Oracle® Application Server Containers for J2EE

Enterprise JavaBeans 開発者ガイド

10*g*（9.0.4）

部品番号 : B12334-01

2004 年 1 月

ORACLE®

Oracle Application Server Containers for J2EE Enterprise JavaBeans 開発者ガイド , 10*g*（9.0.4）

部品番号 : B12334-01

原本名 : Oracle Application Server Containers for J2EE Enterprise JavaBeans Developer's Guide, 10*g* (9.0.4)

原本部品番号 : B10324-01

原本著者 : Sheryl Maring

Copyright © 2002, 2003 Oracle Corporation. 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 は Oracle Corporation およびその関連会社の登録商標です。その他の名称は、Oracle Corporation または各社が所有する商標または登録商標です。

目次

2 EJB 入門

3 CMP Entity Bean

4 エンティティ関連（E-R）のマッピング

5 EJB 問合せ言語

6 BMP Entity Bean

7 Message-Driven Bean

8 EJB アプリケーションのセキュリティの構成

9 高度な EJB のトピック

10 EJB のクラスタリング

11 Active Components for Java

A OC4J 固有の DTD リファレンス

B EJB 1.1 CMP Entity Bean

C EJB 1.1 から EJB 2.0 へのコンテナ管理の永続性の移行

D サード・パーティ・ライセンス

索引

はじめに

このマニュアルでは、Oracle Application Server Containers for J2EE（OC4J）用の Enterprise JavaBeans の作成手順について説明します。アプリケーション開発を支援するサンプル・コードが含まれています。

対象読者

このマニュアルは、OC4J 用の Enterprise JavaBeans を開発するあらゆる人に役立ちます。このマニュアルは、特にプログラマを対象としていますが、アーキテクチャ設計者、システム・アナリスト、プロジェクト・マネージャおよびその他 EJB アプリケーションに関心のある人なら誰にとっても役に立ちます。このマニュアルを効果的に使用するには、J2EE の運用知識が必要です。

前提知識

このマニュアルを読む前に、次のものを読むことをお薦めします。

- J2EE プログラミングの基礎を説明した J2EE に関する書籍。
- 『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』このマニュアルは、J2EE アプリケーションを OC4J 環境で使用するための最低条件の理解に役立ちます。
- このマニュアルを補完するものとして、Sun 社の EJB 2.0 仕様。このマニュアルは、EJB 2.0 仕様の詳細の基礎知識があることを前提としています。

参考文献

書籍

- 『Professional Java Server Programming, J2EE Edition』、Wrox Press Ltd、2000 年。
- 『Mastering Enterprise JavaBeans and the Java2 Platform Enterprise Edition』、Ed Roman 著、Wily Computer Publishing、1999 年。
- 『Designing Enterprise Applications with the Java2 Platform, Enterprise Edition』、Addison-Wesley 著、2000 年。
- 『Core Java second edition、Volume II』（Cornell & Horstmann 著、Prentice-Hall、1997 年）では、EJB に関連のある Java の概念がいくつか説明されています。
- 『Developer's Guide to Understanding Enterprise JavaBeans』は、EJB の概要を説明しており、`http://www.Nova-Labs.com` で参照可能です。

オンラインの情報

Java に関しては、オンラインで様々な有益な情報が参照可能です。たとえば、Sun 社のホーム・ページから、マニュアルやチュートリアルを表示またはダウンロード可能です。

`http://www.sun.com`

最新の EJB 2.0 仕様は、次のサイトで参照可能です。

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

次の Java に関する Web サイトもよく使用されています。

```
http://www.gamelan.com
```

Java API のドキュメントは、次のサイトを参照してください。

```
http://www.javasoft.com
```

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「EJB 概要」: EJB の概要を示します。

第 2 章「EJB 入門」: OC4J サーバー用のステートレス Session Bean の開発について説明します。

第 3 章「CMP Entity Bean」: CMP Entity Bean および CMP Entity Bean に関する高度な問題について説明します。

第 4 章「エンティティ関連（E-R）のマッピング」: OC4J 用の Entity Bean におけるコンテナ管理の関連（CMR）について説明します。

第 5 章「EJB 問合せ言語」: EJB 問合せ言語（EJB QL）を使用する query メソッドの概要と設定例を示します。

第 6 章「BMP Entity Bean」: BMP Entity Bean について説明します。

第 7 章「Message-Driven Bean」: MDB Entity Bean について説明します。

第 8 章「EJB アプリケーションのセキュリティの構成」: MDB Entity Bean について説明します。

第 9 章「高度な EJB のトピック」: EJB に関する高度な問題について説明します。

第 10 章「EJB のクラスタリング」: OC4J ノード間で EJB をクラスタリングする方法について説明します。

第 11 章「Active Components for Java」: 非同期通信とリクエスト / レスポンス通信のメリットを組み合せた新しい方法論について説明します。

付録 A「OC4J 固有の DTD リファレンス」: OC4J 固有のデプロイメント・ディスクリプタについて説明します。

付録 B「EJB 1.1 CMP Entity Bean」: EJB 1.1 CMP Entity Bean の方法論を説明します。

付録 C「EJB 1.1 から EJB 2.0 へのコンテナ管理の永続性の移行」: コンテナ管理の永続性を使用している EJB 1.1 アプリケーションを 2.0 仕様モデルに移行する方法について説明します。

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。

表記規則	意味
. . .	例の中で使用されている垂直の省略記号は、その例に直接関係しない情報が省略されていることを示します。
. . .	文またはコマンドの中で使用されている水平の省略記号は、例に直接関係のない文またはコマンドの一部が省略されていることを示します。
太字のテキスト	テキスト内の太字は、本文または用語集、あるいはその両方で定義されている用語を示します。
< >	山カッコで囲まれた部分は、ユーザーが指定する名称です。
[ ]	大カッコで囲まれている句は、その中から 1 つを選択するか、または何も選択しないオプションの句です。

1

EJB 概要

この章では、J2EE 仕様で完全に指定されている Enterprise JavaBeans（EJB）の概要について説明します。このマニュアルの他の章では、EJB の開発に必要な作業についてのみ説明します。

この章で説明する概要の詳細および例は、Sun 社による、EJB および J2EE Blueprint Architecture の推奨について説明されている文献を参照してください。

この章には、次の項目が含まれます。

EJB 2.0 の新機能

次の各項では、EJB 2.0 の新機能について説明します。

- ローカル・インタフェースのサポート
- ホーム・インタフェースのビジネス・メソッド
- Message-Driven Bean
- Enterprise JavaBeans 問合せ言語（EJB QL）
- CMP の関連
- CORBA サポート ― RMI-over-IIOP

ローカル・インタフェースのサポート

Oracle Application Server は、ローカル・インタフェースを完全にサポートしています。

クライアントは、Bean のインタフェースに定義されているメソッドによってのみ、Session Bean または Entity Bean にアクセスできます。これらのメソッドは、Bean のクライアントのビューを定義しています。Bean に関する他の局面（メソッドの実装、デプロイメント・ディスクリプタ設定、抽象スキーマ、データベース・アクセス・コール）は、モジュール性とカプセル化を提供するためにクライアントからはアクセスできません。適切に設計されたインタフェースは、ビジネス・ロジックの複雑さからクライアントを解放し、クライアントに影響を与えずに EJB の内部的な変更も可能にするため、J2EE アプリケーションの開発とメンテナンスを簡素化します。EJB では、リモートおよびローカルの 2 つのタイプのクライアント・アクセスをサポートしています。

リモート・アクセス

Enterprise Bean のリモート・クライアントには次の特性があります。

1. リモート・クライアントは、アクセスする Enterprise Bean とは異なるマシンおよび異なる Java Virtual Machine（JVM）上で実行できます。
2. このリモート・クライアントは、Web コンポーネント、J2EE アプリケーション・クライアントまたは別の Enterprise Bean のいずれかです。
3. リモート・クライアントにとって、Enterprise Bean の場所は透過的です。リモート・アクセスが可能な Enterprise Bean を作成するには、リモート・インタフェースとホーム・インタフェースをコーディングする必要があります。リモート・インタフェースでは、Bean 固有のビジネス・メソッドが定義されます。

ローカル・アクセス

ローカル・クライアントには次の特性があります。

1. ローカル・クライアントは、アクセスする Enterprise Bean と同じ JVM で実行する必要があります。
2. このローカル・クライアントは、Web コンポーネントまたは別の Enterprise Bean のどちらかです。
3. ローカル・クライアントにとって、アクセスする Enterprise Bean の場所は透過的ではありません。
4. 多くの場合、ローカル・クライアントは、別の Entity Bean とのコンテナ管理の関連を持つ Entity Bean です。ローカル・アクセスが可能な Enterprise Bean を作成するには、ローカル・インタフェースとローカル・ホーム・インタフェースをコーディングする必要があります。ローカル・インタフェースでは Bean のビジネス・メソッドが定義され、ローカル・ホーム・インタフェースではライフ・サイクルと finder メソッドが定義されます。

ローカル・インタフェースとコンテナ管理の関連

Entity Bean がコンテナ管理の関連のターゲットの場合は、ローカル・インタフェースが必要です。さらに、EJB 間の関連が双方向の場合は、両方の Bean にローカル・インタフェースが必要です。また、Entity Bean ではローカル・アクセスが必要であるため、コンテナ管理の関連に含まれる Entity Bean は同じ EJB コンテナに存在する必要があります。このような局所性の主な利点はパフォーマンスの向上にあります。通常、ローカル・コールはリモート・コールより高速です。

ローカル・アクセスとリモート・アクセスの比較

ローカル・アクセスとリモート・アクセスのどちらを可能にするかは、次の要因によって決定します。

1. コンテナ管理の関連 : Entity Bean がコンテナ管理の関連のターゲットの場合は、ローカル・インタフェースを使用する必要があります。
2. 関連する Bean が密結合方式か疎結合方式か : 密結合の Bean は相互に依存します。たとえば、完了した発注には 1 つ以上の明細項目が必要で、発注に属さない明細項目が存在することはありません。この関連をモデル化した `OrderEJB` と `LineItemEJB` の 2 つの Bean は密結合されます。密結合の Bean は、ローカル・アクセスに適しています。密結合の Bean は、1 つの論理ユニットとして組み合わされ、相互に頻繁にコールする可能性があるため、ローカル・アクセスよるパフォーマンスの向上が期待できます。

ホーム・インタフェースのビジネス・メソッド

ホーム・インタフェースのビジネス・メソッドは、Entity Bean の永続データを使用しないパブリック・メソッドで使用します。特定の Bean に関連付けられていない作業を実行するメソッドを指定する場合は、ホーム・インタフェースのビジネス・メソッドによってメソッドを公開できます。

Message-Driven Bean

EJB 2.0 Message-Driven Bean は、Oracle JMS を使用して実装できます。詳細な例は、第 7 章「Message-Driven Bean」で説明します。

Enterprise JavaBeans 問合せ言語（EJB QL）

EJB QL では、コンテナ管理の永続性を使用して Entity Bean の finder メソッドと select メソッドの問合せを定義します。SQL92 のサブセットである EJB QL には、Entity Bean の抽象スキーマに定義されている関連へのナビゲーションを可能にする拡張機能があります。抽象スキーマは、Entity Bean のデプロイメント・ディスクリプタの一部で、Bean の永続フィールドと関連を定義します。「抽象」という用語によって、このスキーマは基礎となるデータ・ストアの物理的なスキーマと区別されます。EJB QL 問合せの有効範囲には、同じ EJB JAR ファイルにパッケージされている関連の Entity Bean の抽象スキーマが含まれるため、抽象スキーマ名は EJB QL 問合せで参照されます。コンテナ管理の永続性を使用する Entity Bean の場合、EJB QL 問合せはすべての finder メソッド（`findByPrimaryKey` を除く）について定義する必要があります。EJB QL 問合せによって、finder メソッドの起動時に EJB コンテナで実行される問合せが決まります。

Oracle Application Server では、EJB QL を次の重要な機能とともにサポートしています。

- 自動コード生成 : EJB QL 問合せは、Entity Bean のデプロイメント・ディスクリプタで定義されます。この問合せは、Oracle Application Server への EJB のデプロイ時に、コンテナによってターゲット・データ・ストアの SQL 言語に自動的に変換されます。この変換によって、コンテナ管理の永続性を使用する Entity Bean は移植可能となり、そのコードは特定タイプのデータ・ストアに固定されなくなります。
- 最適化された SQL コード生成 : SQL コードの生成時に、Oracle Application Server は、データベース・アクセスを効率的にするために、バルク SQL を使用したり、バッチ処理された文をディスパッチするなど、いくつかの最適化を行います。
- Oracle データベースおよび Oracle 以外のデータベースのサポート : Oracle Application Server では、あらゆるデータベース（Oracle、MS SQL-Server、IBM DB/2、Informix および Sybase）に対して EJB QL を実行できます。
- 関連を持つ CMP: Oracle Application Server は、単一の Entity Bean および関連を持つ Entity Bean の両方について EJB QL をサポートし、あらゆる多重度と方向性をサポートします。

詳細および例は、第 5 章「EJB 問合せ言語」を参照してください。

CMP の関連

EJB 2.0 仕様では、Entity Bean 間の関連を指定できます。Entity Bean は、他の Entity Bean と関連を持つように定義できます。たとえば、プロジェクト管理アプリケーションでは、プロジェクトが一連のタスクで構成されているため、`ProjectEJB` Bean と `TaskEJB` Bean を関連付けることができます。Bean 管理の永続性を使用する Entity Bean に対しては、コンテナ管理の永続性を使用する Entity Bean とは異なる方法で関連を実装します。Bean 管理の永続性を使用する場合は、記述するコードによって関連が実装されます。コンテナ管理の永続性を使用する場合は、EJB コンテナが関連を実装します。このため、コンテナ管理の永続性を使用する Entity Bean での関連は、コンテナ管理の関連と呼ばれます。

- 関連フィールド : EJB が関連する Bean を識別する関連フィールド。関連フィールドは仮想フィールドで、access メソッドを使用して Enterprise Bean クラスに定義されます。永続フィールドとは異なり、関連フィールドは Bean の状態を表しません。
- コンテナ管理の関連での多重度 : 次の 4 つのタイプの多重度があり、Oracle Application Server では、そのすべてをサポートしています。
  - 1 対 1: 各 Entity Bean インスタンスが、別の Entity Bean の 1 つのインスタンスに関連付けられています。
  - 1 対多 : 1 つの Entity Bean インスタンスが、別の Entity Bean の複数のインスタンスに関連付けられています。
  - 多対 1: 1 つの Entity Bean の複数のインスタンスが、別の Entity Bean の 1 つのインスタンスに関連付けられています。この多重度は、1 対多の逆になります。
  - 多対多 : 複数の Entity Bean インスタンスが、相互に複数のインスタンスに関連付けられています。
- コンテナ管理の関連での方向 : 関連の方向は、双方向または単方向のどちらかになります。双方向の関連の場合、各 Entity Bean には他の Bean を参照する関連フィールドがあります。Entity Bean のコードは、この関連フィールドを介して関連するオブジェクトにアクセスできます。Entity Bean に関連フィールドがある場合は、関連するオブジェクトが「認識」されていることを示します。たとえば、`ProjectEJB` Bean は複数の `TaskEJB` Bean が属していることを認識し、各 TaskEJB Bean は `ProjectEJB` Bean に属していることを認識している場合、これらの Bean には双方向の関連があります。単方向の関連の場合、1 つの Entity Bean にのみ他の Bean を参照する関連フィールドがあります。Oracle Application Server は、EJB 間の双方向と単方向の両方の関連をサポートしています。
- EJB QL および関連を持つ CMP: EJB QL 問合せは、多くの場合、複数の関連をナビゲートします。関連の方向によって、問合せで Bean 間をナビゲートできるかどうかが決まります。Oracle Application Server では、EJB QL 問合せは、あらゆるタイプの多重度および双方向または単方向の関連について、CMP 関連を横断できます。

詳細は、第 3 章「CMP Entity Bean」、第 4 章「エンティティ関連（E-R）のマッピング」および第 5 章「EJB 問合せ言語」を参照してください。

Oracle Application Server のオブジェクト・リレーショナル・マッピング

Oracle Application Server には、単純なマッピング（1:1）と複雑な関連のマッピング（1:n、m:n）の両方を提供する、Entity Bean 用の独自の永続マネージャが最初から用意されています。Oracle Application Server は、EJB 2.0 O-R マッピング仕様を完全にサポートしています。

詳細は、第 4 章「エンティティ関連（E-R）のマッピング」を参照してください。

O-R マッピング ― TopLink との統合

Oracle Application Server は、TopLink for Java などの代表的な O-R マッピング・ソリューションと EJB コンテナを統合します。TopLink によって、開発者は、影響を最小限に抑えながら、オブジェクトと Enterprise JavaBeans をリレーショナル・データベース・スキーマに柔軟にマップできます。TopLink for Java は拡張マッピング機能を備えています。たとえば、Bean/ オブジェクト識別マッピング、タイプと値の変換、関連のマッピング（1:1、1:n、m:n）、オブジェクトのキャッシュとロック、バッチの記述、拡張された動的な問合せ機能などです。TopLink は、GUI マッピング・ツールである TopLink Mapping Workbench を提供しています。これによって、J2EE コンポーネントをデータベース・オブジェクトにマッピングする処理が簡素化されます。TopLink は、EJB 2.0 サポート、（自動的な、または開発者の構成による）双方向の関連のメンテナンス、XML を使用した（自動的な、または開発者の構成による）キャッシュ同期セッションの管理、および最適読取りロックを提供します。また、Oracle Application Server は、市販されている他の代表的な O-R マッピングとも統合されています。

TopLink の詳細は、『Oracle Application Server TopLink スタート・ガイド』を参照してください。

CORBA サポート ― RMI-over-IIOP

RMI-over-IIOP は J2EE 1.3 仕様の一部で、次の 2 つの重要な利点があります。

- RMI-over-IIOP によって、CORBA インタフェース定義言語（IDL）の知識がなくても Java プラットフォーム用の CORBA アプリケーションを記述できます。
- IIOP では、C++、Smalltalk など CORBA でサポートされている言語で記述されたアプリケーションが J2EE コンポーネントと通信できるため、既存のアプリケーションとプラットフォームを容易に統合できます。

Oracle Application Server は、RMI-over-IIOP をサポートし、次の重要な機能を提供します。

- IDL スタブと Helper クラスの自動生成 : 他の言語で CORBA アプリケーションを使用するために、IDL、CORBA スタブおよびスケルトンを次の方法で生成できます。

  1. J2EE アプリケーションのデプロイ時に、Oracle Application Server で自動的に生成されます。
  2. IDL は、`-idl` オプションを指定した `rmic` コンパイラを使用して、J2EE インタフェースから生成することもできます。また、開発者は、`-iiop` オプションを指定した `rmic` コンパイラを使用して、Java Remote Messaging Protocol（JRMP）スタブおよびスケルトン・クラスではなく、IIOP スタブおよび Tie クラスを生成できます。

- オブジェクトの値渡し : Oracle Application Server の RMI-IIOP 実装によって、開発者は、アプリケーション・コンポーネント間でシリアライズ可能な Java オブジェクトを渡すこと（オブジェクトの値渡し）ができ、柔軟性が向上します。
- POA サポート : Portable Object Adapter（POA）は、複数の ORB 実装で使用できるオブジェクト・アダプタを提供するように設計されています。これによって、異なるベンダーの実装に対処するために必要なリライトを最小限に抑えることができます。また、POA によって、少なくともクライアント側からは永続オブジェクトが使用可能になります。つまり、物理的にサーバーが何回も再起動されたり、その実装が多数の異なるオブジェクト実装で行われた場合でも、クライアントに関するかぎり、永続オブジェクトは常に有効であり、格納されているデータ値はメンテナンスされます。Oracle Application Server は、POA を完全にサポートしています。
- 他の ORB との相互運用 : Oracle Application Server の RMI-IIOP 実装では、CORBA 2.3 仕様をサポートする他の ORB との相互運用が行われます。従来の ORB は、オブジェクトの値渡し用の IIOP エンコーディングを処理できないため、相互運用の対象となりません。このサポートは、RMI 値クラス（文字列を含む）を IIOP で送信するために必要です。また、Oracle Application Server は、J2EE 1.3 仕様による相互運用可能な命名、セキュリティおよびトランザクション要素を完全にサポートします。これによって、開発者は、J2EE アプリケーションを作成し、CORBA を使用して他のアプリケーション・サーバー上の J2EE アプリケーションおよび既存システムと相互運用できます。

詳細は、『Oracle Application Server Containers for J2EE サービス・ガイド』の RMI と相互運用性の章を参照してください。

Oracle Application Server 10*g* のデフォルトの変更

リリース 2（9.0.3）でのデフォルト値は、次のように変更されました。

- CMP finder メソッドの遅延ロードは、デフォルトでオフになりました。詳細は、9-7 ページの「CMP Entity Bean の finder メソッドにおける遅延ロードの構成」を参照してください。
- 以前のリリースでは、1 対多関連のマッピングでデフォルトを使用する場合、関連表を使用していました。このリリースでは、デフォルトで外部キーを使用します。`-DassociateUsingThirdTable=true` システム・プロパティを指定して OC4J を起動し、前の動作をリストアすると、デフォルトで関連表を使用できます。
- CMP 2.0 Entity Bean の `trans-attribute` のデフォルト値は、`Required` に変更されました。詳細は、『Oracle Application Server Containers for J2EE サービス・ガイド』の JTA の章を参照してください。
- `max-tx-retries` のデフォルト値は 0（ゼロ）です。詳細は、『Oracle Application Server 10*g* パフォーマンス・ガイド』の EJB の項を参照してください。
- すべての EJB において、`max-instances` のデフォルト値は 0（ゼロ）に設定されています。

Enterprise JavaBeans の起動

Enterprise JavaBeans（EJB）には、Session Bean、Entity Bean および Message-Driven Bean の 3 種類があります。

- Session Bean は、ステートフルまたはステートレスのいずれかで、ビジネス・ロジック機能に使用されます。
  - ステートレス Session Bean は、ビジネス・サービスに使用されます。複数のコールに渡ってクライアントの状態を保持しません。
  - ステートフル Session Bean は、複数のクライアント・コールに渡って状態を保持します。このため、これらの Bean は、特定のクライアントのビジネス機能を、そのクライアントの存続期間中、管理します。
- Entity Bean は、通常、永続データの管理に使用されます。
- Message-Driven Bean は、JMS キューまたはトピックからのメッセージの受信に使用されます。

EJB には、2 つのクライアント・インタフェースが存在します。

- コンポーネント・インタフェース（リモートおよびローカル）：コンポーネント・インタフェースにより、オブジェクトのクライアントが起動可能なビジネス・メソッドを指定します。

- ホーム・インタフェース : ホーム・インタフェースにより、EJB のライフ・サイクル・メソッドを定義します。たとえば、Bean オブジェクトへの参照を作成および取得するメソッドなどです。

クライアントは、Bean のメソッドを起動する際、両方のインタフェースを使用します。

図 1-1 ステートレス Session Bean のイベント

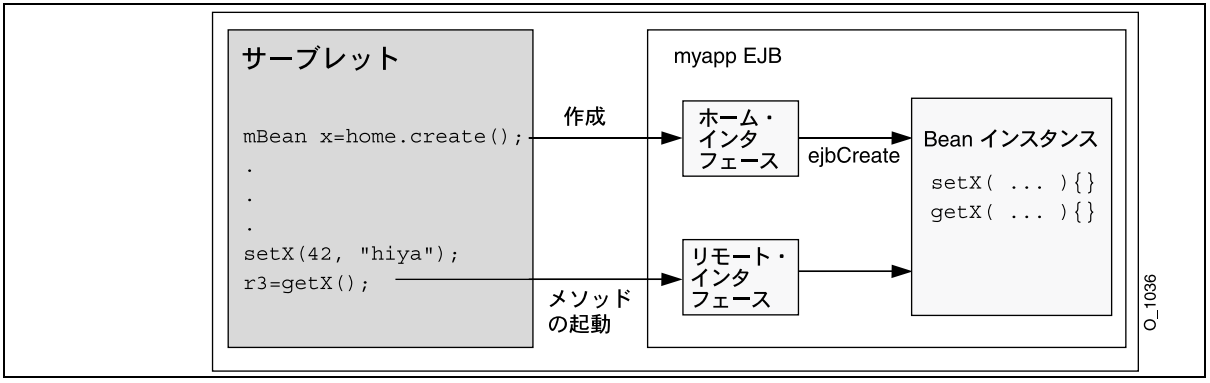


図 1-1 に、ステートレス Session Bean を示します。これは、次のステップに対応しています。

1. クライアント（スタンドアロンの Java クライアント、サーブレット、JSP またはアプレットのいずれか）は、通常、JNDI を通じて Bean のホーム・インタフェースを取得します。
2. クライアントは、ホーム・インタフェースの参照（ホーム・オブジェクト）で `create` メソッドを起動します。これにより、Bean インスタンスが作成され、Bean のコンポーネント・インタフェース（リモートまたはローカル・インタフェース）への参照が返されます。
3. クライアントは、コンポーネント・インタフェース（リモートまたはローカル・インタフェース）で定義されているメソッドを起動し、これにより、メソッド・コールが Bean インスタンス内の対応するメソッドに（スタブを通じて）委任されます。
4. クライアントは、コンポーネント・インタフェース（リモートまたはローカル・インタフェース）で定義されている `remove` メソッドを起動することにより、Bean のインスタンスを破棄できます。ステートレス Session Bean など、一部の Bean では、`remove` メソッドをコールできません。このような場合、コンテナによって Bean が削除されます。

EJB の実装

EJB を開発するには、次の 4 つの主要なコンポーネントを作成する必要があります。

- ホーム・インタフェース
- コンポーネント・インタフェース（リモートまたはローカル・インタフェース）
- Bean の実装
- 各 EJB のデプロイメント・ディスクリプタ

コンポーネント	説明
ホーム・インタフェース	コンテナ自体が実装するオブジェクト、つまりホーム・オブジェクトのインタフェースを指定します。ホーム・インタフェースには、Bean の作成方法を指定する `create()` メソッドなどのライフ・サイクル・メソッドが含まれています。
コンポーネント・インタフェース（リモートまたはローカル）	Bean で実装するビジネス・メソッドを指定します。また、Bean に、その他のコンテナ・サービス・メソッドも実装する必要があります。EJB コンテナは、Bean のライフ・サイクル中に様々なタイミングで、これらのメソッドを起動します。
Bean の実装	ホーム・インタフェースで定義されるメソッド（ライフ・サイクル・メソッド）、コンポーネント・インタフェースで定義されるメソッド（ビジネス・メソッド）、および必須のコンテナ・メソッド（コンテナ・コールバック関数）を実装する Java コードが含まれています。
デプロイメント・ディスクリプタ	デプロイする際の Bean の属性を指定します。これらにより、環境、インタフェース名、トランザクションのサポート、EJB の種類、および永続性情報など、構成の詳細を決定します。

Bean の実装

Bean は、`SessionBean`、`EntityBean` または `MessageDrivenBean` インタフェースのいずれかでメソッドを実装します。実装には、ホーム・インタフェースで定義されたライフ・サイクル・メソッド、コンポーネント・インタフェース（リモートまたはローカル）で定義されたビジネス・メソッド、および `SessionBean`、`EntityBean` または `MessageDrivenBean` インタフェースで定義されたコンテナ・コールバック関数のロジックが含まれています。

パラメータの受渡し

EJB を実装する場合、または EJB メソッドをコールするクライアント・コードを作成する場合、EJB で使用されるパラメータの受渡し規則に注意する必要があります。

Bean メソッドに渡すパラメータ、または Bean メソッドからの戻り値には、シリアライズ可能なすべての Java タイプを使用可能です。int、double など、Java のプリミティブ型は、シリアライズ可能です。java.io.Serializable インタフェースを実装する非リモート・オブジェクトは、すべて受渡し可能です。パラメータとして Bean に渡されるか Bean から返される非リモート・オブジェクトは、参照渡しではなく、値渡しされます。たとえば、次のように Bean メソッドをコールしたとします。

```
public class theNumber {
  int x;
}
...
bean.method1(theNumber);
```

この場合、Bean 内の method1() は、theNumber のコピーを受信します。Bean によってサーバーの theNumber オブジェクトの値が変更されても、値渡しのセマンティクスを使用しているため、クライアントにはこの変更は反映されません。

非リモート・オブジェクトが複合的である場合（複数のフィールドが含まれているクラスなど）、非静的で非一時的なフィールドのみコピーされます。

リモート・オブジェクトをパラメータとして渡す場合、リモート・オブジェクトのスタブが渡されます。パラメータとして渡されるリモート・オブジェクトは、リモート・インタフェースを拡張する必要があります。

次の項では、Bean へのパラメータの受渡しと、戻り値としてのリモート・オブジェクトについて説明します。

パラメータ・オブジェクト

EmployeeBean getEmployee メソッドは EmpRecord オブジェクトを返すため、このオブジェクトをアプリケーション内で定義しておく必要があります。この例では、EmpRecord クラスは、EJB インタフェースと同じパッケージに含まれています。

クラスは public として宣言されており、シリアライズされたリモート・オブジェクトとしてクライアントに値を返せるよう、java.io.Serializable インタフェースを実装する必要があります。次のように宣言します。

```
package employee;

public class EmpRecord implements java.io.Serializable {
  public String ename;
  public int empno;
  public double sal;
}
```

注意： `java.io.Serializable` インタフェースではメソッドを指定しません。クラスがシリアライズ可能であることのみ示します。そのため、`EmpRecord` クラスで他のメソッドを実装する必要はありません。

EJB の種類

EJB には、Session Bean、Entity Bean および Message-Driven Bean の 3 種類があります。

- Session Bean
- Entity Bean
- Message-Driven Bean

Session Bean

Session Bean は、1 つ以上のビジネス・タスクを実装します。Session Bean には、リレーショナル表内のデータの問合せおよび更新を実行するメソッドを含めることが可能です。Session Bean は、サービスの実装によく使用されます。たとえば、アプリケーション開発者は、データベース内の在庫データを取得および更新する Session Bean を 1 つ以上実装することがあります。

Session Bean は、サーバー・クラッシュやネットワーク障害が発生すると存続できないため、一時的です。サーバーのクラッシュ後、以前存在していた Bean をインスタンス化しても、以前のインスタンスの状態はリストアされません。状態は、Entity Bean の場合のみリストア可能です。

Session Bean は、`javax.ejb.SessionBean` インタフェースを実装します。定義は次のとおりです。

```
public interface javax.ejb.SessionBean extends javax.ejb.EnterpriseBean {
   public abstract void ejbActivate();
   public abstract void ejbPassivate();
   public abstract void ejbRemove();
   public abstract void setSessionContext(SessionContext ctx);
}
```

EJB は、`javax.ejb.SessionBean` インタフェースで指定されているように、最低、次のメソッドを実装する必要があります。

EJB メソッド	説明
`ejbCreate()`	コンテナは、Bean の作成直前にこのメソッドを起動します。ステートレス Session Bean は、このメソッドでは何も行いません。ステートフル Session Bean は、このメソッドで状態を初期化可能です。
`ejbActivate()`	コンテナは、Bean を再びアクティブ化した直後にこのメソッドを起動します。
`ejbPassivate()`	コンテナは、Bean を非アクティブ化する直前にこのメソッドを起動します。ステートフル Session Bean の非アクティブ化はオフにできます。9-3 ページの「EJB のライフ・サイクルに関する問題」を参照してください。
`ejbRemove()`	コンテナは、セッション・オブジェクトを破棄する前にこのメソッドを起動します。このメソッドにより、ファイル・ハンドルなどの外部リソースのクローズなど、必要なクリーン・アップが実行されます。
`setSessionContext (SessionContext ctx)`	このメソッドにより、Bean インスタンスをコンテキスト情報に関連付けます。コンテナは、Bean の作成後、このメソッドをコールします。Enterprise Bean は、トランザクション管理で使用するために、コンテキスト・オブジェクトへの参照をインスタンス変数に格納できます。自らのトランザクションを管理する Bean は、セッション・コンテキストを使用して、トランザクション・コンテキストを取得できます。

setSessionContext の使用

このメソッドを使用して、Bean のコンテキストの参照を取得します。Session Bean には、コンテナによって維持され、Bean から使用可能なセッション・コンテキストが存在します。Bean は、セッション・コンテキスト内のメソッドを使用して、コンテナへのコールバック・リクエストを送信できます。

コンテナは、Bean をインスタンス化した後、`setSessionContext` メソッドを起動して、Bean からセッション・コンテキストを取得できるようにします。コンテナは、トランザクション・コンテキストからはこのメソッドをコールしません。この時点で Bean がセッション・コンテキストを保存しなかった場合、Bean は二度とセッション・コンテキストにアクセスできなくなります。

コンテナはこのメソッドをコールする際、`SessionContext` オブジェクトの参照を Bean に渡します。Bean は、この参照を後の使用のために格納できます。次の例では、Bean がセッション・コンテキストを **`sessctx`** 変数に格納するところを示します。

```
import javax.ejb.*;
import oracle.oas.ejb.*;

public class myBean implements SessionBean {
   SessionContext sessctx;

   void setSessionContext(SessionContext ctx) {
      sessctx = ctx;   // session context is stored in
                       // instance variable
   }
   // other methods in the bean
}
```

javax.ejb.SessionContext インタフェースの定義は次のとおりです。

```
public interface SessionContext extends javax.ejb.EJBContext {
    public abstract EJBObject getEJBObject();
}
```

また、javax.ejb.EJBContext インタフェースの定義は次のとおりです。

```
public interface EJBContext {
    public EJBHome         getEJBHome();
    public Properties      getEnvironment();
    public Principal       getCallerPrincipal();
    public boolean         isCallerInRole(String roleName);
    public UserTransaction getUserTransaction();
    public boolean         getRollbackOnly();
    public void            setRollbackOnly();
}
```

Bean は、表 1-1 に示された操作を実行する際、セッション・コンテキストを必要とします。

表 1-1 SessionContext 操作

メソッド	説明
`getEnvironment()`	Bean のプロパティの値を取得します。
`getUserTransaction()`	トランザクション・コンテキストを取得します。このコンテキストにより、トランザクションをプログラムによってデマーケーション可能です。これは、トランザクション型の Bean でのみ有効です。
`setRollbackOnly()`	現在のトランザクションをコミットできないよう設定します。
`getRollbackOnly()`	現在のトランザクションがロールバック専用に指定されているかどうかを調べます。
`getEJBHome()`	Bean の対応する `EJBHome`（ホーム・インタフェース）のオブジェクト参照を取得します。

Session Bean には 2 種類あります。

- ステートレス Session Bean: ステートレス Session Bean は、メソッド間で状態または識別情報を共有しません。これは、特に、頻繁に短いリクエストを処理するための Bean のプールを持つ中間層アプリケーション・サーバーに使用されます。
- ステートフル Session Bean: ステートフル Session Bean は、カンバセーショナル・セッションで使用します。カンバセーショナル・セッションでは、インスタンス変数値やトランザクションの状態などをメソッド間で維持する必要があります。これらの Session Bean は、単一クライアントの存続期間中、そのクライアントにマッピングされます。

ステートレス Session Bean

ステートレス Session Bean は、クライアントの状態を維持しません。1 回のみ起動可能な Bean です。特定のクライアントに固有ではない、再利用可能なビジネス・サービスで使用されます。たとえば、一般的な為替換算、ローン金利の計算などに使用されます。ステートレス Session Bean には、クライアントから独立した、コール間に渡る読取り専用の状態が格納される場合があります。その後のコールは、プール内の他のステートレス Session Bean によって処理されます。情報は、1 回の起動中にのみ使用されます。

EJB コンテナは、複数のクライアントを処理するために、これらのステートレスな Bean のプールを維持しています。クライアントがリクエストを送信すると、プールからインスタンスが取得されます。Bean の情報を初期化する必要はありません。パラメータを持たない単一の `create/ejbCreate` のみ実装されており、これらのメソッド内には Bean の初期化は含まれていません。`remove/ejbRemove`、`ejbPassivate`、`ejbActivate` および `setSessionContext` メソッド内でアクションを実装する必要はありません。さらに、これらのメソッドは、ステートレス Session Bean 内では使用する必要がありません。かわりに、これらのメソッドは主に、状態を持つ EJB、つまりステートフル Session Bean および Entity Bean で使用されます。したがって、これらのメソッドは空であるか、または非常に単純になります。

実装	メソッド
ホーム・インタフェース	javax.ejb.EJBHome を拡張し、引数を取らない 1 つの create() ファクトリ・メソッド、および 1 つの remove() メソッドを必要とします。
コンポーネント・インタフェース（リモートまたはローカル）	リモート・インタフェースの場合は javax.ejb.EJBObject を拡張し、ローカル・インタフェースの場合は javax.ejb.EJBLocalObject を拡張します。また、Bean 実装で実装されるビジネス・ロジック・メソッドを定義します。
Bean の実装	SessionBean を実装します。このクラスは、パブリックとして宣言する必要があり、パブリックで空のデフォルト・コンストラクタを含み、finalize() メソッドは使用しません。また、コンポーネント・インタフェースで定義されたメソッドを実装します。ホーム・インタフェースの create() メソッドに一致するような、引数を持たない 1 つの ejbCreate メソッドが含まれている必要があります。ejbRemove などのコンテナ・サービス・メソッド用の空の実装を含めます。

ステートフル Session Bean

ステートフル Session Bean は、メソッド・コール間で状態を維持します。したがって、各クライアントに対し、ステートフル Session Bean のインスタンスが 1 つずつ作成されます。それぞれのステートフル Session Bean には、個別のクライアントの識別情報と、1 対 1 のマッピングが含まれています。このタイプの Bean の状態は、複数のコール間で維持されます。これは状態のシリアライズ化によって行われ、非アクティブ化と呼ばれます。そのため、非アクティブ化する状態はシリアライズ可能である必要があります。ただし、システム・クラッシュが発生した場合、この情報は維持されません。

プール内の複数のステートフル Bean の状態を維持するために、最近使用されていないステートフル Bean の対話状態が、2 次記憶装置にシリアライズ化されます。Bean のインスタンスがクライアントによって再びリクエストされると、プール内の Bean の状態がアクティブ化します。このようにして、すべてのリソースが高パフォーマンスで使用され、状態は失われません。

保存される状態のタイプには、リソースは含まれません。コンテナにより Bean 内の ejbPassivate メソッドが起動され、Bean がリソースのクリーン・アップを行います。これらのリソースには、保持されたソケット、データベース接続、および静的情報が含まれているハッシュテーブルなどがあります。これらのリソースは、すべて ejbActivate メソッド中に再割当ておよび再作成可能です。

注意： ステートフル Session Bean の非アクティブ化はオフにできます。9-3 ページの「EJB のライフ・サイクルに関する問題」を参照してください。

Bean のインスタンスでエラーが発生すると、Bean 内で継続的に状態を保存するアクションを実行していないかぎり、状態が失われます。ただし、フェイルオーバーに備えて常に状態を保存する必要がある場合、実装に Entity Bean を使用することをお薦めします。または、SessionSynchronization インタフェースを使用して、状態をトランザクションによって維持することも可能です。

たとえば、ステートフル Session Bean は、ショッピング・カート・オンライン・アプリケーションのサーバー・サイドを実装可能です。このアプリケーションには、購入可能な商品のリストを返し、アイテムを顧客のショッピング・カートに入れ、発注を行い、顧客のプロファイルを変更するなどの作業を行うためのメソッドが含まれます。

実装	メソッド
ホーム・インタフェース	javax.ejb.EJBHome を拡張し、1 つ以上の create() ファクトリ・メソッド、および 1 つの remove() メソッドを必要とします。
コンポーネント・インタフェース（リモートまたはローカル）	リモート・インタフェースの場合は javax.ejb.EJBObject を拡張し、ローカル・インタフェースの場合は javax.ejb.EJBLocalObject を拡張します。また、Bean 実装で実装されるビジネス・ロジック・メソッドを定義します。
Bean の実装	SessionBean を実装します。このクラスは、パブリックとして宣言する必要があり、パブリックで空のデフォルト・コンストラクタを含み、finalize() メソッドは使用しません。また、リモート・インタフェースで定義されたメソッドを実装します。ホーム・インタフェースで定義された create() メソッドに対応する ejbCreate メソッドが含まれている必要があります。つまり、各 ejbCreate メソッドは、パラメータ・シグネチャによって、ホーム・インタフェースで定義されている create メソッドに一致している必要があります。ejbRemove などのコンテナ・サービス・メソッドを実装します。また、コンテナ管理によるトランザクション用の SessionSynchronization インタフェースを実装します。これには、afterBegin、beforeCompletion および afterCompletion が含まれます。

Entity Bean

Entity Bean は、複合的なビジネス・エンティティです。Entity Bean は、ビジネス・エンティティをモデル化するか、またはビジネス・プロセス内の複数のアクションをモデル化します。Entity Bean は、データを使用するビジネス・サービスの提供、およびそのデータの計算によく使用されます。たとえば、アプリケーション開発者が、発注されたアイテムを取得し計算する Entity Bean を実装する場合があります。Entity Bean で、必要なタスクの実行中に複数の依存性のある永続オブジェクトを管理できます。

Entity Bean はリモート・オブジェクトで、永続データの管理および複雑なビジネス・ロジックの実行を行います。複数の依存性のある Java オブジェクトを使用可能で、主キーによって一意に識別可能です。Entity Bean は、複数のファイングレインな永続 Java オブジェクト内に格納されている永続データを使用するため、通常は、コースグレインな永続オブジェクトです。

Entity Bean は、サーバー・クラッシュやネットワーク障害が発生しても存続し続けるため、永続的です。Entity Bean が再びインスタンス化されると、以前のインスタンスの状態が自動的にリストアされます。

主キーによる一意の識別

各 Entity Bean には、永続的な識別情報が関連付けられています。つまり、主キーを保有している場合に取得可能な一意の識別情報が含まれています。主キーがあれば、クライアントは Entity Bean を取得可能です。Bean が使用不可の場合、コンテナは Bean をインスタンス化し、永続データを再移入します。

一意のキーのタイプは、Bean プロバイダによって定義されています。

注意： 主キーの詳細は、3-9 ページの「主キー」を参照してください。

永続データの管理

Bean が非アクティブ化された際に状態を保持し、フェイルオーバーが発生した際に状態をリカバリできるよう、Entity Bean のデータは永続性があります。データがコンテナによってデータベースなどのデータ記憶域システムに永続的に格納されるため、Entity Bean は存続可能です。Entity Bean は、次のいずれかの方法により、ビジネス・データを永続的にします。

- コンテナ管理による永続的な（CMP）Entity Bean を使用して、コンテナによって自動的に行う。
- Bean 管理による永続的な（BMP）Entity Bean 内で実装されるメソッドを使用して、プログラムによって行う。これらのメソッドでは、永続性を管理するために、JDBC または SQLJ を使用します。

Entity Bean は、コールバック・メソッドによってデータの永続性を維持します。これは、`javax.ejb.EntityBean` インタフェースで定義されています。Bean クラスに

EntityBean インタフェースを実装する場合は、選択した永続性のタイプ（Bean 管理の永続性またはコンテナ管理の永続性）で指定されているコールバック関数をそれぞれ作成します。コンテナは、指定されたタイミングでコールバック関数を起動します。

javax.ejb.EntityBean インタフェースの定義は次のとおりです。

```
public interface javax.ejb.EntityBean extends javax.ejb.EnterpriseBean {
   public abstract void ejbActivate();
   public abstract void ejbLoad();
   public abstract void ejbPassivate();
   public abstract void ejbRemove();
   public abstract void ejbStore();
   public abstract void setEntityContext(EntityContext ctx);
   public abstract voic unsetEntityContext();
}
```

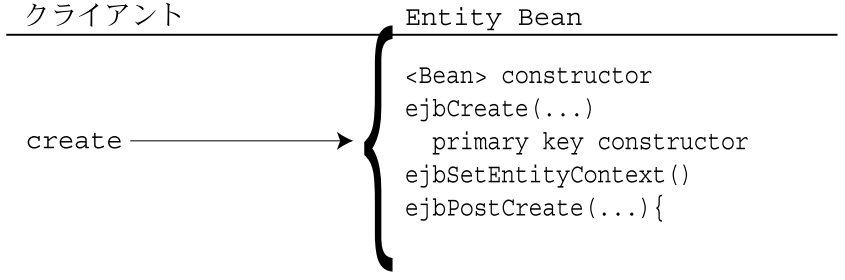
コンテナでは、これらのメソッドに次のような機能を設定します。

EJB メソッド	説明
ejbCreate	ホーム・インタフェースで宣言された各 create メソッドに対し、対応する ejbCreate メソッドを実装する必要があります。クライアントが create メソッドを起動すると、コンテナはまずコンストラクタを起動してオブジェクトをインスタンス化し、次に対応する ejbCreate メソッドを起動します。ejbCreate メソッドにより、次の処理が実行されます。 ■ データ用に、データベース行などの永続記憶域を作成する ■ 一意の主キーを初期化して返す
ejbPostCreate	コンテナは、環境の設定後、このメソッドを起動します。各 ejbCreate メソッドに対し、同じ引数を持つ ejbPostCreate メソッドが存在する必要があります。このメソッドを使用して、パラメータの初期化をエンティティ・コンテキスト内で、またはそこから実行することが可能です。
ejbRemove	コンテナは、セッション・オブジェクトを破棄する前にこのメソッドを起動します。このメソッドにより、ファイル・ハンドルなどの外部リソースのクローズなど、必要なクリーン・アップが実行されます。
ejbStore	コンテナは、トランザクションのコミットの直前にこのメソッドを起動します。これにより、永続データを、データベースなどの外部リソースに格納します。
ejbLoad	コンテナは、データをデータベースから再初期化する必要がある場合にこのメソッドを起動します。通常、これは Entity Bean のアクティブ化の後に行われます。

EJB メソッド	説明
setEntityContext	Bean インスタンスをコンテキスト情報に関連付けます。コンテナは、Bean の作成後、このメソッドをコールします。Enterprise Bean は、トランザクション管理で使用するために、コンテキスト・オブジェクトへの参照をインスタンス変数に格納できます。自らのトランザクションを管理する Bean は、セッション・コンテキストを使用して、トランザクション・コンテキストを取得できます。 また、このメソッド内に、Bean の存続期間中存在するリソースをすべて割り当てることが可能です。これらのリソースは、unsetEntityContext で解放する必要があります。
unsetEntityContext	関連付けられたエンティティ・コンテキストの設定を解除し、setEntityContext で割り当てられたリソースを解放します。
ejbActivate	コンテナは、以前に非アクティブ化されたオブジェクトをアクティブ化する前にこのメソッドを直接コールします。リソースの再取得が必要な場合、このメソッドで実行します。
ejbPassivate	コンテナは、オブジェクトを非アクティブ化する前にこのメソッドをコールします。ejbActivate で容易に再作成できるリソースを解放することにより、記憶領域を節約します。通常は、ソケットまたはデータベース接続などのように、非アクティブ化できないリソースを解放します。これらのリソースは、ejbActivate メソッドで取得します。

ejbCreate および ejbPostCreate の使用 ejbCreate などの特定のコールバック・メソッドが特定のときに起動されるため、Entity Bean と Session Bean は似ています。Entity Bean は、永続データ、主キーおよびコンテキスト情報の管理にコールバック関数を使用します。次の図に、Entity Bean の作成時にコールされるメソッドを示します。

図 1-2 Entity Bean の作成



setEntityContext の使用 Entity Bean のインスタンスは、このメソッドを使用して、コンテキストへの参照を維持します。Entity Bean には、コンテナによって維持され、Bean から使用可能なコンテキストが存在します。エンティティ・コンテキスト内のメソッドを使用して、セキュリティおよびトランザクションのロールなどの Bean に関する情報の取得が、Bean によって行われる場合があります。Bean に関してコンテキストから取得可能なすべての情報は、Sun 社の Enterprise JavaBeans 仕様を参照してください。

コンテナは、Bean をインスタンス化すると、`setEntityContext` メソッドを起動して、Bean からコンテキストを取得できるようにします。コンテナは、トランザクション・コンテキストからはこのメソッドをコールしません。この時点で Bean がコンテキストを保存しなかった場合、Bean は二度とコンテキストにアクセスできなくなります。

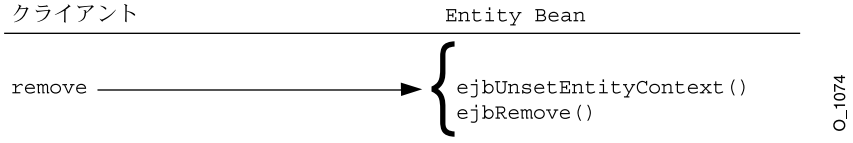
注意： インスタンスの存続期間中存在するリソースの割当ておよび破棄には、`setEntityContext` および `unsetEntityContext` メソッドも使用可能です。

コンテナはこのメソッドをコールする際、`EntityContext` オブジェクトの参照を Bean に渡します。Bean は、この参照を後の使用のために格納できます。次の例では、Bean がコンテキストを **`this.ctx`** 変数に格納するところを示します。

```
public void setEntityContext(EntityContext ctx) { this.ctx = ctx; }
```

ejbRemove の使用 クライアントが `remove` メソッドを起動すると、コンテナは図 1-3 に示されているメソッドを起動します。

図 1-3 Entity Bean の削除



ejbStore および ejbLoad の使用 さらに、永続データの管理のために、`ejbStore` および `ejbLoad` メソッドがコールされます。Bean 管理による永続的な Bean では、これらは最も重要なコールバック・メソッドです。コンテナ管理による永続的な Bean の場合、永続性はコンテナが管理するため、これらのメソッドは空で構いません。

- `ejbStore` メソッドは、オブジェクトが非アクティブ化される前、またはトランザクションが終了する直前に、コンテナによってコールされます。永続データを、データベースなどの外部リソースに格納します。
- `ejbLoad` メソッドは、オブジェクトがアクティブ化する前、トランザクションの開始後、または Entity Bean のインスタンス化の後に、コンテナによってコールされます。特定の Bean インスタンスに対する永続データのリストアを行います。

コンテナ管理の永続性

コンテナによって Bean の永続データを管理するよう選択できます。この場合、コンテナにより、永続データのデータベースへの格納およびリロードが行われるため、Bean のデータの永続性を管理するための一部のコールバック・メソッドを実装する必要がありません。コンテナ管理の永続性を使用する場合、コンテナが永続的マネージャ・クラスを起動し、これによって永続的管理ビジネス・ロジックが提供されます。さらに、主キー用の管理を提供する必要がありません。コンテナによって Bean のキーが提供されます。

- コールバック・メソッド : この場合でもコンテナはコールバック・メソッドを起動するため、他の用途のロジックを追加可能です。最低、すべてのコールバック・メソッド用に、空の実装を用意する必要があります。
- 主キー : CMP Bean の主キー・フィールドは、デプロイメント・ディスクリプタ内でコンテナ管理による永続的フィールドとして宣言する必要があります。主キー内のすべてのフィールドは、プリミティブ型か、シリアライズ可能な型、および SQL 型にマッピング可能な型に制限されています。

> **注意 :** 主キーの詳細は、3-9 ページの「主キー」を参照してください。

次の表に、Bean クラスのコールバック関数の実装要件を示します。

コールバック・メソッド	必要な機能
`ejbCreate`	主キーを含め、すべてのコンテナ管理による永続フィールドを初期化する必要があります。
`ejbPostCreate`	エンティティ・コンテキストに関連のある任意の初期化を追加することが可能です。
`ejbRemove`	外部リソースから永続データを削除するための機能は必要ありません。最低、コールバック用に空の実装を用意する必要があります。これにより、必要なクリーン・アップを実行するロジックが追加可能になります。
`ejbFindByPrimaryKey`	主キーをコンテナに返すための機能は必要ありません。主キーは、`ejbCreate` メソッドによって初期化された後、コンテナによって管理されます。ただし、このメソッド用に、空の実装を用意する必要があります。
`ejbStore`	このメソッド内に永続データを保存するための機能は必要ありません。永続マネージャにより、すべての永続データがデータベースに格納されます。ただし、最低、空の実装を提供する必要があります。
`ejbLoad`	このメソッド内に永続データをリストアするための機能は必要ありません。永続マネージャにより、すべての永続データがリストアされます。ただし、最低、空の実装を提供する必要があります。

コールバック・メソッド	必要な機能
setEntityContext	Bean インスタンスをコンテキスト情報に関連付けます。コンテナは、Bean の作成後、このメソッドをコールします。 Enterprise Bean は、トランザクション管理で使用するために、コンテキスト・オブジェクトへの参照をインスタンス変数に格納できます。自らのトランザクションを管理する Bean は、セッション・コンテキストを使用して、トランザクション・コンテキストを取得できます。 また、このメソッド内に、Bean の存続期間中存在するリソースをすべて割り当てることが可能です。これらのリソースは、unsetEntityContext で解放する必要があります。
unsetEntityContext	関連付けられたエンティティ・コンテキストの設定を解除し、setEntityContext で割り当てられたリソースを解放します。

注意： コンテナ管理の永続性の詳細は、第 3 章「CMP Entity Bean」を参照してください。

Bean 管理の永続性とコンテナ管理の永続性の違い

Entity Bean 内で永続データを管理する方法は、2 つあります。Bean 管理の永続性（BMP）と、コンテナ管理の永続性（CMP）です。BMP Bean と CMP Bean との間の主な違いは、Entity Bean のデータの永続性が何によって管理されるかです。CMP Bean の場合、コンテナによって永続性が管理されます。つまり、Bean のデプロイメント・ディスクリプタが、データのマッピングおよびデータの格納先を指定します。BMP Bean の場合、データを格納するためのロジックおよび格納先は、指定されたメソッド内にプログラミングされています。これらのメソッドが、必要なときにコンテナによって起動されます。

次の表で、具体的に、それぞれの種類の定義、およびそれらのプログラム面での違いと宣言の違いを示します。

	Bean 管理の永続性	コンテナ管理の永続性
永続性の管理	永続性管理を、ejbStore、ejbLoad、ejbCreate および ejbRemove EntityBean メソッド内に実装する必要があります。これらのメソッドには、永続データの格納およびリストアのためのロジックが含まれている必要があります。 たとえば、ejbStore メソッドの場合、Entity Bean のデータを適切なデータベースに格納するためのロジックが含まれている必要があります。そうでない場合、データが失われる可能性があります。	永続データの管理はユーザーが行う必要がありません。つまり、コンテナが、Bean のかわりに永続マネージャを起動します。 コミット前のデータの準備、またはデータベースからリフレッシュされた後のデータ操作には、ejbStore および ejbLoad を使用します。コンテナは、必ず、コミットの直前に ejbStore メソッドを起動します。さらに、CMP データをデータベースから再インスタンス化した直後に ejbLoad メソッドを起動します。
使用可能な finder メソッド	findByPrimaryKey メソッドおよびその他の finder メソッドが使用可能です。	findByPrimaryKey メソッドおよびその他の finder メソッド句が使用可能です。
CMP フィールドの定義	N/A	EJB デプロイメント・ディスクリプタ内で必須。主キーは、CMP フィールドとしても宣言する必要があります。
リソース格納先への CMP フィールドのマッピング	N/A	必須。永続マネージャによって異なります。
永続マネージャの定義	N/A	Oracle 固有のデプロイメント・ディスクリプタ内で必須。永続マネージャの説明は、次の項を参照してください。

Message-Driven Bean

Message-Driven Bean（MDB）は、JMS のみを使用する場合よりも簡単な、非同期通信を実装するための手段を提供します。MDB は、非同期 JMS メッセージを受信するために作成されました。このコンテナにより、JMS のキューおよびトピックに必要なセットアップのほとんどが処理されます。すべてのメッセージが、関連する MDB に送信されます。

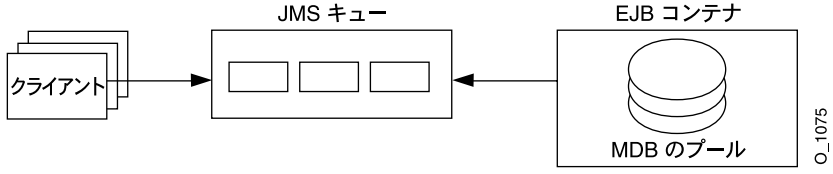
以前の EJB では、JMS メッセージの送受信ができませんでした。EJB タイプのオブジェクトの場合、JMS メッセージを受信するには、MDB を作成する必要がありました。これにより、他の Java オブジェクトと同期を取ることが可能なエンタープライズ・オブジェクトに、すべての非同期および公開、サブスクライブ機能が備わります。

MDB の目的は、プール内に存在し、JMS キューからの受信メッセージを受け取り、処理することです。コンテナは、キューから Bean を起動して、キューからの受信メッセージを処理します。MDB を直接起動するオブジェクトはありません。MDB の起動は、すべてコンテナから指示されます。いったんコンテナが MDB を起動すると、他の EJB または Java オブジェクトを起動して、リクエストを続行することが可能です。

MDB は、対話状態を保存せず、複数の受信リクエストの処理に使用される点において、ステートレス Session Bean に似ています。MDB は、クライアントから直接受信したリクエス

トを処理するのではなく、キューに入れられたリクエストを処理します。図 1-4 に、このように、クライアントがリクエストをキューに入れる様子を示します。コンテナは、キューからリクエストを取り出し、そのリクエストをプール内の MDB に渡します。

図 1-4 Message-Driven Bean



MDB は、javax.ejb.MessageDrivenBean インタフェースを実装します。また、これは javax.jms.MessageListener メソッドを継承します。これらのインタフェース内で、次のメソッドを実装する必要があります。

メソッド	説明
onMessage(msg)	コンテナは、この MDB に関連付けられている JMS キューからのメッセージをデキューし、このメソッドを起動することにより、このインスタンスにメッセージを渡します。このメソッドには、メッセージを適切に処理するための実装が必要です。
setMessageDrivenContext(ctx)	Bean の作成後、setMessageDrivenContext メソッドが起動されます。このメソッドは、EJB の setSessionContext および setEntityContext メソッドと似ています。
ejbCreate()	このメソッドは、ステートレス Session Bean の ejbCreate メソッドと同じように使用されます。このメソッドでは、初期化を行わないでください。ただし、このメソッド内で割り当てたリソースは、このオブジェクト用に存在します。
ejbRemove()	ejbCreate メソッド内で割り当てられたリソースをすべて削除します。

コンテナは、JMS メッセージの取得および確認の処理を行います。MDB には、JMS の詳細は関係ありません。MDB は、既存の JMS キューに関連付けられます。いったん関連付けられると、コンテナがメッセージのデキューおよび確認の送信を処理します。コンテナは、onMessage メソッドを通じて JMS メッセージを通信します。

注意： MDB および JMS の詳細は、第 7 章「Message-Driven Bean」および『Oracle Application Server Containers for J2EE サービス・ガイド』の JMS の章を参照してください。

Session Bean と Entity Bean の違い

Session Bean と Entity Bean の主な違いは、Entity Bean では、永続データ管理用のフレームワーク、永続識別情報および複雑なビジネス・ロジックが使用される点です。次の表に、Session Bean と Entity Bean で異なるインタフェースを示します。これら 2 種類の EJB の違いは、Bean クラスと主キーにあります。永続データ管理は、すべて Bean クラス・メソッド内で行われます。

Bean のコンポーネント	Entity Bean	Session Bean
ローカル・インタフェース	`javax.ejb.EJBLocalObject` を拡張	`javax.ejb.EJBLocalObject` を拡張
リモート・インタフェース	`javax.ejb.EJBObject` を拡張	`javax.ejb.EJBObject` を拡張
ローカル・ホーム・インタフェース	`javax.ejb.EJBLocalHome` を拡張	`javax.ejb.EJBLocalHome` を拡張
リモート・ホーム・インタフェース	`javax.ejb.EJBHome` を拡張	`javax.ejb.EJBHome` を拡張
Bean クラス	`javax.ejb.EntityBean` を拡張	`javax.ejb.SessionBean` を拡張
主キー	特定の Bean インスタンスの識別および取得に使用	Session Bean では不使用。ステートフル Session Bean には識別情報がありますが、外部化されていません。

EJB のコンテナ・サービス

EJB を使用するメリットの 1 つは、EJB コンテナによってセキュリティ・サービスとトランザクション・サービスが提供されることです。これらのサービス、RMI/IIOP、JNDI、データ・ソースおよび JMS については、次のマニュアルで説明しています。

表 1-2 J2EE のトピックに関する参照マニュアル

J2EE のトピック	トピックが説明されている OC4J マニュアル
JTA	『Oracle Application Server Containers for J2EE サービス・ガイド』
データ・ソース	『Oracle Application Server Containers for J2EE サービス・ガイド』
JNDI	『Oracle Application Server Containers for J2EE サービス・ガイド』
JMS	『Oracle Application Server Containers for J2EE サービス・ガイド』
RMI および RMI/IIOP	『Oracle Application Server Containers for J2EE サービス・ガイド』
セキュリティ	『Oracle Application Server Containers for J2EE セキュリティ・ガイド』
CS*i*V2	『Oracle Application Server Containers for J2EE セキュリティ・ガイド』
JCA	『Oracle Application Server Containers for J2EE サービス・ガイド』
Java Object Cache	『Oracle Application Server Containers for J2EE サービス・ガイド』
OracleAS Web Services	『Oracle Application Server Web Services 開発者ガイド』
HTTPS	『Oracle Application Server Containers for J2EE サービス・ガイド』

2

EJB 入門

Oracle Application Server Containers for J2EE（OC4J）をインストールし、ベース・サーバーとデフォルトの Web サイトを構成した後、J2EE アプリケーションの開発を開始します。この章では、簡単な J2EE の運用知識と EJB 開発の基本知識があることを前提としています。

次の各項では、OC4J を使用した EJB アプリケーションの開発およびデプロイについて説明します。

- EJB の開発 : 標準の J2EE 仕様の範囲内での EJB モジュールの開発およびテストを行います。
- EJB アプリケーションのデプロイ準備 : デプロイ前に、エンタープライズ・アプリケーションのマニフェスト・ファイルとして機能する XML ファイルを修正する必要があります。
- エンタープライズ・アプリケーションの OC4J へのデプロイ : エンタープライズ Java アプリケーションを Enterprise ARchive（EAR）ファイルにアーカイブし、OC4J にデプロイします。

この章では、ステートレス Session Bean の例を使用して、EJB の各開発フェーズおよびデプロイ手順について説明します。EJB の説明では、基本的な OC4J 固有の構成による単純な EJB を使用しています。ステートレス Session Bean の例は、OTN-J のサイト `http://otn.oracle.co.jp/sample_code/index.html` の OC4J のサンプル・コードのページからダウンロードしてください。

EJB の開発

OC4J 環境の EJB コンポーネントは、他の標準的な J2EE 環境での開発と同じ方法で開発します。EJB の開発手順は次のとおりです。

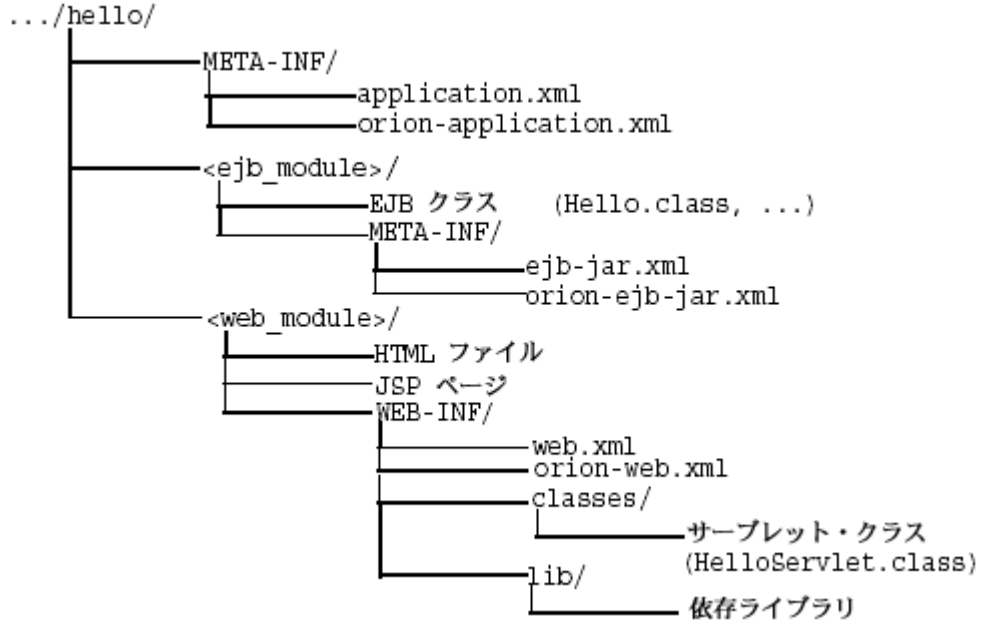
1. 開発ディレクトリの作成 : エンタープライズ・アプリケーション用の開発ディレクトリを作成します（図 2-1 を参照）。
2. EJB の実装 : EJB と、そのホーム・インタフェース、コンポーネント・インタフェースおよび Bean 実装を開発します。
3. EJB へのアクセス : クライアントを開発して、リモート・インタフェースまたはローカル・インタフェースを介して Bean にアクセスします。
4. デプロイメント・ディスクリプタの作成 : EJB アプリケーション内のすべての Bean に対し、標準の J2EE EJB デプロイメント・ディスクリプタを作成します。
5. EJB アプリケーションのアーカイブ : EJB ファイルを JAR ファイルにアーカイブします。

開発ディレクトリの作成

アプリケーションは、任意な方法で開発できますが、アプリケーションを簡単に見つけられるように、一貫性のある命名規則を使用することをお薦めします。1 つの方法としては、エンタープライズ Java アプリケーションを、1 つの親ディレクトリ構造内に実装し、アプリケーションの各モジュールをそれぞれのサブディレクトリに分ける方法があります。

ここで使用している hello の例は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』で説明されているディレクトリ構造を使用して開発されています。図 2-1 では、EJB および Web モジュールは、`hello` アプリケーションの親ディレクトリ内に存在し、それぞれ別々のディレクトリ内で開発されている点に注意してください。

図 2-1 hello のディレクトリ構造



注意： EJB モジュールの場合、モジュールの最上位（`ejb_module`）は、クラスの検索パスの開始を示します。したがって、パッケージに所属するクラスは、この下のネストされたディレクトリ構造内に存在しているとみなされます。たとえば、パッケージ・クラス '`myapp.Hello.class`' の参照は、"`...hello/ejb_module/myapp/Hello.class`" 内に存在するとみなされます。

EJB の実装

Session EJB または Entity EJB を実装する際、次のものを作成します。

> **注意：** Message-Driven Bean の要件は、次に説明する要件と同じではありませんが似ています。詳細は、第 7 章「Message-Driven Bean」を参照してください。

1. Bean のホーム・インタフェース。ホーム・インタフェースは Bean の `create` メソッドを定義します。EJB が Entity Bean の場合は、その Bean の finder メソッドも定義します。
   a. リモート・ホーム・インタフェースは、`javax.ejb.EJBHome` を拡張します。
   b. ローカル・ホーム・インタフェースは、`javax.ejb.EJBLocalHome` を拡張します。
2. Bean のコンポーネント・インタフェース。
   a. リモート・インタフェースは、クライアントがリモートで起動できるメソッドを宣言します。`javax.ejb.EJBObject` を拡張します。
   b. ローカル・インタフェースは、連結された Bean がローカルで起動できるメソッドを宣言します。`javax.ejb.EJBLocalObject` を拡張します。
3. Bean の実装には、次のものが含まれます。
   a. コンポーネント・インタフェースで宣言されているビジネス・メソッドの実装
   b. `javax.ejb.SessionBean` または `javax.ejb.EntityBean` インタフェースを継承しているコンテナのコールバック・メソッド
   c. ホーム・インタフェースの `create` メソッドに一致する次の `ejb*` メソッド
      * ステートレス Session Bean の場合は、パラメータのない `ejbCreate` メソッドを提供します。
      * ステートフル Session Bean の場合は、ホーム・インタフェースで定義された create メソッドのパラメータに一致するパラメータを持つ `ejbCreate` メソッドを提供します。
      * Entity Bean の場合は、ホーム・インタフェースで定義された `create` メソッドのパラメータに一致するパラメータを持つ `ejbCreate` および `ejbPostCreate` メソッドを提供します。

ホーム・インタフェースの作成

ホーム・インタフェース（リモートおよびローカル）は、Bean インスタンスの作成に使用され、Bean の create メソッドを定義します。それぞれの EJB タイプでは、次のようにして create メソッドを定義可能です。

EJB タイプ	create のパラメータ
ステートレス Session Bean	パラメータなしの 1 つの create メソッドのみ使用可能。
ステートフル Session Bean	それぞれ定義されたパラメータを持つ 1 つ以上の create メソッドを使用可能。
Entity Bean	それぞれ定義されたパラメータを持つ 0（ゼロ）以上の create メソッドを使用可能。すべての Entity Bean で、1 つ以上の finder メソッドを定義する必要があります。そのうちの 1 つ以上は findByPrimaryKey メソッドである必要があります。

各 create メソッドにつき、対応する ejbCreate メソッドが Bean 実装で定義されます。

リモート起動 リモート・クライアントは、リモート・インタフェースを介して EJB を起動します。クライアントは、リモート・ホーム・インタフェースで宣言された create メソッドを起動します。コンテナは、Bean 実装内の、適切なパラメータ・シグネチャを持つ ejbCreate メソッドにクライアント・コールを渡します。新しい EJB オブジェクトの状態を初期化するために、パラメータの引数を使用できます。

1. リモート・ホーム・インタフェースでは、javax.ejb.EJBHome インタフェースを拡張する必要があります。
2. すべての create メソッドで、次の例外をスローすることができます。
   - javax.ejb.CreateException
   - javax.ejb.EJBException または別の RuntimeException

例 2-1 Session Bean のリモート・ホーム・インタフェース

次のサンプル・コードでは、HelloHome というステートレス Session Bean のリモート・ホーム・インタフェースを示します。

```
package hello;

import javax.ejb.*;
import java.rmi.*;

public interface HelloHome extends EJBHome
{
  public Hello create() throws CreateException, RemoteException;
}
```

ローカル起動 EJB は、同じコンテナに存在するクライアントからローカルでコールできます。したがって、連結された Bean、JSP またはサーブレットは、ローカル・ホーム・インタフェースで宣言された create メソッドを起動します。コンテナは、Bean 実装内の、適切なパラメータ・シグネチャを持つ ejbCreate メソッドにクライアント・コールを渡します。新しい EJB オブジェクトの状態を初期化するために、パラメータの引数を使用できます。

1. ローカル・ホーム・インタフェースでは、javax.ejb.EJBLocalHome インタフェースを拡張する必要があります。
2. すべての create メソッドで、次の例外をスローすることができます。
   - javax.ejb.CreateException
   - javax.ejb.EJBException または別の RuntimeException

例 2-2 Session Bean のローカル・ホーム・インタフェース

次のサンプル・コードでは、HelloLocalHome というステートレス Session Bean のローカル・ホーム・インタフェースを示します。

```
package hello;

import javax.ejb.*;

public interface HelloLocalHome extends EJBLocalHome
{
 public HelloLocal create() throws CreateException, EJBException;
}
```

コンポーネント・インタフェースの作成

コンポーネント・インタフェースでは、クライアントから起動可能な Bean のビジネス・メソッドを定義します。

リモート・インタフェースの作成 リモート・インタフェースでは、リモート・クライアントによって起動可能なビジネス・メソッドを定義します。リモート・インタフェースを開発するための要件は次のとおりです。

1. Bean のリモート・インタフェースは、javax.ejb.EJBObject インタフェースを拡張する必要があり、そのメソッドは java.rmi.RemoteException 例外をスローする必要があります。
2. リモート・インタフェースとそのメソッドは、リモート・クライアントに対する public として宣言する必要があります。
3. リモート・インタフェース、すべてのメソッド・パラメータおよび戻り型はシリアライズ可能である必要があります。一般的に、RMI は両側のオブジェクトをマーシャリング

およびアンマーシャリングするため、クライアントと EJB の間で受渡しされるオブジェクトは、すべてシリアライズ可能である必要があります。

4. シリアライズ可能であれば、どのような例外でもクライアントにスロー可能です。`EJBException` および `RemoteException` を含めた実行時例外は、リモート実行時例外としてクライアントに転送されます。

例 2-3 Hello Session Bean のリモート・インタフェース例

次のサンプル・コードでは、`Hello` というリモート・インタフェースとその定義済のメソッドが示されています。各メソッドは、ステートレス Session Bean で実装されます。

```
package hello;

import javax.ejb.*;
import java.rmi.*;

public interface Hello extends EJBObject
{
  public String sayHello(String myName) throws RemoteException;
}
```

ローカル・インタフェースの作成 ローカル・インタフェースでは、ローカル（連結された）クライアントから起動可能な Bean のビジネス・メソッドを定義します。

1. Bean のローカル・インタフェースでは、`javax.ejb.EJBLocalObject` インタフェースを拡張する必要があります。
2. ローカル・インタフェースとそのメソッドは、`public` として宣言します。

例 2-4 Hello Session Bean のローカル・インタフェース

次のサンプル・コードでは、`HelloLocal` というローカル・インタフェースとその定義済のメソッドが示されています。各メソッドは、ステートレス Session Bean で実装されます。

```
package hello;

import javax.ejb.*;

public interface HelloLocal extends EJBLocalObject
{
  public String sayHello(String myName) throws EJBException;
}
```

Bean の実装

Bean には、アプリケーションのビジネス・ロジックが含まれています。次のメソッドを実装します。

1. これらの各メソッドのシグネチャは、Bean が RemoteException をスローしない場合を除き、リモートまたはローカル・インタフェースのシグネチャに一致している必要があります。ローカル・インタフェースおよびリモート・インタフェースは Bean 実装を使用するため、Bean 実装では RemoteException をスローできません。
2. ライフ・サイクル・メソッドは、SessionBean または EntityBean インタフェースから継承されます。これらのメソッドには、ejbActivate、ejbPassivate などの ejb<Action> メソッドが含まれます。
3. ホーム・インタフェースで宣言された各 create メソッドに対応する ejbCreate メソッド。クライアントが create メソッドを起動すると、コンテナによって対応する ejbCreate メソッドが起動されます。
4. ビジネス・ロジックに使用される Bean またはパッケージに対してプライベートであるメソッド。これには、パブリック・メソッドがリクエストされた作業の完了に使用するプライベート・メソッドも含まれます。

例 2-5 Hello ステートレス Session Bean の実装

次のサンプル・コードでは、Hello の例の Bean 実装が示されています。

> **注意：** ステートレス Session Bean の例は、OTN-J のサイト http://otn.oracle.co.jp/sample_code/index.html の OC4J のサンプル・コードのページからダウンロードしてください。

```
package hello;

import javax.ejb.*;

public class HelloBean implements SessionBean
{
  public SessionContext ctx;

  public HelloBean()
  {    // constructor
  }

  public void ejbCreate() throws CreateException
  {   // when bean is created
  }
```

```
  public void ejbActivate()
  {    // when bean is activated
  }

  public void ejbPassivate()
  {    // when bean is deactivated
  }

  public void ejbRemove()
  {   // when bean is removed
  }

  public void setSessionContext(SessionContext ctx)
  {    this.ctx = ctx;
  }

  public void unsetSessionContext()
  {    this.ctx = null;
  }

  public String sayHello(String myName) throws EJBException
  {
    return ("Hello " + myName);
  }
}
```

注意： この例は、OTN-J のサイト `http://otn.oracle.co.jp/sample_code/index.html` の OC4J のサンプル・コードのページからダウンロードしてください。

EJB へのアクセス

クライアントから EJB にアクセスするには、次の作業を行う必要があります。

1. リモートの場合は、`oc4j.jar` ファイルをダウンロードします。
2. 必要に応じて、接続用の JNDI プロパティを設定します。
3. 接続で使用する `InitialContextFactory` を決定します。
4. デプロイメント・ディスクリプタで構成された JNDI 名または EJB 参照を使用して、EJB を取得します。

これらの詳細は、次の各項で説明します。

- OC4J.JAR のクライアントでのインストール
- EJB 参照のルックアップ方法

- EJB を起動するためのクライアント実装
- EJB 参照情報
- JNDI プロパティの設定
- 初期コンテキスト・ファクトリ・クラスの使用
- 別のアプリケーションの EJB へのアクセス
- リモート・サーバーの EJB へのアクセス

OC4J.JAR のクライアントでのインストール

EJB にアクセスするためには、クライアント側で OTN-J（`http://otn.oracle.co.jp/`）のダウンロード・ページから、`oc4j_client.zip` ファイルをダウンロードする必要があります。`CLASSPATH` にあるディレクトリに JAR ファイルを解凍します。この JAR ファイルには、クライアントとの相互作用に必要なクラスが含まれています。この JAR ファイルをブラウザにダウンロードする場合は、特定の権限を付与する必要があります。

EJB 参照のルックアップ方法

クライアントで EJB へのコールの実装を開始する前に、Bean の EJB 参照の JNDI 取得について次の点を考慮してください。

- Bean でメソッドを実行するためには、クライアント・コード内でターゲット Bean への EJB 参照を取得します。ターゲット Bean の論理名を設定するか、または JNDI 名を使用します。
  - 論理名を使用する場合 : クライアントの XML 構成ファイルを変更し、`<ejb-ref>` 要素にターゲット Bean の情報を設定します。`<ejb-ref-name>` 要素で指定された論理名が JNDI ルックアップで使用されます。`<ejb-ref>` 要素および `<ejb-ref-name>` 要素の詳細は、2-16 ページの「EJB 参照情報」を参照してください。
  - 実際の名前を使用する場合 : Bean の実際の名前が JNDI ルックアップで使用されます。この名前は、ターゲット Bean の `ejb-jar.xml` XML デプロイメント・ディスクリプタの `<ejb-name>` 要素で指定されています。
- EJB へのアクセス方法は、起動する Bean に対するクライアントの存在場所によって異なります。
  - クライアントがターゲット Bean と一緒に置かれているか。同じアプリケーションにデプロイされているか。または、ターゲット Bean は、このクライアントの親であるアプリケーションの一部か。この場合、JNDI プロパティを設定する必要はありません。
  - そうでない場合、JNDI プロパティを設定する必要があります。JNDI プロパティの設定方法は 2 つあります。詳細は、2-16 ページの「JNDI プロパティの設定」を参照してください。

EJB を起動するためのクライアント実装

すべての EJB クライアントは、次のようにして、Bean のインスタンス化、そのメソッドの起動、および Bean の破棄を行います。

1. JNDI ルックアップによってホーム・インタフェースをルックアップします。JNDI 規則および EJB 仕様規則に従って、Bean 参照を取得します。これには、Bean がクライアントに対してリモートである場合、JNDI プロパティの設定も含まれます。2-10 ページの「EJB 参照のルックアップ方法」を参照してください。
2. JNDI ルックアップから返されたオブジェクトを、次のようにホーム・インタフェースにナローイングします。
   a. リモート・インタフェースにアクセスする場合は、`PortableRemoteObject.narrow` メソッドを使用して、返されたオブジェクトをナローイングします。
   b. ローカル・インタフェースにアクセスする場合は、返されたオブジェクトをローカル・インタフェース型でキャストします。
3. 返されたオブジェクトを通じて、サーバーの Bean のインスタンスを作成します。ホーム・インタフェースで `create` メソッドを起動すると、新しい Bean がインスタンス化され、Bean 参照が返されます。

   注意： すでにインスタンス化されている Entity Bean の場合、Bean 参照を finder メソッドによって取得可能です。

4. コンポーネント・インタフェース（リモートまたはローカル）で定義されたビジネス・メソッドを起動します。
5. 完了後、`remove` メソッドを起動します。これにより、Bean インスタンスが削除されるか、プールに返されます。`remove` メソッドに対する動作は、コンテナが制御します。

例 2-6 に、これらの手順を示します。

例 2-6 ローカル・クライアントとして機能するサーブレット

次の例は、Hello Bean に連結されたサーブレットから実行されます。したがって、Session Bean ではローカル・インタフェースが使用され、JNDI ルックアップに JNDI プロパティは必要ありません。

注意： JNDI 名は、次のように、この Session Bean の EJB デプロイメント・ディスクリプタの <ejb-local-ref> 要素で指定されます。

```
<ejb-local-ref>
  <ejb-ref-name>ejb/HelloBean</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local-home>hello.HelloLocalHome</local-home>
  <local>hello.HelloLocal</local>
</ejb-local-ref>
```

```
package hello;

import javax.servlet.http.*;
import javax.servlet.*;
import javax.ejb.*;
import javax.naming.*;
import java.io.IOException;

public class HelloServlet extends HttpServlet
{
  HelloLocalHome helloHome;
  HelloLocal hello;

  public void init() throws ServletException
  {
    try {
     // 1. Retreive the Home Interface using a JNDI Lookup
     // Retrieve the initial context for JNDI.
     // No properties needed when local
      Context context = new InitialContext();

      // Retrieve the home interface using a JNDI lookup using
      // the java:comp/env bean environment variable
      // specified in web.xml
      helloHome = (HelloLocalHome)
             context.lookup("java:comp/env/ejb/HelloBean");

     //2. Narrow the returned object to be an HelloHome object.
     // Since the client is local, cast it to the correct object type.
     //3. Create the local Hello bean instance, return the reference
```

```
      hello = (HelloLocal)helloHome.create();

    } catch(NamingException e) {
      throw new ServletException("Error looking up home", e);
    } catch(CreateException e) {
      throw new ServletException("Error creating local hello bean", e);
    }
  }

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
    throws ServletException, IOException
  {
    response.setContentType("text/html");
    ServletOutputStream out = response.getOutputStream();
    try
    {
      out.println("<html>");
      out.println("<body>");
     //4. Invoke a business method on the local interface reference.
      out.println(hello.sayHello("James Earl"));
      out.println("</body>");
      out.println("</html>");
    } catch(EJBException e) {
      out.println("EJBException error: " + e.getMessage());
    } catch(IOException e) {
      out.println("IOException error: " + e.getMessage());
    } finally {
      out.close();
    }
  }
}
```

注意： この例は、OTN-J のサイト http://otn.oracle.co.jp/sample_code/index.html の OC4J のサンプル・コードのページからダウンロードしてください。

例 2-7 リモート・クライアントとして機能する Java クライアント

次の例は、リモート・クライアントとして機能する Pure Java クライアントから実行します。リモート・クライアントは、JNDI ルックアップを使用してオブジェクトを取得する前に、JNDI プロパティを設定する必要があります。

注意： JNDI 名は、次のように、このクライアントの `application-client.xml` の `<ejb-ref>` 要素で指定されます。

```
<ejb-ref>
    <ejb-ref-name>ejb/HelloBean</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>hello.HelloHome</home>
    <remote>hello.Hello</remote>
 </ejb-ref>
```

