

Oracle8*i*

Enterprise JavaBeans 開発者ガイドおよびリファレンス

リリース 8.1

2000 年 11 月

部品番号 : J02313-01

ORACLE®

Oracle8i Enterprise JavaBeans 開発者ガイドおよびリファレンス, リリース 8.1

部品番号 : J02313-01

原本名 : Oracle8i Enterprise JavaBeans Developer's Guide and Reference, Release 3 (8.1.7)

原本部品番号 : A83725-01

原本著者 : Sheryl Maring

原本協力者 : Tim Smith, Ellen Barnes, Matthieu Devin, Steve Harris, Hal Hildebrand, Susan Kraft, Thomas Kurian, Wendy Liao, Angie Long, Sastry Malladi, John O'Duinn, Jeff Schafer, Aniruddha Thakur, Iyad Elayyan, Cheuk Chau

Copyright © 1996, 2000, Oracle Corporation. All rights reserved.

Printed in Japan.

制限付権利の説明

プログラム（ソフトウェアおよびドキュメントを含む）の使用、複製または開示は、オラクル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に関する法律により保護されています。

当プログラムのリバース・エンジニアリング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ことがあります。オラクル社は本ドキュメントの無謬性を保証しません。

* オラクル社とは、Oracle Corporation（米国オラクル）または日本オラクル株式会社（日本オラクル）を指します。

危険な用途への使用について

オラクル社製品は、原子力、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を用途として開発されておりません。オラクル社製品を上述のようなアプリケーションに使用することについての安全確保は、顧客各位の責任と費用により行ってください。万一かかる用途での使用によりクレームや損害が発生いたしましても、日本オラクル株式会社と開発元である Oracle Corporation（米国オラクル）およびその関連会社は一切責任を負いかねます。当プログラムを米国国防総省の米国政府機関に提供する際には、『Restricted Rights』と共に提供してください。この場合次の Notice が適用されます。

Restricted Rights Notice

Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication, and disclosure of the Programs, including documentation, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication, and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-19, Commercial Computer Software - Restricted Rights (June, 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

このドキュメントに記載されているその他の会社名および製品名は、あくまでその製品および会社を識別する目的にのみ使用されており、それぞれの所有者の商標または登録商標です。

目次

はじめに	ix
------------	----

1 概要

Enterprise JavaBeans について	1-2
EJB 開発の役割	1-2
Oracle8i の EJB 実装の機能	1-3
IIOP を介した RMI	1-3
IIOP トランスポート・プロトコル	1-4
JNDI	1-4
ステートフルとステートレスのセッション Bean	1-5
EJB の実装	1-6
基本概念	1-7
EJB の種類	1-9
セッション Bean	1-10
エンティティ Bean	1-10

2 Enterprise JavaBeans

Enterprise JavaBeans の起動	2-2
Enterprise JavaBeans の作成	2-3
リモート・インタフェースとホーム・インタフェースの実装要件	2-4
リモート・インタフェースの作成	2-4
ホーム・インタフェースの作成	2-6
例外クラスの作成	2-7
Bean の実装	2-8
実装されるインタフェース	2-8

Bean 実装の例	2-10
クライアント・アプリケーションの開発	2-12
getEJBHome メソッドの使用	2-13
パラメータの受渡し	2-13
パラメータ・オブジェクト	2-14
クライアント・コード	2-14
EJB の配置	2-19
配置の手順	2-20
ディプロイメント・ディスクリプタの作成	2-21
Oracle 配置マップ・ファイルの作成	2-24
JAR ファイルの作成	2-25
ホーム・インタフェースの公開	2-26
EJB の削除	2-26
例の実行	2-26
プログラミングの制限事項	2-27
デバッグ手法	2-28
デバッグ・エージェントを使用したサーバー・アプリケーションのデバッグ	2-29

3 IIOP アプリケーションの構成

概要	3-2
Oracle8i 標準インストールまたは最小インストール	3-3
Oracle8i カスタム・インストール	3-4
手動インストールおよび構成	3-8
ツールを使用して構成する場合	3-8
初期化ファイルを編集して構成する場合	3-9
高度な構成に関するオプション	3-11
データベースのリスナーとディスパッチャ	3-12
動的なリスナー・エンドポイント登録	3-15
ディスパッチャへの直接接続	3-16
EJB および CORBA 用の SSL の構成	3-17

