '1.0' encoding = 'UTF-8'?>
<cache-configuration xmlns="http://www.oracle.com/oracle/ias/cache/configuration"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <communication>
      <isDistributed>true</isDistributed>
      <localAddress>computer.oracle.com</localAddress>
      <discoverer discovery-port="7100" original="true" xmlns=""/>
   </communication>
</cache-configuration>
```

後者の例では、仮想ホスト名の下の IP が変わっても、JOC は影響を受けません。

監視およびデバッグ

Cache.listCacheObjects() および Cache.dump() の他に、Cache Service には、キャッシュの現在ステータスと、リージョン、グループおよびキャッシュ内の個々のオブジェクトの現在ステータスを示すための追加メソッドが用意されています。これらのメソッドは、クラス CacheAccess、Cache および AggregateStatus にあります。

Cache の各メソッドは、キャッシュ独自のステータスを示します。getActiveHostInfo は、キャッシュ・システム内のすべてのアクティブなキャッシュに対して、CacheHostInfo オブジェクトの配列を戻します。getCacheSize は、キャッシュでメモリー・オブジェクトによって使用される総領域（バイト数）を見積ります。getDistributedDiskCacheSize および getLocalDiskCacheSize はそれぞれ、分散ディスク・キャッシュおよびローカル・ディスク・キャッシュでオブジェクトによって使用される総領域（バイト数）を見積ります。getObjectCount は、キャッシュ内の現在のオブジェクトの総数を戻します。

CacheAccess の各メソッドは、リージョン、グループおよび個々のオブジェクトのステータスを示します。listNames は、リージョン下のすべてのオブジェクトに名前を付けます。listObjects は、リージョン下のすべてのオブジェクトに名前を付け、そのオブジェクトへのアクセスも提供します。listRegions は、リージョン下のすべてのサブリージョンに名前を付けます。これら 3 つのメソッドは再帰的ではありません。たとえば、listRegions は、リージョンのサブリージョン下のサブリージョンをリストしません。

CacheAccess.getStatus() は、ObjectStatus オブジェクト形式のリージョン下の個々の名前付きオブジェクトまたはグループについて、より詳細なステータス情報を示します。この情報には、キャッシュ済オブジェクトのアクセス数、作成時間、ディスク上のサイズ（ディスクに格納されている場合）、最後にアクセスされた時間、ロード時間（分）、優先順位（オブジェクトの作成者により設定）およびキャッシュ内のサイズ（バイト）が含まれます。オブジェクトまたはグループの名前が指定されていない場合、getStatus はリージョンのステータスを戻します。

一方、CacheAccess.getAggregateStatus() は、AggregateStatus オブジェクト形式の名前付きグループまたはリージョン（サブリージョン）について、全体的な統計情報を戻します。AggregateStatus は、リージョンまたはグループ下のオブジェクトの属性について、下限、平均および上限の値を示します。これらの属性には、アクセス数、作成時間、最後にアクセスされた時間、ロード時間、優先順位およびキャッシュ内のサイズが含まれます。また、AggregateStatus オブジェクトには、リージョンまたはグループのオブジェクト合計数も含まれています。AggregateStatus クラスの反映メソッドを使用すると、これらの数値すべてに個別にアクセスできます。

Cashe Service では、クリーン間隔ごとに、getAggregateStatus によって示された情報が自動的に集計されます。最新の情報を取得するには、getAggregateStatus をコールする前に Cache.updateStats() をコールする必要があります。

次の例は、getAggregateStatus の使用方法を示しています。

```
import oracle.ias.cache.*;
import java.util.Date;
import java.io.*;

CacheAccess cacc;

// create objects, load objects, etc.
...

AggregateStatus aggStats;
long            avgCreateTime;
Date            avg;

Cache.updateStats();
aggStats = cacc.getAggregateStatus();
avgCreateTime = aggStats.getCreateTime(AggregateStatus.AVG);
avg = new Date(avgCreateTime);

System.out.println("average creation time: " + avg);
```

CacheWatchUtil

デフォルトでは、Cache Service には CacheWatchUtil キャッシュ監視ユーティリティが用意されています。このユーティリティにより、システム内の現在キャッシュの表示、キャッシュ済オブジェクトのリストの表示、キャッシュの属性の表示、キャッシュ・ログ出力の重大度のリセット、キャッシュの内容のログへのダンプなどが可能になります。

CacheWatchUtil を起動するには、キャッシュの稼働時に、次のコマンドのいずれかを入力します。

```
java oracle.ias.cache.CacheWatchUtil [-config=cache_config.xml] [-help]
```

または

```
java –jar $ORACLE_HOME/javacache/lib/cache.jar watch [-config=cache_config.xml] [-help]
```

-config= および -help は、オプションのパラメータです。cache_config.xml は、キャッシュ構成ファイルです。

help で、キャッシュ・ウォッチャで起動できるコマンドのリストが表示されます。次のようなコマンドがあります。

- set severity=<level> [CacheId] は、特定のキャッシュに対してログ出力の重大度レベルを設定します。次のレベルがあります。
  - -1: オフ
  - 0: 致命的
  - 3: エラー
  - 4: デフォルト
  - 6: 警告
  - 7: トレース
  - 10: 情報
  - 15: デバッグ
- set timeout=<value> は、キャッシュ・システムのグループ通信のタイムアウトに値を設定します。
- dump [CacheId] は、特定のキャッシュの内容をログ・ファイルにダンプします。
- invalidate は、キャッシュ・システム内のすべてのオブジェクトを無効化します。
- destroy は、キャッシュ・システム内のすべてのオブジェクト（メモリー、ストリーム、ディスクなど）を破棄します。

get config [CacheId] と入力すると、特定のキャッシュについて、キャッシュ構成情報が戻されます。次の例に示すように、検証のためにリモート・キャッシュ構成を取得できます。

```
cache> get config 3
ache 3 at localhost:53977
distribute = true
version = 9.0.4
max objects = 200
max cache size = 48
diskSize = 32
diskPath = <disk_path>
clean interval = 3
LogFileName = <log_file_name>
Logger = MyCacheLogger
Log severity = 3
cache address list = [127.0.0.1:22222, pos=-1, uid=0, orig, name=, pri=0
]
```

list caches または lc と入力すると、システム内のすべてのアクティブなキャッシュが出力されます。次の例に示すように、キャッシュ・ウォッチャもそのリストの一部を占めます。

UID 列には、各キャッシュの ID が表示されます。キャッシュ・ウォッチャは、構成されていてもアクティブでないキャッシュを検出できません。

```
cache> lc
Current coordinator: [127.0.0.1:53957, pos=0, uid=0, tag=27979955, pri=0]
#       UID     CacheAddress
-       ---     --------------------
1       0       localhost:53957
2       1       localhost:53965
3       2       localhost:53974