このクライアントの `jndi.properties` ファイルは、次のとおりです。

```
java.naming.factory.initial=
        com.evermind.server.ApplicationClientInitialContextFactory
java.naming.provider.url=opmn:ormi://opmnhost:oc4j_inst1/helloworld
java.naming.security.principal=admin
java.naming.security.credentials=welcome
```

`Hello` をリモートで起動する Pure Java クライアントは、次のとおりです。

```
package hello;

import javax.ejb.*;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;
import java.io.*;
import java.util.*;
import java.rmi.RemoteException;

/*
 * A simple client for accessing an EJB.
 */

public class HelloClient
{
  public static void main(String[] args)
  {
    System.out.println("client started...");
    try {
```

```
        // Initial context properties are set in the jndi.properties file
        //1. Retrieve remote interface using a JNDI lookup*/
        Context context = new InitialContext();

       // Lookup the HelloHome object. The reference is retrieved from the
       // application-local context (java:comp/env). The variable is
       // specified in the application-client.xml).
        Object homeObject = context.lookup("java:comp/env/Helloworld");

       //2. Narrow the reference to HelloHome. Since this is a remote
       //   object, use the PortableRemoteObject.narrow method.
        HelloHome home = (HelloHome) PortableRemoteObject.narrow
                                        (homeObject, HelloHome.class);

       //3. Create the remote object and narrow the reference to Hello.
        Hello remote =
          (Hello) PortableRemoteObject.narrow(home.create(), Hello.class);

       //4. Invoke a business method on the remote interface reference.
        System.out.println(remote.sayHello("James Earl"));

      } catch(NamingException e) {
        System.err.println("NamingException: " + e.getMessage());
      } catch(RemoteException e) {
        System.err.println("RemoteException: " + e.getMessage());
      } catch(CreateException e) {
        System.err.println("FinderException: " + e.getMessage());
      }
    }
  }
```

注意： この例は、OTN-J のサイト http://otn.oracle.co.jp/sample_code/index.html の OC4J のサンプル・コードのページからダウンロードしてください。

EJB 参照情報

リモート EJB の EJB 参照情報は、クライアントの次の XML ファイルの <ejb-ref> 要素または <ejb-local-ref> 要素で指定します。

- application-client.xml: クライアントは Pure Java クライアントで、コンテナの外部で Bean を起動します。
- ejb-jar.xml: クライアントは別の EJB です。
- web.xml: クライアントはサーブレットまたは JSP です。

<ejb-ref> 要素の設定方法の詳細は、9-12 ページの「環境参照の構成」で説明されています。

たとえば、クライアントが Hello の例のリモート・インタフェースにアクセスする場合、クライアントの XML は次のように定義します。

```
<ejb-ref>
<ejb-ref-name>ejb/HelloBean</ejb-ref-name>
<ejb-ref-type>Session</ejb-ref-type>
<home>hello.HelloHome</home>
<remote>hello.Hello</remote>
</ejb-ref>
```

クライアントが Hello の例のローカル・インタフェースにアクセスする場合、クライアントの XML は次のように定義します。

```
<ejb-ref>
<ejb-ref-name>ejb/HelloBean</ejb-ref-name>
<ejb-ref-type>Session</ejb-ref-type>
<local-home>hello.HelloLocalHome</local-home>
<local>hello.HelloLocal</local>
</ejb-ref>
```

OC4J は、論理名を、クライアント・サイドの実際の JNDI 名にマッピングします。サーバー・サイドで JNDI 名を受信し、これを JNDI ツリー内で解決します。

JNDI プロパティの設定

クライアントがターゲットと一緒に置かれていて、ターゲットと同じアプリケーション内に存在している場合、またはターゲットが親の中に存在する場合、JNDI プロパティ・ファイルは必要ありません。それ以外の場合、JNDI コールの前に、jndi.properties ファイル、システム・プロパティ、または実装内で、JNDI プロパティを初期化する必要があります。次の各項で、これら 3 つのオプションについて説明します。

- JNDI プロパティなし
- JNDI プロパティ・ファイル

- 実装内の JNDI プロパティ
- OC4J スタンドアロンの JNDI プロパティ

JNDI プロパティ内で資格証明を指定する方法は、8-12 ページの「EJB クライアントの資格証明の指定」を参照してください。

注意： JNDI の使用方法の詳細は、『Oracle Application Server Containers for J2EE サービス・ガイド』の JNDI の章を参照してください。

JNDI プロパティなし ターゲット Bean と一緒に置かれているサーブレットは、そのノードの JNDI プロパティに自動的にアクセスします。したがって、JNDI プロパティは必要ないため、EJB へのアクセスは簡単です。

```
//Get the Initial Context for the JNDI lookup for a local EJB
InitialContext ic = new InitialContext();
//Retrieve the Home interface using JNDI lookup
Object helloObject = ic.lookup("java:comp/env/ejb/HelloBean");
```

また、ターゲット Bean が、同じアプリケーション内またはこのアプリケーションの親としてデプロイされたアプリケーション内に存在している場合も同様です。アプリケーションの親の設定方法については、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』を参照してください。

JNDI プロパティ・ファイル `jndi.properties` ファイル内で JNDI プロパティを設定する場合は、次のように設定します。このファイルが `CLASSPATH` からアクセス可能であることを確認してください。

ファクトリ

使用する初期コンテキスト・ファクトリについては、2-20 ページの「初期コンテキスト・ファクトリ・クラスの使用」を参照してください。

```
java.naming.factory.initial=
        com.evermind.server.ApplicationClientInitialContextFactory
```

場所

RMI ポートを含むすべてのポートは、各 OC4J インスタンスが起動すると OPMN によって動的に設定されます。クライアントの JNDI プロパティで次の URL を指定すると、クライアント側の OC4J はインスタンスの動的ポートを取得し、リストから通信用のポートを 1 つ選択します。

```
java.naming.provider.url=
  opmn:ormi://<opmn_host>:<opmn_port>:<oc4j_instance>/<application-name>
```

OPMN のホスト名とポート番号は、opmn.xml ファイルから取得されます。ほとんどの場合、OPMN は OC4J インスタンスと同じマシン上に存在します。ただし、別のマシン上に存在する場合は、ホスト名を指定する必要があります。OPMN のポート番号の指定はオプションで、指定しない場合のデフォルトはポート 6003 です。OPMN ポートは opmn.xml で指定されます。

OC4J インスタンスの名前は、Oracle Enterprise Manager で定義されています。

セキュリティ

リモート・コンテナ内の EJB にアクセスする場合、このコンテナに有効な資格証明を渡す必要があります。スタンドアロン・クライアントは、クライアントのコードとともにデプロイされた jndi.properties ファイル内で資格証明を定義します。

```
java.naming.security.principal=<username>
java.naming.security.credentials=<password>
```

実装内の JNDI プロパティ プロパティには同じ値を設定しますが、異なる構文を使用します。たとえば、コンテナ内で実行されている JavaBeans は、リモートの EJB のルックアップ用に作成された InitialContext 内で資格証明を渡します。

- java.naming.provider.url では、ロケーション文字列 "opmn:ormi" が指定されます。OPMN と OC4J は両方とも同じホスト上に存在します。OPMN のデフォルト・ポートが使用されるため、ポート番号は指定しません。
- java.naming.factory.initial では、ApplicationClientInitialContextFactory オブジェクトが使用されます。

Hashtable 環境内で JNDI プロパティを渡すには、次のようにしてこれらを設定します。

```
Hashtable env = new Hashtable();
env.put("java.naming.provider.url",
       "opmn:ormi://opmnhost:oc4j_inst1/ejbsamples");
env.put("java.naming.factory.initial",
      "com.evermind.server.ApplicationClientInitialContextFactory");
env.put(Context.SECURITY_PRINCIPAL, "guest");
env.put(Context.SECURITY_CREDENTIALS, "welcome");
Context ic = new InitialContext (env);
Object homeObject = ic.lookup("java:comp/env/ejb/HelloBean");

// Narrow the reference to a HelloHome.
HelloHome empHome =
 (HelloHome) PortableRemoteObject.narrow(homeObject,
                                                  HelloHome.class);
```

OC4J スタンドアロンの JNDI プロパティ 初期コンテキスト・ファクトリのルールは、OC4J スタンドアロン・アプリケーションのルールと同じです。ただし、OC4J スタンドアロンでは OPMN を使用しないため、ロケーション URL で opmn:ormi:// 接頭辞を使用できません。かわりに、ormi:// 接頭辞が使用されます。

ORMI のデフォルト・ポート番号は 23791 で、これは config/rmi.xml で変更可能です。したがって、次のいずれかの方法で、jndi.properties で URL を設定します。

```
java.naming.provider.url=ormi://<hostname>/<application-name>
```
または

```
java.naming.provider.url=ormi://<hostname>:23791/<application-name>\
```

リモート・コンテナ内の EJB にアクセスする場合は、このコンテナに有効な資格証明を渡す必要があります。スタンドアロン・クライアントは、クライアントのコードとともにデプロイされた jndi.properties ファイル内で資格証明を定義します。

```
java.naming.security.principal=<username>
java.naming.security.credentials=<password>
```

Bean 実装内でプロパティを設定する場合は、プロパティに同じ値を設定しますが、異なる構文を使用します。たとえば、コンテナ内で実行されている JavaBeans は、リモートの EJB のルックアップ用に作成された InitialContext 内で資格証明を渡します。

Hashtable 環境内で JNDI プロパティを渡すには、次のようにしてこれらを設定します。

```
Hashtable env = new Hashtable();
env.put("java.naming.provider.url", "ormi://myhost/ejbsamples");
env.put("java.naming.factory.initial",
       "com.evermind.server.ApplicationClientInitialContextFactory");
env.put(Context.SECURITY_PRINCIPAL, "guest");
env.put(Context.SECURITY_CREDENTIALS, "welcome");
Context ic = new InitialContext (env);
Object homeObject = ic.lookup("java:comp/env/ejb/HelloBean");

// Narrow the reference to a HelloHome.
HelloHome helloHome =
  (HelloHome) PortableRemoteObject.narrow(homeObject,
                                                HelloHome.class);
```

初期コンテキスト・ファクトリ・クラスの使用

使用する初期コンテキスト・ファクトリのタイプは、クライアントによって決まります。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。

- クライアントが OC4J コンテナの外部の Pure Java クライアントである場合は、`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・クライアントである場合は、`ApplicationInitialContextFactory` クラスを使用します。`ApplicationInitialContextFactory` クラスはデフォルトのクラスであるため、初期コンテキスト・ファクトリ・クラスを指定しないで新規の `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である場合は、`com.evermind.server.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。

たとえば、Pure Java クライアントがある場合は、初期コンテキスト・ファクトリ・クラス（`"java.naming.factory.initial"`）を `ApplicationClientInitialContextFactory` に設定します。次の例では、初期コンテキスト・ファクトリを環境内に設定していますが、これを JNDI プロパティ・ファイルに置くこともできます。

```
env.put("java.naming.factory.initial",
      "com.evermind.server.ApplicationClientInitialContextFactory");
```

クライアントが EJB、または同じアプリケーション内で EJB をコールするサーブレットの場合は、JNDI プロパティを初期コンテキスト・ファクトリに設定しないでデフォルトを使用でき、次のコードを実行して `ApplicationInitialContextFactory` オブジェクトを使用します。

```
InitialContext ic = new InitialContext();
```

`RMIInitialContextFactory` クラスを使用する場合は、XML 構成ファイルの `<ejb-ref>` で定義した論理名ではなく、ルックアップで JNDI 名を使用する必要があります。

DNS ロード・バランシング固有の初期コンテキスト・ファクトリ ロード・バランシングで DNS を使用するには、次の手順を実行する必要があります。

1. DNS 内で、1 つのホスト名を複数の IP アドレスにマッピングします。各ポート番号は、それぞれの IP アドレスに対して同じであることが必要です。DNS サーバーは、ラウンドロビン法またはランダムでアドレスを返すように設定します。

2. クライアントでの DNS のキャッシュをオフにします。UNIX マシンの場合は、次の手順で DNS のキャッシュをオフにする必要があります。
   a. クライアントでの NSCD デーモン・プロセスを停止します。
   b. `-Dsun.net.inetaddr.ttl=0` オプションを使用して、OC4J クライアントを起動します。
3. 各クライアント内で、初期コンテキスト・ファクトリを使用して初期コンテキストを作成します。プロバイダ URL では、`opmn:ormi://` または `ormi://` のいずれかの接頭辞を使用できます。Oracle Application Server アプリケーションの場合は `opmn:ormi://` 構文を、スタンドアロン OC4J アプリケーションの場合は `ormi://` を使用します。
4. `dedicated.rmicontext` プロパティを true に設定します。

DNS サーバーでルックアップが発生するたびに、DNS サーバーは、マップされている IP アドレスの 1 つを返します。

例 2-8 RMIInitialContextFactory の例

```
java.naming.factory.initial=
          com.evermind.server.rmi.RMIInitialContextFactory
java.naming.provider.url=opmn:ormi://myserver:oc4j_inst/applname
java.naming.security.principal=admin
java.naming.security.credentials=welcome
dedicated.rmicontext=true
```

別のアプリケーションの EJB へのアクセス

通常、EJB は、複数の EAR ファイル間、異なる EAR ファイルにデプロイされたアプリケーション間で通信を行うことはできません。ある EJB が、異なる EAR ファイルにデプロイされている EJB にアクセスする唯一の方法は、EJB をクライアントの親として宣言することです。子のみが親の中でメソッドを起動できます。

たとえば、`sales` および `inventory` という 2 つの EJB があり、それぞれ別の EAR ファイル内にデプロイされているとします。`sales` EJB は、`inventory` EJB を起動して十分なウィジェットが使用可能かどうかをチェックする必要があります。この 2 つの EJB は異なる EAR ファイルにデプロイされているため、`sales` EJB で `inventory` EJB を親として宣言しないかぎり、`sales` EJB は `inventory` EJB 内でメソッドを起動できません。したがって、`inventory` EJB を `sales` EJB の親として定義すると、`sales` EJB は親の中でメソッドを起動できるようになります。

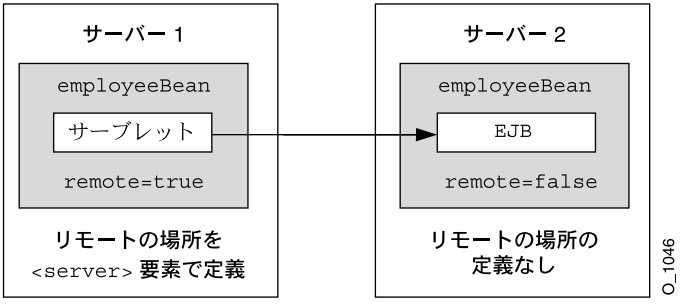
親を定義できるのは、デプロイ・ウィザードを使用してデプロイを行うときのみです。Bean の親アプリケーションの定義方法については、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』の「構成およびデプロイ」の章にある「アプリケーションのデプロイ」の項を参照してください。メソッド起動のためにクラスをパッケージ化する方法の詳細は、9-2 ページの「クラスの共有」を参照してください。

リモート・サーバーの EJB へのアクセス

あるサーバーで実行されているサーブレットが別のサーバーの EJB に接続して通信する場合は、複数層の状態になります。サーブレットと EJB は、両方とも同じアプリケーションに含まれています。アプリケーションを 2 つの異なるサーバーにデプロイすると、通常、サーブレットはローカル EJB を最初に検索します。

図 2-2 では、HelloBean アプリケーションがサーバー 1 と 2 の両方にデプロイされています。サーバー 1 のサーブレットからサーバー 2 の EJB へのコールのみを行うには、アプリケーションを両方のサーバーにデプロイする前に、アプリケーションで remote 属性を適切に設定する必要があります。

図 2-2 複数層の例



EJB モジュールの orion-application.xml における <ejb-module> 要素の remote 属性は、このアプリケーションの EJB がデプロイされているかどうかを示します。

1. サーバー 1 では、orion-application.xml ファイルの <ejb-module> 要素で remote=true を設定してから、アプリケーションをデプロイする必要があります。アプリケーション内の EJB モジュールはデプロイされません。したがって、サーブレットはローカルで EJB を検索しませんが、EJB リクエストに対してリモート・サーバーにアクセスします。
2. サーバー 2 では、orion-application.xml ファイルの <ejb-module> 要素で remote=false を設定してから、アプリケーションをデプロイする必要があります。EJB モジュールも含めて、アプリケーションは通常どおりデプロイされます。remote 属性のデフォルトは false です。したがって、remote 属性が true でないことを確認し、アプリケーションを再度デプロイします。
3. サーバー 1 の rmi.xml ファイルの <server> 要素で、リモート・サーバーであるサーバー 2 の場所を構成します。リモート・サーバーのホスト名、ポート番号、ユーザー名およびパスワードを、次のように指定します。

```
<server host=<remote_host> port=<remote_port> username=<username>
password=<password> />
```

複数のリモート・サーバーが構成されている場合、OC4J コンテナはすべてのリモート・サーバーで EJB アプリケーションを検索します。

例 2-9 リモート OC4J インスタンスの EJB にアクセスするサーブレット

次のサーブレットは、HelloBean というターゲット Bean に JNDI 名を使用します。このサーブレットは、RMIInitialContext オブジェクトで JNDI プロパティを提供します。環境は、次のようにして初期化されます。

- INITIAL_CONTEXT_FACTORY は、RMIInitialContextFactory に初期化されます。
- 新しく InitialContext が作成されるかわりに、取得されます。
- 実際の JNDI 名がルックアップで使用されます。
- リモートのロケーション URL は opmn:ormi://host:oc4j_inst/application です。OPMN のポート番号は、デフォルトが使用されるため省略されます。

```
Hashtable env = new Hashtable();
env.put(Context.PROVIDER_URL, "opmn:ormi://theirhost:oc4j_inst/myapp");
env.put(Context.SECURITY_PRINCIPAL, "admin");
env.put(Context.SECURITY_CREDENTIALS, "welcome");
env.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.evermind.server.rmi.RMIInitialContextFactory");

Context ic =
    new com.evermind.server.rmi.RMIInitialContextFactory().
         getInitialContext(env);

Object homeObject = ic.lookup("ejb/HelloBean");

// Narrow the reference to a HelloHome.
HelloHome helloHome =
      (HelloHome) PortableRemoteObject.narrow(homeObject,
                                              HelloHome.class);
```

デプロイメント・ディスクリプタの作成

クラスの実装およびコンパイルが完了した後、モジュール内のすべての Bean に対し、標準の J2EE EJB デプロイメント・ディスクリプタを作成する必要があります。XML デプロイメント・ディスクリプタ（ejb-jar.xml ファイルで定義）は、アプリケーションの EJB モジュールを記述します。Bean のタイプ、名前および属性を記述します。このファイルの構造は DTD ファイルで規定されています。http://java.sun.com/dtd/ejb-jar_2_0.dtd を参照してください。

また、構成する EJB コンテナ・サービスもデプロイメント・ディスクリプタで指定されます。データ・ソースおよび JTA の詳細は、『Oracle Application Server Containers for J2EE サービス・ガイド』を参照してください。セキュリティの詳細は、『Oracle Application Server Containers for J2EE セキュリティ・ガイド』を参照してください。

EJB アプリケーションのデプロイメント・ディスクリプタは、作成後に、EJB クラスと同じディレクトリ内に存在する META-INF に配置します。詳細は、図 2-1 を参照してください。

次の例は、リモートおよびローカルの両方のインタフェースを実装する Hello の例で必要なセクションです。

例 2-10 Hello Bean の XML デプロイメント・ディスクリプタ

次に、Hello の例でステートレス Session Bean を使用する場合のデプロイメント・ディスクリプタを示します。この例では、ローカルおよびリモートの両方のインタフェースを定義しています。必ずしも両方のタイプのインタフェースを定義する必要はなく、いずれか 1 つのみを定義することもできます。

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">

<ejb-jar>
   <display-name>hello</display-name>
   <description>
      An EJB app containing only one Stateless Session Bean
   </description>
   <enterprise-beans>
      <session>
         <description>no description</description>
         <display-name>HelloBean</display-name>
         <ejb-name>HelloBean</ejb-name>
         <home>hello.HelloHome</home>
         <remote>hello.Hello</remote>
         <local-home>hello.HelloLocalHome</local-home>
         <local>hello.HelloLocal</local>
         <ejb-class>hello.HelloBean</ejb-class>
         <session-type>Stateless</session-type>
         <transaction-type>Container</transaction-type>
```

```
      </session>
   </enterprise-beans>

   <assembly-descriptor>
      <container-transaction>
         <method>
            <ejb-name>HelloBean</ejb-name>
            <method-name>*</method-name>
         </method>
         <trans-attribute>Supports</trans-attribute>
      </container-transaction>
      <security-role>
         <role-name>users</role-name>
      </security-role>
   </assembly-descriptor>
</ejb-jar>
```

注意： この例は、OTN-J のサイト `http://otn.oracle.co.jp/sample_code/index.html` の OC4J のサンプル・コードのページからダウンロードしてください。

EJB アプリケーションのアーカイブ

実装を完了し、デプロイメント・ディスクリプタを作成した後、EJB アプリケーションを JAR ファイルにアーカイブします。JAR ファイルには、すべての EJB アプリケーション・ファイルおよびデプロイメント・ディスクリプタを含めます。

注意： このエンタープライズ Java アプリケーションの一部として Web アプリケーションが含まれている場合は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』の Web アプリケーションの構築手順を実行してください。

たとえば、`Hello` の例のコンパイル済 EJB クラス・ファイルと XML ファイルを JAR ファイルにアーカイブするには、`../hello/ejb_module` ディレクトリで次のコマンドを実行します。

```
% jar cvf helloworld-ejb.jar .
```

これにより、`ejb_module` サブディレクトリ内のすべてのファイルが JAR ファイルにアーカイブされます。

EJB アプリケーションのデプロイ準備

アプリケーションをデプロイする準備のため、次の手順を実行します。

1. エンタープライズ Java アプリケーションのモジュールを使用して、`application.xml` ファイルを修正します。
2. アプリケーションのすべての要素を EAR ファイルにアーカイブします。

これらの手順については、次の各項で説明します。

- application.XML ファイルの変更
- EAR ファイルの作成

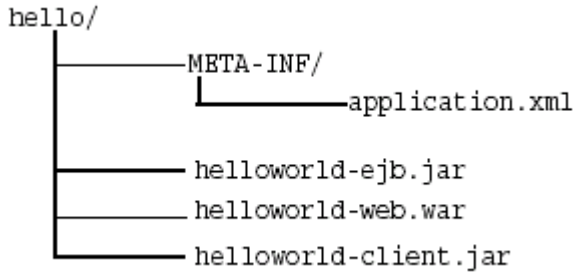
application.XML ファイルの変更

`application.xml` ファイルは、アプリケーションのマニフェスト・ファイルとして機能し、エンタープライズ・アプリケーション内に存在するモジュールのリストが含まれています。エンタープライズ・アプリケーションを構成するモジュールを指定するには、`application.xml` ファイル内の各 `<module>` 要素を使用します。各モジュールで、EJB JAR、Web WAR、またはクライアント・ファイルのうちのいずれかを記述します。別々の `<module>` 要素内で、それぞれ `<ejb>`、`<web>` および `<java>` 要素を指定します。

- `<ejb>` 要素は、EJB JAR ファイル名を指定します。
- `<web>` 要素は、`<web-uri>` 要素内で Web WAR ファイル名を、`<context>` 要素内でそのコンテキストを指定します。
- `<java>` 要素は、クライアント JAR ファイル名を（存在する場合）指定します。

図 2-3 で示すように、`application.xml` ファイルは、アプリケーションの親ディレクトリ内の `META-INF` ディレクトリ内に存在します。JAR、WAR およびクライアント JAR ファイルは、このディレクトリ内に入っている必要があります。`application.xml` ファイルは、JAR および WAR ファイルとこのように近くに存在するため、これらのファイルを参照する際、名前と相対パスを使用し、絶対ディレクトリ・パスは使用しません。これらのファイルが親ディレクトリ内のサブディレクトリに存在する場合、ファイル名に加えてこれらのサブディレクトリを指定する必要があります。

図 2-3 アーカイブ・ディレクトリの形式



たとえば、次の例では、Hello EJB アプリケーションの application.xml 内に存在する <ejb>、<web> および <java> モジュール要素を修正していますが、このアプリケーションには、EJB と対話するサーブレットも含まれています。

```
<?xml version="1.0"?>
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application
1.2//EN" "http://java.sun.com/j2ee/dtds/application_1_2.dtd">
<application>
  <display-name>helloworld j2ee application</display-name>
  <description>
     A sample J2EE application that uses a Helloworld Session Bean
     on the server and calls from java/servlet/JSP clients.
  </description>
  <module>
    <ejb>helloworld-ejb.jar</ejb>
  </module>
  <module>
    <web>
      <web-uri>helloworld-web.war</web-uri>
      <context-root>/helloworld</context-root>
    </web>
  </module>
  <module>
    <java>helloworld-client.jar</java>
  </module>
</application>
```

EAR ファイルの作成

アプリケーションの JAR、WAR および XML ファイルを含める EAR ファイルを作成します。application.xml ファイルが EAR マニフェスト・ファイルとして機能します。

helloworld.ear ファイルを作成するには、図 2-3 で示されている hello ディレクトリ内で、次のコマンドを実行します。

```
% jar cvf helloworld.ear .
```

これにより、application.xml、helloworld-ejb.jar、helloworld-web.war および helloworld-client.jar ファイルが helloworld.ear ファイルにアーカイブされます。

エンタープライズ・アプリケーションの OC4J へのデプロイ

アプリケーションを EAR ファイルにアーカイブした後、OC4J にデプロイします。アプリケーションのデプロイ方法については、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』を参照してください。

3

CMP Entity Bean

この章では、基本的な構成およびデプロイを使用した単純なコンテナ管理の永続性（CMP）EJB の開発方法を説明します。CMP Entity Bean の例（`cmpapp.jar`）は、OTN-J のサイト `http://otn.oracle.co.jp/sample_code/index.html` の OC4J のサンプル・コードのページからダウンロードしてください。

この章では、次の内容を説明します。

- Entity Bean の概要
- トランザクション要件
- Entity Bean の作成
- 主キー
- 永続フィールド
- CMP の型からデータベース型への変換

単純な Bean 管理による永続的な Entity Bean の作成方法の例は、第 6 章「BMP Entity Bean」を参照してください。EJB 間のオブジェクト関連の維持については、第 4 章「エンティティ関連（E-R）のマッピング」を参照してください。

Entity Bean の概要

EJB 2.0 およびローカル・インタフェースのサポートによって、多くの開発者は、Entity Bean を、クライアント・インタフェースとして機能する Session Bean、サーブレットまたは JSP と組み合せて使用します。Entity Bean は、機能をカプセル化し、データと依存オブジェクトを表すコースグレインな Bean です。したがって、クライアントをデータから分離できるため、データが変更されてもクライアントは影響を受けません。効率を上げるため、Session Bean、サーブレットまたは JSP を Entity Bean に連結し、ローカル・インタフェースを通じて複数の Entity Bean 間を調整できます。これは、セッション・ファサード・デザインと呼ばれます。セッション・ファサード・デザインの詳細は、Web サイト `http://java.sun.com` を参照してください。

Entity Bean はオブジェクトを集約し、コンテナを使用してトランザクション、セキュリティおよび同時実行性がサポートされた状態でデータと関連オブジェクトを効率的に維持できます。この章と以降の章では、Entity Bean の永続性機能の使用方法について説明します。

Entity Bean では、コンテナ管理の永続性（CMP）、または Bean 管理の永続性（BMP）の 2 つの方法のどちらかで永続データを管理します。これらの主な違いは、次のとおりです。

- コンテナ管理の永続性（CMB）: EJB コンテナにより、指定されたリソース（通常はデータベース）に格納することによりデータを管理します。このためには、コンテナで管理するデータをデプロイメント・ディスクリプタ内に定義する必要があります。コンテナは、データをデータベースに格納して管理します。
- Bean 管理の永続性（BMP）: Bean の実装により、データをコールバック・メソッド内で管理します。データを永続記憶域に格納するためのロジックをすべて `ejbStore` メソッド内に含め、`ejbLoad` メソッドで記憶域からリロードする必要があります。これらのメソッドは、必要に応じてコンテナから起動します。

注意： このマニュアルでは、EJB コンテナ・サービスについて説明していません。詳細は、『Oracle Application Server Containers for J2EE サービス・ガイド』の JTA、データ・ソースおよび JNDI の各章を参照してください。セキュリティについては、『Oracle Application Server Containers for J2EE セキュリティ・ガイド』を参照してください。

トランザクション要件

トランザクションには、CMP と CMR の関連を含むすべての Entity Bean を含める必要があります。たとえば、NEVER、SUPPORTS または NOT_REQUIRED などのトランザクション属性を持つ Entity Bean は定義できません。これらの属性を持つエンティティは、トランザクションの外部に置かれるためです。

Entity Bean の作成

ここでは、Entity Bean の作成手順の概要を説明します。詳細な作成手順は第 2 章「EJB 入門」を参照してください。

1. Bean のホーム・インタフェースを作成します。ホーム・インタフェースでは、`findByPrimaryKey` を含め、作成する Bean の `create` および finder メソッドを定義します。3-4 ページの「ホーム・インタフェース」を参照してください。
2. Bean のコンポーネント・インタフェースを作成します。コンポーネント・インタフェースは、クライアントによって起動可能なメソッドを宣言します。3-5 ページの「コンポーネント・インタフェース」を参照してください。
3. Bean の主キーを定義します。主キーはシリアライズ可能なクラスで、各 Entity Bean インスタンスを識別します。単純なデータ型クラス（`java.lang.String` など）を使用したり、複合クラス（主キーのコンポーネントとして複数のオブジェクトを持つクラスなど）を定義できます。3-9 ページの「主キー」を参照してください。
4. Bean を実装します。3-6 ページの「Entity Bean クラス」を参照してください。
5. Bean のデプロイメント・ディスクリプタを作成します。デプロイメント・ディスクリプタにより、XML 要素を通じて Bean のプロパティを指定します。このステップで、コンテナによって管理する Bean 内のデータを指定します。永続フィールドの詳細は、3-13 ページの「永続フィールド」を参照してください。このフィールドで他のオブジェクトとの関連を記述する場合は、第 4 章「エンティティ関連（E-R）のマッピング」を参照してください。

   また、構成する EJB コンテナ・サービスもデプロイメント・ディスクリプタで指定されます。データ・ソースおよび JTA の詳細は、『Oracle Application Server Containers for J2EE サービス・ガイド』を参照してください。セキュリティの詳細は、『Oracle Application Server Containers for J2EE セキュリティ・ガイド』を参照してください。

   永続データをデータベースに格納またはリストアする際にコンテナのデフォルトを使用しない場合、Bean に対して正しい表が存在することを確認する必要があります。デフォルトを使用する場合は、コンテナにより、デプロイメント・ディスクリプタおよびデータソース情報に基づいてデータ用の表と列が作成されます。
6. Bean、コンポーネント・インタフェース、ホーム・インタフェースおよびデプロイメント・ディスクリプタを含める EJB JAR ファイルを作成します。作成した後、`application.xml` ファイルを構成し、EAR ファイルを作成し、EJB を OC4J にデプロイします。

次の項で、単純な CMP Entity Bean を説明します。この例では、わかりやすいように、他の章と同様に引き続き employee の例を使用します。

- ホーム・インタフェース
- コンポーネント・インタフェース
- Entity Bean クラス

ホーム・インタフェース

ホーム・インタフェースは主に Bean 参照を取得するために使用され、この参照に対してクライアントはビジネス・メソッドをリクエストできます。

- ローカル・ホーム・インタフェースは、`javax.ejb.EJBLocalHome` を拡張します。
- リモート・ホーム・インタフェースは、`javax.ejb.EJBHome` を拡張します。

ホーム・インタフェースには、クライアントが Bean のインスタンスを作成するために起動する `create` メソッドが含まれている必要があります。各 `create` メソッドには異なるシグネチャを使用できます。Entity Bean の場合、`findByPrimaryKey` メソッドを開発する必要があります。オプションで、Bean 用に他の finder メソッドも開発可能です。これらは、`find<`*name*`>` のように名前を付けます。

メソッドの作成および取得に加えて、ホーム・インタフェース内でホーム・インタフェースのビジネス・メソッドを提供できます。このメソッドの機能では、特定のエンティティ・オブジェクトのデータにアクセスできません。このメソッドは、単一の Entity Bean インスタンスに関連がない情報を取得するために使用します。クライアントがホーム・インタフェースの任意のビジネス・メソッドを起動すると、Entity Bean はプールから移動され、リクエストを処理します。したがって、このメソッドを使用すると、Bean に関連する一般的な情報に関する操作を実行できます。

employee の例では、ローカル・ホーム・インタフェースに `create`、`findByPrimaryKey`、`findAll` および `calcSalary` メソッドが使用されています。`calcSalary` メソッドは、全従業員の給与合計を計算する、ホーム・インタフェースのビジネス・メソッドです。このメソッドは特定の従業員の情報にはアクセスしませんが、全従業員のデータベースに対して SQL 問合せを実行します。

例 3-1 Entity Bean Employee のホーム・インタフェース

employee のホーム・インタフェースは、コンポーネント・インタフェースを作成するメソッドを提供します。このホーム・インタフェースは、2 つの finder メソッドも提供します。1 つは従業員番号によって特定の従業員を検索し、もう 1 つは全従業員を検索します。また、全従業員にかかるコストを計算する、ホーム・インタフェースのビジネス・メソッドの `calcSalary` も提供します。

ホーム・インタフェースは、`javax.ejb.EJBHome` を拡張する必要があり、`create` および `findByPrimaryKey` メソッドを定義します。

```
package employee;

import javax.ejb.*;
import java.rmi.*;

public interface EmployeeLocalHome extends EJBLocalHome
{

  public EmployeeLocal create(Integer empNo) throws CreateException;

  // Find an existing employee
  public EmployeeLocal findByPrimaryKey (Integer empNo) throws FinderException;

  //Find all employees
  public Collection findAll() throws FinderException;

  //Calculate the Salaries of all employees
  public float calcSalary() throws Exception;
}
```

コンポーネント・インタフェース

Entity Bean のコンポーネント・インタフェースは、顧客が参照してメソッドを起動するインタフェースです。コンポーネント・インタフェースは、Entity Bean インスタンスのビジネス・ロジック・メソッドを定義します。

- ローカル・コンポーネント・インタフェースは、`javax.ejb.EJBLocalObject` を拡張します。
- リモート・コンポーネント・インタフェースは、`javax.ejb.EJBObject` を拡張します。

employee Entity Bean の例では、従業員情報の取得と更新を行うメソッドが含まれたローカル・コンポーネント・インタフェースが公開されています。

```
package employee;

import javax.ejb.*;

public interface EmployeeLocal extends EJBLocalObject
{
  public Integer getEmpNo();
  public void setEmpNo(Integer empNo);

  public String getEmpName();
  public void setEmpName(String empName);
```

```
  public Float getSalary();
  public void setSalary(Float salary);
}
```

Entity Bean クラス

Entity Bean クラスでは、次のメソッドを実装します。

- ホーム・インタフェースで宣言されているメソッドのターゲット・メソッド。次のメソッドが含まれます。
  - ホーム・インタフェースで定義された対応する `create` メソッドのパラメータに一致するパラメータを持つ `ejbCreate` および `ejbPostCreate` メソッド。
  - ホーム・インタフェースで定義された Finder メソッド（`ejbFindByPrimaryKey` および `ejbFindAll` 以外）。コンテナによって、`ejbFindByPrimaryKey` および `ejbFindAll` メソッド実装が生成されます。ただし、それぞれの実装に対して空のメソッドを提供する必要があります。
  - ホーム・インタフェースの任意のビジネス・メソッド（Bean 実装では `ejbHome` と前に付加されています）。たとえば、`calcSalary` メソッドは、`ejbHomeCalcSalary` メソッド内に実装されています。
- コンポーネント・インタフェースで宣言されたビジネス・ロジック・メソッド。
- `javax.ejb.EntityBean` インタフェースから継承されているメソッド。

ただし、コンテナ管理の永続性を使用する場合は、コンテナによってほとんどのターゲット・メソッドおよびデータ・オブジェクトが管理されるため、実際に行う実装はほとんどありません。

```
package employee;

import javax.ejb.*;
import java.rmi.*;

public abstract class EmployeeBean implements EntityBean
{

  private EntityContext ctx;

  // Each CMP field has a get and set method as accessors
  public abstract Integer getEmpNo();
  public abstract void setEmpNo(Integer empNo);

  public abstract String getEmpName();
  public abstract void setEmpName(String empName);

  public abstract Float getSalary();
```

```
public abstract void setSalary(Float salary);

public void EmployeeBean()
{
  // Constructor. Do not initialize anything in this method.
  // All initialization should be performed in the ejbCreate method.
  // The passivate() method may destroy these attributes when pooling
}

public float ejbHomeCalcSalary() throws Exception
{
  Collection c = null;
  try {
     c = ((EmployeeLocalHome)this.ctx.getEJBLocalHome()).findAll();

     Iterator i = c.iterator();
     float totalSalary = 0;
     while (i.hasNext())
     {
       EmployeeLocal e = (EmployeeLocal)i.next();
       totalSalary = totalSalary + e.getSalary().floatValue();
     }
     return totalSalary;
  }
  catch (FinderException e) {
    System.out.println("Got finder Exception "+e.getMessage());
    throw new Exception(e.getMessage());
  }
}

public EmployeePK ejbCreate(Integer empNo, String empName, Float salary)
  throws CreateException
{
  setEmpNo(empNo);
  setEmpName(empName);
  setSalary(salary);
  return new EmployeePK(empNo);
}

public void ejbPostCreate(Integer empNo, String empName, Float salary)
  throws CreateException
{
  // Called just after bean created; container takes care of implementation
}

public void ejbStore()
{
```

```
    // Called when bean persisted; container takes care of implementation
  }

  public void ejbLoad()
  {
     // Called when bean loaded; container takes care of implementation
  }

  public void ejbRemove() throws RemoveException
  {
     // Called when bean removed; container takes care of implementation
  }

  public void ejbActivate()
  {
    // Called when bean activated; container takes care of implementation.
    // If you need resources, retrieve them here.
  }

  public void ejbPassivate()
  {
    // Called when bean deactivated; container takes care of implementation.
    // if you set resources in ejbActivate, remove them here.
  }

  public void setEntityContext(EntityContext ctx)
  {
    this.ctx = ctx;
  }

  public void unsetEntityContext()
  {
    this.ctx = null;
  }
}
```

注意： CMP Entity Bean の例全体（cmpapp.jar）は、OTN-J のサイト http://otn.oracle.co.jp/sample_code/index.html の OC4J のサンプル・コードのページから入手できます。

主キー

各 Entity Bean には、他のインスタンスから一意に識別するための主キーが存在します。主キー（または主キーとなる複合キー内のフィールド）を、デプロイメント・ディスクリプタのコンテナ管理による永続的フィールドとして宣言する必要があります。主キー内のすべてのフィールドは、プリミティブ型、シリアライズ可能な型、または SQL 型にマッピング可能な型に制限されています。主キーは、次のいずれかの方法で定義します。

- 主キーに、一般的な型を定義します。型は、デプロイメント・ディスクリプタの <prim-key-class> で定義されます。永続的な主キーとして識別されるデータ・フィールドは、デプロイメント・ディスクリプタの <primkey-field> 要素で識別されます。Bean クラス内で宣言される主キー変数は、public として宣言する必要があります。
- 主キーの型を、シリアライズ可能な <*name*>PK クラス内のシリアライズ可能なオブジェクトとして定義します。このクラスは、デプロイメント・ディスクリプタの <prim-key-class> で宣言されます。これは、主キーの高度な定義方法です。詳細は 3-10 ページの「クラス内での主キーの定義」で説明します。
- 自動生成された主キーを指定します。java.lang.Object を <prim-key-class> に主キーとして指定し、主キー名を <primkey-field> に指定しなかった場合、その主キーはコンテナによって自動生成されます。詳細は、3-12 ページの「自動生成された主キーの定義」を参照してください。

単純な CMP の場合、デプロイメント・ディスクリプタ内の主キーのデータ型を定義することにより、主キーに一般的なデータ型を定義できます。

employee の例では、主キーを java.lang.Integer として定義し、従業員番号（empNo）を主キーとして使用しています。

```
<enterprise-beans>
      <entity>
         <display-name>Employee</display-name>
         <ejb-name>EmployeeBean</ejb-name>
         <local-home>employee.EmployeeLocalHome</local-home>
         <local>employee.EmployeeLocal</local>
         <ejb-class>employee.EmployeeBean</ejb-class>
         <persistence-type>Container</persistence-type>
         <prim-key-class>java.lang.Integer</prim-key-class>
         <reentrant>False</reentrant>
         <cmp-version>2.x</cmp-version>
         <abstract-schema-name>Employee</abstract-schema-name>
         <cmp-field><field-name>empNo</field-name></cmp-field>
         <cmp-field><field-name>empName</field-name></cmp-field>
         <cmp-field><field-name>salary</field-name></cmp-field>
        <primkey-field>empNo</primkey-field>
      </entity>
...
</enterprise-beans>
```

定義した後、コンテナは Entity Bean 表に主キー用に 1 つの列を作成し、デプロイメント・ディスクリプタで定義した主キーをこの列にマッピングします。

注意： CMP Entity Bean の例全体（cmpapp.jar）は、OTN-J のサイト http://otn.oracle.co.jp/sample_code/index.html の OC4J のサンプル・コードのページから入手できます。

---

orion-ejb-jar.xml ファイル内で、ejb-jar.xml ファイルで定義された CMP フィールドまたは主キー・フィールドをデータベース列名にマップすることによって、主キーが基礎となるデータベース永続記憶域にマップされます。次の orion-ejb-jar.xml のコード例では、EmpBean 永続記憶域がデータベース内で EMP 表として定義されています。この表は jdbc/OracleDS データ・ソースで定義されています。<entity-deployment> 要素定義に従って、主キーの empNo は Emp 表の EMPNO 列にマップされ、empName および salary CMP フィールドは、それぞれ EMP 表の EMPNAME および SALARY 列にマップされます。

```
<entity-deployment name="EmpBean" ...table="EMP"
                 data-source="jdbc/OracleDS"...   >
 <primkey-mapping>
  <cmp-field-mapping name="empNo" persistence-name="EMPNO" />
 </primkey-mapping>
 <cmp-field-mapping name="empName" persistence-name="EMPNAME" />
 <cmp-field-mapping name="salary" persistence-name="SALARY" />
```

クラス内での主キーの定義

主キーが単純なデータ型ではなく複合キーの場合、主キーは、シリアライズ可能なクラスで、名前を <*name*>PK にする必要があります。主キー・クラスは、デプロイメント・ディスクリプタの <prim-key-class> で定義します。

主キー変数は、次の規則に従う必要があります。

- デプロイメント・ディスクリプタの <cmp-field><field-name> 要素内で定義されていること。これにより、コンテナから主キー・フィールドを管理できるようになります。
- Bean クラス内で public として宣言され、プリミティブ型、シリアライズ可能、または SQL 型にマッピング可能な型のいずれかに制限されていること。
- 主キーを構成する変数の名前は、<cmp-field><field-name> 要素と主キー・クラスの両方で同一であること。

主キー・クラス内で、主キーのインスタンスを作成するコンストラクタを実装します。主キー・クラスがこのように定義されると、コンテナによってこのクラスが管理されます。

次の例では、主キー・クラス内に従業員番号が配置されています。

```
package employee;

public class EmployeePK implements java.io.Serializable
{
  public Integer empNo;

  public EmployeePK()
  {
    this.empNo = null;
  }

  public EmployeePK(Integer empNo)
  {
    this.empNo = empNo;
  }
}
```

主キー・クラスは、<prim-key-class> 要素内で宣言され、その変数はそれぞれ XML デプロイメント・ディスクリプタ内の <cmp-field><field-name> 要素内で次のように宣言されます。

```
<enterprise-beans>
      <entity>
         <description>no description</description>
         <display-name>EmployeeBean</display-name>
         <ejb-name>EmployeeBean</ejb-name>
         <local-home>employee.LocalEmployeeHome</home>
         <local>employee.LocalEmployee</remote>
         <ejb-class>employee.EmployeeBean</ejb-class>
         <persistence-type>Container</persistence-type>
        <prim-key-class>employee.EmployeePK</prim-key-class>
         <reentrant>False</reentrant>
         <cmp-version>2.x</cmp-version>
         <abstract-schema-name>Employee</abstract-schema-name>
        <cmp-field><field-name>empNo</field-name></cmp-field>
         <cmp-field><field-name>empName</field-name></cmp-field>
         <cmp-field><field-name>salary</field-name></cmp-field>
      </entity>
</enterprise-beans>
```

定義した後、コンテナは、Entity Bean 表に主キー用に 1 つの列を作成し、デプロイメント・ディスクリプタで定義した主キー・クラスをこの列にマッピングします。

CMP フィールドは、3-9 ページの「主キー」で説明されているのと同じ方法で、orion-ejb-jar.xml でマップされます。複合主キーを使用すると、マッピングに複数のフィールドが含まれるため、<primkey-mapping> 要素の <cmp-field-mapping> 要素に

は別のサブ要素である <fields> 要素が含まれます。次に示すように、主キーのすべてのフィールドは、<cmp-field-mapping> 要素内の <fields> 要素で個別に定義されます。

```
<primkey-mapping>
 <cmp-field-mapping>
   <fields>
    <cmp-field-mapping name="empNo" persistence-name="EMPNO" />
   </fields>
 </cmp-field-mapping>
</primkey-mapping>
```

外部キーを含む複合主キーを使用する場合は、特別なマッピングが必要です。詳細は、4-56 ページの「コンポジット主キーでの外部キーの使用」を参照してください。

自動生成された主キーの定義

java.lang.Object を <prim-key-class> に主キーとして指定し、主キー名を <primkey-field> に指定しなかった場合、その主キーはコンテナによって自動生成されます。

employee の例では、主キーを java.lang.Object として定義します。したがって、コンテナは主キーを自動生成します。

```
<enterprise-beans>
      <entity>
         <display-name>Employee</display-name>
         <ejb-name>EmployeeBean</ejb-name>
         <local-home>employee.EmployeeLocalHome</local-home>
         <local>employee.EmployeeLocal</local>
         <ejb-class>employee.EmployeeBean</ejb-class>
         <persistence-type>Container</persistence-type>
         <prim-key-class>java.lang.Object</prim-key-class>
         <reentrant>False</reentrant>
         <cmp-version>2.x</cmp-version>
         <abstract-schema-name>Employee</abstract-schema-name>
         <cmp-field><field-name>empNo</field-name></cmp-field>
         <cmp-field><field-name>empName</field-name></cmp-field>
         <cmp-field><field-name>salary</field-name></cmp-field>
      </entity>
...
</enterprise-beans>
```

定義した後、コンテナは、Entity Bean 表に autoid という LONG 型の主キーの列を作成します。コンテナは、主キーの値に対してランダムな数値を使用します。これは、次に示すように、Bean の orion-ejb-jar.xml で生成されます。

```
<primkey-mapping>
  <cmp-field-mapping name="auto_id"
                                        persistence-name="autoid"/>
</primkey-mapping>
```

永続フィールド

CMP Bean 内の永続データは、次のいずれかです。

- 永続フィールド : データベース表に維持される単純なデータ型。このフィールドは、Bean の直接的な属性です。
- 関連フィールド : 別の Bean との関連。

それぞれの型には、構成方法について独自の複合ルールがあります。この項では、永続フィールドについて説明します。関連フィールドの詳細は、第 4 章「エンティティ関連（E-R）のマッピング」を参照してください。

CMP Entity Bean の場合、永続データを Bean のインスタンスおよびデプロイメント・ディスクリプタの両方で定義します。

- Bean インスタンスの get/set メソッド : 永続フィールドと関連フィールドのそれぞれについて、get メソッドと set メソッドの両方を作成します。永続フィールドの場合は、get メソッドから返され set メソッドに渡されるパラメータのデータ型によって、フィールドの単純なデータ型が定義されます。フィールドの名前は、get メソッドと set メソッドの名前によって指定されます。

  次の XML は、従業員名の永続フィールドに対する get メソッドと set メソッドを示します。`String` が、get メソッドから返され、set メソッドに渡されます。したがって、`String` はフィールドの単純なデータ型です。メソッド名から「get」および「set」を削除し、最初の文字を小文字にした名前が永続フィールド名になります。この例では、`empName` が永続フィールド名です。

  ```
  public abstract String getEmpName() throws RemoteException;
  public abstract void setEmpName(String empName) throws RemoteException;
  ```

- デプロイメント・ディスクリプタにより、これらのフィールドが永続的であると定義されます。各フィールド名は、EJB デプロイメント・ディスクリプタの `<cmp-field><field-name>` 要素に定義する必要があります。employee の例では、データ・アクセッサ・メソッドに `empNo`、`empName` および `salary` の 3 つの永続データ・フィールドが定義されます。

  これらのフィールドは、次のように、`<cmp-field><field-name>` 要素内の `ejb-jar.xml` デプロイメント・ディスクリプタで永続フィールドとして定義されています。

```
<enterprise-beans>
      <entity>
         <display-name>Employee</display-name>
         <ejb-name>EmployeeBean</ejb-name>
         <local-home>employee.EmployeeLocalHome</local-home>
         <local>employee.EmployeeLocal</local>
         <ejb-class>employee.EmployeeBean</ejb-class>
         <persistence-type>Container</persistence-type>
         <prim-key-class>java.lang.Integer</prim-key-class>
         <reentrant>False</reentrant>
         <cmp-version>2.x</cmp-version>
         <abstract-schema-name>Employee</abstract-schema-name>
         <cmp-field><field-name>empNo</field-name></cmp-field>
         <cmp-field><field-name>empName</field-name></cmp-field>
         <cmp-field><field-name>salary</field-name></cmp-field>
         <primkey-field>empNo</primkey-field>
      </entity>
...
</enterprise-beans>
```

データベースにマッピングされるこれらのフィールドに対して、次のいずれかの操作を実行できます。

- これらのフィールドのデフォルトを使用し、追加のデプロイメント・ディスクリプタは構成しません。デフォルトのマッピング方法については、3-15 ページの「永続フィールドのデータベースへのデフォルト・マッピング」を参照してください。
- 指定したデータベースに存在する表内の列に永続データ・フィールドをマッピングします。永続データのマッピングは、`orion-ejb-jar.xml` ファイル内に構成されます。詳細は、3-16 ページの「永続フィールドのデータベースへの明示的なマッピング」を参照してください。

注意： CMP Entity Bean の例全体（`cmpapp.jar`）は、OTN-J のサイト `http://otn.oracle.co.jp/sample_code/index.html` の OC4J のサンプル・コードのページから入手できます。

永続フィールドのデータベースへのデフォルト・マッピング

永続フィールドを ejb-jar.xml ファイルに定義すると、OC4J では、これらのフィールドのデータベースへのマッピングを次のように提供します。

- データベース : 使用中の OC4J インスタンス構成に設定されているデフォルトのデータベース。JNDI 名については、エミュレートされたデータ・ソースの場合は <location> 要素を、エミュレートされていないデータ・ソースの場合は <ejb-location> 要素を使用します。

  インストール後の状態では、デフォルトのデータベースは、ローカルにインストールされた Oracle データベースで、ポート 1521 でリスニングし、SID が ORCL である必要があります。デフォルトのデータベースをカスタマイズするには、最初に、構成済データベースを、使用するデータベースに変更します。

  > **注意 :** データ・ソース・オブジェクトの構成方法の詳細は、『Oracle Application Server Containers for J2EE サービス・ガイド』のデータ・ソースの章を参照してください。

- 表 : 表名の一意性を保証したデフォルト表がコンテナによって自動的に作成されます。再度デプロイする場合に備えて、この表名で生成された orion-ejb-jar.xml ファイルを ejb-jar.xml ファイルと同じディレクトリにコピーします。これによって、再度デプロイするとき、最初に生成された表名と同じ名前を使用できます。このファイルをコピーしないと、異なる表名が生成される場合があります。

  表名は、次の名前をアンダースコア（_）で区切って付けられます。

  – EJB 名 : デプロイメント・ディスクリプタの <ejb-name> に定義されています。

  – JAR ファイル名 : .jar 拡張子を含みます。ただし、SQL 表記規則に従って、ダッシュ（-）とピリオド（.）はすべてアンダースコア（_）に変換されます。たとえば、JAR ファイル名が employee.jar の場合は、employee_jar に変換されて表名に含まれます。

  – アプリケーション名 : デプロイ時に定義するアプリケーション名です。

  構成された名前が 31 文字以上の場合、その名前は 24 文字で切り捨てられます。これに、6 文字の英数字のハッシュ・コードが追加されます。

  たとえば、EJB 名が EmpBean、JAR ファイル名が empl.jar、アプリケーション名が employee の場合、デフォルトの表名は EmpBean_empl_jar_employee になります。

- 列名 : Entity Bean 表内の列。各列の名前は、指定されたデータベースの <cmp-field> 要素と同じ名前です。データベースのデータ型は、特定のデータベースの XML ファイル（oracle.xml など）で定義されています。Java のデータ型はデータベースのデータ型に変換されます。

注意： 次に示すように、VARCHAR の永続型マッピングが orion-ejb-jar.xml で定義されていないと、サード・パーティ・データベースでは、特定のデータ型について表を自動作成できない場合があります。

```
<cmp-field-mapping name="../"
                   persistence-name="..."
                   persistence-type="varchar(10)" />
```

永続フィールドのデータベースへの明示的なマッピング

3-15 ページの「永続フィールドのデータベースへのデフォルト・マッピング」で説明したように、永続データは、コンテナによって自動的にデータベース表にマッピング可能です。ただし、Bean が示すデータが比較的複雑な場合や OC4J のデフォルトを使用しない場合は、永続データを、既存のデータベース表と orion-ejb-jar.xml ファイル内の該当する列にマッピングできます。フィールドをいったんマッピングすると、コンテナによって、永続データの永続記憶域が、指定された表と行に用意されます。

明示的なマッピングの場合は、次の操作をお薦めします。

1. ejb-jar.xml 要素のみを構成してアプリケーションをデプロイします。

   OC4J は、orion-ejb-jar.xml ファイルを作成してデフォルトのマッピングを行います。フィールドは最初から作成するより、変更する方が簡単です。これによって、ここで説明している変更のすべてを行うか、一部の変更を行うかを選択できます。

2. 指定したデータベース表と列を使用するため、orion-ejb-jar.xml ファイルの <entity-deployment> 要素を変更します。

永続フィールドをそれぞれの <cmp-field> 要素内で定義すると、各永続フィールドを特定のデータベース表と列にマッピングできます。これによって、CMP フィールドを既存のデータベース表にマッピングできます。マッピングは、OC4J 固有のデプロイメント・ディスクリプタの orion-ejb-jar.xml を使用して行います。

CMP フィールドの明示的なマッピングは、<entity-deployment> 要素内で完了します。この要素には、Entity Bean に関するすべてのマッピングが含まれています。ただし、CMP フィールドのマッピングに固有の属性と要素は、次のとおりです。

```
<entity-deployment name="..." location="..."
   table="..." data-source="...">
   <primkey-mapping>
      <cmp-field-mapping name="..." persistence-name="..." />
   </primkey-mapping>
   <cmp-field-mapping name="..." persistence-name="..." />
...
</entity-deployment>
```

要素名または属性名	説明
name	Bean 名。<ejb-name> 要素の ejb-jar.xml ファイルに定義されています。
location	JNDI の場所。
table	データベース表の名前。
data-source	表が存在するデータベースのデータ・ソース。
primkey-mapping	主キーを表にマッピングする方法の定義。
cmp-field-mapping	<cmp-field> は、name 属性によってデプロイメント・ディスクリプタに指定され、persistence-name 属性で表の列にマッピングされます。

orion-ejb-jar.xml ファイルで、次のものを構成できます。

1. マッピングされる CMP フィールドを持つ各 Entity Bean について、<entity-deployment> 要素を構成します。
2. Bean 内の、マッピングされる各フィールドについて、<cmp-field-mapping> 要素を構成します。各 <cmp-field-mapping> 要素は、永続的にするフィールドの名前を含んでいる必要があります。
   a. <cmp-field-mapping> 要素内に存在する <primkey-mapping> 要素の主キーを構成します。
   b. 1 つの <cmp-field-mapping> 要素内の 1 つのフィールドにマッピングされる単純なデータ型（プリミティブ、単純なオブジェクトまたはシリアライズ可能なオブジェクト）を構成します。名前およびデータベース・フィールドは、要素属性で完全に定義されています。

例 3-2 永続フィールドの特定データベース表へのマッピング

次の例では、employee 永続データ・フィールドを Oracle データベース表 EMP にマッピングすることにより、Bean インスタンス内の永続データ・フィールドをデータベース表および列にマッピングする方法を示します。

- Bean は、<entity-deployment> の name 属性で指定されています。この Bean の JNDI 名は、location 属性で定義されています。
- データベース表の名前は、table 属性で定義されています。また、データベースは、data-source 属性で指定されています。これは、data-sources.xml ファイルで定義されている DataSource の <ejb-location> の名前と同じです。

- Bean の主キー empNo は、<primkey-mapping> 要素内で、データベース表の列である EMPNO にマッピングされています。
- Bean の永続データ・フィールドである empName および salary は、<cmp-field-mapping> 要素内のデータベース表の列である ENAME および SAL にマッピングされています。

```
<entity-deployment name="EmpBean" location="emp/EmpBean"
   wrapper="EmpHome_EntityHomeWrapper2" max-tx-retries="3"
   table="emp" data-source="jdbc/OracleDS">
   <primkey-mapping>
      <cmp-field-mapping name="empNo" persistence-name="empno" />
   </primkey-mapping>
   <cmp-field-mapping name="empName" persistence-name="ename" />
   <cmp-field-mapping name="salary" persistence-name="sal" />
  ...
</entity-deployment>
```

デプロイ後、OC4J は、要素値を次のようにマッピングします。

Bean	データベース
emp/EmpBean	data-sources.xml ファイルの jdbc/OracleDS に存在する EMP 表
empNo	主キーである EMPNO 列
empName	ENAME 列
salary	SAL 列

CMP の型からデータベース型への変換

主キーの型および <cmp-field> でコンテナ管理の永続フィールドを定義するときは、シリアライズ可能な Java ユーザー・クラスおよび単純なデータ型を定義できます。

- 単純なデータ型
- シリアライズ可能クラス
- 他の Entity Bean または Collections

単純なデータ型

次の表は、persistence-type 属性で提供できる単純なデータ型、およびそのデータ型の SQL 型と Oracle データベース型へのマッピングを示します。Oracle 以外のデータベースでは、これらのマッピングが機能することは保証されていません。

表 3-1 単純なデータ型

一般的な型（システム固有）	SQL 型	Oracle 型
java.lang.String	VARCHAR(255)	VARCHAR2(255)
java.lang.Integer[ ]	INTEGER	NUMBER(20,0)
java.lang.Long[ ]	INTEGER	NUMBER(20,0)
java.lang.Short[ ]	INTEGER	NUMBER(10,0)
java.lang.Double[ ]	DOUBLE PRECISION	NUMBER(30,0)
java.lang.Float[ ]	FLOAT	NUMBER(20,5)
java.lang.Byte[ ]	SMALLINT	NUMBER(10,0)
java.lang.Character[ ]	CHAR	CHAR(1)
java.lang.Boolean[ ]	BIT	NUMBER(1,0)
java.util.Date	DATETIME	DATE
java.sql.Date	DATE	DATE
java.util.Time	DATE	DATE
java.sql.Timestamp	TIMESTAMP	TIMESTAMP
java.lang.String	CLOB	CLOB
char[ ]	CLOB	CLOB

一般的な型（システム固有）	SQL 型	Oracle 型
byte[ ]	BLOB	BLOB
java.io.Serializable（4KB に制限）	LONGVARBINARY	BLOB

注意： これらのデータ型のマッピングは、XML 構成ファイル config/database-schema/<db>.xml で変更できます。

DATE には時間が含まれるため、Date および Time はデータベースの DATE にマップされます。ただし、Timestamp はデータベースの TIMESTAMP にマップされ、時間をナノ秒で提供します。

java.sql.CLOB および java.sql.BLOB の直接マッピングは、これらのオブジェクトがシリアライズできないため、現在はサポートされていません。ただし、String または char[] と byte[] は、データベース列の CLOB 型と BLOB 型にそれぞれマップできます。char[] から CLOB、または byte[] から BLOB へのマッピングは、Oracle データベースでのみ実行できます。この操作を処理するため、Oracle JDBC API が変更されています。

JDBC Thin ドライバで、シリアライズされたオブジェクトを BLOB 型にマップする場合は、4KB に制限されます。

String および char[] 変数をデータベースの VARCHAR2 にマップする場合、保持できるのは最大 2KB です。ただし、2KB を超える String オブジェクトまたは char[] は、次のように CLOB 型にマップできます。

1. Bean 実装では、String または char[] オブジェクトを使用します。
2. <cmp-field-mapping> 要素の persistence-type 属性で、次のようにオブジェクトを CLOB 型に定義します。

```
<cmp-field-mapping name="stringdata" persistence-name="stringdata"
            persistence-type="CLOB" />
```

同じ方法で、次のように Bean 実装の byte[] を BLOB 型にマップできます。

```
<cmp-field-mapping name="bytedata" persistence-name="bytedata"
           persistence-type="BLOB" />
```

シリアライズ可能クラス

単純なデータ型以外に、Serializable を実装するユーザー・クラスを定義できます。このクラスは、データベース内の BLOB に格納されます。

他の Entity Bean または Collections

他の Entity Bean または Collections を CMP の型として定義しないでください。かわりに、これらは関連であるため、CMR フィールド内で定義する必要があります。

- 別の Entity Bean との関連は、常に <cmr-field> 関連で定義されます。
- Collections は、「多」関連を形成し、<cmr-field> 関連内で定義する必要があります。その他の型（Lists など）は、Collections のサブインタフェースです。Collections の使用をお薦めします。

4

エンティティ関連（E-R）のマッピング

この章では、エンティティ間の関連を開発する方法を説明します。開発者は、次のいずれかの観点から E-R にアプローチできます。

- EJB 開発 : UML 図を使用して、Entity Bean および各 Bean 間の関連のカーディナリティと方向を、EJB オブジェクトの観点から設計します。
- データベース開発 : ERD 図を使用し、主キーと外部キーによってカーディナリティと方向を指定して、Entity Bean をサポートするデータベース表を設計します。ここでは、データベースを各 Entity Bean にマッピングする方法、および Entity Bean 間の関連について説明します。

この章では、最初に、EJB 開発の観点から E-R を説明します。次に、デプロイメント・ディスクリプタをデータベース表にマッピングする方法について説明します。データベース開発の観点から設計を行う場合は、4-12 ページの「データベースへのオブジェクト関連フィールドのマッピング」に進んでください。

注意： オブジェクト関連の Entity Bean の例（`ormapdemo.jar`）は、OTN-J のサイト `http://otn.oracle.co.jp/sample_code/index.html` の OC4J のサンプル・コードのページから入手できます。

この章には、次の項目が含まれます。

トランザクション要件

トランザクションには、CMP と CMR の関連を含むすべての Entity Bean を含める必要があります。たとえば、NEVER、SUPPORTS または NOT_REQUIRED などのトランザクション属性を持つ Entity Bean は定義できません。これらの属性を持つエンティティは、トランザクションの外部に置かれるためです。

エンティティ間の関連の定義

次の各項では、Entity Bean の関連とその定義方法について説明します。

カーディナリティと方向の選択

カーディナリティとは、関連のそれぞれの側にあるエンティティ・オブジェクトの数です。EJB 間では、次の種類の関連を定義できます。

- 1 対 1
- 1 対多または多対 1（方向による）
- 多対多

さらに、各関連は単方向または双方向のいずれかです。たとえば、従業員から住所への関連は単方向です。従業員情報から住所を取得できます。ただし、住所から従業員情報は取得できません。また、従業員とプロジェクトの関連は双方向です。プロジェクト番号を指定すると、そのプロジェクトに従事している従業員を取得できます。そして、従業員番号を指定すると、その従業員が従事しているすべてのプロジェクトを取得できます。したがって、関連は両方向で有効です。

ターゲットを複数のエンティティから再利用する場合は、通常、単方向の関連を使用します。たとえば、夫婦が 2 人とも同じ会社に勤務しているとします。2 人の従業員レコードは、単方向の関連で同じ自宅電話番号を指します。双方向の関連では、このような状況は生じません。

2 つの Bean 間の関連についてのカーディナリティと方向は、デプロイメント・ディスクリプタで定義します。

1 対 1 関連の概要

1 対 1 関連は、2 つの Bean 間における最も単純な関連です。1 つの Entity Bean が別の 1 つの Entity Bean にのみ関連付けられています。たとえば、会社のオフィスがすべて個室に区切られ、各個室を 1 名の従業員のみで使用する場合は、1 対 1 の関連（1 個室に 1 従業員）となります。この関連は、次のように、単方向の関連として定義します。

employee － > cubicle

ただし、個室番号があり、その番号に従業員を割り当てる場合は、双方向の関連を割り当てることができます。この場合は、従業員を取得すると、その従業員に割り当てられた個室を検索できます。さらに、個室番号を取得すると、その個室を割り当てられた従業員を特定できます。双方向の 1 対 1 関連は、次のように定義します。

employee < － > cubicle

1 対多関連または多対 1 関連の概要

1 対多関連では、1 つのオブジェクトが別の複数のインスタンスを参照できます。多対 1 関連は、複数のオブジェクトが 1 つのオブジェクトを参照する場合です。たとえば、従業員は、自宅と勤務先の 2 つの住所を持つことができます。この関連を従業員の観点から単方向として定義した場合は、従業員を検索してその従業員の複数の住所を参照できますが、住所を検索して該当する従業員を参照することはできません。この関連を双方向として定義した場合は、住所を検索して該当する従業員を参照できます。

多対多関連の概要

多対多関連は複合した関連です。たとえば、各従業員は複数のプロジェクトに従事している可能性があります。そして、各プロジェクトには、従事している複数の従業員がいます。したがって、多対多のカーディナリティを設定できます。このインスタンスでは、方向は意味を持ちません。次のようなカーディナリティを設定できます。

employees < － > projects

多対多関連では、複数のオブジェクトが複数のオブジェクトを参照できます。これは、管理が最も難しいカーディナリティです。

関連の定義に関する要件

関連の定義に関する制限事項は、次のとおりです。

- 関連は、CMP 2.0 の Entity Bean 間のみで定義できます。
- 関連がある両方の EJB は、同じデプロイメント・ディスクリプタ内で宣言する必要があります。
- 各関連が使用できるのは、ターゲット EJB のローカル・インタフェースのみです。

カーディナリティの種類とその方向を定義するための要件は、次のとおりです。

1. 各関連フィールドに対して抽象アクセッサ・メソッド（`get` または `set` メソッド）を定義します。命名は、永続フィールドの抽象アクセッサ・メソッドと同じ命名規則に従います。たとえば、`getAddress` メソッドと `setAddress` メソッドは、住所を取得して設定するための抽象アクセッサ・メソッドです。
2. Bean 実装で、関連を設定します。主キーは常に `ejbCreate` メソッドで設定する必要があります。外部キーは `ejbCreate` メソッドの後にいつでも設定できますが、このメソッド内で設定することはできません。
3. 各関連（カーディナリティと方向）をデプロイメント・ディスクリプタで定義します。関連フィールド名は、`<cmr-field-name>` 要素で定義されます。この関連フィールド名は、`get/set` を付けずに最初の文字を小文字にした抽象アクセッサ・メソッド名と同じであることが必要です。たとえば、抽象アクセッサ・メソッド名が `getAddress/setAddress` の場合、`<cmr-field-name>` は `address` になります。
4. カスケード削除が必要な場合は、1 対 1、1 対多および多対 1 の関連に対してカスケード削除オプションを宣言します。カスケード削除は常に関連のスレーブ側で指定されるため、マスター Entity が削除されると、関連するすべてのスレーブ Entity も続けて削除されます。たとえば、従業員の電話番号が複数ある場合、カスケード削除は電話番号側で定義します。この場合、その従業員を削除すると、関連するすべての電話番号も削除されます。

次の各項では、これらの要件の実装方法の例を示します。

- 各関連フィールドの get/set メソッドの定義
- Bean 実装での関連の設定
- デプロイメント・ディスクリプタでの関連の宣言
- カスケード削除オプションを使用するかどうかの決定

各関連フィールドの get/set メソッドの定義

各関連フィールドには、その関連フィールドに対して定義された抽象アクセッサ・メソッドが必要です。単一のエンティティのみを設定または取得する関連の場合、受渡しされるオブジェクト型は、ターゲット Entity Bean のローカル・インタフェースであることが必要です。複数のエンティティを設定または取得する関連の場合、受渡しされるオブジェクト型は、ローカル・インタフェース・オブジェクトを含む `Set` または `Collection` です。

例 4-1 Employee の例での抽象アクセッサ・メソッドの定義

この例の従業員には、1 つの従業員番号と 1 つの住所があります。その従業員番号と住所は、この従業員からのみ取得できます。この場合は、従業員の観点からの単方向の 1 対 1 関連を定義します。employee Bean の抽象アクセッサ・メソッドは、次のようになります。

```
public Integer getEmpNo();
public void setEmpNo(Integer empNo);
public AddressLocal getAddress();
public void setAddress(AddressLocal address);
```

カーディナリティは 1 対 1 であるため、address Entity Bean のローカル・インタフェースは、抽象アクセッサ・メソッドで受渡しされるオブジェクト型になります。

関連のカーディナリティと方向は、デプロイメント・ディスクリプタで定義されます。

例 4-2 1 対多の抽象アクセッサ・メソッドの定義

employee の例に 1 対多の関連が含まれている場合、抽象アクセッサ・メソッドは、それぞれにターゲット Bean のローカル・インタフェース・オブジェクトが含まれているオブジェクトの `Set` または `Collection` を受け渡すことになります。「多」の関連の場合は、複数のレコードが受渡しされます。

たとえば、ある部門に複数の従業員が所属しているとします。このような 1 対多の例の場合、その部門の抽象アクセッサ・メソッドは複数の従業員を取得します。したがって、抽象アクセッサ・メソッドは、次のように、従業員の `Collection` または `Set` を渡します。

```
public Collection getDeptEmployees();
public void setDeptEmployees(Collection deptEmpl);
```

Bean 実装での関連の設定

関連の get/set メソッドを定義した後は、これらのメソッドを Bean 実装で使用して関連を設定します。3-6 ページの「Entity Bean クラス」で示すように、主キーのすべての関連は `ejbCreate` メソッド内で設定する必要があります。外部キーを使用する場合は、4-56 ページの「コンポジット主キーでの外部キーの使用」で説明するように、`ejbPostCreate` メソッドで簡単に外部キーを設定できます。

`ejbCreate` で主キーを設定すると、set メソッドによって、デプロイメント・ディスクリプタで定義した CMP フィールドが入力されます。`ejbCreate` メソッドの最後に、このフィールドは適切なデータベース行に書き込まれます。

従業員は、従業員番号の主キーを持っています。次のコード例は、部門の主キーを設定します。

```
public Integer ejbCreate(Integer empNo) throws CreateException
{
    setEmpNo(empNo);
    return empNo;
}
```

デプロイメント・ディスクリプタでの関連の宣言

Entity Bean 間の関連は、その Entity Bean が宣言されたのと同じデプロイメント・ディスクリプタで定義します。エンティティ間のすべての関連は <relationships> 要素内に定義され、複数の関連はこの要素内で定義できます。エンティティ間の各関連は、<ejb-relation> 要素内に定義されます。次の XML は、1 つのアプリケーション内に定義された、エンティティ間の 2 つの関連を示します。

```
<relationships>
      <ejb-relation>
      ...
      </ejb-relation>
      <ejb-relation>
      ...
      </ejb-relation>
</relationships>
```

次の XML は、関連を構成するすべての要素の構造を示します。

```
<relationships>
  <ejb-relation>
    <ejb-relation-name> </ejb-relation-name>
    <ejb-relationship-role>
      <ejb-relationship-role-name> </ejb-relationship-role-name>
      <multiplicity> </multiplicity>
      <relationship-role-source>
         <ejb-name> </ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name> </cmr-field-name>
        <cmr-field-type> </cmr-field-type>
      </cmr-field>
    </ejb-relationship-role>
  </ejb-relation>
</relationships>
```

注意： オブジェクト関連の Entity Bean の例は、OTN-J のサイト http://otn.oracle.co.jp/sample_code/index.html の OC4J のサンプル・コードのページから入手できます。

表 4-1 では、各要素の使用方法を説明します。

表 4-1 デプロイメント・ディスクリプタの関連要素の説明

デプロイメント・ディスクリプタの要素	説明
<ejb-relation>	エンティティ間の各関連は、単一の <ejb-relation> 要素内に記述されます。
<ejb-relation-name>	エンティティ間の関連のユーザー定義名。
<ejb-relationship-role>	関連内の各エンティティは、それぞれの <ejb-relationship-role> 内に記述されます。したがって、<ejb-relation> 内には、常に 2 つの <ejb-relationship-role> エンティティが存在します。
<ejb-relationship-role-name>	関連内での Entity Bean のロールまたは関係を記述するためのユーザー定義名。
<multiplicity>	このエンティティのカーディナリティの宣言。値は「One」または「Many」になります。
<relationship-role-source><ejb-name>	Entity Bean の名前。この名前は、ejb-jar.xml ファイルの <entity><ejb-name> 要素に定義した EJB 名と同一にする必要があります。
<cmr-field><cmr-field-name>	ターゲット Bean 参照を表すユーザー定義名。この名前は、抽象アクセッサ・メソッド名と一致する必要があります。たとえば、抽象アクセッサ・フィールドが getAddress() および setAddress() の場合、CMR フィールドは address になります。
<cmr-field><cmr-field-type>	オプション。「多」の関連の場合、この型は Collection または Set になります。これは、Collection または Set が返されるかどうかを通知するために、「多」の側でのみ指定します。

関連には、1 対 1、1 対多または多対多があります。カーディナリティは、<multiplicity> 要素内に定義します。各 Bean では、それぞれの関連内でのカーディナリティを定義します。次に例を示します。

- 1 対 1: 1 名の従業員に対して 1 つの住所が関連付けられている場合、employee Bean は <multiplicity> を One に指定して宣言し、address Bean も <multiplicity> を One に指定して宣言します。
- 1 対多、多対 1: 1 つの部門に対して複数の従業員が関連付けられている場合、department Bean は <multiplicity> を One に指定して宣言し、employee Bean は <multiplicity> を Many に指定して宣言します。複数の従業員が 1 つの部門に所属している場合は、同じ <multiplicity> を定義します。
- 多対多: 各従業員が複数のプロジェクトに関連付けられ、各プロジェクトには複数の従業員が従事している場合、employee Bean は <multiplicity> を Many に指定して宣言し、project も <multiplicity> を Many に指定して宣言します。

関連の方向は、<cmr-field> 要素の有無によって定義します。ターゲット・エンティティへの参照は、<cmr-field> 要素内で定義します。関連が単方向の場合、ターゲットへの参照が含まれるのは、関連内の 1 つのエンティティのみです。この場合、<cmr-field> 要素は、ソース・エンティティで宣言され、ターゲット Bean 参照が含まれます。関連が双方向の場合は、両方のエンティティが互いの Bean への参照を <cmr-field> 要素内で宣言する必要があります。

次に、従業員と住所が 1 対 1 関連の例を使用して、方向を宣言する方法を説明します。

- 単方向: address Bean を参照する employee Bean セクション内に <cmr-field> 要素を定義します。関連の address Bean セクション内に <cmr-field> 要素を定義しないでください。
- 双方向: address Bean を参照する employee Bean セクション内に、<cmr-field> 要素を定義します。さらに、employee Bean を参照する address Bean セクション内に、<cmr-field> 要素を定義します。

E-R のカーディナリティと方向の宣言方法を理解すると、各関連の EJB デプロイメント・ディスクリプタを簡単に構成できます。

例 4-3 1 対 1 関連の例

employee の例では、各従業員が 1 つの住所のみ持つ 1 対 1 の単方向の関連を定義しています。この関連は、従業員から住所を取得できますが住所からは従業員を取得できないため、単方向の関連です。したがって、employee オブジェクトが address オブジェクトへの関連を持ちます。

この例では、次のように ejb-jar.xml ファイルが構成されています。

```
<enterprise-beans>
  <entity>
    ...
    <ejb-name>EmpBean</ejb-name>
    <local-home>employee.EmpHome</local-home>
    <local>employee.Emp</local>
    <ejb-class>employee.EmpBean</ejb-class>
    ...
  </entity>
  <entity>
    ...
    <ejb-name>AddressBean</ejb-name>
    <local-home>employee.AddressHome</local-home>
    <local>employee.Address</local>
    <ejb-class>employee.AddressBean</ejb-class>
    ...
  </entity>
</enterprise-beans>
...
<relationships>
   <ejb-relation>
         <ejb-relation-name>Emp-Address</ejb-relation-name>
         <ejb-relationship-role>
            <ejb-relationship-role-name>Emp-has-Address
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source><ejb-name>EmpBean</ejb-name>
            </relationship-role-source>
            <cmr-field>
               <cmr-field-name>address</cmr-field-name>
            </cmr-field>
         </ejb-relationship-role>
         <ejb-relationship-role>
            <ejb-relationship-role-name>Address-has-Emp
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source><ejb-name>AddressBean</ejb-name>
            </relationship-role-source>
         </ejb-relationship-role>
      </ejb-relation>
</relationships>
```

ejb-jar.xml ファイルでは、次の要素を定義しています。

- 関連に含まれる各 Entity Bean について、<enterprise-beans> セクション内で <entity> 要素を構成します。この例では、従業員に関する <entity> 要素（<ejb-name> は EmpBean）および住所に関する <entity> 要素（<ejb-name> は AddressBean）があります。
- 1 対 1 関連について、<relationships> セクション内の <ejb-relationship> 要素を構成します。この例では、次の要素を定義します。
  - <multiplicity> 要素でカーディナリティを「One」に定義した employee Bean に対する <ejb-relationship-role> 要素。<relationship-role-source> 要素では、<ejb-name> を EmpBean に定義します。これは、<entity> 要素内の名前と同じです。
  - <multiplicity> 要素でカーディナリティを「One」に定義した address Bean に対する <ejb-relationship-role> 要素。<relationship-role-source> 要素では、<ejb-name> を AddressBean に定義します。これは、<entity> 要素内の名前と同じです。
- AddressBean を指定する EmpBean 関連の <cmr-field> 要素を構成します。<cmr-field> 要素では、address を AddressBean への参照として定義します。この要素の名前は、get メソッドと set メソッドの名前（getAddress と setAddress）と一致します。これらのメソッドは、address Entity Bean のローカル・インタフェースを、get メソッドから返されて set メソッドに渡されるデータ型として識別します。

注意： オブジェクト関連の Entity Bean の例は、OTN-J のサイト http://otn.oracle.co.jp/sample_code/index.html の OC4J のサンプル・コードのページから入手できます。

カスケード削除オプションを使用するかどうかの決定

Entity Bean 間に関連が定義され、マスター Entity Bean が削除された場合、スレーブ Bean は次のように処理されます。カスケード削除を指定した場合は、マスター Entity Bean を削除すると、関連内のすべてのスレーブ Entity Bean が自動的に削除されます。カスケード削除オプションは、自動的に削除されるオブジェクトであるスレーブ関連定義で指定します。

たとえば、employee オブジェクトが address オブジェクトに関連付けられているとします。address オブジェクトでカスケード削除を指定します。この関連のマスターである employee オブジェクトが削除されると、スレーブの address オブジェクトも削除されます。

あるケースにおいては、カスケード削除が不要な場合があります。たとえば、1 つの部門内に複数の従業員が関連付けられている場合、部門が削除されたとき、その部門のすべての従業員を削除する必要はありません。

関連にカスケード削除を指定できるのは、マスター Entity Bean の <multiplicity> が One の場合のみです。したがって、1 対 1 関連では、マスターはいずれかの「1」の側になります。カスケード削除は 1 対多の関連で指定できますが、多対 1 または多対多の関連では指定できません。

例 4-4 Employee の例でリクエストされるカスケード削除

次のデプロイメント・ディスクリプタは、従業員とその住所の 1 対 1 関連を定義しています。従業員を削除すると、スレーブ Entity Bean（住所）も自動的に削除されます。この削除を指定するには、関連のスレーブ Entity Bean 内で <cascade-delete/> 要素を指定します。この場合は、AddressBean 定義で <cascade-delete/> 要素を指定します。