4 エンティティ Bean

エンティティ Bean の定義	4-2
永続データの管理	4-2
主キーによる一意な識別	4-2

依存オブジェクトを伴う複雑なロジックの実行	4-2
セッション Bean とエンティティ Bean の違い	4-5
コールバック・メソッドの実装	4-5
ejbCreate および ejbPostCreate の使用	4-7
setEntityContext の使用	4-8
ejbRemove の使用	4-8
ejbStore および ejbLoad の使用	4-9
エンティティ Bean の作成	4-9
ホーム・インタフェース	4-10
リモート・インタフェース	4-11
主キー	4-12
エンティティ Bean クラス	4-15
エンティティ・データ用のデータベース表および列の作成	4-22
エンティティ Bean の配置	4-23
クライアントから配置済みエンティティ Bean へのアクセス	4-25
Bean 管理の永続性とコンテナ管理の永続性の違い	4-26
コンテナ管理の永続性	4-27
EJB の参照と JDBC の DataSources へのアクセス	4-35
EJB の参照	4-35
JDBC の DataSources	4-35

5 JNDI 接続とセッション IIOP サービス

JNDI 接続の基本事項	5-2
ネームスペース	5-3
データベース・オブジェクトに対する実行権限	5-4
URL 構文	5-5
URL のコンポーネントとクラス	5-6
JNDI を使用したバインド済みオブジェクトへのアクセス	5-7
JNDI サポート・クラスのインポート	5-8
JNDI InitialContext の取得	5-9
セッション IIOP サービス	5-12
セッション IIOP サービスの概要	5-13
セッション管理	5-15
サービス・コンテキスト・クラス	5-15
セッション・コンテキスト・クラス	5-17

セッション管理の使用例	5-17
セッション・タイムアウトの設定	5-26
Oracle8i JVM のバージョン・ナンバーの検索	5-27
非 IIOP プレゼンテーションからのセッション中の EJB オブジェクトの活性化	5-27
アプレットからの EJB オブジェクトの起動	5-28
署名付き JAR ファイルを使用したサンドボックス・セキュリティへの準拠	5-28
アプレット内のオブジェクト検索の実行	5-28
EJB オブジェクトにアクセスするアプレット用の HTML の修正	5-30

6 IIOP のセキュリティ

概要	6-2
データ整合性	6-3
セキュア・ソケット・レイヤー (SSL) の使用	6-3
SSL のバージョンのネゴシエーション	6-4
認証	6-5
クライアント側の認証	6-6
認証のための JNDI の使用	6-8
ユーザー名とパスワードを使用したクライアント側の認証	6-9
クライアント認証のための証明書の使用	6-12
AuroraCertificateManager クラス	6-16
サーバー側の認証	6-20
認可	6-26
トラスト・ポイントの設定	6-27
サーバー証明書の連鎖の解析	6-27
AuroraCurrent クラス	6-28

7 トランザクション操作

トランザクションの概要	7-2
グローバル・トランザクションとローカル・トランザクション	7-2
トランザクションのデマーケート	7-3
コンテナ管理または Bean 管理のトランザクション	7-4
トランザクション・コンテキストの伝播	7-5
リソースの確保	7-7
2 フェーズ・コミット	7-8
JTA の概要	7-10

JTA の制限事項	7-13
JTA のサーバー側デマーケーション	7-13
コンテナ管理のトランザクション	7-13
Bean 管理のトランザクション	7-14
JTA のクライアント側デマーケーション	7-16
サーバー側でのリソースの確保	7-25
2 フェーズ・コミット・エンジンの構成	7-29
DataSource オブジェクトの動的作成	7-37
トランザクションのタイムアウト設定	7-38
セッション同期インタフェースの使用	7-39
JDBC の制限事項	7-41