4       3       localhost:53977
5       4       localhost:53980
6       5       localhost:53997 <-- this cache watcher
```

次のように入力します。

```
"list objects [CacheId] [region=<region>] [sort=<0...7>]"
```

または

```
"lo [CacheId] [region=<region>] [sort=<0...7>]"
```

これにより、指定されたキャッシュ内、指定されたリージョン下のすべてのオブジェクトが、ソート・オプションによって指定された順序で出力されます。次のソート・オプションがあります。

- 0: リージョン名の順
- 1: オブジェクト名の順
- 2: グループ名の順
- 3: オブジェクト・タイプの順
- 4: 有効なステータスの順
- 5: 参照数の順
- 6: アクセス数の順
- 7: 有効期限の順

オプションを一切指定しない場合、lo は、すべてのキャッシュ内のすべてのオブジェクトをソートせずに出力します。次の例は、キャッシュ 3、A リージョンについてオブジェクト名順に出力する lo を示しています。例の各列は、読みやすいように調整されています。

```
cache> lo 3 region=A-Region sort=1
Cache 3 at localhost:53977
 REGION     OBJNAME    GROUP      TYPE   REFCNT   ACCCNT   EXPIRE      VALID   LOCK
--------   ---------  -------    ------ -------- -------- --------    ------- ------
[A-Region] [A-Group]  [A-Region] Group  0        1        None        true    null
[A-Region] [A-Region] [null]     Region 0        4        295 Seconds true    null
[A-Region] [B-Group]  [A-Region] Group  0        3        None        true    null
[A-Region] [bar]      [B-Group]  Loader 0        1        None        true    null
```

最後に、groupdump と入力すると、すべてのキャッシュについて、すべてのグループ通信情報がログ・ファイルにダンプされます。このコマンドを使用したり、その出力が役に立つ可能性は少ないですが、グループ通信エラーの場合には、技術サポートで問題を診断するために情報の提供を要求されることがあります。

キャッシュ構成用の XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://www.oracle.com/oracle/ias/cache/configuration"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns="http://www.oracle.com/oracle/ias/cache/configuration"
elementFormDefault="qualified" attributeFormDefault="unqualified">
   <xs:element name="cache-configuration" type="CacheConfigurationType">
      <xs:annotation>
         <xs:documentation>Oracle JavaCache implementation</xs:documentation>
      </xs:annotation>
   </xs:element>
   <xs:complexType name="CacheConfigurationType">
      <xs:sequence>
         <xs:element name="logging" type="loggingType" minOccurs="0"/>
         <xs:element name="communication" type="communicationType" minOccurs="0"/>
         <xs:element name="persistence" type="persistenceType" minOccurs="0"/>
         <xs:element name="preload-file" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
         <xs:element name="max-objects" type="xs:positiveInteger" default="1000" minOccurs="0"/>
         <xs:element name="max-size" type="xs:positiveInteger" default="1000" minOccurs="0"/>
         <xs:element name="clean-interval" type="xs:positiveInteger" default="60" minOccurs="0"/>
         <xs:element name="ping-interval" type="xs:positiveInteger" default="60" minOccurs="0"/>
         <xs:element name="cacheName" type="xs:string" minOccurs="0"/>
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="loggingType">
      <xs:sequence>
         <xs:element name="location" type="xs:string" minOccurs="0"/>
         <xs:element name="level" type="loglevelType" minOccurs="0"/>
         <xs:element name="logger" type="xs:string" minOccurs="0"/>
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="communicationType">
      <xs:sequence>
         <xs:element name="isDistributed" type="xs:boolean" default="false" minOccurs="0"/>
         <xs:element name="transport" type="transportType" minOccurs="0"/>
         <xs:element name="useSSL" type="xs:boolean" minOccurs="0"/>
         <xs:element name="sslConfigFile" type="xs:string" minOccurs="0"/>
         <xs:element name="keyStore" type="xs:string" minOccurs="0"/>
         <xs:element name="port" minOccurs="0">
            <xs:complexType>
               <xs:attribute name="lower" type="xs:nonNegativeInteger" use="optional" default="0"/>
               <xs:attribute name="upper" type="xs:nonNegativeInteger" use="optional" default="0"/>
            </xs:complexType>
         </xs:element>
         <xs:element name="localAddress" type="xs:string" minOccurs="0"/>
         <xs:element name="discoverer" minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
               <xs:complexContent>
                  <xs:extension base="discovererType">
                     <xs:attribute name="order" type="xs:nonNegativeInteger"/>
                     <xs:attribute name="original" type="xs:boolean"/>
                  </xs:extension>
               </xs:complexContent>
            </xs:complexType>