```
<relationships>
   <ejb-relation>
         <ejb-relation-name>Emp-Address</ejb-relation-name>
         <ejb-relationship-role>
            <ejb-relationship-role-name>Emp-has-Address
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source><ejb-name>EmpBean</ejb-name>
            </relationship-role-source>
            <cmr-field>
               <cmr-field-name>address</cmr-field-name>
            </cmr-field>
         </ejb-relationship-role>
         <ejb-relationship-role>
            <ejb-relationship-role-name>Address-has-Emp
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
           <cascade-delete/>
            <relationship-role-source><ejb-name>AddressBean</ejb-name>
            </relationship-role-source>
         </ejb-relationship-role>
      </ejb-relation>
</relationships>
```

データベースへのオブジェクト関連フィールドのマッピング

各 Entity Bean は、データベース内の表にマッピングされます。永続フィールドと関連フィールドは、それぞれデータベース表の列に保存されます。データベースにマッピングされるこれらのフィールドに対して、次のいずれかの操作を実行します。

- これらのフィールドのデフォルトを使用し、追加のデプロイメント・ディスクリプタは構成しません。デフォルトのマッピング方法については、4-12 ページの「関連フィールドのデータベースへのデフォルト・マッピング」を参照してください。表は、`ejb-jar.xml` ファイルの情報に基づいて、Bean に対して自動的に作成されます。
- 指定したデータベースにすでに存在する表内の列に、フィールドをマッピングします。永続データのマッピングは、`orion-ejb-jar.xml` ファイル内に構成されます。詳細は、4-16 ページの「関連フィールドのデータベースへの明示的なマッピング」を参照してください。

関連フィールドのデータベースへのデフォルト・マッピング

注意： ここでは、OC4J で関連フィールドをデータベースにマッピングする方法を説明します。永続フィールドのマッピングについては、第 3 章「CMP Entity Bean」を参照してください。

関連フィールドを `ejb-jar.xml` ファイルで宣言すると、OC4J では、`orion-ejb-jar.xml` ファイルを自動生成するときに、これらのフィールドのデータベースへのデフォルト・マッピングを提供します。関連フィールドのデフォルト・マッピングは、3-15 ページの「永続フィールドのデータベースへのデフォルト・マッピング」で説明する永続フィールドのマッピングと同じです。

注意： 再度デプロイする場合に備えて、この表名で自動生成された `orion-ejb-jar.xml` ファイルを、`J2EE_HOME/application-deployments` ディレクトリから `ejb-jar.xml` ファイルと同じディレクトリにコピーします。これによって、再度デプロイするとき、最初に生成された表名と同じ名前を使用できます。このファイルをコピーしないと、異なる表名が生成される場合があります。

要約すると、次のようにデフォルト設定されます。

- データベース : 使用中の OC4J インスタンス構成に設定されているデフォルトのデータベース。
- デフォルト表 : 関連内の各 Entity Bean は、それぞれのデータベース表のデータを表します。Entity Bean 表の名前は、次の名前をアンダースコア（_）で区切って一意の名前になるように生成されます。
  - EJB 名 : デプロイメント・ディスクリプタの <ejb-name> に定義されています。
  - JAR ファイル名 : .jar 拡張子を含みます。ただし、SQL 表記規則に従って、ダッシュ（-）とピリオド（.）はすべてアンダースコア（_）に変換されます。たとえば、JAR ファイル名が employee.jar の場合は、employee_jar に変換されて表名に含まれます。
  - アプリケーション名 : デプロイ時に定義するアプリケーション名です。

  構成された名前が 31 文字以上の場合、その名前は 24 文字で切り捨てられます。一意の名前にするため、これに、アンダースコアと 5 文字で構成される英数字のハッシュ・コードが追加されます。

  たとえば、EJB 名が EmpBean、JAR ファイル名が empl.jar、アプリケーション名が employee の場合、デフォルトの表名は EmpBean_empl_jar_employee になります。
- 各表の列名 : コンテナは、デプロイメント・ディスクリプタで定義された <cmp-field> および <cmr-field> 要素に基づいて、各表に列を生成します。列は、Entity Bean データに関連した <cmp-field> 要素ごとに作成されます。さらに、関連を表す <cmr-field> 要素ごとにも列が作成されます。単方向の関連の場合、デプロイメント・ディスクリプタ内で <cmr-field> を定義するのは、関連内の一方のエンティティのみです。双方向の関連の場合は、関連内の両方のエンティティで <cmr-field> を定義します。

  各 <cmr-field> 要素について、コンテナは、関連オブジェクトの主キーを指す外部キーを次のように作成します。
  - デフォルトの 1 対 1 関連では、外部キーがソース EJB のデータベース表に作成され、ターゲット・データベース表の主キーにダイレクトされます。たとえば、1 名の従業員が 1 つの住所を持つ場合は、address 表の主キーを指す外部キーが employee 表内に作成されます。詳細は、4-14 ページの「1 対 1 関連のデフォルト・マッピング例」を参照してください。
  - 1 対多関連のデフォルトでは、4-28 ページの「外部キーを使用する 1 対多関連」で説明するように、外部キーを使用します。
  - 多対多関連のデフォルトでは、関連表（3 番目の表）が作成されます。関連表には 2 つの外部キーが含まれ、各外部キーはいずれかのエンティティ表の主キーを指します。詳細は、4-15 ページの「1 対多関連および多対多関連のデフォルト・マッピング例」を参照してください。

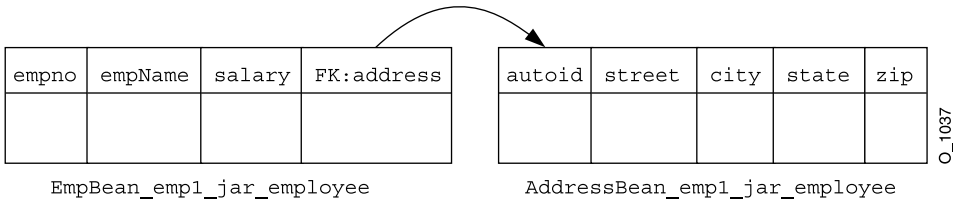
<cmp-field> 要素および <cmr-field> 要素は、Java データ型を表すため、データベース型への変換方法が従来の方法と異なります。変換方法については、3-19 ページの「CMP の型からデータベース型への変換」の表を参照してください。ただし、Java データ型をデータベース・データ型に変換するための変換ルールは、j2ee/home/config/database-schemas にある特定のデータベース XML ファイルで変更できます。このディレクトリには、すべてのデータベース・ファイルが格納されます。Oracle データベースの変換ファイルの名前は、oracle.xml です。

- 主キーの生成： 主キーは両方のエンティティ表にあります。主キーは、手動で定義するか、または自動生成できます。詳細は、3-9 ページの「主キー」を参照してください。
  - 主キーの定義： 主キーは、<primkey-field> 要素で指定されたように、単純なデータ型またはクラスとして生成されます。したがって、列名は、<primkey-field> 要素内の名前と同じです。
  - コンポジット主キー： 主キーはクラス内で定義され、複数のフィールドで構成されます。コンポジット主キー内の各フィールドはデータベース表の列で表され、表内の各列は主キーの一部とみなされます。
  - 主キーの自動生成： java.lang.Object を <prim-key-class> に主キーとして指定し、主キー名を <primkey-field> に指定しなかった場合、その主キーはコンテナによって自動生成されます。列の名前は AUTOID になります。

1 対 1 関連のデフォルト・マッピング例

1 対 1 の E-R は、外部キーを使用してエンティティ表の間で管理されます。図 4-1 は、employee Bean と address Bean の間における 1 対 1 の単方向の関連のデフォルト表マッピングを示します。

図 4-1 employee の例での 1 対 1 関連



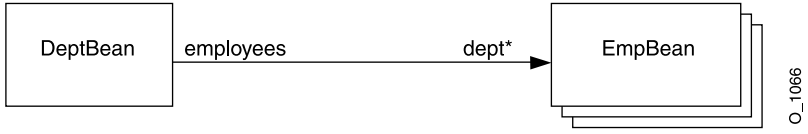
- コンテナは、Entity Bean 名、Bean のアーカイブ先 JAR ファイル、および Bean がデプロイされるアプリケーションの名前に基づいて、表名を生成します。JAR ファイル名が empl.jar、アプリケーション名が employee の場合、表名は表名 EmpBean_empl_jar_employee および AddressBean_empl_jar_employee になります。

- コンテナは、デプロイメント・ディスクリプタで宣言された <cmp-field> および <cmr-field> 要素に基づいて、各表に列を生成します。
  - EmpBean 表の列は empno、empname および salary です。<cmr-field> 宣言からは、AddrBean 表の主キー列を指す address という名前の外部キーが作成されます。
  - AddressBean 表の列は自動生成され、long 型の主キーおよび street、city、state、zip の各列が生成されます。
- employee 表の主キーは、デプロイメント・ディスクリプタで empno という名前で指定されます。Address Bean は、java.lang.Object の <primkey-class> のみを指定することによって、主キーが自動生成されるように構成されます。

1 対多関連および多対多関連のデフォルト・マッピング例

4-3 ページの「1 対多関連または多対 1 関連の概要」で説明したように、1 つの Bean が別の Bean の複数のインスタンスと関連を持つことができます。たとえば、1 つの部門に複数の従業員を関連付けることができます。この場合、各部門に複数の従業員が所属します。この関連は双方向であるため、従業員から部門を参照できます。DeptBean と EmpBean の間の関連は、図 4-2 に示すように、CMR フィールドの employees および deptno で表します。

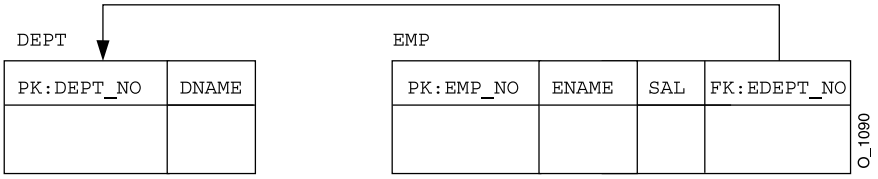
図 4-2 Bean の 1 対多関連



この関連をデータベース表にマッピングする方法は、開発者が選択します。デフォルトの方法では、外部キーを関連の「多」の側を定義する表に追加します。この例では、EmpBean を表す表です。外部キーは、各従業員が所属する部門を指します。

図 4-3 に、department<—>employee の例を示します。この例では、各従業員が 1 つの部門にのみ所属し、各部門には複数の従業員が所属している可能性があります。department 表には主キーがあります。employee 表には、各従業員を識別するための主キーと、従業員の部門を指すための外部キーがあります。特定の従業員の部門を検索する場合は、単純な SQL 文を使用して、外部キーから部門情報を取得できます。部門内の全従業員を検索する場合、コンテナは、department 表と employee 表の両方で JOIN 文を実行し、指定した部門番号を持つ全従業員を取得します。

図 4-3 1 対多の双方向の関連のデフォルト・マッピング例



この動作を行うようにデプロイメント・ディスクリプタを構成する方法は、4-28 ページの「外部キーを使用する 1 対多関連」で説明します。再度デプロイする場合に同じデフォルトを使用するために、デフォルトの表名で自動生成された orion-ejb-jar.xml ファイルを、J2EE_HOME/application-deployments ディレクトリから ejb-jar.xml ファイルと同じディレクトリにコピーします。これによって、再度デプロイするとき、最初に生成された表名と同じ名前を使用できます。このファイルをコピーしないと、異なる表名が生成される場合があります。デフォルトを変更する場合は、このファイルをコピーして、4-28 ページの「外部キーを使用する 1 対多関連」で説明する手順に従ってください。

関連フィールドのデータベースへの明示的なマッピング

4-12 ページの「関連フィールドのデータベースへのデフォルト・マッピング」で説明したように、関連フィールドは、コンテナによって自動的にデータベース表にマッピング可能です。ただし、OC4J で提供されているデフォルトを使用しない場合、またはフィールドを既存のデータベース表にマッピングする必要がある場合は、orion-ejb-jar.xml ファイルで Entity Bean 間の関連を既存のデータベース表とその列内にマッピングできます。

CMP フィールドの明示的なマッピング方法は、3-16 ページの「永続フィールドのデータベースへの明示的なマッピング」で説明しています。ここでは、CMR フィールドのマッピング方法について説明し、その情報に基づいて関連のマッピング方法を説明します。

重要： 関連フィールドを明示的にマッピングするには、orion-ejb-jar.xml ファイルの <entity-deployment> 要素の要素と属性を変更してください。**JDeveloper は、Entity Bean とデータベース表との間の複雑なマッピングを管理するために作成されています。JDeveloper ではデプロイメント・ディスクリプタが検証され、不一致が防止されます。orion-ejb-jar.xml ファイルは手動で変更することもできますが、コンテナ管理の関連を変更する場合は、JDeveloper を使用することをお薦めします。**これは、CMR の構成が複雑で、理解するのが困難なためです。JDeveloper は、次のサイトからダウンロードできます。
http://otn.oracle.co.jp/software/products/jdev/index.html

この章では、orion-ejb-jar.xml 要素について、次の 2 つのレベルで説明します。

- 既存のデータベースにマッピングした場合に変更対象フィールドを識別するためのクイック・ガイド。4-17 ページの「既存のデータベースと Bean マッピングを一致させる方法のクイック・ガイド」を参照してください。
- CMR マッピングで使用するすべての要素の説明、およびそれらの要素の変更手順の説明。4-18 ページの「CMR マッピング要素の変更手順」を参照してください。

既存のデータベースと Bean マッピングを一致させる方法のクイック・ガイド

orion-ejb-jar.xml ファイルの各要素に関する知識がなく、JDeveloper を使用せずにこの XML ファイルを変更するには、次の手順を実行します。

1. orion-application.xml ファイルで autocreate-tables 要素を false に設定して Bean をデプロイします。
2. orion-ejb-jar.xml ファイルを application-deployments/ ディレクトリから、使用する開発ディレクトリにコピーします。
3. data-source 要素を適切なデータ・ソースに変更します。相互に関連付けられた Bean はすべて同じデータ・ソースを使用する必要があることに注意してください。
4. table 属性を適切な表に変更します。変更した表が、<entity-deployments> 要素で定義した Bean に対して適切な表であることを確認してください。
5. 各 Bean の永続タイプに応じて、persistence-name 属性を適切な列（CMP または CMR フィールド）に変更します。
6. orion-application.xml ファイルの autocreate-tables 要素を true に設定します。
7. アプリケーションを再アーカイブして再デプロイします。

注意： オブジェクト関連の Entity Bean の例は、OTN-J のサイト http://otn.oracle.co.jp/sample_code/index.html の OC4J のサンプル・コードのページから入手できます。

CMR マッピング要素の変更手順

必要なマッピングが JDeveloper に用意されていない場合、または手動で XML を管理する必要がある場合は、次の手順を実行します。

1. `orion-application.xml` ファイルで `autocreate-tables` 要素を false に設定し、`ejb-jar.xml` 要素を構成して、Bean をデプロイします。

   OC4J では、`orion-ejb-jar.xml` ファイルを作成してデフォルトのマッピングを行います。フィールドは最初から作成するより、変更する方が簡単です。

2. コンテナが作成した `orion-ejb-jar.xml` ファイルを、`$J2EE_HOME/application-deployments` ディレクトリから、使用する開発環境にコピーします。

3. 関連タイプに応じて `orion-ejb-jar.xml` ファイルの `<entity-deployment>` 要素を変更し、指定したデータベース表と列が使用されるようにします。概要は、4-19 ページの「Bean 関連をデータベース表にマッピングするための orion-ejb-jar.xml ファイルの手動編集」を参照してください。

   次の各項では、関連タイプに応じた CMR マッピング方法を説明します。

   - 1 対 1 関連の明示的なマッピング
   - autoid を主キーとして使用する表のマッピング
   - 外部キーを使用する 1 対多関連
   - 関連に対する関連表の明示的なマッピングの概要
   - 1 対多の双方向の関連における関連表の使用
   - 1 対多の単方向の関連における関連表の使用
   - 多対多関連における関連表の使用

4. `orion-application.xml` ファイルの `autocreate-tables` 要素を true に設定します。

5. アプリケーションを再アーカイブして再デプロイします。

注意： `autocreate-tables` を false に設定せずにデプロイした場合は、OC4J によってデフォルト表が自動的に作成されています。アプリケーションを再デプロイする前に、これらの表をすべて削除する必要があります。関連表を使用している場合は、その関連表も削除する必要があります。

Bean 関連をデータベース表にマッピングするための orion-ejb-jar.xml ファイルの手動編集

Bean 間の関連は、ejb-jar.xml ファイルの <relationships> 要素で定義されます。Bean と、データベース表および列とのマッピングは、orion-ejb-jar.xml ファイルの <entity-deployment> 要素で指定されます。

orion-ejb-jar.xml ファイルでは、<cmp-field-mapping> 要素内で、Bean の E-R がデータベース表と列にマッピングされます。次に、単純な 1 対 1 関連の <entity-deployment> および <cmp-field-mapping> 要素の XML 構造を示します。

```
<entity-deployment name="SourceBeanName" location="JNDIlocation"
    table="TableName" data-source="DataSourceJNDIName">
...
<cmp-field-mapping name="CMRfield_name">
    <entity-ref home="targetBeanName">
      <cmp-field-mapping name="CMRfield_name"
persistence-name="targetBean_PKcolumn" />
    </entity-ref>
</cmp-field-mapping>
```

この要素内では、Bean 名（方向を指定する関連のソース）、JNDI ロケーション、および情報を維持するデータベース表を定義でき、ejb-jar.xml ファイルで定義された CMP および CMR フィールドをそれぞれ基礎となる永続記憶域であるデータベースにマッピングできます。

> **注意：** このマニュアルでは、Bean を関連のソースまたはターゲットとみなします。単方向参照で従業員が複数の電話番号を持っている場合、その従業員はソース Bean になり、ターゲットである電話番号を指します。

<entity-deployment> 要素の属性によって、Bean に関する次の項目が定義されます。

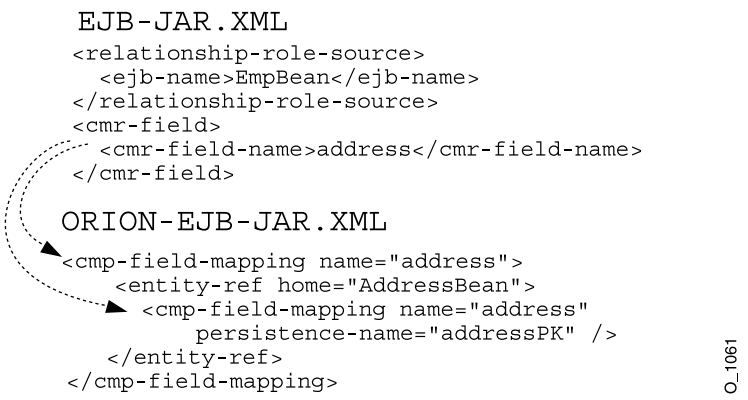
- name 属性は、Bean の EJB 名を識別します。これは、ejb-jar.xml ファイルの <ejb-name> 要素で定義されています。この name 属性によって、Bean の ejb-jar.xml ファイル定義と、データベースへのマッピングが関連付けられます。
- location 属性は、Bean の JNDI 名を識別します。
- table 属性は、この Entity Bean のマッピング先のデータベース表を識別します。
- data-source 属性は、表が存在するデータベースを識別します。相互に作用する Bean、または相互に関連付けられた Bean はすべて同じデータ・ソースを使用する必要があります。これには、同じアプリケーション内の Bean、同じトランザクションの一部である Bean、または親子関係にある Bean も含まれます。

orion-ejb-jar.xml ファイルの <cmp-field-mapping> 要素は、次のフィールドをデータベース列にマッピングします。

- ejb-jar.xml ファイルの <cmp-field> 要素は、CMP フィールドを定義します。
- ejb-jar.xml ファイルの <cmr-field> 要素は、CMR フィールドを定義します。

図 4-4は、ejb-jar.xml ファイルの <cmr-field> 要素を、orion-ejb-jar.xml ファイルの <cmp-field-mapping> 要素にマッピングする方法を示しています。<cmp-field-mapping> の name 属性によって、2 つの XML ファイル間のリンクが提供されます。ここでは、いずれの name 属性も変更する必要はありません。

図 4-4 1 対 1 関連のマッピングの例



CMR フィールドを完全に識別してマッピングするために、ネストされた <cmp-field-mapping> 要素が使用されます。ネストの形式は、関連タイプによって異なります。ターゲット Bean の主キーであるデータベース列は、内部の <cmp-field-mapping> 要素の persistence-name 属性で定義されます。既存のデータベースがある場合は、その列名と一致するように、各 <cmp-field-mapping> 要素の persistence-name 属性を変更します。

次の各項では、各関連タイプおよびマッピング方法を説明します。

- 1 対 1 関連の明示的なマッピング
- autoid を主キーとして使用する表のマッピング
- 外部キーを使用する 1 対多関連
- 関連に対する関連表の明示的なマッピングの概要
- 1 対多の双方向の関連における関連表の使用

1 対 1 関連の明示的なマッピング

図 4-5 は、従業員とその住所の 1 対 1 の単方向の関連を示します。EmpBean は、CMR フィールド address を使用して、従業員の住所である AddressBean を指します。

図 4-5 Bean の 1 対 1 関連

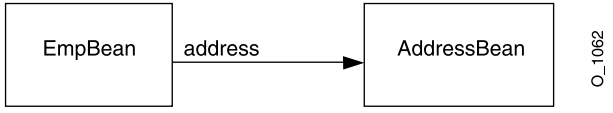
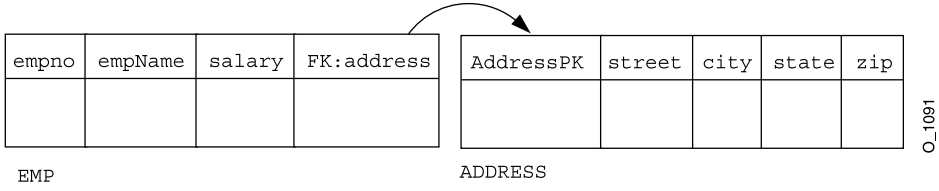


図 4-6 に、これらの Bean をマッピングするデータベース表の EMP および ADDRESS を示します。EMP 表には address という名前の外部キーがあり、ADDRESS 表の主キー AddressPK を指します。

図 4-6 employee の例での 1 対 1 関連



Bean とその関連は、両方のデプロイメント・ディスクリプタで指定されます。図 4-7 に示すように、ejb-jar.xml ファイルでは、EmpBean と AddressBean の間の 1 対 1 関連が <relationships> 要素で定義されています。方向は、1 つまたは 2 つの <cmr-field> 要素で指定されています。

Bean からデータベース永続記憶域へのマッピングは、orion-ejb-jar.xml ファイルで定義されています。1 対 1 関連は、双方向か単方向かに関係なく、<cmp-field-mapping> 要素内で <entity-ref> 要素の両側にマッピングされます。<entity-ref> では、関連のターゲット Entity Bean を記述します。

図 4-7 1 対 1 関連のマッピングの例

Bean フィールドを既存のデータベースにマッピングするには、orion-ejb-jar.xml ファイルの <cmp-field-mapping> 要素内のフィールドについて理解する必要があります。この要素の構造は、次のとおりです。

```
<cmp-field-mapping name="CMRfield_name">
    <entity-ref home="targetBeanName">
      <cmp-field-mapping name="CMRfield_name"
                  persistence-name="targetBean_PKcolumn" />
    </entity-ref>
</cmp-field-mapping>
```

- <cmp-field-mapping> 要素の name> 属性は、ejb-jar.xml ファイルの <cmp-field 要素と同じです。<cmp-field-mapping> 要素の name 属性は変更しないでください。
- ターゲット Bean 名は、<entity-ref> 要素の home 属性で指定されます。
- ターゲット Bean の主キーであるデータベース列は、内部の <cmp-field-mapping> 要素の persistence-name 属性で定義されます。既存のデータベースがある場合は、その列名と一致するように、各 <cmp-field-mapping> 要素の persistence-name 属性を変更します。

例 4-5 1 対 1 の単方向の関連の XML 構成

ejb-jar.xml ファイル構成では、EmpBean と AddressBean との間の 1 対 1 の単方向の関連を定義します。

```
<enterprise-beans>
  <entity>
    ...
    <ejb-name>EmpBean</ejb-name>
    <local-home>employee.EmpHome</local-home>
    <local>employee.Emp</local>
    <ejb-class>employee.EmpBean</ejb-class>
    ...
    <cmp-field><field-name>empNo</field-name></cmp-field>
    <cmp-field><field-name>empName</field-name></cmp-field>
    <cmp-field><field-name>salary</field-name></cmp-field>
    <primkey-field>empNo</primkey-field>
    <prim-key-class>java.lang.Integer</prim-key-class>
    ...
  </entity>
  <entity>
    ...
    <ejb-name>AddressBean</ejb-name>
    <local-home>employee.AddressHome</local-home>
    <local>employee.Address</local>
    <ejb-class>employee.AddressBean</ejb-class>
```

```
    ...
    <cmp-field><field-name>addressPK</field-name></cmp-field>
    <cmp-field><field-name>addressDescription</field-name></cmp-field>
    <primkey-field>addressPK</primkey-field>
    <prim-key-class>java.lang.Integer</prim-key-class>
    ...
  </entity>
</enterprise-beans>
<relationships>
   <ejb-relation>
         <ejb-relation-name>Emp-Address</ejb-relation-name>
         <ejb-relationship-role>
            <ejb-relationship-role-name>Emp-has-Address
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source><ejb-name>EmpBean</ejb-name>
            </relationship-role-source>
            <cmr-field>
               <cmr-field-name>address</cmr-field-name>
            </cmr-field>
         </ejb-relationship-role>
         <ejb-relationship-role>
            <ejb-relationship-role-name>Address-has-Emp
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source><ejb-name>AddressBean</ejb-name>
            </relationship-role-source>
       </ejb-relationship-role>
      </ejb-relation>
</relationships>
```

EmpBean では、各従業員が 1 つの住所を持つことを示す関連の方向を <cmr-field> で定義します。EmpBean をサポートする EMP 表では AddressBean をサポートする表を指すために外部キーが必要です。

EMP 表から ADDRESS 表への外部キーは、<cmr-field-name> 要素内で address として識別されます。これは、orion-ejb-jar.xml ファイルにある <cmp-field-mapping> 要素の name 属性で必要です。したがって、address は、ejb-jar.xml ファイルで定義された関連を、orion-ejb-jar.xml ファイルで指定された永続記憶域マッピングにリンクする識別子になります。

次に、既存のデータベース表にマッピングするために要素を変更した orion-ejb-jar.xml ファイルを示します。

```
<entity-deployment name="EmpBean" location="emp/EmpBean" ...
    table="EMP" data-source="jdbc/OracleDS" ...>
       <primkey-mapping>
            <cmp-field-mapping name="empNo" persistence-name="EMPNO" />
       </primkey-mapping>
       <cmp-field-mapping name="empName" persistence-name="ENAME" />
       <cmp-field-mapping name="salary" persistence-name="SAL" />
       <cmp-field-mapping name="address">
           <entity-ref home="AddressBean">
              <cmp-field-mapping name="address"
                                 persistence-name="addressPK" />
           </entity-ref>
       </cmp-field-mapping>
       ...
</entity-deployment>
<entity-deployment name="AddressBean" location="emp/AddressBean" ...
    table="ADDRESS" data-source="jdbc/OracleDS"... >
        <primkey-mapping>
            <cmp-field-mapping name="addressPK"
                persistence-name="addressPK" />
        </primkey-mapping>
        <cmp-field-mapping name="street" persistence-name="street" />
        <cmp-field-mapping name="city" persistence-name="city" />
        <cmp-field-mapping name="state" persistence-name="state" />
        <cmp-field-mapping name="zip" persistence-name="zip" />
        <cmp-field-mapping name="EmpBean_address">
            <entity-ref home="EmpBean">
                <cmp-field-mapping name="EmpBean_address"
                                   persistence-name="EMPNO" />
            </entity-ref>
        </cmp-field-mapping>
...
</entity-deployment>
```

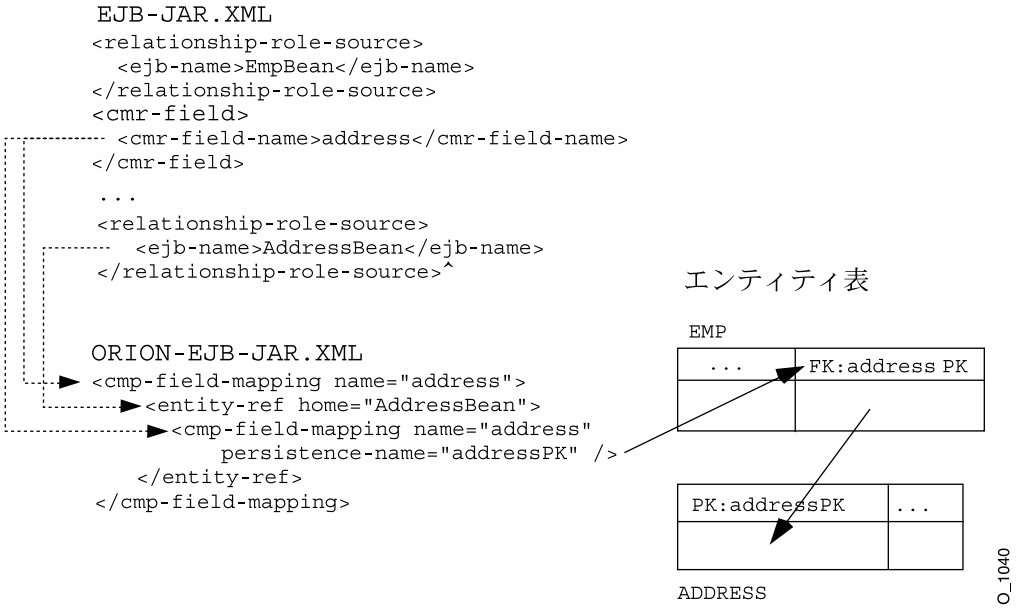
注意： ここでは、ejb-jar.xml ファイルで定義された論理名を orion-ejb-jar.xml ファイルで定義された論理名に関連付ける方法について説明します。次に、orion-ejb-jar.xml ファイルで定義された論理変数をデータベース表と列名に関連付ける方法を説明します。この章では、ejb-jar.xml および orion-ejb-jar.xml ファイル内で近くに並べられる要素に対して、意図的に異なる名前を使用しています。これによって、マッピングを行う要素名が明確になります。ただし、すべて同じ名前を使用する方が効率的で簡単です。たとえば、CMR フィールド名とデータベース列名にそれぞれ address と addressPK という名前を付けて識別するのではなく、両方に address という名前を付けることができます。**この場合、同じ名前を使用すると、構成がより簡単になります。**

EmpBean の <entity-deployment> マッピングでは、次のように指定します。

- <entity-deployment> の属性で定義する内容は、次のとおりです。
  - name 属性 : ソース Bean の名前は EmpBean です。
  - location 属性 : JNDI ロケーションは emp/EmpBean です。
  - table 属性 : この Entity Bean の永続データが格納されるデータベース表は emp です。
  - data-source 属性 : この表が存在するデータベースは、データ・ソース jdbc/OracleDS で定義されます。
- <cmp-field-mapping> 要素は、表列および各列に格納される永続データを識別します。この表の列は、empno、ename、sal および address です。
  - empno 列は、EmpBean で empNo と定義され、主キーが格納されます。
  - CMP データの empName および salary は、それぞれ ename 列と sal 列に保存されます。
  - address 列は、EmpBean 表である EMP の外部キーで、AddressBean 表の主キーを指します。
- 外部キーの <cmp-field-mapping> 要素で定義する内容は、次のとおりです。
  - name 属性は両方とも、ejb-jar.xml ファイルで定義されている <cmr-field> を識別します。この名前は address です。
  - <entity-ref> home 属性は、ターゲット Bean の <ejb-name> を識別します。この例では、ターゲットは AddressBean です。
  - persistence-name 属性は、ターゲット Bean の主キー列名を識別します。この例では、AddressBean 表、ADDRESS の主キーは、addressPK 列です。

図 4-8 は、EmpBean address 外部キーから AddressBean addressPK 主キーへの関連のマッピングを示します。

図 4-8 1 対 1 関連の明示的なマッピングの例



要約すると、EMP 表の address 列は、ADDRESS 表の主キー addressPK を指す外部キーです。自動生成された主キーが AddressBean にある場合、EMP 表の address 列は、ADDRESS 表の主キー autoid を指す外部キーです。

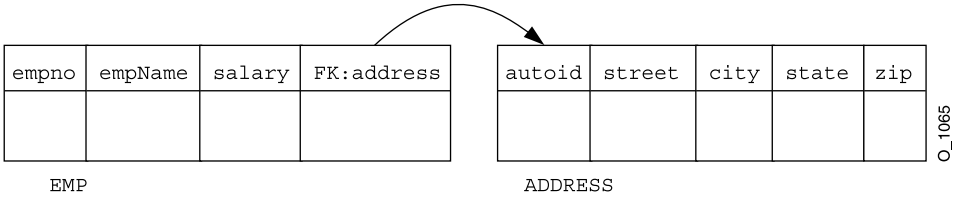
autoid を主キーとして使用する表のマッピング

3-12 ページの「自動生成された主キーの定義」で説明したように、自動生成された識別子を主キーとして表で使用するように定義できます。これによって、Bean の orion-ejb-jar.xml ファイルの XML 構成は次のようになります。

```
<primkey-mapping>
  <cmp-field-mapping name="auto_id"
                                    persistence-name="autoid"/>
</primkey-mapping>
```

employee/address の例で、AddressBean に未定義の主キーがある場合、その主キーは autoid にデフォルト設定され、表のマッピングは次のようになります。

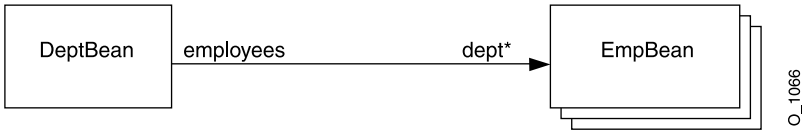
図 4-9 autoid を使用する employee の例での 1 対 1 関連



外部キーを使用する 1 対多関連

4-3 ページの「1 対多関連または多対 1 関連の概要」で説明したように、1 つの Bean が別の Bean の複数のインスタンスと関連を持つことができます。たとえば、1 つの部門に複数の従業員を関連付けることができます。この場合、各部門に複数の従業員が所属します。この関連は双方向であるため、従業員から部門を参照できます。`DeptBean` と `EmpBean` の間の関連は、図 4-10 に示すように、CMR フィールドの `employees` および `deptno` で表します。

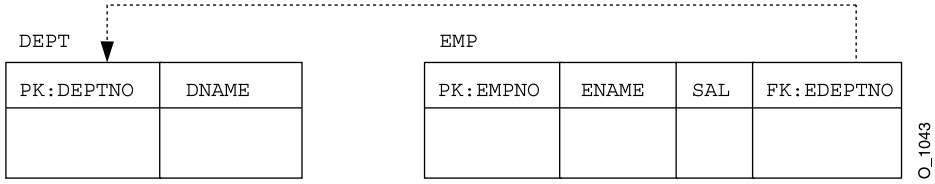
図 4-10 Bean の 1 対多関連



この関連をデータベース表にマッピングする方法は、開発者が選択します。デフォルトの方法では、外部キーを関連の「多」の側を定義する表に追加します。この例では、`EmpBean` を表す表です。外部キーは、各従業員が所属する部門を指します。

図 4-11 に、department<—>employee の例を示します。この例では、各従業員が 1 つの部門にのみ所属し、各部門には複数の従業員が所属している可能性があります。department 表には主キーがあります。employee 表には、各従業員を識別するための主キーと、従業員の部門を指すための外部キーがあります。特定の従業員の部門を検索する場合は、単純な SQL 文を使用して、外部キーから部門情報を取得できます。部門内の全従業員を検索する場合、コンテナは、department 表と employee 表の両方で JOIN 文を実行し、指定した部門番号を持つ全従業員を取得します。

図 4-11 1 対多の双方向の関連の明示的なマッピング例



これはデフォルトの動作です。マッピングを他のデータベース表に変更する必要がある場合は、JDeveloper を使用するか、または orion-ejb-jar.xml ファイルを手動で編集して、<collection-mapping> または <set-mapping> 要素を操作します。

重要： 関連フィールドを明示的にマッピングするには、orion-ejb-jar.xml ファイルの <entity-deployment> 要素の要素と属性を変更してください。**JDeveloper は、Entity Bean とデータベース表との間の複雑なマッピングを管理するために作成されています。JDeveloper ではデプロイメント・ディスクリプタが検証され、不一致が防止されます。orion-ejb-jar.xml ファイルは手動で変更することもできますが、コンテナ管理の関連を変更する場合は、JDeveloper を使用することをお薦めします。**これは、CMR の構成が複雑で、理解するのが困難なためです。JDeveloper は次のサイトからダウンロードできます。
http://otn.oracle.co.jp/software/products/jdev/index.html

例 4-6 は、1 つの部門に複数の従業員が所属する双方向の関連の表のマッピングを示します。関連の「1」の側は部門で、「多」の側は従業員です。図 4-11 に表の設計を示します。この例では、この関連で外部キーを使用するために orion-ejb-jar.xml ファイルを手動で編集する方法を示します。

例 4-6 外部キーを使用する 1 対多関連

次に示すように、ejb-jar.xml <relationships> セクションでは、部門と従業員の双方向の関連を定義します。

```
<relationships>
    <ejb-relation>
        <ejb-relation-name>Dept-Emps</ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>Dept-has-Emps
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>DeptBean</ejb-name>
            </relationship-role-source>
```

```
            <cmr-field>
                <cmr-field-name>employees</cmr-field-name>
                <cmr-field-type>java.util.Set</cmr-field-type>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>Emps-have-Dept
            </ejb-relationship-role-name>
            <multiplicity>Many</multiplicity>
            <cascade-delete/>
            <relationship-role-source>
                <ejb-name>EmpBean</ejb-name>
            </relationship-role-source>
            <cmr-field><cmr-field-name>dept</cmr-field-name></cmr-field>
        </ejb-relationship-role>
    </ejb-relation>
</relationships>
```

次の XML で、orion-ejb-jar.xml ファイルはこの定義をマッピングします。関連の「1」の側（department）の <collection-mapping> または <set-mapping> 要素で識別された表が、ターゲット Bean の表（employee Bean の表）の名前である場合、この 1 対多関連は外部キーを使用して定義されます。たとえば、部門の定義では、table 属性は EMP です。

```
<?xml version = '1.0' encoding = 'windows-1252'?>
<!DOCTYPE orion-ejb-jar PUBLIC "-//Evermind//DTD Enterprise JavaBeans 2.0
runtime//EN" "
http://xmlns.oracle.com/ias/dtds/orion-ejb-jar.dtd">
<orion-ejb-jar>
  <enterprise-beans>
    <entity-deployment name="DeptBean" data-source="jdbc/scottDS" table="DEPT">
      <primkey-mapping>
        <cmp-field-mapping name="deptno" persistence-name="DEPTNO" />  /*PK*/
      </primkey-mapping>
       <cmp-field-mapping name="dname" persistence-name="DNAME" />
      <cmp-field-mapping name="employees">
        /*points from DEPTNO column in EMP to DEPTNO in DEPT*/
1.     <collection-mapping table="EMP"> /*table where FK lives*/
          <primkey-mapping>
           <cmp-field-mapping name="DeptBean_deptno"> /*CMR field name*/
              <entity-ref home="DeptBean"> /*points to DeptBean*/
2.                    <cmp-field-mapping name="DeptBean_deptno"
                     persistence-name="EDEPTNO"/>
              </entity-ref>
            </cmp-field-mapping>
          </primkey-mapping>
          <value-mapping type="mypackage1.EmpLocal">
            <cmp-field-mapping name="EmpBean_empno">
```

```
            <entity-ref home="EmpBean">
                <cmp-field-mapping name="EmpBean_empno"
                      persistence-name="EMPNO"/>
            </entity-ref>
          </cmp-field-mapping>
        </value-mapping>
      </collection-mapping>
    </cmp-field-mapping>
  </entity-deployment>
  <entity-deployment name="EmpBean" data-source="jdbc/scottDS" table="EMP">
    <primkey-mapping>
      <cmp-field-mapping name="empNo" persistence-name="EMPNO"/>
    </primkey-mapping>
    <cmp-field-mapping name="empName" persistence-name="ENAME" />
    <cmp-field-mapping name="salary" persistence-name="SAL" />
    <cmp-field-mapping name="dept"> /*foreign key*/
      <entity-ref home="DeptBean">
2.          <cmp-field-mapping name="dept" persistence-name="EDEPTNO" />
      </entity-ref>
    </cmp-field-mapping>
  </entity-deployment>
 </enterprise-beans>
 <assembly-descriptor>
   <default-method-access>
     <security-role-mapping impliesAll="true"
                 name="&lt;default-ejb-caller-role>"/>
   </default-method-access>
 </assembly-descriptor>
</orion-ejb-jar>
```

外部キーは、関連の「多」の側のデータベース表で定義されています。この例では、EDEPTNO 外部キー列が EMP データベース表に存在します。これは、EmpBean 構成の <cmp-field-mapping> 要素の persistence-name 属性で定義されています。

したがって、orion-ejb-jar.xml ファイルで <collection-mapping> または <set-mapping> 要素を操作するには、「1」の側の Entity Bean（Collection を含む）の <entity-deployment> 要素を次のように変更します。

1. 関連の「1」の側の <collection-mapping> または <set-mapping> table 属性で、表を「多」の側のデータベース表に変更します。この例では、この属性を EMP 表に変更します。
2. 関連の「多」の構成内で、「1」の側を指す外部キーを変更します。この例では、persistence-name 属性で EDEPTNO 外部キーを指定するように、<cmp-field-mapping> 要素を変更します。

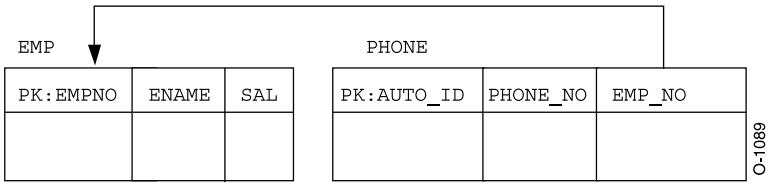
これらの手順は、例 4-6 のコード例に示されています。

外部キーを使用する単方向の 1 対多関連 単方向の 1 対多関連の例には、従業員と電話番号の関連があります。各従業員は 1 つ以上の電話番号を持つことができます。ただし、特定の電話番号から従業員を参照することはできません。図 4-12 は、Bean の関連を示します。

図 4-12 Bean の 1 対多関連

図 4-13 に、従業員ー > 電話番号の例を示します。各従業員は複数の電話番号を持つことができます。employee 表には主キーがあります。phone numbers 表には、主キーの autoid、電話番号、および従業員を指すための外部キーがあります。ある従業員のすべての電話番号を検索する場合、コンテナは、employee 表と phone number 表の両方で JOIN 文を実行し、指定した従業員番号を持つすべての電話番号を取得します。

図 4-13 1 対多の双方向の関連の明示的なマッピング例



例 4-7 外部キーを使用する単方向の 1 対多関連の例

次に示すように、`ejb-jar.xml` `<relationships>` セクションでは、従業員と電話番号の双方向の関連を定義します。

```
<entity>
   <ejb-name>EmpBean</ejb-name>
   ...
   <cmp-field><field-name>empNo</field-name></cmp-field>
   <cmp-field><field-name>empName</field-name></cmp-field>
   <cmp-field><field-name>salary</field-name></cmp-field>
   <primkey-field>empNo</primkey-field>
   <prim-key-class>java.lang.Integer</prim-key-class>
</entity>
<entity>
   <ejb-name>PhoneBean</ejb-name>
   ...
   <cmp-field><field-name>phoneNo</field-name></cmp-field>
```

```
   <prim-key-class>java.lang.Object</prim-key-class>
</entity>

<relationships>
   <ejb-relation>
      <ejb-relation-name>Emp-Phone</ejb-relation-name>
      <ejb-relationship-role>
         <ejb-relationship-role-name>Emp-has-Phones
         </ejb-relationship-role-name>
         <multiplicity>One</multiplicity>
         <relationship-role-source>
            <ejb-name>EmpBean</ejb-name>
         </relationship-role-source>
         <cmr-field>
            <cmr-field-name>phones</cmr-field-name>
         </cmr-field>
      </ejb-relationship-role>
      <ejb-relationship-role>
         <ejb-relationship-role-name>Phones-have-Emp
         </ejb-relationship-role-name>
         <multiplicity>Many</multiplicity>
         <cascade-delete/>
         <relationship-role-source>
            <ejb-name>PhoneBean</ejb-name>
         </relationship-role-source>
      </ejb-relationship-role>
   </ejb-relation>
</relationships>
```

次の XML で、orion-ejb-jar.xml ファイルはこの定義をマッピングします。関連の「1」の側（employee）の <collection-mapping> または <set-mapping> 要素で識別された表が、ターゲット Bean の表（phone Bean の表）の名前である場合、コンテナではこの 1 対多関連を外部キーを使用して定義します。この例では、ターゲット Bean の表は PHONE データベース表です。

```
<entity-deployment name="EmpBean"  table="EMP">
    <primkey-mapping>
        <cmp-field-mapping name="empNo" persistence-name="EMPNO" />
    </primkey-mapping>
    <cmp-field-mapping name="empName" persistence-name="ENAME" />
    <cmp-field-mapping name="salary" persistence-name="SAL" />
    <cmp-field-mapping name="phones">
1.      <collection-mapping table="PHONE">
            <primkey-mapping>
                <cmp-field-mapping name="EmpBean_empno">
                    <entity-ref home="EmpBean">
2.                      <cmp-field-mapping name="EmpBean_empNo"
```

```
                        persistence-name="EMPNO"/>
                    </entity-ref>
                </cmp-field-mapping>
            </primkey-mapping>
            <value-mapping type="hr.PhoneLocal">
                <cmp-field-mapping name="autoid">
                    <entity-ref home="PhoneBean">
                        <cmp-field-mapping name="autoid"
                        persistence-name="AUTOID"/>
                    </entity-ref>
                </cmp-field-mapping>
            </value-mapping>
        </collection-mapping>
    </cmp-field-mapping>
</entity-deployment>
<entity-deployment name="PhoneBean"  table="PHONE">
    <primkey-mapping>
        <cmp-field-mapping name="autoid" persistence-name="AUTOID"/>
    </primkey-mapping>
    <cmp-field-mapping name="phoneNo" persistence-name="PHONE_NO" />
    <cmp-field-mapping name="EmpBean_phones">
        <entity-ref home="EmpBean">
2.          <cmp-field-mapping name="EmpBean_phones" persistence-name="EMPNO" />
        </entity-ref>
    </cmp-field-mapping>
</entity-deployment>
```

外部キーは、関連の「多」の側のデータベース表で定義されています。この例では、EMPNO 外部キー列が PHONE データベース表に存在します。これは、PhoneBean 構成の <cmp-field-mapping> 要素の persistence-name 属性で定義されています。

したがって、orion-ejb-jar.xml ファイルで <collection-mapping> または <set-mapping> 要素を操作するには、「1」の側の Entity Bean（Collection を含む）の <entity-deployment> 要素を次のように変更します。

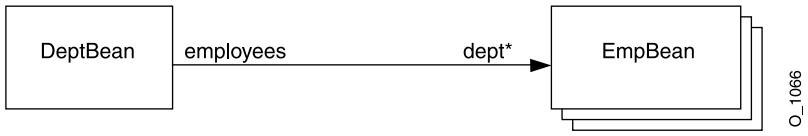
1. 関連の「1」の側の <collection-mapping> または <set-mapping> table 属性で、表を「多」の側のデータベース表に変更します。この例では、この属性を PHONE 表に変更します。
2. 関連の「多」の構成内で、「1」の側を指す外部キーを変更します。この例では、persistence-name 属性で EMPNO 外部キーを指定するように、<cmp-field-mapping> 要素を変更します。

これらの手順は、例 4-7 のコード例に示されています。

関連に対する関連表の明示的なマッピングの概要

4-3 ページの「1 対多関連または多対 1 関連の概要」で説明したように、1 つの Bean が別の Bean の複数のインスタンスと関連を持つことができます。たとえば、1 つの部門に複数の従業員を関連付けることができます。この場合、各部門に複数の従業員が所属します。この関連は双方向であるため、従業員から部門を参照できます。`DeptBean` と `EmpBean` の間の関連は、図 4-14 に示すように、CMR フィールドの `employees` および `deptno` で表します。

図 4-14 Bean の双方向の 1 対多関連



この関連をデータベース表にマッピングする方法は、開発者が選択します。関連表と呼ばれる別の表を使用して、2 つの表をまとめて適切にマッピングすることもできます。関連表では 2 つの外部キーを使用し、各外部キーは関連内の各エンティティ表を指します。

> **注意：** 一方または両方の表にコンポジット主キーがある場合、外部キーはコンポジット外部キーになります。したがって、関連表には、コンポジット外部キーの各部について適切な数の列があります。

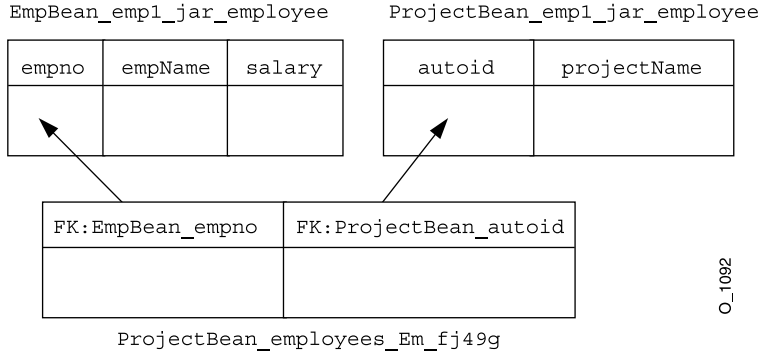
これはデフォルトの動作ではありません。このような関連を設定するには、次のどちらか、または両方を実行します。

- デプロイを行う前に、OC4J 起動オプションで `-DassociateUsingThirdTable=true` を指定します。次に、OC4J インスタンスを再起動します。これによって、再起動後にデプロイされるすべてのアプリケーションに対して関連表が生成されます。
- JDeveloper を使用するか、または `orion-ejb-jar.xml` ファイルを手動で編集して、マッピングを変更できます。

重要： 関連フィールドを明示的にマッピングするには、orion-ejb-jar.xml ファイルの <entity-deployment> 要素の要素と属性を変更してください。**JDeveloper は、Entity Bean とデータベース表との間の複雑なマッピングを管理するために作成されています。JDeveloper ではデプロイメント・ディスクリプタが検証され、不一致が防止されます。orion-ejb-jar.xml ファイルは手動で変更することもできますが、コンテナ管理の関連を変更する場合は、JDeveloper を使用することをお薦めします。**これは、CMR の構成が複雑で、理解するのが困難なためです。JDeveloper は次のサイトからダウンロードできます。
http://otn.oracle.co.jp/software/products/jdev/index.html

図 4-15 に、従業員とプロジェクトの関連について作成された表を示します。

図 4-15 employee の例での多対多関連



各プロジェクトには複数の従業員が所属し、各従業員は複数のプロジェクトに所属している可能性があります。したがって、従業員とプロジェクトは多対多の関連になります。コンテナは、この関連を管理するために、employee 表、project 表、および両方の表に対する関連表を作成します。

この例の関連表には、employee 表を指す外部キー列および project 表を指す外部キー列があります。関連表の列名は、ejb-jar.xml ファイルの <ejb-name> 要素内の Entity Bean 名とその主キー名を連結した名前になります。Bean の主キーが自動生成される場合は、主キー名として autoid が使用されます。たとえば、従業員とプロジェクトの例では、外部キーの名前は次のようになります。

- employee 表を指す外部キーは、Bean 名の EmpBean の後に主キー名の empno が連結されて、EmpBean_empno という列名になります。
- project 表を指す外部キーは、Bean 名の ProjectBean の後に autoid（主キーは自動生成のため）が連結されて、ProjectBean_autoid という列名になります。

次に、従業員とプロジェクトの関連を定義するための関連表を示します。従業員 1 はプロジェクト a、b および c に割り当てられています。プロジェクト a には従業員 1、2 および 3 が所属しています。この場合、関連表は次のようになります。

EmpBean_empno	ProjectBean_autoid
1	a
1	b
1	c
2	a
3	a

関連表は、2 つの Entity Bean 間のすべての関連を示します。

例 4-8 多対多関連のデプロイメント・ディスクリプタ

従業員とプロジェクトの多対多関連のデプロイメント・ディスクリプタには、<ejb-relation> 要素を含めます。この要素内で、各 Bean について <multiplicity> を Many に定義し、<cmr-field> を他の Bean の Collection 型または Set 型に定義します。

```
<enterprise-beans>
  <entity>
    ...
    <ejb-name>EmpBean</ejb-name>
    <local-home>employee.EmpHome</local-home>
    <local>employee.Emp</local>
    ...
    <cmp-field><field-name>empNo</field-name></cmp-field>
    <cmp-field><field-name>empName</field-name></cmp-field>
    <cmp-field><field-name>salary</field-name></cmp-field>
    <primkey-field>empNo</primkey-field>
    <prim-key-class>java.lang.Integer</prim-key-class>
    ...
  </entity>
  <entity>
    ...
    <ejb-name>ProjectBean</ejb-name>
    <local-home>employee.ProjectHome</local-home>
    <local>employee.Project</local>
    ...
    <cmp-field><field-name>projectName</field-name></cmp-field>
    <prim-key-class>java.lang.Object</prim-key-class>
    ...
```

```
  </entity>
</enterprise-beans>
<relationships>
  <ejb-relation>
    <ejb-relation-name>Emps-Projects</ejb-relation-name>
    <ejb-relationship-role>
      <ejb-relationship-role-name>Project-has-Emps</ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
        <ejb-name>ProjectBean</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>employees</cmr-field-name>
        <cmr-field-type>java.util.Collection</cmr-field-type>
      </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
      <ejb-relationship-role-name>Emp-has-Projects</ejb-relationship-role-name>
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
        <ejb-name>EmpBean</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>projects</cmr-field-name>
        <cmr-field-type>java.util.Collection</cmr-field-type>
      </cmr-field>
    </ejb-relationship-role>
  </ejb-relation>
</relationships>
```

コンテナは、この定義を次のようにマッピングします。

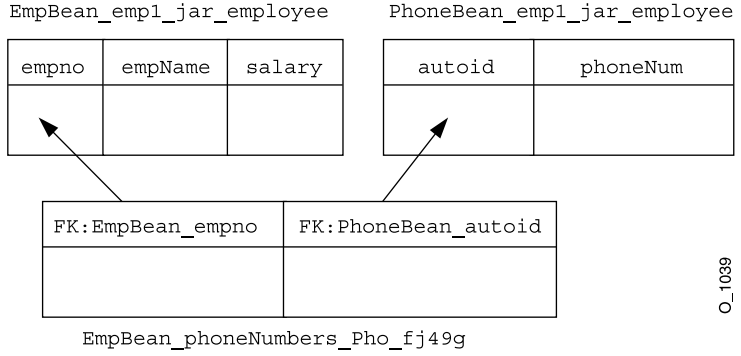
- コンテナは、Entity Bean 名、Bean のアーカイブ先 JAR ファイル、および Bean がデプロイされるアプリケーションの名前に基づいて、エンティティ表を生成します。JAR ファイル名が `empl.jar`、アプリケーション名が `employee` の場合、表名は表名 `EmpBean_empl_jar_employee` および `ProjectBean_empl_jar_employee` になります。
- コンテナは、デプロイメント・ディスクリプタで宣言された `<cmp-field>` 要素に基づいて、各エンティティ表に列を生成します。
  - `EmpBean` 表の列は `empno`、`empname` および `salary` です。主キーは `empno` フィールドとして指定されます。
  - `ProjectBean` 表の列は、自動生成された主キーの `autoid`、および `projectName` 列になります。`<prim-key-class>` が `java.lang.Object` として定義されているため、主キーは自動生成され、`<primkey-field>` 要素は定義されません。

- コンテナは、エンティティ表と同じ方法で関連表を生成します。
  - 関連表の名前は、関連内の各 Entity Bean の 2 つの <cmr-field> 定義から作成されます。関連表の名前は、次のように、1 番目の Bean 名、2 番目の Bean に対する <cmr-field>、2 番目の Bean 名、1 番目の Bean に対する <cmr-field>、JAR ファイル名およびアプリケーション名をアンダースコアで区切って構成されます。エンティティ表の場合と同様に、名前が 31 文字以上の場合のルールがこの表名にも適用されます。たとえば、従業員とプロジェクトの関連で使用する関連表の名前は、表名 ProjectBean_employees_EmpBean_projects_empl_jar_employee となります。この名前は 31 文字以上であるため、24 文字で切り捨てられ、アンダースコアと 5 文字のハッシュ・コードが追加されます。したがって、正式な関連表の名前は、表名 ProjectBean_employees_Em_fj49g のようになります。
  - 関連表には、2 つの外部キーが作成されます。この例の場合、各外部キーは列で定義され、各列の名前は、Bean 名と主キー（自動生成の場合は autoid）を連結した名前になります。前述の例では、列名は EmpBean_empno および ProjectBean_autoid になります。これらの列は、関連に含まれるエンティティ表に対する外部キーです。EmpBean_empno 外部キーは employee 表を指し、ProjectBean_autoid 外部キーは projects 表を指します。

例 4-9 1 対多の単方向の関連のデプロイメント・ディスクリプタ

図 4-16 に、従業員とプロジェクトの関連の例で使用するデフォルトのデータベース表を示します。

図 4-16 Employee の例での 1 対多関連



各従業員は、複数の電話番号を持つことができます。従業員の Entity Bean の EmpBean では、PhoneBean 内の phoneNumbers の Collection を指定して、<cmr-field> 要素が定義されます。この例のデプロイメント・ディスクリプタは、次のようになります。

```
<relationships>
  <ejb-relation>
   <ejb-relation-name>Emp-Phone</ejb-relation-name>
   <ejb-relationship-role>
     <ejb-relationship-role-name>Emp-PhoneNumbers</ejb-relationship-role-name>
     <multiplicity>One</multiplicity>
     <relationship-role-source>
        <ejb-name>EmpBean</ejb-name>
     </relationship-role-source>
     <cmr-field>
        <cmr-field-name>phoneNumbers</cmr-field-name>
        <cmr-field-type>java.util.Collection</cmr-field-type>
     </cmr-field>
   </ejb-relationship-role>
   <ejb-relationship-role>
     <ejb-relationship-role-name>Phone-has-Emp</ejb-relationship-role-name>
     <multiplicity>Many</multiplicity>
     <relationship-role-source>
        <ejb-name>PhoneBean</ejb-name>
     </relationship-role-source>
    </ejb-relationship-role>
  </ejb-relation>
</relationships>
```

注意： オブジェクト関連の Entity Bean の例は、OTN-J のサイト http://otn.oracle.co.jp/sample_code/index.html の OC4J のサンプル・コードのページから入手できます。

---

1 対多関連のマッピングの XML 構造

ejb-jar.xml ファイルで定義された関連は、orion-ejb-jar.xml ファイルの <cmp-field-mapping> 要素内でマッピングされます。<cmp-field-mapping> 要素には、<collection-mapping> または <set-mapping> 要素が含まれます。前述の例では、1 つの部門に複数の従業員が含まれます。部門では、<collection-mapping> 要素を使用して、従業員に対する「多」の関連を記述します。

注意： 関連の「多」の側は、<collection-mapping> または <set-mapping> 要素で定義します。関連の「1」の側は、<entity-ref> 要素で定義します。したがって、1 対多関連では、1 つの <collection-mapping> を使用して「多」の側を記述します。

---

1 対多関連を定義するための XML 構造には、次の要素および属性が含まれます。

```
<cmp-field-mapping name="CMRfield">
    <collection-mapping table="association_table">
      <primkey-mapping>
         <cmp-field-mapping name="CMRfield"
persistence-name="first_column_name_assoc_table" />
      </primkey-mapping>
      <value-mapping type="target_bean_local_home_interface">
        <cmp-field-mapping>
          <entity-ref home="target_bean_EJBname">
            <cmp-field-mapping name="CMRfield"
            persistence-name="second_column_name_assoc_table"/>
          </entity-ref>
        </cmp-field-mapping>
      </value-mapping>
    </collection-mapping>
</cmp-field-mapping>
```

要素または属性	説明
<cmp-field-mapping>	この要素は、永続フィールドまたは関連フィールドをマッピングします。関連フィールドには、<entity-ref>（1 対 1 のマッピングの場合）または <collection-mapping>（1 対多、多対 1 または多対多の関連の場合）が含まれます。 ■ name 属性は、マッピングされる <cmp-field> または <cmr-field> を識別します。この名前は変更しないでください。 ■ persistence-name 属性はデータベース列を識別します。**この名前は、データベース列名と一致するように変更できます。**
<entity-ref>	この要素は、ターゲット Bean と、外部キーが指す主キーを識別します。 ■ home 属性はホーム・インタフェースではなく、ターゲット Bean の EJB 名を識別します。この名前は、ejb-jar.xml ファイルの <ejb-name> で定義された Bean の論理名です。 ■ この要素内の <cmp-field-mapping> は、外部キーの列名を識別します。

要素または属性	説明
<collection-mapping>	この要素は、関連の「多」の側を明示的にマッピングします。 ■ table 属性は、関連表を識別します。**この名前は、使用する関連表の名前と一致するように変更できます。** この要素は、関連表の各列に対する 2 つの要素を定義します。 ■ <primkey-mapping> は、関連表の 1 番目の外部キーを識別します。 ■ <value-mapping> は、関連表の 2 番目の外部キーを識別します。
<primkey-mapping>	<collection-mapping> 内でこの要素を使用して、1 番目の外部キーを識別します。**この要素内の persistence-name 属性は、使用する関連表の列名と一致するように変更できます。**
<value-mapping>	この要素を使用して、2 番目の外部キーを指定します。type 属性は、ターゲット Bean のローカル・インタフェースを識別します。**この要素内の persistence-name 属性は、使用する関連表の列名と一致するように変更できます。**

1 対多の双方向の関連における関連表の使用

次の例では、関連表を使用するように 1 対多の双方向の関連を構成する方法を示します。部門は、ejb-jar.xml ファイルで関連の「1」の側として定義され、<cmr-field> 要素内の Collection の定義によって「複数」の従業員を受け取ることが示されます。従業員は、関連の「多」の側として定義されます。

```
<relationships>
   <ejb-relation>
       <ejb-relation-name>Dept-Emps</ejb-relation-name>
       <ejb-relationship-role>
          <ejb-relationship-role-name>Dept-has-Emps
          </ejb-relationship-role-name>
          <multiplicity>One</multiplicity>
          <relationship-role-source>
             <ejb-name>DeptBean</ejb-name>
          </relationship-role-source>
          <cmr-field>
             <cmr-field-name>employees</cmr-field-name>
             <cmr-field-type>java.util.Collection</cmr-field-type>
          </cmr-field>
       </ejb-relationship-role>
       <ejb-relationship-role>
          <ejb-relationship-role-name>Emp-has-Dept
          </ejb-relationship-role-name>
          <multiplicity>Many</multiplicity>
```

```
            <relationship-role-source>
               <ejb-name>EmpBean</ejb-name>
            </relationship-role-source>
            <cmr-field>
               <cmr-field-name>dept</cmr-field-name>
            </cmr-field>
         </ejb-relationship-role>
   </ejb-relation>
</relationships>
```

この関連の関連表へのマッピングは、orion-ejb-jar.xml ファイル内の <collection-mapping> 要素で記述します。これは 1 対多関連であるため、「1」の側の Entity Bean（部門）で <collection-mapping> 要素を定義します。これは、ターゲット（従業員）の Collection または Set を受け取るためです。

orion-ejb-jar.xml ファイルでは、DeptBean の <entity-deployment> 要素で <collection-mapping> 要素を定義し、従業員の Collection を指定します。<collection-mapping> 要素で関連表を定義します。

```
<entity-deployment name="DeptBean" location="DeptBean"
    table="DEPT" data-source="jdbc/OracleDS" ... >
    <primkey-mapping>
           <cmp-field-mapping name="deptNo" persistence-name="deptNo" />
    </primkey-mapping>
    <cmp-field-mapping name="deptName" persistence-name="deptName" />
    <cmp-field-mapping name="employees">
           <collection-mapping table="DEPT_EMP">
               <primkey-mapping>
                   <cmp-field-mapping name="DeptBean_deptno">
                       <entity-ref home="DeptBean">
                           <cmp-field-mapping name="DeptBean_deptno"
                              persistence-name="DEPARTMENT" />
                       </entity-ref>
                   </cmp-field-mapping>
               </primkey-mapping>
               <value-mapping type="hr.EmpLocal">
                   <cmp-field-mapping name="EmpBean_empNo">
                       <entity-ref home="EmpBean">
                           <cmp-field-mapping name="EmpBean_empNo"
                              persistence-name="EMPLOYEE" />
                       </entity-ref>
                   </cmp-field-mapping>
               </value-mapping>
           </collection-mapping>
    </cmp-field-mapping>
...
</entity-deployment>
<entity-deployment name="EmpBean" location="EmpBean"
```

```
    table="EMP" data-source="jdbc/OracleDS" ... >
    <primkey-mapping>
        <cmp-field-mapping name="empNo" persistence-name="EMPNO" />
    </primkey-mapping>
    <cmp-field-mapping name="empName" persistence-name="ENAME" />
    <cmp-field-mapping name="salary" persistence-name="SAL" />
    <cmp-field-mapping name="dept">
        <entity-ref home="DeptBean">
            <cmp-field-mapping name="dept" persistence-name="DEPARTMENT" />
        </entity-ref>
    </cmp-field-mapping>
...
</entity-deployment>
```

次に、orion-ejb-jar.xml ファイルで DeptBean を構成する方法を説明します。

- department Bean から employee Bean への関連は、employees フィールドで定義します。このフィールドは <collection-mapping> 要素内でマッピングされます。
- 関連表の名前は table 属性で指定します。この例では、関連表の名前は DEPT_EMP に定義されています。
- 関連表の外部キーは、次のように定義します。
  - <primkey-mapping> 要素の persistence-name 属性で、現行の Entity Bean の外部キーの列名を定義します。この例では DEPARTMENT に定義されています。
  - <value-mapping> 要素の persistence-name 属性で、現行のターゲット Bean の外部キーの列名を定義します。この例では EMPLOYEE に定義されています。
- <value-mapping> 要素では、ターゲット Entity Bean を指定します。
  - <value-mapping> 要素の type 属性では、ソース Entity Bean に返されるターゲット Bean のローカル・インタフェースを定義します。
  - ターゲット Entity Bean の <ejb-name> は、<entity-ref> home 属性で定義します。

次に、orion-ejb-jar.xml ファイルで EmpBean を構成する方法を説明します。

- employee Bean から department Bean への関連は、dept フィールドで定義します。このフィールドは <cmp-field-mapping><entity-ref> 要素内でマッピングされます。persistence-name 属性には、department Bean を指す、関連表の外部キーが格納されます。

1 対多の単方向の関連における関連表の使用

4-3 ページの「1 対多関連または多対 1 関連の概要」で説明したように、1 つの Bean は別の Bean の複数のインスタンスと関連を持つことができます。たとえば、1 名の従業員に複数の電話番号を関連付けることができます。各従業員に対して、複数の電話番号を指定できます。ただし、これは単方向の関連です。特定の電話番号から従業員を参照することはできません。

`EmpBean` と `PhoneBean` の間の関連は、図 4-17 に示すように、CMR フィールドの `phones` で表します。

図 4-17 Bean の単方向の 1 対多関連

関連は、関連表を使用してデータベース表にマッピングされます。この関連表によって 2 つの表がまとめて適切にマッピングされます。関連表は、2 つの外部キーで構成されます。

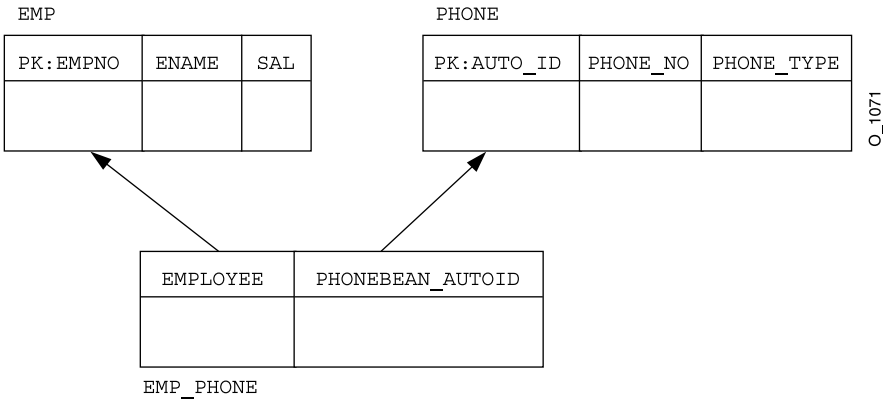
> **注意：** 一方または両方の表にコンポジット主キーがある場合、外部キーはコンポジット外部キーになります。したがって、関連表には、コンポジット外部キーの各部について適切な数の列があります。

関連表の使用方法の詳細は、4-15 ページの「1 対多関連および多対多関連のデフォルト・マッピング例」を参照してください。この項では、このマッピングを行うために XML 構成を変更する方法を説明します。

> **注意：** 関連表を使用しない場合、関連の「1」の側で外部キーを使用する方法については、4-28 ページの「外部キーを使用する 1 対多関連」を参照してください。

図 4-18 に、従業員ー > 電話番号の例を示します。各従業員は複数の電話番号を持つことができます。employee 表と phone 表の両方に 1 つの主キーがあります。関連表と呼ばれる別の表には、2 つの外部キーがあります。1 つの外部キーは従業員を指し、もう 1 つの外部キーは電話番号を指します。すべての関連には、関連を示す独自の行があります。したがって、各電話番号に対して 1 行が作成されます。1 番目の外部キーは電話番号を所有する従業員を指し、2 番目の外部キーは電話番号レコードを指します。図 4-18 に、`EMPLOYEE` と `PHONEBEAN_AUTOID` という名前の外部キーを持つ関連表 `EMP_PHONE` を示します。

図 4-18 1 対多の単方向の関連の明示的なマッピング例



マッピングを他のデータベース表に変更するには、JDeveloper を使用するか、または orion-ejb-jar.xml ファイルを手動で編集して、<collection-mapping> または <set-mapping> 要素を操作します。

重要： 関連フィールドを明示的にマッピングするには、orion-ejb-jar.xml ファイルの <entity-deployment> 要素の要素と属性を変更してください。**JDeveloper は、Entity Bean とデータベース表との間の複雑なマッピングを管理するために作成されています。JDeveloper ではデプロイメント・ディスクリプタが検証され、不一致が防止されます。orion-ejb-jar.xml ファイルは手動で変更することもできますが、コンテナ管理の関連を変更する場合は、JDeveloper を使用することをお薦めします。**これは、CMR の構成が複雑で、理解するのが困難なためです。JDeveloper は次のサイトからダウンロードできます。
http://otn.oracle.co.jp/software/products/jdev/index.html

---

1 対多の単方向の関連を XML デプロイメント・ディスクリプタで指定 カーディナリティは、ejb-jar.xml ファイル内の <relationships> 要素で定義します。次の ejb-jar.xml ファイルは、従業員とその電話番号の 1 対多の単方向の関連を構成しています。

- EmpBean の主キー・フィールドは empNo で、<primkey-field> 要素で定義されています。
- PhoneBean の主キーは定義されていません。<primkey-field> 要素がなく、<prim-key-class> 要素が定義されています。したがって、主キーは自動生成され、AUTOID として表されます。自動生成される主キーの詳細は、3-12 ページの「自動生成された主キーの定義」を参照してください。

- 関連の「多」の側を定義する CMR フィールド（<cmr-field> 要素）は Collection で、phones として識別されます。

```
<entity>
     <ejb-name>EmpBean</ejb-name>
     ...
     <cmp-field><field-name>empNo</field-name></cmp-field>
     <cmp-field><field-name>empName</field-name></cmp-field>
     <cmp-field><field-name>salary</field-name></cmp-field>
     ...
     <primkey-field>empNo</primkey-field>
     <prim-key-class>java.lang.Integer</prim-key-class>
     ...
</entity>
<entity>
     ...
     <ejb-name>PhoneBean</ejb-name>
     ...
     <cmp-field><field-name>phoneNo</field-name></cmp-field>
     <cmp-field><field-name>phoneType</field-name></cmp-field>
     <prim-key-class>java.lang.Object</prim-key-class>
     ...
</entity>
<relationships>
  <ejb-relation>
   <ejb-relation-name>Emp-Phone</ejb-relation-name>
   <ejb-relationship-role>
     <ejb-relationship-role-name>Emp-PhoneNumbers</ejb-relationship-role-name>
     <multiplicity>One</multiplicity>
     <relationship-role-source>
        <ejb-name>EmpBean</ejb-name>
     </relationship-role-source>
     <cmr-field>
        <cmr-field-name>phones</cmr-field-name>
        <cmr-field-type>java.util.Collection</cmr-field-type>
     </cmr-field>
   </ejb-relationship-role>
   <ejb-relationship-role>
     <ejb-relationship-role-name>Phone-has-Emp</ejb-relationship-role-name>
     <multiplicity>Many</multiplicity>
     <relationship-role-source>
        <ejb-name>PhoneBean</ejb-name>
     </relationship-role-source>
    </ejb-relationship-role>
  </ejb-relation>
</relationships>
```

この関連の関連表へのマッピングは、orion-ejb-jar.xml ファイル内の <collection-mapping> 要素で記述します。関連の「1」の側（従業員）は「多」の側のエンティティ（電話番号）を所有します。したがって、従業員の側では、電話番号との関連を記述する <collection-mapping> 要素を定義します。すべての 1 対多関連では、関連の「1」の側を表す Entity Bean で <collection-mapping> 要素を定義します。これは、ターゲット Entity Bean の Collection または Set を受け取るためです。関連の「多」の側の Entity Bean では、<cmp-field-mapping> の <entity-ref> 要素を定義して、関連の「1」の側の Entity Bean を指す関連を示します。したがって、従業員の側では、<collection-mapping> 要素を定義して電話番号との関連を定義します。電話番号の側では、<entity-ref> 要素を使用して従業員との関連を定義します。

従業員と電話番号の例の orion-ejb-jar.xml ファイルでは、EmpBean の <entity-deployment> 要素内で <collection-mapping> 要素を定義して、電話番号の Collection を指定します。<collection-mapping> 要素で関連表を指定します。

```
<entity-deployment name="EmpBean"  table="EMP">
    <primkey-mapping>
        <cmp-field-mapping name="empNo" persistence-name="EMPLOYEENO" />
    </primkey-mapping>
    <cmp-field-mapping name="empName" persistence-name="EMPLOYEENAME" />
    <cmp-field-mapping name="salary" persistence-name="SAL" />
    <cmp-field-mapping name="phones">
        <collection-mapping table="EMP_PHONE">
            <primkey-mapping>
                <cmp-field-mapping name="EmpBean_empNo">
                    <entity-ref home="EmpBean">
                        <cmp-field-mapping name="EmpBean_empNo"
                            persistence-name="EMPLOYEENO"/>
                    </entity-ref>
                </cmp-field-mapping>
            </primkey-mapping>
            <value-mapping type="hr.PhoneLocal">
                <cmp-field-mapping name="PhoneBean_autoid">
                    <entity-ref home="PhoneBean">
                        <cmp-field-mapping name="PhoneBean_autoid"
                            persistence-name="AUTOID"/>
                    </entity-ref>
                </cmp-field-mapping>
            </value-mapping>
        </collection-mapping>
...
</entity-deployment>
<entity-deployment name="PhoneBean"  table="PHONE">
    <primkey-mapping>
        <cmp-field-mapping name="autoid" persistence-name="AUTOID"/>
    </primkey-mapping>
    <cmp-field-mapping name="phoneNo" persistence-name="PHONE_NO" />
```

```
    <cmp-field-mapping name="phoneType" persistence-name="PHONE_TYPE" />
    <cmp-field-mapping name="EmpBean_phones">
        <entity-ref home="EmpBean">
            <cmp-field-mapping name="EmpBean_phones"
                persistence-name="EMPLOYEENO" />
        </entity-ref>
    </cmp-field-mapping>
</entity-deployment>
```

次に、ejb-jar.xml および orion-ejb-jar.xml ファイルで EmpBean を定義する方法を説明します。このマッピングの図は、図 4-19 を参照してください。

- ejb-jar.xml ファイルの <cmr-field> 要素内で、電話番号との関連の名前を phones と定義します。
- phones<cmr-field> 要素は、orion-ejb-jar.xml ファイルの関連表にマッピングされます。orion-ejb-jar.xml ファイルでは、phones の <cmp-field-mapping> に <collection-mapping> 要素が含まれます。この <collection-mapping> 要素の table 属性で、関連表の名前を EMP_PHONE と定義します。
- 関連表には、2 つの外部キーがあります。この例では、単純な外部キーを使用しています。ただし、主キーがコンポジット主キーの場合は、この 2 つの外部キーもコンポジット外部キーになります。

  関連表の外部キーは両方とも、次のように定義します。

  - <primkey-mapping> 要素の persistence-name 属性で、現行の Entity Bean の関連表の外部キーの列名を定義します。この例では EMPLOYEENO に定義されています。
  - <value-mapping> 要素の persistence-name 属性で、ターゲット Bean の関連表の外部キーの列名を定義します。この例では PhoneBean_AUTOID に定義されています。
- <value-mapping> 要素では、ターゲット Entity Bean を指定します。
  - <value-mapping> 要素の type 属性では、ソース Entity Bean に返されるターゲット Bean のローカル・インタフェースを定義します。この例では、phone Bean のローカル・ホーム・インタフェースが hr.PhoneLocal として定義されます。
  - ターゲット Entity Bean の <ejb-name> は、<entity-ref> home 属性で定義します。この例では PhoneBean に定義されています。

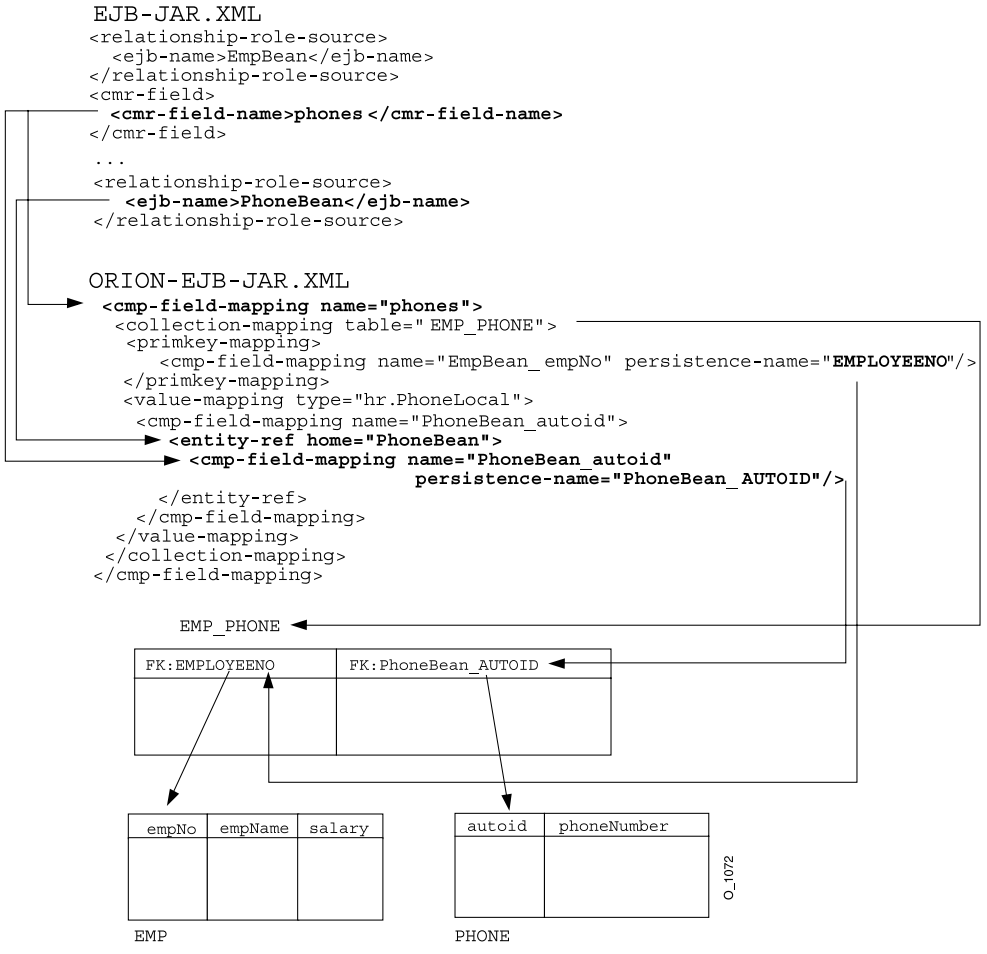
orion-ejb-jar.xml ファイルの phone Bean 構成内では、<entity-ref> で employee Bean への関連を定義します。

- ターゲット Entity Bean の <ejb-name> は、<entity-ref> home 属性で定義します。この例では EmpBean に定義されています。
- <cmp-field-mapping > 要素の persistence-name 属性で、現行の Entity Bean の関連表の外部キーを定義します。この例では EMPLOYEENO に定義されています。

図 4-19 は次の内容を示します。

- CMR フィールド名を orion-ejb-jar.xml ファイルの <cmp-field-mapping> 要素にマッピングする方法
- employee Bean 定義内で、関連表を <collection-mapping> 要素で定義する方法

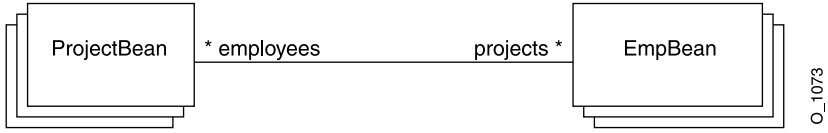
図 4-19　1 対多関連の明示的なマッピング



多対多関連における関連表の使用

4-3 ページの「多対多関連の概要」で説明したように、複数の Bean が別の Bean の複数のインスタンスと関連を持つことができます。たとえば、複数の従業員に複数のプロジェクトを関連付けることができます。各プロジェクトには複数の従業員が所属し、各従業員は複数のプロジェクトに割り当てられている可能性があります。この関連は双方向であるため、従業員からプロジェクトを参照できます。`ProjectBean` と `EmpBean` の間の関連は、図 4-20 に示すように、CMR フィールドの `employees` および `projects` で表します。

図 4-20 Bean の双方向の多対多関連



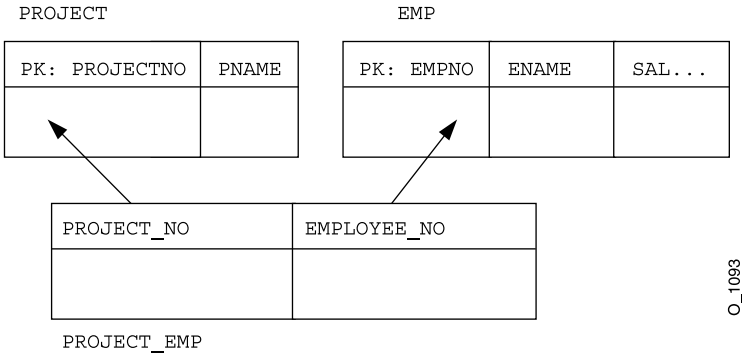
関連表を使用して、この関連をデータベース表にマッピングします。関連表は、2 つの外部キーで構成されます。

> **注意：** 一方または両方の表にコンポジット主キーがある場合、外部キーはコンポジット外部キーになります。したがって、関連表には、コンポジット外部キーの各部について適切な数の列があります。

関連表の使用方法の詳細は、4-15 ページの「1 対多関連および多対多関連のデフォルト・マッピング例」を参照してください。この項では、このマッピングを行うために XML 構成を変更する方法を説明します。

図 4-21 に、projects<—>employee の例を示します。この例では、各従業員が 1 つ以上のプロジェクトに所属し、各プロジェクトには複数の従業員が所属している可能性があります。project 表と employee 表の両方に 1 つの主キーがあります。関連表と呼ばれる別の表には、2 つの外部キーがあります。1 つの外部キーはプロジェクトを指し、もう 1 つの外部キーは従業員を指します。すべての関連には、関連を示す独自の行があります。したがって、各従業員に対して 1 行が作成されます。1 番目の外部キーは従業員が所属するプロジェクトを指し、2 番目の外部キーは従業員レコードを指します。図 4-21 に、`PROJECT_NO` と `EMPLOYEE_NO` という名前の外部キーを持つ関連表 `PROJECT_EMP` を示します。

図 4-21 多対多の双方向の関連の明示的なマッピング例



マッピングを他のデータベース表に変更する必要がある場合は、JDeveloper を使用するか、または orion-ejb-jar.xml ファイルを手動で編集して、<collection-mapping> または <set-mapping> 要素を操作します。

重要： 関連フィールドを明示的にマッピングするには、orion-ejb-jar.xml ファイルの <entity-deployment> 要素の要素と属性を変更してください。**JDeveloper は、Entity Bean とデータベース表との間の複雑なマッピングを管理するために作成されています。JDeveloper ではデプロイメント・ディスクリプタが検証され、不一致が防止されます。orion-ejb-jar.xml ファイルは手動で変更することもできますが、コンテナ管理の関連を変更する場合は、JDeveloper を使用することをお薦めします。**これは、CMR の構成が複雑で、理解するのが困難なためです。JDeveloper は次のサイトからダウンロードできます。
http://otn.oracle.co.jp/software/products/jdev/index.html

例 4-10 多対多関連のマッピングの XML 構造

ejb-jar.xml ファイルで定義された関連は、orion-ejb-jar.xml ファイルの <cmp-field-mapping> 要素内でマッピングされます。<cmp-field-mapping> 要素には、<collection-mapping> または <set-mapping> 要素が含まれます。プロジェクトと従業員の例では、<collection-mapping> 要素を使用して関連の両方の「多」の側を記述します。したがって、関連の両方の側の情報が同じ場合でも、<collection-mapping> を使用してそれぞれの側の関連を記述します。

ejb-jar.xml ファイルで、関連の両方の側で相互への「多」の関連を定義します。つまり、両方の側で <multiplicity> 要素を Many に宣言し、CMR フィールドで相互への関連を定義します。project Bean では CMR フィールドを employees に定義します。employee Bean では CMR フィールドを projects に定義します。これらの CMR フィール

ドを orion-ejb-jar.xml ファイルで使用して、関連をデータベース表にマッピングします。