A XML デプロイメント・ディスクリプタ

Enterprise JavaBean のデプロイメント・ディスクリプタ	A-3
ヘッダー	A-3
JAR ファイル	A-3
Enterprise JavaBeans の記述子	A-4
アプリケーション・アセンブラ・セクション	A-15
Oracle 固有のデプロイメント・ディスクリプタ	A-22
ヘッダー	A-22
マッピングの定義	A-23
トランザクションの 2 フェーズ・コミット・エンジンの定義	A-26
run-as アイデンティティの定義	A-26
コンテナ管理の永続性の定義	A-27
EJB クライアントの JAR セクション	A-32
Oracle 固有のデプロイメント・ディスクリプタの DTD	A-32

B コード例 : EJB

基本例	B-2
README	B-2
クライアント	B-6
Hello のホーム・インタフェース	B-6
Hello のリモート・インタフェース	B-7
Hello の Bean 実装	B-7

SQLJ の例	B-8
README	B-8
クライアント	B-11
ホーム・インタフェース	B-12
リモート・インタフェース	B-12
Bean 実装	B-12
Bean の継承例	B-13
README	B-13
クライアント	B-17
ホーム・インタフェース	B-18
リモート・インタフェース	B-19
Bean 実装	B-19
エンティティ Bean の例	B-21
Bean 管理のエンティティ Bean の例	B-21
クライアント	B-21
ホーム・インタフェース	B-23
リモート・インタフェース	B-23
Bean 実装	B-24
ディプロイメント・ディスクリプタ	B-28
コンテナ管理のエンティティ Bean の例	B-29
クライアント	B-29
ホーム・インタフェース	B-32
リモート・インタフェース	B-32
Bean 実装	B-32
XML ディプロイメント・ディスクリプタ	B-34
Oracle 固有のディプロイメント・ディスクリプタ	B-36
データベース表の更新	B-36
セッションの例	B-37
README	B-37
クライアント	B-40
ホーム・インタフェース	B-41
リモート・インタフェース	B-41
Bean 実装	B-41
SSL の例	B-43
クライアント側の認証の例	B-43
README	B-43

クライアント	B-44
ホーム・インタフェース	B-45
リモート・インタフェース	B-45
Bean 実装	B-45
サーバー側の認証の例	B-46
README	B-46
クライアント	B-46
ホーム・インタフェース	B-47
リモート・インタフェース	B-48
Bean 実装	B-48

C 略称と頭字語

索引

はじめに

このマニュアルは、Oracle8i 用の Enterprise JavaBeans を作成する出発点となるものです。
アプリケーション開発に役立つ多数のコード例が含まれています。

対象読者

このマニュアルは、Oracle8i 用のサーバー側 Enterprise JavaBeans の開発に役立ちます。プログラマー向けに書かれていますが、設計者、システム・アナリスト、プロジェクト・マネージャおよびネットワーク中心データベース・アプリケーションに関心のある担当者にも参考になります。このマニュアルを有効に活用するには、Java と Oracle8i に関する実務知識が必要です。

前提となる関連ドキュメント

このマニュアルの前に、次のマニュアルを読んでください。

- 『Oracle8i Java 開発者ガイド』には、データベース・サーバーにおける Java を理解するための技術上の背景情報が記載されています。エンタープライズ・アプリケーション開発用の Oracle8i JVM 実装の利点を総合的に説明するとともに、Oracle8i JVM の基本事項と Oracle8i JVM に付属するツールの技術面の概要も述べます。
- このマニュアルの補足資料として Sun Microsystems の EJB 1.1 仕様書。このマニュアルは、EJB 1.1 仕様の詳細に関する基礎知識を前提としています。

Enterprise JavaBeans に関する情報源の詳細は、x ページの「[参考資料](#)」を参照してください。

参考資料

書籍

- 『Core Java』（Cornell & Horstmann 著、第 2 版、Volume II、Prentice-Hall、1997 年）では、EJB に関係するいくつかの Java の概念をわかりやすく説明します。たとえば、Remote Method Invocation（RMI）インタフェースについて記述されています。
- EJB の概要をまとめた『Developer's Guide to Understanding Enterprise JavaBeans』は <http://www.Nova-Labs.com> から入手可能です。