         </xs:element>
         <xs:element name="discovererElection" type="electionType" minOccurs="0"/>
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="discovererType">
      <xs:attribute name="ip" type="xs:string"/>
      <xs:attribute name="discovery-port" type="xs:positiveInteger" use="required"/>
   </xs:complexType>
```

```
  <xs:complexType name="persistenceType">
    <xs:sequence>
      <xs:element name="location" type="xs:string"/>
      <xs:element name="disksize" type="xs:positiveInteger" default="30" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name="loglevelType">
    <xs:restriction base="xs:token">
      <xs:enumeration value="OFF"/>
      <xs:enumeration value="FATAL"/>
      <xs:enumeration value="ERROR"/>
      <xs:enumeration value="DEFAULT"/>
      <xs:enumeration value="WARNING"/>
      <xs:enumeration value="TRACE"/>
      <xs:enumeration value="INFO"/>
      <xs:enumeration value="DEBUG"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="transportType">
    <xs:restriction base="xs:token">
      <xs:enumeration value="TCP"/>
      <xs:enumeration value="HTTP"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="electionType">
    <xs:sequence>
      <xs:element name="useMulticast" type="xs:boolean" minOccurs="0"/>
      <xs:element name="updateInterval" type="xs:positiveInteger" minOccurs="0"/>
      <xs:element name="resolutionInterval" type="xs:positiveInteger" minOccurs="0"/>
      <xs:element name="multicastAddress" minOccurs="0">
        <xs:complexType>
          <xs:attribute name="ip" type="xs:string" use="optional"/>
          <xs:attribute name="port" type="xs:string" use="optional"/>
          <xs:attribute name="TTL" type="xs:nonNegativeInteger" use="optional"/>
        </xs:complexType>
      </xs:element>
      <xs:element name="usePriorityOrder" type="xs:boolean" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

属性の宣言用の XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://www.oracle.com/oracle/ias/cache/configuration/declarative"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns="http://www.oracle.com/oracle/ias/cache/configuration/declarative"
 elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:complexType name="regionType">
    <xs:sequence>
      <xs:element name="attributes" type="attributesType" minOccurs="0"/>
      <xs:element name="region" type="regionType" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="group" type="groupType" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="cached-object" type="cached-objectType" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:complexType name="attributesType">
    <xs:sequence>
      <xs:element name="time-to-live" type="xs:positiveInteger" minOccurs="0"/>
      <xs:element name="default-ttl" type="xs:positiveInteger" minOccurs="0"/>
      <xs:element name="idle-time" type="xs:positiveInteger" minOccurs="0"/>
```

```
        <xs:element name="version" type="xs:string" minOccurs="0"/>
        <xs:element name="max-count" type="xs:positiveInteger" minOccurs="0"/>
        <xs:element name="priority" type="xs:positiveInteger" minOccurs="0"/>
        <xs:element name="size" type="xs:positiveInteger" minOccurs="0"/>
        <xs:element name="flag" minOccurs="0" maxOccurs="unbounded">
           <xs:simpleType>
              <xs:restriction base="flagType">
                 <xs:enumeration value="distribute"/>
                 <xs:enumeration value="reply"/>
                 <xs:enumeration value="synchronize"/>
                 <xs:enumeration value="spool"/>
                 <xs:enumeration value="group_ttl_destroy"/>
                 <xs:enumeration value="original"/>
                 <xs:enumeration value="synchronize-default"/>
                 <xs:enumeration value="allownull"/>
                 <xs:enumeration value="measure"/>
              </xs:restriction>
           </xs:simpleType>
        </xs:element>
        <xs:element name="event-listener" type="event-listenerType" minOccurs="0"/>
        <xs:element name="cache-loader" type="userDefinedObjectType" minOccurs="0"/>
        <xs:element name="capacity-policy" type="userDefinedObjectType" minOccurs="0"/>
        <xs:element name="user-defined" minOccurs="0" maxOccurs="unbounded">