```
<entity>
    ...
    <relationships>
        <ejb-relation>
            <ejb-relation-name>Emps-Projects</ejb-relation-name>
            <ejb-relationship-role>
                <ejb-relationship-role-name>Projects-have-Emps
                </ejb-relationship-role-name>
                <multiplicity>Many</multiplicity>
                <relationship-role-source>
                    <ejb-name>ProjectBean</ejb-name>
                </relationship-role-source>
                <cmr-field>
                    <cmr-field-name>employees</cmr-field-name>
                    <cmr-field-type>java.util.Collection</cmr-field-type>
                </cmr-field>
            </ejb-relationship-role>
            <ejb-relationship-role>
                <ejb-relationship-role-name>Emps-have-Projects
                </ejb-relationship-role-name>
                <multiplicity>Many</multiplicity>
                <relationship-role-source>
                    <ejb-name>EmpBean</ejb-name>
                </relationship-role-source>
                <cmr-field>
                    <cmr-field-name>projects</cmr-field-name>
                    <cmr-field-type>java.util.Collection</cmr-field-type>
                </cmr-field>
            </ejb-relationship-role>
        </ejb-relation>
    </relationships>
...
</entity>
```

次に orion-ejb-jar.xml ファイルで、両方の側で <collection-mapping> 要素を使用して相互の関連を定義します。この要素は関連表を定義します。2 つの外部キーを含む関連表が作成されます。それぞれの外部キーは、ソース表とターゲット表の主キーを指します。したがって、この関連を明示的にマッピングするには、関連表の名前とその外部キー名を変更する必要があります。両方の <collection-mapping> 要素には関連表に関する同じ情報が含まれているため、両方の <collection-mapping> 要素を同じ情報に変更する必要があります。唯一異なる点は、情報の格納先が各 Bean 定義の <primary-key> 要素か、または <value-mapping> 要素であるかです。project Bean 定義の <primary-key> 要素で定義された内容は、employee Bean 定義の <value-mapping> 要素でも定義されます。

```
<entity-deployment name="EmpBean" location="EmpBean"
    table="EmpBean_ormap_ormap_ejb" data-source="jdbc/OracleDS" >
...
<entity-deployment name="EmpBean"  table="EMP">
 <primkey-mapping>
       <cmp-field-mapping name="empNo" persistence-name="EMPNO" />
 </primkey-mapping>
 <cmp-field-mapping name="empName" persistence-name="ENAME" />
 <cmp-field-mapping name="salary" persistence-name="SAL" />
 <cmp-field-mapping name="projects">
  <collection-mapping table="PROJECT_EMP">
       <primkey-mapping>
           <cmp-field-mapping name="EmpBean_empNo">
                   <entity-ref home="EmpBean">
                   <cmp-field-mapping name="EmpBean_empNo"
                     persistence-name="EMPLOYEE_NO" />
              </entity-ref>
           </cmp-field-mapping>
       </primkey-mapping>
       <value-mapping type="hr.ProjectLocal">
           <cmp-field-mapping name="ProjectBean_projectNo">
              <entity-ref home="ProjectBean">
                   <cmp-field-mapping name="ProjectBean_projectNo"
                     persistence-name="PROJECT_NO" />
              </entity-ref>
           </cmp-field-mapping>
       </value-mapping>
  </collection-mapping>
 </cmp-field-mapping>
...
</entity-deployment>
...
<entity-deployment name="ProjectBean" location="ProjectBean"
 table="ProjectBean_ormap_ormap_ejb" data-source="jdbc/OracleDS" >
 <primkey-mapping>
     <cmp-field-mapping name="projectNo" persistence-name="PROJECTNO" />
 </primkey-mapping>
 <cmp-field-mapping name="projectName" persistence-name="PNAME" />
 <cmp-field-mapping name="employees">
   <collection-mapping table="PROJECT_EMP">
       <primkey-mapping>
          <cmp-field-mapping name="ProjectBean_projectNo">
              <entity-ref home="ProjectBean">
                 <cmp-field-mapping name="ProjectBean_projectNo"
                   persistence-name="PROJECT_NO" />
              </entity-ref>
          </cmp-field-mapping>
```

```
      </primkey-mapping>
      <value-mapping type="hr.EmpLocal">
         <cmp-field-mapping name="EmpBean_empNo">
            <entity-ref home="EmpBean">
               <cmp-field-mapping name="EmpBean_empNo"
                  persistence-name="EMPLOYEE_NO" />
            </entity-ref>
         </cmp-field-mapping>
      </value-mapping>
   </collection-mapping>
</cmp-field-mapping>
```

次に、orion-ejb-jar.xml ファイルのフィールドについて説明します。

- project Bean では、<ejb-jar.xml ファイルの <cmr-field 要素を定義して、従業員との関連の名前を employees> と定義します。employees の cmr-field> 要素では、プロジェクトとの関連の名前を projects と定義します。
- projects および employees の <cmr-field> 要素は、orion-ejb-jar.xml ファイルの関連表にマッピングされます。このファイルでは、projects および employees の <cmp-field-mapping> 要素に、それぞれ <collection-mapping> 要素が含まれています。この <collection-mapping> 要素の table 属性で、関連表の名前を PROJECT_EMP と定義します。
- 関連表には、2 つの外部キーがあります。この例では、単純な外部キーを使用しています。ただし、主キーがコンポジット主キーの場合は、この 2 つの外部キーもコンポジット外部キーになります。

  2 つの外部キーは、関連表の EmpBean で次のように定義されています。

  - <primkey-mapping> 要素の persistence-name 属性で、現行の Entity Bean の関連表の外部キーを定義します。この例では EMPLOYEE_NO に定義されています。
  - <value-mapping> 要素の persistence-name 属性で、ターゲット Bean の関連表の外部キーを定義します。この例では PROJECT_NO に定義されています。
- EmpBean の <value-mapping> 要素では、ターゲット Entity Bean を指定します。
  - <value-mapping> 要素の type 属性では、ソース Entity Bean に返されるターゲット Bean のローカル・インタフェースを定義します。この例では、project Bean のローカル・ホーム・インタフェースが hr.ProjectLocal として定義されます。
  - ターゲット Entity Bean の <ejb-name> は、<entity-ref> home 属性で定義します。この例では ProjectBean に定義されています。

コンポジット主キーでの外部キーの使用

EJB 仕様では、Entity Bean の主キーは ejbCreate メソッドで初期化する必要があります。この Bean から別の Bean への関連を ejbCreate メソッドで設定することはできません。この関連を外部キーに設定するには、ejbPostCreate メソッドで設定するのが最も簡単です。

ただし、コンポジット主キー内に外部キーがある場合は、次のような問題があります。

- コンポジット主キー内のすべてのフィールドを ejbCreate メソッドで設定する必要があります。
- 外部キーを ejbCreate メソッドで設定することはできません。

この項では、次の例を使用して、この問題を回避する方法を説明します。

> 会社の注文には、1 つ以上の品目が含まれている可能性があります。order Bean には、複数の品目が含まれています。各品目は 1 つの注文に属します。品目の主キーは、品目識別子と注文識別子で構成されるコンポジット主キーです。注文識別子は、注文を指す外部キーです。

デプロイメント・ディスクリプタおよび Bean 実装を変更して、実際の外部キー・フィールドに似たプレースホルダの CMP フィールドを追加する必要があります。このフィールドは、ejbCreate メソッドで設定されます。ただし、プレースホルダの CMP フィールドと外部キーは、両方とも同じデータベース列を指します。実際の外部キーは、ejbPostCreate メソッドで更新されます。

次の例は、デプロイメント・ディスクリプタおよび Bean 実装の両方を変更する方法を示します。

注意： プレースホルダの CMP フィールドおよび外部キーを使用して、ejb-jar.xml ファイルを変更します。アプリケーションをデプロイするときは、表を作成しないで orion-ejb-jar.xml ファイルを自動生成するように、orion-application.xml ファイルの autocreate-tables 要素を false に設定することをお薦めします。次に、適切なデータベース列を指すように orion-ejb-jar.xml ファイルを変更し、autocreate-tables 要素を true に設定して再デプロイします。

例 4-11 主キー内に存在する外部キー

各注文には 1 つ以上の品目が含まれます。したがって、2 つの Bean が作成されます。1 つは注文を表す OrderBean で、もう 1 つは注文の品目を表す OrderItemBean です。各品目には、品目番号およびその品目が属する注文番号で構成される主キーがあります。したがって、品目の主キーには、order Bean を指す外部キーが含まれます。

コンポジット主キーを調整するには、ejb-jar.xml ファイルで次の手順を実行します。

1. 主キーの CMP フィールドを、外部キーのプレースホルダとして定義します。このプレースホルダは、コンポジット主キーのクラス定義で使用されます。

   この例では、orderId CMP フィールドが <cmp-field> 要素で定義されています。orderId および itemId CMP フィールドを使用して、OrderItemPK.java のコンポジット主キーを識別します。

2. 主キー定義の外側にある外部キーを <relationships> セクションの <cmr-field> 要素内で定義します。

   この例では、belongToOrder 外部キーが OrderItemBean の <cmr-field> 要素で定義され、品目から注文への関連を定義しています。

```
<entity>
        <ejb-name>OrderItemBean</ejb-name>
        <local-home>OrderItemLocalHome</local-home>
        <local>OrderItemLocal</local>
        <ejb-class>OrderItemBean</ejb-class>
        ...
        <cmp-field><field-name>itemId</field-name></cmp-field>
        <cmp-field><field-name>orderId</field-name></cmp-field>
        <cmp-field><field-name>price</field-name></cmp-field>
        <prim-key-class>OrderItemPK</prim-key-class>
        ...
</entity>
<relationships>
        <ejb-relation>
             <ejb-relation-name>Order-OrderItem</ejb-relation-name>
             <ejb-relationship-role>
                 <ejb-relationship-role-name>Order-Has-OrderItems
                 </ejb-relationship-role-name>
                 <multiplicity>One</multiplicity>
                 <relationship-role-source>
                     <ejb-name>OrderBean</ejb-name>
                 </relationship-role-source>
                 <cmr-field>
                     <cmr-field-name>items</cmr-field-name>
                     <cmr-field-type>java.util.Collection</cmr-field-type>
                 </cmr-field>
             </ejb-relationship-role>
             <ejb-relationship-role>
                 <ejb-relationship-role-name>OrderItems-form-Order
                 </ejb-relationship-role-name>
                 <multiplicity>Many</multiplicity>
                 <cascade-delete/>
                 <relationship-role-source>
                     <ejb-name>OrderItemBean</ejb-name>
                 </relationship-role-source>
```

```
                <cmr-field>
                    <cmr-field-name>belongToOrder</cmr-field-name>
                </cmr-field>
            </ejb-relationship-role>
        </ejb-relation>
</relationships>
```

OrderItemPK.java クラスは、次のように、複合主キーの内容を定義します。

```
public class OrderItemPK implements java.io.Serializable
{
  public Integer itemId;
  public Integer orderId;

  public OrderItemPK()
  {
    this.itemId = null;
    this.orderId = null;
  }

  public OrderItemPK(Integer itemId, Integer orderId)
  {
    this.itemId = itemId;
    this.orderId = orderId;
  }
}

public boolean equals(Object o)
  {
      if (o instanceof OrderItemPK) {
          OrderItemPK pk = (OrderItemPK) o;
          if (pk.itemId.intValue() == itemId.intValue() &&
              pk.orderId.intValue() == orderId.intValue())
              return true;
      }
      return false;
  }

  public int hashCode()
  {
      return itemId.hashCode() * orderId.hashCode();
  }
}
```

自動生成されたデータベース表で十分な場合は、orion-ejb-jar.xml ファイルの変更は不要です。ただし、既存のデータベース表へのマッピングが必要な場合は、その表を指すように orion-ejb-jar.xml ファイルを変更します。

orion-ejb-jar.xml ファイルでの自動生成を設定した後、このファイルを開発ディレクトリにコピーします。データベース列名は、CMP および CMR フィールド名の各マッピングの persistence-name 属性で定義します。プレースホルダの CMP フィールドおよび外部キーの両方の persistence-name 属性が同じであることを確認します。

次の orion-ejb-jar.xml ファイルは、注文と注文品目の例を示します。OrderItemBean の <entity-deployment> セクションで、次の手順を実行します。

- 表は table 属性で定義します。この例では ORDER_ITEM に定義します。
- itemId の列名は、persistence-name 属性で Item_ID と定義します。
- プレースホルダの CMP フィールド orderId の列名は、persistence-name 属性で Order_ID と定義します。
- 外部キー belongToOrder は、データベース列 Order_ID にマッピングされます。これは、プレースホルダの CMP フィールド orderId と同じ列です。

外部キー belongToOrder およびプレースホルダの CMP フィールド orderId は、同じデータベース列を指す必要があります。

```
<entity-deployment name="OrderItemBean"  table="ORDER_ITEM">
   <primkey-mapping>
      <cmp-field-mapping name="itemId" persistence-name="Item_ID" />
      <cmp-field-mapping name="orderId" persistence-name="Order_ID" />
   </primkey-mapping>
   <cmp-field-mapping name="price" persistence-name="Price" />
   <cmp-field-mapping name="belongToOrder">
      <entity-ref home="OrderBean">
         <cmp-field-mapping name="belongToOrder"
            persistence-name="Order_ID" />
      </entity-ref>
   </cmp-field-mapping>
</entity-deployment>

<entity-deployment name="OrderBean"  table="ORDER">
   <primkey-mapping>
      <cmp-field-mapping name="orderId" persistence-name="Order_ID" />
   </primkey-mapping>
   <cmp-field-mapping name="orderDesc"
      persistence-name="Order_Description" />
   <cmp-field-mapping name="items">
      <collection-mapping table="ORDER_ITEM">
      <primkey-mapping>
         <cmp-field-mapping name="OrderBean_orderId">
            <entity-ref home="OrderBean">
```

```
                <cmp-field-mapping name="OrderBean_orderId"
                   persistence-name="Order_ID"/>
             </entity-ref>
          </cmp-field-mapping>
       </primkey-mapping>
       <value-mapping type="OrderItemLocal">
          <cmp-field-mapping name="OrderItemBean_itemId">
             <entity-ref home="OrderItemBean">
                <cmp-field-mapping name="OrderItemBean_itemId">
                   <fields>
                      <cmp-field-mapping name="OrderItemBean_itemId"
                         persistence-name="Item_ID"/>
                      <cmp-field-mapping name="OrderItemBean_orderId"
                         persistence-name="Order_ID"/>
                   </fields>
                </cmp-field-mapping>
             </entity-ref>
          </cmp-field-mapping>
       </value-mapping>
       </collection-mapping>
   </cmp-field-mapping>
</entity-deployment>
```

最後に、Bean 実装を更新して、プレースホルダの CMP フィールドおよび外部キーを使用可能にします。

1. `ejbCreate` メソッドで、次の手順を実行します。
   a. 外部キー・フィールドにかわるプレースホルダの CMP フィールドを作成します。
   b. `ejbCreate` メソッドで、プレースホルダの CMP フィールドの値を設定します。この値は、データベース表の外部キー・フィールドに書き込まれます。
2. `ejbPostCreate` メソッドで、外部キーに、CMP フィールドを複製した値を設定します。

注意： 外部キーは主キーの一部であるため、設定できるのは 1 回のみです。

この例では、CMP フィールド orderId は ejbCreate メソッドで設定され、関連フィールド belongToOrder は ejbPostCreate メソッドで設定されます。

```
public OrderItemPK ejbCreate(OrderItem orderItem) throws CreateException
{
    setItemId(orderItem.getItemId());
    setOrderId(orderItem.getOrderId());
    setPrice(orderItem.getPrice());
    return new OrderItemPK(orderItem.getItemId(),orderItem.getOrderId()) ;
}

public void ejbPostCreate(OrderItem orderItem) throws CreateException
{
    // when just after bean created
    try {
        Context ctx = new InitialContext();
        OrderLocalHome orderHome =
        (OrderLocalHome)ctx.lookup("java:comp/env/OrderBean");
        OrderLocal order = orderHome.findByPrimaryKey(orderItem.getOrderId());

        setBelongToOrder(order);
    }
    catch(Exception e) {
        e.printStackTrace();
        throw new EJBException(e);
    }
}
```

ejbCreate および ejbPostCreate メソッドに渡される OrderItem オブジェクトは、次のとおりです。

```
public class OrderItem implements java.io.Serializable
{

    private Integer itemId;
    private Integer orderId;
    private Double price;

    public OrderItem(Integer itemId, Integer orderId, Double price)
    {
        this.itemId = itemId;
        this.orderId = orderId;
        this.price = price;
    }

    public Integer getItemId() {
        return itemId;
    }
```

```
    public void setItemId(Integer itemId) {
        this.itemId = itemId;
    }

    public Integer getOrderId() {
        return orderId;
    }

    public void setOrderId(Integer orderId) {
        this.orderId = orderId;
    }

    public Double getPrice() {
        return price;
    }
    public void setPrice(Double price) {
        this.price = price;
    }

    public boolean equals(Object other)
    {
        if(other instanceof OrderItem) {
            OrderItem orderItem = (OrderItem)other;
            if (itemId.equals(orderItem.getItemId()) &&
                orderId.equals(orderItem.getOrderId()) &&
                price.equals(orderItem.getPrice()) ) {
                return true;
            }
        }
        return false;
    }
}
```

外部キーのデータベース制約のオーバーライド方法

NOT NULL などの制約を設定してデータベース列を定義した場合は、ejbCreate メソッドの後にエラーが発生する場合があります。INSERT は ejbCreate メソッドの後に実行されるため、データベース行に NULL のフィールドがあると、データベース表の制約違反が発生します。外部キーは ejbPostCreate メソッドを実行するまで割り当てることができないため、この違反は主に外部キーで発生します。この問題を回避するには、該当するフィールドで制約を解除する必要があります。

データベース制約を解除するには、制約を解除する列を DEFERRABLE に再定義します。制約を解除した場合は、トランザクションをコミットする前にデータベース・フィールドを設定して、データベース制約違反を回避します。

次に、TEST 表に対する遅延可能制約の作成方法を示します。

```
create table test (test varchar2(10) not null INITIALLY DEFERRED DEFERRABLE )
```

5

EJB 問合せ言語

EJB 2.0 では、標準化された EJB 問合せ言語（EJB QL）を使用して query メソッドを指定できます。

EJB QL の詳細は、EJB 2.0 仕様の第 11 章および様々な市販本で説明されています。この章では、メソッドの開発ルールの概要を説明しますが、EJB QL 構文の詳細は説明しません。

構文の詳細は、EJB 2.0 仕様および次のマニュアルを参照してください。

- 『Enterprise JavaBeans, 3rd Edition』（Richard Monson-Haefel 著、O'Reilly Publishers 刊）
- 『Special Edition Using Enterprise JavaBeans 2.0』（Chuck Cavaness、Brian Keeton 共著、Que Publishers 刊）

この章には、次の内容が含まれます。

EJB QL の概要

EJB QL は、SQL に類似した問合せ言語です。SQL の知識は EJB QL を使用する際にも役立ちます。SQL では、列名を使用し、表に対して問合せを行います。これに対して EJB QL では、Bean の抽象スキーマ名および CMP フィールドと CMR フィールドを問合せ内で使用し、Entity Bean に対して問合せを行います。EJB QL 文では、オブジェクト用語を使用します。

コンテナは、アプリケーションのデプロイ時に、EJB QL 文を適切なデータベース SQL 文に変換します。したがって、コンテナは、Entity Bean 名、CMP フィールド名および CMR フィールド名を、適切なデータベース表名と列名に変換します。EJB QL は、コンテナでサポートされているすべてのデータベースに移植可能です。

query メソッドの概要

query メソッドには、finder メソッドと select メソッドがあります。

- finder メソッド : finder メソッドは、Entity Bean 参照を取得するために使用します。
- select メソッド : select メソッドは、Entity Bean に対してのみ内部使用されます。これらのメソッドは、Entity Bean 参照または CMP 値のいずれかを取得するために使用します。

両方の種類のメソッドとも、`FinderException` をスローする必要があります。

finder メソッド

finder メソッドは、Entity Bean 参照を取得するために使用します。`findByPrimaryKey` finder メソッドは、常に両方のホーム・インタフェース（ローカルおよびリモート）で、この Bean に関するエンティティ参照を主キーを使用して取得するように定義されます。その他の finder メソッドは、両方またはいずれかのホーム・インタフェースで定義して、1 つまたは複数の Entity Bean 参照を取得できます。

finder メソッドを定義するには、次のようにします。

1. 任意のホーム・インタフェースで `find<name>` メソッドを定義します。リモート・ホーム・インタフェースとローカル・ホーム・インタフェースで、異なる finder メソッドを指定できます。両方のホーム・インタフェースで同じ finder メソッドを定義すると、同じ Bean クラス定義にマッピングされます。コンテナは、適切なホーム・インタフェース型を返します。
2. デプロイメント・ディスクリプタで、finder メソッドの完全な問合せ文、または条件文（`WHERE` 句）のみを定義します。

   問合せは、EJB QL 構文または OC4J 固有の構文を使用して定義できます。完全な問合せ、または問合せの条件部分（`WHERE` 句）のみのいずれかを指定できます。

- EJB QL 構文は、ejb-jar.xml ファイル内に定義されます。この構文は、Sun 社によって EJB 2.0 仕様の第 11 章に定義されています。EJB QL 文は、<query> 要素内の各 finder メソッドに対して作成されます。コンテナはこの文を使用して、Entity Bean 参照を取得する条件を、対応する SQL 文に変換します。

  EJB QL では現在、AVERAGE や SUM など、GROUP BY 機能と ORDER BY 機能のサポートに制限があります。

  詳細は、5-7 ページの「EJB QL 構文による finder メソッドの指定」を参照してください。

- OC4J 固有の構文は、orion-ejb-jar.xml ファイル内に定義されます。アプリケーションをデプロイすると、OC4J では EJB QL 構文が OC4J 固有の構文に変換され、<finder-method> 要素の query 属性に指定されます。OC4J 構文を使用してさらに複合的な問合せを作成するために、query 属性の文を変更できます。orion-ejb-jar.xml ファイル内の OC4J 固有の問合せ文は、ejb-jar.xml ファイル内の EJB QL 文より優先されます。

  詳細は、5-9 ページの「OC4J 固有の構文による finder メソッドの指定」を参照してください。

単一の Entity Bean 参照を取得する場合、コンテナは、find<name> メソッドで返されるのと同じ型を返します。複数の Entity Bean 参照をリクエストする場合は、Collection が返されるように、find<name> メソッドの戻り型を定義する必要があります。重複した項目を返さないようにするには、EJB QL 文で DISTINCT キーワードを指定します。一致する項目が見つからない場合は、空の Collection が返されます。

両方の finder メソッドの詳細は、5-7 ページの「finder メソッドの例」を参照してください。

select メソッド

select メソッドは主に CMP または CMR フィールドの値を返すために使用されます。値はすべて独自のオブジェクト型で返されます。プリミティブ型は類似の機能を持つオブジェクトでラップされます（たとえば、int プリミティブ型は Integer オブジェクトでラップされます）。select メソッドの詳細は、EJB 2.0 仕様の第 10.5.7 項を参照してください。

select メソッドは、Bean 内で内部使用されます。クライアントは、このメソッドをコールできません。したがって、このメソッドはホーム・インタフェースに定義しません。select メソッドは、Entity Bean 参照または CMP フィールド値を取得するために使用されます。

select メソッドを定義するには、次のようにします。

1. 各 select メソッドの Bean クラスに、ejbSelect<name> メソッドを定義します。各メソッドは、public abstract として定義します。このメソッドで必要な SQL は、実装に含まれていません。

2. デプロイメント・ディスクリプタで、select メソッドの完全な問合せ文、または条件文（WHERE 句）のみを定義します。EJB QL 文は、<query> 要素内の各 select メソッドに対して作成されます。コンテナはこの文を使用して、条件を対応する SQL 文に変換します。

両方の finder メソッドの詳細は、5-12 ページの「select メソッドの例」を参照してください。

返されるオブジェクト

select メソッドの戻り型を定義するルールは、次のとおりです。

- オブジェクトがない場合 : オブジェクトが見つからない場合は、FinderException が発生します。
- 単一オブジェクトの場合 : 単一の項目を取得する場合、コンテナは、ejbSelect<name> メソッドで返されるのと同じ型を返します。複数のオブジェクトが返された場合は、FinderException が発生します。
- 複数オブジェクトの場合 : 複数の項目をリクエストする場合は、ejbSelect<name> メソッドの戻り型を Set または Collection のいずれかに定義する必要があります。Set に定義すると、重複した項目が削除されます。Collection に定義すると、重複した項目も含まれます。たとえば、すべての顧客のすべての郵便番号を取得する場合は、Set を使用して重複するコードを削除します。すべての顧客の名前を取得する場合は、Collection を使用して完全なリストを取得します。一致する項目が見つからない場合は、空の Collection または Set が返されます。
  - Bean インタフェース : Bean インタフェースを返す場合、Set または Collection 内で返されるデフォルトのインタフェース型は、ローカル Bean インタフェースになります。このインタフェース型は、次のように、<result-type-mapping> 要素内でリモート Bean インタフェースに変更できます。

    ```
    <result-type-mapping>Remote</result-type-mapping>
    ```

  - CMP 値 : CMP 値の Set または Collection を返す場合、コンテナは、EJB QL の select 文からオブジェクト型を判断します。

デプロイメント・ディスクリプタのセマンティクス

デプロイメント・ディスクリプタにおいて、両方の種類の query メソッドを定義するために必要な構造は同じです。

1. EJB QL 文で参照する各 Entity Bean について、<entity> 要素に <abstract-schema-name> 要素を定義する必要があります。この要素では、EJB QL 文の Entity Bean を識別する名前を定義します。したがって、<abstract-schema-name> を Employee に定義すると、EJB QL では Employee を使用して EmpBean Entity Bean を参照します。
2. 各 query メソッド（finder および select）に対して <query> 要素を定義します（findByPrimaryKey finder メソッドを除きます）。

   注意： OC4J 固有の構文を使用する場合は、EJB QL の <query> 要素の構成から始めます。デプロイ後、orion-ejb-jar.xml ファイル内の問合せ文を必要に応じて変更します。

   <query> 要素には、次の 2 つの主要な要素があります。

   - <method-name> 要素は、finder または select メソッドを識別します。finder メソッドの名前は、コンポーネント・インタフェースとホーム・インタフェースで定義した名前と同じです。select メソッドの名前は、Bean クラスで定義した名前と同じです。
   - <ejb-ql> 要素には、このメソッドの EJB QL 文が含まれます。

例 5-1 employee の例での findAll デプロイメント・ディスクリプタの定義

次の例は、EmpBean Entity Bean の定義を示します。

- <entity> 要素では、<abstract-schema-name> を Employee に定義します。
- 2 つの <query> 要素では、finder メソッドの findAll と findByEmpNo をそれぞれ定義します。ここでは、EJB QL 文で Employee の名前を参照します。

```
<entity>
  <display-name>EmpBean</display-name>
  <ejb-name>EmpBean</ejb-name>
  ...
  <abstract-schema-name>Employee</abstract-schema-name>
  <cmp-field><field-name>empNo</field-name></cmp-field>
  <cmp-field><field-name>empName</field-name></cmp-field>
  <cmp-field><field-name>salary</field-name></cmp-field>
  <primkey-field>empNo</primkey-field>
    <prim-key-class>java.lang.Integer</prim-key-class>
    ...
    <query>
     <description></description>
      <query-method>
        <method-name>findAll</method-name>
        <method-params />
      </query-method>
     <ejb-ql>Select OBJECT(e) From Employee e</ejb-ql>
    </query>
    <query>
     <description></description>
     <query-method>
        <method-name>findByEmpNo</method-name>
        <method-params>
          <method-param>java.lang.Integer</method-param>
        </method-params>
      </query-method>
      <ejb-ql>SELECT OBJECT(e) FROM Employee e WHERE e.empNo = ?1
      </ejb-ql>
    </query>
...
</entity>
```

findAll メソッドの EJB QL 文は単純です。この文は、Employee Entity Bean から、変数 e によって識別されたオブジェクトを選択します。したがって、この文は、Employee Entity Bean のすべてのオブジェクトを選択します。findByEmpNo メソッドの EJB QL 文は、従業員名がこのメソッドに対する最初の入力パラメータと同等であるすべてのオブジェクトを選択します。デプロイ後、OC4J は次のように EJB QL 文を orion-ejb-jar.xml ファイルの <finder-method> 要素に変換します。

```
<finder-method query=""> /*the empty where clause finds all employees*/
<finder-method query="$empname = $1"> /*this finds all records where
employee is equal to the first input parameter.*/
```

詳細および例は、5-7 ページの「finder メソッドの例」を参照してください。

finder メソッドの例

finder メソッドを CMP Entity Bean 内で定義するには、次のようにします。

1. 一方または両方のホーム・インタフェースで finder メソッドを定義します。
2. デプロイメント・ディスクリプタで finder メソッド定義を定義します。

次の各項では、EJB QL 構文または OC4J 固有の構文のいずれかを使用して finder メソッドを作成する方法を説明します。

- EJB QL 構文による finder メソッドの指定
- OC4J 固有の構文による finder メソッドの指定

EJB QL 構文による finder メソッドの指定

finder メソッドを作成するための手順は 2 つあります。

1. ホーム・インタフェースでの finder メソッドの定義
2. デプロイメント・ディスクリプタでの finder メソッド定義の定義

ホーム・インタフェースでの finder メソッドの定義

finder メソッドをホーム・インタフェースに追加する必要があります。たとえば、全従業員を取得する場合は、次のように、ホーム・インタフェース（この例ではローカル・ホーム・インタフェース）で `findAll` メソッドを定義します。

```
public Collection findAll() throws FinderException;
```

1 名の従業員のデータを取得するには、次のように、ホーム・インタフェースで `findByEmpNo` を定義します。

```
public EmployeeLocal findByEmpNo(Integer empNo)
     throws FinderException;
```

返される Bean インタフェースは、ローカル・インタフェースの `EmployeeLocal` です。入力パラメータは従業員番号の `empNo` で、EJB QL の `?1` パラメータに代入されます。

デプロイメント・ディスクリプタでの finder メソッド定義の定義

各 finder メソッドは、`<query>` 要素内のデプロイメント・ディスクリプタで定義されます。例 5-1 には、`findAll` メソッドの EJB QL 文が含まれています。次の例は、`findByEmpNo` メソッドのデプロイメント・ディスクリプタを示します。

```
<query>
  <description></description>
  <query-method>
    <method-name>findByEmpNo</method-name>
    <method-params>
      <method-param>java.lang.Integer</method-param>
    </method-params>
  </query-method>
  <ejb-ql>SELECT OBJECT(e) FROM Employee e WHERE e.empNo = ?1
  </ejb-ql>
</query>
```

findByEmpName メソッドの EJB QL 文は、EJB QL の ?1 パラメータに従業員番号が代入された Employee オブジェクトを選択します。? 記号は、メソッド・パラメータのプレース・ホルダを示します。したがって、findByEmpNo では、少なくとも 1 つのパラメータを指定する必要があります。ここでは、findByEmpNo メソッドで渡される empNo が ?1 の位置に代入されます。変数の e は、WHERE 条件内で Employee オブジェクトを識別します。

関連における finder メソッドの例

Entity Bean 間の関連を含む EJB QL 文の場合、双方の Entity Bean は EJB QL 文内で相互に参照されます。次に、findByDeptNo メソッドの例を示します。この finder メソッドは、employee Bean 内で定義され、department Entity Bean を参照します。このメソッドは、1 つの部門に所属する全従業員を取得します。

```
<query>
  <description></description>
  <query-method>
    <method-name>findByDeptNo</method-name>
    <method-params>
      <method-param>java.lang.Integer</method-param>
    </method-params>
  </query-method>
  <ejb-ql>SELECT OBJECT(e) From Employee e, IN (e.dept)
        AS d WHERE d.deptNo = ?1
  </ejb-ql>
</query>
```

employee Bean の <abstract-schema-name> 要素は Employee です。employee Bean では、dept という CMR フィールドを使用して、department Bean との関連を定義します。したがって、department Bean は、dept CMR フィールドを介して EJB QL 文で参照されます。部門の主キーは deptNo です。問合せの対象になる部門番号は、入力パラメータで指定され、?1 に代入されます。

OC4J 固有の構文による finder メソッドの指定

finder メソッドを作成するための手順は 2 つあります。

1. ホーム・インタフェースへの finder メソッドの追加
2. OC4J 固有のデプロイメント・ディスクリプタへの問合せの追加

ホーム・インタフェースへの finder メソッドの追加

まず、finder メソッドをホーム・インタフェースに追加する必要があります。たとえば、employee Entity Bean の場合、すべての従業員を取得するには、`findAll` メソッドをホーム・インタフェースで次のように定義します。

```
public Collection findAll() throws FinderException, RemoteException;
```

OC4J 固有のデプロイメント・ディスクリプタへの問合せの追加

finder メソッドをホーム・インタフェースで指定した後、finder メソッドの問合せを使用して `orion-ejb-jar.xml` ファイルを変更します。

`<finder-method>` 要素は、`findByPrimaryKey` メソッド以外のすべての finder メソッドを定義します。定義が最も単純な finder メソッドは、`findByAll` メソッドです。`<finder-method>` 要素の `query` 属性には、完全な問合せ、または問合せの `WHERE` 句のみを指定できます。すべての行を取得する場合、空の問合せ（`query=""`）によってすべてのレコードが返されます。

OC4J 固有の finder メソッドは、`orion-ejb-jar.xml` ファイルの `<finder-method>` 要素に構成されています。また、次のように、各 `<finder-method>` 要素が `query` 属性で、SQL 文の一部または全体を指定しています。

```
<finder-method query=""> /*the empty where clause finds all */
OR
<finder-method query="$empname = $1"> /*this finds all records where
                  employee is equal to the first input parameter.*/
```

`query` 属性を持つ `<finder-method>` を使用した場合は、`ejb-jar.xml` ファイル内の同じメソッドに対する EJB QL の変更よりも優先されます。

複合的な finder メソッドを定義するには、次のようにします。

1. EJB QL を使用して、類似する単純な問合せを `ejb-jar.xml` に定義します。
2. アプリケーションをデプロイします。デプロイすると、OC4J によって EJB QL 文が OC4J 固有の文に変換されます。実行される SQL 文全体がコメントで表示されます。
3. `orion-ejb-jar.xml` ファイル内の `<finder-method>` の `query` 属性を、希望どおりの複合的な問合せに変更します。再デプロイすると、OC4J は新しい問合せを変換し、実行される正確な SQL 文が指定された新規コメントを書き出します。このコメントをチェックして、正しい構文が設定されていることを確認します。

EJB QL 構文を使用し、orion-ejb-jar.xml ファイルには既存の定義が存在する場合、次のようにします。

1. orion-ejb-jar.xml ファイル内の <finder-method> の query 属性を消去します。
2. アプリケーションを再デプロイします。OC4J により、query 属性が指定されていないことが通知され、かわりに ejb-jar.xml ファイルから EJB QL の方法論が使用されます。

例 5-2 OC4J 固有の finder 構文

次の例では、EmployeeBean からすべてのレコードを取得します。メソッド名は findAll で、すべての従業員の Collection を返すため、パラメータは必要ありません。

```
<finder-method query="">
<!-- Generated SQL: "select EmployeeBean.empNo, EmployeeBean.empName,
    EmployeeBean.salary from EmployeeBean" -->
  <method>
     <ejb-name>EmployeeBean</ejb-name>
     <method-name>findAll</method-name>
         <method-params></method-params>
     </method>
</finder-method>
```

デプロイ後、OC4J は問合せ内容に関するコメント行を追加します。このコメントを使用して、問合せのタイプが正しいことを確認してください。

より具体的な問合せを行うには、query 属性に適切な WHERE 句を追加します。この句では、$ 記号を使用して渡されたパラメータを参照します。最初のパラメータは $1 で示され、2 番目のパラメータは $2 で示されます。WHERE 句内で使用されるすべての <cmp-field> 要素は、$<cmp-field> 名で示されます。

次の例では、findByName メソッド（ホーム・インタフェースで定義する）を指定します。従業員の名前がメソッド・パラメータとして渡されており、$1 に置換されています。これは、CMP 名 "empName" に一致しています。このように、query 属性は、WHERE 句について、"$empname=$1" を含めるよう変更されています。

```
<finder-method query="$empname = $1">
  <method>
    <ejb-name>EmployeeBean</ejb-name>
    <method-name>findByName</method-name>
    <method-params>
      <method-param>java.lang.String</method-param>
    </method-params>
  </method>
</finder-method>
```

メソッド・パラメータが複数存在する場合、各パラメータ型は連続した <method-param> 要素で定義され、問合せ文では、連続した $*n* で示されます。*n* は番号を示します。

注意： query 属性で SQL JOIN を指定することも可能です。

WHERE 句の後のセクションだけではなく、完全な問合せ文を指定する場合、partial 属性を FALSE に指定し、完全な問合せ文を query 属性で定義します。partial のデフォルト値は true であるため、前の finder メソッドの例では指定されていません。

```
<finder-method partial="false"
               query="select * from EMP where $empName = $1">
        <!-- Generated SQL: "select * from EMP where EMP.ENAME = ?" -->
        <method>
                <ejb-name>EmployeeBean</ejb-name>
                <method-name>findByName</method-name>
                <method-params>
                        <method-param>java.lang.String</method-param>
                </method-params>
        </method>
</finder-method>
```

完全な SQL 問合せ文の指定は、複合 SQL 文の場合に役立ちます。

Entity Bean の finder メソッドの場合、遅延ロードによって select メソッドを複数回起動できます。デフォルトでは、遅延ロードはオフです。取得するオブジェクト数が大量で、アクセスするのはその中の一部である場合は、遅延ロードをオンにすることをお薦めします。

遅延ロードをオンにするには、lazy-loading プロパティを true に設定します。

```
<finder-method partial="false"
               query="select * from EMP where $empName = $1"
               lazy-loading=true>
        <!-- Generated SQL: "select * from EMP where EMP.ENAME = ?" -->
        <method>
                <ejb-name>EmployeeBean</ejb-name>
                <method-name>findByName</method-name>
                <method-params>
                        <method-param>java.lang.String</method-param>
                </method-params>
        </method>
</finder-method>
```

さらに、次のように prefetch-size 属性を設定して、JDBC ドライバが一度にフェッチする行数を指定できます。

```
<finder-method partial="false"
               query="select * from EMP where $empName = $1"
               prefetch-size="15" >
        <!-- Generated SQL: "select * from EMP where EMP.ENAME = ?" -->
        <method>
```

```
                <ejb-name>EmployeeBean</ejb-name>
                <method-name>findByName</method-name>
                <method-params>
                        <method-param>java.lang.String</method-param>
                </method-params>
        </method>
</finder-method>
```

Oracle JDBC Drivers には、問合せの過程で結果セットを移入する際にクライアントにプリフェッチする行数を設定できる拡張機能が含まれています。この機能を使用してデータをフェッチする際に複数のデータ行をフェッチすることによって、データベースへのラウンドトリップを削減できます。余分なデータは、後でクライアントがアクセスするためにクライアント側のバッファに格納されます。プリフェッチする行数は、自由に設定できます。クライアントにプリフェッチするデフォルトの行数は 10 です。ここに設定した行数は JDBC ドライバに渡されます。JDBC ドライバでプリフェッチを使用する方法の詳細は、『Oracle9*i* JDBC 開発者ガイドおよびリファレンス』を参照してください。

select メソッドの例

select メソッドを CMP Entity Bean 内で定義するには、次のようにします。

1. select メソッドを `ejbSelect<name>` として Bean クラスで定義します。
2. デプロイメント・ディスクリプタで select メソッド定義を定義します。

> **注意：** `orion-ejb-jar.xml` ファイル内の `ejbSelect` メソッドの問合せ文は、finder メソッドの場合と同じようには変更できません。

Bean クラスでの select メソッドの定義

select メソッドを抽象メソッドとして Bean クラスで追加します。たとえば、給与が指定範囲内にある全従業員を取得する場合は、次のように、`ejbSelectBySalaryRange` メソッドを定義します。

```
public abstract Collection ejbSelectBySalaryRange(Float s1, Float s2)
throws FinderException;
```

select メソッドは複数の従業員を取得するため、`Collection` が返されます。入力パラメータは給与範囲の上限と下限で、それぞれ EJB QL の ?1 パラメータと ?2 パラメータに代入されます。最初の入力パラメータは ?1 に、2 番目の入力パラメータは ?2 に返されます。宣言されたすべてのメソッド・パラメータの順序は、EJB QL パラメータの順序（?1、?2、... ?n）と同じです。

デプロイメント・ディスクリプタでの select メソッド定義の定義

各 select メソッドは、<query> 要素のデプロイメント・ディスクリプタで定義します。次の例は、ejbSelectBySalaryRange と ejbSelectNameBySalaryRange の両方のメソッドのデプロイメント・ディスクリプタを示します。

```
<query>
  <description></description>
  <query-method>
    <method-name>ejbSelectBySalaryRange</method-name>
    <method-params>
      <method-param>java.lang.Float</method-param>
      <method-param>java.lang.Float</method-param>
    </method-params>
  </query-method>
  <ejb-ql>SELECT DISTINCT OBJECT(e) From Employee e
          WHERE e.salary BETWEEN ?1 AND ?2
  </ejb-ql>
</query>
<query>
  <description></description>
  <query-method>
    <method-name>ejbSelectNameBySalaryRange</method-name>
    <method-params>
      <method-param>java.lang.Float</method-param>
      <method-param>java.lang.Float</method-param>
    </method-params>
  </query-method>
  <ejb-ql>SELECT e.empName From Employee e
          WHERE e.salary BETWEEN ?1 AND ?2
  </ejb-ql>
</query>
```

これら両方のメソッドには、float 型の 2 つの入力パラメータが必要です。これらの入力パラメータの型は、<method-param> 要素で定義します。

EJB QL は、<ejb-ql> 要素で定義します。両方のメソッドが、e.salary によって EJB QL 文内の salary の CMP フィールドを評価します。e は Employee オブジェクトを表し、salary はそのオブジェクト内の CMP フィールドを表します。ピリオドで区切ることによって、Entity Bean とその CMP フィールドの関連を示します。

2 つの入力パラメータで給与範囲の上限と下限を指定し、それぞれが ?1 と ?2 の位置に代入されます。

ejbSelectBySalaryRange メソッドはオブジェクトを返します。DISTINCT キーワードを使用すると、重複したレコードは返されません。ejbSelectNameBySalaryRange は、従業員の名前のみを String で返します。これは、オブジェクト内の CMP フィールドの値のみを返すことができる、select 文の利点の 1 つを示しています。

Oracle での EJB QL 型の拡張機能 : Date、Time、Timestamp および SQRT

EJB 仕様の現在のバージョンでは Date、Time、Timestamp および SQRT はサポートされていませんが、Oracle ではこれらの型を次のようにサポートしています。

- SQRT(v) : double プリミティブ型と java.lang.Double 型の両方が、引数に対してサポートされます。
- java.util.Date、java.sql.Date、java.sql.Time および java.sql.Timestamp を、等式などの EJB QL の二項式で使用できます。

次の例は、これらの EJB QL 型拡張機能の使用方法を示しています。

例 5-3 SQRT の使用方法

```
<query>
  <query-method>
    <method-name>ejbSelectDoubleTypeSqrt</method-name>
    <method-params>
      <method-param>double</method-param>
    </method-params>
  </query-method>
  <result-type-mapping>Remote</result-type-mapping>
  <ejb-ql>
    SELECT OBJECT(a) FROM Dept a WHERE a.deptDoubleType = SQRT(?1)
  </ejb-ql>
</query>
```

例 5-4 Date の例

```
<query>
  <query-method>
   <method-name>ejbSelectDate</method-name>
   <method-params>
    <method-param>java.util.Date</method-param>
   </method-params>
  </query-method>
  <result-type-mapping>Remote</result-type-mapping>
  <ejb-ql>
    SELECT OBJECT(a) FROM Dept a WHERE a.deptDate = ?1
   </ejb-ql>
</query>
```

例 5-5 Date の別の例

```
<query>
  <query-method>
   <method-name>ejbSelectSqlDate</method-name>
   <method-params>
    <method-param>java.sql.Date</method-param>
   </method-params>
  </query-method>
  <result-type-mapping>Remote</result-type-mapping>
  <ejb-ql>
    SELECT OBJECT(a) FROM Dept a WHERE a.deptSqlDate = ?1
  </ejb-ql>
</query>
```

例 5-6 Timestamp の例

```
<query>
  <query-method>
   <method-name>findByTimestamp</method-name>
   <method-params>
    <method-param>java.sql.Timestamp</method-param>
   </method-params>
  </query-method>
  <result-type-mapping>Remote</result-type-mapping>
  <ejb-ql>
    SELECT OBJECT(a) FROM Dept a WHERE a.deptTimestamp = ?1
  </ejb-ql>
</query>
```

例 5-7 Time の例

```
<query>
  <query-method>
   <method-name>findByTimestamp</method-name>
   <method-params>
    <method-param>java.sql.Time</method-param>
   </method-params>
  </query-method>
  <result-type-mapping>Remote</result-type-mapping>
  <ejb-ql>
    SELECT OBJECT(a) FROM Dept a WHERE a.deptTime = ?1
  </ejb-ql>
</query>
```

6

BMP Entity Bean

データの手動による格納およびリロードを実装する場合は、Bean 管理の永続性（BMP）を持つ Bean を使用します。コンテナは、データをコールバック・メソッド内で管理するため、これらのメソッドの実装が必要です。データを永続記憶域に格納するためのロジックはすべて `ejbStore` メソッド内に含まれ、`ejbLoad` メソッドで記憶域からリロードされます。これらのメソッドは、必要に応じてコンテナから起動します。

この章では、基本的な構成およびデプロイを使用した単純な BMP EJB の開発方法を説明します。BMP Entity Bean の例は、OTN-J のサイト `http://otn.oracle.co.jp/sample_code/index.html` の OC4J のサンプル・コードのページからダウンロードしてください。

次の各項で、データの永続性の実装手順について説明します。

- BMP Entity Bean の作成
- コンポーネント・インタフェースとホーム・インタフェース
- BMP Entity Bean の実装
- エンティティ・データのデータベース表および列の作成

BMP Entity Bean の作成

2-2 ページの「EJB の開発」では、ステートレス Session Bean の開発方法を説明しています。第 3 章「CMP Entity Bean」では、CMP Entity Bean の実装の基礎と、追加の手順について説明しています。CMP Bean では、永続性に関する主キーおよびすべての機能は、コンテナによって実行されます。BMP Bean では、Bean の永続性を保存するために、主キーおよびすべての機能を実装する必要があります。主キーは `ejbCreate` メソッドで管理されます。永続性は、次の機能で管理されます。

- `ejbStore` メソッドにおけるデータの永続保存。
- `ejbLoad` メソッドの実装内での Bean への永続データのリストア。
- `ejbPassivate` メソッドにおける Bean インスタンスの非アクティブ化。
- `ejbActivate` メソッド内における、非アクティブ化された Bean インスタンスのアクティブ化。

次に、第 3 章「CMP Entity Bean」で説明した手順の概要を示します。これらの手順は、Bean の作成時に実行する必要があります。詳細は、2-2 ページの「EJB の開発」および第 3 章「CMP Entity Bean」を参照してください。この章の残りの部分では、主キーと永続性機能の実装方法を説明します。

1. Bean のコンポーネント・インタフェースを作成します。コンポーネント・インタフェースは、クライアントによって起動可能なメソッドを宣言します。
2. Bean のホーム・インタフェースを作成します。ホーム・インタフェースでは、`findByPrimaryKey` を含め、作成する Bean の `create` および finder メソッドを定義します。
3. Bean の主キーを定義します。主キーはシリアライズ可能なクラスで、各 Entity Bean インスタンスを識別します。単純なデータ型クラス（`java.lang.String` など）を使用したり、複合クラス（主キーのコンポーネントとして複数のオブジェクトを持つクラスなど）を定義できます。
4. Bean を実装します。
5. 永続データをデータベースに格納またはリストアする場合、Bean に対して正しい表が存在することを確認する必要があります。
6. Bean のデプロイメント・ディスクリプタを作成します。デプロイメント・ディスクリプタにより、XML 要素を通じて Bean のプロパティを指定します。
7. Bean、コンポーネント・インタフェース、ホーム・インタフェースおよびデプロイメント・ディスクリプタを含める EJB JAR ファイルを作成します。作成した後、`application.xml` ファイルを構成し、EAR ファイルを作成し、EJB を OC4J にデプロイします。

注意： このマニュアルでは、EJB コンテナ・サービスについて説明していません。詳細は、『Oracle Application Server Containers for J2EE サービス・ガイド』の JTA、データ・ソースおよび JNDI の各章を参照してください。この章ではトランザクションについて説明していないため、BMP Bean の例ではコンテナ管理のトランザクションを使用しています。

セキュリティについては、『Oracle Application Server Containers for J2EE セキュリティ・ガイド』を参照してください。

コンポーネント・インタフェースとホーム・インタフェース

コンポーネント・インタフェースおよびホーム・インタフェースの BMP Entity Bean の定義は、CMP Entity Bean の場合と同じです。コンポーネント・インタフェースおよびホーム・インタフェースの実装方法の例は、3-3 ページの「Entity Bean の作成」を参照してください。

BMP Entity Bean の実装

主キーの管理や永続データの格納はコンテナによって行われないため、Bean のコールバック関数には、これらの機能の実装ロジックが含まれている必要があります。コンテナは、`ejbCreate`、`ejbFindByPrimaryKey`、その他の finder メソッドを起動し、必要な場合は、`ejbStore` および `ejbLoad` メソッドを起動します。

次の各項では、BMP Bean の管理に関する実装を追加する方法について説明します。

- ejbCreate の実装
- ejbFindByPrimaryKey の実装
- その他の finder メソッド
- ejbStore の実装
- ejbLoad の実装
- ejbPassivate の実装
- ejbActivate の実装
- ejbRemove の実装

ejbCreate の実装

ejbCreate メソッドは、主に主キーの作成を実行します。次のものが含まれます。

1. 主キーの作成
2. キーに対する永続データ表現の作成
3. 一意の値へのキーの初期化および重複がないことの確認
4. コンテナへのこのキーの返却

コンテナにより、キーが Entity Bean の参照にマッピングされます。

次の例では、employee の例の ejbCreate メソッドを示します。このメソッドは、主キーである empNo を初期化します。これは、本来なら、自動的に、次に使用可能な従業員番号である主キーを生成します。ただし、ここでは例を簡単にするために、この ejbCreate メソッドはユーザーに一意の従業員番号を入力するよう求めます。

注意： この説明では、サンプル内のすべての TRY ブロックが省略されています。ただし、TRY ブロックも含めた BMP Entity Bean の例全体は、OTN-J のサイト http://otn.oracle.co.jp/sample_code/index.html の OC4J のサンプル・コードのページから入手できます。

さらに、従業員の全データがこのメソッドで提供されるため、データはこのインスタンスのコンテキスト変数内に格納されます。初期化後、このキーがコンテナに返されます。

```
// The create methods takes care of generating a new empNo and returns
// its primary key to the container
public Integer ejbCreate (Integer empNo, String empName, Float salary)
          throws CreateException
{
 /* in this implementation, the client gives the employee number, so
    only need to assign it, not create it. */
 this.empNo = empNo;
 this.empName = empName;
 this.salary = salary;

  /* insert employee into database */
  conn = getConnection(dsName);
  ps = conn.prepareStatement("INSERT INTO EMPLOYEEBEAN (EmpNo, EmpName, SAL)
       VALUES ( "+this.empNo.intValue()+", "+this.empName+","
       +this.salary.floatValue()+")");
  ps.executeUpdate();
  ps.close();
```

```
  /* return the new primary key.*/
  return (empNo);
}
```

このデプロイメント・ディスクリプタでは、<prim-key-class> 要素内で、主キー・クラスのみ定義します。Bean がデータを保存するため、デプロイメント・ディスクリプタには、永続データの定義は存在しません。ただし、デプロイメント・ディスクリプタの <resource-ref> 要素で、Bean が使用するデータベースを定義します。データベース構成の詳細は、6-11 ページの「XML デプロイメント・ディスクリプタの修正」を参照してください。

```
<enterprise-beans>
   <entity>
      <display-name>EmployeeBean</display-name>
      <ejb-name>EmployeeBean</ejb-name>
      <local-home>employee.EmployeeHome</local-home>
      <local>employee.Employee</local>
      <ejb-class>employee.EmployeeBean</ejb-class>
      <persistence-type>Bean</persistence-type>
      <prim-key-class>java.lang.Integer</prim-key-class>
      <reentrant>False</reentrant>
      <resource-ref>
            <res-ref-name>jdbc/OracleDS</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Application</res-auth>
      </resource-ref>
   </entity>
</enterprise-beans>
```

または、複数のデータ型に基づいた複合主キーを作成可能です。複合主キーは、次のように、そのクラス内で定義します。

```
package employee;

public class EmployeePK implements java.io.Serializable
{
  public Integer empNo;
  public String empName;
  public Float salary;

  public EmployeePK(Integer empNo)
  {
    this.empNo = empNo;
    this.empName = null;
    this.salary = null;
  }
```

```
  public EmployeePK(Integer empNo, String empName, Float salary)
  {
    this.empNo = empNo;
    this.empName = empName;
    this.salary = salary;
  }

}
```

主キー・クラスの場合、クラスを <prim-key-class> 要素で定義します。これは、単純な主キー定義と同じです。

```
<enterprise-beans>
   <entity>
      <display-name>EmployeeBean</display-name>
      <ejb-name>EmployeeBean</ejb-name>
      <local-home>employee.EmployeeHome</local-home>
      <local>employee.Employee</local>
      <ejb-class>employee.EmployeeBean</ejb-class>
      <persistence-type>Bean</persistence-type>
      <prim-key-class>employee.EmployeePK</prim-key-class>
      <reentrant>False</reentrant>
      <resource-ref>
            <res-ref-name>jdbc/OracleDS</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Application</res-auth>
      </resource-ref>
   </entity>
</enterprise-beans>
```

employee の例では、ユーザーが従業員番号を Bean に対して指定する必要があります。別の手段としては、次に使用可能な従業員番号を計算し、この番号を従業員の名前および勤務地と組み合せて従業員番号を生成する方法があります。

複合主キー・クラスの定義後、次のように、ejbCreate メソッド内で主キーを作成します。

```
public EmployeePK ejbCreate(Integer empNo, String empName, Float salary)
    throws CreateException
{
  pk = new EmployeePK(empNo, empName, salary);
  ...
}
```

ejbCreate（または ejbPostCreate）が処理する作業には、他に、Bean の存続期間中に必要なリソースの割当てがあります。この例では、すでに従業員情報が用意されているため、ejbCreate は次の処理を実行します。

1. データベースへの接続を取得します。この接続は、Bean の存続期間中オープンされています。データベース内の従業員情報の更新に使用されます。ejbPassivate および ejbRemove で解放し、ejbActivate で再割当てを行います。
2. データベースの従業員情報を更新します。

これは、次のように実行されます。

```
public EmployeePK ejbCreate(Integer empNo, String empName, Float salary)
    throws CreateException
{
  pk = new EmployeePK(empNo, empName, salary);
  conn = getConnection(dsName);
  ps = conn.prepareStatement("INSERT INTO EMPLOYEEBEAN (EmpNo, EmpName, SAL)
       VALUES ( "+this.empNo.intValue()+", "+this.empName+","
       +this.salary.floatValue()+")");
  ps.executeUpdate();
  ps.close();
  return pk;
}
```

ejbFindByPrimaryKey の実装

ejbFindByPrimaryKey 実装は、すべての BMP Entity Bean に必要です。主な用途は、主キーが有効な Bean に対応しているかを確認することです。妥当性が検証されると、主キーをコンテナに返し、コンテナはそのキーを使用して Bean 参照をユーザーに返します。

このサンプルでは、従業員番号が有効であることを検証し、主キー（従業員番号）をコンテナに返します。主キーがクラスの場合、より複雑な検証が必要になります。

```
public Integer ejbFindByPrimaryKey(Integer empNoPK)
    throws FinderException
{
  if (empNoPK == null) {
       throw new FinderException("Primary key cannot be null");
  }

  ps = conn.prepareStatement("SELECT EMPNO FROM EMPLOYEEBEAN
              WHERE EMPNO = ?");
  ps.setInt(1, empNoPK.intValue());
  ps.executeQuery();
  ResultSet rs = ps.getResultSet();
  if (rs.next()) {
   /*PK is validated because it exists already*/
  } else {
  throw new FinderException("Failed to select this PK");
 }
```

```
 ps.close();

 return empNoPK;
}
```

その他の finder メソッド

ejbFindByPrimaryKey 以外にも、他の finder メソッドを作成可能です。

他の finder メソッドを作成するには、次のようにします。

1. finder メソッドをホーム・インタフェースに追加します。
2. BMP Bean 実装で、finder メソッドを実装します。

finder メソッドでは、WHERE 句に従って、1 つ以上の Bean を取得できます。1 つ以上の Bean を返す場合、BMP finder メソッドは、主キーの Collection を返す必要があります。これらの finder メソッドは、ユーザーに返す必要のあるすべての Entity Bean の主キーのみ収集する必要があります。コンテナは、Collection 内に存在する各 Entity Bean への参照(複数の参照が返された場合)、または単一のクラス・タイプへの参照へ、主キーをマッピングします。

次の例では、すべての従業員のレコードを返す finder メソッドの実装を示します。

```
public Collection ejbFindAll() throws FinderException
{
  Vector recs = new Vector();

  ps = conn.prepareStatement("SELECT EMPNO FROM EMPLOYEEBEAN");
  ps.executeQuery();
  ResultSet rs = ps.getResultSet();

  int i = 0;

  while (rs.next())
  {
   retEmpNo = new Integer(rs.getInt(1));
   recs.add(retEmpNo);
  }

 ps.close();
 return recs;
}
```

ejbStore の実装

コンテナは、永続データをデータベースに格納する必要がある場合に ejbStore メソッドを起動します。これによって、インスタンスの状態が、基礎となるデータベース内のエンティティと同期化されます。この起動は、たとえば、コンテナによる Bean インスタンスの非アクティブ化やインスタンスの削除前に行われます。BMP Bean により、このメソッド内で、すべてのデータが同じリソース（データベースなど）に確実に格納されます。

```
public void ejbStore()
{
 //Container invokes this method to instruct the instance to
 //synchronize its state by storing it to the underlying database
 ps = conn.prepareStatement("UPDATE EMPLOYEEBEAN SET EMPNAME=?,
              SALARY=? WHERE EMPNO=?)";
 ps.setString(1, this.empName);
 ps.setFloat(2, this.salary.floatValue());
 ps.setInt(3, this.empNo.intValue());
 if (ps.executeUpdate() != 1) {
        throw new EJBException("Failed to update record");
 }
 ps.close();
}
```

ejbLoad の実装

コンテナは、Bean の状態をデータベース内の状態と同期化する必要がある場合に ejbLoad メソッドを起動します。このメソッドは、データベース内の状態でリフレッシュするために Bean インスタンスをアクティブ化した後に起動されます。このメソッドの用途は、永続データを、格納されているデータの状態に戻すことです。ほとんどの ejbLoad メソッドの場合、これは、データベースからインスタンス・データ変数にデータを読み取ることを意味します。

```
public void ejbLoad()
{
  //Container invokes this method to instruct the instance to
  //synchronize its state by loading it from the underlying database
  this.empNo = ctx.getPrimaryKey();
  ps = conn.prepareStatement("SELECT EMP_NO, EMP_NAME, SALARY WHERE EMPNAME=?");
  ps.setInt(1, this.empNo.intValue());
  ps.executeQuery();
  ResultSet rs = ps.getResultSet();
  if (rs.next()) {
    this.empNo = new Integer(rs.getInt(1));
    this.empName = new String(rs.getString(2));
    this.salary = new Float(rs.getFloat(3));
  } else {
```

```
    throw new FinderException("Failed to select this PK");
  }
  ps.close();
}
```

ejbPassivate の実装

ejbPassivate メソッドは、後で使用するため、Bean インスタンスがシリアライズ化される直前に起動されます。ユーザーが次にこのインスタンスでメソッドを起動すると、ejbActivate メソッドによってインスタンスが再びアクティブ化します。

Bean が非アクティブ化される前に、すべてのリソースおよびシリアライズ化するには大きすぎる静的データをすべて解放する必要があります。ejbActivate メソッドによって容易に再生成可能である大きな静的情報は、このメソッドで解放します。

この例では、シリアライズ化できないリソースは、オープンしているデータベース接続のみです。これは、このメソッドでクローズされ、ejbActivate メソッドで再びオープンされます。

```
public void ejbPassivate()
{
 // Container invokes this method on an instance before the instance
 // becomes disassociated with a specific EJB object
 conn.close();
}
```

ejbActivate の実装

コンテナは、Bean インスタンスを再びアクティブ化するときに、このメソッドを起動します。つまり、ユーザーがこのインスタンスでのメソッドの起動を要求した場合です。このメソッドは、ejbPassivate メソッドで解放されたリソースのオープンおよび静的情報の再構築に使用されます。

さらに、コンテナは、トランザクションの開始後に、このメソッドを起動します。

employee の例では、従業員情報が格納されているデータベース接続をオープンします。

```
public void ejbActivate()
{
 // Container invokes this method when the instance is taken out
 // of the pool of available instances to become associated with
 // a specific EJB object
 conn = getConnection(dsName);
}
```

ejbRemove の実装

コンテナは、Bean インスタンス自体を削除するか、またはインスタンスを Bean プールに戻す前に、ejbRemove メソッドを起動します。つまり、この Entity Bean で保持していた情報は、永続記憶域内から削除する必要があります。employee の例では、インスタンスが破棄される前に、従業員および関連情報をすべてデータベースから削除します。データベース接続をクローズします。

```
public void ejbRemove() throws RemoveException
{
 //Container invokes this method befor it removes the EJB object
 //that is currently associated with the instance
 ps = conn.prepareStatement("DELETE FROM EMPLOYEEBEAN WHERE EMPNO=?");
 ps.setInt(1, this.empNo.intValue());
 if (ps.executeUpdate() != 1) {
        throw new RemoveException("Failed to delete record");
 }
 ps.close();
 conn.close();
}
```

XML デプロイメント・ディスクリプタの修正

3-3 ページの「Entity Bean の作成」で説明した構成に加え、次のものを修正して、ejb-jar.xml デプロイメント・ディスクリプタに追加する必要があります。

1. <persistence-type> 要素で、永続タイプを "Bean" に構成します。
2. <resource-ref> 要素で、データベース永続記憶域に対するリソースの参照を構成します。

   employee の例では、"jdbc/OracleDS" というデータベース環境要素を使用しています。これは、次のように、<resource-ref> 要素で構成されています。

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

<res-ref-name> 要素で指定されたデータベースは、data-sources.xml ファイルの <ejb-location> 要素にマッピングされます。例の中の "jdbc/OracleDS" データベースは、次に示すように、data-sources.xml ファイルで構成されています。

```
<data-source
  class="com.evermind.sql.DriverManagerDataSource"
  name="Oracle"
  location="jdbc/OracleCoreDS"
  pooled-location="jdbc/pool/OraclePoolDS"
  ejb-location="jdbc/OracleDS"
  xa-location="jdbc/xa/OracleXADS"
  connection-driver="oracle.jdbc.driver.OracleDriver"
  url="jdbc:oracle:thin:@myhost:1521:orcl"
  username="scott"
  password="tiger"
  max-connections="300"
  min-connections="5"
  max-connect-attempts="10"
  connection-retry-interval="1"
  inactivity-timeout="30"
  wait-timeout="30"
/>
```

注意： BMP Entity Bean の例全体は、OTN-J のサイト `http://otn.oracle.co.jp/sample_code/index.html` の OC4J のサンプル・コードのページから入手できます。

エンティティ・データのデータベース表および列の作成

Entity Bean がデータベース内に永続データを格納する場合、Entity Bean 用に、適切な列が含まれている表を作成する必要があります。この表は、Bean をデータベースにロードする前に作成する必要があります。コンテナは、BMP Bean の場合はこの表を作成しませんが、CMP Bean の場合は自動的に作成します。

employee の例の場合、`data-sources.xml` ファイルで定義されたデータベースに次の表を作成する必要があります。

表	列
EMPLOYEEBEAN	■ 従業員番号（EMPNO） ■ 従業員名（EMPNAME） ■ 給料（SALARY）

次に、これらのフィールドを作成する SQL コマンドを示します。

```
CREATE TABLE EMPLOYEEBEAN (
 EMPNO NUMBER NOT NULL,
 EMPNAME VARCHAR2(255) NOT NULL,
 SALARY FLOAT NOT NULL,
 CONSTRAINT EMPNO PRIMARY KEY
)
```

注意： このマニュアルでは、EJB コンテナ・サービスについて説明していません。データ・ソース・オブジェクトの構成方法の詳細は、『Oracle Application Server Containers for J2EE サービス・ガイド』のデータ・ソースの章を参照してください。

7

Message-Driven Bean

次の各項では、Oracle Application Server Containers for J2EE（OC4J）で MDB を作成する手順を説明し、基本的な構成で MDB を開発して、OC4J JMS または Oracle JMS を JMS プロバイダとして使用する方法を説明します。

- MDB の概要
- MDB の例
- OC4J JMS を使用する MDB
- Oracle JMS を使用する MDB
- MDB のクライアント・アクセス
- Windows で MDB を使用する場合の考慮事項
- RAC データベース使用時のフェイルオーバー

この章の MDB の例は、OTN-J のサイト `http://otn.oracle.co.jp/sample_code/index.html` の OC4J のサンプル・コードのページからダウンロードしてください。

MDB の概要

Message-Driven Bean（MDB）は、キューまたはトピックから確実にメッセージをコンシュームできる Java Message Service（JMS）メッセージ・リスナーです。MDB は、JMS リスナーの非同期的な性質とあわせて、次の操作を実行する EJB コンテナの利点を活用します。

- EJB コンテナによって、リスナーに対してタイプ `QueueReceiver` または `TopicSubscriber` のコンシューマが作成されます。
- デプロイ時に、EJB コンテナによって、MDB はコンシューマ（`QueueReceiver` または `TopicSubscriber`）およびそのファクトリに登録されます。
- EJB コンテナによって、メッセージ通知モードが指定されます。

通常の JMS オブジェクト内では、JMS メッセージ・リスナーが存在し、そのコード内でコンシューマとそのファクトリを明示的に指定する必要があります。MDB を使用すると、コンシューマとそのファクトリはコンテナによって指定されます。したがって、MDB は JMS メッセージ・リスナーを作成する簡単な手段です。オブジェクトの取得やインタフェースを使用した作成はユーザーが行う必要がありますが、ほとんどの作業はコンテナによって実行されます。

OC4J MDB は JMS プロバイダと対話します。この章では、OC4J JMS と Oracle JMS の 2 つの JMS プロバイダについて説明します。各プロバイダは、適切にインストールおよび構成されている必要があります。

- OC4J JMS は、OC4J コードベース内で内部的にインストールされます。
- Oracle JMS（Advanced Queuing）は、Oracle データベース内にインストールされ、構成されます。Oracle JMS を使用する前に、データベースに適切なキューまたは表を作成しておく必要があります。

注意： 各 JMS プロバイダの使用方法の詳細は、『Oracle Application Server Containers for J2EE サービス・ガイド』の JMS の章を参照してください。また、セキュリティの詳細は、『Oracle Application Server Containers for J2EE セキュリティ・ガイド』を参照してください。

JMS プロバイダを使用して MDB を作成し、使用可能にする一般的な手順は次のとおりです。

1. JMS プロバイダをインストールします。
2. JMS プロバイダ、MDB 用の `Destination` オブジェクト、およびプロバイダがインストールされる MDB に対する接続詳細を構成します。
3. JMS プロバイダの詳細を使用して、OC4J XML ファイルで OC4J を構成します。
4. MDB を実装し、使用される JMS の `Destination` オブジェクトをそのデプロイメント・ディスクリプタでマッピングします。

この章では、OC4J JMS プロバイダと Oracle JMS プロバイダの両方に関して、これらの各手順を実施する方法を説明します。各項では MDB の例を使用します。この例は、OTN-J のサイト `http://otn.oracle.co.jp/sample_code/index.html` の OC4J のサンプル・コードのページからダウンロードできます。

主な MDB の実装と EJB デプロイメント・ディスクリプタは、両方の JMS タイプで同じです。これについては、7-3 ページの「MDB の例」で示します。この MDB および JMS の構成に対する OC4J 固有のデプロイメント・ディスクリプタは、各 JMS タイプによって異なります。これらについては、各プロバイダの項で個別に説明します。

- OC4J JMS を使用する MDB
- Oracle JMS を使用する MDB

MDB の例

MDB は、着信非同期リクエストを処理できます。MDB のメッセージは、キューまたはトピックから、MDB の `onMessage` メソッドにルーティングされます。他のクライアントは、同じキューまたはトピックにアクセスして、MDB に対するメッセージを送信します。ほとんどの MDB は、メッセージをキューまたはトピックから受信し、メッセージ内のリクエストを処理するために、Entity Bean を起動します。

MDB の作成手順は次のとおりです。これについては、次の項で説明します。

1. 7-4 ページの「MDB 実装の例」で示すように、Bean を実装します。
2. MDB デプロイメント・ディスクリプタを作成します。
   a. 使用する JMS コネクション・ファクトリおよび `Destination` を EJB デプロイメント・ディスクリプタ（`ejb-jar.xml`）で定義します。永続的なサブスクリプションまたはメッセージ・セレクタを使用するかどうかを定義します。詳細は、7-9 ページの「MDB の EJB デプロイメント・ディスクリプタ（ejb-jar.xml）」を参照してください。
   b. リソース参照を使用する場合は、これらを `ejb-jar.xml` ファイルで定義し、OC4J 固有のデプロイメント・ディスクリプタ（`orion-ejb-jar.xml`）で実際の JNDI 名にマッピングします。
   c. MDB でコンテナ管理のトランザクション境界が使用される場合は、`ejb-jar.xml` ファイルの `<container-transaction>` 要素に `onMessage` メソッドを指定します。MDB に関するすべての手順は、`onMessage` メソッドに記述されている必要があります。MDB はステートレスであるため、`onMessage` メソッドがすべての作業を実行する必要があります。`ejbCreate` メソッドに、JMS コネクションおよびセッションを作成しないでください。ただし、OracleAS JMS を使用している場合は、JMS コネクションとセッションを `ejbCreate` メソッドで作成し、`ejbRemove` メソッドで破棄することで、MDB を最適化できます。

3. Bean およびデプロイメント・ディスクリプタを含める EJB JAR ファイルを作成します。アプリケーション固有の `application.xml` ファイルを構成し、EAR ファイルを作成し、EJB を OC4J にインストールします。

MDB の実装と `ejb-jar.xml` デプロイメント・ディスクリプタは、OC4J JMS プロバイダと Oracle JMS プロバイダで完全に同じです（コネクション・ファクトリおよび `Destination` オブジェクトの JNDI ルックアップに対してリソース参照を使用する場合）。`orion-ejb-jar.xml` デプロイメント・ディスクリプタには、リソース参照のマッピングなど、プロバイダ固有の構成が含まれます。`orion-ejb-jar.xml` デプロイメント・ディスクリプタの固有の構成については、7-11 ページの「OC4J JMS を使用する MDB」および 7-17 ページの「Oracle JMS を使用する MDB」を参照してください。

注意： MDB の例に使用されている例では、MDB に汎用性を持たせるためリソース参照が使用されています。各 JMS プロバイダに対して JNDI 文字列を明示的に定義する方法については、7-27 ページの「MDB のクライアント・アクセス」を参照してください。これは、クライアントでは、明示的な JNDI 文字列とリソース参照の両方が使用されるためです。

MDB 実装の例

MDB を実装するときに実行する主なポイントは、次のとおりです。

注意： MDB の実装に関するその他の詳細は、EJB の仕様を参照してください。

---

1. Bean クラスは、（`final` または `abstract` ではなく）`public` で定義する必要があります。
2. Bean クラスで、`javax.ejb.MessageDrivenBean` および `javax.jms.MessageListener` インタフェースを実装する必要があります。次のものが含まれます。
   - `MessageListener` インタフェースの `onMessage` メソッド
   - `MessageDrivenBean` インタフェースの `setMessageDrivenContext` メソッド
3. Bean クラスでは、通常は EJB ホーム・インタフェースのメソッドに一致するコンテナのコールバック・メソッドを実装する必要があります。リモート・インタフェース、ローカル・インタフェースおよびホーム・インタフェースは、MDB では実装されません。ただし、これらのインタフェースに必要なコールバック・メソッドの一部は、Bean 実装で実装されます。これには、次のメソッドが含まれます。
   - `ejbCreate` メソッド
   - `ejbRemove` メソッド

例 7-1 MDB の実装

次の MDB の例 rpTestMdb MDB は、キューを介して送信されたメッセージを出力し、応答します。キューは、デプロイメント・ディスクリプタおよび JMS 構成で識別されます。onMessage メソッドで、MDB はクライアントに送信される新規メッセージを作成します。また、メッセージ・セレクタのプロパティ RECIPIENT を CLIENT に設定します。次に、応答先を設定し、新規メッセージを JMS クライアントに送信します。

この例では、キューからメッセージを受信し、応答を送信する方法を示します。メッセージを受信する方法はいくつかあります。この例では、Message オブジェクトのメソッドを使用して、メッセージのすべての属性を取得します。

応答をキューに送信するには、最初にセンダーを設定する必要があります。次のことを実行する必要があります。

1. QueueConnectionFactory オブジェクトを取得します。この例では、リソース参照 "jms/myQueueConnectionFactory" が使用されます。この参照は、ejb-jar.xml ファイルで定義され、orion-ejb-jar.xml ファイルで実際の JNDI 名にマッピングされています。
2. QueueConnectionFactory オブジェクトの createQueueConnection メソッドを使用して、JMS キュー・コネクションを作成します。
3. QueueConnection オブジェクトの createQueueSession メソッドを使用して、コネクション上に JMS セッションを作成します。
4. セッションの設定後、QueueSession オブジェクトの createSender メソッドを通じてセッションを使用するセンダーを作成します。

これらの手順の実装例は、次のとおりです。

```
private QueueConnection         m_qc    = null;
private QueueSession            m_qs    = null;
private QueueSender             m_snd   = null;
QueueConnectionFactory qcf = (QueueConnectionFactory)
     ctx.lookup("java:comp/env/jms/myQueueConnectionFactory");
m_qc = qcf.createQueueConnection();
m_qs = m_qc.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
m_snd = m_qs.createSender(null);
```

センダーが作成されると、QueueSender オブジェクトの send メソッドを使用してメッセージを送信できます。この例では、受信したメッセージから応答を作成し、センダーを使用してその応答を送信します。

5. Message オブジェクトの createMessage メソッドを使用してメッセージを作成します。
6. Message オブジェクトの setStringProperty や setIntProperty などのメソッドを使用して、メッセージのプロパティを設定します。

7. この例では、Message オブジェクトの getJMSReplyTo メソッドを通じてその応答先を取得します。応答先は、センダーによってメッセージ内で初期化されています。

8. QueueSender オブジェクトの send メソッドを通じて、センダーを使用して応答を送信します。応答先および応答メッセージを指定します。

```
Message rmsg = m_qs.createMessage();
rmsg.setStringProperty("RECIPIENT", "CLIENT");
rmsg.setIntProperty("count",
msg.getIntProperty("JMSXDeliveryCount"));
rmsg.setJMSCorrelationID(msg.getJMSMessageID());
Destination d = msg.getJMSReplyTo();
m_snd.send((Queue) d, rmsg);
```

例 7-2 MDB の実装

次に、メッセージを受信し、応答を返信する MDB の完全な例を示します。

```
import java.util.*;
import javax.ejb.*;
import javax.jms.*;
import javax.naming.*;

public class rpTestMdb implements MessageDrivenBean, MessageListener
{
    private QueueConnection         m_qc    = null;
    private QueueSession            m_qs    = null;
    private QueueSender             m_snd   = null;
    private MessageDrivenContext    m_ctx   = null;

    /* Constructor, which is public and takes no arguments.*/
    public rpTestMdb()
    {    }

    /* setMessageDrivenContext method */
    public void setMessageDrivenContext(MessageDrivenContext ctx)
    {
      /* As with all EJBs, you must set the context in order to be
         able to use it at another time within the MDB methods. */
      m_ctx = ctx;
    }

   /* ejbCreate method, declared as public (but not final or
    * static), with a return type of void, and with no arguments.
    */
    public void ejbCreate()
    {    }
```

```
    /* ejbRemove method */
    public void ejbRemove()
    {    }

   /**
    * onMessage method
    * Receives the incoming Message and displays the text.
    */
    public void onMessage(Message msg)
    {
       /* An MDB does not carry state for an individual client. */
       try
        {
            Context ctx = new InitialContext();
          // 1. Retrieve the QueueConnectionFactory using a
          // resource reference defined in the ejb-jar.xml file.
            QueueConnectionFactory qcf = (QueueConnectionFactory)
                ctx.lookup("java:comp/env/jms/myQueueConnectionFactory");
            ctx.close();

            /*You create the queue connection first, then a session
              over the connection. Once the session is set up, then
              you create a sender */
          // 2. Create the queue connection
            m_qc = qcf.createQueueConnection();
          // 3. Create the session over the queue connection.
            m_qs = m_qc.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
// 4. Create the sender to send messages over the session.
            m_snd = m_qs.createSender(null);

            /* When the onMessage method is called, a message has
               been sent. You can retrieve attributes of the message using the
             Message object. */
            String txt = ("mdb rcv: " + msg.getJMSMessageID());
            System.out.println(txt + " redel="
                + msg.getJMSRedelivered() + " cnt="
                + msg.getIntProperty("JMSXDeliveryCount"));

            /* Create a new message using the createMessage
              method. To send it back to the originator of the other message,
            set the String property of ""RECIPIENT" to "CLIENT."
            The client only looks for messages with string property CLIENT.
            Copy the original message ID into new msg's Correlation ID for
            tracking purposes using the setJMSCorrelationID method. Finally,
              set the destination for the message using the getJMSReplyTo method
            on the previously received message. Send the message using the
            send method on the queue sender.
```

```
            */
             // 5. Create a message using the createMessage method
              Message rmsg = m_qs.createMessage();
             // 6. Set properties of the message.
              rmsg.setStringProperty("RECIPIENT", "CLIENT");
              rmsg.setIntProperty("count",
                  msg.getIntProperty("JMSXDeliveryCount"));
              rmsg.setJMSCorrelationID(msg.getJMSMessageID());
              // 7. Retrieve the reply destination.
              Destination d = msg.getJMSReplyTo();
                //  8. Send the message using the send method of the sender.
              m_snd.send((Queue) d, rmsg);

              System.out.println(txt + " snd: " + rmsg.getJMSMessageID());

              /* close the connection*/
              m_qc.close();
          }
          catch (Throwable ex)
          {
              ex.printStackTrace();
          }
      }
}
```

注意： MDB の例全体は、OTN-J のサイト http://otn.oracle.co.jp/sample_code/index.html の OC4J のサンプル・コードのページからダウンロードしてください。

MDB の EJB デプロイメント・ディスクリプタ（ejb-jar.xml）

EJB デプロイメント・ディスクリプタ（ejb-jar.xml）内の <message-driven> 要素で、MDB の名前、クラス、JNDI 参照および JMS の Destination タイプ（キューまたはトピック）を定義します。トピックが指定されている場合は、それが永続的であるかどうかを定義します。リソース参照を使用している場合は、コネクション・ファクトリと Destination オブジェクトの両方に対してリソース参照を定義します。

次の例は、<message-driven> 要素で rpTestMdb MDB のデプロイ情報を示します。

- MDB 名は <ejb-name> 要素で指定されます。
- MDB クラスは <ejb-class> 要素で定義されます。この要素によって、<message-driven> 要素が特定の MDB 実装に結び付けられます。
- JMS の Destination タイプは、<message-driven-destination> の <destination-type> 要素で指定されている Queue です。
- メッセージ・セレクタによって、この MDB は、RECIPIENT が MDB のメッセージのみを受信することが指定されます。

> **注意：** このタイプ定義でトピックも指定できます。タイプに Topic を指定した場合は、トピックの永続性も定義できます。これを指定するには、<message-driven-destination> の <subscription-durability> 要素を "Durable" または "nonDurable" に設定します。

- 使用するトランザクションのタイプは、<transaction-type> 要素で定義されます。値は、Container または Bean のいずれかです。Container を指定した場合は、<container-transaction> 要素内で CMT サポート・タイプを指定して onMessage メソッドを定義します。
- コネクション・ファクトリのリソース参照は <resource-ref> 要素で定義されます。Destination オブジェクトのリソース参照は <resource-env-ref> 要素で定義されます。JMS オブジェクト・タイプに対するリソース参照の詳細は、7-33 ページの「クライアントが MDB にアクセスするときに論理名を使用する方法」を参照してください。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
2.0//EN' 'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>

<ejb-jar>
    <display-name>Mdb Test</display-name>
    <enterprise-beans>
        <message-driven>
            <display-name>testMdb</display-name>
            <ejb-name>testMdb</ejb-name>
            <ejb-class>rpTestMdb</ejb-class>
```

```
            <transaction-type>Container</transaction-type>
            <message-selector>RECIPIENT='MDB'</message-selector>
            <message-driven-destination>
                <destination-type>javax.jms.Queue</destination-type>
            </message-driven-destination>
            <resource-ref>
                <description>description</description>
                <res-ref-name>jms/myQueueConnectionFactory</res-ref-name>
                <res-type>javax.jms.QueueConnectionFactory</res-type>
                <res-auth>Application</res-auth>
                <res-sharing-scope>Shareable</res-sharing-scope>
            </resource-ref>
            <resource-env-ref>
               <resource-env-ref-name>jms/persistentQueue
             </resource-env-ref-name>
             <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
            </resource-env-ref>
        </message-driven>
    </enterprise-beans>

    <assembly-descriptor>
        <container-transaction>
            <method>
                <ejb-name>testMdb</ejb-name>
                <method-name>onMessage</method-name>
                <method-params>
                    <method-param>javax.jms.Message</method-param>
                </method-params>
            </method>
            <trans-attribute>Required</trans-attribute>
        </container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

キューのかわりに永続的な Topic を構成しようとしている場合は、<message-driven-destination> 要素を次のように構成します。