オンライン情報

Java に関しては役立つ情報源が多数あり、オンラインで入手できます。たとえば、Sun Microsystems のホーム・ページでオンライン・マニュアルとチュートリアルを参照またはダウンロードできます。URL は次のとおりです。

<http://www.sun.com>

現行の 1.1 EJB 仕様書は、次の URL で入手可能です。

<http://java.sun.com/products/ejb/docs.html>

代表的な Java Web サイトとしては他に次のサイトも参照してください。

<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>

Patricia Seybold グループの Anne Thomas による白書（Sun Microsystems がスポンサーとなっている論文）は、次の URL で入手可能です。

http://java.sun.com/products/ejb/white_paper.html

関連資料

このマニュアルでは、記述の中で、詳細の参照先として次の Oracle 関連資料を紹介しています。

『Oracle8i アプリケーション開発者ガイド 基礎編』

『Oracle8i Java 開発者ガイド』

『Oracle8i JDBC 開発者ガイドおよびリファレンス』

『Oracle8i SQL リファレンス』

『Oracle8i SQLJ 開発者ガイドおよびリファレンス』

構成

このマニュアルの構成は、次のとおりです。

第1章「概要」では、Oracle8i の観点から EJB 開発モデルの概要を記述しています。

第2章「Enterprise JavaBeans」では、Oracle8i JVM 用の EJB 開発を説明します。この章は EJB に関するチュートリアルではありませんが、Sun Microsystems の仕様書に記載されている EJB の基本概念をいくつか説明します。各例は、セッション Bean 実装に重点が置かれています。

第3章「IIOP アプリケーションの構成」では、データベース内で EJB を実行するための必須構成について説明します。

第4章「エンティティ Bean」では、エンティティ Bean の実装方法について説明します。エンティティ Bean のコンテナ管理の永続モデルと Bean 管理の永続モデルの詳細を説明します。

第5章「JNDI 接続とセッション IIOP サービス」では、セッション管理、セッション IIOP サービスおよび JNDI ネームスペースについて説明します。JNDI およびセッション IIOP サービスを使用して、サーバー内に配置されている EJB にアクセスする例と使用例が含まれています。

第6章「IIOPのセキュリティ」では、認証用のセキュリティ・オプションについて説明します。

第7章「トランザクション操作」では、EJBの開発時に使用するJTAトランザクション・インタフェースについて説明します。

付録A「XMLディプロイメント・ディスクリプタ」では、EJBおよびOracle固有のディプロイメント・ディスクリプタについて説明します。両方のディプロイメント・ディスクリプタに含まれる全要素の詳細と意味が含まれています。

付録B「コード例:EJB」には、EJBアプリケーションの例を掲載します。

付録C「略称と頭字語」には、頭字語のリストを掲載します。

表記規則

このマニュアルは、次の規約に従います。

固定幅フォント 固定幅フォントは、Javaのプログラム名、ファイル名、パス名およびインターネット・アドレスを表します。

Javaコード例では、次の規約に従います。

{ }	中カッコ {} は、文のブロックを囲みます。
//	ダブルスラッシュが先頭にある行は、行末までがコメントであることを示します。
/* */	スラッシュ - アスタリスクとアスタリスク - スラッシュは、複数の行にまたがるコメントであることを示します。
...	省略記号 (...) は、説明内容と無関係な文または句が割愛されていることを示します。
小文字	キーワードと、1 単語からなる変数、メソッドおよびパッケージの名前を表します。
大文字	定数名 (static final 変数)、および組込み SQL データ型にマッピングされる、提供されるクラス名には大文字を使用します。
大文字と小文字の組合せ	クラスおよびインタフェースの名前と、複数の単語からなる変数、メソッドおよびパッケージの名前は、大文字小文字を混ぜて表します。クラスとインタフェースの名前は、大文字で始まります。複数の単語からなる名前は、2 番目以降の単語も大文字で始まります。

この章では、Oracle8i JVM における分散オブジェクト開発の全体像を説明します。ここでは、Enterprise JavaBeans (EJB) 開発のうち Oracle8i JVM に特有の側面に重点を置き、EJB の標準的な開発モデル全般について簡単に説明します。

この章では、次のトピックを説明します。

- [Enterprise JavaBeans について](#)
- [EJB の実装](#)
- [基本概念](#)
- [EJB の種類](#)