           <xs:complexType>
              <xs:sequence>
                 <xs:element name="key" type="xs:string"/>
                 <xs:element name="value" type="xs:string"/>
              </xs:sequence>
           </xs:complexType>
        </xs:element>
     </xs:sequence>
  </xs:complexType>
  <xs:simpleType name="flagType">
     <xs:list itemType="xs:token"/>
  </xs:simpleType>
  <xs:complexType name="userDefinedObjectType">
     <xs:sequence>
        <xs:element name="classname" type="xs:string"/>
        <xs:element name="parameter" type="propertyType" minOccurs="0" maxOccurs="unbounded"/>
     </xs:sequence>
  </xs:complexType>
  <xs:complexType name="propertyType">
     <xs:simpleContent>
        <xs:extension base="xs:string">
           <xs:attribute name="name" type="xs:string" use="required"/>
        </xs:extension>
     </xs:simpleContent>
  </xs:complexType>
  <xs:complexType name="event-listenerType">
     <xs:sequence>
        <xs:element name="classname" type="xs:string"/>
     </xs:sequence>
     <xs:attribute name="handle-event" type="handle-eventType" use="required"/>
     <xs:attribute name="default" type="xs:boolean"/>
  </xs:complexType>
  <xs:simpleType name="handle-eventType">
     <xs:restriction>
        <xs:simpleType>
           <xs:list itemType="xs:token"/>
        </xs:simpleType>
        <xs:enumeration value="object-invalidated"/>
        <xs:enumeration value="object-updated"/>
     </xs:restriction>
```

```
   </xs:simpleType>
   <xs:complexType name="groupType">
      <xs:sequence>
         <xs:element name="attributes" type="attributesType" minOccurs="0"/>
         <xs:element name="group" type="groupType" minOccurs="0" maxOccurs="unbounded"/>
         <xs:element name="cached-object" type="cached-objectType" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
   </xs:complexType>
   <xs:complexType name="cached-objectType">
      <xs:sequence>
         <xs:element name="attributes" type="attributesType" minOccurs="0"/>
         <xs:element name="name" type="nameType" minOccurs="0"/>
         <xs:element name="object" type="userDefinedObjectType" minOccurs="0"/>
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="nameType">
      <xs:choice>
         <xs:element name="string-name" type="xs:string"/>
         <xs:element name="object-name" type="userDefinedObjectType"/>
      </xs:choice>
   </xs:complexType>
   <xs:element name="cache">
      <xs:complexType>
         <xs:sequence maxOccurs="unbounded">
            <xs:element name="region" type="regionType"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

索引

数字

1pc
　「1 フェーズ・コミット」を参照
1 フェーズ・コミット
　構成，7-3
2pc
　「2 フェーズ・コミット」を参照
2 フェーズ・コミット
　OracleTwoPhaseCommitDriver，7-10
　エンジンの制限事項，7-11
　概要，7-7
　定義，7-2
　データ・ソース，4-20

A

AbstractPrincipalMapping
　拡張，8-17
admin.jar
　リソース・アダプタ , アンデプロイ，8-8
　リソース・アダプタ , デプロイ，8-8
admin.jar ツール，6-4，6-5
ALLOWNULL Java Object Cache の属性，9-13
ApplicationClientInitialContextFactory，2-5
application-client.jar
　JNDI，2-3，2-5
application-client.xml，6-14
　JNDI，2-5
<application-server> 要素，8-9
application.xml，7-9
　data-sources.xml の指定，4-8
　<data-sources> タグ，4-8
　位置，4-8
AQ，3-24
Attributes.setCacheEventListener() メソッド，9-36

B

Bean 管理のトランザクション
　MDB, JMS クライアント，7-14
BMT
　リカバリ，7-12
browse
　JMS ユーティリティ，3-9

C

CacheAccess
　createPool() メソッド，9-43
CacheAccess.getOwnership() メソッド，9-48
CacheAccess.releaseOwnership() メソッド，9-48
CacheAccess.save() メソッド，9-40
CacheEventListener
　Java Object Cache の属性，9-14
CacheEventListener インタフェース，9-36
CacheLoader.createStream() メソッド，9-42
CapacityPolicy
　Java Object Cache の属性，9-13
check
　JMS ユーティリティ，3-9
class
　<data-source> 属性，4-9
clean-available-connections-threshold
　<data-source> 属性，4-10
clean-interval 構成用 XML 要素，9-24
CMP
　再試行回数，7-12
　接続のリカバリ，7-12
CMT
　再試行回数，7-12
　リカバリ，7-12
com.evermind.server パッケージ
　ApplicationClientInitialContextFactory，5-10，6-14
　ApplicationInitialContextFactory，5-10
　JNDI，2-5
　RMIInitialContextFactory，5-10
<commit-class> 要素，7-10
Common Secure Interoperability Version 2
　「CSIv2」を参照
com.oracle.iiop.server パッケージ
　IIOPInitialContextFactory，6-14
connection-driver
　<data-source> 属性，4-9
ConnectionFactory
　JMS，3-5
connection-factory 要素，3-18
connection-retry-interval
　<data-source> 属性，4-10
<container-transaction> 要素，7-6
Context.bind API コール，2-2
contextFactory