```
<message-driven-destination>
  <destination-type>javax.jms.Topic</destination-type>
  <subscription-durability>Durable</subscription-durability>
</message-driven-destination>
```

注意： MDB の例全体は、OTN-J のサイト http://otn.oracle.co.jp/sample_code/index.html の OC4J のサンプル・コードのページからダウンロードしてください。

この MDB および JMS プロバイダの必須構成に対する OC4J 固有のデプロイメント・ディスクリプタ（orion-ejb-jar.xml）については、次の各項で説明します。

- OC4J JMS を使用する MDB
- Oracle JMS を使用する MDB

クライアントが MDB に JMS メッセージを送信する方法については、7-27 ページの「MDB のクライアント・アクセス」で説明します。

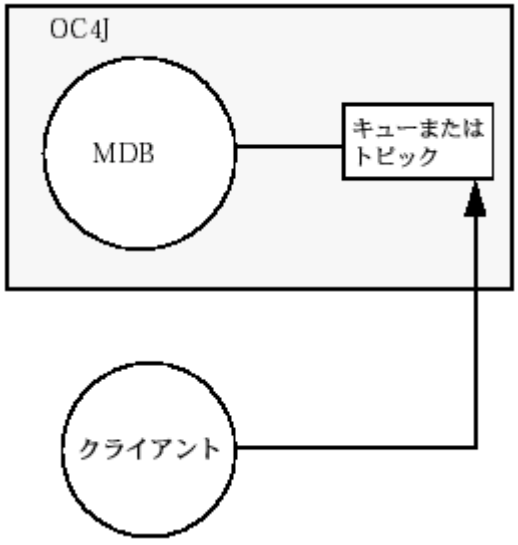
OC4J JMS を使用する MDB

MDB は、OC4J JMS を使用して着信非同期リクエストを処理できます。OC4J JMS を使用している場合、この JMS プロバイダは OC4J にバンドルされているため、すでに使用可能な状態です。JMS プロバイダに関する構成はすべて OC4J XML ファイルに記述されているため、実行する必要がある手順は 3 ～ 4 つ（7-2 ページの「MDB の概要」を参照）です。

注意： MDB の例全体は、OTN-J のサイト http://otn.oracle.co.jp/sample_code/index.html の OC4J のサンプル・コードのページからダウンロードしてください。

図 7-1 に、OC4J 内に内部的に格納されている OC4J JMS のキューまたはトピックに、クライアントが非同期リクエストを直接送信する方法を示します。MDB は OC4J JMS から直接メッセージを受信します。

図 7-1 MDB と OC4J JMS Destination との対話例



次の各項では、OC4J JMS を JMS プロバイダとして使用する MDB について説明します。

- XML ファイルでの OC4J JMS の構成
- OC4J JMS を使用する OC4J 固有のデプロイメント・ディスクリプタ（orion-ejb-jar.xml）の作成
- MDB のデプロイ

注意： 各 JMS プロバイダの使用方法の詳細は、『Oracle Application Server Containers for J2EE サービス・ガイド』の JMS の章を参照してください。

XML ファイルでの OC4J JMS の構成

OC4J JMS は自動的に使用可能になります。実行する必要があるのは、MDB で使用される JMS の Destination オブジェクトの構成のみです。照会などのために MDB でデータベースにアクセスする場合は、使用される DataSource を構成できます。JMS の構成の詳細は、7-13 ページの「JMS の Destination オブジェクトの構成」を参照してください。データ・ソースの構成の詳細は、『Oracle Application Server Containers for J2EE サービス・ガイド』のデータ・ソースの章を参照してください。

JMS の Destination オブジェクトの構成

jms.xml ファイルでトピックまたはキューを構成します。クライアントは、MDB 宛のすべてのメッセージをこのトピックまたはキューに送信します。いずれの Destination タイプの場合も、名前、位置およびコネクション・ファクトリを指定する必要があります。

次の jms.xml ファイルの構成では、rpTestMdb の例で使用される jms/Queue/rpTestQueue という名前のキューが指定されます。キュー・コネクション・ファクトリは、jms/Queue/myQCF で定義されます。また、トピックの名前は jms/Topic/rpTestTopic、コネクション・ファクトリは jms/Topic/myTCF で定義されます。

```
<?xml version="1.0" ?>
<!DOCTYPE jms-server PUBLIC "OC4J JMS server" "http://xmlns.oracle.com/ias/dtds
/jms-server.dtd">

<jms-server port="9128">
   <queue location="jms/Queue/rpTestQueue"> </queue>
   <queue-connection-factory location="jms/Queue/myQCF">
   </queue-connection-factory>

   <topic location="jms/Topic/rpTestTopic"> </topic>
   <topic-connection-factory location="jms/Topic/myTCF">
   </topic-connection-factory>

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

OC4J JMS を使用する OC4J 固有のデプロイメント・ディスクリプタ（orion-ejb-jar.xml）の作成

OC4J 固有のデプロイメント・ディスクリプタで、次のように構成します。

- orion-ejb-jar.xml ファイルの <message-driven-deployment> 要素を使用して、Destination およびコネクション・ファクトリの JNDI ロケーションを MDB に対して指定します。詳細は、7-24 ページの「Destination およびコネクション・ファクトリの指定」を参照してください。
- ejb-jar.xml ファイルでリソース参照として定義された論理名を、適切なキューまたはトピックに関連付けます。キューまたはトピックは、OC4J JMS の場合は jms.xml ファイルに定義されます。jms.xml ファイルに複数のトピックおよびキューを定義できます。orion-ejb-jar.xml ファイルでリソース参照をマッピングする方法の詳細は、7-26 ページの「リソース参照の JNDI 名へのマッピング」を参照してください。

  この例では、ejb-jar.xml ファイルのリソース参照が使用されるため、orion-ejb-jar.xml ファイルで、これらの論理名が、jms.xml ファイルで定義されるコネクション・ファクトリおよび JMS の Destination オブジェクトの実際の JNDI 名にマッピングされます。この例で、MDB は jms.xml ファイルに jms/Queue/rpTestQueue と定義されているキューを使用します。キュー・コネクション・ファクトリは、jms.xml に jms/Queue/myQCF と定義されています。

Destination およびコネクション・ファクトリの指定

orion-ejb-jar.xml ファイルの <message-driven-deployment> 要素を使用して、Destination およびコネクション・ファクトリの JNDI ロケーションを MDB にマッピングします。次に、rpTestMdb の例の orion-ejb-jar.xml デプロイメント・ディスクリプタを示します。このデプロイメント・ディスクリプタは、JMS の Queue を rpTestMdb MDB にマッピングし、次のものを提供します。

- EJB デプロイメント・ディスクリプタの <ejb-name> で定義されている MDB 名は、name 属性で指定されます。
- jms.xml ファイルで定義されている JMS の Destination は、destination-location 属性で指定されます。
- jms.xml ファイルで定義されている JMS の Destination Connection Factory は、connection-factory-location 属性で指定されます。
- トピックの場合は、ユーザーが定義した永続的なトピック名が subscription-name 属性で指定されます。
- listener-threads 属性で定義されているリスナー・スレッドは、オプションのパラメータです。リスナー・スレッドは、MDB のデプロイ時に作成され、トピックまたはキューで JMS の受信メッセージをリスニングするために使用されます。このスレッドは、JMS メッセージを並行してコンシュームします。デフォルトのスレッドは 1 つです。トピックの場合、スレッドは常に 1 つです。

これらすべてを <message-driven-deployment> 要素で指定すると、コンテナは適切な JMS の Destination への MDB のマッピングを識別できるようになります。

```
<enterprise-beans>
  ...
  <message-driven-deployment name="rpTestMdb"
     connection-factory-location="jms/Queue/myQCF"
     destination-location="jms/Queue/rpTestQueue" >
  </message-driven-deployment>
  ...
</enterprise-beans>
```

トピックを指定する場合は、次のようにサブスクリプション名も指定する必要があります。

```
<enterprise-beans>
  <message-driven-deployment name="rpTestMdb"
      connection-factory-location="jms/Queue/myQCF"
      destination-location="jms/Queue/rpTestQueue"
      subscription-name="MDBSUB" >
  ...
</enterprise-beans>
```

> **注意：** これらのフィールドで論理名を使用することはできません。コネクション・ファクトリおよび Destination オブジェクトの両方に対して、完全な JNDI 構文を指定する必要があります。

リソース参照の JNDI 名へのマッピング

コネクション・ファクトリおよび Destination オブジェクトのリソース参照として論理名を定義するときは、これらを実際の JNDI 名にマッピングする必要があります。

- <resource-ref-mapping> 要素でキュー・コネクション・ファクトリのリソース参照をマッピングします。rpTestMdb の例では、コネクション・ファクトリの論理名は jms/myQueueConnectionFactory です。この論理名を、jms.xml ファイルで定義される JNDI 文字列 jms/Queue/myQCF にマッピングする必要があります。
- <resource-env-ref-mapping> 要素で Destination オブジェクトのリソース参照をマッピングします。rpTestMdb の例では、キューの論理名は jms/persistentQueue です。この論理名が、jms.xml ファイルで定義される JNDI 文字列 jms/Queue/rpTestQueue にマッピングされます。

```
<resource-ref-mapping name="jms/myQueueConnectionFactory"
       location="jms/Queue/myQCF"/>
<resource-env-ref-mapping name="jms/persistentQueue"
       location="jms/Queue/rpTestQueue" />
```

例 7-3 rpTestMdb の例の orion-ejb-jar.xml ファイル

次に、rpTestMdb の例の完全な orion-ejb-jar.xml ファイルを示します。OC4J JMS オブジェクトの定義とリソース参照のマッピングの両方が含まれています。

```
<enterprise-beans>
  <message-driven-deployment name="testMdb"
     connection-factory-location="jms/Queue/myQCF"
    destination-location="jms/Queue/rpTestQueue" listener-threads="1">

    <resource-ref-mapping name="jms/myQueueConnectionFactory"
       location="jms/Queue/myQCF"/>
    <resource-env-ref-mapping name="jms/persistentQueue"
       location="jms/Queue/rpTestQueue" />
  </message-driven-deployment>
</enterprise-beans>
<assembly-descriptor>
  <default-method-access>
    <security-role-mapping name="&lt;default-ejb-caller-role&gt;"
                           impliesAll="true" />
  </default-method-access>
</assembly-descriptor>
```

MDB のデプロイ

EJB を JAR ファイルにアーカイブします。MDB は、Session Bean と同様にデプロイします。これについては、2-26 ページの「EJB アプリケーションのデプロイ準備」および 2-28 ページの「エンタープライズ・アプリケーションの OC4J へのデプロイ」で説明されています。

注意： クライアントが MDB に JMS メッセージを送信する方法については、7-27 ページの「MDB のクライアント・アクセス」で説明します。

Oracle JMS を使用する MDB

MDB は、Oracle JMS（Advanced Queuing）を使用して、着信非同期リクエストを次のように処理します。

警告： MDB は特定のバージョンの Oracle データベースでのみ動作します。詳細は、『Oracle Application Server Containers for J2EE サービス・ガイド』の JMS の章に記載されている動作保証のマトリックスを参照してください。

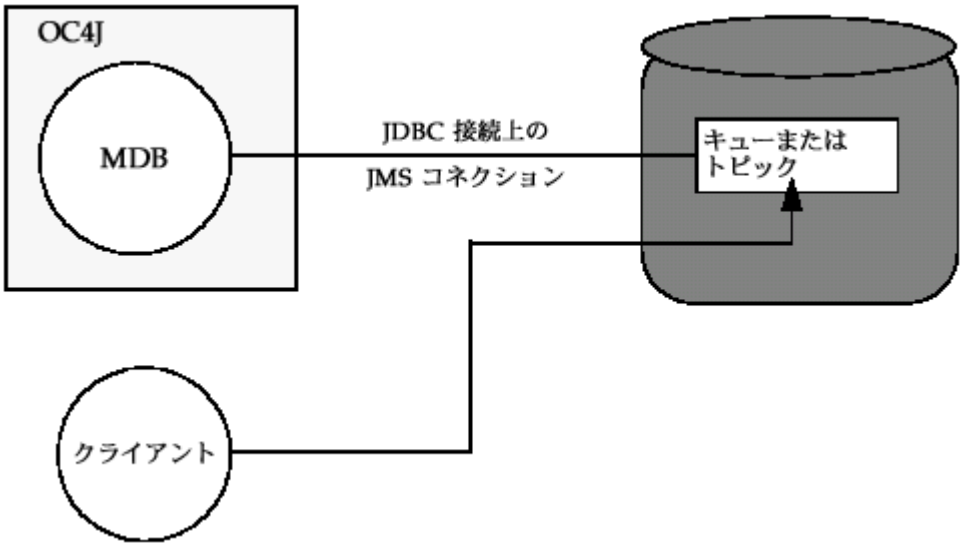
---

1. MDB は、ユーザー名とパスワードを持つデータ・ソースを使用して、JMS コネクションをデータベースに対してオープンします。このデータ・ソースは Oracle JMS プロバイダを表し、JDBC ドライバを使用して JMS コネクションを提供します。
2. MDB は、JMS コネクション上で JMS セッションをオープンします。
3. MDB のメッセージは、MDB の `onMessage` メソッドにルーティングされます。

クライアントはいつでも、MDB がリスニングしている Oracle JMS のトピックまたはキューにメッセージを送信できます。Oracle JMS のトピックまたはキューは、データベースにあります。

注意： MDB の例全体は、OTN-J のサイト `http://otn.oracle.co.jp/sample_code/index.html` の OC4J のサンプル・コードのページからダウンロードしてください。

図 7-2 MDB と Oracle JMS Destination との対話例



次の各項では、Oracle JMS を JMS プロバイダとして使用する MDB について説明します。

- JMS プロバイダのインストールと構成
- JMS プロバイダ用の OC4J XML ファイルの構成
- Oracle JMS を使用する OC4J 固有のデプロイメント・ディスクリプタの作成
- MDB のデプロイ

> **注意：** Oracle JMS プロバイダの使用方法の詳細は、『Oracle Application Server Containers for J2EE サービス・ガイド』の JMS の章を参照してください。また、『Oracle9*i* アプリケーション開発者ガイド - アドバンスト・キューイング』も参照してください。

JMS プロバイダのインストールと構成

ユーザーまたは DBA は、『Oracle9*i* アプリケーション開発者ガイド - アドバンスト・キューイング』および汎用のデータベース・マニュアルに従って Oracle JMS をインストールする必要があります。JMS プロバイダをインストールおよび構成した後、MDB ごとに追加の構成を適用する必要があります。次のものが含まれます。

1. ユーザーまたは DBA は、MDB でデータベースに接続するための RDBMS ユーザーを作成する必要があります。このユーザーに、Oracle JMS 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。7-19 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 は、JMS の `Destination` オブジェクトをサポートする表およびキューを作成する必要があります。7-20 ページの「JMS の Destination オブジェクトの作成」を参照してください。

> **注意：** 次の各項では、キュー、トピック、それらの表の作成および権限の割当てに SQL を使用します。この SQL は、OTN-J のサイト `http://otn.oracle.co.jp/sample_code/index.html` の OC4J のサンプル・コードのページの MDB デモ内に用意されています。

ユーザーの作成と権限の割当て

MDB がデータベースに接続するための RDBMS ユーザーを作成します。このユーザーに、Oracle JMS 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、リクエストする機能によって決まります。各機能に必要な権限の詳細は、『Oracle9*i* アプリケーション開発者ガイド - アドバンスト・キューイング』を参照してください。

次の例では、Oracle JMS 操作に必要な権限のある `jmsuser` を作成します。このユーザーは、そのスキーマ内で作成する必要があります。これら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・コミットやシステム管理権限など、他の権限の付与が必要な場合があります。2 フェーズ・コミット権限の詳細は、『Oracle Application Server Containers for J2EE サービス・ガイド』の JTA の章を参照してください。

JMS の Destination オブジェクトの作成

各 JMS プロバイダには、JMS の Destination オブジェクトを作成するための独自のメソッドが必要です。DBMS_AQADM パッケージおよび Oracle JMS のメッセージ・タイプの詳細は、『Oracle9*i* アプリケーション開発者ガイド - アドバンスト・キューイング』を参照してください。この例では、Oracle JMS で次のメソッドが必要です。

> **注意：** Oracle JMS の表を作成する SQL の例は、OTN-J のサイト http://otn.oracle.co.jp/sample_code/index.html の OC4J のサンプル・コードのページから入手できる MDB の例に含まれています。

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

   Oracle JMS では、トピックとキューの両方がキュー表を使用します。rpTestMdb JMS の例では、キュー用に単一の表 rpTestQTab が作成されます。

   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            => 'rpTestQTab',
           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示します。したがって、このパラメータは常に、キューについては false、トピックについては true に設定します。

2. JMS の Destination を作成します。トピックを作成する場合は、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。rpTestMdb JMS の例では、単一のキュー rpTestQueue が必要です。

   次の例では、キュー表 rpTestQTab 内に rpTestQueue というキューを作成します。作成後にキューを開始します。

   ```
   DBMS_AQADM.CREATE_QUEUE(
         Queue_name          => 'rpTestQueue',
         Queue_table         => 'rpTestQTab');

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

   トピックを追加する場合のために、次の例で、トピック表 rpTestTTab 内に rpTestTopic というトピックを作成する方法を示します。作成後に、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、トピックが開始され、ユーザーにそのトピックに関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             => 'rpTestTTab',
        Queue_payload_type      => 'SYS.AQ$_JMS_MESSAGE',
        multiple_consumers  => true,
        compatible              => '8.1.5');
DBMS_AQADM.CREATE_QUEUE( 'rpTestTopic', 'rpTestTTab');
DBMS_AQADM.ADD_SUBSCRIBER('rpTestTopic',
                        sys.aq$_agent('MDSUB', null, null));
DBMS_AQADM.ADD_SUBSCRIBER('rpTestTopic',
                        sys.aq$_agent('MDSUB2', null, null));
DBMS_AQADM.START_QUEUE('rpTestTopic');
```

注意： ここで定義する名前は、orion-ejb-jar.xml ファイルでキューまたはトピックを定義するために使用した名前と同じ名前にする必要があります。

JMS プロバイダ用の OC4J XML ファイルの構成

Oracle JMS プロバイダを使用するには、OC4J XML ファイルで次のものを構成する必要があります。

- データ・ソースの構成
- Oracle JMS データ・ソースの JNDI 名の識別

データ・ソースの構成

Oracle JMS プロバイダがインストールされているデータベースに対してデータ・ソースを構成します。JMS のトピックおよびキューは、データベース表とキューを使用してメッセージ機能を提供します。使用するデータ・ソースのタイプは、必要な機能によって決まります。

トランザクション機能 トランザクションがないか、1 フェーズのトランザクションの場合は、エミュレートされた、またはエミュレートされていないデータ・ソースのいずれも使用できます。2 フェーズ・コミットのトランザクションをサポートする場合は、エミュレートされていないデータ・ソースのみ使用できます。

例 7-4 Thin JDBC ドライバを使用するエミュレートされたデータ・ソース

次の例では、Thin JDBC ドライバを使用するエミュレートされたデータ・ソースを示します。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・ソースの違いは、『Oracle Application Server Containers for J2EE サービス・ガイド』のデータ・ソースの章を参照してください。

この例は、XML 定義の書式で表示されています。EM ツールを使用して新規データ・ソースを構成に追加する方法は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』を参照してください。

```
<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:@myhost.foo.com:1521:orcl"
/>
```

このデータ・ソースを現在の環境にあわせてカスタマイズしてください。たとえば、`mysun:1521:orcl` の部分は、ホスト名、ポートおよび使用しているデータベースの SID に置換してください。

> **注意：** パスワードをクリアテキストで指定するかわりに、間接的にパスワードを使用できます。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。

Oracle JMS データ・ソースの JNDI 名の識別

`<resource-provider>` 要素内で、Oracle JMS プロバイダとして使用されるデータ・ソースの JNDI 名を識別します。

- すべてのアプリケーション用（グローバル）の JMS プロバイダの場合は、グローバルな `application.xml` ファイルを構成します。
- 1 つのアプリケーション用（ローカル）の JMS プロバイダの場合は、アプリケーションの `orion-application.xml` ファイルを構成します。

次のコード例は、Oracle JMS の 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。

- class 属性 : Oracle JMS プロバイダは、class 属性で構成される oracle.jms.OjmsContext クラスによって実装されます。
- property 属性 : property 要素内で、この JMS プロバイダとして使用されるデータ・ソースを識別します。トピックまたはキューは、このデータ・ソースに接続して、メッセージ機能を提供する表とキューにアクセスします。

次の例では、"jdbc/emulatedDS" によって識別されるデータ・ソースが、Oracle JMS プロバイダとして使用されるデータ・ソースです。この JNDI 名は、例 7-4 の ejb-location 要素で識別されます。この例でエミュレートされていないデータ・ソースを使用した場合、名前は location 要素内の名前と同じになります。

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

Oracle JMS を使用する OC4J 固有のデプロイメント・ディスクリプタの作成

OC4J 固有のデプロイメント・ディスクリプタで、次のように構成します。

- orion-ejb-jar.xml ファイルの <message-driven-deployment> 要素を使用して、Destination およびコネクション・ファクトリの JNDI ロケーションを MDB に対して指定します。詳細は、7-24 ページの「Destination およびコネクション・ファクトリの指定」を参照してください。
- ejb-jar.xml ファイルでリソース参照として定義された論理名を、適切なキューまたはトピックに関連付けます。キューまたはトピックは、Oracle JMS の場合は SQL を使用してデータベースで定義されています。データベースで複数のトピックおよびキューを定義できます。orion-ejb-jar.xml ファイルでリソース参照をマッピングする方法の詳細は、7-26 ページの「リソース参照の JNDI 名へのマッピング」を参照してください。

  この例では ejb-jar.xml ファイルでリソース参照が使用されるため、orion-ejb-jar.xml ファイルで、これらの論理名が、データベースで定義されているコネクション・ファクトリおよび JMS の Destination オブジェクトの実際の JNDI 名にマッピングされます。この例で、MDB はデータベースに rpTestQueue と定義されているキューを使用します。キュー・コネクション・ファクトリはデータベースでは定義されないため、任意の名前を使用できます。一貫性を維持するために、キュー・コネクション・ファクトリ名は myQCF です。

Destination およびコネクション・ファクトリの指定

orion-ejb-jar.xml ファイルの <message-driven-deployment> 要素を使用して、Destination およびコネクション・ファクトリの JNDI ロケーションを MDB にマッピングします。次に、rpTestMdb の例の orion-ejb-jar.xml デプロイメント・ディスクリプタを示します。このデプロイメント・ディスクリプタは、JMS の Queue を rpTestMdb MDB にマッピングし、次のものを提供します。

- EJB デプロイメント・ディスクリプタの <message-driven><ejb-name> で定義されている MDB 名は、name 属性で指定されます。
- ユーザーが指定する JMS の Destination Connection Factory は、connection-factory-location 属性で指定されます。コネクション・ファクトリ用の Oracle JMS の構文は、"java:comp/resource" + JMS プロバイダ名 + "TopicConnectionFactories" または "QueueConnectionFactories" + ユーザー定義名です。ユーザー定義名は任意であるため、他の構成と一致しません。xxxConnectionFactories は、定義するファクトリのタイプを指定します。この例では、JMS プロバイダ名は application.xml ファイルの <resource-provider> 要素で myProvider と定義されます。
  - キュー・コネクション・ファクトリの場合 : JMS プロバイダ名が myProvider で、ユーザー定義名が myQCF であるため、コネクション・ファクトリ名は、"java:comp/resource/myProvider/QueueConnectionFactories/myQCF" です。
  - トピック・コネクション・ファクトリの場合 : JMS プロバイダ名が myProvider で、ユーザー定義名が myTCF であるため、コネクション・ファクトリ名は、"java:comp/resource/myProvider/TopicConnectionFactories/myTCF" です。

  前述の myQCF および myTCF で示されたユーザー定義名は、ロジックの他の場所では使用されません。したがって、選択する名前は任意です。
- データベースで定義されている JMS の Destination は、destination-location 要素で指定されます。Destination 用の Oracle JMS の構文は、"java:comp/resource" + JMS プロバイダ名 + "Topics" または "Queues" + Destination 名です。Topic または Queue は、定義される Destination タイプを指定します。Destination 名は、データベースで定義された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 application.xml ファイルの <resource-provider> 要素で myProvider と定義されます。データベースでは、トピック名は rpTestQueue です。
  - キューの場合 : JMS プロバイダ名が myProvider で、キュー名が rpTestQueue の場合、キューの JNDI 名は "java:comp/resource/myProvider/Queues/rpTestQueue" です。
  - トピックの場合 : JMS プロバイダ名が myProvider で、トピック名が rpTestTopic の場合、トピックの JNDI 名は "java:comp/resource/myProvider/Topics/rpTestTopic" です。

- トピックの場合は、ユーザーが定義した永続的なトピック名が subscription-name 属性で指定されます。
- リスナー・スレッドはオプションのパラメータで、listener-threads 属性で定義されます。リスナー・スレッドは、MDB のデプロイ時に作成され、トピックまたはキューで JMS の受信メッセージをリスニングするために使用されます。このスレッドは、JMS メッセージを並行してコンシュームします。デフォルトのスレッドは 1 つです。トピックは常に 1 つのスレッドのみを使用し、キューは複数のスレッドを使用できます。
- transaction-timeout 属性で定義されているトランザクション・タイムアウトは、オプションのパラメータです。この属性によって、コンテナ管理のトランザクション型 MDB のトランザクション・タイムアウト時間（秒）を制御します。デフォルトは 1 日（86,400 秒）です。この時間枠内でトランザクションが完了しない場合、トランザクションはロールバックされ、メッセージは、Destination オブジェクトに再度配信されます。JMS では、メッセージの再配信を試行します（デフォルトの試行回数は 5 回で、この値は、データベースにキューを作成するときに DBMS_AQADM.CREATE_QUEUE メソッドに設定されます）。試行後、メッセージは例外キューに移動します。例外キューのメッセージは、SQL*Plus を使用して参照できます。再配信の試行回数の設定方法と例外キューの参照方法は、『Oracle9*i* アプリケーション開発者ガイド - アドバンスト・キューイング』を参照してください。

これらすべてを <message-driven-deployment> 要素で指定すると、コンテナは適切な JMS の Destination への MDB のマッピングを識別できるようになります。

```
  <message-driven-deployment name="testMdb"
     connection-factory-location=
       "java:comp/resource/myProvider/QueueConnectionFactories/myQCF"
    destination-location="java:comp/resource/myProvider/Queues/rpTestQueue"
listener-threads="5">
```

トピックを指定する場合は、次のようにサブスクリプション名も指定する必要があります。

```
<enterprise-beans>
  <message-driven-deployment
      name="rpTestMdb"
      connection-factory-location=
          "java:comp/resource/myProvider/TopicConnectionFactories/myTCF"
      destination-location="java:comp/resource/cartojms1/Topics/rpTestTopic"
      subscription-name="MDBSUB"
      listener-threads=1 >
  ...
</enterprise-beans>
```

> **注意：** これらのフィールドで論理名を使用することはできません。コネクション・ファクトリおよび Destination オブジェクトの両方に対して、完全な JNDI 構文を指定する必要があります。

リソース参照の JNDI 名へのマッピング

コネクション・ファクトリおよび Destination オブジェクトのリソース参照として論理名を定義するときは、これらを実際の JNDI 名にマッピングする必要があります。

- <resource-ref-mapping> 要素でキュー・コネクション・ファクトリのリソース参照をマッピングします。rpTestMdb の例では、コネクション・ファクトリの論理名は jms/myQueueConnectionFactory です。この論理名を、JNDI 文字列 java:comp/resource/myProvider/QueueConnectionFactories/myQCF にマッピングする必要があります。
- <resource-env-ref-mapping> 要素で Destination オブジェクトのリソース参照をマッピングします。rpTestMdb の例では、キューの論理名は jms/persistentQueue です。この論理名は、JNDI 文字列 java:comp/resource/myProvider/Queues/rpTestQueue にマッピングされます。

Oracle JMS の JNDI 構文の構成については、7-24 ページの「Destination およびコネクション・ファクトリの指定」を参照してください。

```
<resource-ref-mapping name="jms/myQueueConnectionFactory"
       location="java:comp/resource/myProvider/QueueConnectionFactories/myQCF"/>
<resource-env-ref-mapping name="jms/persistentQueue"
       location="java:comp/resource/myProvider/Queues/rpTestQueue" />
```

例 7-5 rpTestMdb の例の orion-ejb-jar.xml ファイル

次に、rpTestMdb の例の完全な orion-ejb-jar.xml ファイルを示します。Oracle JMS オブジェクトの定義とリソース参照のマッピングの両方が含まれています。

```
<enterprise-beans>
  <message-driven-deployment name="testMdb"
     connection-factory-location=
       "java:comp/resource/myProvider/QueueConnectionFactories/myQCF"
    destination-location="java:comp/resource/myProvider/Queues/rpTestQueue"
    listener-threads="5">

    <resource-ref-mapping name="jms/myQueueConnectionFactory"
       location="java:comp/resource/myProvider/QueueConnectionFactories/myQCF"/>
    <resource-env-ref-mapping name="jms/persistentQueue"
       location="java:comp/resource/myProvider/Queues/rpTestQueue" />
  </message-driven-deployment>
</enterprise-beans>
<assembly-descriptor>
  <default-method-access>
    <security-role-mapping name="&lt;default-ejb-caller-role&gt;"
                           impliesAll="true" />
  </default-method-access>
</assembly-descriptor>
```

MDB のデプロイ

MDB を JAR ファイルにアーカイブします。MDB は、Session Bean と同様にデプロイします。これについては、2-26 ページの「EJB アプリケーションのデプロイ準備」および 2-28 ページの「エンタープライズ・アプリケーションの OC4J へのデプロイ」で説明されています。

注意： クライアントが MDB に JMS メッセージを送信する方法については、7-27 ページの「MDB のクライアント・アクセス」で説明します。

MDB のクライアント・アクセス

クライアントは、JMS の `Destination` を通じて MDB にメッセージを送信します。クライアントは、明示的な名前または論理名のいずれかを使用して、JMS の `Destination` およびコネクション・ファクトリを取得できます。次の各項では、JNDI 名を取得する両方の方法について説明します。

- JNDI ルックアップに対する明示的な名前の使用
- クライアントが MDB にアクセスするときに論理名を使用する方法

注意： クライアントが MDB と同じ場所にない場合は、JNDI プロパティの追加が必要となる場合があります。次の各項で説明する例に、JNDI プロパティの設定は含まれていません。これらのプロパティの設定方法は、2-16 ページの「JNDI プロパティの設定」を参照してください。

JNDI ルックアップに対する明示的な名前の使用

クライアント内で、実際の JNDI 名を使用して JMS の `Destination` オブジェクトを取得できます。OC4J JMS と Oracle JMS の両方に専用の命名方法論があります。これについては次の各項で説明します。

- 明示的な JNDI 名を使用した OC4J JMS の Destination へのアクセス
- 明示的な JNDI 名を使用した Oracle JMS の Destination へのアクセス

注意： `orion-ejb-jar.xml` ファイルで、`Destination` および JMS プロバイダ・オブジェクトの JNDI 名をすべてリソース参照として指定することもできます。詳細は、7-33 ページの「クライアントが MDB にアクセスするときに論理名を使用する方法」を参照してください。

明示的な JNDI 名を使用した OC4J JMS の Destination へのアクセス OC4J JMS の JNDI ルックアップでは、OC4J JMS の `Destination` およびコネクション・ファクトリは、`jms.xml`

ファイル内でユーザーが定義した名前の先頭に "java:comp/env/" を付加して使用する必要があります。OC4J JMS のキューおよびトピックがどのように構成されるかについては、7-13 ページの「JMS の Destination オブジェクトの構成」を参照してください。

> **注意：** 論理名を使用する場合には、同じ JNDI 構文を使用します。論理名は移植可能なため、論理名を使用することをお薦めします。詳細は、7-33 ページの「クライアントが MDB にアクセスするときに論理名を使用する方法」を参照してください。

testResourceProvider の例で、OC4J JMS を使用して JNDI ルックアップでキューをルックアップする方法は、次のとおりです。

```
//Lookup the Queue
queue = (Queue)jndiContext.lookup("java:comp/env/jms/Queue/rpTestQueue");

//Lookup the Queue Connection factory
queueConnectionFactory = (QueueConnectionFactory)
    jndiContext.lookup("java:comp/env/jms/Queue/myQCF");
```

トピックをルックアップする場合は文字列が多少異なり、次のように queue ではなく topic を指定します。

```
//Lookup the Topic
topic = (Topic)jndiContext.lookup("java:comp/env/jms/Topic/rpTestTopic");

//Lookup the Connection factory
topicConnectionFactory = (TopicConnectionFactory)
       jndiContext.lookup("java:comp/env/jms/Topic/myTCF");
```

クライアントの構成、jms.xml および MDB デプロイメント・ディスクリプタで、トピックとコネクション・ファクトリに同じ名前が使用されていることに注意してください。

明示的な JNDI 名を使用した Oracle JMS の Destination へのアクセス Oracle JMS を使用する場合の JNDI ルックアップでは、Oracle JMS の Destination およびコネクション・ファクトリの構文を使用する必要があります。この構文は、7-24 ページの「Destination およびコネクション・ファクトリの指定」で、connection-factory-location および destination-location 属性について説明されている命名規則と同じです。

> **注意：** 論理名を使用する場合には、同じ JNDI 構文を使用します。詳細は、7-33 ページの「クライアントが MDB にアクセスするときに論理名を使用する方法」を参照してください。

JNDI ルックアップにおけるキューおよびトピックの両方に対する実装は次のとおりです（完全な例は例 7-6 を参照してください）。

```
/* Retrieve an Oracle JMS Queue through JNDI */
queue = (Queue) ic.lookup("java:comp/resource/myProvider/Queues/rpTestQueue");
/*Retrieve the Oracle JMS Queue connection factory */
queueConnectionFactory = (QueueConnectionFactory) ic.lookup
     ("java:comp/resource/myProvider/QueueConnectionFactories/myQCF");

/* Retrieve an Oracle JMS Topic through JNDI */
topic = (Topic) ic.lookup("java:comp/resource/myProvider/Topics/rpTestTopic");
/*Retrieve the Oracle JMS Topic connection factory */
topicConnectionFactory = (TopicConnectionFactory) ic.lookup
     ("java:comp/resource/myProvider/TopicConnectionFactories/myTCF");
```

メッセージを MDB に送信する手順

実装で論理名を使用するか実際の JNDI 名を使用するかに関係なく、クライアントが JMS メッセージを MDB に送信する手順は、次のとおりです。

1. JNDI ルックアップを使用して、構成済の JMS の `Destination` とそのコネクション・ファクトリを取得します。
2. このコネクション・ファクトリからコネクションを作成します。キューに対するメッセージを受信している場合は、コネクションを開始します。
3. コネクション上にセッションを作成します。
4. 取得した JMS の `Destination` を使用して、キューのセンダーまたはトピックの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. キューのセンダーまたはトピックのパブリッシャのいずれかを使用して、メッセージを送信します。
7. キュー・セッションを閉じます。JMS のいずれかの `Destination` タイプのコネクションを閉じます。

例 7-6 クライアントがメッセージをキューに送信するサーブレット

```
public final class testResourceProvider extends HttpServlet
{
  private String resProvider = "myResProvider";
  private HashMap msgMap = new HashMap();
  Context ctx = new InitialContext();

 public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
 {
```

```
   doPost(req, res);
}

public void doPost(HttpServletRequest req, HttpServletResponse res)
   throws ServletException, IOException
{
  //Retrieve the name of the JMS provider from the request, which is
  // to be used in creating the JNDI string for retrieval
   String rp = req.getParameter ("provider");
   if (rp != null)
     resProvider = rp;

   try
   {
     // 1a. Look up the Queue Connection Factory
     QueueConnectionFactory qcf = (QueueConnectionFactory)
              ctx.lookup ("java:comp/resource/" + resProvider +
              "/QueueConnectionFactories/myQCF");
     // 1b. Lookup the Queue
     Queue queue = (Queue) ctx.lookup ("java:comp/resource/" + resProvider +
                                       "/Queues/rpTestQueue");

     // 2 & 3. Retrieve a connection and a session on top of the connection
     // 2a. Create queue connection using the connection factory.
     QueueConnection qconn = qcf.createQueueConnection();
     // 2a. We're receiving msgs, so start the connection.
     qconn.start();

     // 3. create a session over the queue connection.
     QueueSession qsess = qconn.createQueueSession(false,
                                                   Session.AUTO_ACKNOWLEDGE);

      // 4. 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);

   drainQueue (sess, q);
   TextMessage msg = null;

   /* Send msgs to queue.  */
   for (int i = 0; i < 3; i++)
   {
     // 5. Create message
     msg = sess.createTextMessage();
     msg.setText ("TestMessage:" + i);

      // set property of the recipient to be the MDB
```

```
      //and set the reply destination.
      msg.setStringProperty ("RECIPIENT", "MDB");
      msg.setJMSReplyTo(q);

      //6. send the message using the sender.
      snd.send (msg);

      // You can store the messages IDs and sent-time  in a map (msgMap),
      // so that when messages are received, you can verify if you
      // *only* received those messages that you were
       // expecting. See receiveFromMDB() method where msgMap gets used.
      msgMap.put (msg.getJMSMessageID(), new Long (msg.getJMSTimestamp()));
    }

    // receive a reply from the MDB.
    receiveFromMDB (sess, q);

     //7. Close sender, session, and connection for queue
     snd.close();
     sess.close();
     qconn.close();
    }
    catch (Exception e)
    {
      System.err.println ("** TEST FAILED **"+ e.toString());
      e.printStackTrace();
    }
    finally
    {
    }
}

 /*
  * Receive any msgs sent to us via the MDB
  */
 private void receiveFromMDB (QueueSession sess, Queue q)
   throws Exception
 {
   //The MDB sends out a message (as a reply) to this client. The MDB sets
   // the receipient as CLIENT. Thus, we will only receive msgs that have
   // RECIPIENT set to 'CLIENT'
   QueueReceiver rcv = sess.createReceiver (q, "RECIPIENT = 'CLIENT'");

   int nrcvd = 0;
   long trtimes = 0L;
   long tctimes = 0L;
   // First msg needs to come from MDB. May take  a little while
```

```
    //Receiving Messages
    for (Message msg = rcv.receive (30000); msg != null;
         msg = rcv.receive (30000))
    {
      nrcvd++;

      String rcp = msg.getStringProperty ("RECIPIENT");
      // Verify if msg in message Map
      // We check the msgMap to see if this is the message that we are
      // expecting.
      String corrid = msg.getJMSCorrelationID();
      if (msgMap.containsKey(corrid))
      {
        msgMap.remove(corrid);
      }
      else
      {
        System.err.println ("** received unexpected message
                             [" + corrid + "] **");
      }
    }
    rcv.close();
  }

  /*
   * Drain messages from queue
   */
  private int drainQueue (QueueSession sess,
                          Queue q)
    throws Exception
  {

    QueueReceiver rcv = sess.createReceiver (q);
    int nrcvd = 0;

    /*
     * First drain any old msgs from queue
     */
    for (Message msg = rcv.receive(1000);
         msg != null;
         msg = rcv.receive(1000))
      nrcvd++;
    rcv.close();

    return nrcvd;
  }
}
```

クライアントが MDB にアクセスするときに論理名を使用する方法

クライアント・アプリケーション・コードで論理名を使用する場合は、次のいずれかの XML ファイルで論理名を定義します。

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

論理名を、OC4J デプロイメント・ディスクリプタの実際のトピック名またはキュー名にマッピングします。

コネクション・ファクトリおよび 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・ファクトリは、クライアントの XML デプロイメント・ディスクリプタの <resource-ref> 要素内で識別されます。
  - コネクション・ファクトリの識別に使用する論理名は、<res-ref-name> 要素で定義されます。
  - コネクション・ファクトリのクラス・タイプは、<res-type> 要素で javax.jms.QueueConnectionFactory または javax.jms.TopicConnectionFactory に定義されます。
  - 認証機能（Container または Bean）は、<res-auth> 要素で定義されます。
  - 有効範囲の共有（Shareable または Unshareable）は、<res-sharing-scope> 要素で定義されます。
- JMS の Destination（トピックまたはキュー）は、<resource-env-ref> 要素で識別されます。
  - トピックまたはキューの識別に使用する論理名は、<resource-env-ref-name> 要素で定義されます。
  - Destination のクラス・タイプは、<resource-env-ref-type> 要素で javax.jms.Queue または javax.jms.Topic に定義されます。

次の例は、トピックの論理名の指定方法を示します。

```
<resource-ref>
  <res-ref-name>myTCF</res-ref-name>
  <res-type>javax.jms.TopicConnectionFactory</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
<resource-env-ref>
  <resource-env-ref-name>rpTestTopic</resource-env-ref-name>
```

```
  <resource-env-ref-type>javax.jms.Topic</resource-env-ref-type>
</resource-env-ref>
```

次に、OC4J のデプロイメント・ディスクリプタで、論理名を実際の名前にマッピングします。実際の名前（JNDI 名）は、OC4J JMS と Oracle JMS で異なります。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`
- クライアントとして機能する EJB の場合 : `orion-ejb-jar.xml`
- 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

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、次のようにマッピングされます。

- `<resource-ref>` 要素で定義されたコネクション・ファクトリの論理名は、`<resource-ref-mapping>` 要素内の JNDI にマッピングされます。
- `<resource-env-ref>` 要素で定義された JMS の `Destination` の論理名は、`<resource-env-ref-mapping>` 要素内の JNDI にマッピングされます。

OC4J JMS および Oracle JMS のそれぞれにおけるマッピング方法は、次の各項を参照してください。

- OC4J JMS 用の JNDI ネーミング
- Oracle JMS 用の JNDI ネーミング

OC4J JMS 用の JNDI ネーミング

OC4J JMS の Destination およびコネクション・ファクトリの JNDI 名は、`jms.xml` ファイル内でユーザーが定義します。7-13 ページの「JMS の Destination オブジェクトの構成」で示したように、トピックおよびトピック・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/Topic/rpTestTopic`" です。
- トピック・コネクション・ファクトリの JNDI 名は、"`jms/Topic/myTCF`" です。

これらの名前の先頭に "`java:comp/env/`" を付け、`orion-ejb-jar.xml` ファイルに次のようにマッピングを設定します。

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

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

Oracle JMS 用の JNDI ネーミング

Oracle JMS の Destination およびコネクション・ファクトリ・オブジェクトの JNDI ネーミングは、7-24 ページの「Destination およびコネクション・ファクトリの指定」で説明したように、MDB に対して orion-ejb-jar.xml ファイルで指定した名前と同じです。

次の例では、コネクション・ファクトリとトピックの論理名を実際の JNDI 名にマッピングします。ejb-jar.xml ファイルで "rpTestTopic" と論理的に定義されたトピックは、JNDI 名 "java:comp/resource/cartojms1/Topics/rpTestTopic" にマッピングされます。

```
<resource-ref-mapping
        name="myTCF"
       location="java:comp/resource/myProvider/TopicConnectionFactories/myTCF">
</resource-ref-mapping>

<resource-env-ref-mapping
       name="rpTestTopic"
       location="java:comp/resource/myProvider/Topics/rpTestTopic">
</resource-env-ref-mapping>
```

クライアントによる論理名を使用した JMS メッセージの送信

リソースの定義後、クライアントは次の手順で JMS メッセージを MDB に送信します。

1. JNDI ルックアップを使用して、構成済の JMS の Destination とそのコネクション・ファクトリを取得します。
2. このコネクション・ファクトリからコネクションを作成します。キューに対するメッセージを受信している場合は、コネクションを開始します。
3. コネクション上にセッションを作成します。
4. 取得した JMS の Destination を使用して、キューのセンダーまたはトピックの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. キューのセンダーまたはトピックのパブリッシャのいずれかを使用して、メッセージを送信します。
7. キュー・セッションを閉じます。JMS のいずれかの Destination タイプのコネクションを閉じます。

例 7-7 JSP クライアントによるトピックへのメッセージ送信

トピックを介してメッセージを送信する方法はほとんど同じです。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、サブスクライバを作成します。

次の JSP クライアント・コード例では、トピックを介してメッセージを MessageBean MDB に送信します。コードでは論理名を使用しています。この論理名は、OC4J デプロイメント・ディスクリプタでマッピングされている必要があります。

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

//1a. Lookup the MessageBean topic
jndiContext = new InitialContext();
topic = (Topic)jndiContext.lookup("rpTestTopic");

//1b. Lookup the MessageBean 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);

//5. Create the publisher for any messages destined for the topic
topicPublisher = topicSession.createPublisher(topic);

//6. 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. Close publisher, session, and connection for topic
topicPublisher.close();
topicSession.close();
topicConnection.close();

%>
Message sent!
```

Windows で MDB を使用する場合の考慮事項

Windows 環境で MDB を実行している場合、または Windows 環境でバックエンド・データベースが稼動中の場合は、oracle.mdb.fastUndeploy システム・プロパティによって、OC4J を正しく停止できます。通常、使用している MDB は着信メッセージを待機する受信状態にブロックされています。ただし、MDB が Windows 環境で待機状態のときに OC4J をシャットダウンした場合、MDB がブロックされているため、OC4J インスタンスは停止できません。またアプリケーションはアンデプロイされません。ただし、oracle.mdb.fastUndeploy システム・プロパティを設定することで、この環境で MDB の動作を変更できます。このプロパティを整数に設定すると、MDB が着信メッセージの処理中でなく待機状態のときに、OC4J コンテナは、データベースに移動（データベースへのラウンドトリップが必要）し、ポーリングしてセッションがシャットダウンされているかどうかを確認します。この整数は、システムがデータベースのポーリングを待機する秒数を示します。これは、パフォーマンスに影響を与える場合があります。このプロパティを 60（秒）に設定した場合、OC4J では 60 秒ごとにデータベースをチェックします。このプロパティを設定しないで、[CTRL] キーと [C] キーを使用して OC4J をシャットダウンしようとすると、OC4J のプロセスは、2.5 時間以上停止します。

RAC データベース使用時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、データベースのフェイルオーバーを処理する必要があります。MDB ランタイムは、使用可能な新規データベースにフェイルオーバーしません。フェイルオーバーを有効にするには、デプロイメント・ディスクリプタ dequeue-retry-count および dequeue-retry-interval を orion-ejb-jar.xml ファイルに指定する必要があります。最初のパラメータ dequeue-retry-count は、障害時にデータベース接続を再試行する回数をコンテナに指示します。デフォルトは 0（ゼロ）です。2 番目のパラメータ dequeue-retry-interval は、試行間の待機時間（データベースのフェイルオーバーに必要な時間）を指示します。デフォルトは 60（秒）です。

注意： データ・ソースの RAC-enabled 属性の詳細は、『Oracle Application Server Containers for J2EE サービス・ガイド』のデータ・ソースの章を参照してください（RAC は、Real Application Clusters の略称です。このフラグのインフラストラクチャ・データベースでの使用方法の詳細は、『Oracle Application Server 10*g* 高可用性ガイド』を参照してください）。

この 2 つのパラメータは、<message-driven-deployment> 要素の属性です。次に例を示します。

```
<message-driven-deployment name="MessageBeanTpc"
   connection-factory-location=
     "java:comp/resource/cartojms1/TopicConnectionFactories/aqTcf"
   destination-location=
     "java:comp/resource/cartojms1/Topics/topic1"
   subscription-name="MDBSUB"
   dequeue-retry-count=3
   dequeue-retry-interval=90/>
```

RAC データベースに対してスタンドアロンの OJMS クライアントを実行する場合は、接続を再度取得するために同じコードを記述し、API `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する必要があります。必要に応じて、データベース接続を再度確立する必要があります。次にロジックの例を示します。

```
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
            }
          }
     }
}
```

8

EJB アプリケーションのセキュリティの構成

EJB アプリケーションのセキュリティは、2 つのレルムに関連しています。1 つはブラウザにダウンロードする場合の権限の付与、もう 1 つは認証と認可を受けるためのアプリケーションの構成です。この章では、EJB のユーザー、ロールおよびグループの設定方法について説明します。ただし、CSiV2 などの基本的な OC4J セキュリティ構成の詳細は、『Oracle Application Server Containers for J2EE セキュリティ・ガイド』を参照してください。

この章には、次の内容が含まれます。

ブラウザにおける権限の付与

Security Manager がアクティブなクライアントで EJB アプリケーションをダウンロードする場合は、ダウンロードの前に次の権限を付与する必要があります。

```
permission java.net.SocketPermission "*:*", "connect,resolve";
permission java.lang.RuntimePermission "createClassLoader";
permission java.lang.RuntimePermission "getClassLoader";
permission java.util.PropertyPermission "*", "read";
permission java.util.PropertyPermission "LoadBalanceOnLookup",
"read,write";
```

EJB アプリケーションの認証と認可

EJB の認証と認可については、EJB デプロイメント・ディスクリプタを構成して、各メソッドを実行する基礎となるプリンシパルを定義します。コンテナでは、メソッドを実行するユーザーが、デプロイメント・ディスクリプタで定義されたユーザーと同じであることが必要です。

EJB デプロイメント・ディスクリプタを使用すると、各メソッドを実行できるセキュリティ・ロールを定義できます。このメソッドは、OC4J 固有のデプロイメント・ディスクリプタで、ユーザーまたはグループにマッピングされます。ユーザーとグループは、指定したセキュリティ・ユーザー・マネージャ（OracleAS JAAS Provider または XML ユーザー・マネージャのいずれかを使用）内で定義されます。セキュリティ・ユーザー・マネージャの詳細は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』および『Oracle Application Server Containers for J2EE サービス・ガイド』を参照してください。

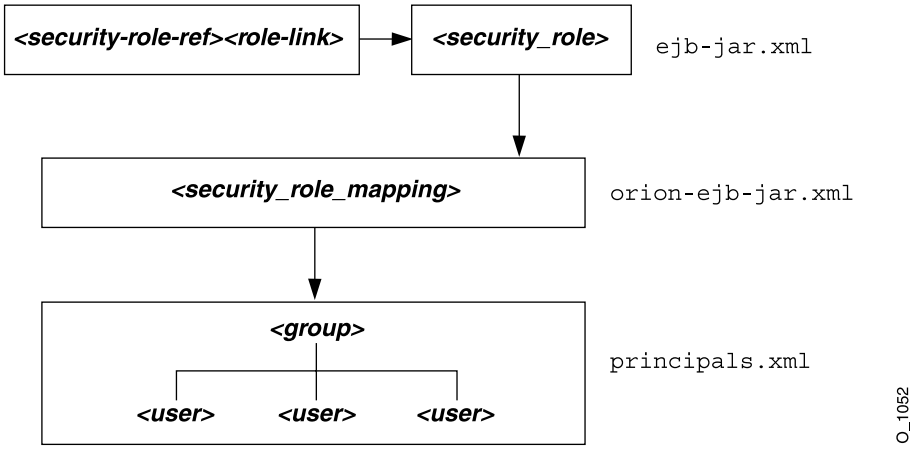
この項では、認証と認可に関して、EJB デプロイメント・ディスクリプタ内の XML 構成を説明します。EJB の認可は、EJB および OC4J 固有のデプロイメント・ディスクリプタ内で指定されます。セキュリティの許可の部分は、次のように、デプロイメント・ディスクリプタ内で管理できます。

- EJB デプロイメント・ディスクリプタは、論理ロールを使用して、アクセス・ルールを記述します。
- OC4J 固有のデプロイメント・ディスクリプタは、論理ロールを具体的なユーザーおよびグループにマッピングします。これは、OracleAS JAAS Provider または XML ユーザー・マネージャのいずれかで定義されています。

ユーザーおよびグループは、コンテナによって認識される認識情報です。ロールは、アプリケーションが、各オブジェクトへのアクセス権を示すために使用する論理識別情報です。ユーザー名およびパスワードには、デジタル証明が使用可能で、SSL の場合、秘密鍵も使用可能です。

ロールの定義およびマッピングを、図 8-1 に示します。

図 8-1 ロールのマッピング



ユーザー、グループおよびロールの定義について、次の各項で説明します。

- ユーザーおよびグループの指定
- EJB デプロイメント・ディスクリプタでの論理ロールの指定
- EJB メソッドに対するセキュリティ・チェックなしの指定
- runAs セキュリティ識別情報の指定
- ユーザーおよびグループへの論理ロールのマッピング
- 未定義メソッドに対するデフォルト・ロール・マッピングの指定
- クライアントによるユーザーとグループの指定

注意： CSiV2 などの基本的な OC4J セキュリティ構成の詳細は、『Oracle Application Server Containers for J2EE セキュリティ・ガイド』を参照してください。

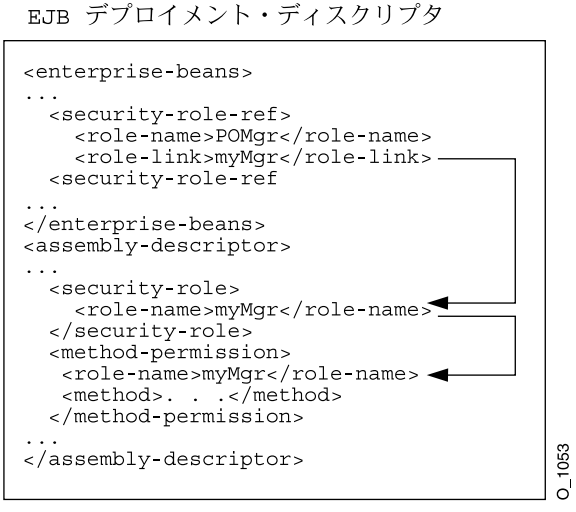
ユーザーおよびグループの指定

OC4J では、ユーザーおよびグループの定義をサポートしています。これには、すべてのデプロイ済アプリケーションで共有されているものと、特定のアプリケーション固有のものの両方が含まれます。共有またはアプリケーション固有のユーザーとグループは、OracleAS JAAS Provider または XML ユーザー・マネージャのいずれかで定義します。詳細は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』および『Oracle Application Server Containers for J2EE サービス・ガイド』を参照してください。

EJB デプロイメント・ディスクリプタでの論理ロールの指定

図 8-2 に示すように、Bean 実装内でロールの論理名を使用して、この論理名を適切なデータベース・ロールまたはユーザーにマッピングできます。論理名のデータベース・ロールへのマッピングは、OC4J 固有のデプロイメント・ディスクリプタで指定されます。詳細は、8-9 ページの「ユーザーおよびグループへの論理ロールのマッピング」を参照してください。

図 8-2 セキュリティのマッピング



isCallerInRole などのメソッドの Bean 実装内でデータベース・ロールの論理名を使用する場合は、次の手順を実行して、実際のデータベース・ロールに論理名をマッピングできます。

1. <enterprise-beans> セクションの <security-role-ref> 要素内で論理名を宣言します。たとえば、発注の例で使用するロールを定義する場合は、Bean 実装内で、コール元が発注にサインする認可を受けているかどうかをチェックしておくことができます。したがって、コール元は、適切なロールでサインオンする必要があります。Bean によるデータベース・ロールの認識を不要にするため、POMgr などの論理名で isCallerInRole をチェックできます。これは、注文を承認できるのは発注マネージャのみであるためです。したがって、論理セキュリティ・ロールの POMgr を、次のように、<enterprise-beans> セクションの <security-role-ref><role-name> 要素で定義します。

```
<enterprise-beans>
...
  <security-role-ref>
   <role-name>POMgr</role-name>
   <role-link>myMgr</role-link>
  </security-role-ref>
</enterprise-beans>
```

<security-role-ref> 要素内の <role-link> 要素は、実際のデータベース・ロールの場合があり、<assembly-descriptor> セクション内で詳細に定義されます。また、この要素は別の論理名の場合があり、<assembly-descriptor> セクションで詳細に定義され、Oracle 固有のデプロイメント・ディスクリプタ内で実際のデータベース・ロールにマッピングされます。

注意： <security-role-ref> 要素は必要ありません。この要素を指定するのは、Bean 内でセキュリティ・コンテキスト・メソッドを使用する場合のみです。

---

2. ロールおよびロールを適用するメソッドを定義します。発注の例にある PurchaseOrder Bean で実行されるメソッドは、myMgr として認可されている必要があります。PurchaseOrder は、<entity | session><ejb-name> 要素で宣言された名前です。

   次の例では、ロールを myMgr、EJB を PurchaseOrder、およびすべてのメソッドを * 記号で示して定義しています。

注意： <security-role> 要素内の myMgr ロールは、<enterprise-beans> セクション内の <role-link> 要素と同じです。これによって、POMgr の論理名が myMgr 定義に関連付けられます。

```
<assembly-descriptor>
 <security-role>
  <description>Role needed purchase order authorization</description>
  <role-name>myMgr</role-name>
 </security-role>
 <method-permission>
  <role-name>myMgr</role-name>
  <method>
   <ejb-name>PurchaseOrder</ejb-name>
   <method-name>*</method-name>
  </method>
 </method-permission>
...
</assembly-descriptor>
```

前述の 2 つの手順を実行した後は、Bean 実装内で POMgr を参照でき、コンテナは POMgr を myMgr に変換します。

> **注意：** 同じ EJB のメソッドに対して <method-permission> 要素内で別のロールを定義すると、この Bean のメソッドに対して定義されたすべてのメソッド許可の組合せが付与されます。

<method-permission><method> 要素を使用して、インタフェースまたは実装内の 1 つ以上のメソッドについてセキュリティ・ロールを指定します。この定義は、EJB 仕様に従って、次のいずれかの形式になります。

1. Bean 名を指定し、Bean 内のすべてのメソッドを示す * 文字を使用して、Bean 内のすべてのメソッドを定義します。次に例を示します。

```
<method-permission>
  <role-name>myMgr</role-name>
  <method>
    <ejb-name>EJBNAME</ejb-name>
    <method-name>*</method-name>
  </method>
</method-permission>
```

2. Bean 内で一意に識別できる特定のメソッドを定義します。適切なインタフェース名とメソッド名を使用します。次に例を示します。

```
<method-permission>
  <role-name>myMgr</role-name>
  <method>
    <ejb-name>myBean</ejb-name>
    <method-name>myMethodInMyBean</method-name>
  </method>
</method-permission>
```

注意： オーバーロードされた同じ名前のメソッドが複数ある場合、このスタイルの要素は、オーバーロードされた名前を持つすべてのメソッドを参照します。

3. オーバーロードされた多数のメソッドの中から、特定のシグネチャを持つメソッドを定義します。次に例を示します。

```
<method-permission>
  <role-name>myMgr</role-name>
  <method>
    <ejb-name>myBean</ejb-name>
    <method-name>myMethod</method-name>
    <method-params>
       <method-param>javax.lang.String</method-param>
       <method-param>javax.lang.String</method-param>
    </method-params>
  </method>
</method-permission>
```

パラメータは、完全に修飾された Java タイプのメソッドの入力パラメータです。メソッドに入力引数がない場合、<method-params> 要素内に要素は含まれません。配列を指定するには、配列要素のタイプの後に 1 つ以上の角カッコ（int[ ][ ] など）を指定します。

EJB メソッドに対するセキュリティ・チェックなしの指定

特定のメソッドでセキュリティ・ロールをチェックしないようにするには、そのメソッドをチェックなしとして定義します。次に例を示します。

```
<method-permission>
  <unchecked/>
  <method>
     <ejb-name>EJBNAME</ejb-name>
     <method-name>*</method-name>
  </method>
</method-permission>
```

<role-name> 要素を定義するかわりに、<unchecked/> 要素を定義します。これによって、EJBNAME Bean で任意のメソッドを実行すると、コンテナはセキュリティをチェックしません。チェックなしのメソッドは、常に、他のロール定義をオーバーライドします。

runAs セキュリティ識別情報の指定

EJB のすべてのメソッドが特定の識別情報を使用して実行されるように指定できます。つまり、コンテナは、特定のメソッドを実行する許可について別のロールをチェックせず、かわりに、指定されたセキュリティ識別情報を使用してすべての EJB メソッドを実行します。セキュリティ識別情報として、特定のロールまたはコール元の識別情報を指定できます。

runAs セキュリティ識別情報は、<enterprise-beans> セクションの <security-identity> 要素で指定します。次の XML は、すべての Entity Bean メソッドが POMgr というロールを使用して実行されることを示します。

```
<enterprise-beans>
 <entity>
 ...
  <security-identity>
     <run-as>
        <role-name>POMgr</role-name>
     </run-as>
  </security-identity>
...
 </entity>
</enterprise-beans>
```

また、次の XML の例は、コール元の識別情報を使用して Bean のすべてのメソッドを実行するように指定する方法を示します。

```
<enterprise-beans>
 <entity>
 ...
  <security-identity>
     <use-caller-identity/>
  </security-identity>
...
 </entity>
</enterprise-beans>
```

ユーザーおよびグループへの論理ロールのマッピング

論理ロールまたは実際のユーザーとグループは、EJB デプロイメント・ディスクリプタで使用できます。ただし、論理ロールを使用する場合は、OracleAS JAAS Provider または XML ユーザー・マネージャのいずれかで定義した実際のユーザーとグループに、論理ロールをマッピングする必要があります。

アプリケーションのデプロイメント・ディスクリプタで定義した論理ロールを OracleAS JAAS Provider または XML ユーザー・マネージャのユーザーまたはグループにマッピングするには、OC4J 固有のデプロイメント・ディスクリプタで、`<security-role-mapping>` 要素を使用します。

- この要素の `name` 属性では、マッピングされる論理ロールを定義します。

  注意： 要素が異なる XML 構成ファイルにある場合でも、複数の `<security-role-mapping>` 要素で同じ名前を使用しないでください。

- `group` または `user` 要素では、論理ロールをグループまたはユーザー名にマッピングします。このグループまたはユーザーは、OracleAS JAAS Provider または XML ユーザー・マネージャ構成で定義する必要があります。OracleAS JAAS Provider および XML ユーザー・マネージャの詳細は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』および『Oracle Application Server Containers for J2EE サービス・ガイド』を参照してください。

例 8-1 論理ロールの実際のロールへのマッピング

この例では、論理ロール `POMGR` を、`orion-ejb-jar.xml` ファイル内の `managers` グループにマッピングします。このグループの一部としてログイン可能なユーザーは、すべて `POMGR` ロールを所有しているとみなされます。したがって、`PurchaseOrderBean` のメソッドを実行可能です。

```
<security-role-mapping name="POMGR">
 <group name="managers" />
</security-role-mapping>
```

注意： 論理ロールは、1 つのグループにマッピングすることも、複数のグループにマッピングすることも可能です。

このロールを特定のユーザーにマッピングするには、次のようにします。

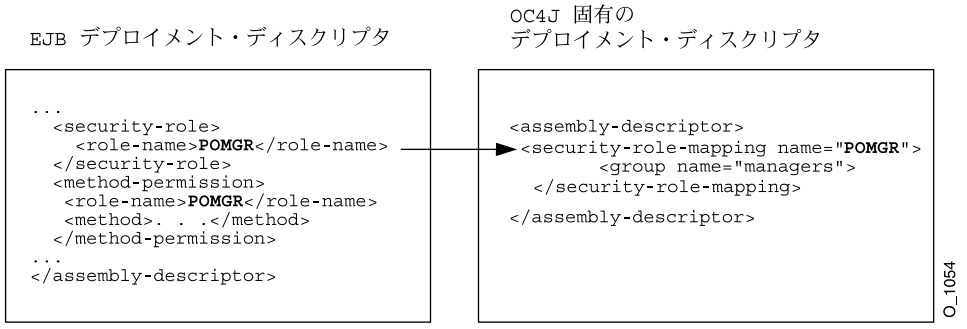
```
<security-role-mapping name="POMGR">
 <user name="guest" />
</security-role-mapping>
```

最後に、次のように、ロールを特定のグループ内の特定のユーザーにマッピングすることも可能です。

```
<security-role-mapping name="POMGR">
 <group name="managers" />
 <user name="guest" />
</security-role-mapping>
```

図 8-3 で示されているように、EJB デプロイメント・ディスクリプタで定義されている POMGR の論理ロール名は、OC4J 固有のデプロイメント・ディスクリプタ内で <security-role-mapping> 要素で managers にマッピングされています。

図 8-3 セキュリティのマッピング



EJB デプロイメント・ディスクリプタ内の <role-name> は、OC4J 固有のデプロイメント・ディスクリプタ内の <security-role-mapping> 要素内の name 属性と同じです。これによりマッピングが識別されます。

未定義メソッドに対するデフォルト・ロール・マッピングの指定

メソッドがロール・マッピングに関連付けられていない場合、そのメソッドは、orion-ejb-jar.xml ファイルの <default-method-access> 要素を介してデフォルト・セキュリティ・ロールにマッピングされます。次に、保護されていないメソッドの自動マッピングを示します。

```
<default-method-access>
   <security-role-mapping name="&lt;default-ejb-caller-role&gt;"
                          impliesAll="true" >
   </security-role-mapping>
</default-method-access>
```

デフォルト・ロールは <default-ejb-caller-role> で、name 属性で定義されます。この文字列は、デフォルト・ロールの名前に置換できます。impliesAll 属性は、メソッドに対するセキュリティ・ロールのチェックが実行されるかどうかを示します。この属性のデフォルトは true で、メソッドに対するセキュリティ・ロールのチェックが実行されないことを示します。この属性を false に設定すると、コンテナは、メソッドに対するデフォルト・ロールをチェックします。

impliesAll 属性が false の場合は、<user> 要素および <group> 要素を使用して、name 属性で定義したデフォルト・ロールを OracleAS JAAS Provider か XML のユーザーまたはグループにマッピングする必要があります。次の例では、メソッド許可に関連付けられていないすべてのメソッドを "others" グループにマッピングする方法を示します。

```
<default-method-access>
   <security-role-mapping name="default-role" impliesAll="false" >
      <group name="others" />
   </security-role-mapping>
</default-method-access>
```

クライアントによるユーザーとグループの指定

クライアントは、ユーザーとグループに保護されたメソッドにアクセスするため、OracleAS JAAS Provider または XML ユーザー・マネージャが認識できる正確なユーザー名またはグループ名とパスワードを提供する必要があります。また、ユーザーまたはグループは、対象となるメソッドのセキュリティ・ロールで指定されている内容と同じであることが必要です。詳細は、8-12 ページの「EJB クライアントの資格証明の指定」を参照してください。

注意： CSiV2 などの基本的な OC4J セキュリティ構成の詳細は、『Oracle Application Server Containers for J2EE セキュリティ・ガイド』を参照してください。

EJB クライアントの資格証明の指定

リモート・コンテナ内の EJB にアクセスする場合、このコンテナに有効な資格証明を渡す必要があります。詳細は、2-16 ページの「JNDI プロパティの設定」を参照してください。

- Pure Java クライアントは、EAR ファイルとともにデプロイされた `jndi.properties` ファイル内で資格証明を定義します。
- コンテナ内で実行されているサーブレットまたは JavaBeans は、リモートの EJB のルックアップ用に作成された `InitialContext` 内で資格証明を渡します。

> **注意：** CSiV2 などの基本的な OC4J セキュリティ構成の詳細は、『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 プロパティの資格証明

`jndi.properties` ファイル内でリモートの EJB をルックアップする際に使用するユーザー名（プリンシパル）およびパスワード（資格証明）を指定します。

たとえば、リモートの EJB に `POMGR/welcome` としてアクセスする場合、次のプロパティを定義します。`factory.initial` プロパティは、Oracle JNDI の実装を使用することを示します。

```
java.naming.security.principal=POMGR
java.naming.security.credentials=welcome
java.naming.factory.initial=
   com.evermind.server.ApplicationClientInitialContextFactory
java.naming.provider.url=opmn:ormi://opmnhost:oc4j_inst1/ejbsamples
```

アプリケーション・プログラム内で、次のように、リモートの EJB を認証し、アクセスします。

```
InitialContext ic = new InitialContext();
CustomerHome = (CustomerHome)ic.lookup("java:comp/env/purchaseOrderBean");
```

InitialContext 内の資格証明

サーブレットまたは JavaBeans からリモートの EJB にアクセスするには、次のようにして、InitialContext オブジェクトで資格証明を渡します。

```
Hashtable env = new Hashtable();
env.put("java.naming.provider.url",
   "opmn:ormi://opmnhost:oc4j_inst1/ejbsamples");
env.put("java.naming.factory.initial",
        "com.evermind.server.ApplicationClientInitialContextFactory");
env.put(Context.SECURITY_PRINCIPAL, "POMGR");
env.put(Context.SECURITY_CREDENTIALS, "welcome");
Context ic = new InitialContext (env);
CustomerHome =
    (CustomerHome)ic.lookup("java:comp/env/purchaseOrderBean")
```

9

高度な EJB のトピック

この章では、以前の章で説明した基本事項の範囲外の事項について説明します。

デプロイメント・ディスクリプタに構成する EJB コンテナ・サービスの一部は、他のマニュアルで説明されています。データ・ソース、JTA、JNDI および RMI/IIOP の詳細は、『Oracle Application Server Containers for J2EE サービス・ガイド』を参照してください。CSiV2 などのセキュリティの詳細は、『Oracle Application Server Containers for J2EE セキュリティ・ガイド』を参照してください。

この章には、次の内容が含まれます。

クラスの共有

EJB 間でクラスを共有する場合は、次のいずれかを実行します。

- 2 つの EJB が同じクラスを使用する場合は、すべてのクラスと EJB を同じ JAR ファイルに含めます。デプロイ後、両方の EJB が同じ共通クラスを使用できるようになります。
- 共有クラスを、アプリケーションのそれぞれの JAR ファイルに配置します。次のように、EJB JAR `manifest.mf` ファイルの `class-path` にある共有 JAR ファイルを参照します。

  ```
  class-path:shared_classes.jar
  ```

  `shared_classes.jar` の場所は、これを参照する JAR ファイルが EAR ファイル内のどこに存在するかによって異なります。この例では、`shared_classes.jar` ファイルは、EJB JAR ファイルと同じレベルに存在します。
- すべてのアプリケーションがこれらのクラスを参照するようにする場合は、共有クラスを JAR ファイルにアーカイブし、この JAR ファイルをデフォルト・アプリケーションの共有ライブラリ・ディレクトリに配置します。デフォルトの共有ライブラリは `home/lib` です。ただし、共有ライブラリ・ディレクトリは、Oracle Enterprise Manager を使用してデフォルト・アプリケーションの「一般プロパティ」ページに設定できます。
- 特定のアプリケーションのみがこれらのクラスを参照するようにする場合は、共有クラスをそれぞれの共有クラス自身のアプリケーションにアーカイブし、アプリケーションの EAR ファイルをデプロイし、共有クラスを参照するアプリケーションで、その共有クラス・アプリケーションを親として宣言します。Oracle Application Server のデフォルトの親は、デフォルト・アプリケーションです。

  子アプリケーションは、親アプリケーションの名前空間を認識します。これは、EJB などのサービスを複数のアプリケーションで共有するために使用されます。親アプリケーションの指定方法は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』を参照してください。

EJB アプリケーションと Web アプリケーション間でクラスを共有する場合は、参照されるクラスを共有 JAR ファイルに配置する必要があります。

`ClassCastException` が発生した場合は、次の状況であることが考えられます。

- 開発を容易にするために、サーブレットが存在している WAR ファイルに EJB インタフェースをコピーしていて、WAR ファイルを作成する前にインタフェースの削除を忘れていた場合で、**さらに**
- `orion-web.xml` ファイルで、`<web-app-class-loader>` 要素の `search_local_classes_first` 属性をオンにしていた場合。

この問題を解決するには、コピーしたクラスを WAR ファイルから削除するか、または search_local_classes_first 属性をオフにします。この属性をオンにすると、クラス・ローダーは、EJB JAR ファイル内のクラスなどの他のクラスをロードする前に、WAR ファイル内のクラスをロードします。この属性の詳細は、『Oracle Application Server Containers for J2EE サーブレット開発者ガイド』の「サーブレットの開発」の章の「OC4J におけるシステム・クラスより前の WAR ファイル・クラスのロード」の項を参照してください。

EJB のライフ・サイクルに関する問題

次の各項では、OC4J における EJB のライフ・サイクルに関する問題について説明します。

- ステートフル Session Bean の非アクティブ化が発生する状況
- Entity Bean のプール・サイズの構成
- CMP Entity Bean の finder メソッドにおける遅延ロードの構成

ステートフル Session Bean の非アクティブ化が発生する状況

非アクティブ化を使用すると、コンテナは、Bean とその状態を 2 次記憶装置にシリアライズしてメモリーから削除することによって、非アクティブなアイドル状態の Bean インスタンスの対話状態を保持できます。非アクティブ化の前に、コンテナは `ejbPassivate()` メソッドを起動し、データベース接続、TCP/IP ソケットまたはオブジェクトのシリアライズ化によって透過的に非アクティブ化されないリソースなど、保持されたリソースを Bean 開発者がクリーン・アップできるようにします。シリアライズおよび非アクティブ化可能なオブジェクトの種類は、この項の最後に示します。

注意： OC4J では、ステートフル Session Bean のみが非アクティブ化されます。ステートレス Session Bean には非アクティブ化する状態がないため、Entity Bean はその状態をデータベース内に保持する必要があります。

非アクティブ化された Bean インスタンスのメソッドの 1 つをクライアントが起動すると、Bean を 2 次記憶装置からデシリアライズしてメモリーに戻すことによって、保持されていた対話状態のデータがアクティブ化されます。アクティブ化の前に、コンテナは `ejbActivate()` メソッドを起動し、`ejbPassivate()` 時に解放したリソースを Bean 開発者がリストアできるようにします。非アクティブ化の詳細は、EJB の仕様を参照してください。

非アクティブ化はデフォルトで有効化されています。ステートフル Session Bean の非アクティブ化をオフにするには、`server.xml` ファイルの `<sfsb-config>` 要素を false に設定します。ステートフル Session Bean では、9-5 ページの「非アクティブ化可能なオブジェクトの種類」に示されている特定の種類のオブジェクトのみが非アクティブ化されます。ユーザーがすべてのリソースを解放し、使用可能な種類のオブジェクト内でのみ状態が存在するようにして、ステートフル Session Bean を非アクティブ化する準備をしていない場合、非ア

クティブ化は常に失敗します。オブジェクトの種類を変更する必要がなく、オブジェクトを非アクティブ化する予定がない場合は、非アクティブ化を無効にできます。この他に、パフォーマンスの理由から非アクティブ化を無効にする場合もあります。非アクティブ化の処理にはオーバーヘッドがかかるため、処理速度を優先する必要があり、リソースに関して実質的な問題がない場合は、非アクティブ化をオフにできます。

非アクティブ化をオフにする方法の例は、次のとおりです。

```
<sfsb-config enable-passivation="false"/>
```

注意： 詳細は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』の付録の「server.xml ファイルの要素」の項で定義されている <sfsb-config> 要素を参照してください。

非アクティブ化は、次の基準の組合せに基づいて起動されます。

- アイドル・タイムアウトが経過した場合

  各 Bean に対してアイドル・タイムアウトを秒単位で設定できます。このタイムアウトが経過すると、非アクティブ化が発生します。<session-deployment> の idletime 属性を適切な秒数に設定します。デフォルトは 300 秒（5 分）です。この属性を無効にするには、"never" を指定します。

- リソース不足の場合

  <session-deployment> 内の次の各属性によって、リソースのしきい値、しきい値をチェックする時期、およびしきい値に達した場合に非アクティブ化する Bean の数が定義されます。

  * memory-threshold - 非アクティブ化が発生するまでに使用可能な JVM メモリーの量に対するしきい値を定義します。パーセンテージとして解釈される整数を指定します。この値に達すると、アイドル・タイムアウトが経過していない場合でも Bean は非アクティブ化されます。デフォルトは 80% です。この属性を無効にするには、"never" を指定します。

  * max-instances-threshold - max-instances 属性の定義に応じて、存在するアクティブ Bean 数に対するしきい値を定義します。パーセンテージとして解釈される整数を指定します。たとえば、max-instances を 100、max-instances-threshold を 90% に定義した場合は、アクティブ Bean インスタンスの数が 90 を超えると、Bean の非アクティブ化が発生します。デフォルトは 90% です。この属性を無効にするには、"never" を指定します。

  * resource-check-interval - コンテナは、すべてのリソースをこの時間間隔でチェックします。この時点でいずれかのしきい値に達している場合は、非アクティブ化が発生します。デフォルトは 180 秒（3 分）です。この属性を無効にするには、"never" を指定します。

    * passivate-count - いずれかのリソースしきい値に達した場合に非アクティブ化される Bean の数を定義する整数です。Bean の非アクティブ化は、最低使用頻度アルゴリズムを使用して実行されます。デフォルトは、max-instances 属性の 1/3 です。この属性を無効にするには、カウントを 0（ゼロ）または負の数に設定します。

- Bean インスタンスの最大数に達した場合

  この数は、<session-deployment> の max-instances 属性内で設定されます。max-instances 属性では、メモリー内に存在できる Bean インスタンスの数が制御されます。この値に達すると、コンテナは最も古い Bean インスタンスをメモリーから非アクティブ化しようとします。非アクティブ化に失敗した場合、コンテナは、call-timeout 属性に設定されたミリ秒数待機して、別の非アクティブ化、Bean の remove() メソッドのコールまたは Bean の期限切れのいずれかによって、メモリーから削除された Bean インスタンスがあるかどうかを確認し、その後で TimeoutExpiredException をクライアントにスローします。Bean インスタンスの数を無限に許可する場合は、max-instances の値を 0（ゼロ）のままにします。デフォルトは 0（ゼロ）で、無限を意味します。

- OC4J インスタンスが終了した場合

  コンテナのメモリー内の非アクティブ化されていないすべての Bean インスタンスが、2 次記憶装置にシリアライズされます。OC4J の起動時に、非アクティブ化されたこれらの Bean はメモリーにリストアされます。

非アクティブ化時のシリアライズ化に失敗した場合、コンテナは Bean をメモリーにリカバリして、処理前の状態にしようとします。非アクティブ化に失敗した Bean については、その後非アクティブ化は試行されません。また、アクティブ化に失敗した場合、Bean とその参照はコンテナから完全に削除されます。

クラスタ内の非アクティブ化された Bean に対して新規 Bean データが伝播されると、その Bean インスタンスのデータは、伝播されたデータによって上書きされます。

非アクティブ化可能なオブジェクトの種類

（非アクティブ化時の）2 次記憶装置へのシリアライズ化が成功するために、Bean の対話状態は、プリミティブ値と次の特別なタイプのみで構成されている必要があります。

- シリアライズ可能なオブジェクト
- NULL
- コンポーネント・インタフェース（EJBObject または EJBLocalObject）の参照
- ホーム・インタフェース（EJBHome または EJBLocalHome）の参照
- SessionContext オブジェクトの参照
- 環境ネーミング・コンテキストの参照

- UserTransaction インタフェースの参照
- リソース・マネージャのコネクション・ファクトリの参照

Bean 開発者は、ejbPassivate() メソッド内のすべてのフィールドがこれらのタイプであることを確認する必要があります。一時的なフィールドやシリアライズ不可のフィールドは、このメソッドでは NULL に設定する必要があります。

非アクティブ化された EJB の格納

OC4J によって非アクティブ化されたステートフル Session Bean は、OC4J デプロイメント・ディスクリプタの <session-deployment> 要素の persistence-filename 属性が指定するディレクトリとファイル名で格納されます。非アクティブ化によってこのディレクトリ内の領域が使用され、非アクティブ化された Bean が格納されます。デフォルトは application-deployments/persistence ディレクトリです。非アクティブ化によって大量のディスク領域が割り当てられる場合は、使用可能な領域があるシステム上の別の場所にディレクトリを変更するか、または非アクティブ化をオフにしてください。

Entity Bean のプール・サイズの構成

Bean インスタンス・プールの最大数と最小数を設定できます。このプールには、状態が割り当てられていない EJB 実装インスタンスが含まれます。プール状態の間、Bean インスタンスは固有の状態を持たず、ラッパー・インスタンスに割り当てることができます。

プールされる数は、<entity-deployment> 要素の次の属性を使用して設定できます。

- max-instances 属性は、プール内に含めることができる Entity Bean インスタンスの最大数を設定します。Entity Bean は、ラッパー・インスタンスに関連付けられていない場合、プールされた状態に設定されます。したがって、それは固有の状態を持ちません。

  デフォルトは 0（ゼロ）で、無限を意味します。Bean 実装の最大インスタンス数を 20 に設定する場合は、次のように設定します。

  ```
  <entity-deployment ...  max-instances="20"
   ...
  </entity-deployment>
  ```

- min-instances 属性は、プール内に含めることができるインスタンスの最小数を設定します。次のように設定します。

  ```
  <entity-deployment ... min-instances="2"
   ...
  </entity-deployment>
  ```

CMP Entity Bean の finder メソッドにおける遅延ロードの構成

各 finder メソッドでは、1 つ以上のオブジェクトが取得されます。デフォルト（遅延ロードの設定は「NO」）を使用する場合は、finder メソッドによって、単一の SQL select 文がデータベースに対して実行されます。CMP Bean の場合、1 つ以上のオブジェクトがそのすべての CMP フィールドとともに取得されます。このため、たとえば、`findAllEmployees` メソッドを実行した場合は、この finder によって、すべての従業員オブジェクトが各従業員オブジェクトのすべての CMP フィールドとともに取得されます。

遅延ロードをオンにすると、finder 内で取得されたオブジェクトの主キーのみが戻されます。その後、実装内でオブジェクトにアクセスしたときのみ、OC4J コンテナによって、実際のオブジェクトが主キーに基づいてアップロードされます。`findAllEmployees` finder メソッドの例では、すべての従業員の主キーが `Collection` に戻されます。`Collection` 内のいずれかの従業員に初めてアクセスすると、OC4J では、主キーを使用してデータベースから単一の従業員オブジェクトを取得します。取得するオブジェクト数が大量で、ローカル・キャッシュにすべてロードするとパフォーマンスが低下する恐れがある場合は、遅延ロード機能をオンにすることができます。

遅延ロードを使用する際にパフォーマンスを考慮する必要があります。複数のオブジェクトを取得しても使用するのはその中の一部である場合は、遅延ロードをオンにすることをお薦めします。また、`getPrimaryKey` メソッドを通じてのみオブジェクトを使用する場合も、遅延ロードをオンにすることをお薦めします。

`findByPrimaryKey` メソッドで遅延ロードをオンにするには、次のように `findByPrimaryKey-lazy-loading` 属性を true に設定します。