Enterprise JavaBeans について

Oracle8i JVM は EJB 1.1 仕様に準拠しており、拡張性とパフォーマンスの高い EJB の実行環境を提供します。

Enterprise JavaBeans (EJB) は、分散アプリケーションを開発するためのアーキテクチャです。さらに、EJB アプリケーションは Java のみで開発できます。開発者は、CORBA アプリケーション用の IDL などの新しい言語を習得せずに済みます。

EJB は、トランザクションを単位とするコンポーネント・ベースの分散コンピューティングのためのアーキテクチャです。EJB の仕様には、Bean 自体の形式のみでなく、Bean が動作するコンテナで提供する一連のサービスも規定されています。このため、EJB は分散アプリケーション開発のための強力な開発手法となっています。Bean 開発者もクライアント・アプリケーション・プログラマも、トランザクション・サポート、セキュリティ、リモート・オブジェクト・アクセスなどのサービスの詳細や、その他多数の複雑で間違いを起こしやすい問題に気を配らなくて済みます。EJB サーバーおよびコンテナにより、これらの機能が提供されます。

EJB アーキテクチャを採用すると、サーバー側の開発が Java アプリケーション・プログラマにとって非常に容易になります。開発者は実装の細かな部分からは解放され、トランザクション・サポートやセキュリティなどのサービスは使用しやすい方法で提供されるので、比較的短期間で EJB を開発できます。さらに、EJB には移植性があります。EJB サーバー上で開発された Bean は、EJB 仕様を満たす他の EJB サーバー上でも動作します。

EJB 開発の役割

EJB 仕様は、エンタープライズ Bean 開発を次の 5 つの役割分担の上に規定しています。

- EJB 開発者が、個々の EJB を実装するコードを作成します。このコードは、通常データベース・アクセスを伴う、アプリケーションのビジネス・ロジックです。

EJB 開発者とは、SQL と、SQLJ または JDBC を使用するデータベース・アクセスの両方に精通した Java アプリケーション・プログラマのことです。

- EJB 実行者が、EJB をインストールし、公開します。この作業には、EJB のトランザクション面の性質の理解が必要であるため、EJB 開発者と連携を取りながら実行します。EJB 実行者は、配置するそれぞれの Bean のプロパティを指定したディプロイメント・ディスクリプタ・ファイルを記述します。

EJB 実行者は、ネットワーク・ポート、必要なデータベース・ロール、その他スキーマに固有の要件などのデータベースに固有の事項も含めて、EJB のランタイム環境をよく理解している必要があります。Oracle8i Server に対して、EJB 実行者には、データベース内で EJB ホーム・インタフェースを公開し、この情報をクライアント側のアプリケーション開発者に通知する責任があります。

- EJB サーバー・ベンダーで、EJB コンテナの動作するフレームワークが実装されます。Oracle では、Oracle8i データ・サーバーが EJB コンテナをサポートしているフレームワークです。

- EJB コンテナ・ベンダーで、実行時に EJB をサポートするサービスが提供されます。たとえば、クライアントが Bean でトランザクション・サポートを自動処理にする場合には、コンテナ・フレームワークとデータ・リソースとを合わせたもので EJB をサポートします。
- アプリケーション開発者が、サーバー EJB 上のメソッドをコールするクライアント側コードを作成します。

EJB サーバー開発者と EJB コンテナ開発者の役割は、明確には区別されていません。たとえば、コンテナとサーバーの間の標準化された API はありません。このため、EJB サーバーとコンテナの初期の実装を、同じベンダーが行うことは、よくあることです。Oracle8i の場合もそうです。

Oracle8i の EJB 実装の機能

Oracle8i の EJB 実装では Oracle データベース・サーバーを利用することによって、次の機能を提供しています。

- JNDI インタフェースを介した OMG CosNaming サービスによる、Bean の簡単なロケティングおよび活性化。
- 索引付き表への高速なアクセスなどのパフォーマンス上の利点を持つ、データベースをネーム・サーバーとして使用するセッション・ネームスペース。
- セキュリティを高めるセキュア・ソケット・レイヤー (SSL) 接続。
- オブジェクトに対する標準 Oracle データベース認証およびマルチレイヤー・アクセス制御。
- トランザクション・デマーケーションのための Java Transaction Architecture (JTA) の実装。
- Bean 管理のトランザクションのための UserTransaction インタフェース。
- EJB アプリケーションを配置するための支援ツール。