　ApplicationClientInitialContextFactory，6-14
　IIOPInitialContextFactory，6-14

contextFactory プロパティ，6-14
context.SECURITY_CREDENTIAL
JNDI 関連の環境プロパティ，2-6
context.SECURITY_PRINCIPAL
JNDI 関連の環境プロパティ，2-6
copy
JMS ユーティリティ，3-9
CORBA Object Service Naming
「CosNaming」を参照
CORBA Transaction Service
「OTS」を参照
corbaname の URL，6-11
CosNaming，6-2，6-11
createDiskObject() メソッド，9-19，9-40
createInstance() メソッド，9-44
CreatePool() メソッド，9-43
createStream() メソッド，9-19
CSIv2，6-2

D

Data Guard，4-26
database-schema，4-10，4-15
DataDirect JDBC ドライバ
インストール，4-24
DataDirect ドライバ，4-24
<data-source>
属性，4-9
data-sources.xml，4-7，7-10
EAR ファイル，4-8
JTA での使用，7-3
位置，4-8
位置の指定，4-8
事前にインストールされる定義，4-12
説明，4-8
DataSource オブジェクト
取得，7-3
タイプ，4-2
ルックアップ，4-18，7-4
<data-source> 属性
class，4-9
clean-available-connections-threshold，4-10
connection-driver，4-9
connection-retry-interval，4-10
ejb-location，4-9
inactivity-timeout，4-10
location，4-9
max-connect-attempts，4-10
max-connections，4-10
min-connections，4-10
name，4-9
password，4-9
rac-enabled，4-10
schema，4-10，4-15
stmt-cache-size，4-10
URL，4-9
username，4-9
wait-timeout，4-10
xa-location，4-9
<data-source> タグ，4-9
DBMS_AQADM.CREATE_QUEUE，3-26
DBMS_AQADM パッケージ，3-25
DbUtil
oracleFatalError メソッド，7-12
dcmctl
リソース・アダプタ , アンデプロイ，8-7
リソース・アダプタ , デプロイ，8-7
dedicated.rmicontext
JNDI 関連の環境プロパティ，2-6
DefaultTimeToLive
Java Object Cache の属性，9-13
default-web-site.xml，5-13
defineGroup() メソッド，9-16，9-17
defineObject() メソッド，9-17
defineRegion() メソッド，9-16
dequeue-retry-count，7-13
dequeue-retry-interval，7-13
destinations
JMS ユーティリティ，3-9
destroy() メソッド，9-20
destroyInstance() メソッド，9-44
disallowed-field，4-15
discoveryAddress プロパティ，9-46
DISTRIBUTE
Java Object Cache の属性，9-11，9-46
drain
JMS ユーティリティ，3-9
DTC，4-20
durables
JMS ユーティリティ，3-9

E

EJB
相互運用可能化，6-4，6-8
相互運用性，6-1
ejb-jar.xml，8-5
<message-driven-deployment> 要素，7-13
ejb-location
<data-source> 属性，4-9
EJB の相互運用性
概要，6-2
Enterprise Manager
データ・ソースの定義，4-11
<entity-deployment> 要素，6-13
exceptionHandler() メソッド，9-19

G

getConnection メソッド，4-18，7-4
getID() メソッド，9-36
getName() メソッド，9-19
getOwnership() メソッド，9-48
getOwnsership() メソッド，9-51
getParent() メソッド，9-17
getRegion() メソッド，9-19
getSource() メソッド，9-36
global-web-application.xml，5-12
GROUP_TTL_DESTROY
Java Object Cache の属性，9-11
GROUP_TTL_DESTROY 属性，9-20

H

handleEvent() メソッド，9-36
help

JMS ユーティリティ，3-9
http.tunnel.path
JNDI 関連の環境プロパティ，2-6

I

IdleTime
Java Object Cache の属性，9-14
IIOP，1-2，6-2
iiopClientJar スイッチ，6-4，6-5
IIOPInitialContextFactory，2-11
inactivity-timeout
<data-source> 属性，4-10
INITIAL_CONTEXT_FACTORY
InitialContext のプロパティ，2-4
InitialContext
JNDI での構成，2-4
コンストラクタ，2-4
InitialContext オブジェクト，2-2
InitialContext のプロパティ
INITIAL_CONTEXT_FACTORY，2-4
PROVIDER_URL，2-4
SECURITY_CREDENTIAL，2-4
SECURITY_PRINCIPAL，2-4
Internet Inter-ORB Protocol
「IIOP」を参照
invalidate() メソッド，9-20
<ior-security-config> 要素，6-13

J

J2EE Connector，8-1
デプロイメント・ディスクリプタ，8-4
リソース・アダプタ，8-2
J2EE Connector Architecture
概要，1-3
デプロイメント・ディレクトリの位置，8-9
ファイルの位置，8-10
J2EE アプリケーション・クライアント
JNDI 初期コンテキスト，2-5
J2EE アプリケーション・コンポーネント
JNDI 初期コンテキスト，2-9
JAAS
交換可能認証クラス，8-18
Java Message Service，「JMS」を参照，3-1
Java Naming and Directory Interface
「JNDI」を参照
Java Object Cache，9-1，9-2
distribute プロパティ，9-46
StreamAccess オブジェクト，9-7
オブジェクト・タイプ，9-5，9-6
オブジェクトの識別，9-6
オブジェクトの定義，9-17
オブジェクトのネーミング，9-6
オブジェクトの破棄，9-20
オブジェクトの無効化，9-20
概要，1-3
機能，9-6
基本アーキテクチャ，9-3
基本インタフェース，9-4
キャッシュ環境，9-8
キャッシュの整合性レベル，9-50
クラス，9-4
グループ，9-9
グループの定義，9-16，9-17
構成
clean-interval XML 要素，9-24
maxObjects プロパティ，9-24
maxSize プロパティ，9-24
ping-interval XML 要素，9-24
サブリージョン，9-9
整合性レベル
応答ありの分散，9-51
応答なしの分散，9-51
同期化，9-51
ローカル，9-51
属性，9-10
ディスク・オブジェクト，9-39
使用方法，9-40
定義，9-7
分散，9-40
ローカル，9-40
ディスク・キャッシュ
オブジェクトの追加，9-40
デフォルト・リージョン，9-8
プール・オブジェクト
アクセス，9-44
作成，9-43
使用方法，9-43
定義，9-8
プログラミングに関する注意点，9-38
分散オブジェクト，9-47
分散グループ，9-47
分散ディスク・オブジェクト，9-39
分散モード，9-46
分散リージョン，9-47
メモリー・オブジェクト
更新，9-7
スプール・メモリー・オブジェクト，9-7
定義，9-7
ローカル・メモリー・オブジェクト，9-7
リージョン，9-8
リージョンの定義，9-16
ローカル・ディスク・オブジェクト，9-39
ローカル・モード，9-46
Java Object Cache の属性
ALLOWNULL，9-13
CacheEventListener，9-14
CapacityPolicy，9-13
DefaultTimeToLive，9-13
DISTRIBUTE，9-11，9-46
GROUP_TTL_DESTROY，9-11
IdleTime，9-14
LOADER，9-11
MaxCount，9-15
MaxSize，9-14
MEASURE，9-13
ORIGINAL，9-12
Priority，9-14
REPLY，9-12
SPOOL，9-12
SYNCHRONIZE，9-12
SYNCHRONIZE_DEFAULT，9-13
TimeToLive，9-14
Version，9-14
ユーザー定義，9-15