```
<entity-deployment ... findByPrimaryKey-lazy-loading="true" ... >
```

カスタムの finder メソッドで遅延ロードをオンにするには、次のように、そのカスタムの finder に対する `<finder-method>` 要素の `lazy-loading` 属性を true に設定します。

```
<finder-method ... lazy-loading="true" ...>
 ...
</finder-method>
```

永続性を更新する手法

デフォルトでは、コンテナは、Bean 内の変更されたフィールドのみ維持します。各コールの終了時に、変更されたフィールドを更新するための SQL コマンドが作成されます。ただし、永続的なすべてのフィールドを更新する場合は、次の属性を false に設定します。

```
<entity-deployment ...  update-changed-fields-only="false"
 ...
</entity-deployment>
```

Entity Bean 同時実行性モードおよびデータベース分離モード

同時実行が可能な間に、リソースが競合したり、互いの変更がデータベース表で上書きされるのを防ぐため、Entity Bean 同時実行性モードおよびデータベース分離モードが用意されています。

- データベース分離モード
- Entity Bean 同時実行性モード

データベース分離モード

`java.sql.Connection` オブジェクトは、特定データベースへの接続を表します。データベース分離モードは、リソースの競合からの保護を定義するために提供されています。複数のユーザーが同一リソースを更新しようとすると、更新内容が失われる可能性があります。つまり、あるユーザーが、他のユーザーのデータを気づかずに上書きする場合があります。`java.sql.Connection` 標準では 4 種類の分離モードが提供され、オラクル社ではその中の 2 種類のみサポートしています。次の 2 種類のモードです。

- `TRANSACTION_READ_COMMITTED`: 内容を保証しない読取りを防ぎ、非リピータブル・リードおよび仮読取りが行われます。このレベルでは、コミットされていない変更を含む行をトランザクションで読み取らないようにするのみです。
- `TRANSACTION_SERIALIZABLE`: 内容を保証しない読取り、非リピータブル・リードおよび仮読取りを防ぎます。このレベルには、`TRANSACTION_REPEATABLE_READ` の禁止事項が含まれます。さらに、あるトランザクションで WHERE 条件を満たすすべての行を読み取り、次のトランザクションでその WHERE 条件を満たす行を挿入した後、最初のトランザクションで同じ条件で再読取りをし、2 番目の読取りで追加の仮の行を取得するのを防ぎます。

注意： エミュレートされていないデータ・ソースを使用している場合は、分離レベルを serializable に設定することはできません。分離レベルを serializable に設定すると、エミュレートされていないデータ・ソースは動作しなくなります。

特定の Bean について、いずれかのデータベース分離モードを構成できます。つまり、Bean でトランザクションを起動すると、その Bean のデータベース分離モードが、OC4J 固有のデプロイメント・ディスクリプタに指定したモードとなるように指定できます。分離モードは、その Bean にとって、パラレル実行が重要かデータの一貫性が重要かによって指定します。Bean の分離モードは、トランザクション全体に対して設定されます。

分離モードは、Entity Bean ごとに <entity-deployment> 要素の isolation 属性で設定できます。値は、committed または serializable です。デフォルトは committed です。serializable に変更するには、対象の Bean の orion-ejb-jar.xml を次のように構成します。

```
<entity-deployment ...  isolation="serializable"
 ...
</entity-deployment>
```

パフォーマンスとデータの一貫性の間には常にトレードオフがあります。serializable 分離モードによりデータの一貫性が提供され、committed 分離モードによりパラレル実行が提供されます。

注意： OC4J 固有のデプロイメント・ディスクリプタの max-tx-retries 要素が 0（ゼロ）より大きい場合は、serializable モードで更新内容が失われる危険があります。この値のデフォルトは 0（ゼロ）です。この要素を 0（ゼロ）より大きい値に設定した場合は、2 番目にブロックされたクライアントが ORA-8177 例外を受け取ると、コンテナは更新を再試行します。この再試行によって、ロックされていない行が検出されて更新が行われます。この結果、2 番目のクライアントの更新は正常終了し、最初のクライアントの更新内容が上書きされます。serializable モードを使用する場合は、max-tx-retries 要素を 0（ゼロ）のままにして、データの一貫性を維持することを考慮してください。

分離モードを設定しない場合は、データベースに構成されているモードが使用されます。OC4J 固有のデプロイメント・ディスクリプタ内に設定した分離モードは、その Bean のグローバル・トランザクションの存続期間中、データベースに構成されている分離モードを一時的にオーバーライドします。つまり、serializable モードを使用する Bean を定義した場合、OC4J コンテナは、トランザクション終了までの間のみ、この Bean についてデータベースを強制的に serializable モードにします。

Entity Bean 同時実行性モード

OC4J では、コンテナ管理による永続的な（CMP）Entity Bean 内でのリソースの競合およびパラレル実行を処理するため、同時実行性モードも提供されます。Bean 管理による永続的な Entity Bean は、Bean 実装自体内でリソースのロックを管理します。モードによって、リソースの競合を管理するためのブロック時期、またはパラレルで実行する時期を構成します。

次の同時実行性モードがあります。

- PESSIMISTIC: リソースの競合を管理し、パラレル実行はできません。Entity Bean を実行できるのは、一度に 1 ユーザーのみです。
- OPTIMISTIC: 複数のユーザーがパラレルで Entity Bean を実行できます。リソースの競合は監視しないため、データの一貫性を維持するには、データベース分離モードを使用する必要があります。
- READ-ONLY: 複数のユーザーがパラレルで Entity Bean を実行できます。コンテナでは、Bean の状態を更新できません。

CMP Entity Bean の同時実行性モードを使用可能にするには、適切な同時実行性モードの値（"pessimistic"、"optimistic" または "read-only"）を、OC4J 固有のデプロイメント・ディスクリプタ（orion-ejb-jar.xml）の <entity-deployment> 要素の locking-mode 属性に追加します。デフォルトは "optimistic" です。同時実行性モードを pessimistic に変更するには、次のようにします。

```
<entity-deployment ...  locking-mode="pessimistic"
 ...
</entity-deployment>
```

これらの同時実行性モードは Bean ごとに定義され、ロックはトランザクション境界に適用されます。

パラレル実行では、ラッパーのプール・サイズおよび Bean インスタンスを正しく設定する必要があります。プール・サイズの構成方法については、9-12 ページの「環境参照の構成」を参照してください。

データベースへの排他的書込みアクセス

<entity-deployment> 要素の exclusive-write-access 属性は、その Bean が、データベース内のその表にアクセス可能な唯一の Bean であることを示し、リソースの更新に外部からの方法が使用されないことを示します。この Bean に対して維持されているキャッシュは、この Bean によってのみ操作されることを、OC4J インスタンスに伝えます。つまり、この属性を true に設定すると、この Bean 内で使用されている表は、この Bean のみによって更新可能であることを、コンテナに伝えます。したがって、この Bean に対して維持されているキャッシュは、バックエンド・データベースから常に更新する必要がありません。

このフラグは、表の更新を妨げません。つまり、表をロックするわけではありません。ただし、別の Bean によって、または手動で表を更新しても、結果は自動的にこの Bean 内には更新されません。

この属性のデフォルトは false です。Entity Bean 同時実行性モードの影響を受けるため、read-only の Entity Bean に対してのみ、この要素を true に設定できます。OC4J は、pessimistic 同時実行性モードおよび optimistic 同時実行性モードの場合、常にこの属性を false にリセットします。

```
<entity-deployment ...  exclusive-write-access="true"
 ...
</entity-deployment>
```

データベース分離モードと Bean 同時実行性モードの組合せによる影響

pessimistic 同時実行性モードおよび read-only 同時実行性モードの場合は、データベース分離モードの設定に問題はありません。分離モードで問題があるのは、外部ソースによってデータベースが変更される場合のみです。

optimistic を committed と組み合せて選択すると、更新内容が失われる可能性があります。optimistic を serializable と組み合せて選択すると、更新内容が失われることはありません。したがって、データは常に一貫しています。ただし、リソースの競合エラーとして ORA-8177 例外を受け取ります。

Pessimistic と Optimistic/Serializable の違い

同時実行性モードが pessimistic の Entity Bean では、（同じ主キーの同じインスタンス上または異なるインスタンス上のいずれでも）複数のクライアントが Bean を実行することはできません。インスタンスを実行できるのは、一度に 1 つのクライアントのみです。

同時実行性モードが optimistic の Entity Bean では、Bean 実装の複数のインスタンスをパラレルで実行できます。ただし、2 つの別々のトランザクションで同じ行が同時に更新される場合があるため、潜在的に更新（競合）内容が失われる可能性があります。

トランザクション分離モードを serializable に設定すると、競合が発生した場合に検出が可能です。競合した時点で、一方のトランザクションからの更新では SQLException が発生し、そのトランザクションがロールバックされます。

オプションで、トランザクションが再試行されるように、<entity-deployment> 要素の tx-retries 属性の値を複数回に設定できます。

同時実行性モードのクラスタリングへの影響

すべての同時実行性モードは、スタンドアロン環境でもクラスタリング環境でも、同様に動作します。これは、同時実行性モードがデータベース・レベルでロックされるためです。したがって、pessimistic の Bean インスタンスがノード間でクラスタリングされていても、1 つのインスタンスが実行されるとすぐに、データベースは他のすべてのインスタンスをロックします。

環境参照の構成

実行時に Bean にアクセス可能な 3 種類の環境要素を作成できます。環境変数、EJB 参照およびリソース・マネージャです。これらの環境要素は静的で、Bean によって変更できません。

一般のソフトウェア・ベンダーは、通常、EJB コンテナから独立した EJB を開発します。Bean 実装をコンテナの仕様から分離するために、事前定義変数、Entity Bean またはリソース・マネージャのいずれかにマッピングされている環境要素を作成できます。このように間接的に設定することにより、Bean 開発者は、実際の名前を指定せずに、既存の変数、EJB および JDBC の `DataSource` を参照できます。これらの名前は、デプロイメント・ディスクリプタで定義され、OC4J 固有のデプロイメント・ディスクリプタ内で実際の名前にリンクされています。

環境変数

`InitialContext` のルックアップを通じて Bean がアクセスする環境変数を作成できます。これらの変数は、`<env-entry>` 要素内で定義され、`String`、`Integer`、`Boolean`、`Double`、`Byte`、`Short`、`Long` および `Float` のいずれかの型が使用可能です。. 環境変数の名前は `<env-entry-name>` で定義し、型は `<env-entry-type>` で定義し、初期値は `<env-entry-value>` で定義します。`<env-entry-name>` は "`java:comp/env`" コンテキストに対して相対的です。

たとえば、`java:comp/env/minBalance` および `java:comp/env/maxCreditBalance` の XML デプロイメント・ディスクリプタ内で、次の 2 つの環境変数が宣言されています。

```
<env-entry>
     <env-entry-name>minBalance</env-entry-name>
     <env-entry-type>java.lang.Integer</env-entry-type>
     <env-entry-value>500</env-entry-value>
</env-entry>
<env-entry>
     <env-entry-name>maxCreditBalance</env-entry-name>
     <env-entry-type>java.lang.Integer</env-entry-type>
     <env-entry-value>10000</env-entry-value>
</env-entry>
```

Bean のコード内で、次のように、InitialContext を通じてこれらの環境変数にアクセスします。

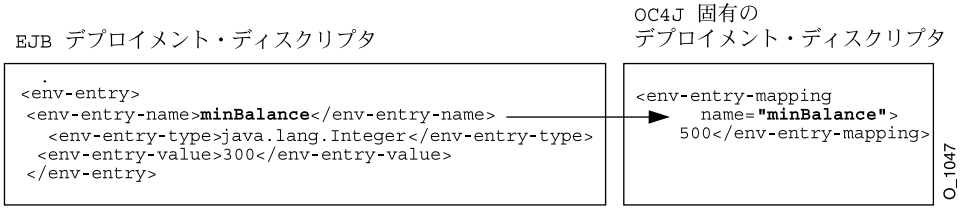
```
InitialContext ic = new InitialContext();
Integer min = (Integer) ic.lookup("java:comp/env/minBalance");
Integer max = (Integer) ic.lookup("java:comp/env/maxCreditBalance"));
```

環境変数の値を取得するには、各環境変数の先頭に "java:comp/env/" を付加する必要があります。これは、コンテナが環境変数を格納する場所です。

環境変数の値を OC4J 固有のデプロイメント・ディスクリプタ内で定義する場合、OC4J 固有のデプロイメント・ディスクリプタ内で、<env-entry-name> を <env-entry-mapping> 要素にマッピングします。これにより、orion-ejb-jar.xml ファイルで指定された値は、ejb-jar.xml ファイルで指定された値をオーバーライドします。EJB デプロイメント・ディスクリプタで指定された型は変わりません。

図 9-1 に、OC4J 固有のデプロイメント・ディスクリプタ内で、minBalance 環境変数が 500 に定義されていることを示します。

図 9-1 環境変数のマッピング



他の Enterprise JavaBeans の環境参照

EJB の環境参照は、デプロイメント・ディスクリプタ内でローカルまたはリモートのインタフェースを使用して定義できます。Bean が別の Bean をコールする場合は、デプロイメント・ディスクリプタ内で定義された参照を使用して、Bean が 2 番目の Bean を起動するようにできます。EJB デプロイメント・ディスクリプタ内で、論理名を作成します。この名前は、OC4J 固有のデプロイメント・ディスクリプタ内で、Bean の実際の名前にマッピングされます。

ターゲット Bean を環境参照として宣言すると、間接性が実現されます。つまり、起点となる Bean は、論理名を使用してターゲット Bean を参照できます。

Bean のローカル・インタフェースの参照は、<ejb-local-ref> 要素で定義されます。Bean のリモート・インタフェースの参照は、<ejb-ref> 要素で定義されます。

別の EJB に対する参照を、JAR 内、または親として宣言された Bean 内で定義する場合は、次の情報を指定します。

1. 名前 : ターゲット Bean の名前を指定します。この名前は、Bean が JNDI ロケーション内で、ターゲット Bean へのアクセスに使用する名前です。名前の先頭には "ejb/myEmployee" などのように、"ejb/" を使用します。これは "java:comp/env/ejb" コンテキスト内で使用可能になります。
   - この名前には、Bean の実際の名前、つまり <session> または <entity> 要素内で定義された <ejb-name> 要素を使用可能です。
   - この名前には、実装内で使用する論理名も使用可能です。これは、Bean の実際の名前ではありません。論理名を使用する場合は、実際の名前を、OC4J 固有のデプロイメント・ディスクリプタ内の <ejb-link> 要素または <ejb-ref-mapping> 要素で指定する必要があります。
2. 型 : Bean が Session Bean または Entity Bean のいずれであるかを定義します。値は、"Session" または "Entity" のいずれかです。
3. ホーム : ホーム・インタフェースの絶対名を指定します。
4. リモート : リモート・インタフェースの絶対名を指定します。
5. リンク : ターゲット Bean の EJB 名を指定します。これはオプションで、name 属性で論理名を使用した場合にのみ使用されます。

ローカル・インタフェースの参照の例

JAR 内に、BeanA と BeanB の 2 つの Bean が存在するとします。BeanB で BeanA のローカル・インタフェースの参照を作成した場合、この参照は、次の 3 つのうちのいずれかの方法で定義可能です。

- Bean の実際の名前を指定する。BeanB は、定義内で、次の <ejb-local-ref> を定義します。

```
<ejb-local-ref>
 <ejb-ref-name>myBeans/BeanA</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <local-home>myBeans.BeanALocalHome</local-home>
 <local>myBeans.BeanALocal</local>
</ejb-local-ref>
```

  ターゲットの EJB 名は <ejb-ref-name> 要素で指定されているため、この方法の場合、<ejb-link> は必要ありません。ただし、BeanB の実装では、JNDI 取得時に BeanA を参照する必要があります。これは、EJB または Java クライアント内では java:comp/env/myBeans/BeanA を使用して取得し、サーブレット内では "myBeans/BeanA" を使用して取得します。

  注意 : サーブレットは、JNDI ルックアップにおいて "java:comp/env" 接頭辞を必要としません。したがって、常に、実際の JNDI 名または EJB の論理名のみを参照します。

- <ejb-link> 要素で Bean の EJB 名を指定する。<ejb-ref-name> 要素で論理名を定義することによって、Bean の実装で JNDI 取得時に任意の論理名を使用し、<ejb-link> 要素でターゲット EJB 名を指定することによって、その論理名をターゲット Bean にマッピングできます。次の例は、この Bean が JNDI 取得時にそのコードで使用できる論理名 ejb/nextVal を定義します。コンテナはその論理名を <ejb-link> 要素で指定されているターゲット Bean myBeans/BeanA にマッピングします。

```
<ejb-local-ref>
 <ejb-ref-name>ejb/nextVal</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <local-home>myBeans.BeanALocalHome</local-home>
 <local>myBeans.BeanALocal</local>
 <ejb-link>myBeans/BeanA</ejb-link>
</ejb-local-ref>
```

  BeanB は、BeanA の JNDI 取得時に java:comp/env/ejb/nextVal を使用します。

- OC4J 固有のデプロイメント・ディスクリプタ内で、Bean の論理名を <ejb-ref-name> で、Bean の実際の名前を <ejb-ref-mapping> 要素で指定する。

  EJB デプロイメント・ディスクリプタの参照は、次のようになります。

```
<ejb-local-ref>
 <ejb-ref-name>ejb/nextVal</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <local-home>myBeans.BeanALocalHome</local-home>
 <local>myBeans.BeanALocal</local>
</ejb-local-ref>
```

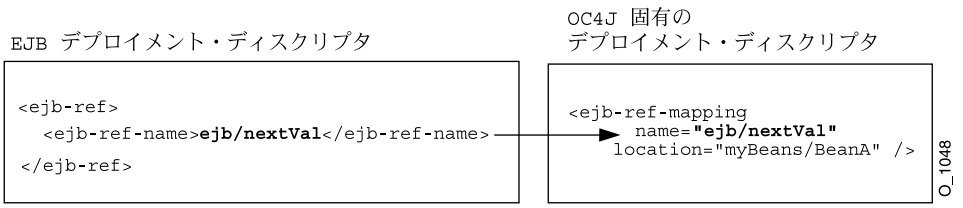
  "ejb/nextVal" 論理名は、次のように、OC4J 固有のデプロイメント・ディスクリプタ内で実際の名前にマッピングされます。

```
<ejb-ref-mapping name="ejb/nextVal" location="myBeans/BeanA"/>
```

  BeanB は、BeanA の JNDI 取得時に java:comp/env/ejb/nextVal を使用します。

図 9-2 で示すように、Bean の論理名を JNDI 名にマッピングするために、"ejb/nextVal" という同じ名前が、EJB デプロイメント・ディスクリプタ内の <ejb-ref-name>、および OC4J 固有のデプロイメント・ディスクリプタ内の <ejb-ref-mapping> 要素内の name 属性の両方で指定されています。

図 9-2 EJB 参照のマッピング



環境参照を使用した EJB へのアクセス

参照を使用して、実装内から Bean にアクセスするには、JNDI ルックアップで EJB デプロイメント・ディスクリプタに定義されている <ejb-ref-name> を使用します。

InitialContext の取得時に、デフォルト・コンテキストを使用している場合は、次のいずれかを実行します。

- <ejb-ref-name> 内に定義されている論理名の先頭に "java:comp/env/ejb/" を付加します。これは、コンテナがデプロイメント・ディスクリプタで定義された EJB 参照を格納する場所です。
- 論理名の先頭に文字列を付加せずに、<ejb-ref-name> で定義された論理名のみを指定します。

次の例は、java:comp/env 接頭辞を使用した、EJB クライアントによるルックアップです。論理名は "ejb/HelloWorld" と仮定しています。

```
InitialContext ic = new InitialContext();
HelloHome hh = (HelloHome)ic.lookup("java:comp/env/ejb/HelloWorld");
```

次の例は、論理名 "ejb/HelloWorld" のみを使用したルックアップです。

```
InitialContext ic = new InitialContext();
HelloHome hh = (HelloHome)ic.lookup("ejb/HelloWorld");
```

ただし、デフォルト・コンテキストを使用していない場合でも、RMIInitialContext オブジェクトなどの別のコンテキストを特別に使用しているときは、次のように論理名のみを使用できます。

```
InitialContext ic = new InitialContext();
HelloHome hh = (HelloHome)ic.lookup("ejb/HelloWorld");
```

例 9-1 環境内でのローカル EJB 参照の定義

次の例では、Hello Bean のローカル・インタフェースの参照を定義します。

1. 起点 Bean 内でターゲット Bean に使用されている論理名は "java:comp/env/ejb/HelloWorld" です。
2. ターゲット Bean は Session Bean です。
3. ローカル・ホーム・インタフェースは hello.HelloLocalHome で、ローカル・インタフェースは hello.HelloLocal です。
4. <ejb-ref-name> 属性は、起点の Bean 内で使用する論理名です。これはオプションです。この例では、Bean は、"ejb/HelloWorld" 名の下の EJB デプロイメント・ディスクリプタで定義されます。

EJB デプロイメント・ディスクリプタ

```
<ejb-local-ref>
  <description>Hello World Bean</description>
  <ejb-ref-name>ejb/HelloWorld</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local-home>hello.HelloLocalHome</local-home>
  <local>hello.HelloLocal</local>
</ejb-local-ref>
```

同じ論理名を両方の要素で指定することによって、EJB デプロイメント・ディスクリプタ内の <ejb-ref-name> 要素が、OC4J 固有のデプロイメント・ディスクリプタの <ejb-ref-mapping> 内の name 属性にマッピングされています。Oracle 固有のデプロイメント・ディスクリプタの場合、Bean の論理名 "java:comp/env/ejb/HelloWorld" を JNDI ロケーション "/test/myHello" にマッピングするには、次の定義を使用します。

OC4J 固有のデプロイメント・ディスクリプタ

```
<ejb-ref-mapping
  name="ejb/HelloWorld"
  location="/test/myHello"/>
```

この Bean を実装内で起動するには、EJB デプロイメント・ディスクリプタで定義された <ejb-ref-name> を使用します。EJB または Pure Java クライアントの場合、この名前の先頭に "java:comp/env/ejb/" を付加します。これは、コンテナがデプロイメント・ディスクリプタで定義された EJB 参照を入れる場所です。サーブレットの場合、<ejb-ref-name> で定義された論理名のみ必要です。

次に、クライアントとして機能する EJB によるルックアップを示します。

```
InitialContext ic = new InitialContext();
HelloHome hh = (HelloHome)ic.lookup("java:comp/env/ejb/HelloWorld");
```

または、次のように名前をルックアップできます。

```
InitialContext ic = new InitialContext();
HelloHome hh = (HelloHome)ic.lookup("ejb/HelloWorld");
```

リモート・インタフェースの参照の例

リモート・インタフェースの参照の定義では、9-14 ページの「ローカル・インタフェースの参照の例」で説明したローカル・インタフェースと完全に同じルールが使用されます。定義方法で異なるのは次の点のみです。

- `<ejb-local-ref>` 要素のかわりに、`<ejb-ref>` を使用します。
- `<local-home>` および `<local>` 要素のかわりに、`<home>` および `<remote>` 要素を使用します。

その他はすべて同じです。

次に、JAR で `BeanA` と `BeanB` の 2 つの Bean を使用する例を示します。`BeanB` で `BeanA` への参照を作成した場合、この参照は、次の 3 つのうちのいずれかの方法で定義可能です。

- Bean の実際の名前を指定する。

```
<ejb-ref>
 <ejb-ref-name>myBeans/BeanA</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <home>myBeans.BeanAHome</home>
 <remote>myBeans.BeanA</remote>
</ejb-ref>
```

- `<ejb-link>` 要素で Bean の EJB 名を指定する。

```
<ejb-ref>
 <ejb-ref-name>ejb/nextVal</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <home>myBeans.BeanAHome</home>
 <remote>myBeans.BeanA</remote>
 <ejb-link>myBeans/BeanA</ejb-link>
</ejb-ref>
```

- OC4J 固有のデプロイメント・ディスクリプタ内で、Bean の論理名を <ejb-ref-name> で、Bean の実際の名前を <ejb-ref-mapping> 要素で指定する。

```
<ejb-ref>
 <ejb-ref-name>ejb/nextVal</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <home>myBeans.BeanAHome</home>
 <remote>myBeans.BeanA</remote>
</ejb-ref>
```

"ejb/nextVal" 論理名は、次のように、OC4J 固有のデプロイメント・ディスクリプタ内で実際の名前にマッピングされます。

```
<ejb-ref-mapping name="ejb/nextVal" location="myBeans/BeanA"/>
```

詳細とコード例については、9-14 ページの「ローカル・インタフェースの参照の例」を参照してください。

リソース・マネージャのコネクション・ファクトリ参照への環境参照

リソース・マネージャのコネクション・ファクトリ参照には、JMS、Java mail、URL および JDBC の DataSource オブジェクトなどのリソース・マネージャを含めることが可能です。EJB 参照と同様、各オブジェクト参照に対する環境要素を作成することにより、これらのオブジェクトに JNDI からアクセス可能です。ただし、これらの参照は、これらの参照を定義する Bean 内のオブジェクトの取得にのみ使用可能です。次の項で、それぞれについて詳細に説明します。

- JDBC の DataSource
- mail セッション
- URL

JDBC の DataSource

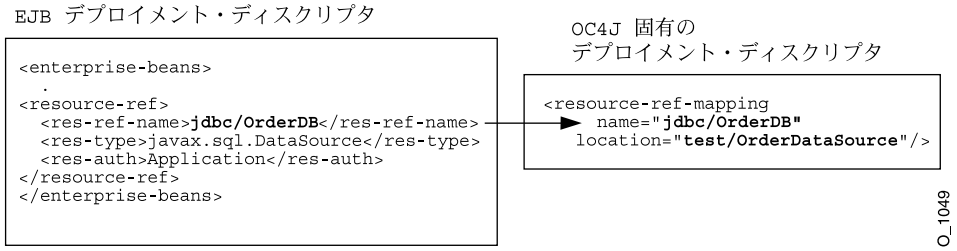
JDBC を通じてデータベースにアクセスする場合、従来の方法を使用するか、または JDBC の DataSource 用の環境要素を作成する方法があります。JDBC の DataSource の環境要素を作成するには、次のようにします。

1. data-sources.xml ファイルで、DataSource を定義します。
2. EJB デプロイメント・ディスクリプタ内の <res-ref-name> 要素で、論理名を作成します。この名前には、必ず先頭に "jdbc" を使用します。Bean コードでは、この参照のルックアップは、常に先頭に "java:comp/env/jdbc" が付きます。

3. EJB デプロイメント・ディスクリプタ内の論理名を、ステップ 1 で作成した OC4J 固有のデプロイメント・ディスクリプタ内の JNDI 名にマッピングします。
4. Bean 内で、"java:comp/env/jdbc" 接頭辞および EJB デプロイメント・ディスクリプタで定義された論理名でオブジェクト参照をルックアップします。

図 9-3 で示すように、JDBC の DataSource は、"test/OrderDataSource" という JNDI 名を使用します。Bean がこのリソースを認識する論理名は、"jdbc/OrderDB" です。これらの名前は、OC4J 固有のデプロイメント・ディスクリプタ内でマッピングされています。したがって、Bean は、実装内で、"java:comp/env/jdbc/OrderDB" 環境要素を使用して、OrderDataSource への接続を取得できます。

図 9-3 JDBC リソース・マネージャのマッピング



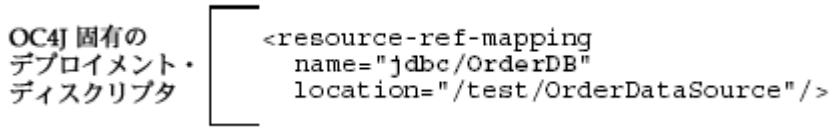
例 9-2 JDBC コネクションの環境要素の定義

環境要素は、EJB デプロイメント・ディスクリプタ内で、論理名、"jdbc/OrderDB"、javax.sql.DataSource のタイプ、および "Application" の認証機能を指定することにより、定義されます。

EJB デプロイメント・ディスクリプタ

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

"jdbc/OrderDB" の環境要素は、Oracle 固有のデプロイメント・ディスクリプタ内で、"test/OrderDataSource" という接続にバインドされている JNDI 名にマッピングされています。



デプロイされると、Bean は次のようにして JDBC の DataSource を取得できます。

```
javax.sql.DataSource db;
java.sql.Connection conn;
.
.
.
db = (javax.sql.DataSource) initCtx.lookup("java:comp/env/jdbc/OrderDB");
conn = db.getConnection();
```

> **注意：** この例では、DataSource が、"/test/OrderDataSource" という JNDI 名で data-sources.xml ファイルで指定されていることを前提としています。

mail セッション

Java mail の Session オブジェクトの環境要素は、次のようにして作成できます。

1. 次のようにして、application.xml ファイル内の JNDI 名前空間内の javax.mail.Session 参照を、<mail-session> 要素を使用してバインドします。

   ```
   <mail-session location="mail/MailSession"
      smtp-host="mysmtp.oraclecorp.com">
      <property name="mail.transport.protocol" value="smtp"/>
      <property name="mail.smtp.from" value="emailaddress@oracle.com"/>
   </mail-session>
   ```

   location 属性には、OC4J 固有のデプロイメント・ディスクリプタ内の <resource-ref-mapping> 要素の location 属性で指定された JNDI 名が含まれています。

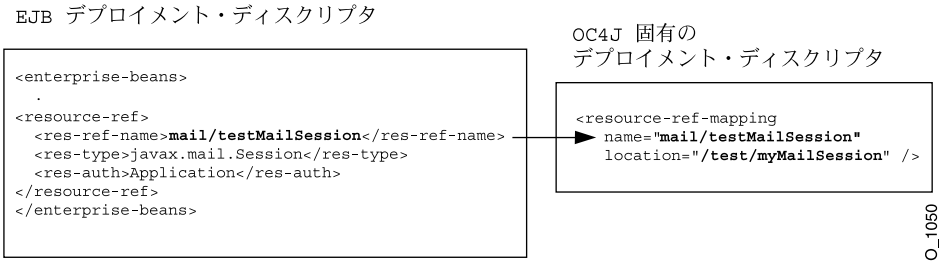
2. EJB デプロイメント・ディスクリプタ内の <res-ref-name> 要素で、論理名を作成します。この名前には、必ず先頭に "mail" を使用します。Bean コードでは、この参照のルックアップは、常に先頭に "java:comp/env/mail" が付きます。

3. EJB デプロイメント・ディスクリプタ内の論理名を、ステップ 1 で作成した OC4J 固有のデプロイメント・ディスクリプタ内の JNDI 名にマッピングします。

4. Bean 内で、"java:comp/env/mail" 接頭辞および EJB デプロイメント・ディスクリプタで定義された論理名でオブジェクト参照をルックアップします。

図 9-4 で示すように、Session オブジェクトは、JNDI 名 "/test/myMailSession" にバインドされています。Bean がこのリソースを認識する論理名は、"mail/testMailSession" です。これらの名前は、OC4J 固有のデプロイメント・ディスクリプタ内でマッピングされています。したがって、Bean は、実装内で、"java:comp/env/mail/testMailSession" 環境要素を使用して、バインドされた Session オブジェクトへの接続を取得できます。

図 9-4 Session リソース・マネージャのマッピング



この環境要素は、次の情報を使用して定義されています。

要素	説明
<res-ref-name>	起点の Bean 内で使用する Session オブジェクトの論理名。この名前には、先頭に "mail/" を使用する必要があります。この例では、mail セッションの論理名は "mail/testMailSession" です。
<res-type>	リソースの Java タイプ。Java mail の Session オブジェクトの場合、javax.mail.Session です。
<res-auth>	データベースへのサインオンが何によって行われるかを定義します。認識情報の提供者により、値は "Application" または "Container" のいずれかになります。

例 9-3 Java mail の Session の環境要素の定義

この環境要素は、EJB デプロイメント・ディスクリプタ内で、論理名、"mail/testMailSession"、javax.mail.Session のタイプ、および "Application" の認証機能を指定することにより、定義されます。

EJB デプロイメント・ディスクリプタ

```
<resource-ref>
  <res-ref-name>mail/testMailSession</res-ref-name>
  <res-type>javax.mail.Session</res-type>
  <res-auth>Application</res-auth>
</resource-ref>
```

"mail/testMailSession" の環境要素は、OC4J 固有のデプロイメント・ディスクリプタ内で、"test/myMailSession" という接続にバインドされている JNDI 名にマッピングされています。

OC4J 固有のデプロイメント・ディスクリプタ

```
<resource-ref-mapping
  name="mail/testMailSession"
  location="/test/myMailSession" />
```

デプロイされると、Bean は次のようにして Session オブジェクト参照を取得できます。

```
InitialContext ic = new InitialContext();
Session session = (Session) ic.lookup("java:comp/env/mail/testMailSession");

//The following uses the mail session object
//Create a message object
MimeMessage msg = new MimeMessage(session);

//Construct an address array
String mailTo = "whosit@oracle.com";
InternetAddress addr = new InternetAddress(mailto);
InternetAddress addrs[] = new InternetAddress[1];
addrs[0] = addr;

//set the message parameters
msg.setRecipients(Message.RecipientType.TO, addrs);
msg.setSubject("testSend()" + new Date());
msg.setContent(msgText, "text/plain");

//send the mail message
Transport.send(msg);
```

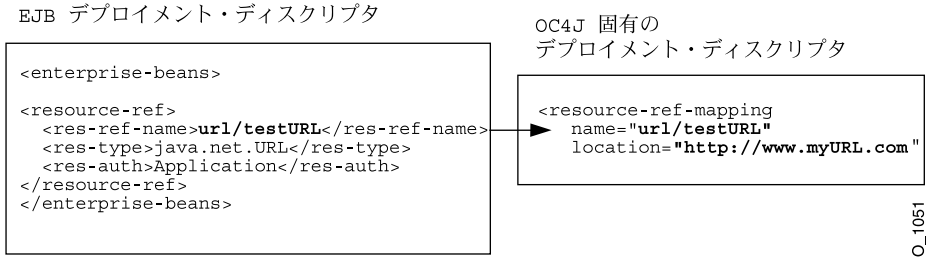
URL

Java の URL オブジェクトの環境要素は、次のようにして作成できます。

1. EJB デプロイメント・ディスクリプタ内の <res-ref-name> 要素で、論理名を作成します。この名前には、必ず先頭に "url" を使用します。Bean コードでは、この参照のルックアップは、常に先頭に "java:comp/env/url" が付きます。
2. EJB デプロイメント・ディスクリプタ内の論理名を、OC4J 固有のデプロイメント・ディスクリプタ内の URL にマッピングします。
3. Bean 内で、"java:comp/env/url" 接頭辞および EJB デプロイメント・ディスクリプタで定義された論理名でオブジェクト参照をルックアップします。

図 9-5 で示すように、URL オブジェクトは、URL"http://www.myURL.com" にバインドされています。Bean がこのリソースを認識する論理名は、"url/testURL" です。これらの名前は、OC4J 固有のデプロイメント・ディスクリプタ内でマッピングされています。したがって、Bean は、実装内で、"java:comp/env/url/testURL" 環境要素を使用して、URL オブジェクト参照を取得できます。

図 9-5 URL リソース・マネージャのマッピング



この環境要素は、次の情報を使用して定義されています。

要素	説明
<res-ref-name>	起点の Bean 内で使用する URL オブジェクトの論理名。この名前には、先頭に "url/" を使用する必要があります。この例では、URL の論理名は "url/testURL" です。
<res-type>	リソースの Java タイプ。Java の URL オブジェクトの場合、java.net.URL です。
<res-auth>	データベースへのサインオンが何によって行われるかを定義します。現在サポートされている値は "Application" のみです。認証情報は、アプリケーションが提供します。

例 9-4 URL の環境要素の定義

環境要素は、EJB デプロイメント・ディスクリプタ内で、論理名、"url/testURL"、java.net.URL のタイプ、および "Application" の認証機能を提供することにより、定義されます。

EJB デプロイメント・ディスクリプタ

```
<resource-ref>
  <res-ref-name>url/testURL</res-ref-name>
  <res-type>java.net.URL</res-type>
  <res-auth>Application</res-auth>
</resource-ref>
```

"url/testURL" の環境要素は、OC4J 固有のデプロイメント・ディスクリプタ内で、URL "http://www.myURL.com" にマッピングされています。

OC4J 固有のデプロイメント・ディスクリプタ

```
<resource-ref-mapping
  name="url/testURL"
  location="http://www.myURL.com" />
```

デプロイされると、Bean は次のようにして URL オブジェクト参照を取得できます。

```
InitialContext ic = new InitialContext();
URL url = (URL) ic.lookup("java:comp/env/url/testURL");

//The following uses the URL object
URLConection conn = url.openConnection();
```

一般的なエラーのトラブルシューティング

EJB の実行時に、次のエラーが発生する場合があります。

デプロイ時のメモリー不足エラー

デプロイ・プロセスがなんらかの理由で中断された場合、一時ディレクトリのクリーン・アップが必要となる場合があります。一時ディレクトリは、デフォルトでシステム上の `/var/tmp` です。デプロイ・ウィザードによって、デプロイ・プロセス時に情報を格納するために、一時ディレクトリのスワップ領域で 20MB が使用されます。プロセス完了時に、一時ディレクトリから追加のファイルがクリーン・アップされます。ただし、ウィザードが中断されると、一時ディレクトリをクリーン・アップする時間や機会がない場合があります。したがって、このディレクトリから追加のデプロイメント・ファイルを手動でクリーン・アップする必要があります。クリーン・アップを実行しないと、ディレクトリがいっぱいになる可能性があり、その後のデプロイができなくなります。`Out of Memory` エラーを受信した場合は、一時ディレクトリの使用可能領域をチェックしてください。

一時ディレクトリを変更するには、OC4J プロセスのコマンドライン・オプションを `java.io.tmpdir=<new_tmp_dir>` に設定します。このコマンドライン・オプションは「サーバー・プロパティ」ページで設定できます。最初に OC4J のホーム・ページにドリルダウンします。次に「管理」セクションにスクロールダウンします。「**サーバー・プロパティ**」を選択します。このページで、「コマンドライン・オプション」セクションにスクロールダウンし、「OC4J オプション」行に `java.io.tmpdir` の変数定義を追加します。新規 OC4J プロセスはすべて、このプロパティを使用して起動されます。

実行時のメモリー不足

実行時に OC4J メモリーが一貫して増大し続ける場合は、`application.xml` ファイルに無効なシンボリック・リンクが指定されている可能性があります。OC4J は、この `application.xml` ファイルのリンクを使用して、すべてのリソースをロードします。これらのリンクが無効な場合は、C ヒープの増大が続き、OC4J がメモリー不足となります。すべてのシンボリック・リンクが有効であることを確認し、OC4J を再起動します。

また、シンボリック・リンクが指し示すディレクトリ内の JAR ファイル数は最小限にしてください。使用していない JAR ファイルすべてをこれらのディレクトリから削除します。OC4J は、クラスとリソースの JAR をすべて検索します。したがって、アドレス空間へのマップおよびファイル・キャッシュによる時間やメモリーの消費量が増加します。

NamingException のスロー

EJB にリモートでアクセスしようとし、`javax.naming.NamingException` エラーが発生する場合、JNDI プロパティが正しく初期化されていない可能性があります。リモート・オブジェクトまたはリモート・サーブレットから EJB にアクセスする場合の JNDI プロパティの設定方法については、2-16 ページの「JNDI プロパティの設定」を参照してください。

デッドロック状態

デッドロックの原因が複数の Bean のコール・シーケンスにある場合、OC4J コンテナはデッドロック状態を検出し、違反している Bean の 1 つにあるデッドロック状態の詳細を示すリモート例外をスローします。

ClassCastException

他の共有 EJB クラスを参照する EJB または Web アプリケーションを使用する場合は、参照されるクラスを共有 JAR ファイルに配置する必要があります。状況によっては、WAR ファイルまたは共有 EJB クラスを参照する別のアプリケーションに共有 EJB クラスをコピーすると、クラス・ローダーの問題のために `ClassCastException` が発生する場合があります。正常に終了するためには、参照される EJB クラスを、そのアプリケーションの WAR ファイルまたは別のアプリケーションにコピーしないでください。

詳細は、9-2 ページの「クラスの共有」を参照してください。

リモート EJB からの NullPointerException のスロー

Web アプリケーションからリモート EJB にアクセスすると、「java.lang.NullPointerException: domain was null」というエラーが表示されます。この場合、dedicated.rmicontext が true に設定されている EJB にアクセスするときは、環境プロパティをクライアントに設定する必要があります。

次の例は、この追加環境プロパティを使用する方法を示しています。

```
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, "admin");
env.put(Context.PROVIDER_URL, "ormi://myhost-us/ejbsamples");
env.put("dedicated.rmicontext","true"); // for 9.0.2.1 and above
Context context = new InitialContext(env);
```

dedicated.rmicontext の詳細は、10-7 ページの「ロード・バランシングのオプション」を参照してください。

10

EJB のクラスタリング

ロード・バランシングやフェイルオーバーを含めたクラスタリングの提供方法は、HTTP リクエストと EJB 通信では異なります。これは、Web コンポーネントと EJB コンポーネントで使用するプロトコルが異なるためです。この章では、EJB のクラスタリングについて説明します。HTTP フェイルオーバーおよびロード・バランシング環境の設定方法など、Oracle Application Server のクラスタリングの概要については、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』のクラスタリングに関する章を参照してください。

この章には、次の項目が含まれます。

EJB のクラスタリングの概要

クラスタリングされた EJB は、それぞれ独自の方法で動作します。ただし、クラスタリングされるのはステートフル Session Bean のみです。EJB クラスタを作成するには、クラスタに含めるノードを指定し、ノード内の各 OC4J インスタンスを同一のマルチキャスト・アドレス、ユーザー名およびパスワードで構成し、これらのノードの 1 つに EJB をデプロイします。

警告： EJB クラスタリングは、ORMI プロトコルを介してのみ動作します。RMI/IIOP プロトコルでは動作しません。

EJB クラスタリングには次の特性があります。

- HTTP のクラスタリングとは異なり、クラスタに含まれる EJB は、アイランドにサブグループ化できません。かわりに、クラスタ内のすべての EJB が 1 つにグループ化されます。
- トランザクションはフェイルオーバーできません。中断されたトランザクションを別の Bean に再インスタンス化する機能はありません。かわりに、トランザクションはロールバックされるため、最初からやり直す必要があります。
- ロード・バランシングは、EJB 用のクラスタ内の OC4J プロセス全体にわたってランダムな方法で発生します。
- ステートフル Session Bean のクラスタリングのパフォーマンスは、選択するレプリケーションのタイプおよびロード・バランシングのオプションによって変わります。

各種 Session Bean のクラスタリングについては、次の各項で説明します。

- ステートレス・セッションのクラスタリング
- ステートフル Session Bean のクラスタリング
- HTTP と EJB のクラスタリングの組合せ

注意： クラスタリングの機能の概要は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』のクラスタリングに関する章を参照してください。

ステートレス・セッションのクラスタリング

ステートレス Session Bean では、クラスタ内のホスト間でレプリケート対象となる状態がありません。したがって、フェイルオーバー・オプションは必要ありません。ロード・バランシングは、ランダム・アルゴリズムを使用する OPMN によって自動的に提供されます。ステートレス Session Bean はクラスタリングされません。ロード・バランシングは、OPMN のコンポーネントが互いに認識する環境で発生します。10-7 ページの「ロード・バランシングのオプション」で説明されているオプションを使用して、ロード・バランシングの頻度をクライアントから構成できます。

ステートフル Session Bean のクラスタリング

ロード・バランシングは、ランダム・アルゴリズムを使用する OPMN によって自動的に提供されます。フェイルオーバーでは、元の Bean が予期しないときに終了した場合にリクエストが別の OC4J プロセスに転送されるように、Bean の状態がレプリケートされる必要があります。10-7 ページの「ロード・バランシングのオプション」で説明されているオプションを使用して、ロード・バランシングの頻度をクライアントから構成できます。

フェイルオーバーのために、ステートフル Session Bean では、ホスト間の状態をレプリケートする必要があります。ステートフル Session Bean のレプリケーションにはオプションが 3 つあります。各オプションでは、Bean の状態が送信される間隔が定義されます。状態のすべてがクラスタ内の他のすべての OC4J プロセスに送信されるため、パフォーマンスに影響を与える場合があります。状態を送信する回数が少ないほど、パフォーマンスは向上します。ただし、パフォーマンスと、Bean インスタンスの障害の全範囲を対象とするように Bean の状態がレプリケートされる確実性との間にはトレードオフがあります。これらを考慮して、次のレプリケーション・モードの 1 つを選択します。

- JVM 終了レプリケーション : ステートフル Session Bean は、JVM が終了すると、クラスタ内の（マルチキャスト・アドレス、ポートが同一の）別のホストの 1 つにレプリケートされます。このモードでは、JDK 1.3 シャットダウン・フックが使用されるため、JVM バージョン 1.3 以上を使用する必要があります。このオプションは、状態のレプリケートが 1 回のみであるため、パフォーマンスが最も良くなります。ただし、次の理由により、信頼性は高くありません。
  - 予期しないときにホストが終了した場合、状態はレプリケートされません。
  - Bean の状態は常に 1 つのホストにのみ存在するため、状態がレプリケートされずに失われる危険性が高くなります。
- コール終了レプリケーション : ステートフル Session Bean の状態は、各 EJB メソッドのコールの終了時に、クラスタ内の（マルチキャスト・アドレス、ポートが同一の）すべてのホストにレプリケートされます。ノードの電源が切断された場合でも、状態はすでにレプリケートされています。この方法は、状態の送信回数が多くなるため、JVM 終了レプリケーション・モードよりパフォーマンスが低下します。ただし、信頼性の保証は高くなります。

これらのステートフル Session Bean の各クラスタリング・オプションの構成および実装の詳細は、10-6 ページの「ステートフル Session Bean 用の EJB レプリケーションの構成」を参照してください。

HTTP と EJB のクラスタリングの組合せ

EJB を起動するサーブレットがある場合は、HTTP と EJB の両方のクラスタリングを構成する必要があります。HTTP のクラスタリングのオプションについては、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』のクラスタリングに関する章を参照してください。

EJB のクラスタリングの有効化

OC4J クラスタの設定方法の詳細は、『Oracle Application Server Containers for J2EE ユーザーズ・ガイド』を参照してください。この項では、クラスタ内の EJB の状態レプリケーションに関する構成方法のみを説明します。

OC4J ノードで EJB のクラスタリングを有効にするには、次の手順を実行します。

1. ユーザー名およびパスワードを含め、同一のマルチキャスト・アドレス（ホストとポート番号）でクラスタ内の各ホストを構成します。
2. ステートフル Session Bean を使用している場合は、状態レプリケーションのタイプを選択します。
3. クラスタリングする EJB をデプロイします。

EJB クラスタリング用のマルチキャスト・アドレスの構成

Oracle Enterprise Manager の「OC4J インスタンス」ページ内で、次の手順を実行します。

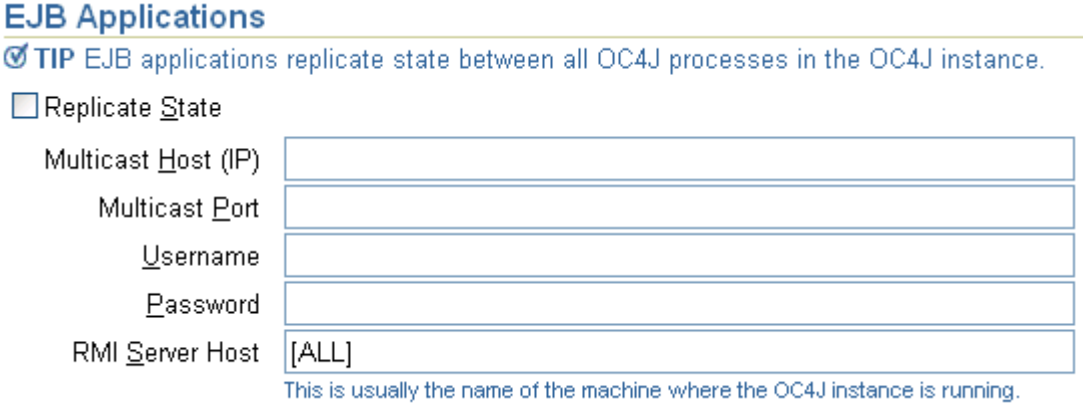
1. 「管理」ページを選択します。
2. 「インスタンス・プロパティ」列の「レプリケーション・プロパティ」を選択します。
3. 「EJB アプリケーション」セクションにスクロールダウンします。図 10-1 にこのセクションを示します。
4. 「レプリケート状態」チェックボックスを選択します。
5. オプションで、マルチキャスト・ホストの IP アドレスとポート番号を指定します。マルチキャスト・アドレスのホストとポートを指定しない場合、ホストの IP アドレスは 230.230.0.1、ポート番号は 9127 にデフォルト設定されます。ホスト IP アドレスは、224.0.0.2 ～ 239.255.255.255 の間で設定する必要があります。HTTP と EJB のマルチキャスト・アドレスに同じマルチキャスト・アドレスを使用しないでください。

   次のホストを ping することで、マルチキャスト機能のネットワークをテストできます。

- すべてのマルチキャスト・ホストを ping するには、`ping 224.0.0.1` を実行します。
- すべてのマルチキャスト・ルーターを ping するには、`ping 224.0.0.2` を実行します。

6. ユーザー名とパスワードを指定します。これは、マルチキャスト・アドレス上のクラスタ内の他のホストに対してそのホスト自体を認証するために使用されます。ユーザー名とパスワードは、同一クラスタ内に存在するためには、マルチキャスト・アドレス内で一貫している必要があります。
7. 「RMI サーバー・ホスト」フィールドに、OC4J インスタンスが存在しているホスト名を指定します。
8. JAR ファイル内の `orion-ejb-jar.xml` ファイル内でステートフル Session Bean のレプリケーションのタイプを構成します。詳細は、10-6 ページの「ステートフル Session Bean 用の EJB レプリケーションの構成」を参照してください。これらは、デプロイ前に `orion-ejb-jar.xml` ファイル内で構成するか、またはデプロイ後に Oracle Enterprise Manager の画面から追加します。デプロイ後に追加する場合は、アプリケーション・ページから JAR ファイルにドリルダウンします。

図 10-1 EJB の状態レプリケーションの構成



ステートフル Session Bean 用の EJB レプリケーションの構成

orion-ejb-jar.xml ファイルを変更して、ステートフル Session Bean に状態レプリケーション構成を追加します。ステートフル Session Bean のレプリケーション・タイプは、Bean のデプロイメント・ディスクリプタ内で構成するため、各 Bean は、それぞれ異なるレプリケーション・タイプを使用できます。

VM 終了レプリケーション

orion-ejb-jar.xml ファイルの <session-deployment> タグの replication 属性を "VMTermination" に設定します。次のようにします。

```
<session-deployment replication="VMTermination" .../>
```

コール終了レプリケーション

orion-ejb-jar.xml ファイルの <session-deployment> タグの replication 属性を "EndOfCall" に設定します。次のようにします。

```
<session-deployment replication="EndOfCall" .../>
```

JNDI 名前空間レプリケーションを含めた EJB のクラスタリング

EJB のクラスタリングが有効な場合は、JNDI 名前空間レプリケーションもクラスタ内の OC4J インスタンス間で有効です。1 つの OC4J インスタンスの JNDI 名前空間への新規バインドは、クラスタ内の他の OC4J インスタンスに伝播されます。再バインドやバインドの解除はレプリケートされません。レプリケーションは、OC4J アイランドの範囲外で完了します。つまり、OC4J インスタンス内の複数のアイランドには、レプリケートされた同じ JNDI 名前空間への可視性があります。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。

ロード・バランシングのオプション

EJB のロード・バランシングは、クラスタ内に含まれる OC4J プロセス全体で発生します。

クライアントは、最初のルックアップが実行されるときに、OC4J プロセスをランダムに取得します。クライアントを処理する OC4J プロセスの選択は、クラスタ内のプールされている OC4J プロセスから常にランダムに行われます。ただし、クライアントが次のことを実行するように選択できます。

- オプションを何も設定していない場合、クライアントは、最初のルックアップで選択された OC4J プロセスと対話を続けます。
- 2 つのオプションのいずれかを設定している場合、クライアントは、実装における特定のポイントで対話する OC4J プロセスを選択します。クライアントが別の OC4J プロセスを要求するたびに、このプロセスも、クラスタに含まれる OC4J プロセスからランダムに選択されます。

  オプションは、次のとおりです。

  - `LoadBalanceOnLookup` プロパティ : このプロパティを true に設定すると、クライアントは、ルックアップが実行されるたびに、クラスタ内のプールされているプロセスから別の OC4J プロセスをランダムに選択します。このオプションでは、`RMIInitialContextFactory` オブジェクトのみを使用してください。

    次の例は、`InitialContext` を取得する前に、JNDI プロパティ内のクライアントの `LoadBalanceOnLookup` プロパティを true に構成します。

    ```
    env.put("LoadBalanceOnLookup", "true");
    ```

- `dedicated.rmicontext` プロパティ : このプロパティを true に設定すると、クライアントが新しい `InitialContext` を取得するたびに、新しい OC4J プロセスも取得します。クライアント内で複数の OC4J プロセスを使用する場合は、`LoadBalanceOnLookup` プロパティよりパフォーマンスが高く、アプリケーション・サーバーにかかる負荷が少なくなります。

EJB 状態のレプリケーション機能は使用せずに、OC4J プロセス間のリクエストのロード・バランシングを希望する場合は、次の各項のオプションの説明を参照してください。

- 静的な検出を使用したロード・バランシング
- DNS ロード・バランシング

静的な検出を使用したロード・バランシング

EJB のレプリケーションを使用せずに、複数の OC4J プロセス間のリクエストをロード・バランシングする場合は、これらのプロセスすべての URL を JNDI の URL プロパティに指定することによって、静的な検出を使用できます。

ロード・バランシングおよびフェイルオーバーのために検出する必要があるすべての OC4J ノードの JNDI アドレスは、ルックアップ URL で提供され、各アドレスはカンマで区切られています。たとえば、次の URL 定義は、ロード・バランシングおよびフェイルオーバーで使用する、3 つの OC4J ノードをクライアント・コンテナに提供します。

```
java.naming.provider.url=ormi://s1:23791/ejbsamples,
         ormi://s2:23793/ejbsamples, ormi://s3:23791/ejbsamples;
```

DNS ロード・バランシング

EJB のレプリケーションを使用せずに、DNS を使用してリクエストをロード・バランシングする場合は、次のようにします。

1. DNS 内で、1 つのホスト名を複数の IP アドレスにマッピングします。各ポート番号は、それぞれの IP アドレスに対して同じであることが必要です。DNS サーバーは、ラウンドロビン法またはランダムでアドレスを返すように設定します。

   IP アドレスで OC4J の実行を識別します。ポート番号は RMI ポート番号です。

2. クライアントでの DNS のキャッシュをオフにします。UNIX マシンの場合は、次の手順で DNS のキャッシュをオフにする必要があります。

   a. クライアントでの NSCD デーモン・プロセスを停止します。

   b. `-Dsun.net.inetaddr.ttl=0` オプションを使用して、OC4J クライアントを起動します。

3. 各クライアント内で、初期コンテキスト・ファクトリを使用して初期コンテキストを作成します。プロバイダ URL には、`ormi://` 接頭辞を使用します。OC4J の IP アドレスがマップされる DNS サーバー内で単一のホスト名と、クライアントのプロバイダ URL 内の共通の RMI ポートを使用します。

4. `dedicated.rmicontext` プロパティを true に設定します。

DNS サーバーでルックアップが発生するたびに、DNS サーバーは、マップされている IP アドレスの 1 つを返します。

例 10-1 RMIInitialContextFactory の例

この例では RMIInitialContextFactory オブジェクトが使用されていますが、DNS ロード・バランシングには任意の初期コンテキスト・ファクトリを使用できます。この例にある myserver は、サーバーのリスト用に DNS サーバーに設定されているホスト名です。RMI ポートはデフォルトのポートに設定されています。

```
java.naming.factory.initial=
          com.evermind.server.rmi.RMIInitialContextFactory
java.naming.provider.url=ormi://myserver/applname
java.naming.security.principal=admin
java.naming.security.credentials=welcome
dedicated.rmicontext=true
```

11

Active Components for Java

Active Components for Java（AC4J）は、J2EE を拡張するフレームワークで、複数のアプリケーションがピアとして疎結合方式で相互に作用できるようにします。ビジネスの相互作用で互いに影響がある複数のアプリケーションは、サービスのリクエストや結果の応答を行うために情報を非同期で交換します。

この章では、自律型アプリケーション間における疎結合の相互作用を管理するためのオラクル社のソリューションについて説明します。次の項目が含まれます。

- AC4J のメリット
- AC4J のアーキテクチャの概要
- AC4J コンポーネントの概要
- AC4J およびデータベースのインストールと構成
- AC4J の例
- 例の説明
- AC4J でのアクティブ EJB のデプロイメント

AC4J のメリット

多くの場合、ビジネス・アプリケーションでは、長期間存続する相互作用を様々なアプリケーション・サービス間で実行できる必要があります。アプリケーションは、リソースの制限なしに、システムのクラッシュに対応する機能があり、長期間にわたり他のアプリケーションと通信できる必要があります。各アプリケーションは、別のアプリケーションと通信するとき、自律型ピアとして存在します。つまり、両方のアプリケーションは相互にリクエストできますが、いずれのアプリケーションも、ピア・アプリケーションが所有するリソースは制御できません。この環境では、多くの場合、2 つのアプリケーション間の通信は切断されています。つまり、アプリケーションはシステムの常時接続に依存できません。実行されるタスクは、完了までに数日から数か月を要する場合があるため、長期間存続する相互作用では非同期通信が必須です。

具体的な例でこの処理について説明します。発注（PO）処理システムを実装するとします。このシステムでは、クライアント（顧客）が非同期の発注リクエストを送信でき、時間のかかる発注処理の完了を待機しないで済むようにします。発注処理サービス自体は、2 つの非同期リクエストを送信します。1 つは与信サービス用、もう 1 つは在庫サービス用で、発注明細品目について顧客の与信を確認し、在庫をチェックします。サービスに対する 2 つのパラレルなリクエストの非同期的性質によって、発注処理サービスは、顧客のリクエストに基づいて発注を作成し、対応する発注に関して適切な時期に応答できます。両方のサービスがその応答とともに返されると、発注処理サービスは 2 つの結果を統合し、発注を取り消すか、処理を続行します（応答が同時に返されることはほとんどなく、数時間から数日、場合によっては数週間を要する可能性があります）。クライアントは、最初に発注処理サービスから返された発注番号に基づいて、いつでも発注のステータスを問い合せることができます。

このようなシステムを実装するには、フレームワークによって、アプリケーションが自律型ピアとして長期間存続する相互作用を実行できるようにする必要があります。存続期間が指定されていないアプリケーションには高い信頼性が必要です。期間中にシステム障害が発生した場合に備えて、リカバリ可能また再起動可能であることが必要です。アプリケーションはスケーラブルであることも必要です。つまり、長期間実行のアプリケーションは、長期間にわたって実行をブロックしたり、リソースをロックすることはできません。アプリケーションを適切な時間内で実行するために、フレームワークでは、複数の操作を並行して実行し、パフォーマンスを強化する必要があります。

Sun 社が開発した Java 2 Platform, Enterprise Edition（J2EE）では、信頼性があり、スケーラブルな密結合アプリケーションを作成するための優れた環境が提供されます。これに役立つ主なコンポーネントの 1 つが Enterprise JavaBeans（EJB）です。また、J2EE は、前述の例で説明したシステムなど、長期間存続する相互作用の要件を満たす新しいフレームワークを開発するためのビルディング・ブロックでもあります。長期間存続する相互作用をサポートするアプリケーションを簡単に作成できるように、EJB、Java Message Service（JMS）および Java Transaction API（JTA）を組み合せて使用できます。

J2EE の機能を使用できる状況で AC4J を提供する意味について考えます。その後、EJB、JMS および JTA など既存の J2EE テクノロジが、このような発注処理システムの作成に十分ではない理由を説明します。

EJB の密結合の同期通信では、長期間存続する相互作用または自律型 peer-to-peer 通信を実行できません。一方、JMS の疎結合の非同期通信では、アプリケーション・サービス間で必要となるコンポーネント・ベースのリクエスト / レスポンスによる通信を、少なくとも単純な方法では実行できません。さらに、2 フェーズ・コミット内のすべてのリソースを制御するために JTA コーディネータが必要な場合は、自律型リソースをグローバル・トランザクションに含めることができません。このような理由から、J2EE 単体では、長期間存続する相互作用を使用するアプリケーションを容易に開発するために必要な完全なソリューションは提供されません。

長期間存続する相互作用に関する要件をサポートするためのビジネス・ソリューションには、EJB、JMS および JTA のメリットを組み合せたアプリケーション・コンポーネント方法論が必要であると考えられています。特に、新しい方法論には、次のニーズに対する解決方法が組み込まれている必要があります。

アプリケーションは、EJB メソッドを対話形式で起動できる必要がありますが、非同期通信によって提供される非接続、非ブロック・モードで実行される必要があります。つまり、EJB メソッドの起動と JMS メッセージ機能のプロパティの真の意味での統合です。この結果、Bean 実装にリクエストを直接送信できますが、これは両通信者間の静的な接続や、応答が受信されるまでのリクエスタのブロックを必要としない疎結合方式で送信されます。さらに、例外を非同期環境内で処理し、クライアントに返す必要があります。非同期環境内で実行されているすべてのサービスに、パラメータやローカル変数など、データに対するコンテキストが含まれている必要があります。環境では、プログラムまたは宣言による実行フローの遅延、およびタイムアウト・プロパティを基準とした特定サービスの操作の強制実行を可能にする基本的な時間管理機能も提供される必要があります。

AC4J のアーキテクチャの概要

AC4J Bean は、アクティブ EJB とも呼ばれます。これらは、JMS の属性を拡張する EJB（ステートレス Session Bean または Entity Bean）です。したがって、アクティブ EJB には、JMS の非同期機能に加えて、すべてのプロパティ、サービスおよび EJB のサービス品質が備わっています。アクティブ EJB には、ビジネス・ロジックが含まれます。これらの Bean は疎結合 Bean であるため、各 Bean は、ピア・オブジェクトとの通信で、リクエスト / レスポンス型の同期通信またはメッセージ・ドリブン型の非同期通信のいずれか（あるいはその両方）を使用できます。アクティブ EJB は EJB の仕様に完全に準拠しているため、各アクティブ EJB は、コンポーネント・ベースで再利用可能であること、およびセキュリティやトランザクション動作などの EJB サービスに対するアクセスの提供など、EJB のすべての機能を使用します。

アクティブ EJB はすべて、AC4J の相互作用内に存在します。リクエスト / レスポンス型の同期通信または一方向の非同期通信に関するすべてのプロパティなど、長期間存続する相互作用に必要なものはすべて、AC4J の相互作用内にカプセル化されます。AC4J の相互作用は長期間存続する作業ユニットで、ビジネス・トランザクションの動作を反映します。リソースの競合を防ぐためにグローバル・トランザクションを独自の方法論で置き換え、自律型リソースをトランザクション内に含めることができ、異なる制約のシステム間で相互に作用できるようにします。また、プロセス間の一連のデータ交換をグループ化します。

すべての AC4J の相互作用の基礎として、AC4J データ・バスは、AC4J プロセス間で AC4J データ・トークン（アクティブ・データおよびイベント）をルーティングします。AC4J データ・バスは、AC4J の基本コンポーネントです。アプリケーションは、AC4J データ・バスに連結され、データを交換したりサービスをリクエストします。データ・バスは、登録されている AC4J リアクションを使用して AC4J データ・トークンのルーティングとマッチングを行い、連結したアプリケーションの透過的なロード・バランシングを可能にします。AC4J データ・トークンは、サービスのリクエストやサービス・リクエストからの応答、またはタイマーの時間切れなどの例外条件を表します。

AC4J の相互作用には、1 つ以上の AC4J プロセスを含めることができます。1 つの AC4J プロセスは、1 つのビジネス・タスクを表します。各 AC4J プロセスでは、アプリケーション・ロジックが実行されるトランザクション・コンテキストおよびセキュリティ・コンテキストが提供されます。また、複数の操作の並行実行を管理します。これは AC4J リアクションと呼ばれます。さらに、アクティブ・データ（ローカル変数、入力パラメータおよび応答）をカプセル化し、受信アクティブ・データとその宛先の受信者を照合します。これは AC4J プロセスに登録されている AC4J リアクションが実行します。AC4J プロセスは、コール元に応答を返す方法を決定するデータ・フロー・コンテキストも維持します。コンテキストには、応答データの返信先が記述されます。

アプリケーションは、並行して実行される AC4J リアクションに動的に分割されます。各 AC4J リアクションは、指定した条件が満たされ、必要なすべてのデータが着信すると実行されます。AC4J リアクションは、適切なアクティブ・データの着信を待機し、着信後に対象の Bean メソッドを起動するリアクティブなエンティティです。メソッドによってデータが処理された後、AC4J リアクションはコンテキストに基づいて応答を返します。AC4J リアクションの有効範囲内で実行されるすべてのアクティビティは、JTA トランザクションの ACID（原子性、一貫性、独立性および永続性）プロパティに基づいて実行されます。

AC4J リアクションは、ビジネス・タスクの詳細な作業を実行します。次の処理で使用します。

- AC4J データ・バスとの間でデータのプッシュやプルを行う場合
- サービス・リクエストを処理する場合
- 他のアクティブ EJB からサービスをリクエストする場合
- コール元のアクティブ EJB のビジネス・タスク、またはアプリケーション・クライアントに結果を返す場合
- ビジネス・トランザクションの一貫性を保持するためのビジネス制約を強制する場合
- 障害時にアプリケーションを再起動可能にする場合

AC4J では特定の時間管理機能も提供されます。この機能を使用すると、コンポーネントの実行前に一定時間実行フローを遅延させたり、実行に必要なデータ・トークンがまだ着信していない場合でも、一定時間（タイムアウト）経過後にコンポーネントを実行できます。

要約すると、AC4J では、EJB は次のことを実行することによって疎結合方式で相互に作用できます。

- キューとトピックおよび関連する JMS 構成をアプリケーションから隠します。
- 通信メッセージの書式の自動定義を提供します。
- メッセージを自動的にパックまたはアンパックします。
- サービス・リクエストを適切なサービス・プロバイダに自動的にルーティングします。
- セキュリティ・コンテキストを自動的に伝播します。
- 認可と識別情報の偽装を提供します。
- 例外の自動ルーティングと処理を行います。
- EJB アプリケーションのトランザクション型データ・ドリブン実行を提供します。
- EJB の透過的なスケジューリングとアクティブ化、および EJB メソッドの実行を提供します。
- フォーク操作および結合操作をサポートします。これには、EJB メソッドのパラレル起動、およびその結果の同期が含まれます。
- 進行中の作業の自動追跡を提供します。

これらの機能はすべて、EJB のフレームワークに完全に統合されています。

AC4J コンポーネントの概要

この項には、次のトピックが含まれます。

アクティブ EJB

非アクティブな従来の EJB では、クライアントのリクエストを即時に処理し、結果を迅速に返す必要がありました。迅速に配信できない場合、EJB は使用不可になります。AC4J によって、標準的なステートレス Session EJB とエンティティ EJB をアクティブにすることができます。アクティブ EJB では、サービスのリクエストを実際のサービス実行から分離できます。実際に起動する EJB メソッドとその起動時期を制御するポリシーは、サービス・プロバイダの EJB によって制御されます。この分離によって、サービス・リクエストとサービス・プロバイダは、自律型ピアとして相互に作用できます。

注意： AC4J 内のサンプル・コード、ファイル名およびディレクトリ・パスで JEM という名前が使用されている場合があります。これは AC4J の内部名です。

アプリケーションは、JEMHandle を作成またはルックアップして、EJB インタフェース内で公開されているビジネス・タスクからサービスをリクエストできます。

アクティブ EJB は、JEMHandle オブジェクトによって一意に識別されます。JEMHandle オブジェクトは、次のものをカプセル化します。

- アクティブ EJB 名
- J2EE アプリケーション名
- EJB JAR 名
- EJB 名
- クラス名
- EJB ホーム・インタフェース名
- EJB リモート・インタフェース名
- AC4J データ・バスが存在するデータベースのインスタンス名（SID）
- EJB の主キー（Entity Bean でのみ使用可能）

相互作用

相互作用は長期間存続する作業ユニットで、ビジネス・トランザクションの動作を反映します。ビジネス・トランザクションは、異なる組織に存在する複数のアプリケーションに関連する場合があります。このような非接続環境でのビジネス・トランザクションの継続時間は長くなる場合があるため、ビジネス・トランザクションの存続期間は、ローカル・トランザクションまたはグローバル・トランザクションの存続期間とは異なります。

相互作用では、達成するビジネスの目標が示されます。たとえば、顧客がある品目を購入する場合、顧客が代金を支払い品目を受け取るまでの、必要なすべての操作を相互作用と呼びます。相互作用は、ビジネス・トランザクションのグローバル実行コンテキストを提供して、一連のビジネス・データ交換をグループ化します。

これらのアプリケーションは、それぞれ独立して実行でき、データをコミットまたはロールバックできます。他のアプリケーションに関する知識は必要ありません。ただし、これらのアプリケーションは、相互に無関係であるとはみなされません。これは、アプリケーション間に形成されている関連の調整と一貫性の維持が必要なためです。ビジネス・トランザクションに一貫性がない場合は、そこに含まれるアプリケーションのリカバリが必要になる場合があります。アプリケーションのリカバリは、補償リアクションを登録することにより実行できます。たとえば、サプライヤが発注リクエストを確認して購入者に返した後、購入者

は、補償リアクションを登録する必要があります。このリアクションは、たとえば、製造部門での遅延のため、注文の履行が不可能になったことを通知する追加応答がサプライヤから送信されるのを監視します。サプライヤによるリクエストの確認が取り消されると、購入者の補償リアクションがそれに一致して起動され、購入者のアプリケーションは一貫性をリカバリできます。このリアクションでは、別のサプライヤを選択して品目をリクエストしたり、発注プロセスを完全に中止することができます。

相互作用は、相互作用識別子（IID）によって一意に識別されます。相互作用には、複数のプロセスを含めることができます。

プロセス

プロセスは、ビジネス・タスクを識別します。発注の例では、発注の作成、在庫のチェックおよび顧客の与信チェックの各ビジネス・タスクにプロセスが存在します。

各プロセスは、次の処理を実行します。

- 詳細な作業を実行するリアクションをカプセル化します。
- ビジネス・タスクの入力パラメータとその応答を含むデータ・トークンをカプセル化します。
- 起動元のビジネス・タスクに応答を返す方法を決定する、データ・フロー・コンテキストを維持します。

プロセスは、`JEMPortHandle` オブジェクトによって一意に識別されます。このオブジェクトは、プロセス・コンテキスト、およびそのプロセスが属するアクティブ EJB の `JEMHandle` をカプセル化します。プロセス・コンテキストとは、相互作用識別子とプロセス・アクティブ化識別子の組合せです。AC4J は、コール操作内で、相互作用識別子およびプロセス・アクティブ化識別子を自動的に作成します。アプリケーションでは、これらの識別子を AC4J の `JEMSession::call` 操作で提供することもできます。

リアクション

リアクションは、プロセスの詳細な作業を実行します。アプリケーションは、この構成を使用して、アクティブ EJB メソッドの実行をトリガーする相関データ・トークンのコレクションの可用性について、永続的な関連を指定できます。

AC4J のコール操作の結果としてプロセスが作成されると、AC4J によってベース・リアクションが暗黙的に作成されます。アプリケーションは、実行時に `JEMReaction::registerReaction` 操作を使用して明示的にリアクションを作成し、データ・トークンで同期することができます。暗黙的または明示的な `registerReaction` 操作では、マッチングの成功時に実行するアクティブ EJB メソッドが指定されます。

リアクションは、受信リクエストを処理し、リクエストに基づいて結果を返し、ビジネス制約を強制してアプリケーションの一貫性を維持します。すべてのデータ・トークンが使用可能になり、条件に一致すると、リアクションが起動されます。起動されたリアクションは、1 つ以上の入力データ・パラメータをコンシュームして処理し、他のリアクションに対する

1 つ以上の出力データ・トークンを作成します。リアクション（アクティブ EJB メソッド）から返された結果は、AC4J のインフラストラクチャによってデータ・トークンに変換され、コール元にルーティングされます。リアクションは、他のアクティブ EJB から追加サービスをリクエストして、ビジネス・タスクを完了できます。このリクエストによって作成された新規のデータ・トークンは、AC4J のデータ・バスによってプッシュされ、ルーティングされます。

プロセス・コンテキスト・インスタンス内のリアクションは、次の方法で、データ・トークンを AC4J のデータ・バスにプッシュできます。

- 同一または異なる相互作用コンテキスト・インスタンス内の別のプロセスからサービスをリクエストする、1 つ以上の `JEMReaction::call` 操作を発行する方法。
- 例外操作をコール元プロセスに返すかスローする方法。
- `JEMReaction::registerReactionTimer` 操作を使用して、タイマーを登録する方法。

タイマーが時間切れになると、AC4J は、タイムアウト例外のデータ・トークンを現行のリアクション・コンテキスト・インスタンスにプッシュします。

プロセス・コンテキスト・インスタンス内のリアクションは、`JEMReaction::registerReaction` メソッドを使用して 1 つ以上のリアクションを現行のプロセス・コンテキスト・インスタンスに登録することによって、AC4J のデータ・バスからデータ・トークンをプルできます。

各ビジネス・タスクには、1 つ以上のリアクションを含めることができます。1 つのリアクションをリクエスト用に、別のリアクションを応答用に使用して、リクエスト / レスポンス環境の非同期的な性質をサポートします。リアクションの数は、必要なリクエスト数と応答数によって決まります。

11-15 ページの「AC4J の例」の例は、プロセス間の非同期通信を受信しながら、リクエスト / レスポンス環境を維持する方法を示します。`processOrder` プロセスは、発注を作成するビジネス・タスクです。発注を作成するには、在庫と顧客の与信をチェックする必要があります。`processOrder` リアクションは、次のプロセスを起動します。

- `checkINV`

  顧客が新規の購買を要求してその品目のデータを指定すると、`checkINV` プロセスがアクティブ化され、アクティブ EJB の `JEMInventoryBean` がインスタンス化され、ベース・リアクション（`checkINV`）が起動されます。その後、このリアクションは結果を `processOrder` プロセスとアクティブ EJB の `JEMPurchaseOrderBean` に返します。

- `checkCRED`

  顧客の与信をチェックするために、このプロセスがアクティブ化され、アクティブ EJB の `JEMCreditBean` がインスタンス化され、ベース・リアクションの `checkCRED` がこれに反応します。その後、このリアクションは結果を `takeOrder` プロセスとアクティブ EJB の `JEMPurchaseOrderBean` に返します。

非同期のリクエストを checkINV プロセスと checkCRED プロセスに送信した後、processOrder リアクションは、同じプロセス内に別のリアクション（processOrderCallback）を登録して、checkCRED プロセスと checkINV プロセスから応答が返されるのを待機します。これらのプロセスから返されたすべてのデータ・トークンが使用可能になると、processOrderCallback リアクションが起動して応答を処理します。

注意： リソースをロックしないという AC4J の要件を満たすには、コールは非同期の AC4J コールであることが必要です。ただし、別の Bean に対して、同期の EJB コールを実行することもできます。

AC4J データ・トークン

リアクションのアクティブ化は、データ・トークンの可用性によってトリガーされます。可用性は、1 つ以上のデータ・トークンの着信によって、正しい条件と正しいアクセス・モードで定義されます。

AC4J のコール操作を使用してアプリケーションがサービスをリクエストすると、次の内容で構成されたリクエスト・データ・トークンが自動的にプッシュされます。

- プロセス・ディスクリプタ。リクエストされたサービスを指定します（takeOrder など）。
- リクエストの宛先であるサービス・プロバイダの JEMPortHandle リクエスト・オブジェクト。
- JEMPortHandle 応答オブジェクト。プロセス・コンテキスト（相互作用識別子とプロセス・アクティブ化識別子）インスタンス、およびサービス・プロバイダから後で結果を受信するリクエスタ・プロセスの JEMHandle が含まれます。
- ビジネス・タスクの入力引数。サービス・プロバイダがサービスを実行するために使用します。

その後、リアクションにより、アクティブ EJB が結果を返すか例外をスローしたときに AC4J によって自動的に生成された応答データ・トークンが返されると、AC4J はルーティング情報を指定します。返された情報をコール元プロセスに送信し、その応答データ・トークンのポート・ハンドル・オブジェクトを指定します。返すプロセスのコール元が別のプロセスではなくクライアントの場合、データ・バスは、応答データ・トークンを特別なデータ・バス領域に格納します。クライアントは、JEMSession::receiveReactionResponseObjectInstance 操作を使用して、この領域から応答データ・トークンを取得できます。

入力または出力のデータ・トークン内で搬送されるオブジェクトのデータ型は、基本的なデータ型（Integer、String、Float、Boolean など）または構成されたクラス・タイプ（Java のシリアライズ可能なオブジェクトなど）です。

データ・バス

アプリケーションのスケーラビリティ、自律性、および可用性を改善するためには、サービスをリクエストするコンポーネントが、そのサービスを提供するコンポーネントの識別情報、場所および数を認識していないことが必要です。AC4J では、アプリケーションの操作を開始する前に、アプリケーションをデータ・バスに連結します。AC4J のデータ・バスは、登録済リアクションによってプッシュされ、プルされる必要があるデータ・トークンを、ルーティングしてマッチングします。さらに、一致したリアクションのスケジューリング、アクティブ化および実行を行います。

リアクションのマッチング

データ・バス・ルーティング・サブシステムは、JEMPortHandle オブジェクトによって指定された接続先であるプロセス・コンテキスト・インスタンスで、異なるタイプのデータ・トークンを使用可能にします。このプロセス・コンテキスト・インスタンスは相互作用識別子とプロセス・アクティブ化識別子で構成されます。

データ・トークンがルーティングされ、プロセス・コンテキスト・インスタンス内のデータ・バスで使用可能になると、AC4J は、データ・トークンと、そのコンテキスト・インスタンス内で使用可能なすべての登録済リアクションとのマッチングを試行します。リアクションのテンプレートで指定したデータ・トークン・タグのマッチングが試行され、一致したデータ・トークンと照合してすべての制約条件が評価され、不適切なデータ・トークンはフィルタにかけられて廃棄されます。

一部のデータ・トークンが使用可能であることは、必ずしも、登録されたリアクションが即時に一致することを意味しません。リアクションに必要なすべてのデータ・トークンが使用可能になった場合のみ、マッチングは成功します。

たとえば、processOrder プロセス内の processOrder ベース・リアクションには、checkCRED プロセスおよび checkINV プロセスの応答を待機する processOrderCallback リアクションが登録されています。checkINV プロセスが takeOrder プロセスに応答しても、processOrderCalback リアクションは一致したことにはなりません。これは、checkCRED プロセスからの応答も待機しているためです。processOrderCallback リアクションが一致するのは、checkCRED プロセスが processOrder プロセスに応答したときです。

さらに、データ・バスで使用可能なデータ・トークンは、今後登録されるリアクションとの一致も可能です。これは、1 つのプロセスが完了すると別のプロセスが使用可能になる、連続したプロセスで使用できます。

データ・トークンとリアクションのマッチングによって、0（ゼロ）以上のリアクションのアクティブ化がトリガーされます。これらのリアクションは、共有リソースの競合がない場合、パラレルで実行されます。

リアクションの起動

アクティブ EJB のリモート・インタフェースの各メソッドは、アプリケーション・ビジネス・ロジックを実装します。データ・トークンが使用可能になり、リアクションと一致すると、AC4J は、タグ上で一致したデータ・トークンのタイプ（プリミティブ・タイプまたはクラス・タイプ）が、リアクションのアクティブ EJB メソッドのタイプとも一致していることを検証します。次に、AC4J は、一致したリアクションが、一致して使用可能になったデータ・トークンのプルを許可されていることを検証します。すべての検証内容が正常な場合、AC4J は、リアクションのアクティブ化をスケジュールします。

一致したリアクションが起動すると、AC4J コンテナは、JTA トランザクションを開始し、`JEMHandle` リクエスト・オブジェクト内の主キーを使用して、リクエストされたアクティブ EJB（ステートレス Session Bean またはエンティティ EJB）をインスタンス化します（主キーが必要なのは Entity Bean の場合のみです）。次に、起動したリアクションの EJB メソッドが、リアクションの一致したデータ・トークンを使用して実行されます。

AC4J は、すべてのアクティブ EJB メソッドの終了時に、現行リアクションを自動的にコミットします。リアクションのコミットにより、JTA トランザクション終了のマークが付けられるため、共有データ・トークンへのすべての変更、および送信されたすべてのサービス・リクエストおよび応答を参照できます。リアクションがコミットされた場合、リアクションのアクティブ化には、1 回のみのセマンティクス（コードで 1 回のみ実行されるように指定）が設定されます。コミット後に障害が発生した場合、リアクションはロールバックできず、変更は永続的に存在します。コミット前またはコミット中に障害が発生した場合、コンテナは現行リアクションをロールバックします。リアクションのロールバックによって、共有データ・トークンに対する変更は元に戻され、サービス・リクエストおよび応答は受信者コンポーネントに送信されることはありません。障害が発生した場合は、データ・バスによって、事前に構成された回数のみリアクションの起動が再試行されます。最大試行回数に達した場合、リアクションは完了としてマークされ、例外完了ステータスになります。

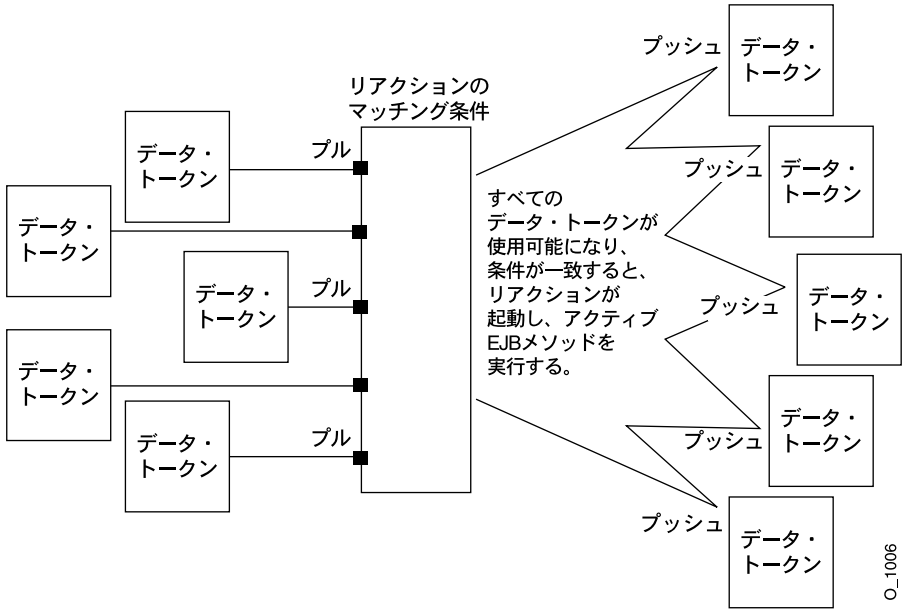
従来のデータベースではトランザクションの継続時間は短く、異常な状態が発生するとトランザクション全体が元に戻されるため、実行済のすべての作業が失われ、再発行して実行する必要がありました。通常、相互作用の継続時間は長く、多数のリアクションが含まれるため、AC4J は、例外を処理するための追加メカニズムを提供します。

正常に完了したリアクションは、AC4J によって、データ・バスに自動的に維持されます。格納されている状態（プロセスの入力変数データ、プロセスのローカル変数データ、およびデータ・フロー・コンテキスト情報）を使用すると、アプリケーションを最短時間で前回のリアクションから再開して続行できます。ノードが使用不能になった場合、実行中で正常終了していないすべてのリアクションはロールバックされます。中断されたリアクションは、AC4J によって別の OC4J インスタンスで再実行されます。

AC4J は、予期しないエラー後のアプリケーション再開に必要なアプリケーションの状態および制御フロー情報を取得、伝播および一致させるメカニズムを使用します。さらに、リアクションはデータ・ドリブンで実行されるため、制御フロー・ディスクリプタの格納またはデータ変数の格納を容易にするために、プログラム全体の状態（プログラム実行スタックなど）の揮発性コピーまたは永続的なコピーをシステムで保持する必要はありません。

図 11-1 は、すべてのデータ・トークンが使用可能になって条件が一致し、リアクションの起動によってメソッドが実行される様子を示します。このメソッドは、結果を返すことができます。結果は、AC4J のインフラストラクチャによってデータ・トークンに変換され、コール元にルーティングされます。さらにメソッドは、他のアクティブ EJB から追加サービスをリクエストして、ビジネス・タスクを完了できます。このリクエストによって作成された新規のデータ・トークンは、AC4J のデータ・バスによってプッシュされ、ルーティングされます。

図 11-1 リアクションの起動



AC4J およびデータベースのインストールと構成

AC4J アプリケーションを実行する前に、Oracle9*i* データベースを AC4J データ・バス用のリポジトリとして初期化する必要があります。次の要素をデータベースに設定します。

- AC4J の接続機能とセッション機能 : AC4J がデータ・バスで監視できるスレッド数を定義します。
- AC4J の SYSTEM 表領域。
- AC4J スーパー・ユーザー : スーパー・ユーザーを作成し、トランザクション、セキュリティおよび管理に関する特別な権限を付与する必要があります。
- AC4J データ・バス : 表および AQ トピックとキューの構成可能な数を設定する必要があります。
- 1 つ以上のクライアント・ユーザー。

データベースで AC4J データ・バスを初期化するには、次の手順を実行します。

注意 : この章で、**テキスト**で *J2EE_HOME* という表記が使用されている箇所は、J2EE のホーム・ディレクトリのフルパスを示します。これは、ファイル `oc4j.jar` が格納されているディレクトリです。UNIX の構文では `$J2EE_HOME` で、Windows の環境変数 `%J2EE_HOME%` と機能的に同じです。**コード例**では、`$J2EE_HOME` が使用されています。これらは UNIX の例で、Windows の場合は `%J2EE_HOME%` に置換し、パス名に円記号 (¥) を使用してください。

---

1. 環境変数 `$AC4J_DEMO_DIR` に AC4J デモの最上位ディレクトリ（ファイル `common.xml`、`purchaseOrder` および `README.txt` が格納されているディレクトリ）のパスを設定します。UNIX では、次のコマンドを使用します。

   ```
   setenv AC4J_DEMO_DIR path_to_AC4J_demo
   ```

2. AC4J の SQL スクリプトを、*J2EE_HOME*`/sql/ac4j-sql.jar` から新規サブディレクトリ *AC4J_DEMO_DIR*`/ac4j/sql` にアンパックします。

   ```
   cd $AC4J_DEMO_DIR/sql
   mkdir databus
   cd databus
   jar xvf $J2EE_HOME/sql/ac4j-sql.jar
   ```

3. 手順 1 でアンパックされたスクリプト `createall.sql` を実行する前に、スクリプト内の 2 つのコールに対する引数が現在のデータベース構成に対して適切であることを確認します。2 つのコールは次のとおりです。

   ```
   createjem.sql sys_user sys_pwd DB_instance tablespace
   createclient.sql sys_user sys_pwd AC4J_user AC4J_pwd tablespace
   ```

表 11-1 に、引数の意味とデフォルト値を示します。

表 11-1 createall スクリプトのコール引数

引数	意味	スクリプトのデフォルト
sys_user	データベース・システムのユーザー	sys
sys_pwd	データベース・システムのパスワード	knl_test7
DB_instance	データベース・インスタンス	通常は inst1
tablespace	AC4J 表の表領域	通常は system
AC4J_user	データベースの AC4J ユーザー	JEMCLIUSER
AC4J_pwd	データベースの AC4J パスワード	JEMCLIPASSWD

引数の値はインストール方法によって異なる場合があります。その場合は、スクリプトを編集して変更してください。

4. スクリプト createall.sql を実行します。