IIOP を介した RMI

EJB は、Java の Remote Method Invocation (RMI) をトランスポート・プロトコルとして規定します。EJB は、概念上は、Java の Remote Method Invocation (RMI) モデルに基づいています。たとえば、EJB のリモート・オブジェクト・アクセスとパラメータの受渡しは、RMI 仕様に従います。

EJB 仕様では、トランスポート・メカニズムが純粋な RMI であることを規定してはいません。Oracle8i EJB サーバーでは、トランスポート・プロトコルに IIOP を介した RMI が使用されます。CORBA Internet Inter-ORB Protocol (IIOP) は、CORBA および次期バージョンの RMI 用のトランスポート・プロトコルであるため、Oracle8i では、様々なオープン・システムへの直接的なオブジェクト指向アクセスを効率よく実行できます。

IIOP トランスポート・プロトコル

Oracle8i は、Java で実装された IIOP プロトコルのインタプリタを備えています。大手 CORBA ベンダーの Java ORB を埋め込み、データベース上で動作するように Visigenic Java IIOP インタプリタをパッケージし直しています。Oracle8i は拡張性が高いサーバーであるため、インタプリタの主要コンポーネントのみを必要とします。これは、次の作業を行う一連の Java クラスです。

- IIOP プロトコルのデコード
- 関連する Java オブジェクトの検出または活性化
- IIOP メッセージが指定するメソッドの起動
- クライアントへの IIOP リプライの返信

Oracle8i では、ORB スケジュール機能を使用しません。Oracle マルチスレッド・サーバーはディスパッチを行うため、サーバーは IIOP メッセージを効率よく、拡張性の高い方法で処理できます。

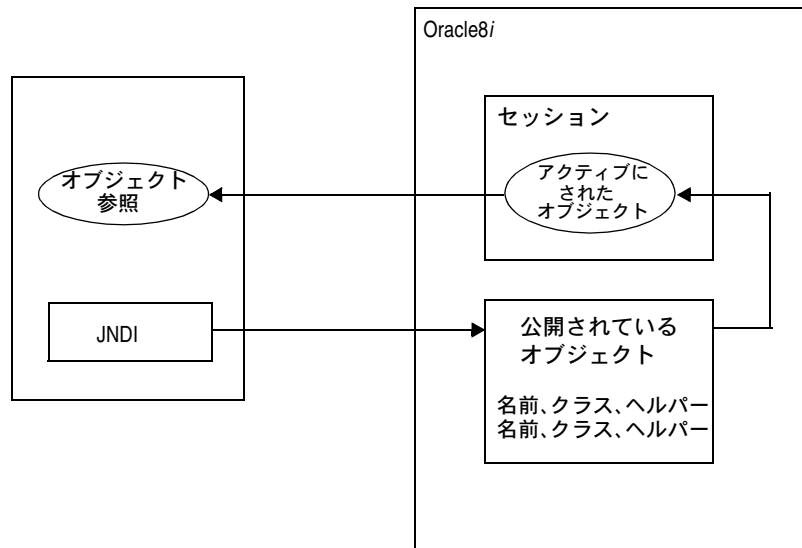
Oracle8i では、このインフラストラクチャの上に EJB のプログラミング・モデルが実装されます。

JNDI

EJB 開発者は、EJB 仕様に従ってアクセスに JNDI を使用します。各 Bean は、配置プロセス中に自動的に公開されます。JNDI は、CosNaming レイヤーを介して、公開されている Bean へのアクセスを可能にします。