Java Transaction API
　「JTA」を参照
Java-CORBA の例外マッピング，6-12
java.naming.factory.initial プロパティ，2-5，5-8
java.naming.provider.url
　JNDI 関連の環境プロパティ，2-6
　プロパティ，5-8，6-14
java.util.Hashtable
　JNDI，2-3
javax.naming.Context インタフェース
　JNDI，2-4
javax.naming パッケージ，2-2
javax.sql.DataSource，4-1，4-2
JDBC
　Oracle の拡張機能，4-21
　トランザクション，7-7
JMD
　デフォルトのコネクション・ファクトリ，3-5
JMS，3-1
　ConnectionFactory，3-5
　Destination，3-25
　OracleAS，3-2
　QueueConnectionFactory，3-4，3-5
　TopicConnectionFactory，3-4，3-5
　XAConnectionFactory，3-5
　XAQueueConnectionFactory，3-5
　XATopicConnectionFactory，3-5
　概要，1-2，3-1
　カスタム・リソース・プロバイダの構成，3-23
　キュー・コネクション・ファクトリ，3-4
　高可用性とクラスタリング，3-39
　システム・プロパティ，3-21
　トピック・コネクション・ファクトリ，3-4
　プログラミング・モデル，3-2
　プロバイダのインストール，3-24
　プロバイダの構成，3-24
　メッセージの受信 , JMS のステップ，3-7
　メッセージの送信 , JMS のステップ，3-5
　リソース・プロバイダ，3-23
　例 , ダウンロード・ページ，3-1
<jms-config> 要素，3-3
jms/ConnectionFactory，3-5
jms/QueueConnectionFactory，3-4，3-5
jms-server 要素，3-16
jms/TopicConnectionFactory，3-4，3-5
jms/XAConnectionFactory
　JMS，3-5
jms/XAQueueConnectionFactory，3-5
jms/XATopicConnectionFactory，3-5
jms.xml，3-3
　Oracle Enterprise Manager を使用した変更，3-3
　persistent-file 属性，3-11
JMS の概要，3-1
JMS プロバイダ
　インストール，3-24
　構成，3-24
JMS ユーティリティ
　browse，3-9
　check，3-9
　copy，3-9
　destinations，3-9
　drain，3-9
　durables，3-9
　help，3-9
　knobs，3-9
　move，3-9
　stats，3-9
　subscribe，3-9
　unsubscribe，3-9
JNDI，2-1
　application-client.jar，2-3，2-5
　application-client.xml，2-5
　com.evermind.server パッケージ，2-5
　InitialContext コンストラクタ，2-4
　java.util.Hashtable，2-3
　javax.naming.Context インタフェース，2-4
　jndi.properties ファイル，2-3
　orion-application-client.xml，2-5
　概要，1-2，2-1
　環境，2-4
　コンテキストの構成，2-3
　状態レプリケーション
　　概要，2-11
　　制限事項，2-12
　　有効化，2-12
　初期コンテキスト，2-2
　初期コンテキスト・ファクトリ，2-5
　例 , サーブレットがデータ・ソースを取得，2-9
jndi.jar ファイル，2-2
jndi.properties ファイル，5-8，6-14
　JNDI，2-3
JNDI 関連の環境プロパティ，2-6
　context.SECURITY_CREDENTIAL，2-6
　context.SECURITY_PRINCIPAL，2-6
　dedicated.rmicontext，2-6
　http.tunnel.path，2-6
　java.naming.provider.url，2-6
JNDI 初期コンテキスト
　J2EE アプリケーション・クライアントからの使用，
　　2-5
JNDI 初期コンポーネント
　J2EE アプリケーション・クライアントからの使用，
　　2-9
JNDI ルックアップ
　orion-ejb-jar.xml のプロパティ，7-4
JTA
　1 フェーズ・コミット
　　定義，7-2
　1 フェーズ・コミット , 構成，7-3
　2 フェーズ・コミット，7-7
　2 フェーズ・コミット , 構成，7-8
　2 フェーズ・コミット , 定義，7-2
　Bean 管理のトランザクション，7-2，7-7
　MDB，7-13
　概要，1-2
　境界設定，7-2，7-5
　クライアント・サイドのトランザクション境界，7-7
　コード・ダウンロード・サイト，7-2
　コンテナ管理のトランザクション，7-2，7-5
　再試行回数，7-12
　仕様の Web サイト，7-2
　タイムアウトの構成，7-11
　データ・ソースの取得，7-3
　デプロイメント・ディスクリプタ，7-5
　トランザクション，7-7
　トランザクション属性のタイプ，7-5

プログラムによるトランザクション境界，7-7
リソースの登録，7-2，7-3

K

knobs
JMS ユーティリティ，3-9

L

LOADER
Java Object Cache の属性，9-11
location
<data-source> 属性，4-9
log() メソッド，9-19
log 要素，3-19

M

MAA，4-26
Mandatory トランザクション属性タイプ，7-6
max-connect-attempts
<data-source> 属性，4-10
max-connections
<data-source> 属性，4-10
MaxCount
Java Object Cache の属性，9-15
maxObjects プロパティ，9-24
MaxSize
Java Object Cache の属性，9-14
maxSize プロパティ，9-24
max-tx-retries 属性，7-12
MDB
Bean 管理のトランザクションと JMS クライアントを使用，7-14
JTA，7-13
OC4J JMS でのトランザクション，7-13
OJMS，3-39
Oracle JMS でのトランザクション，7-13
コンテナ管理のトランザクションを使用，7-13
トランザクション，7-13
トランザクションのタイムアウト，7-13
MEASURE
Java Object Cache の属性，9-13
Message-Driven Bean，「MDB」を参照
<message-driven-deployment> 要素，7-13
min-connections
<data-source> 属性，4-10
move
JMS ユーティリティ，3-9

N

name
<data-source> 属性，4-9
netSearch() メソッド，9-19，9-51
Never トランザクション属性タイプ，7-6
NotSupported トランザクション属性タイプ，7-5

O

OBJECT_INVALIDATION イベント，9-36
OBJECT_UPDATED イベント，9-36
OC4J
RMI トンネリングをサポートするための構成，5-12
OC4J JMS
異常終了，3-14
永続性ファイルの管理，3-12
oc4j-connectors.xml，8-6
oc4j.jms.debug OracleAS JMS 制御ノブ，3-22
oc4j.jms.forceRecovery OracleAS JMS 制御ノブ，3-22
oc4j.jms.listenerAttempts OracleAS JMS 制御ノブ，3-21
oc4j.jms.maxOpenFiles OracleAS JMS 制御ノブ，3-21
oc4j.jms.messagePoll OracleAS JMS 制御ノブ，3-21
oc4j.jms.noDms OracleAS JMS 制御ノブ，3-22
oc4j.jms.pagingThreshold，3-22
oc4j.jms.saveAllExpired OracleAS JMS 制御ノブ，3-21
oc4j.jms.saveAllExpired プロパティ，3-15
oc4j.jms.serverPoll OracleAS JMS 制御ノブ，3-21
oc4j.jms.socketBufsize OracleAS JMS 制御ノブ，3-22
oc4j-ra.xml，8-4，8-5
OC4J クライアント JAR ファイル，5-3，6-3
OC4J サービスの概要，1-1
OC4J サンプル・コード・ページ，3-1
OC4J ホスティング Bean
非 OC4J コンテナからの起動，6-12
OC4J マウント・ポイント
構成，5-13
OCI ドライバ，4-23
OJMS
Enterprise Manager を使用したリソース・プロバイダの構成，3-26
リソース・プロバイダ，3-24
リソース・プロバイダとして，3-24
リソース・プロバイダとしての使用，3-24
リソース・プロバイダの構成，3-27，3-28
リソース・プロバイダの定義，3-26
OJMS の構成，3-46
opmn.xml ファイル
編集，5-7
OPMN の URL，6-12