```
cd $AC4J_DEMO_DIR/sql/databus
sqlplus /nolog @createall.sql
```

> **注意：** 初めてスクリプトを実行すると、通常 4 つのエラー・メッセージが表示され、1 つのシノニムと 3 つのデータベース・リンクが、存在しないため削除できないことが通知されます。これらは無視して問題ありません。他のエラー・メッセージが表示されないことを確認してください。

データ・ソースの構成

J2EE_HOME/config/ の data-sources.xml ファイルで次のデータ・ソースを構成します。

```
<data-sources>

   <!-- NON-Emulated DataSources: used for JEM server -->
   <data-source
       class="com.evermind.sql.OrionCMTDataSource"
               name="nonEmulatedDS"
               location="jdbc/nonEmulatedDS"
               connection-driver="oracle.jdbc.driver.OracleDriver"
       username="jemuser"
       password="jempasswd"
       url="jdbc:oracle:thin:@host:port:sid"
```

```
        inactivity-timeout="30" >
    </data-source>

    <!-- Emulated DataSources: used for JEM client -->
    <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="jemuser"
         password="jempasswd"
         url="jdbc:oracle:thin:@host:port:sid"
         inactivity-timeout="30" >
    </data-source>

</data-sources>
```

このコードで、*host*、*port* および *sid* は、ホスト名、ポート番号および AC4J データ・バスがインストールされている Oracle データベース・インスタンスのデータベース SID で置換する必要があります。両方のデータ・ソースに対してこのように置換してください。

jemuser はスーパー・ユーザーのユーザー名で、jemcliuser はデフォルトのクライアント・ユーザー名です。これらは、SQL スクリプト createall.sql で作成されます。

AC4J の例

AC4J は、長期間にわたって相互に作用する複合アプリケーション用に設計されています。この項では、簡単な発注の例を使用して AC4J の使用方法を説明します。例のコードを簡素化するために、エラー処理やインポートの文は含まれていません。例は、OC4J 9.0.4 の EJB 機能のデモの 1 つとしてパッケージされています。デモは、OTN-J の OC4J サンプル・コードのサイトからダウンロードできます。

http://otn.oracle.com/tech/java/oc4j/demos/904

例の実行

例を実行するには、次の手順を実行します。

1. 環境変数 $AC4J_DEMO_DIR に AC4J デモの最上位ディレクトリ（ファイル common.xml、purchaseOrder および README.txt が格納されているディレクトリ）のパスを設定します。UNIX では、次のコマンドを使用します。

   ```
   setenv AC4J_DEMO_DIR path_to_AC4J_demo
   ```

2. AC4J データ・バスを初期化します。

   次のコマンドで createTables.sql スクリプトを実行することによって、例の実行に必要なデータベース表を作成し、データを格納します。

   ```
   cd $AC4J_DEMO_DIR/sql/
   sqlplus /nolog @createTables.sql
   ```

3. データベース構成ファイルを編集します。

   デモで使用するデータ・ソースは、ファイル *J2EE_HOME*/config/data-sources.xml で設定する必要があります。

4. 例を作成します。

   3 つのサービスをすべて作成する必要があります。次のように、それぞれのディレクトリでデフォルトの ant ルールを実行します。

   ```
   cd $AC4J_DEMO_DIR
   ant
   ```

   注意： ant ユーティリティはオープン・ソースで、（アプリケーション・サーバー間およびオペレーティング・システム間で）移植可能です。したがって、このユーティリティは Java ベースのアプリケーションの理想です。ant と付属のドキュメントは次のサイトから入手できます。

   http://jakarta.apache.org/ant/

   OC4J に付属のサンプル・アプリケーションの一部は、ant を使用するように設定されています。モデルについては、付属の build.xml ファイルを参照してください。

   同様の方法でサービスを消去することもできます。

   ```
   cd $AC4J_DEMO_DIR
   ant clean
   ```

5. OC4J インスタンスを起動します。次に例を示します。

   ```
   cd $J2EE_HOME
   java -Doracle.aurora.jem.aq.close.interval=2 -jar oc4j.jar
   ```

6. 3 つのサービスをデプロイします。

   Oracle Application Server で EJB をデプロイする方法については、『Oracle Application Server Containers for J2EE User's Guide』を、スタンドアロン実装については、『Oracle Application Server Containers for J2EE Standalone User's Guide』を参照してください。次の例で、スタンドアロンの使用方法を示します。

```
cd $AC4J_DEMO_DIR
java -jar $J2EE_HOME/admin.jar ormi://localhost admin welcome -deploy \
 -file src/ejb/purchaseOrderService-ejb/lib/purchaseOrderService.ear \
 -deploymentName purchaseOrderService
java -jar $J2EE_HOME/admin.jar ormi://localhost admin welcome -deploy \
 -file src/ejb/inventoryService-ejb/lib/inventoryService.ear \
 -deploymentName inventoryService
java -jar $J2EE_HOME/admin.jar ormi://localhost admin welcome -deploy \
 -file src/ejb/creditService-ejb/lib/creditService.ear \
 -deploymentName creditService
```

デプロイされる各サービスごとに 1 つずつ、3 つのメッセージがアプリケーション・サーバーから表示されます。それぞれ JEM Server started というメッセージが表示されます。

7. クライアントを実行します。

purchaseOrderService サブディレクトリの ant ルール reqpo を起動して、発注リクエストを送信します。次に例を示します。

```
cd $AC4J_DEMO_DIR
ant reqpo
```

注意： 与信サービスと在庫サービスにはクライアントがありません。ユーザーは PurchaseOrder サービスのみと直接対話し、このサービスが他の 2 つのサービスと相互に作用します。

アプリケーション・サーバーの出力

アプリケーション・サーバーから次のような出力が生成されます。

```
----------- sample of expected app server output from ant reqpo ---------------
=======>PurchaseorderServiceBean.ejbCreate(): begin/end
=======>PurchaseOrderServiceBean.takeOrder(): begin
         : clientName = scott
         : creditCardNumber = 1111-3333-4444-8888
         : (productName, quantity) = pen, 3
         : (productName, quantity) = pencil, 1
=======>PurchaseOrderServiceBean.createPO(): begin
=======>PurchaseOrderBean.ejbCreate(): begin
=======>PurchaseOrderBean.createPONumber(): begin
         : poNum=1
=======>PurchaseOrderBean.createPONumber(): end
=======>PurchaseOrderBean.evaluateTotalCost(): begin
=======>PurchaseOrderBean.evaluateTotalCost(): end---Total Cost = 0.75
=======>PurchaseOrderBean.ejbCreate(): end---poNumber= 1
=======>PurchaseOrderBean.ejbPostCreate(): begin
```

```
=======>LineItemBean: ejbCreate:  lineItemName = pen  quantity = 3
=======>LineItemBean: ejbCreate:  lineItemName = pencil  quantity = 1
=======>PurchaseOrderBean.ejbPostCreate(): end---getLineItems().size= 2
       createPO()---poNumber= 1
=======>PurchaseOrderServiceBean.createPO(): end
         : poNumber= 1
=======>PurchaseOrderServiceBean.startProcessingPOrder(): begin
=======>PurchaseOrderServiceBean.startProcessingPOrder(): end
=======>PurchaseOrderServiceBean.takeOrder(): end
=======>PurchaseOrderServiceBean.processOrder(): begin---poNumber= 1
=======>PurchaseOrderServiceBean.callCreditService(): begin
=======>PurchaseOrderServiceBean.callCreditService(): end
=======>PurchaseOrderServiceBean.callInventoryService(): begin
=======>PurchaseOrderServiceBean.callInventoryService(): end
=======>PurchaseOrderServiceBean.registerAsyncRespHandler(): begin
=======>PurchaseOrderServiceBean.registerAsyncRespHandler(): end
=======>PurchaseOrderServiceBean.processOrder(): end---status= STATUS_INPROCESS
=======>CreditServiceBean: ejbCreate: begin/end
=======>CreditServiceBean: checkCRED: begin
         : clientName = SCOTT
         : creditCardNumber = 1111-3333-4444-8888
         : amount = 0.75
=======>InventoryServiceBean: ejbCreate: begin/end
=======>InventoryServiceBean: checkINV: begin
         : (product, quantity) = pen, 3
         : (product, quantity) = pencil, 1
         : CreditorRemote found
         : creditorName = SCOTT
         : availableCredit = 5000.0
=======>CreditServiceBean: checkCRED: end--returning  CREDIT APPROVED
         : BEFORE: availableUnits[0] = 700
         : AFTER: availableUnits[0] = 697
         : BEFORE: availableUnits[1] = 600
         : AFTER: availableUnits[1] = 599
         : Returning:
         : invCheck[0]=true
         : invCheck[1]=true
=======>InventoryServiceBean: checkINV: end
=======>PurchaseOrderServiceBean.processOrderCallback(): begin.credInfo=CREDIT
APPROVED
=======>PurchaseOrderServiceBean.callBackMethod(): poNumber= 1
         : invInfo[0] = true
         : invInfo[1] = true
=======>PurchaseOrderServiceBean.callBackMethod(): end---credInfo=CREDIT APPROVED
poNumber=1  currentStatus= STATUS_SHIPPED
=======>PurchaseOrderServiceBean.getStatus(): begin---poNumber= 1
=======>PurchaseOrderServiceBean.getStatus(): end---poNumber= 1
```

```
status=STATUS_SHIPPED
-------- (end) sample of expected app server output from ant reqpo ------------
```

クライアントの出力

クライアントから次のような出力が生成されます。

```
---------------  sample of expected client output from ant reqpo --------------
reqpo:
     [java] Getting AC4J Connection and Session...


     [java] sendRequest: begin
     [java]   customer name = IID = scott
     [java]   credit card number = 1111-3333-4444-8888
     [java]   item names = pen,pencil
     [java]   quantities = 3,1


     [java] takeOrder request made. Process Context is:
     [java]   Interaction Identifier (IID) = scott
     [java]   Activation  Identifier (AID) = AE3D71D56E835817E0340003BA137479


     [java] Execute the following to receive PO response and track status:
     [java]   ant -DAID=AE3D71D56E835817E0340003BA137479 resppo
------------- (end) sample of expected client output from ant reqpo ----------
```

応答の収集

発注リクエストの応答は、2 番目の ant ターゲット resppo を実行することによって収集されます。リクエストのアクティビティ ID（AID）をプロパティとして指定する必要があります（ant -DAID=<AID> resppo）。わかりやすいように、reqpo は、そのリクエストの必須 ant コマンドラインを出力の最終行に出力します。すばやく入力できるように、切取りと貼付けを使用して、テキストをコマンドラインにコピーします。

次は、前述の AID を使用する例です。

```
cd $AC4J_DEMO_DIR
ant -DAID=AE3D71D56E835817E0340003BA137479 resppo
```

プログラムは、STATUS_SHIPPED ステータスを受信すると自動的に終了します。

正確な出力は、発注リクエストの作成後、ant resppo が実行されるまでの時間によって異なります。

ant resppo の出力

ant の resppo クライアントを実行すると、次のような出力が生成されます。

```
--------------- sample of expected output from ant resppo -------------------
resppo:
     [java] Getting AC4J Connection and Session...

     [java] receiveResponse: begin
     [java] receiveResponse: receiving async response
     [java] receiveResponse: response received
     [java] receiveResponse: Purchase Order number = 1
     [java] receiveResponse: polling for PO status...
     [java]   status = STATUS_SHIPPED

     [java] receiveResponse: Status = SHIPPED. Done.
------------ (end) sample of expected output from ant resppo ------------------
```

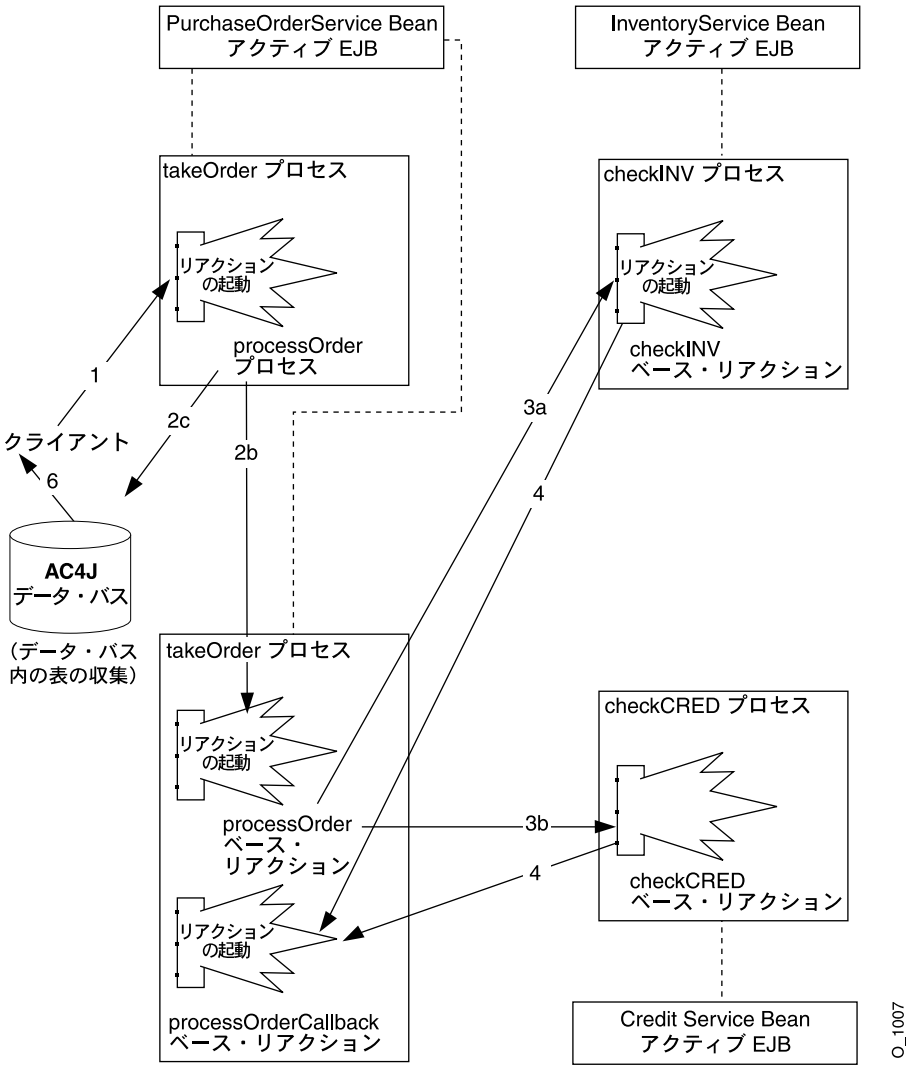
クライアントの再実行

クライアントを再実行（および別の発注を発行）するには、11-15 ページの「例の実行」の手順 6 の説明に従って、ant reqpo を再実行します。

例の説明

この項では、発注の例の実行で発生する手順の概要を説明します。図 11-2 は、この後の説明の手順番号に対応する番号を付けて手順を示しています。

図 11-2 発注の例の手順

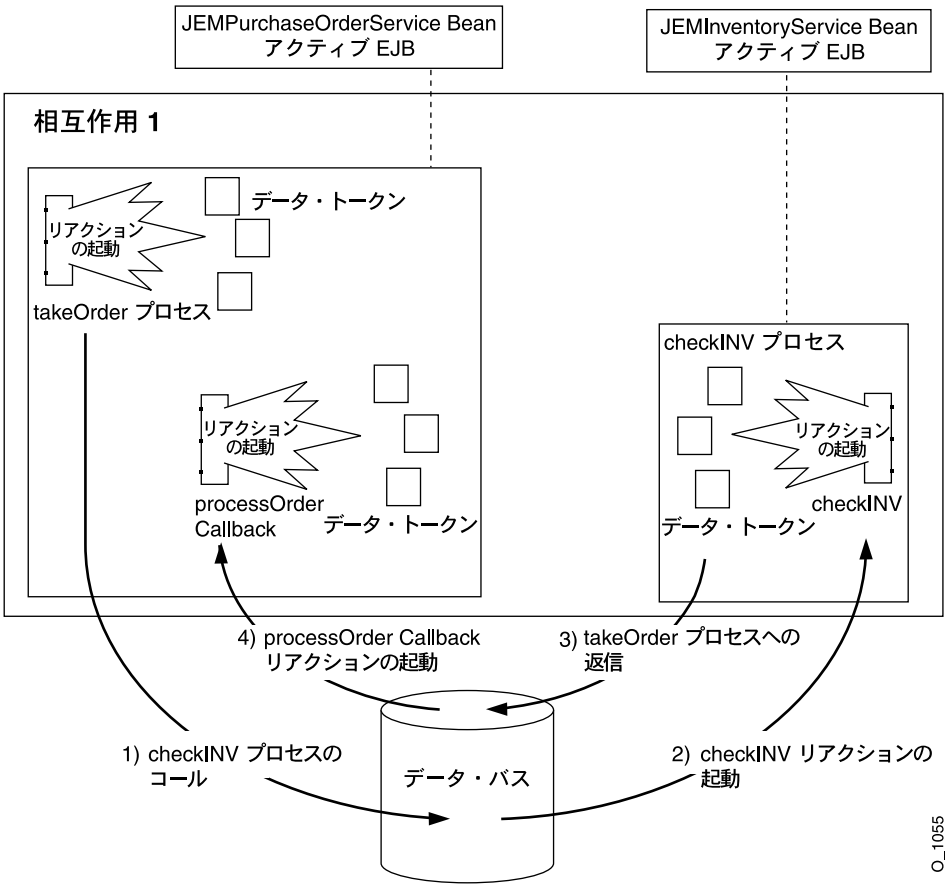


11-23 ページの「手順の概要」の後に、各手順別の項が続きます。各手順の項で、その手順のコードについて詳細に説明します。

図 11-2 には、takeOrder プロセスからクライアントへの発注番号の戻り（手順 2 ～ 6）のみが、AC4J データ・バスを通じて行われる様子が示されています。この図は簡略化されており、様々なプロセスとリアクション間の制御フローが強調されています。

実際には、11-9 ページの「AC4J データ・トークン」で説明されているように、（応答を含む）すべての AC4J コールが、AC4J データ・バスによって実行されます。図 11-3 は、このことを例の中の 2 つの手順について説明します。2 つの手順は、processOrder ベース・リアクションから checkINV プロセスへのコール（手順 3a）、および checkINV プロセスの応答の takeOrder プロセスへの返信（手順 4）です。

図 11-3 データ・バスによって実行されるすべての AC4J コール



手順の概要

発注の例の実行で発生する手順は、次のとおりです。

1. 手順 1: クライアントによる発注サービス・アクティブ EJB への非同期リクエストの送信

   リクエスト（takeOrder）がデータ・バスを通じて送信され、新規 AC4J プロセスが開始されます。

2. 手順 2: 発注サービスのアクティブ EJB によるクライアントの takeOrder リクエストの処理

   takeOrder メソッドで次のことが実行され、新規発注の処理が開始されます。

   a. 新規発注を表す発注の Entity Bean が作成されます。この Bean の作成によって、適切な製品および明細品目の Bean が作成されます。これには、標準的な EJB Entity Bean が使用されます。AC4J は関係しません。

   b. 非同期リクエストが processOrder プロセスに送信されます。これは同じ Bean（PurchaseOrderService Bean）のメソッドですが、新規 AC4J プロセスのベース・リアクションを形成します。

   c. 発注の Entity Bean の作成時に割り当てられた発注番号が返されます。

   別のリアクションを開始すると、takeOrder リアクションを終了でき、クライアントが発注ステータスのポーリングを開始できるように、この発注に割り当てられた発注番号ができるだけ早くクライアントに返されます。

3. 手順 3: 発注サービスのアクティブ EJB による processOrder リクエストの処理

   processOrder ベース・リアクションは、新規の発注を開始します。次の処理を実行します。

   a. 非同期リクエストをアクティブ EJB の InventoryService の checkINV プロセスに送信し、品目の在庫があることを確認します。

   b. 非同期リクエストをアクティブ EJB の CreditService の checkCRED プロセスに送信し、顧客の与信が十分であることを確認します。

   c. processOrderCallback リアクションを現行プロセスに登録し、前述の 2 つのリクエストから結果を受信します。

4. 手順 4: 在庫サービスおよび与信サービスのアクティブ EJB によるリクエストの処理と非同期応答

   checkINV プロセスと checkCRED プロセスの両方が応答を返します。

5. 手順 5: processOrderCallback リアクションによる非同期応答の処理

   processOrder プロセス内の processOrderCallback リアクションは、checkINV プロセスおよび checkCRED プロセスにより提供された情報に対応して動作します。処理が完了すると、発注の Entity Bean のステータスが更新されます。クライアントに確認の通知は送信されません。

6. 手順 6: クライアントによる発注番号の非同期受信と発注ステータスのポーリング

   手順 2 で返される発注番号は、AC4J のデータ・バスを通じて返され、クライアントはその後いつでも非同期で受信できます。発注番号を受信した後、クライアントは標準的な EJB コールを使用して発注のステータスをポーリングします。

手順 1: クライアントによる発注サービス・アクティブ EJB への非同期リクエストの送信

例 11-1 のサンプル・コードは、クライアントがアクティブ EJB の `PurchaseOrderService` に非同期リクエストを送信するときの手順を示します。

例 11-1 クライアントによる発注サービス・アクティブ EJB への非同期リクエストの送信

```
static final String DATA_SOURCE_NAME  = "java:comp/env/jdbc/OracleDS";
static final String BEAN_NAME = "java:comp/env/ejb/PurchaseOrderService";
static final String ACTIVE_EJB_NAME = "JEMPurchaseOrderService";
static final String METHOD_NAME = "takeOrder";
static final String DB_USER    = "JEMUSER";
static final String DB_PASSWD  = "JEMPASSWD";

public static void main(String[] args)
{
    // 1.0. Create a JNDI Context
    Context context = new InitialContext();

    // 1.1. Look up the DataSource where AC4J data bus resides
          DataSource client_ds = (DataSource) context.lookup(DATA_SOURCE_NAME);

    // 1.2. Obtain a JDBC connection to the DataSource
    OracleConnection conn =
        (OracleConnection)client_ds.getConnection(DB_USER, DB_PASSWD);

    // 1.3. Create an AC4J connection, using the JDBC Connection
    ac4j_conn = new JEMConnection(conn);

    // 1.4. Create an AC4J session on the data bus
    ac4j_sess = new JEMSession(ac4j_conn);

    // 1.5. Look up the handle of the PurchaseOrderService Active EJB
    activeEJBHandle = (JEMHandle)context.lookup(ACTIVE_EJB_NAME);

    // 1.6. Prepare input parameters for asynchronous call
```

```
    // 1.6a. Make String and int arrays from itemNames and quantities args
    String[] itemNames  = getStringArrayFromInput(args[3]);
    int[]    quantities = getIntArrayFromInput(args[4]);

    // 1.6b. Make a Class array of input parameter types
    Class[] inputClassTypes = new Class[] { String.class,
            String.class,
            itemNames.getClass(),
            quantities.getClass() };

    // 1.6c. Make an Object array of input parameter values
    Object[] inputParams = new  Object[] { (Object) new  String(args[1]),
            (Object) args[2],
            (Object) itemNames,
            (Object) quantities };

    // 1.7. Make the asynchronous call
    // IID = customer name (args[1]), AID = null = auto-assigned
    JEMEmitToken request = ac4j_sess.call(args[1],
            null,
            activeEJBHandle,
            METHOD_NAME,
            inputClassTypes,
            inputParams,
            null, 0, 0);

    // 1.8. Commit the transaction
    ((OracleConnection)ac4j_sess.getJEMConnection().getConnection()).commit();

    // 1.9. Get the AID auto-assigned to the request just made
    //    Also, Confirm IID (will be value set above - just showing the API)
    //    IID + AID = 'Process Context' of the request
    JEMPortHandle portHandle = request.getPortHandle();
    String iid = portHandle.getIid();
    String aid = portHandle.getAid();
    System.out.println("takeOrder request made. Process Context is:");
    System.out.println("  Interaction Identifier (IID) = " + iid);
    System.out.println("  Activation  Identifier (AID) = " + aid);

    // 1.10 Close AC4J session and connection and JDBC connection
    ac4j_sess.close();
    ac4j_conn.close();
    conn.close();
}
```

クライアントは、AC4J サーバー外に存在し、アクティブ EJB から AC4J データ・バスを通じてサービスをリクエストします。AC4J データ・バスはパイプとして動作し、クライアントとすべてのリアクションの間の非同期通信を制御します。AC4J サーバー外に存在するすべてのクライアントは、最初に AC4J データ・バスに接続し、相互作用のための新規セッションを作成する必要があります。

AC4J データ・バスとの接続を取得し、データ・バス内に AC4J セッションを作成した後は、同一または他の AC4J インスタンス内のアクティブ EJB に、非同期メッセージを送信できます。AC4J データ・バスは、非同期メッセージを調整し、トランザクション内のすべてのアクティブ EJB に対するトランザクション・マネージャとして動作します。

次の説明は、例 11-1 の番号付けに対応しています。

1.0 ～ 1.4 AC4J 接続の取得

AC4J 接続は JDBC 接続上に存在するため、手順 1.0 ～ 1.4 で、その AC4J 接続を取得します。

1.1 AC4J パイプとして動作するデータベースに定義されている DataSource を取得します。

`data-sources.xml` ファイルで、使用する `DataSource` をエミュレートされたデータ・ソースとして定義します。詳細は、11-14 ページの「データ・ソースの構成」を参照してください。

```
Context context = new InitialContext();
DataSource client_ds = (DataSource)
               context.lookup(DATA_SOURCE_NAME);
```

1.2 DataSource オブジェクトから JDBC 接続を取得します。

```
OracleConnection conn =
   (OracleConnection)client_ds.getConnection(DB_USER, DB_PASSWD);
```

1.3 JDBC 接続オブジェクトから AC4J 接続を作成します。

```
ac4j_conn = new JEMConnection(conn);
```

1.4 指定されたデータ・バスに AC4J セッションを作成します。

データ・バスの名前を指定して、データベースへの AC4J 接続を使用し、指定した Oracle データベースのデータ・バス内にセッションを作成します。

```
ac4j_sess = new JEMSession(ac4j_conn);
```

1.5 ～ 1.11 非同期リクエストの送信

AC4J データ・バスに AC4J セッションを作成すると、クライアントは非同期メッセージをアクティブ EJB に送信できます。クライアントは、アクティブ EJB ハンドル、プロセス・ハンドル、およびベース・リアクションに対する必須の入力パラメータをすべて提供する必要があります。手順 1.5 ～ 1.11 は、AC4J リクエストを完了するためにクライアントが実行する必要があるコールの詳細を説明します。

1.5 アクティブ EJB ハンドルを取得します。

同期 EJB 環境では、リモート EJB ハンドルを使用して起動します。AC4J 非同期環境では、アクティブ EJB を識別する JEMHandle クラス・タイプのハンドルを提供する必要があります。アクティブ EJB ハンドルは、orion-ejb-jar.xml ファイルで定義された jem-name を参照して取得できます（11-40 ページの「AC4J でのアクティブ EJB のデプロイメント」を参照）。

```
// 1.5. Look up the handle of the PurchaseOrderService Active EJB
activeEJBHandle = (JEMHandle)context.lookup(ACTIVE_EJB_NAME);
```

1.6 非同期コール用の入力パラメータを準備します。

クライアントは、各パラメータのタイプを識別するクラス・タイプの配列と、値を提供するオブジェクトの配列を準備します。各配列には 4 つのパラメータがあります。2 つの文字列（顧客名とクレジット・カード番号）、文字列の配列（品目）および整数の配列（数量）です。

```
Class[] inputClassTypes = new Class[] { String.class,
   String.class,
   itemNames.getClass(),
   quantities.getClass() };

Object[] inputParams = new  Object[] { (Object) new  String(args[1]),
   (Object) args[2],
   (Object) itemNames,
   (Object) quantities };
```

1.7 非同期コールを実行します。

JEMSession::call には、EJB の相互作用識別子、メソッドをインスタンス化するプロセスを識別するためのプロセス・アクティブ化識別子、およびアクティブ EJB の JEMHandle が含まれます。

相互作用識別子とプロセス・アクティブ化識別子（両方でプロセス・コンテキストを形成します）はオプションで、省略したり NULL にできます。その場合、これらの識別子はシステムで自動的に作成されます。データ・バスは、プロセスのコンテキストを識別し、データ・トークンを対象のプロセスにルーティングします。したがって、すべての EJB コールは、データ・バスで調整され、非同期で起動されます。

```
// IID = customer name (args[1]), AID = null = auto-assigned
JEMEmitToken request = ac4j_sess.call(args[1],
   null,
   activeEJBHandle,
   METHOD_NAME,
   inputClassTypes,
   inputParams,
   null, 0, 0);
```

1.8 トランザクションをコミットします。

クライアントは、AC4J データ・バスに対して変更をコミットする必要があります。トランザクションがコミットされない場合は、リクエストが失われ、AC4J データ・バスでは参照できません。リクエストを AC4J データ・バスで参照可能にするには、JDBC コミットを次のように実行します。

```
((OracleConnection)ac4j_sess.getJEMConnection().getConnection()).commit();
```

1.9 作成されたリクエストに自動的に割り当てられた AID を取得します。

この場合、IID は与えられていますが、AID は AC4J で自動的に割り当てられるように NULL のままです。割り当てられる AID は、ポート・ハンドルから読み取られ、後の手順 6 の応答で非同期で収集されるように保持される必要があります。

```
JEMPortHandle portHandle = request.getPortHandle();
String aid = portHandle.getAid();
```

1.10 セッションと接続を閉じます。

最後に、クライアントは AC4J セッションと接続、および JDBC 接続を閉じる必要があります。これは、クライアントは AC4J コンテナ内に存在していないために必要な処理です。AC4J コンテナ内で実行されるアプリケーション（この例のアクティブ EJB など）の場合、セッションと接続はコンテナによって自動的に閉じられます。

```
ac4j_sess.close();
ac4j_conn.close();
conn.close();
```

手順 2: 発注サービスのアクティブ EJB によるクライアントの takeOrder リクエストの処理

例 11-2 のサンプル・コードに、クライアントの `takeOrder` リクエストの処理で発注サービスのアクティブ EJB が実行する手順を示します。

クライアントがリクエストをコミットした後、AC4J データ・バスは、そのクライアントが提供したデータ・トークンと、リクエストされたリアクションのデータ・トークンを一致させ、アクティブ EJB の `PurchaseOrderBean` のインスタンス化、および `takeOrder` プロセスのアクティブ化を内部的にスケジュールします。`takeOrder` プロセスは、新規の発注を開始する `takeOrder` ベース・リアクションを開始します。

このリアクション `takeOrder` は、次の手順を実行することでクライアントのリクエストを処理します。

1. 新規発注を表す発注の Entity Bean が作成されます。この Bean の作成によって、適切な製品および明細品目の Bean が作成されます。これには、標準的な EJB Entity Bean が使用されます。AC4J は関係しません。

2. 非同期リクエストが processOrder プロセスに送信されます。これは同じ Bean（PurchaseOrderService Bean）のメソッドですが、新規 AC4J プロセスのベース・リアクションを形成します。
3. 発注の Entity Bean の作成時に割り当てられた発注番号が返されます。

別のリアクションを開始すると、takeOrder リアクションを終了でき、クライアントが発注ステータスのポーリングを開始できるように、この発注に割り当てられた発注番号ができるだけ早くクライアントに返されます。

例 11-2 に、これらの手順を実行するコードを示します。

例 11-2 発注サービスのアクティブ EJB によるクライアントの takeOrder リクエストの処理

```
public int takeOrder(String clientName, String creditCardNumber,
                     String[] productNames, int[] quantities)
      throws RemoteException, TestException
{
   int poNumber = 0;

   // 2.1. Create Purchase Order Entity Bean, and get PO Number in return
   // (Regular EJB Entity Beans code. Not shown)
   poNumber = createPO(clientName, creditCardNumber, productNames, quantities);

   // 2.2. Get the current AC4J reaction
   JEMReaction currentAC4JReaction = (JEMReaction)JEMReaction.getReaction();

   // 2.3. Make an asynchronous call to the processOrder reaction on this
   //    Bean, so can return PO Number to client now

   // 2.3a. Look up the AC4J handle for this Bean (PurchaseOrderService)
   Context context = new InitialContext();
   JEMHandle ac4jPOSBeanHandle =
            (JEMHandle)context.lookup(jemPurchaseOrderServiceBeanName);

   // 2.3b. Prepare input parameter Types for processOrder method
   Class[] inputClassTypes = new Class[] { Integer.TYPE,
                                           productNames.getClass(),
                                           quantities.getClass() };

   // 2.3c. Prepare input parameter Values for processOrder method
   Object[] inputParams = new Object[] { (Object) (new Integer(poNumber)),
                                         (Object) productNames,
                                         (Object) quantities };
```

```
    // 2.3d. Make asynchronous call to processOrder
    JEMEmitToken emitToken = null;
    emitToken =  currentAC4JReaction.call(null, null, ac4jPOSBeanHandle,
                                                "processOrder", inputClassTypes,
                                                null, inputParams, null, null,
                                                null, 0,  0);

    // 2.4 Return the PO Number of the new Purchase Order
    return poNumber;

}
```

AC4J データ・バスは、AC4J サーバーで、アクティブ EJB の JEMPurchaseOrderBean（クライアントが提供する JEMHandle に対応しています）をインスタンス化します。takeOrder プロセスは、takeOrder ベース・リアクションを開始します。takeOrder メソッドの実装によって、例に示した手順が実行されます。次の説明は、例 11-2 の番号付けに対応しています。

2.1 発注の Entity Bean を作成し、返される発注番号を取得します。

この手順には標準的な EJB Entity Bean のプログラミングが含まれますが、これについては示されていません。

2.2 現行の AC4J リアクションを取得します。

現行リアクションの takeOrder は、AC4J サーバーで実行中です。アプリケーション・コードで、次のように、現行リアクションを取得できます。

```
JEMReaction currentAC4JReaction = (JEMReaction)JEMReaction.getReaction();
```

これは、手順 2.4 で実行されるコールを現行リクエストのコンテキストで（つまり、同じ AC4J プロセスの別のリアクションとして）実行するために行われます。

現行のプロセス・コンテキストの相互作用 ID（IID）とアクティブ ID（AID）は、次のように現行リアクションから取得できます。

```
String iid = currentAC4JReaction.getIid();
String aid = currentAC4JReaction.getAid();
```

この結果、特に、新規コールを同じ相互作用のコンテキスト内で実行できますが、別のアクティビティ ID を使用できるため、同じ相互作用内に新規プロセスが作成されます。

2.3 この Bean の processOrder リアクションに対する非同期コールを実行します。

非同期コールの実行手順は、手順 1.5 ～ 1.7 のクライアントによる処理と同じです。トランザクションは、アプリケーションによってコミットされる必要はありません。メソッドが返されたときに、AC4J が自動的にコミットします。

別のリアクションを開始すると、takeOrder リアクションを終了でき、クライアントが発注ステータスのポーリングを開始できるように、この発注に割り当てられた発注番号ができるだけ早くクライアントに返されます。

2.4 新規発注の発注番号を返信します。

アプリケーションは、次のように通常の同期方式を使用して値を返します。

```
return poNumber;
```

この値がクライアントにどのように返されるかは、クライアントによるコールの方式によって異なります。

この場合、クライアントは AC4J データ・バスを通じてメソッドを非同期でコールしました。したがって、AC4J が戻り値を取得し、データ・バスに格納される AC4J データ・トークンにパッケージします。クライアントは、そのデータ・トークンをコンシュームするリアクションを登録して、値を非同期で受信します。この例では、処理は手順 6 で実行されます（手順 6 は、手順 2 の完了後、つまり応答データ・トークンがデータ・バスに格納された後はいつでも実行できます）。

クライアントが同じアクティブ EJB メソッドを標準的な EJB コールによって同期式でコールした場合、値は通常の方法でクライアントに同期式で返されます。これは、AC4J 対応のアクティブ EJB は、標準的な EJB としてもコールできるためです。

手順 3: 発注サービスのアクティブ EJB による processOrder リクエストの処理

processOrder ベース・リアクションは、新規の発注を開始します。次の処理を実行します。

1. 非同期リクエストをアクティブ EJB の InventoryService の checkINV プロセスに送信し、品目の在庫があることを確認します。
2. 非同期リクエストをアクティブ EJB の CreditService の checkCRED プロセスに送信し、顧客の与信が十分であることを確認します。
3. processOrderCallback リアクションを現行プロセスに登録し、前述の 2 つのリクエストから結果を受信します。

例 11-3 に、これらの手順を実行するコードを示します。

例 11-3 発注サービスのアクティブ EJB による processOrder リクエストの処理

```
public void processOrder(int poNumber, String[] productNames,
                         int[] quantities)
      throws RemoteException, TestException
{
   PurchaseOrderRemote poRemote = null;

   // 3.1. Get the current AC4J reaction
   JEMReaction currentAC4JReaction = (JEMReaction)JEMReaction.getReaction();

   // 3.2. Look up purchase order Entity Bean, so can pass reference to
   //    Credit and Inventory Services
   poRemote = lookupPOBean(new Integer(poNumber));

   // 3.3. Make an asynchronous call to the Credit Service
   JEMEmitToken creditToken = callCreditService(currentAC4JReaction, poRemote);

   // 3.4. Make an asynchronous call to the Inventory Service
   JEMEmitToken inventoryToken = callInventoryService(currentAC4JReaction,
                                                      productNames,
                                                      quantities);

   // 3.5. Register processOrderCallback Reaction
   registerCallback(currentAC4JReaction,
   new JEMEmitToken[] {inventoryToken, creditToken});

   // 3.6. Update Purchase Order status to INPROCESS
   poRemote.setStatus(Status.getString(Status.STATUS_INPROCESS));
   }
```

次の説明は、例 11-3 の番号付けに対応しています。

3.1 現行の AC4J リアクションを取得します。

この処理の目的は、3.6 で新規（コールバック）リアクションを登録することです。コールバック・リアクションは、プロセスに関連付けられているデータにアクセスできるように、現行リアクションと同じプロセスに登録されます。詳細は、手順 5 を参照してください。

```
JEMReaction currentAC4JReaction = (JEMReaction)JEMReaction.getReaction();
```

3.2 発注の Entity Bean をルックアップします。

この処理は、発注の Entity Bean の参照を与信サービスへのリクエストで渡すことができ、Bean を直接問い合せることができるようにするために行われます。

```
poRemote = lookupPOBean(new Integer(poNumber));
```

3.3 与信サービスの非同期コールを実行します。

非同期コールの実行手順は、手順 1.5 ～ 1.7 のクライアントによる処理と同じです。トランザクションは、アプリケーションによってコミットされる必要はありません。メソッドが返されたときに、AC4J が自動的にコミットします。与信サービスのアクティブ EJB の checkCRED メソッドがコールされ、新規 checkCRED プロセスが起動します。

```
JEMEmitToken creditToken = callCreditService(currentAC4JReaction, poRemote);
```

3.4 在庫サービスの非同期コールを実行します。

この手順は 3.3 と類似しています。異なる点は、発注の Entity Bean 参照がサービスに渡されないことのみです。情報は、製品名と数量で十分です。この相違は AC4J の問題ではなく、この例の実装方法の問題です。

与信サービスのアクティブ EJB の checkINV メソッドがコールされ、新規 checkINV プロセスが起動します。

```
JEMEmitToken inventoryToken = callInventoryService(currentAC4JReaction,
                                                   productNames,
                                                   quantities);
```

3.5 processOrderCallback リアクションを登録します。

現行プロセスに新規リアクションを登録します。

```
registerCallback(currentAC4JReaction,
    new JEMEmitToken[] {inventoryToken, creditToken});
```

リアクションは、手順 3.3 および 3.4 で作成した与信サービスと在庫サービスに非同期コールによって返されるトークンの関連を指定します。新規リアクションは、両方のコールからの応答トークンが AC4J データ・バスに格納されたとき（両方のサービスがコールから結果を返したとき）に実行されるようにスケジュールされます。

コールバック・リアクションは、プロセスに関連付けられているデータにアクセスできるように、現行リアクションと同じプロセスに登録されます。詳細は、11-34 ページの「手順 5: processOrderCallback リアクションによる非同期応答の処理」を参照してください。

3.6 発注ステータスを INPROCESS に更新します。

この状態は、ポーリングして発注のステータスをチェックするクライアントに返すことができるように維持されます。詳細は、11-37 ページの「手順 6: クライアントによる発注番号の非同期受信と発注ステータスのポーリング」を参照してください。

```
poRemote.setStatus(Status.getString(Status.STATUS_INPROCESS));
```

takeOrder ベース・リアクションは、AC4J インフラストラクチャが、2 つのアクティブ EJB および登録済リアクションへのコールを含むトランザクションをコミットした後にのみ完了します。

手順 4: 在庫サービスおよび与信サービスのアクティブ EJB によるリクエストの処理と非同期応答

checkINV プロセスおよび checkCRED プロセスは、他の EJB から起動されたかのように、AC4J データ・バスからリクエストを受信します。アクティブ EJB の JEMInventoryBean および JEMCreditBean は、インスタンス化されます。ベース・リアクションの checkINV および checkCRED は、AC4J データ・バスからデータ・トークンを受信すると起動します。このデータ・トークンは、processOrder リアクションから起動されたものです。両方のベース・リアクションは、リクエストを受信し、タスクを実行して返されます。返された値は、AC4J データ・バスによって、登録済リアクションの processOrderCallback に転送されます。

例 11-4 のサンプル・コードは、checkINV メソッドを示します。checkCRED メソッドも同様です。

例 11-4 checkINV でのリクエスト処理

```
public boolean[] checkINV(String[] productNames, int[] quantities)
            throws RemoteException, TestException
{
    boolean[] invCheck = new boolean[productNames.length];

    // The business logic is not shown
    ...

    return invCheck;
}
```

アプリケーションは、次のように通常の同期方式を使用して値を返します。

```
return invCheck;
```

ただし、値はデータ・バスに格納されたデータ・トークンによって非同期で返されます。詳細は、手順 2.4 を参照してください。

手順 5: processOrderCallback リアクションによる非同期応答の処理

checkINV と checkCRED の両プロセスは、AC4J データ・バスを通じて processOrderCallback リアクションに応答を返します。AC4J データ・バスは、返されるデータ・トークンに有効な processOrder プロセス・コンテキストがあり、processOrderCallback リアクションの入力パラメータ・タイプと一致していることを確認します。両方のパラメータが返されると、processOrcerCallback リアクションが起動し、アクティブ EJB の JEMPurchaseOrderBean の processOrderCallback メソッドを実行します。

processOrderCallback リアクションは、checkINV および checkCRED プロセスで提供される情報に対応して動作し、それに従って発注の Entity Bean の状態を更新します。11-37 ページの「手順 6: クライアントによる発注番号の非同期受信と発注ステータスのポーリング」で説明されているように、クライアントに確認の通知は送信されないことに注意してください。クライアントは発注のステータスをポーリングします。

例 11-5 のサンプル・コードは、processOrderCallback メソッドを示します。

例 11-5 processOrderCallback リアクションによる非同期応答の処理

```
public void processOrderCallback(boolean[] invInfo, String credInfo)
      throws RemoteException, TestException
{
    PurchaseOrderRemote poRemote = null;
    Integer poNumber = null;

    //    This Reaction (processOrderCallback) was invoked asynchronously by
    //    it's parent Reaction (processOrder) by doing
    //    "JEMReaction.registerReaction".
    //
    //    This reaction can access all it's parent's variables by using
    //    the JEMProcess concept as shown in the following 3 steps.
    //
    //    In this case, it is the PO Number that is accessed.

    // 5.1. Obtain the current AC4J process handle
    JEMProcess  currentAC4JProcess = (JEMProcess) JEMProcess.getProcess();

    // 5.2. Retrieve the first input parameter of parent reaction (processOrder)
    JEMTuple inTuple = currentAC4JProcess.getInTupleByIndx(0);

    // 5.3. Get PO Number from parent's input parameter
    poNumber = (Integer)inTuple.getObjInst();

    // 5.4. Look up PurchaseOrder Entity Bean
    poRemote = lookupPOBean(poNumber);

    // 5.5. Start updating the PO status depending on the replies...
    poRemote.setStatus(Status.getString(Status.STATUS_PROCESSED));

    // 5.6. Check the credit info
    int local_status = Status.STATUS_PROCESSED;
    if(credInfo != null)
    {
        if(credInfo.equalsIgnoreCase("credit approved"))
            local_status = Status.STATUS_VALID_CREDIT;
        else if(credInfo.equalsIgnoreCase("credit failed"))
```

```
            local_status = Status.STATUS_CANCELLED_NOCREDIT;
        else if(credInfo.equalsIgnoreCase("Invalid Credit Card"))
            local_status = Status.STATUS_INVALID_CREDIT_CARD;
    } else
        local_status = Status.STATUS_CANCELLED_NOCREDIT;

    // 5.7. Check the inventory info
    for(int i=0; local_status == Status.STATUS_VALID_CREDIT && i<invInfo.length;
        i++)
    {
        if(!invInfo[i])
            local_status = Status.STATUS_CANCELLED_NOINV;
    }

    // 5.8. Set the final status of the Purchase Order
    if(local_status == Status.STATUS_VALID_CREDIT)
        poRemote.setStatus(Status.getString(Status.STATUS_SHIPPED));
    else
        poRemote.setStatus(Status.getString(local_status));
}
```

次の説明は、例 11-5 の番号付けに対応しています。

5.1 現行の AC4J プロセス・ハンドルを取得します。

processOrderCallback リアクションは、それを作成した processOrder リアクションと同じプロセスに存在します。processOrder は processOrder プロセスのベース・リアクションであり、processOrderCallback は子リアクションです。

現行リアクションがプロセスに対してグローバルなデータにアクセスできるように、プロセスへのハンドルが取得されます。この場合、ベース・リアクションの入力パラメータの 1 つである発注番号にアクセスします。

```
JEMProcess currentAC4JProcess = (JEMProcess) JEMProcess.getProcess();
```

5.2 親リアクションの最初の入力パラメータを取得します。

親リアクションは、このプロセスのベース・リアクションです。最初のパラメータは、タプルの最初のトークンです。

```
JEMTuple inTuple = currentAC4JProcess.getInTupleByIndx(0);
```

5.3 親の入力パラメータから発注番号を取得します。

パラメータの全セットから、発注番号が取得されます。

```
poNumber = (Integer)inTuple.getObjInst();
```

5.4 発注の Entity Bean をルックアップします。

この Entity Bean は、手順 5.5 ～ 5.7 で受信ステータスを更新できるように取得されます。

```
poRemote = lookupPOBean(poNumber);
```

5.5 ～ 5.7 応答に従って、発注ステータスを更新します。

これらの手順は標準の Entity Bean 操作で、AC4J には関係しません。

手順 6: クライアントによる発注番号の非同期受信と発注ステータスのポーリング

クライアントは、発注リクエストに対する応答を受信する必要があります。前述のとおり、各リクエスト（またはコール）は、プロセス・コンテキスト（相互作用 ID（IID）およびアクティブ化 ID（AID））によって識別されます。プロセス・コンテキストを使用して、クライアントは、AC4J データ・バスから応答をプルできます。

クライアントは、応答から受信した `JEMEmitToken` を解析できます。クライアントが OC4J コンテナ内に存在する場合、コンテナは、`JEMEmitToken` を必要なタイプに分解します。コンテナの外部に存在する場合、例 11-6 に示すように、クライアントは応答を正しく解析する必要があります。

非同期応答では発注番号のみが返されます。発注のステータス自体は返されません。発注番号を取得した後、クライアントは、標準的な同期の EJB コールによって発注が完了するまで Entity Bean を直接ポーリングします。

例 11-6 クライアントによる発注番号の非同期受信と発注ステータスのポーリング

```
static final String DATA_SOURCE_NAME  = "java:comp/env/jdbc/OracleDS";
static final String BEAN_NAME = "java:comp/env/ejb/PurchaseOrderService";
static final String ACTIVE_EJB_NAME = "JEMPurchaseOrderService";
static final String DB_USER    = "JEMUSER";
static final String DB_PASSWD  = "JEMPASSWD";

public static void main(String[] args) throws  ClassNotFoundException, Exception
{
    // 6.0. Create a JNDI Context
    Context context = new InitialContext();

    // 6.1. Look up the DataSource where AC4J data bus resides
    DataSource client_ds = (DataSource) context.lookup(DATA_SOURCE_NAME);

    // 6.2. Obtain a JDBC connection to the DataSource
    OracleConnection conn =
    (OracleConnection)client_ds.getConnection(DB_USER, DB_PASSWD);
```

```
        // 6.3. Create an AC4J connection, using the JDBC Connection
        ac4j_conn = new JEMConnection(conn);

        // 6.4. Create an AC4J session on the data bus
        ac4j_sess = new JEMSession(ac4j_conn);

        // 6.5. Look up the handle of the PurchaseOrderService Active EJB
        activeEJBHandle = (JEMHandle)context.lookup(ACTIVE_EJB_NAME);

        // 6.6. Receive Response for the specified Process Context
        //    (IID + AID) plus reaction (takeOrder)
        //    Blocks indefintely, because timeout is 0
        JEMEmitToken resptoken =
            ac4j_sess.receiveReactionResponse(args[1],         // IID
                args[2],         // AID
                activeEJBHandle, // Active EJB Handle
                "takeOrder",     // Reaction/Method
                0);              // Timeout (seconds)

        // 6.7. Retrieve data from Reaction Response
        Object object = resptoken.getReactionResponseObjectInstance();

        // 6.8. Extract PurchaseOrder Number from data
        Integer poNumber = retrievePONumber(object);

        // 6.9. Commit the transaction
        ((OracleConnection)ac4j_sess.getJEMConnection().getConnection()).commit();

        // 6.10. Look up the PurchaseOrder Service Bean
        PurchaseOrderServiceRemote posb = lookupPurchaseOrderServiceBean();

        // 6.11. Poll the Purchase Order status - synchronously (regular EJB calls)
        //    until the status = shipped
        System.out.println("receiveResponse: polling for PO status...");
        String status = "";
        while(status.compareTo("STATUS_SHIPPED") != 0)
        {
            status = posb.getStatus(poNumber.intValue());
            System.out.println("  status = " + status);
        }

        if (status.compareTo("STATUS_SHIPPED") == 0)
        {
            System.out.println("\nreceiveResponse: Status = SHIPPED. Done.");
        }
    }
```

次の説明は、例 11-6 の番号付けに対応しています。

6.0 ～ 6.4 AC4J 接続を取得します。

これらの手順は、手順 1.0 ～ 1.4 と同じです。

6.5 発注サービスのアクティブ EJB のハンドルをルックアップします。

```
activeEJBHandle = (JEMHandle)context.lookup(ACTIVE_EJB_NAME);
```

このハンドルは、手順 6.6 のコールに提供する必要があります。

6.6 指定したプロセス・コンテキストに対する応答を受信します。

```
JEMEmitToken resptoken =
   ac4j_sess.receiveReactionResponse(args[1],         // IID
      args[2],          // AID
      activeEJBHandle, // Active EJB Handle
      "takeOrder",      // Reaction/Method
      0);               // Timeout (seconds)
```

IID および AID は「手順 1: クライアントによる発注サービス・アクティブ EJB への非同期リクエストの送信」の終了時の出力で、「手順 6: クライアントによる発注番号の非同期受信と発注ステータスのポーリング」に渡されます。

6.7 リアクションの応答からデータを取得します。

戻り値は、返されたトークンから Java オブジェクトとして抽出されます。

```
Object object = resptoken.getReactionResponseObjectInstance();
```

6.8 データから発注番号を抽出します。

次に、戻り値がその実際の型にキャストされます。

```
Integer poNumber = retrievePONumber(object);
```

6.9 トランザクションをコミットします。

この手順は、クライアントが AC4J コンテナの外部に存在しているため必要です。

```
((OracleConnection)ac4j_sess.getJEMConnection().getConnection()).commit();
```

6.10 ～ 6.11 発注サービスの Bean をルックアップし、ステータスが出荷済になるまでポーリングします。

発注番号の取得後、クライアントは発注の Entity Bean を直接ポーリングして発注ステータスを取得します。このプロセスは、ステータスが出荷済に変わるまで継続されます。これは標準的な EJB コードであるため、ここでは示しません。

AC4J でのアクティブ EJB のデプロイメント

アクティブ EJB は、普通の EJB として開発されています。EJB を AC4J の相互作用で使用できるようにするための変更点は、OC4J 固有のデプロイメント・ディスクリプタにあります。これについて、次に説明します。

OC4J 固有のデプロイメント・ディスクリプタの AC4J 要素仕様を使用して、EJB をデプロイします。次の例では、アクティブ EJB として、takeOrder EJB を定義しています。

- <jem-server-extension> 要素は、この JAR ファイルのアクティブ EJB が AC4J 通信で使用するデータ・バスが含まれたデータベースを定義します。

```
<jem-server-extension data-source-location="jdbc/jemSuperuserDS">
  <description>AC4J datasource location</description>
</jem-server-extension>
```

- orion-ejb-jar.xml ファイルの <jem-deployment> 要素は、ejb-jar.xml ファイルで定義された EJB をアクティブ EJB として識別します。この要素は、AC4J コール内で Bean を識別するために使用する AC4J 名（jem-name）を提供します。たとえば、この Bean は JEMPurchaseOrderBean として定義され、JEMHandle の作成で使用されています。サービスをリクエストし、アクティブ EJB から応答を取得できるコール元の識別情報は、called-by タグで宣言できます。この caller タグによって、データ・バス内のユーザーを識別します。たとえば、JEMCLIUSER は、jem-session の作成で使用されたユーザー名です。

```
<jem-deployment jem-name="JEMPurchaseOrderBean"
                ejb-name="PurchaseOrderBean">
  <description>AC4J EJB</description>
  <called-by>
    <caller caller-identity="JEMCLIUSER"/>
  </called-by>
</jem-deployment>
```

次のコードは、3 つのアクティブ EJB に対する orion-ejb-jar.xml の全ファイルです。

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE orion-ejb-jar PUBLIC "-//Evermind//DTD Enterprise JavaBeans 1.1
runtime//EN" "http://xmlns.oracle.com/ias/dtds/orion-ejb-jar.dtd">

<orion-ejb-jar>
 <enterprise-beans>
   <entity-deployment name="Products" location="ejb/Product"
       table="PRODUCTS_PURCHASEORDERSERVICE" data-source="jdbc/nonEmulatedDS" >
   </entity-deployment>

   <jem-server-extension data-source-location="jdbc/nonEmulatedDS"
     scheduling-threads="1">
```

```
     <description>JEMServer Deployment</description>
   </jem-server-extension>

   <jem-deployment jem-name="JEMPurchaseOrderService"
ejb-name="PurchaseOrderService" >
     <called-by>
           <caller caller-identity="JEMUSER" />
     </called-by>

     <security-identity>
       <description>using the caller identity</description>
       <use-caller-identity />
     </security-identity>
   </jem-deployment>
 </enterprise-beans>

 <assembly-descriptor>
   <security-role-mapping name="JEMUSER">
     <user name="JEMUSER" />
   </security-role-mapping>
   <default-method-access>
     <security-role-mapping name="&lt;default-ejb-caller-role&gt;" impliesAll="true"
/>
   </default-method-access>
 </assembly-descriptor>
</orion-ejb-jar>
```

A

OC4J 固有の DTD リファレンス

この付録では、OC4J 固有の EJB デプロイメント・ディスクリプタである `orion-ejb-jar.dtd` に含まれている要素について説明します。この付録では、DTD 内の構造と要素を簡単に説明しますが、ほとんどの要素は、このマニュアルの他の項で詳しく説明しています。

DTD は、`http://xmlns.oracle.com/ias/dtds/orion-ejb-jar.dtd` にあります。

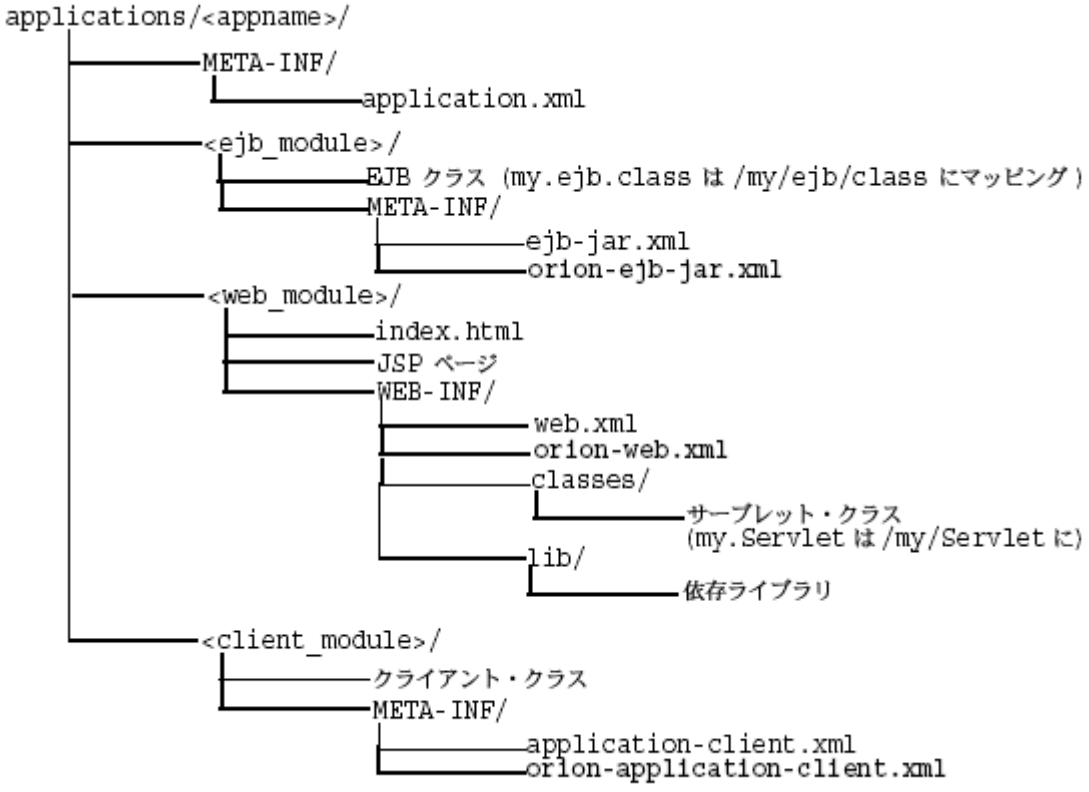
このデプロイメント・ディスクリプタについて、次の項で説明しています。

- 各要素セクションの全般的な説明 : この XML ファイルの要素の各セクションについては、A-3 ページの「OC4J 固有の EJB のデプロイメント・ディスクリプタ」で説明しています。
- 要素の説明 : 各要素については、A-22 ページの「要素の説明」で、アルファベット順に説明しています。

アプリケーションをデプロイするたびに、OC4J により、デフォルトの要素を持つ OC4J 固有の XML ファイルが自動的に生成されます。これらのデフォルトを変更する場合、`orion-ejb-jar.xml` ファイルを、元の `ejb-jar.xml` ファイルが存在している場所にコピーし、ここで変更する必要があります。XML ファイルをデプロイした場所で変更すると、アプリケーションが再びデプロイされた場合に、OC4J によってこれらの変更が上書きされます。開発ディレクトリで変更が行われた場合のみ、変更が維持されます。

オラクル社では、OC4J 固有の XML ファイルを、図 A-1 に示す推奨する開発用ディレクトリ構造に追加することをお薦めします。

図 A-1 開発アプリケーションのディレクトリ構造



OC4J 固有の EJB のデプロイメント・ディスクリプタ

OC4J 固有のデプロイメント・ディスクリプタには、Session Bean、Entity Bean、Message-Driven Bean、およびこれらの EJB のセキュリティに関する高度なデプロイ情報が含まれています。このデプロイメント・ディスクリプタ内の主な要素構造は、次のようになっています。

```
<orion-ejb-jar deployment-time=... deployment-version=...>
 <enterprise-beans>
   <session-deployment ...></session-deployment>
   <entity-deployment ...></entity-deployment>
   <message-driven-deployment ...></message-driven-deployment>
   <jem-deployment ...></jem-deployment>
   <jem-server-extension ...></jem-server-extension>
 </enterprise-beans>
 <assembly-descriptor>
   <security-role-mapping ...></security-role-mapping>
   <default-method-access></default-method-access>
 </assembly-descriptor>
</orion-ejb-jar>
```

<orion-ejb-jar> メイン・タグの下の各セクションは、それぞれ用途が決まっています。これらについては、次の項で説明します。

- Enterprise Beans セクション
- アセンブリ・ディスクリプタ・セクション

Enterprise Beans セクション

<enterprise-beans> セクションでは、すべての EJB（Session Bean、Entity Bean および Message-Driven Bean）の追加のデプロイ情報を定義します。各 EJB のタイプごとにセクションが分かれています。

次の各項で、<enterprise-beans> 要素内の要素を説明します。

- Session Bean セクション
- Entity Bean セクション
- Message-Driven Bean セクション
- EJB 1.1 CMP フィールド・マッピング・セクション
- メソッドの定義

Session Bean セクション

<session-deployment> セクションでは、この JAR ファイル内でデプロイされた Session Bean の追加のデプロイ情報を提供します。<session-deployment> セクションには、次の構造が含まれています。

```
<session-deployment pool-cache-timeout=... call-timeout=... copy-by-value=...
      location=... max-instances=... min-instances=... max-tx-retries=...
      tx-retry-wait=... name=... persistence-filename=... replication=...
      timeout=... idletime=... memory-threshold=... max-instances-threshold=...
      resource-check-interval=... passivate-count=... wrapper=...
      local-wrapper=...
  <ior-security-config>
    <transport-config>
      <integrity></integrity>
      <confidentiality></confidentiality>
      <establish-trust-in-target></establish-trust-in-target>
      <establish-trust-in-client></establish-trust-in-client>
    </transport-config>
    <as-context>
      <auth-method></auth-method>
      <realm></realm>
      <required></required>
    </as-context>
    <sas-context>
      <caller-propagation></caller-propagation>
    </sas-context>
  </ior-security-config>
  <env-entry-mapping name=...> </env-entry-mapping
  <ejb-ref-mapping location=... name=... />
  <resource-ref-mapping location=... name=... >
    <lookup-context location=...>
       <context-attribute name=... value=... />
    </lookup-context>
  </resource-ref-mapping>
  <resource-env-ref-mapping location=... name=... />
</session-deployment>
```

各要素グループについては、OC4J ドキュメント・セットの次の項で説明されています。

- <session-deployment> 要素を含む Session Bean の例は、第 2 章「EJB 入門」にある 2-24 ページの「デプロイメント・ディスクリプタの作成」で説明されています。
- <ior-security-config> 要素は相互運用性の要素です。詳細は、『Oracle Application Server Containers for J2EE サービス・ガイド』の相互運用性の章で説明されています。
- <env-entry-mapping> 要素は、環境変数を JNDI 名にマッピングします。詳細は、9-12 ページの「環境変数」で説明されています。

- <ejb-ref-mapping> 要素は、EJB 参照を JNDI 名にマッピングします。詳細は、9-19 ページの「他の Enterprise JavaBeans の環境参照」で説明されています。
- <resource-ref-mapping> 要素は、EJB 参照を JNDI 名にマッピングします。詳細は、9-19 ページの「リソース・マネージャのコネクション・ファクトリ参照への環境参照」で説明されています。
- <resource-env-ref-mapping> 要素は、リソースの管理オブジェクトをマッピングするために使用されます。たとえば、JMS を使用するために、Bean は JMS ファクトリ・オブジェクトと接続先オブジェクトの両方を取得する必要があります。これらのオブジェクトは、JNDI から同時に取得されます。<resource-ref> 要素で JMS ファクトリを宣言し、<resource-env-ref> 要素を使用して接続先を宣言します。したがって、<resource-env-ref-mapping> 要素は接続先オブジェクトをマッピングします。詳細は、7-33 ページの「クライアントが MDB にアクセスするときに論理名を使用する方法」を参照してください。

<session-deployment> 要素の属性は次のとおりです。

表 A-1 <session-deployment> 要素の属性

属性	説明
pool-cache-timeout	pool-cache-timeout は、ステートレス Session EJB に適用されます。このパラメータは、プールにキャッシュされたステートレス・セッションを維持する期間を指定します。 ステートレス Session Bean の場合、pool-cache-timeout を指定すると、pool-cache-timeout ごとに、プール内の対応する Bean タイプの Bean がすべて削除されます。値が 0（ゼロ）または負の場合は、pool-cache-timeout が無効になり、Bean はプールから削除されません。 デフォルト値は 60（秒）です。
call-timeout	このパラメータは、ビジネス・メソッドまたはライフ・サイクル・メソッドを起動するリソースを待機する最大時間を指定します。これは、ビジネス・メソッドが起動するまでのタイムアウトではありません。 タイムアウトに達すると、TimedOutException がスローされます。これによって、データベース接続が除外されます。 デフォルトは 90000 ミリ秒です。0（ゼロ）に設定すると、タイムアウトはありません。詳細は、『Oracle Application Server 10*g* パフォーマンス・ガイド』の EJB の項を参照してください。

表 A-1 <session-deployment> 要素の属性（続き）

属性	説明
copy-by-value	EJB コールのすべての受信および送信パラメータをコピーする（クローンを作成する）かどうか。速度を上げるには、使用するアプリケーションが copy-by-value セマンティクスを前提としないことが確実な場合、この値を 'false' に設定します。デフォルトは 'true' です。
location	この Bean がバインドされる JNDI 名。
max-instances	メモリー内に存在できるインスタンス化またはプールされた Bean インスタンスの数。この値に達すると、コンテナは最も古い Bean インスタンスをメモリーから非アクティブ化しようとします。非アクティブ化に失敗した場合、コンテナは call-timeout 属性に設定されたミリ秒数待機して、非アクティブ化、remove() メソッドまたは Bean の期限切れのいずれかによって、メモリーから削除された Bean インスタンスがあるかどうかを確認し、その後で TimeoutExpiredException をクライアントにスローします。Bean インスタンスの数を無限に許可する場合は、max-instances 属性を 0（ゼロ）に設定できます。デフォルトは 0（ゼロ）で、無限を意味します。この属性は、ステートレス Session Bean およびステートフル Session Bean の両方に適用されます。
min-instances	インスタンス化またはプールされた状態で維持される Bean 実装インスタンスの最小数。デフォルトは 0 です。この設定が有効なのは、ステートレス Session Bean のみです。
max-tx-retries	このパラメータは、システム・レベルの障害によって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。 通常は、再試行によってエラーの解決が見込める場合にのみ再試行を追加することをお薦めします。たとえば、分離レベル serializable を使用していて、競合が発生したときには自動的にトランザクションが再試行されるようにするために、再試行を使用することが考えられます。ただし、競合の発生が Bean に通知される場合は、max-tx-retries=0 のままにしておく必要があります。 デフォルト値は 3 です。詳細は、『Oracle Application Server 10*g* パフォーマンス・ガイド』の EJB の項を参照してください。
tx-retry-wait	このパラメータは、トランザクションの再試行の時間間隔を秒数で指定します。デフォルトは 60 秒です。

表 A-1 <session-deployment> 要素の属性（続き）

属性	説明
name	Bean の名前。これは、EJB デプロイメント・ディスクリプタのアセンブリ・セクション（ejb-jar.xml）内の Bean の名前に一致します。
persistence-filename	再起動のたびにセッションが保存されるファイルへのパス。
replication	ステートフル Session Bean に対する状態レプリケーションの構成。値は、VMTermination、EndOfCall または None のいずれかです。None がデフォルトです。詳細は、10-6 ページの「ステートフル Session Bean 用の EJB レプリケーションの構成」を参照してください。
timeout	秒数による timeout 属性は、ステートフル Session EJB に適用されます。値が 0（ゼロ）または負の数の場合、すべてのタイムアウトが使用禁止になります。 この timeout パラメータは、ステートフル Session Bean に関する非アクティブのタイムアウトです。30 秒ごとに、プール・クリーンアップ・ロジックが起動します。プール・クリーンアップ・ロジックの実行中に、タイムアウト値を渡すことによって削除されるのは、タイムアウトしたセッションのみです。 アプリケーションでのステートフル Session Bean の使用状況に応じて、タイムアウトを調整します。たとえば、ステートフル Session Bean を明示的に削除しないアプリケーションの場合は、多数のステートフル Session Bean が作成されるため、タイムアウト値を短い時間に設定できます。 アプリケーションでステートフル Session Bean を 1800 秒（30 分）以上使用可能にする必要がある場合は、タイムアウト値をそれにあわせて調整します。 デフォルト値は 1800 秒（30 分）です。
idletime	各 Bean に対してアイドル・タイムアウトを設定できます。このタイムアウトが経過すると、非アクティブ化が発生します。この属性を適切な秒数に設定します。デフォルトは 300 秒（5 分）です。この属性を無効にするには、"never" を指定します。
memory-threshold	非アクティブ化が発生するまでに使用可能な JVM メモリーの量に対するしきい値を定義します。パーセンテージとして解釈される整数を指定します。この値に達すると、アイドル・タイムアウトが経過していない場合でも Bean は非アクティブ化されます。デフォルトは 80% です。この属性を無効にするには、"never" を指定します。

表 A-1 <session-deployment> 要素の属性（続き）

属性	説明
max-instances-threshold	max-instances 属性の定義に応じて、存在するアクティブ Bean の数に対するしきい値を定義します。パーセンテージとして解釈される整数を指定します。max-instances を 100、max-instances-threshold を 90% に定義した場合は、アクティブ Bean インスタンスの数が 90 を超えると、Bean の非アクティブ化が発生します。デフォルトは 90% です。この属性を無効にするには、"never" を指定します。
resource-check-interval	コンテナは、すべてのリソースをこの時間間隔でチェックします。この時点でいずれかのしきい値に達している場合は、非アクティブ化が発生します。デフォルトは 180 秒（3 分）です。この属性を無効にするには、"never" を指定します。
passivate-count	いずれかのリソースしきい値に達した場合に非アクティブ化される Bean の数を定義する整数です。Bean の非アクティブ化は、最低使用頻度アルゴリズムを使用して実行されます。デフォルトは、max-instances 属性の 1/3 です。この属性を無効にするには、カウントを 0（ゼロ）または負の数に設定します。
wrapper	この Bean の OC4J ラッパー・クラスの名前。内部サーバー値のため、編集しないでください。
local-wrapper	この Bean の OC4J ローカル・ホーム・ラッパー・クラスの名前。内部サーバー値のため、編集しないでください。

Entity Bean セクション

<entity-deployment> セクションでは、この JAR ファイル内でデプロイされた Entity Bean の追加のデプロイ情報を提供します。<entity-deployment> セクションには、次の構造が含まれています。

```
<entity-deployment call-timeout=... clustering-schema=...
      copy-by-value=... data-source=... exclusive-write-access=...
      do-select-before-insert=... instance-cache-timeout=... isolation=...
      location=... locking-mode=... max-instances=... min-instances=...
      max-tx-retries=... tx-retry-wait=... update-chnaged-fields-only=...
      name=... pool-cache-timeout=...
      table=... validity-timeout=... force-update=...
      wrapper=... local-wrapper=... delay-updates-until-commit=...
      findByPrimaryKey-lazy-loading=... >
  <ior-security-config>
    <transport-config>
      <integrity></integrity>
      <confidentiality></confidentiality>
      <establish-trust-in-target></establish-trust-in-target>
```

```
      <establish-trust-in-client></establish-trust-in-client>
    </transport-config>
    <as-context>
      <auth-method></auth-method>
      <realm></realm>
      <required></required>
    </as-context>
    <sas-context>
      <caller-propagation></caller-propagation>
    </sas-context>
  </ior-security-config>
  <primkey-mapping>
   <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
      persistence-type=...></cmp-field-mapping>
  </primkey-mapping>
  <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
      persistence-type=...> </cmp-field-mapping>
  <finder-method partial=... query=... lazy-loading=... prefetch-size=... >
   <method></method>
  </finder-method>
  <env-entry-mapping name=...></env-entry-mapping>
  <ejb-ref-mapping location=... name=... />
  <resource-ref-mapping location=... name=... >
    <lookup-context location=...>
       <context-attribute name=... value=... />
    </lookup-context>
  </resource-ref-mapping>
  <resource-env-ref-mapping location=... name=... />
</entity-deployment>
```

各要素グループについては、OC4J ドキュメント・セットの次の項で説明されています。

- <entity-deployment> 要素を含む Entity Bean の例は、第 3 章「CMP Entity Bean」、第 4 章「エンティティ関連（E-R）のマッピング」、第 5 章「EJB 問合せ言語」および第 6 章「BMP Entity Bean」で説明されています。
- <ior-security-config> 要素は、相互運用性の CSIv2 セキュリティ・ポリシーを構成します。詳細は、『Oracle Application Server Containers for J2EE サービス・ガイド』の相互運用性の章で説明されています。
- <primkey-mapping> 要素は、主キーを対応する CMP フィールドにマッピングします。詳細は、3-16 ページの「永続フィールドのデータベースへの明示的なマッピング」を参照してください。
- <cmp-field-mapping> 要素は、各 <cmp-field> 要素をそのデータベース行にマッピングします。詳細は、3-16 ページの「永続フィールドのデータベースへの明示的なマッピング」を参照してください。

- <finder-method> 要素は、EJB 1.1 Entity Bean の finder メソッドを作成するために使用されます。EJB 2.0 の finder メソッドを作成する方法は、「EJB 問合せ言語」を参照してください。この要素で EJB 1.1 の finder メソッドを継続して使用する方法は、B-11 ページの「EJB 1.1 の高度な finder メソッド」を参照してください。
- <env-entry-mapping> 要素は、環境変数を JNDI 名にマッピングします。詳細は、9-12 ページの「環境変数」で説明されています。
- <ejb-ref-mapping> 要素は、EJB 参照を JNDI 名にマッピングします。詳細は、9-19 ページの「他の Enterprise JavaBeans の環境参照」で説明されています。
- <resource-ref-mapping> 要素は、EJB 参照を JNDI 名にマッピングします。詳細は、9-19 ページの「リソース・マネージャのコネクション・ファクトリ参照への環境参照」で説明されています。
- <resource-env-ref-mapping> 要素は、リソースの管理オブジェクトをマッピングするために使用されます。たとえば、JMS を使用するために、Bean は JMS ファクトリ・オブジェクトと接続先オブジェクトの両方を取得する必要があります。これらのオブジェクトは、JNDI から同時に取得されます。<resource-ref> 要素で JMS ファクトリを宣言し、<resource-env-ref> 要素を使用して接続先を宣言します。したがって、<resource-env-ref-mapping> 要素は接続先オブジェクトをマッピングします。詳細は、7-33 ページの「クライアントが MDB にアクセスするときに論理名を使用する方法」を参照してください。

<entity-deployment> 要素の属性は次のとおりです。

表 A-2 <entity-deployment> 要素の属性

属性	説明
call-timeout	このパラメータは、ビジネス・メソッドまたはライフ・サイクル・メソッドを起動するリソースを待機する最大時間を指定します。これは、ビジネス・メソッドが起動するまでのタイムアウトではありません。 タイムアウトに達すると、TimedOutException がスローされます。これによって、データベース接続が除外されます。 デフォルトは 90000 ミリ秒です。0（ゼロ）に設定すると、タイムアウトはありません。詳細は、『Oracle Application Server 10*g* パフォーマンス・ガイド』の EJB の項を参照してください。
clustering-schema	使用しません。このリリースでは必要ありません。

表 A-2 <entity-deployment> 要素の属性（続き）

属性	説明
copy-by-value	EJB コールのすべての受信および送信パラメータをコピーする（クローンを作成する）かどうか。速度を上げるには、使用するアプリケーションが copy-by-value セマンティクスを前提としないことが確実な場合、この値を 'false' に設定します。デフォルトは 'true' です。
data-source	コンテナ管理の永続性を使用している場合、データ・ソースの名前。
exclusive-write-access	EJB サーバーがバックエンド・データベースへの排他的書込み（更新）アクセスがあるかどうか。"read_only" の Entity Bean でのみ使用できます。使用した場合、共有の Bean 操作のパフォーマンスが向上し、キャッシングの効率が向上します。 このパラメータは、使用するコミット・オプション（EJB 仕様で定義されている A、B または C、）に対応しています。exclusive-write-access = true の場合は、コミット・オプション A になります。 デフォルトは、Bean に対して locking-mode が optimistic または pessimistic の場合は false、locking-mode が read-only の場合は true です。 exclusive-write-access 属性は、locking-mode が pessimistic または optimistic の場合は必ず false に設定され、EJB クラスタリングでは使用されません。locking-mode が read-only の場合は、exclusive-write-access 属性を false に設定できますが、パフォーマンスへの影響はありません。これは、変更されたフィールドがない場合は ejbStores メソッドがスキップされるためです。パフォーマンスを向上させ、読取り専用の Bean に対して ejbLoads メソッドを実行しないようにするには、exclusive-write-access=true に設定する必要があります。 詳細は、9-10 ページの「データベースへの排他的書込みアクセス」を参照してください。