図 1-1 は、JNDI を使用してデータベース内で公開されているリモート・オブジェクトにアプリケーションからアクセスする方法を示しています。

図 1-1 リモート・オブジェクト・アクセス



ステートフルとステートレスのセッション Bean

EJB 仕様では、ステートレス Bean とステートフル Bean の、2 種類のセッション Bean が規定されています。

- ステートレス Bean は、あるメソッドの起動と、その次の起動との間で、状態またはを共有しない Bean です。主に、OLTP アプリケーションに関連する要求など、頻繁に発生する短い要求を処理するために Bean をプールしている、中間層アプリケーション・サーバーで使用されます。
- ステートフル Bean は、長期間のセッションに役立つ Bean です。長期間のセッションでは、メソッドを起動してから次のメソッドを起動するまでの間、インスタンス変数やトランザクション状態などの状態を維持する必要があります。

Oracle8i ORB と Java VM はマルチスレッド・サーバー（MTS）のもとで実行されるので、Oracle8i JVM の場合、ステートレス・セッション Bean とステートフル・セッション Bean の違いは重要ではありません。したがって、EJB では、新しいセッション中に要求に応じてステートフル・セッション Bean のみが活性化されます。ステートフル Bean では、ステートレス Bean と同じパフォーマンスを出しつつ、カンパセショナル・ステートも保持されています。

EJB の実装

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

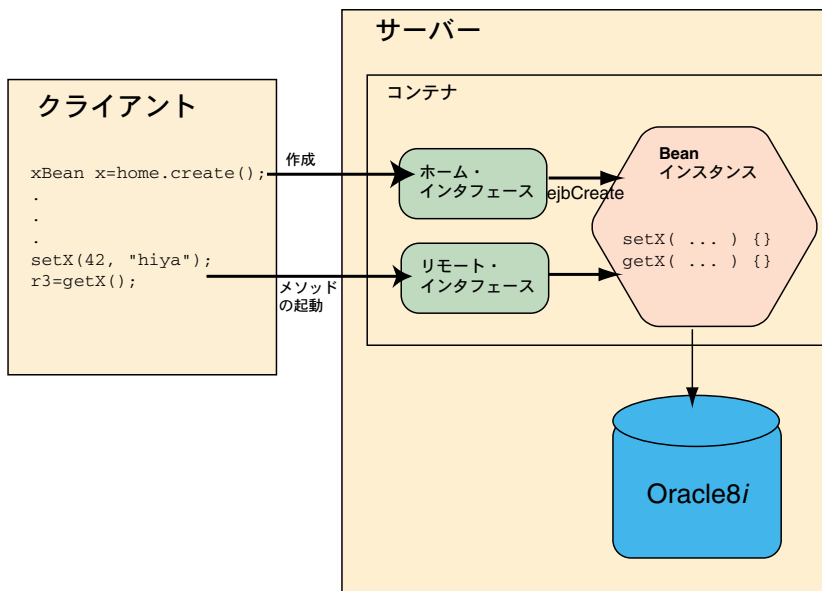
- ホーム・インタフェース
- リモート・インタフェース
- リモート・インタフェースの実装 — 実際の Bean クラス
- 各 EJB のディプロイメント・ディスクリプタ

コンポーネント	説明
ホーム・インタフェース	コンテナ自体により実装されるオブジェクトである、ホーム・オブジェクトへ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 <code>create()</code> メソッドがあります。ホーム・インタフェースはホーム・オブジェクトとともに、EJB のファクトリ・オブジェクトとして機能します。
リモート・インタフェース	Bean に実装するメソッドを指定します。これらのメソッドでは、Bean のビジネス・ロジックが実行されます。Bean もまた、Bean のライフ・サイクルの様々な時期に EJB コンテナによりコールされるサービス・メソッドを実装する必要があります。この種のサービス・メソッドの詳細は、1-7 ページの「 基本概念 」を参照してください。
Bean 実装	リモート・インタフェースと必要なコンテナ・メソッドを実装した Java コードが含まれます。
ディプロイメント・ディスクリプタ	配置とデータベースへのロードに関する Bean の属性を指定します。たとえば、ディプロイメント・ディスクリプタにより、Bean のトランザクション・プロパティを宣言します。配置時に、EJB 実行者はアプリケーション開発者とともに、コンテナでトランザクション・サポートを管理するのか、クライアントにそれを管理させるのかを決定できます。