Oracle Application Server Containers for J2EE（OC4J）
相互運用性，6-1
相互運用性フラグ，6-13
Oracle Enterprise Manager
jms.xml の変更，3-3
JMS ポートの構成，3-3
Oracle JMS
構成，3-26
Oracle JMS，「OJMS」を参照
Oracle JMS プロバイダ
OC4J XML ファイルでの構成，3-27
Oracle Maximum Availability Architecture，4-26
OracleAS JMS，3-2
管理，3-21
管理プロパティ , 表，3-21
構成，3-3
構成要素の階層ツリー，3-3
事前定義済の例外キュー，3-14
制御ノブ oc4j.jms.debug，3-22
制御ノブ oc4j.jms.forceRecovery，3-22
制御ノブ oc4j.jms.listenerAttempts，3-21
制御ノブ oc4j.jms.maxOpenFiles，3-21
制御ノブ oc4j.jms.messagePoll，3-21
制御ノブ oc4j.jms.noDms，3-22
制御ノブ oc4j.jms.saveAllExpired，3-21

制御ノブ oc4j.jms.serverPoll，3-21
制御ノブ oc4j.jms.socketBufsize，3-22
ファイル・ベースの永続性，3-10
ポート
構成，3-3
メッセージの期限切れ，3-15
メッセージのページング，3-15
ユーティリティ，3-8
ユーティリティ , 表，3-9
例外キュー , 事前定義済，3-14
OracleAS JMS の構成，3-40
OracleAS JMS ポート
構成，3-3
OracleAS Web Cache，9-2
oracleFatalError メソッド，7-12
oracle.ias.cache パッケージ，9-15
oracle.j2ee.connector パッケージ
AbstractPrincipalMapping，8-17
OracleTwoPhaseCommitDriver，7-10
ORIGINAL
Java Object Cache の属性，9-12
orion-application-client.xml
JNDI，2-5
orion-application.xml ファイル
EAR ファイル，4-8
JNDI リソース・プロバイダ，3-23
<resource-provider>，3-36，3-37，3-38
OrionCMTDataSource，7-10
orion-ejb-jar.xml ファイル，7-4，8-5
<entity-deployment> 要素，6-13
<ior-security-config> 要素，6-13
<session-deployment> 要素，6-13
ORMI，5-2
ORMI トンネリング，5-12
OTS，6-2

P

password
<data-source> 属性，4-9
persistent-file 属性，3-11
ping-interval 構成用 XML 要素，9-24
PoolAccess
close() メソッド，9-44
get() メソッド，9-44
getPool() メソッド，9-44
returnToPool() メソッド，9-44
オブジェクト，9-44
PoolInstanceFactory
実装，9-44
Priority
Java Object Cache の属性，9-14
PROVIDER_URL
InitialContext のプロパティ，2-4

Q

QoS
「Quality of Service」を参照
Quality of Service
J2CA タイプ，8-3
規約 , 指定，8-10
QueueConnectionFactory
JMS，3-4，3-5
queue-connection-factory 要素，3-18
queue 要素，3-17

R

RAC，4-26
rac-enabled
<data-source> 属性，4-10
RAR ファイル，8-2
ra.xml ファイル，8-4
release_Ownsership() メソッド，9-51
releaseOwnership() メソッド，9-48
Remote Method Invocation
「RMI」を参照
REPLY
Java Object Cache の属性，9-12
REPLY 属性，9-47
Required トランザクション属性タイプ，7-5
RequiresNew トランザクション属性タイプ，7-6
Resource Adapter Archive
「RAR ファイル」を参照
<resource-env-ref> 要素，3-32
ResourceProvider インタフェース
JMS，3-23
OJMS，3-24
<resource-provider> 要素，3-27，3-36，3-37，3-38
<resource-ref> 要素，3-32，4-16
<res-ref-name> 要素，4-17
returnToPool() メソッド，9-44
RMI
IIOP，6-2
ORMI，5-2
概要，1-2，5-1，5-2
<rmi-config> 要素，5-5
RMI/IIOP
contextFactory プロパティ，6-14
Java-CORBA の例外マッピング，6-12
java.naming.factory.initial プロパティ，5-8
java.naming.provider.url プロパティ，5-8，6-14
jndi.properties ファイル，5-8，6-14
OC4J マウント・ポイントの構成，5-13
Oracle Enterprise Manager を使用した拡張相互運用性のための構成，6-9
Oracle Enterprise Manager を使用した簡易相互運用性のための構成，6-6
OracleAS 環境での拡張相互運用性，6-9
OracleAS 環境での簡易相互運用性，6-6
拡張相互運用性のための手動による構成，6-10
簡易相互運用性のための手動による構成，6-8
スタンドアロン環境での拡張相互運用性，6-5
スタンドアロン環境での簡易相互運用性，6-4
RMIInitialContextFactory，2-10
<rmi-server> 要素，5-5
rmi.xml
編集，5-5
RMI トンネリング
サポートするための OC4J の構成，5-12

S

save() メソッド，9-40
schema

<data-source> 属性，4-10，4-15
SECURITY_CREDENTIAL
InitialContext のプロパティ，2-4
SECURITY_PRINCIPAL
InitialContext のプロパティ，2-4
<sep-config> 要素，5-7，6-13
server.xml
<application-server> 要素，8-9
RMI，5-5
<sep-config> 要素，5-7，6-13
タイムアウトの構成，7-11
<session-deployment> 要素，6-13
setAttributes() メソッド，9-19
setCacheEventListener() メソッド，9-36
SPI，2-2
SPOOL
Java Object Cache の属性，9-12，9-40
SQLServer
DataDirect でのデータ・ソース・エントリ，4-25
stats
JMS ユーティリティ，3-9
stmt-cache-size
<data-source> 属性，4-10
StreamAccess オブジェクト，9-7
InputStream，9-42
OutputStream，9-42
使用方法，9-42
Streams Advanced Queuing（AQ），3-24
subscribe
JMS ユーティリティ，3-9
Supports トランザクション属性タイプ，7-6
SYNCHRONIZE
Java Object Cache の属性，9-12，9-48
SYNCHRONIZE_DEFAULT
Java Object Cache の属性，9-13，9-48

T

TAF，4-27
記述子，4-30
構成，4-28，4-29
構成オプション，4-30
例外，4-31
TimeToLive
Java Object Cache の属性，9-14
TopicConnectionFactory
JMS，3-4，3-5
topic-connection-factory 要素，3-18
topic 要素，3-18
transaction-timeout 属性，7-13
<transaction-type> 要素，7-5，7-7
trans-attribute 要素，7-13
<trans-attribute> 要素，7-5，7-6
tx-retry-wait 属性，7-12
type-mapping，4-15

U

unsubscribe
JMS ユーティリティ，3-9
URL
corbaname，6-11
<data-source> 属性，4-9
OPMN，6-12
username
<data-source> 属性，4-9

V

Version
Java Object Cache の属性，9-14

W

wait-timeout
<data-source> 属性，4-10
Web Cache，9-2
Web Object Cache，9-2

X

xa-connection-factory 要素，3-18
xa-location
<data-source> 属性，4-9
XAQueueConnectionFactory
JMS，3-5
xa-queue-connection-factory 要素，3-18