表 A-2 <entity-deployment> 要素の属性（続き）

属性	説明
do-select-before-insert	false の場合は、挿入前の SELECT 文の実行を回避します。通常、この余分な SELECT 文は、重複を回避するために、エンティティがすでに存在するかどうかを挿入前にチェックします。 エンティティに一意キー制約を定義している場合は、この属性を false に設定することをお薦めします。一意キー制約がない場合は、この属性を false に設定すると、重複した挿入が検出されなくなります。この場合に重複した挿入を防止するには、この属性を true のままにします。 パフォーマンスの理由から、この属性は false に設定し、挿入前の余分な SELECT 文の実行を回避することをお薦めします。デフォルト値は true です。
instance-cache-timeout	エンティティ・ラッパー・インスタンスを識別情報に割り当てておく秒単位の時間。'never' を指定すると、ラッパー・インスタンスは、ガベージ・コレクションが行われるまで維持されます。デフォルトは 60 秒です。
location	この Bean がバインドされる JNDI 名。
isolation	データベース処理の分離レベルを指定します。Oracle データベースに対する有効な値は、'serializable' および 'committed' です。デフォルトは 'committed' です。Oracle データベース以外の場合は、'none'、'committed'、'serializable'、'uncommitted' および 'repeatable_read' が有効です。 詳細は、9-8 ページの「Entity Bean 同時実行性モードおよびデータベース分離モード」および『Oracle Application Server 10*g* パフォーマンス・ガイド』を参照してください。

表 A-2 <entity-deployment> 要素の属性（続き）

属性	説明
locking-mode	モードによって、リソースの競合を管理するためのブロック時期、またはパラレルで実行する時期を構成します。詳細は、9-8 ページの「Entity Bean 同時実行性モードおよびデータベース分離モード」および『Oracle Application Server 10*g* パフォーマンス・ガイド』を参照してください。次の同時実行性モードがあります。 ■ PESSIMISTIC: リソースの競合を管理し、パラレル実行はできません。Entity Bean を実行できるのは、一度に 1 ユーザーのみです。 ■ OPTIMISTIC: 複数のユーザーがパラレルで Entity Bean を実行できます。リソースの競合は監視しないため、データの一貫性を維持するには、データベース分離モードを使用する必要があります。これはデフォルトです。 ■ READ-ONLY: 複数のユーザーがパラレルで Entity Bean を実行できます。コンテナでは、Bean の状態を更新できません。
max-instances	インスタンス化またはプールされた状態で維持される Bean 実装インスタンスの最大数。デフォルトは 0（ゼロ）で、無限を意味します。詳細は、9-12 ページの「環境参照の構成」を参照してください。
min-instances	インスタンス化またはプールされた状態で維持される Bean 実装インスタンスの最小数。デフォルトは 0 です。詳細は、9-12 ページの「環境参照の構成」を参照してください。
max-tx-retries	このパラメータは、システム・レベルの障害によって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。 通常は、再試行によって解決できるエラーがある場合のみ再試行の回数を追加することをお薦めします。たとえば、分離レベル serializable を使用していて、競合が発生したとき自動的にトランザクションが再試行されるようにする場合には、再試行を使用できます。ただし、競合の発生時に Bean に通知する場合は、max-tx-retries=0 のままにしておく必要があります。 デフォルト値は 3 です。詳細は、『Oracle Application Server 10*g* パフォーマンス・ガイド』の EJB の項を参照してください。

表 A-2 <entity-deployment> 要素の属性（続き）

属性	説明
tx-retry-wait	このパラメータは、トランザクションの再試行の時間間隔を秒数で指定します。デフォルトは 60 秒です。
update-changed-fields-only	ejbStore の起動時に、コンテナが、CMP Entity Bean の永続記憶域に対して変更されたフィールドのみ更新するか、または全フィールドを更新するかを指定します。デフォルトは true で、変更されたフィールドのみ更新されます。詳細は、9-8 ページの「永続性を更新する手法」を参照してください。
name	Bean の名前。これは、EJB デプロイメント・ディスクリプタのアセンブリ・セクション（ejb-jar.xml）内の Bean の名前に一致します。
pool-cache-timeout	Bean 実装インスタンスを、プールされた（割り当てられていない）状態で維持する秒単位の時間。'never' を指定すると、ガベージ・コレクションが行われるまでインスタンスを維持します。デフォルトは 60 です。
table	コンテナ管理の永続性を使用している場合、データベース内の表の名前。
validity-timeout	エンティティがキャッシュ内で有効である（再ロードされるまでの）最大の期間（ミリ秒単位）。既存システムからの更新がほとんど発生しない疎結合環境で役立ちます。この属性は、locking-mode が read_only の Entity Bean について、exclusive-write-access="true"（デフォルト）の場合にのみ有効です。 外部でのデータ変更がない場合（したがって、exclusive-write-access=true に設定）は、0 または -1 に設定してこのオプションを使用禁止にすることをお薦めします。これは、外部での変更がない読取り専用の EJB の場合は、キャッシュ内のデータは常に有効であるためです。 EJB の外部での変更は通常ない（したがって、exclusive-write-access=true に設定）が、表の更新があるため、キャッシュの更新がときどき必要な場合は、外部でのデータ変更の間隔に応じた値を設定します。

表 A-2 <entity-deployment> 要素の属性（続き）

属性	説明
force-update	OC4J で永続データの変更が不明の場合は、force-update 属性を true に設定すると、OC4J は ejbStore メソッドを起動して EJB のライフ・サイクルを実行します。これによって、一時フィールドのデータが管理され、ejbStore メソッドの実行時に適切な永続フィールドが設定されます。たとえば、イメージを、メモリー内に保持している形式とは異なる形式で、データベース内に格納することができます。デフォルトは false です。
wrapper	この Bean の OC4J リモート・ホーム・ラッパー・クラスの名前。内部サーバー値のため、編集しないでください。
local-wrapper	この Bean の OC4J ローカル・ホーム・ラッパー・クラスの名前。内部サーバー値のため、編集しないでください。
delay-updates-until-commit	この属性は、CMP Entity Bean でのみ有効です。トランザクション・データのフラッシュをコミット・タイムまで遅延するかどうかを指定します。デフォルトは true です。この値を false に設定すると、EJB メソッド（ejbRemove() および finder メソッドを除く）の起動が完了するたびに、永続データが更新されます。

Message-Driven Bean セクション

<message-driven-deployment> セクションでは、この JAR ファイル内でデプロイされた Message-Driven Bean の追加のデプロイ情報を提供します。
<message-driven-deployment> セクションには、次の構造が含まれています。

```
<message-driven-deployment cache-timeout=... connection-factory-location=...
      destination-location=... name=... subscription-name=...
      listener-threads=... transaction-timeout=...
      dequeue-retry-count=... dequeue-retry-interval=... >
  <env-entry-mapping name=...></env-entry-mapping>
  <ejb-ref-mapping location=... name=... />
  <resource-ref-mapping location=... name=... >
    <lookup-context location=...>
       <context-attribute name=... value=... />
    </lookup-context>
  </resource-ref-mapping>
  <resource-env-ref-mapping location=... name=... />
</message-driven-deployment>
```

各要素グループについては、OC4J ドキュメント・セットの次の項で説明されています。

- <message-driven-deployment> 要素を含む Message-Driven Bean の例は、第 7 章「Message-Driven Bean」で説明されています。
- <env-entry-mapping> 要素は、環境変数を JNDI 名にマッピングします。詳細は、9-12 ページの「環境変数」で説明されています。
- <ejb-ref-mapping> 要素は、EJB 参照を JNDI 名にマッピングします。詳細は、9-19 ページの「他の Enterprise JavaBeans の環境参照」で説明されています。
- <resource-ref-mapping> 要素は、EJB 参照を JNDI 名にマッピングします。詳細は、9-19 ページの「リソース・マネージャのコネクション・ファクトリ参照への環境参照」で説明されています。
- <resource-env-ref-mapping> 要素は、リソースの管理オブジェクトをマッピングするために使用されます。たとえば、JMS を使用するために、Bean は JMS ファクトリ・オブジェクトと接続先オブジェクトの両方を取得する必要があります。これらのオブジェクトは、JNDI から同時に取得されます。<resource-ref> 要素で JMS ファクトリを宣言し、<resource-env-ref> 要素を使用して接続先を宣言します。したがって、<resource-env-ref-mapping> 要素は接続先オブジェクトをマッピングします。詳細は、7-33 ページの「クライアントが MDB にアクセスするときに論理名を使用する方法」を参照してください。

<message-driven-deployment> 要素の属性は次のとおりです。

表 A-3 <message-driven-deployment> 要素の属性

属性	説明
cache-timeout	この要素は使用しません。
connection-factory-location	使用するコネクション・ファクトリの JNDI の場所。JMS の Destination Connection Factory は、connection-factory-location 属性で指定されます。構文は、"java:comp/resource" + リソース・プロバイダ名 + "TopicConnectionFactories" または "QueueConnectionFactories" + ユーザー定義名です。xxxConnectionFactories は、定義するファクトリのタイプを指定します。詳細は、7-23 ページの「Oracle JMS を使用する OC4J 固有のデプロイメント・ディスクリプタの作成」を参照してください。

表 A-3 <message-driven-deployment> 要素の属性（続き）

属性	説明
destination-location	使用する接続先（キュー、トピック）の JNDI の場所。JMS の Destination は、destination-location 属性で指定されます。構文は、"java:comp/resource" + リソース・プロバイダ名 + "Topics" または "Queues" + Destination 名です。Topic または Queue は、定義される Destination タイプを指定します。Destination 名は、データベースで定義された実際のキュー名またはトピック名です。詳細は、7-23 ページの「Oracle JMS を使用する OC4J 固有のデプロイメント・ディスクリプタの作成」を参照してください。
max-instances	この要素は使用しません。かわりに、listener-threads を使用します。
min-instances	この要素は使用しません。
name	Bean の名前。これは、EJB デプロイメント・ディスクリプタのアセンブリ・セクション（ejb-jar.xml）内の Bean の名前に一致します。
subscription-name	これがトピックの場合、サブスクリプション名は subscription-name 属性で定義されます。詳細は、7-23 ページの「Oracle JMS を使用する OC4J 固有のデプロイメント・ディスクリプタの作成」を参照してください。
listener-threads	リスナー・スレッドは、JMS メッセージを並行してコンシュームするために使用されます。デフォルトのスレッドは 1 つです。トピックに指定できるスレッドは 1 つのみです。キューには複数のスレッドを指定できます。詳細は、7-14 ページの「OC4J JMS を使用する OC4J 固有のデプロイメント・ディスクリプタ（orion-ejb-jar.xml）の作成」を参照してください。
transaction-timeout	この属性によって、コンテナ管理のトランザクション型 MDB のトランザクション・タイムアウト時間（秒）を制御します。デフォルトは 1 日（86,400 秒）です。この時間枠内でトランザクションが完了しない場合、トランザクションはロールバックされます。詳細は、7-23 ページの「Oracle JMS を使用する OC4J 固有のデプロイメント・ディスクリプタの作成」を参照してください。
dequeue-retry-count	データベース・フェイルオーバーの発生後、リスナー・スレッドによって試行される JMS セッションの再取得頻度を指定します。デフォルトは 0（ゼロ）です。この値は、MDB 内の CMT トランザクションに対してのみ有効です。詳細は、7-37 ページの「RAC データベース使用時のフェイルオーバー」を参照してください。
dequeue-retry-interval	再試行の間隔を指定します。デフォルトは 60 秒です。

AC4J のアクティブ EJB セクション

<jem-server-extension> セクションでは、AC4J データ・バスがインストールされているデータベースの JNDI 名を定義します。<jem-server-extension> セクションには、次の構造が含まれています。

```
<jem-server-extension data-source-location=... scheduling-threads=...>
  <description></description>
  <data-bus data-bus-name=... url=.../>
</jem-server-extension>
```

この要素の詳細は、『Oracle Application Server Containers for J2EE サービス・ガイド』を参照してください。

<jem-deployment> セクションでは、この JAR ファイル内でデプロイされたアクティブ EJB の追加のデプロイ情報を提供します。<jem-deployment> セクションには、次の構造が含まれています。

```
<jem-deployment jem-name=... ejb-name=...>
  <description></description>
  <data-bus data-bus-name=... url=.../>
  <called-by>
    <caller caller-identity=.../>
  </called-by>
  <security-identity>
   <description></description>
   <use-caller-identity></use-caller-identity>
  </security-identity>
</jem-deployment>
```

called-by 要素によって、アプリケーション・デプロイヤは、AC4J Bean で定義された非同期メソッドの使用を管理または制限できます。次の例にある "CLIUSER"、"SVRUSER" および "XTRAUSER" は、name="ABean" の EJB に対応する AC4JBeanA で定義されているすべてのメソッドを起動できます。"USER1" または "USER2" がこの AC4JBeanA を起動すると、コンテナは SecurityException をスローします。

```
<jem-deployment jem-name="AC4JBeanA" ejb-name="ABean">
  <called-by>
    <caller  caller-identity="CLIUSER"/>
    <caller  caller-identity="SVRUSER"/>
    <caller  caller-identity="XTRAUSER"/>
  </called-by>
</jem-deployment>
```

アプリケーション・デプロイヤが ABean EJB の security-role を role="USER1" に定義した場合、"USER1" は ABean EJB のすべてのメソッドを同時に起動できます。ただし、`called-by` 要素が "USER1" に対して定義されていないかぎり、"USER1" は、`AC4JBeanA` の同じ非同期メソッドを起動できません。

この要素の詳細は、『Oracle Application Server Containers for J2EE サービス・ガイド』を参照してください。

EJB 1.1 CMP フィールド・マッピング・セクション

EJB 1.1 CMP Entity Bean を使用する場合は、次の要素を使用して、CMP フィールドをデータベースにマッピングします。EJB 1.1 CMP データ・フィールドのマッピングに関する説明は、B-14 ページの「データベース表および列への EJB 1.1 CMP フィールドのマッピング」を参照してください。

次に、`orion-ejb-jar.xml` ファイル内で、CMP 永続データ・フィールドのマッピングに使用される XML 要素を示します。

```
<cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
      persistence-type=...>
   <fields>
      <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
          persistence-type=...></cmp-field-mapping>
   </fields>
   <properties>
      <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
           persistence-type=...></cmp-field-mapping>
   </properties>
   <entity-ref home=...>
      <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
          persistence-type=...></cmp-field-mapping>
   </entity-ref>
   <collection-mapping table=...>
      <primkey-mapping>
         <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
             persistence-type=...></cmp-field-mapping>
      </primkey-mapping>
      <value-mapping immutable="true|false" type=...>
         <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
             persistence-type=...></cmp-field-mapping>
      </value-mapping>
   </collection-mapping>
   <set-mapping table=...>
      <primkey-mapping>
         <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
             persistence-type=...></cmp-field-mapping>
      </primkey-mapping>
      <value-mapping immutable="true|false" type=...>
```

```
        <cmp-field-mapping ejb-reference-home=... name=... persistence-name=...
            persistence-type=...></cmp-field-mapping>
      </value-mapping>
   </set-mapping>
</cmp-field-mapping>
```

メソッドの定義

次の構造は、Bean のメソッド（および場合によってはメソッドのパラメータ）の指定に使用します。

```
<method>
   <description></description>
   <ejb-name></ejb-name>
   <method-intf></method-intf>
   <method-name></method-name>
   <method-params>
     <method-param></method-param>
   </method-params>
</method>
```

使用可能なスタイルは、次のいずれかです。

1. 指定した Enterprise Bean のホーム・インタフェースおよびリモート・インタフェースのすべてのメソッドを指す場合、次のようにしてメソッドを指定します。

   ```
   <method>
        <ejb-name>EJBNAME</ejb-name>
            <method-name>*</method-name>
     </method>
   ```

2. 同じオーバーロードされた名前を持つ複数のメソッドを指す場合、次のようにしてメソッドを指定します。

   ```
   <method>
     <ejb-name>EJBNAME</ejb-name>
          <method-name>METHOD</method-name>
     </method>>
   ```

3. オーバーロードされた名前を持つ一連のメソッドのうちの 1 つのメソッドを指す場合、次のようにしてメソッド内の各パラメータを指定します。

   ```
   <method>
         <ejb-name>EJBNAME</ejb-name>
         <method-name>METHOD</method-name>
             <method-params>
                   <method-param>PARAM-1</method-param>
                   <method-param>PARAM-2</method-param>
                      ...
   ```

```
                <method-param>PARAM-n</method-param>
            </method-params>
    </method>
```

<method> 要素は、security セクションおよび MDB セクション内で使用します。詳細は、8-4 ページの「EJB デプロイメント・ディスクリプタでの論理ロールの指定」を参照してください。

アセンブリ・ディスクリプタ・セクション

個別の Bean に対するデプロイ情報の指定以外に、<assembly-descriptor> セクションで、セキュリティ用の追加デプロイ・マッピング情報を指定できます。<assembly-descriptor> セクションには、次の構造が含まれています。

```
<assembly-descriptor>
  <security-role-mapping impliesAll=... name=...>
   <group name=... />
   <user name=... />
  </security-role-mapping>
  <default-method-access>
   <security-role-mapping impliesAll=... name=...>
    <group name=... />
    <user name=... />
   </security-role-mapping>
  </default-method-access>
</assembly-descriptor>
```

各要素グループについては、OC4J ドキュメント・セットの次の項で説明されています。

- <security-role-mapping> 要素については、8-9 ページの「ユーザーおよびグループへの論理ロールのマッピング」で説明されています。
- <default-method-access> 要素については、8-10 ページの「未定義メソッドに対するデフォルト・ロール・マッピングの指定」で説明されています。

要素の説明

<assembly-descriptor>
アセンブリ・ディスクリプタ要素のマッピング。

<called-by>
アプリケーション・デプロイヤは、AC4J Bean で定義された非同期メソッドの使用を管理または制限できます。この要素内で、Bean のすべてのメソッドを実行できるユーザー ID を指定します。AC4J Bean を実行できる ID は、1 つ以上の `<caller>` 要素で識別されます。

<caller>
AC4J Bean でメソッドを実行できるコール元 ID は、それぞれ単一の `<caller>` 要素内で定義されます。

属性 :

- caller-identity − AC4J Bean メソッドを実行できるセキュリティ・ロール。

<cmp-field-mapping>
コンテナ管理による永続的なフィールドのデプロイ情報。異なる動作を定義するサブタグが使用されていない場合、このフィールドは、シリアライズ化、または認識可能なプリミティブ型のネイティブ処理の際も維持されます。

属性 :

- ejb-reference-home −フィールドがエンティティ EJBObject または EJBHome の場合、フィールドのリモート EJB ホームの JNDI の場所。
- name −フィールドの名前。
- persistence-name −データベース表内のフィールドの名前。
- persistence-type −フィールドのデータベース型（有効な値はデータベースによって異なる）。

<collection-mapping>
Collection 型のリレーショナル・マッピングを指定します。Collection は、n 個の順序付けされていない（順序が指定されておらず、必要ない）アイテムで構成されます。マッピングを含んでいるフィールドは、java.util.Collection 型である必要があります。

属性 :

- table −データベース内の表の名前。

<context-attribute>
コンテキストに送信される属性。JNDI で必須の属性は、'java.naming.factory.initial' のみです。これは、コンテキスト・ファクトリ実装のクラス名です。

属性 :

- name －属性の名前。
- value －属性の値。

<data-bus>
AC4J オブジェクトに対する特定データ・バスの名前と URL。

属性 :

- data-bus-name －データ・バスのユーザー定義名。
- url －データ・バスの URL。JDBC URL と同じです。

<default-method-access>
対応するメソッド許可が存在しないメソッド用の、デフォルトのメソッド・アクセス・ポリシー。

<description>
短い説明。

<ejb-name>
ejb-name 要素は、Enterprise Bean の名前を指定します。この名前は、ejb-jar ファイルの作成者により、ejb-jar ファイルのデプロイメント・ディスクリプタ内の Enterprise Bean に対して割り当てられます。名前は、同じ ejb-jar ファイル内の Enterprise Bean の名前の中で一意である必要があります。Enterprise Bean のコードは、名前に依存していません。そのため、アプリケーションのアセンブリ・プロセス中に名前を変更しても、Enterprise Bean の機能を損ねません。デプロイメント・ディスクリプタの ejb-name と、Deployer が Enterprise Bean のホームに割り当てる JNDI 名の間には、設計上の関連性はありません。名前は、NMTOKEN の字句規則に従う必要があります。

<ejb-ref-mapping>
別の Enterprise Bean のホームの参照の宣言に使用される ejb-ref 要素。
ejb-ref-mapping 要素は、デプロイ時にこれを JNDI の場所に関連付けます。

属性 :

- location － EJB ホームのルックアップ元の JNDI の場所。
- name － ejb-ref の名前。ejb-jar.xml 内の ejb-ref の名前に一致します。

<enterprise-beans>
この EJB JAR ファイルに含まれている Bean。

<entity-deployment>
Entity Bean のデプロイ情報。

属性 :

- call-timeout － EJB が使用中の場合、EJB で使用するリソース（データベース接続を除く）の待機時間（10 進数による Long 型のミリ秒）。つまり、`RemoteException` をスローしてデッドロックとして扱うまでの時間。これは、SQL 問合せタイムアウトとしても使用されます。SQL 問合せが完了する前にタイムアウトが発生すると、SQL 例外がスローされます。タイムアウトに 0（ゼロ）を指定すると、タイムアウトは使用禁止になります。デフォルトは 90 秒です。
- clustering-schema －使用しないことをお薦めします。
- copy-by-value －すべての受信および送信 EJB コールについて、すべての受信および送信パラメータをコピーするかどうか。使用するアプリケーションが copy-by-value セマンティクスをこれらのパラメータの前提としていない場合、この値は 'false' に設定します。デフォルトは 'true' です。
- data-source －コンテナ管理の永続性を使用している場合、データ・ソースの名前。
- delay-updates-until-commit －トランザクション・データのフラッシュをコミット時まで遅延するかどうかを指定します。デフォルトは true です。変更ごとにデータベースを更新する場合は、この要素を false に設定します。
- do-select-before insert － false の場合は、挿入前の SELECT 文の実行を回避します。通常、この余分な SELECT 文は、重複を回避するために、エンティティがすでに存在するかどうかを挿入前にチェックします。

  エンティティに一意キー制約を定義している場合は、この属性を false に設定することをお薦めします。一意キー制約がない場合は、この属性を false に設定すると、重複した挿入が検出されなくなります。この場合に重複した挿入を防止するには、この属性を true のままにします。

  パフォーマンスの理由から、この属性は false に設定し、挿入前の余分な SELECT 文の実行を回避することをお薦めします。デフォルト値は true です。
- exclusive-write-access － EJB サーバーがバックエンド・データベースへの排他的書込み（更新）アクセスがあるかどうか。"`read_only`" の Entity Bean でのみ使用できます。使用した場合、共有の Bean 操作のパフォーマンスが向上し、キャッシングの効率が向上します。デフォルトは false です。詳細は、9-10 ページの「データベースへの排他的書込みアクセス」を参照してください。
- findByPrimaryKey-lazy-loading="true|false" － Entity Bean の finder メソッドの場合、遅延ロードによって select メソッドを複数回起動できます。遅延ロードをオンにし、この finder メソッドを 1 回のみ実行するには、このプロパティを true に設定します。デフォルトは false です。詳細は、9-7 ページの「CMP Entity Bean の finder メソッドにおける遅延ロードの構成」を参照してください。

- instance-cache-timeout －エンティティ・ラッパー・インスタンスを識別情報に割り当てておく秒単位の時間。'never' を指定すると、ラッパー・インスタンスは、ガベージ・コレクションが行われるまで維持されます。デフォルトは 60 秒です。
- isolation －データベース処理の分離レベルを指定します。Oracle データベースに対する有効な値は、'serializable' および 'committed' です。デフォルトは 'committed' です。Oracle データベース以外の場合は、'none'、'committed'、'serializable'、'uncommitted' および 'repeatable_read' が有効です。詳細は、9-8 ページの「Entity Bean 同時実行性モードおよびデータベース分離モード」および『Oracle Application Server 10*g* パフォーマンス・ガイド』を参照してください。
- local-wrapper －この Bean の OC4J ローカル・ホーム・ラッパー・クラスの名前。内部サーバー値のため、編集しないでください。
- location －この Bean がバインドされる JNDI 名。
- locking-mode －同時実行性モードによって、リソースの競合を管理するためのブロック時期、またはパラレルで実行する時期を構成します。詳細は、9-8 ページの「Entity Bean 同時実行性モードおよびデータベース分離モード」および『Oracle Application Server 10*g* パフォーマンス・ガイド』を参照してください。次の同時実行性モードがあります。
  - `PESSIMISTIC`: リソースの競合を管理し、パラレル実行はできません。Entity Bean を実行できるのは、一度に 1 ユーザーのみです。
  - `OPTIMISTIC`: 複数のユーザーがパラレルで Entity Bean を実行できます。リソースの競合は監視しないため、データの一貫性を維持するには、データベース分離モードを使用する必要があります。これはデフォルトです。
  - `READ-ONLY`: 複数のユーザーがパラレルで Entity Bean を実行できます。コンテナでは、Bean の状態を更新できません。
- max-instances －インスタンス化またはプールされた状態で維持される Bean 実装インスタンスの最大数。デフォルトは 0（ゼロ）で、無限を意味します。詳細は、9-12 ページの「環境参照の構成」を参照してください。
- min-instances －インスタンス化またはプールされた状態で維持される Bean 実装インスタンスの最小数。デフォルトは 0 です。詳細は、9-12 ページの「環境参照の構成」を参照してください。
- max-tx-retries －システム・レベルの障害によってロールバックされたトランザクションの再試行回数。デフォルトは 0（ゼロ）です。serializable 分離レベルを使用している場合は、0（ゼロ）のままにしてください。トランザクション内で、コンテナは、最初にトランザクション内で起動した Bean の max-tx-retries 値を使用します。パフォーマンスを向上させるために、この値を 0（ゼロ）のままにし、再試行によって解決できるエラーがある場合のみ、再試行の回数を追加することをお薦めします。
- tx-retry-wait －このパラメータは、トランザクションの再試行の時間間隔を秒数で指定します。デフォルトは 60 秒です。

- name － Bean の名前。これは、アセンブリ・ディスクリプタ（ejb-jar.xml）内の Bean の名前に一致します。
- pool-cache-timeout － Bean 実装インスタンスを、プールされた（割り当てられていない）状態で維持する秒単位の時間。'never' を指定すると、ガベージ・コレクションが行われるまでインスタンスを維持します。デフォルトは 60 です。
- table －コンテナ管理の永続性を使用している場合、データベース内の表の名前。
- validity-timeout －エンティティがキャッシュ内で有効である（再ロードされるまでの）最大の期間（ミリ秒単位）。既存システムからの更新がほとんど発生しない疎結合環境で役立ちます。この属性は、locking-mode が read_only の Entity Bean について、exclusive-write-access="true"（デフォルト）の場合にのみ有効です。

  外部でのデータ変更がない場合（したがって、exclusive-write-access=true に設定）は、0 または -1 に設定してこのオプションを使用禁止にすることをお薦めします。これは、外部での変更がない読取り専用の EJB の場合は、キャッシュ内のデータは常に有効であるためです。

  EJB の外部での変更は通常ない（したがって、exclusive-write-access=true に設定）が、表の更新があるため、キャッシュの更新がときどき必要な場合は、外部でのデータ変更の間隔に応じた値を設定します。
- update-changed-fields-only － ejbStore の起動時に、コンテナが、CMP Entity Bean の永続記憶域に対して変更されたフィールドのみ更新するか、または全フィールドを更新するかを指定します。デフォルトは true で、変更されたフィールドのみ更新されます。詳細は、9-8 ページの「永続性を更新する手法」を参照してください。
- wrapper －この Bean の OC4J リモート・ホーム・ラッパー・クラスの名前。（内部サーバー属性のため、編集しないでください）

<entity-ref>
主キーを通じてエンティティ参照を維持するための構成を指定。このタグの子タグは、主キーを維持するための指定です。

属性 :

- home － Bean をルックアップする EJBHome の JNDI の場所。

<env-entry-mapping>
アセンブリ・ディスクリプタの env-entry の値をオーバーライドします。EAR にデプロイ固有の値が指定されるのを防ぎます。本体が値です。

属性 :

- name －コンテキスト・パラメータの名前。

<fields>

このフィールドについて、フィールド・ベース（Java クラス・フィールド）のマッピングの永続性の構成を指定します。永続的にするフィールドは、パブリックで非静的および非ファイナルであり、さらに含まれているオブジェクトの型には空のコンストラクタが存在する必要があります。

<finder-method>

コンテナ管理による finder メソッドの定義。これにより、Bean のホームの `findByXXX()` メソッドの選択条件が定義されます。

属性 :

- partial －指定された問合せがパーシャルであるかどうか。パーシャルな問合せとは、SQL 問合せの 'where' 句または 'order'（order で始まる場合）句です。デフォルトでは、問合せはパーシャルです。partial="false" と指定した場合、query 属性の値として完全な問合せを入力する必要があり、問合せがすべての CMP フィールドを含んでいる結果セットを返すことを確認する必要があります。これは、表の結合などを行う高度な問合せを実行する際に便利です。
- query － SQL 文の問合せ部分。これは、文中の WHERE キーワードに続くセクションです。特殊なトークンは、メソッドの引数番号を示す $number、および cmp フィールド名を示す $name です。たとえば、"findByAge(int age)" の問合せは、cmp フィールドの名前が 'age' の場合、"$1 = $age" となります。
- lazy-loading － Entity Bean の finder メソッドの場合、遅延ロードによって select メソッドを複数回起動できます。遅延ロードをオンにし、この finder メソッドを 1 回のみ実行するには、このプロパティを true に設定します。デフォルトは false です。詳細は、9-7 ページの「CMP Entity Bean の finder メソッドにおける遅延ロードの構成」を参照してください。
- prefetch-size － Oracle JDBC Drivers には、問合せの過程で結果セットを移入する際にクライアントにプリフェッチする行数を設定できる拡張機能が含まれています。この機能を使用してデータをフェッチする際に複数のデータ行をフェッチすることによって、データベースへのラウンドトリップを削減できます。余分なデータは、後でクライアントがアクセスするためにクライアント側のバッファに格納されます。プリフェッチする行数は、自由に設定できます。クライアントにプリフェッチするデフォルトの行数は 10 です。ここに設定した行数は JDBC ドライバに渡されます。JDBC ドライバでプリフェッチを使用する方法の詳細は、『Oracle9*i* JDBC 開発者ガイドおよびリファレンス』を参照してください。

<group>

この `<security-role-mapping>` が示すグループ。つまり、指定されたグループの全メンバーがこのロールに含まれます。

属性 :

- name －グループの名前。

<ior-security-config>

<ior-security-config> 要素は、相互運用性の CSIv2 セキュリティ・ポリシーを構成します。詳細は、『Oracle Application Server Containers for J2EE サービス・ガイド』の相互運用性の章で説明されています。

<jem-deployment>

AC4J コンテナにデプロイするアクティブ EJB を指定します。

属性 :

- jem-name － AC4J コール内で Bean を識別するために使用する AC4J 名。
- ejb-name － ejb-jar.xml ファイルで定義した EJB をアクティブ EJB として識別します。

<jem-server-extension>

データ・バスがインストールされているデータベース・サーバーを示します。

属性 :

- data-source-location －データ・バスが存在するデータベースの JNDI データ・ソース定義を提供します。データ・ソースは、data-sources.xml ファイルで構成されます。
- scheduling-threads － 1 より大きい場合は、複数の OC4J スレッドがパラレルで動作できます。デフォルトは 1 です。

<lookup-context>

リソースの取得に使用される、オプションの javax.naming.Context 実装の仕様。サード・パーティ製の JMS サーバーなど、サード・パーティ製のモジュールを使用する場合、これが役立ちます。リソース・ベンダーが提供しているコンテキスト実装を使用するか、それが存在しない場合は、ベンダーのソフトウェアとネゴシエーションを行う実装を作成します。

属性 :

- location －リソースの取得時に外部キー・コンテキストで検索する名前。

<map-key-mapping>

マップ・キーのマッピングを指定します。マップ・キーは、常に不変です。

属性 :

- type －値の型の完全修飾クラス名。たとえば、com.acme.Product、java.lang.String などがあります。

<message-driven-deployment>
MDB のデプロイ情報。

属性 :

- connection-factory-location －使用するコネクション・ファクトリの JNDI の場所。JMS の Destination Connection Factory は、connection-factory-location 属性で指定されます。構文は、"java:comp/resource" + リソース・プロバイダ名 + "TopicConnectionFactories" または "QueueConnectionFactories" + ユーザー定義名です。xxxConnectionFactories は、定義するファクトリのタイプを指定します。
- destination-location －使用する接続先（キュー、トピック）の JNDI の場所。JMS の Destination は、destination-location 属性で指定されます。構文は、"java:comp/resource" + リソース・プロバイダ名 + "Topics" または "Queues" + Destination 名です。Topic または Queue は、定義される Destination タイプを指定します。Destination 名は、データベースで定義された実際のキュー名またはトピック名です。
- name － Bean の名前。これは、アセンブリ・ディスクリプタ（ejb-jar.xml）内の Bean の名前に一致します。
- subscription-name －これがトピックの場合、サブスクリプション名は subscription-name 属性で定義されます。
- listener-threads －リスナー・スレッドは、JMS メッセージを並行してコンシュームするために使用されます。デフォルトのスレッドは 1 つです。トピックに指定できるスレッドは 1 つのみです。キューには複数のスレッドを指定できます。
- transaction-timeout －この属性によって、コンテナ管理のトランザクション型 MDB のトランザクション・タイムアウト時間（秒）を制御します。デフォルトは 1 日（86,400 秒）です。この時間枠内でトランザクションが完了しない場合、トランザクションはロールバックされます。
- dequeue-retry-count －データベース・フェイルオーバーの発生後、リスナー・スレッドによって試行される JMS セッションの再取得頻度を指定します。この値は、MDB 内の CMT トランザクションに対してのみ有効です。デフォルトは 0（ゼロ）です。詳細は、7-37 ページの「RAC データベース使用時のフェイルオーバー」を参照してください。
- dequeue-retry-interval －再試行の間隔を指定します。デフォルトは 60 秒です。

<method>
Bean のメソッド（および場合によってはメソッドのパラメータ）を指定します。

<method-intf>
method-intf 要素により、method 要素が、リモート・インタフェースおよびホーム・インタフェースで定義されている、同じ名前およびシグネチャを持つメソッドを区別できるようになります。method-intf 要素は、Home または Remote のいずれかである必要があります。

<method-name>

method-name 要素には、Enterprise Bean メソッドの名前、またはアスタリスク（*）記号が含まれます。アスタリスクは、要素が、Enterprise Bean のリモート・インタフェースおよびホーム・インタフェースのすべてのメソッドを示す場合に使用されます。

<method-param>

method-param 要素には、メソッド・パラメータの完全修飾の Java タイプ名が含まれます。

<method-params>

method-params 要素には、メソッド・パラメータの完全修飾の Java タイプ名のリストが含まれます。

<orion-ejb-jar>

`orion-ejb-jar.xml` ファイルには、EJB の OC4J 固有のデプロイ情報が含まれています。初期のデプロイ・プロパティの指定に使用されます。毎回デプロイ後に、追加情報用にサーバーによってデプロイ・ファイルが再フォーマットされ、修正されます。

属性 :

- deployment-time －最終デプロイ時刻（10 進数による Long 型のミリ秒）。最終編集日と一致しない場合、JAR は再デプロイされます。（内部サーバー値のため、編集しないでください）
- deployment-version －この JAR のデプロイに使用された OC4J のバージョン。現在のバージョンと一致しない場合、再デプロイされます。（内部サーバー値のため、編集しないでください）

<primkey-mapping>

主キーのマッピング方式を指定します。

<properties>

このフィールドについて、プロパティ・ベース（Bean プロパティ）のマッピングの永続性の構成を指定します。プロパティは、通常の JavaBeans 仕様に従う必要があり、含まれているオブジェクトには、空のコンストラクタが存在する必要があります。これは、EJB 仕様でも指定されています。

<resource-ref-mapping>

`resource-ref` 要素は、データ・ソース、JMS キューまたはメール・セッションなどの外部リソースの参照の宣言に使用されます。`resource-ref-mapping` は、デプロイ時にこれを JNDI の場所に結合します。

属性 :

- location －リソース・ファクトリのルックアップ元の JNDI の場所。
- name － `resource-ref` の名前。`ejb-jar.xml` 内の `resource-ref` の名前に一致します。

<resource-env-ref-mapping>

resource-env-ref-mapping 要素は、リソースの管理オブジェクトをマッピングするために使用されます。たとえば、JMS を使用するために、Bean は JMS ファクトリ・オブジェクトと接続先オブジェクトの両方を取得する必要があります。これらのオブジェクトは、JNDI から同時に取得されます。<resource-ref> 要素で JMS ファクトリを宣言し、<resource-env-ref> 要素を使用して接続先を宣言します。したがって、<resource-env-ref-mapping> 要素は接続先オブジェクトをマッピングします。詳細は、7-33 ページの「クライアントが MDB にアクセスするときに論理名を使用する方法」を参照してください。

属性 :

- location ー管理リソースのルックアップ元の JNDI の場所。
- name ー ejb-jar.xml 内の resource-env-ref の名前。

<role-name>

<run-as-specified-identity> 要素を使用するとき、AC4J EJB メソッドを実行するセキュリティ・ロール。

<run-as-specified-identity>

AC4J EJB のすべてのメソッドが特定の識別情報を使用して実行されるように指定できます。つまり、コンテナは、特定のメソッドを実行する許可について別のロールをチェックせず、かわりに、指定されたセキュリティ識別情報を使用してすべての AC4J EJB メソッドを実行します。

<security-identity>

AC4J データ・バスで AC4J Bean セキュリティのコール元または run-as 識別情報を使用するかどうかを記述します。

<security-role-mapping>

グループおよびユーザーへの、ロールの実行時のマッピング。アセンブリ・ディスクリプタ内で、同じ名前の security-role にマッピングされます。

属性 :

- impliesAll ーこのマッピングが全ユーザーを含めるかどうか。デフォルトは false です。
- name ーロールの名前。

<session-deployment>

セッション Bean のデプロイ情報。

属性 :

- pool-cache-timeout ープールにキャッシュされたステートレス・セッションを維持する期間。ステートレス Session Bean にのみ適用されます。正しい値は、正の整数値または 'never' です。ステートレス Session Bean の場合、pool-cache-timeout を指定すると、pool-cache-timeout ごとに、プール内の対応する Bean タイプの Bean がすべて削除され

ます。値が 0（ゼロ）または負の場合は、pool-cache-timeout が無効になり、Bean はプールから削除されません。

デフォルト値は 60（秒）です。

- call-timeout － EJB が使用中の場合、EJB で使用するリソース（データベース接続を除く）の待機時間（10 進数による Long 型のミリ秒）。タイムアウト後、`RemoteException` がスローされ、EJB はデッドロックにあるとみなされます。値が 0 に設定されている場合、OC4J は永久に EJB を待機します。これはデフォルトです。
- copy-by-value － EJB コールのすべての受信および送信パラメータをコピーする（クローンを作成する）かどうか。速度を上げるには、使用するアプリケーションが copy-by-value セマンティクスを前提としないことが確実な場合、この値を 'false' に設定します。デフォルトは 'true' です。
- local-wrapper －この Bean の OC4J ラッパー・クラスの名前。内部サーバー値のため、編集しないでください。
- location －この Bean がバインドされる JNDI 名。

max-instances －メモリー内に存在できるインスタンス化またはプールされた Bean インスタンスの数を制御します。この値に達すると、コンテナは最も古い Bean インスタンスをメモリーから非アクティブ化しようとします。非アクティブ化に失敗した場合、コンテナは `call-timeout` 属性に設定されたミリ秒数待機して、非アクティブ化、`remove()` メソッドまたは Bean の期限切れのいずれかによって、メモリーから削除された Bean インスタンスがあるかどうかを確認し、その後で `TimeoutExpiredException` をクライアントにスローします。Bean インスタンスの数を無限に許可する場合は、`max-instances` 属性を 0（ゼロ）に設定できます。デフォルトは 0（ゼロ）で、無限を意味します。この属性は、ステートレス Session Bean およびステートフル Session Bean の両方に適用されます。

- max-instances-threshold － max-instances 属性の定義に応じて、存在するアクティブ Bean 数に対するしきい値を定義します。パーセンテージとして解釈される整数を指定します。max-instances を 100、max-instances-threshold を 90% に定義した場合は、アクティブ Bean インスタンスの数が 90 を超えると、Bean の非アクティブ化が発生します。デフォルトは 90% です。この属性を無効にするには、"never" を指定します。詳細は、9-3 ページの「EJB のライフ・サイクルに関する問題」を参照してください。
- max-tx-retries －システム・レベルの障害によってロールバックされたトランザクションの再試行回数。デフォルトは 0（ゼロ）です。トランザクション内で、コンテナは、最初にトランザクション内で起動した Bean の max-tx-retries 値を使用します。パフォーマンスを向上させるために、この値を 0（ゼロ）のままにし、再試行によって解決できるエラーがある場合のみ、再試行の回数を追加することをお薦めします。
- tx-retry-wait －このパラメータは、トランザクションの再試行の時間間隔を秒数で指定します。デフォルトは 60 秒です。

- memory-threshold －非アクティブ化が発生するまでに使用可能な JVM メモリーの量に対するしきい値を定義します。パーセンテージとして解釈される整数を指定します。この値に達すると、アイドル・タイムアウトが経過していない場合でも Bean は非アクティブ化されます。デフォルトは 80% です。この属性を無効にするには、"never" を指定します。詳細は、9-3 ページの「EJB のライフ・サイクルに関する問題」を参照してください。
- min-instances －インスタンス化またはプールされた状態で維持される Bean 実装インスタンスの最小数。デフォルトは 0（ゼロ）です。この属性は、ステートレス Session Bean にのみ適用されます。
- name － Bean の名前。これは、EJB デプロイメント・ディスクリプタのアセンブリ・セクション（`ejb-jar.xml`）内の Bean の名前に一致します。
- resource-check-interval －コンテナは、すべてのリソースをこの時間間隔でチェックします。この時点でいずれかのしきい値に達している場合は、非アクティブ化が発生します。デフォルトは 180 秒（3 分）です。この属性を無効にするには、"never" を指定します。詳細は、9-3 ページの「EJB のライフ・サイクルに関する問題」を参照してください。
- passivate-count －いずれかのリソースしきい値に達した場合に非アクティブ化される Bean の数を定義する整数です。Bean の非アクティブ化は、最低使用頻度アルゴリズムを使用して実行されます。デフォルトは、`max-instances` 属性の 1/3 です。この属性を無効にするには、カウントを 0（ゼロ）または負の数に設定します。詳細は、9-3 ページの「EJB のライフ・サイクルに関する問題」を参照してください。
- persistence-filename －再起動のたびにセッションが保存されるファイルへのパス。
- timeout －非アクティブな場合のタイムアウト（秒単位）。値が 0（ゼロ）または負の数の場合、すべてのタイムアウトが使用禁止になります。デフォルトは 30 分です。30 秒ごとに、プール・クリーンアップ・ロジックが起動します。プール・クリーンアップ・ロジックの実行中に、タイムアウト値を渡すことによって削除されるのは、タイムアウトしたセッションのみです。

  アプリケーションでのステートフル Session Bean の使用状況に応じて、タイムアウトを調整します。たとえば、ステートフル Session Bean を明示的に削除しないアプリケーションの場合は、多数のステートフル Session Bean が作成されるため、タイムアウト値を短い時間に設定できます。

  アプリケーションでステートフル Session Bean を 30 分より長く使用可能にする必要がある場合は、タイムアウト値をそれにあわせて調整します。
- wrapper －この Bean の OC4J ラッパー・クラスの名前。内部サーバー値のため、編集しないでください。

<set-mapping>
Set 型のリレーショナル・マッピングを指定します。Set は、n 個の一意の順序付けされていない（順序が指定されておらず、必要ない）アイテムで構成されます。マッピングを含んでいるフィールドは、java.util.Set 型である必要があります。

属性 :

- table －データベース内の表の名前。

<use-caller-identity>
AC4J EJB のすべてのメソッドがコール元の識別情報を使用して実行されるように指定できます。

<user>
この security-role-mapping によって示されるユーザー。

属性 :

- name －ユーザーの名前。

<value-mapping>
一連のフィールドの主キー部分のマッピングを指定します。

属性 :

- immutable －値が `Collection` に追加された後、必ず不変となるかどうか。この値を true に設定すると、データベース処理が大幅に最適化されます。デフォルト値は、set-mapping の場合は "true"、collection-mapping の場合は "false" です。
- type －値の型の完全修飾クラス名。たとえば、`com.acme.OrderEntry`、`java.lang.String` などがあります。

B

EJB 1.1 CMP Entity Bean

この付録では、以前のリリースの EJB 1.1 CMP Entity Bean を使用する場合、OC4J で EJB 1.1 CMP デプロイメント・ディスクリプタ要素を OC4J 固有のマッピングにマッピングする方法を説明します。CMP Entity Bean については、EJB 2.0 メソッドに移行することをお薦めしますが、Oracle では両方の仕様をサポートしています。

この章では、基本的な構成およびデプロイを使用した単純な CMP EJB 1.1 の開発方法を説明します。CMP Entity Bean の例は、OTN-J のサイト `http://otn.oracle.co.jp/sample_code/index.html` の OC4J のサンプル・コードのページからダウンロードしてください。

この章では、次の内容を説明します。

- Entity Bean の作成: 単純なコンテナ管理による永続的な Entity Bean の作成方法を説明します。
- 高度な CMP Entity Bean: finder メソッド、オブジェクト・リレーショナル・マッピングなどの高度な構成を説明します。

EJB 2.0 CMP Entity Bean の詳細は、第 3 章「CMP Entity Bean」を参照してください。

Entity Bean の作成

Entity Bean を作成するには、次の手順を実行します。

1. Bean のリモート・インタフェースを作成します。リモート・インタフェースは、クライアントによって起動可能なメソッドを宣言します。ここでは、`javax.ejb.EJBObject` を拡張する必要があります。
2. Bean のホーム・インタフェースを作成します。ホーム・インタフェースでは、`javax.ejb.EJBHome` を拡張する必要があります。ここでは、`findByPrimaryKey` を含め、作成する Bean の `create` および finder メソッドを定義します。
3. Bean の主キーを定義します。主キーにより、各 Entity Bean のインスタンスを識別します。主キーは、`java.lang.String` などの一般的なクラスにするか、または自らのクラス内で定義されている必要があります。
4. Bean を実装します。次のものが含まれます。
   a. リモート・インタフェースで宣言されているメソッドの実装。
   b. `javax.ejb.EntityBean` インタフェースで定義されているメソッド。
   c. ホーム・インタフェースで宣言されているメソッドに一致するメソッド。次のものが含まれます。
      * ホーム・インタフェースで定義された対応する `create` メソッドのパラメータに一致するパラメータを持つ `ejbCreate` および `ejbPostCreate` メソッド
      * ホーム・インタフェースの `findByPrimaryKey` メソッドに対応する `ejbFindByPrimarykey` メソッド
      * ホーム・インタフェースで定義されたその他の finder メソッド
5. Bean のデプロイメント・ディスクリプタを作成します。デプロイメント・ディスクリプタにより、XML 要素を通じて Bean のプロパティを指定します。このステップで、コンテナによって管理する Bean 内のデータを指定します。
6. 永続データをデータベースに格納またはリストアする際にコンテナのデフォルトを使用しない場合、Bean に対して正しい表が存在することを確認する必要があります。すべてデフォルトを使用する場合は、コンテナにより、デプロイメント・ディスクリプタおよびデータソース情報に基づいてデータ用の表と列が作成されます。
7. Bean、リモート・インタフェース、ホーム・インタフェースおよびデプロイメント・ディスクリプタを含める EJB JAR ファイルを作成します。作成した後、`application.xml` ファイルを構成し、EAR ファイルを作成し、EJB を OC4J にインストールします。

次の項で、単純な CMP Entity Bean を説明します。この例では、わかりやすいように、他の章と同様に引き続き employee の例を使用します。

ホーム・インタフェース

ホーム・インタフェースには、クライアントが Bean のインスタンスを作成するために起動する create メソッドが含まれている必要があります。各 create メソッドには異なるシグネチャを使用できます。Entity Bean の場合、findByPrimaryKey メソッドを開発する必要があります。オプションで、Bean 用に他の finder メソッドも開発可能です。これらは、find<*name*> のように名前を付けます。

例 B-1 Entity Bean Employee のホーム・インタフェース

Entity Bean を説明するため、この例では従業員を管理する Bean を作成します。Entity Bean には、従業員情報が含まれます。

ホーム・インタフェースでは、javax.ejb.EJBHome を拡張し、create および findByPrimaryKey メソッドを定義します。

```
package employee;

import javax.ejb.*;
import java.rmi.*;

public interface EmployeeHome extends EJBHome
{

  public Employee create(Integer empNo)
    throws CreateException, RemoteException;

  // Find an existing employee
  public Employee findByPrimaryKey (Integer empNo)
    throws FinderException, RemoteException;

  //Find all employees
  public Collection findAll()
    throws FinderException, RemoteException;
}
```

リモート・インタフェース

Entity Bean のリモート・インタフェースは、クライアントに表示され、クライアントがメソッドを起動するインタフェースです。javax.ejb.EJBObject を拡張し、ビジネス・ロジック・メソッドを定義します。employee という Entity Bean の場合、リモート・インタフェースには、従業員情報の取得と設定を行うメソッドが含まれています。

```
package employee;

import javax.ejb.*;
import java.rmi.*;
import java.util.*;

public interface Employee extends EJBObject
{
  // getter remote methods
  public Integer getEmpNo() throws RemoteException;
  public String getEmpName() throws RemoteException;
  public Float getSalary() throws RemoteException;

  // setter remote methods
  public void setEmpName(String newEmpName) throws RemoteException;
  public void setSalary(Float newSalary) throws RemoteException;
}
```

Entity Bean クラス

Entity Bean クラスでは、次のメソッドを実装する必要があります。

- ホーム・インタフェースで宣言されているメソッドのターゲット・メソッド。ejbCreate メソッド、および存在する場合は ejbFindByPrimaryKey などの finder メソッドが含まれます。
- リモート・インタフェースで宣言されたビジネス・ロジック・メソッド。
- EntityBean インタフェースを継承するメソッド。

ただし、コンテナ管理の永続性を使用する場合、コンテナによってほとんどのターゲット・メソッドおよびデータ・オブジェクトが管理されます。そのため、ユーザーが実装するものはあまりありません。

```
package employee;

import javax.ejb.*;
import java.rmi.*;

public class EmployeeBean extends Object implements EntityBean
{
```

```
public Integer empNo;
public String empName;
public Float salary;
public EntityContext entityContext;

public EmployeeBean()
{
  // Constructor. Do not initialize anything in this method.
  // All initialization should be performed in the ejbCreate method.
}

public Integer getEmpNo()
{
  return empNo;
}

public String getEmpName()
{
  return empName;
}

public Float getSalary()
{
  return salary;
}

public void setEmpName(String empName)
{
  this.empName = empName;
}

public void setSalary(Float salary) {
  this.salary = salary;
}

public Integer ejbCreate(Integer empNo)
  throws CreateException, RemoteException
{
  this.empNo = empNo;
  return empNo;
}

public void ejbPostCreate(Integer empNo)
  throws CreateException, RemoteException
{
  // Called just after bean created; container takes care of implementation
}
```

```
  public void ejbStore()
  {
    // Called when bean persisted; container takes care of implementation
  }

  public void ejbLoad()
  {
    // Called when bean loaded; container takes care of implementation
  }

  public void ejbRemove()
  {
    // Called when bean removed; container takes care of implementation
  }

  public void ejbActivate()
  {
    // Called when bean activated; container takes care of implementation.
    // If you need resources, retrieve them here.
  }

  public void ejbPassivate()
  {
    // Called when bean deactivated; container takes care of implementation.
    // if you set resources in ejbActivate, remove them here.
  }

  public void setEntityContext(EntityContext entityContext)
  {
    this.entityContext = entityContext;
  }

  public void unsetEntityContext()
  {
    this.entityContext = null;
  }
}
```

永続データ

CMP Entity Bean の場合、永続データを Bean のインスタンスおよびデプロイメント・ディスクリプタの両方で定義します。Bean のインスタンスのデータ・フィールド宣言により、フィールドのリソースが作成されます。デプロイメント・ディスクリプタにより、これらのフィールドが永続的であると定義されます。

employee の例では、データ・フィールドは、次のように Bean のインスタンスで定義されています。

```
public Integer empNo;
public String empName;
public Float salary;
```

これらのフィールドは、次のように、<cmp-field><field-name> 要素内の ejb-jar.xml デプロイメント・ディスクリプタで永続フィールドとして定義されています。

```
<enterprise-beans>
      <entity>
         <display-name>Employee</display-name>
         <ejb-name>EmployeeBean</ejb-name>
         <home>employee.EmployeeHome</home>
         <remote>employee.Employee</remote>
         <ejb-class>employee.EmployeeBean</ejb-class>
         <persistence-type>Container</persistence-type>
         <prim-key-class>java.lang.Integer</prim-key-class>
         <reentrant>False</reentrant>
         <cmp-field><field-name>empNo</field-name></cmp-field>
         <cmp-field><field-name>empName</field-name></cmp-field>
         <cmp-field><field-name>salary</field-name></cmp-field>
         <primkey-field>empNo</primkey-field>
      </entity>
...
</enterprise-beans>
```

ほとんどの場合、指定したデータベースに存在する表内の列に永続データ・フィールドをマッピングします。ただし、これらのフィールドのデフォルトを使用すると、その他のデプロイメント・ディスクリプタの構成を行う必要がありません。

OC4J には、これらのフィールドを、データベースとそこに含まれる表へマッピングするためのデフォルトがいくつか用意されています。

- データベース : 使用中の OC4J インスタンス構成に設定されているデフォルトのデータベース。JNDI 名については、エミュレートされたデータ・ソースの場合は <location> 要素を、エミュレートされていないデータ・ソースの場合は <ejb-location> 要素を使用します。

インストール後の状態では、デフォルトのデータベースは、ローカルにインストールされた Oracle データベースで、ポート 1521 でリスニングし、SID が ORCL である必要があります。デフォルトのデータベースをカスタマイズするには、最初に構成されたデータベース（<ejb-location> を含め）を、使用するデータベースに変更します。

- 正しい列名を持つ表 : コンテナは、指定されたデータベース内に、Bean 名（<ejb-name> で定義）と同じ名前のデフォルトの表を作成し、その列に <cmp-field> 要素と同じ名前を付けます。データベースのデータ型は、特定のデータベースの XML ファイル（oracle.xml など）で定義されています。Java のデータ型はデータベースのデータ型に変換されます。

別のデータベースを指定したり、別の命名規則を持つ表を生成する場合は、B-14 ページの「EJB 1.1 の永続フィールドのオブジェクト・リレーショナル・マッピング」に示されているデータベース、表および列名のカスタマイズ方法を参照してください。

主キー

各 Entity Bean には、他のインスタンスから一意に識別するための主キーが存在します。主キー（または主キーとなる複合キー内のフィールド）を、デプロイメント・ディスクリプタのコンテナ管理による永続的フィールドとして宣言する必要があります。主キー内のすべてのフィールドは、プリミティブ型、シリアライズ可能な型、または SQL 型にマッピング可能な型に制限されています。主キーは、次のいずれかの方法で定義します。

- 主キーに、一般的な型を定義します。型は、デプロイメント・ディスクリプタの <prim-key-class> で定義されます。永続的な主キーとして識別されるデータ・フィールドは、デプロイメント・ディスクリプタの <primkey-field> 要素で識別されます。Bean クラス内で宣言される主キー変数は、public として宣言する必要があります。
- 主キーの型を、シリアライズ可能な *<name>*PK クラス内のシリアライズ可能なオブジェクトとして定義します。このクラスは、デプロイメント・ディスクリプタの <prim-key-class> で宣言されます。これは、主キーの高度な定義方法であるため、これについては B-9 ページの「クラス内での主キーの定義」で説明します。

単純な CMP の場合、デプロイメント・ディスクリプタ内の主キーのデータ型を定義することにより、主キーに一般的なデータ型を定義できます。

employee の例では、主キーを java.lang.Integer として定義し、従業員番号（empNo）を主キーとして使用しています。

```
<enterprise-beans>
      <entity>
         <display-name>Employee</display-name>
         <ejb-name>EmployeeBean</ejb-name>
         <home>employee.EmployeeHome</home>
         <remote>employee.Employee</remote>
         <ejb-class>employee.EmployeeBean</ejb-class>
         <persistence-type>Container</persistence-type>
```

```
            <prim-key-class>java.lang.Integer</prim-key-class>
            <reentrant>False</reentrant>
            <cmp-field><field-name>empNo</field-name></cmp-field>
            <cmp-field><field-name>empName</field-name></cmp-field>
            <cmp-field><field-name>salary</field-name></cmp-field>
            <primkey-field>empNo</primkey-field>
        </entity>
...
</enterprise-beans>
```

クラス内での主キーの定義

主キーが単純なデータ型ではなく複合キーの場合、主キーは、シリアライズ可能なクラスで、名前を *<name>*PK にする必要があります。主キー・クラスは、デプロイメント・ディスクリプタの <prim-key-class> で定義します。

主キー変数は、次の規則に従う必要があります。

- デプロイメント・ディスクリプタの <cmp-field><field-name> 要素内で定義されていること。これにより、コンテナから主キー・フィールドを管理できるようになります。
- Bean クラス内で public として宣言され、プリミティブ型、シリアライズ可能、または SQL 型にマッピング可能な型のいずれかに制限されていること。

主キー・クラス内で、主キーのインスタンスを作成するコンストラクタを実装します。このように定義すると、コンテナにより主キーの管理および永続データの格納が行われます。

次の例では、従業員番号と国コードで構成される複合主キーを示します。この企業は大企業であるため、異なる国で同じ従業員番号を利用しています。そのため、従業員番号と国コードの組合せにより、各従業員が一意に識別されます。

```
package employee;

public class EmpPK implements java.io.Serializable
{
  public Integer empNo;
  public String countryCode;

  //constructor
  public EmpPK ( ) { }
}
```

主キー・クラスは、次のように、それぞれ XML デプロイメント・ディスクリプタ内の <prim-key-class> 要素およびその変数（<cmp-field><field-name> 要素内で）で宣言されます。

```
<enterprise-beans>
      <entity>
         <display-name>Employee</display-name>
         <ejb-name>EmployeeBean</ejb-name>
         <home>employee.EmployeeHome</home>
         <remote>employee.Employee</remote>
         <ejb-class>employee.EmployeeBean</ejb-class>
         <persistence-type>Container</persistence-type>
         <prim-key-class>employee.EmpPK</prim-key-class>
         <reentrant>False</reentrant>
         <cmp-field><field-name>empNo</field-name></cmp-field>
         <cmp-field><field-name>countryCode</field-name></cmp-field>
      </entity>
      ...
</enterprise-beans>
```

Entity Bean のデプロイ

EJB を JAR ファイルにアーカイブします。Entity Bean は、Session Bean と同様にデプロイします。これについては、2-26 ページの「EJB アプリケーションのデプロイ準備」および 2-28 ページの「エンタープライズ・アプリケーションの OC4J へのデプロイ」で詳細に説明されています。

高度な CMP Entity Bean

この項では、単純な CMP Entity Bean に比べ、より高度な Bean の実装方法について説明します。次の項が含まれます。

- EJB 1.1 の高度な finder メソッド
- EJB 1.1 の永続フィールドのオブジェクト・リレーショナル・マッピング

EJB 1.1 の高度な finder メソッド

`findByPrimaryKey` メソッドの指定は、OC4J では簡単です。単純な主キー、または複合主キーを定義するフィールドは、すべて `ejb-jar.xml` デプロイメント・ディスクリプタで指定されます。ただし、その他の finder メソッドを CMP Entity Bean で定義する場合、次の作業を行う必要があります。

1. finder メソッドをホーム・インタフェースに追加します。
2. EJB 1.1 の finder メソッドの定義を、OC4J 固有のデプロイメント・ディスクリプタ、つまり `orion-ejb-jar.xml` ファイルに追加します。

finder メソッドのホーム・インタフェースへの追加

まず、finder メソッドをホーム・インタフェースに追加する必要があります。たとえば、employee Entity Bean の場合、すべての従業員を取得するには、`findAll` メソッドをホーム・インタフェースで次のように定義します。

```
public Collection findAll() throws FinderException, RemoteException;
```

OC4J デプロイメント・ディスクリプタへの EJB 1.1 の finder メソッド定義の追加

finder メソッドをホーム・インタフェースで指定した後、EJB 1.1 の finder メソッドの仕様に基づいて `orion-ejb-jar.xml` ファイルを修正します。コンテナは、必要なフィールドの取得に必要な正しい問合せを識別します。

EJB 1.1 の `<finder-method>` 要素は、`findByPrimaryKey` メソッド以外のすべての finder メソッドを定義します。定義が最も単純な finder メソッドは、`findByAll` メソッドです。`<finder-method>` 要素の `query` 属性は、問合せの `WHERE` 句を指定します。すべての行を取得する場合、空の問合せ（`query=""`）によってすべてのレコードが返されます。

次の例では、`EmployeeBean` からすべてのレコードを取得します。メソッド名は `findAll` で、すべての従業員の `Collection` を返すため、パラメータは必要ありません。

```
<finder-method query="">
  <method>
     <ejb-name>EmployeeBean</ejb-name>
     <method-name>findAll</method-name>
         <method-params></method-params>
     </method>
</finder-method>
```

この Bean でアプリケーションをデプロイした後、OC4J は、finder メソッド定義で、起動する問合せ文をコメントとして追加します。

```
<finder-method query="">
<!-- Generated SQL: "select EmployeeBean.empNo, EmployeeBean.empName,
    EmployeeBean.salary from EmployeeBean" -->
  <method>
     <ejb-name>EmployeeBean</ejb-name>
     <method-name>findAll</method-name>
         <method-params></method-params>
     </method>
</finder-method>
```

問合せのタイプが正しいことを確認してください。

より具体的な問合せを行うには、query 属性に適切な WHERE 句を追加します。この句では、$ 記号を使用して渡されたパラメータを参照します。最初のパラメータは $1 で示され、2 番目のパラメータは $2 で示されます。WHERE 句内で使用されるすべての <cmp-field> 要素は、$<cmp-field> 名で示されます。

次の例では、findByName メソッド（ホーム・インタフェースで定義する）を指定します。従業員の名前がメソッド・パラメータとして渡されており、$1 に置換されています。これは、CMP 名 "empName" に一致しています。このように、query 属性は、WHERE 句について、"$empname=$1" を含めるよう変更されています。

```
<finder-method query="$empname = $1">
  <method>
    <ejb-name>EmployeeBean</ejb-name>
    <method-name>findByName</method-name>
    <method-params>
      <method-param>java.lang.String</method-param>
    </method-params>
  </method>
</finder-method>
```

メソッド・パラメータが複数存在する場合、各パラメータ型は連続した <method-param> 要素で定義され、問合せ文では、連続した $*n* で示されます。*n* は番号を示します。

注意： query 属性で SQL JOIN を指定することも可能です。

WHERE 句の後のセクションだけではなく、完全な問合せ文を指定する場合、partial 属性を FALSE に指定し、完全な問合せ文を query 属性で定義します。partial のデフォルト値は true であるため、前の finder メソッドの例では指定されていません。

```
<finder-method partial="false"
               query="select * from EMP where $empName = $1">
        <!-- Generated SQL: "select * from EMP where EMP.ENAME = ?" -->
        <method>
                <ejb-name>EmployeeBean</ejb-name>
                <method-name>findByName</method-name>
                <method-params>
                        <method-param>java.lang.String</method-param>
                </method-params>
        </method>
</finder-method>
```

完全な SQL 問合せ文の指定は、複合 SQL 文の場合に役立ちます。

Entity Bean の finder メソッドの場合、遅延ロードによって select メソッドを複数回起動できます。遅延ロードをオンにし、この finder メソッドを 1 回のみ実行するには、lazy-loading プロパティを true に設定します。

```
<finder-method partial="false"
               query="select * from EMP where $empName = $1"
               lazy-loading=true>
        <!-- Generated SQL: "select * from EMP where EMP.ENAME = ?" -->
        <method>
                <ejb-name>EmployeeBean</ejb-name>
                <method-name>findByName</method-name>
                <method-params>
                        <method-param>java.lang.String</method-param>
                </method-params>
        </method>
</finder-method>
```

EJB 1.1 の永続フィールドのオブジェクト・リレーショナル・マッピング

B-7 ページの「永続データ」で説明したように、永続データは、コンテナによって自動的にデータベース表にマッピング可能です。ただし、Bean が指すデータがこれより複雑であるか、OC4J のデフォルトを使用しない場合、orion-ejb-jar.xml ファイルで、CMP 指定フィールドを既存のデータベース表と該当する行にマッピングします。いったんマッピングすると、コンテナにより、CMP データの永続記憶域が、指定された表および行に用意されます。

オブジェクト・リレーショナル・マッピングを構成する前に、格納先に使用した DataSource を ejb-jar.xml ファイルの <resource-ref> 要素に追加します。

データベース表および列への EJB 1.1 CMP フィールドのマッピング

orion-ejb-jar.xml ファイルで、次のものを構成します。

1. マッピングされる CMP フィールドを持つ各 Entity Bean について、<entity-deployment> 要素を構成します。
2. Bean 内の、マッピングされる各フィールドについて、<cmp-field-mapping> 要素を構成します。各 <cmp-field-mapping> 要素は、永続的にするフィールドの名前を含んでいる必要があります。
   a. <cmp-field-mapping> 要素内に存在する <primkey-mapping> 要素の主キーを構成します。
   b. 1 つの <cmp-field-mapping> 要素内の 1 つのフィールドにマッピングされる単純なデータ型（プリミティブ、単純なオブジェクトまたはシリアライズ可能なオブジェクト）を構成します。名前およびデータベース・フィールドは、要素属性で完全に定義されています。
   c. <cmp-field-mapping> 要素の多数のサブ要素の 1 つを使用して、複合データ型を構成します。次のいずれかです。
      * オブジェクトを複合データ型として定義する場合、オブジェクト内の各フィールドまたはプロパティを <fields> または <properties> 要素で指定します。
      * 別の Entity Bean で定義されているフィールドを指定する場合、この Entity Bean のホーム・インタフェースを <entity-ref> 要素で定義します。
      * フィールドの Collection または Set を定義する場合は、これらのフィールドを <collection-mapping> 要素、<set-mapping> 要素で定義します。

C

EJB 1.1 から EJB 2.0 へのコンテナ管理の永続性の移行

Oracle9*i*AS リリース 2（9.0.3）より前のコンテナ管理の永続性と関連性は、Oracle 固有のデプロイメント・ディスクリプタで定義されていました。このディスクリプタには、当時は完全に設計が終了していなかった EJB 2.0 の機能に対し Oracle 固有の実装が使用されていました。たとえば、finder メソッドは、SQL 文を使用して `orion-ejb-jar.xml` ファイルに定義していましたが、現在は、EJBQL SQL を使用して `ejb-jar.xml` ファイル内に定義しています。次の各項では、Oracle 固有の EJB 2.0 機能を実際の EJB 2.0 仕様の方法論に移行する方法を説明します。

- EJB 2.0 への EJB 1.1 アプリケーションの移行に関する概要
- EJB 2.0 デプロイメント・ディスクリプタ ID の使用
- 抽象的な Bean 実装の使用
- 標準の EJB 2.0 関連の使用
- 関連を持つ Bean に対するローカル・インタフェースの使用
- EJB 問合せ言語（EJBQL）の使用

EJB 2.0 への EJB 1.1 アプリケーションの移行に関する概要

EJB 2.0 仕様における最も重要な変更は、コンテナ管理の永続性（CMP）とコンテナ管理の関連（CMR）の両方に対するものです。Oracle9*i*AS リリース 2（9.0.3）から、OC4J は J2EE 1.3 に準拠し、EJB 2.0 仕様を完全に実装しています。以前のバージョンの Oracle9*i*AS では、EJB 2.0 仕様が完成するまで、一部の EJB 2.0 機能に対し Oracle 固有の実装を提供していました。EJB 2.0 仕様が完成した現在、Oracle 固有のプレビュー機能を使用しているアプリケーションは、EJB 2.0 の機能を利用するように変更する必要があります。

この章では、Oracle 固有のプレビュー機能を使用している EJB アプリケーションを、EJB 2.0 の標準機能（特に、EJB 2.0 の CMP 機能と CMR 機能）を利用するように移行する方法を説明します。この章を読む必要があるのは、CMP ベースの Entity Bean および次の機能のいずれかを使用していた場合のみです。

- Entity Bean 間の関連
- 依存オブジェクトとデータベース表のマッピング

EJB 2.0 デプロイメント・ディスクリプタ ID の使用

次の変更は、EJB 2.0 機能を使用しているすべてのアプリケーションに対する基本的な変更です。

1. 使用している EJB デプロイメント・ディスクリプタを変更します。

   EJB 2.0 を使用するには、EJB デプロイメント・ディスクリプタに次の変更が必要です。

   a. `DOCTYPE` タグを次のように変更し、ドキュメントを検証する DTD に EJB 2.0 を指定します。

      ```
      <!DOCTYPE ejb-jarPUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
      2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
      ```

      これは、EJB 2.0 DTD を使用して `ejb-jar.xml` ファイルを検証することを XML パーサーに示しています。

   b. EJB 2.0 CMP メカニズムを使用するように EJB コンテナを構成します。`<entity>` 要素のボディ内で、`<cmp-version>` 要素を 2.x に指定します。

      ```
      <entity>
        ...
        <cmp-version>2.x</cmp-version>
        ...
      </entity>
      ```

c. Bean の抽象スキーマ名を指定します。この名前は、EJBQL 問合せで Bean を参照するために使用されます。

```
<entity>
  ...
  <abstract-schema-name>Topping</abstract-schema-name>
  ...
</entity>
```

抽象的な Bean 実装の使用

EJB 2.0 では、CMP を使用する Entity Bean は、抽象的な Bean として定義されます。指定する必要があるのは、Bean クラスの定義のみで、コードの実装は不要です。デプロイメント時に、コンテナは、ユーザーが Bean に定義したデプロイメント・ディスクリプタとメソッドを参照して、抽象的な Bean の定義に必要な実装クラスを生成します。コンテナで生成された実装によって、Bean の状態の保持に必要なインスタンス・フィールドおよび Bean の状態の操作に使用するアクセッサ・メソッドが定義されます。このように、コンテナは、Bean について実行される操作に深く関わっているため、Bean を管理および制御する方法を最適化する立場にあります。

既存の Bean クラスを抽象化する手順は、次のとおりです。

1. 抽象クラスとして宣言されるように、Bean クラスの定義を変更します。

```
public class EmployeeBean implements javax.ejb.EntityBean
{
  ..
}
```

変更後の定義は次のとおりです。

```
public abstract class EmployeeBean implements javax.ejb.EntityBean
{
  ..
}
```

2. Entity Bean の状態の保持に使用しているインスタンス・フィールドを抽象アクセッサ・メソッドで置換します。メソッドのシグネチャを変更して抽象キーワードを追加し、コードをメソッドのボディから削除します。コンテナによって、インスタンス・フィールドの値の取得と設定に必要なコードが生成されます。

たとえば、age フィールドに対する標準の EJB 1.1 アクセッサ・モデルは、次のとおりです。

```
public int getAge()
{
  return _age;
}
```

```
public void setAge(int age)
{
  this._age = age;
}
```

EJB 2.0 アクセッサ・モデルでは、次のようになります。

```
public abstract int getAge();
public abstract setAge(int age);
```

3. Bean 実装における非アクセッサ・メソッドについては、インスタンス・フィールドに直接アクセスするコードを変更して、対応する取得または設定のアクセッサ・メソッドをかわりに使用します。インスタンス・フィールドは Bean クラスに直接定義されていないため、このフィールドを、Bean の状態の取得と操作に使用することはできません。Bean フィールドへのアクセスには、かわりにアクセッサ・メソッドを使用する必要があります。

標準の EJB 2.0 関連の使用

Oracle9*i*AS リリース 2（9.0.3）より前は、オブジェクト・モデル内の関連は、初期段階の EJB 2.0 仕様に基づいた Oracle 固有のメカニズムを使用してサポートされていました。単純な 1 対多の関連を Bean 実装クラス内でプログラムできました。また、Oracle 固有の EJB デプロイメント・ディスクリプタを構成して、依存オブジェクト、他の Entity Bean またはオブジェクトのコレクションを使用した複雑なオブジェクト・モデルを構成できました。いずれのアプローチを使用しても、移植可能な EJB アプリケーションは作成されませんでした。

このリリースの OC4J には、完全準拠の実装が用意されています。したがって、Oracle 固有の関連メカニズムを使用しているコードは、標準の EJB 2.0 メカニズムを使用するように移行する必要があります。

Oracle 固有の関連は、`orion-ejb-jar.xml` ファイルの `<cmp-field-mapping>` 要素を使用して定義されています。各関連を削除し、`ejb-jar.xml` ファイルの `<relationship>` 要素を使用して再度定義する必要があります。EJB 2.0 仕様では、2 つのオブジェクト（Entity Bean）間の関連は、標準の `ejb-jar.xml` デプロイメント・ディスクリプタの `<relationship>` 要素を使用して定義されます。この関連は、EJB コンテナによって自動的に管理され、EJB 2.0 をサポートする J2EE 製品にデプロイできます。

次の例は、1 対多関連の EJB 2.0 構成です。1 つの部門に複数の従業員が所属しています。部門は `DeptBean` で実装され、従業員は `EmpBean` 内で実装されます。コンテナ管理の永続性の詳細は、第 3 章「CMP Entity Bean」を、関連の詳細は、第 4 章「エンティティ関連（E-R）のマッピング」を参照してください。

```
<ejb-relation>
 <ejb-relation-name>Dept-Emps</ejb-relation-name>
 <ejb-relationship-role>
  <ejb-relationship-role-name>Dept-has-Emps</ejb-relationship-role-name>
  <multiplicity>One</multiplicity>
  <relationship-role-source>
    <ejb-name>DeptBean</ejb-name>
  </relationship-role-source>
  <cmr-field>
    <cmr-field-name>employees</cmr-field-name>
    <cmr-field-type>java.util.Collection</cmr-field-type>
  </cmr-field>
 </ejb-relationship-role>
 <ejb-relationship-role>
  <ejb-relationship-role-name>Emps-have-Dept</ejb-relationship-role-name>
  <multiplicity>Many</multiplicity>
  <relationship-role-source>
   <ejb-name>EmpBean</ejb-name>
  </relationship-role-source>
  <cmr-field><cmr-field-name>dept</cmr-field-name></cmr-field>
 </ejb-relationship-role>
</ejb-relation>
```

`ejb-jar.xml` ファイル内に指定されている関連の一部を構成する各 Entity Bean は、同じ EJB-JAR ファイル内にデプロイする必要があります。

関連を持つ Bean に対するローカル・インタフェースの使用

EJB-JAR ファイル内の Entity Bean は、ローカル・インタフェースを使用して構成することをお薦めします。クライアントは、同じアプリケーション（同じ JAR ファイル）内にデプロイされ、同じ OC4J JVM 内で実行されているコンポーネントからローカル・インタフェースを参照できます。

すべての Entity Bean は、リモート・インタフェースの他に、そのクライアントへのローカル・インタフェースを指定するために変更する必要があります。

EJB 問合せ言語（EJBQL）の使用

リリース 2（9.0.3）より前の Oracle9*i*AS では、すべての finder メソッドとその SQL は、<finder-method> 要素を使用し、Oracle 固有の方法で orion-ejb-jar.xml ファイルに定義されていました。EJB 2.0 仕様が完成した後、この Oracle 固有の方法は不要となりました。第 5 章「EJB 問合せ言語」で説明したように、既存の finder メソッドは、変更しない場合でも、機能します。ただし、以前の finder メソッドを削除して、ejb-jar.xml ファイルの <query> 要素内で EJB 2.0 の finder メソッドを使用できます。これらの finder メソッドでは、EJBQL を使用して問合せの SQL を定義します。

次に、SQL によってすべての部門が選択される EJBQL の finder メソッドの例を示します。

```
<query>
 <description></description>
 <query-method>
  <method-name>findAll</method-name>
  <method-params/>
 </query-method>
 <result-type-mapping>Local</result-type-mapping>
 <ejb-ql>Select OBJECT(d) From Dept d</ejb-ql>
</query>
```

デプロイ時に、EJB デプロイメント・ディスクリプタ（ejb-jar.xml ファイル）が解析されます。Oracle 固有のデプロイメント・ディスクリプタ（orion-ejb-jar.xml ファイル）があれば、それも解析されます。orion-ejb-jar.xml ファイルがない場合は、デフォルトと ejb-jar.xml ファイルに設定した内容に基づいた構成を持つファイルが作成されます。EJBQL の場合、SQL は orion-ejb-jar.xml ファイルに配置されます。SQL は、orion-ejb-jar.xml ファイルを変更することによって、さらにカスタマイズできます。

例 C-1 1 対 1 単方向

次は ejb-jar.xml ファイルの要素の例で、従業員と住所の 1 対 1 の単方向の関連を示しています。従業員は EmpBean EJB で、住所は AddressBean EJB で示されています。

```
<ejb-relation>
     <ejb-relation-name>Emp-Address</ejb-relation-name>
     <ejb-relationship-role>
          <ejb-relationship-role-name>Emp-has-Address
          </ejb-relationship-role-name>
          <multiplicity>One</multiplicity>
          <relationship-role-source>
              <ejb-name>EmpBean</ejb-name>
          </relationship-role-source>
          <cmr-field>
              <cmr-field-name>address</cmr-field-name>
          </cmr-field>
     </ejb-relationship-role>
     <ejb-relationship-role>
```

```
            <ejb-relationship-role-name>Address-has-Emp
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <cascade-delete/>
            <relationship-role-source>
                <ejb-name>AddressBean</ejb-name>
            </relationship-role-source>
        </ejb-relationship-role>
</ejb-relation>
```

この例と 1 対多の関連との間には、2 つの大きな違いがあります。関連の両側には、1 対 1 の性質を定義するために One の多重度があります。2 番目の違いは、Address 関連には <cmr-field> 要素がないことです。これは、この関連が単方向であることを示しています。

結論

EJB 間の関連のモデル化は、EJB 1.1 では簡単ではありませんでした。EJB 2.0 では、関連と永続性の定義が簡単になりました。Oracle JDeveloper の試用をお薦めします。これには、ウイザードによる代替方法が用意されており、エラーの多い手動によるデプロイメント・ディスクリプタ管理の負担を軽減します。JDeveloper は次のサイトからダウンロードできます。 http://otn.oracle.co.jp/software/products/jdev/index.html.

D

サード・パーティ・ライセンス

この付録には、Oracle Application Server に付属するすべてのサード・パーティ製品のサード・パーティ・ライセンスが記載されています。この付録には次の項目が含まれています。

Apache HTTP Server

Apache のライセンス条件に基づき、Oracle は次のライセンス文書を表示することが求められています。ただし、Oracle プログラム（Apache ソフトウェアを含む）を使用する権利は、この製品に付随する Oracle プログラム・ライセンスによって決定され、次のライセンス文書に含まれる条件でこの権利が変更されることはありません。反対の内容が Oracle プログラム・ライセンス内にあった場合でも、Apache ソフトウェアは現状のままで Oracle から提供されるものであり、いかなる種類の保証またはサポートも Oracle または Apache から提供されません。

The Apache Software License

```
/* ====================================================================
 * The Apache Software License, Version 1.1
 *
 * Copyright (c) 2000 The Apache Software Foundation.  All rights
 * reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the
 *    distribution.
 *
 * 3. The end-user documentation included with the redistribution,
 *    if any, must include the following acknowledgment:
 *       "This product includes software developed by the
 *        Apache Software Foundation (http://www.apache.org/)."
 *    Alternately, this acknowledgment may appear in the software itself,
 *    if and wherever such third-party acknowledgments normally appear.
 *
 * 4. The names "Apache" and "Apache Software Foundation" must
 *    not be used to endorse or promote products derived from this
 *    software without prior written permission. For written
 *    permission, please contact apache@apache.org.
 *
 * 5. Products derived from this software may not be called "Apache",
 *    nor may "Apache" appear in their name, without prior written
 *    permission of the Apache Software Foundation.
 *
```

```
* THIS SOFTWARE IS PROVIDED ''AS IS'' AND ANY EXPRESSED OR IMPLIED
* WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
* OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
* DISCLAIMED.  IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR
* ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
* LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
* USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
* ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
* OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
* OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
* ====================================================================
*
* This software consists of voluntary contributions made by many
* individuals on behalf of the Apache Software Foundation.  For more
* information on the Apache Software Foundation, please see
* <http://www.apache.org/>.
*
* Portions of this software are based upon public domain software
* originally written at the National Center for Supercomputing Applications,
* University of Illinois, Urbana-Champaign.
*/
```

Apache JServ

Apache のライセンス条件に基づき、Oracle は次のライセンス文書を表示することが求められています。ただし、Oracle プログラム（Apache ソフトウェアを含む）を使用する権利は、この製品に付随する Oracle プログラム・ライセンスによって決定され、次のライセンス文書に含まれる条件でこの権利が変更されることはありません。反対の内容が Oracle プログラム・ライセンス内にあった場合でも、Apache ソフトウェアは現状のままで Oracle から提供されるものであり、いかなる種類の保証またはサポートも Oracle または Apache から提供されません。

Apache JServ Public License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistribution of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistribution in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- All advertising materials mentioning features or use of this software must display the following acknowledgment:

  This product includes software developed by the Java Apache Project for use in the Apache JServ servlet engine project (http://java.apache.org/).

- The names "Apache JServ", "Apache JServ Servlet Engine" and "Java Apache Project" must not be used to endorse or promote products derived from this software without prior written permission.

- Products derived from this software may not be called "Apache JServ" nor may "Apache" nor "Apache JServ" appear in their names without prior written permission of the Java Apache Project.

- Redistribution of any form whatsoever must retain the following acknowledgment:

  This product includes software developed by the Java Apache Project for use in the Apache JServ servlet engine project (http://java.apache.org/).

THIS SOFTWARE IS PROVIDED BY THE JAVA APACHE PROJECT "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE JAVA APACHE PROJECT OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

索引

記号

A

B

C

D

E

F

G

P

Q

R

S

T

U

す

せ

ち

て

と

な

は

ひ

ふ

ほ

ま

め

り

ろ