XATopicConnectionFactory
JMS，3-5

い

異常終了
OC4J JMS，3-14
位置
J2EE Connector Architecture，8-9
J2EE Connector Architecture ファイル，8-10
デプロイメント・ディレクトリ，8-9
インストール
JMS プロバイダ，3-24
OC4J クライアント JAR ファイル，5-3，6-3
クライアント・サイド , RMI/IIOP，6-3
クライアント・サイド , RMI/ORMI，5-3
インポート
oracle.ias.cache，9-15

え

永続性ファイルの管理
OC4J JMS，3-12
エミュレートされたデータ・ソース，4-3
エミュレートされていないデータ・ソース，4-4
オブジェクト , 動作，4-18
エンタープライズ情報システム（EIS），8-2

お

オブジェクトの識別，9-6
オブジェクトのネーミング，9-6

か

階層ツリー
OracleAS JMS の構成要素，3-3
カスタム・リソース・プロバイダの構成
JMS，3-23

環境プロパティ
　JNDI 関連，2-6
管理
　OracleAS JMS，3-21
管理プロパティ
　OracleAS JMS，3-21

き

キャッシュ
　概念，9-2
キャッシュ・リージョン，9-8
キャッシング・スキーム，4-22
キュー・コネクション・ファクトリ
　JMS，3-4

く

クライアント・サイドのインストール要件
　RMI/IIOP，6-3
　RMI/ORMI，5-3
クライアント・サイドのトランザクション境界，7-7
クラスタリング
　問題 , JMS と OracleAS JMS，3-49

こ

高可用性，3-40，3-46
　Data Guard，4-26
　Oracle Maximum Availability Architecture，4-26
　Real Application Clusters，4-26
　SQL 例外，4-32
　TAF，4-27
　クラスタリング , JMS，3-39
　構成，3-46
　ネットワーク・フェイルオーバー，4-27
高可用性の概要，3-39
交換可能認証クラス，8-18
構成
　1 フェーズ・コミット，7-3
　2 フェーズ・コミット・トランザクション，7-8
　JMS プロバイダ，3-24
　JNDI コンテキスト，2-3
　JNDI の InitialContext，2-4
　JTA のタイムアウト，7-11
　OC4J XML ファイルで Oracle JMS プロバイダ，3-27
　Oracle JMS，3-26
　OracleAS JMS，3-3
　OracleAS JMS ポート，3-3
　RMI/IIOP での OC4J マウント・ポイント，5-13
　RMI トンネリングをサポートするための OC4J 構成，5-12
　server.xml でのタイムアウト，7-11
　高可用性，3-46
　高可用性 , OJMS，3-46
　高可用性 , OracleAS JMS，3-40
　接続プーリング，8-10
　相互運用性のための OC4J 構成，6-13
　データ・ソース・プロパティを使用したリソース・プロバイダ，3-27
構成ファイル
　データ・ソース，4-8
構成要素
　OracleAS JMS の階層ツリー，3-3
コネクション・ファクトリ
　構成例，3-20
　デフォルト , JMS，3-5
コンテキスト・ファクトリ
　使用，5-10，6-14
コンテナ管理のサインオン，8-13
コンテナ管理のトランザクション
　MDB，7-13
コンポーネント管理のサインオン，8-12

さ

サービス・プロバイダ・インタフェース，2-2
サンプル・コード・ページ , OC4J，3-1

し

事前定義済の例外キュー
　OracleAS JMS，3-14
状態レプリケーション
　JNDI
　　概要，2-11
　　制限事項，2-12
　　有効化，2-12
初期コンテキスト
　JNDI，2-2
　OC4J での作成，2-5
初期コンテキスト・ファクトリ
　JNDI，2-5
　同じアプリケーション内のオブジェクトへのアクセス，2-9
　同じアプリケーションにないオブジェクトへのアクセス，2-10

す

スタンドアロン・リソース・アダプタ，8-2

せ

生成されたスタブ JAR ファイル，6-4，6-5
セキュリティ相互運用性，6-2
接続
　キャッシング・スキーム，4-22
接続プーリング
　構成，8-10
宣言的なコンテナ管理のサインオン，8-14

そ

相互運用可能トランスポート，6-4
相互運用性
　EJB への追加，6-4，6-8
　OC4J の構成，6-13
　OC4J フラグ，6-13
　概要，1-2，6-1
　拡張 , Oracle Enterprise Manager を使用した構成，6-9
　拡張 , OracleAS 環境，6-9
　拡張 , 手動による構成，6-10
　簡易 , Oracle Enterprise Manager を使用した構成，6-6

簡易 , OracleAS 環境，6-6
簡易 , 手動による構成，6-8
構成ファイル，6-13
セキュリティ，6-2
トランザクション，6-2
トランスポート，6-2
ネーミング，6-2
相互運用性 , 拡張
スタンドアロン環境，6-5
相互運用性 , 簡易
スタンドアロン環境，6-4

た

タイムアウト
構成 , JTA，7-11

て

データ・ソース
2 フェーズ・コミット，4-20
DataDirect ドライバの使用方法，4-24
Enterprise Manager での定義，4-11
JDBC 接続，4-2
JNDI，4-2
OCI ドライバの使用方法，4-23
Oracle JDBC の拡張機能，4-21
XML ファイルの位置，4-8
移植可能 , ルックアップ，4-16
エミュレートされた，4-3，4-12
エミュレートされていない，4-4
JTA トランザクション，4-18
動作，4-18
エラー条件，4-19
JDBC ドライバ，4-19
ユーザー名，4-19
概要，1-2，4-1
構成，4-7
構成ファイル，4-8
混在，4-6
使用方法，4-16
接続の共有，4-18
接続の取得，4-16
タイプ，4-2
定義，4-2，4-7
デフォルト，4-12
ネイティブ，4-5
データ・ソース・エントリ
SQLServer, DataDirect，4-25
データ・ソースの概要，4-1
データ・ソース・プロパティ
リソース・プロバイダの構成，3-27
データベース構成，7-8
デプロイメント
相互運用性，6-13
デプロイメント・ディスクリプタ
J2EE Connector，8-4
JTA，7-5
JTA 属性
Mandatory トランザクション属性タイプ，7-6
Never トランザクション属性タイプ，7-6
NotSupported トランザクション属性タイプ，7-5
Required トランザクション属性タイプ，7-5
RequiresNew トランザクション属性タイプ，7-6
Supports トランザクション属性タイプ，7-6

と

透過的アプリケーション・フェイルオーバー
「TAF」を参照
トピック・コネクション・ファクトリ
JMS，3-4
トランザクション
2 フェーズ・コミット，7-8
Bean 管理，7-2
JDBC，7-7
JTA，7-7
MDB，7-13
OC4J JMS での MDB，7-13
Oracle JMS での MDB，7-13
境界設定，7-2，7-5
コンテナ管理，7-2
リソースの登録，7-2，7-3
トランザクション境界
クライアント・サイド , JTA，7-7
プログラムによる , JTA，7-7
トランザクション相互運用性，6-2
トランザクション属性のタイプ，7-5
トランスポート相互運用性，6-2
トンネリング
ORMI，5-12

に

認証クラス
OC4J 固有，8-15

ね

ネイティブ・データソース，4-5
ネーミング相互運用性，6-2
ネットワーク・フェイルオーバー，4-27

は

パスワード
間接化，4-13
不明瞭化，4-13

ふ

ファイル
相互運用性デプロイメント，6-13
ファイル・ベースの永続性
OracleAS JMS，3-10
フラグ
OC4J, 相互運用の開始，6-13
プログラミング・モデル
JMS，3-2
プログラム的なコンテナ管理のサインオン，8-15
プログラムによるトランザクション境界，7-7
分散トランザクション・コーディネータ，4-20

め

メッセージ

JMS での受信 , ステップ，3-7
JMS での送信 , ステップ，3-5
メッセージの期限切れ
OracleAS JMS，3-15
メッセージの受信
JMS のステップ，3-7
メッセージの送信
JMS のステップ，3-5
メッセージのページング
OracleAS JMS，3-15

ゆ

ユーザー定義
Java Object Cache の属性，9-15
ユーティリティ
OracleAS JMS，3-8
OracleAS JMS, 表，3-9

り

リソース・アダプタ
admin.jar，8-8
アンデプロイ，8-4
埋込み，8-2，8-9
概要，8-2
スタンドアロン，8-7
デプロイ，8-4
リソース・プロバイダ
JMS，3-23
OJMS，3-24
OJMS, Enterprise Manager を使用した構成，3-26
OJMS, 定義，3-26
データ・ソース・プロパティを使用した構成，3-27
リソース・プロバイダの構成
OJMS，3-27，3-28
リソース・プロバイダの使用
OJMS，3-24

れ

例
JNDI, サーブレットがデータ・ソースを取得，2-9
コネクション・ファクトリ構成，3-20
例外キュー , 事前定義済
OracleAS JMS，3-14