クライアント・アプリケーション自体が Bean に直接アクセスすることはありません。直接アクセスしないかわりに、コンテナでは Bean のサーバー側プロキシの役割を果たす、EJBObject と呼ばれるサーバー側オブジェクトが生成されます。EJBObject がクライアントからメッセージを受け取るので、コンテナは Bean 実装にメッセージを送信する前に独自の処理を挿入できます。

1-7 ページの [図 1-2](#) に、これらのコンポーネント間の相互作用を示します。

図 1-2 基本的な EJB コンポーネントの関係



基本概念

EJB の実装の詳細を説明する前に、基本概念をいくつか明確にしておく必要があります。まず第 1 に、Bean はコンテナ内で実行されることを思い出してください。コンテナは、EJB サーバーの一部であり、多数のサービスを Bean に提供します。このようなサービスとしては、トランザクション・サービス、同期サービスおよびセキュリティなどがあります。

これらのサービスを提供するために、Bean コンテナは Bean メソッドのコールに割り込むことが可能です。たとえば、クライアント・アプリケーションが、新しいトランザクション・コンテキストの作成を Bean に要求するようなトランザクション属性を持つ Bean メソッドをコールするとします。Bean コンテナは、メソッド・コールの前に新しいトランザクションを開始し、可能な場合には、メソッドの完了時に、値がクライアントに戻される前に、トランザクションをコミットするよう、コードを挿入できます。

このような理由や、その他の理由から、クライアント・アプリケーションはリモート Bean メソッドを直接的には起動しません。そのかわりに、クライアントは、ORB とコンテナが仲介する 2 段階のプロセスで Bean メソッドをコールします。

1. クライアントが、リモート・インタフェース・オブジェクト以外から Bean メソッドを起動します。
 - a. クライアントが、リモート・メソッドのローカル・プロキシ・スタブを実際にコールします。
 - b. スタブはパラメータ・データのマーシャリングを実行し、サーバー上のリモート・スケルトンをコールします。
2. スケルトンは、データのアンマーシャリングと、Bean コンテナへのアップコールを実行します。

このステップが必要なのは、このコールがリモート的な性質を持つためです。このステップはクライアント・アプリケーション開発者と Bean 開発者の双方に対し完全に透過的である点に注意してください。クライアントあるいはサーバー・アプリケーションを作成するうえで、詳細を知っている必要はありません。しかし、何が起きているのかを把握しておく、特に Bean の配置中に起こることを理解する場合に役立ちます。

3. Bean コンテナがスケルトン・アップコールを受け取り、コンテキストで要求されるサービスを挿入します。次のものがあります。
 - 最初のメソッド・コールでの、クライアントに対する認証の実行
 - トランザクション管理の実行
 - Bean 自身の中の同期メソッドのコール
 - 認証チェックと切替え
4. コンテナがメソッドのコールを Bean に委譲します。
5. Bean メソッドが実行されます。
6. メソッドが戻ると、制御のスレッドは Bean コンテナに戻り、コンテキストで必要とされるサービスが挿入されます。

たとえば、メソッドがトランザクション・コンテキストで実行されている場合、Bean コンテナでは Bean 記述子内のトランザクション属性に応じて、可能であればコミット操作が実行されます。

7. Bean コンテナではスケルトンがコールされて、そのスケルトンではリターン・データのマーシャリングが実行され、その結果がクライアント・スタブに戻されます。

これらのステップが、クライアント側とサーバー側のアプリケーション開発者に対して表示されることはありません。EJB 開発モデルの大きな利点の1つは、開発者が複雑なトランザクションと認証管理から解放されることです。

EJB の種類

EJB にはセッション Bean とエンティティ Bean の 2 種類があります。セッション Bean では 1 つ以上のビジネス・タスクが実装されますが、エンティティ Bean は複雑なビジネス・エンティティです。それが両者を簡単に区別できる相違点です。セッション 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行うメソッドが含まれる場合があります。エンティティ Bean では、ビジネス・データが直接、または他の永続 Bean を介して間接的に表現されます。

セッション Bean は、多くの場合、サービスの実装に使用されます。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の検索と更新を行う 1 つまたは複数のセッション Bean を実装することがあります。セッション Bean を使用してデータベース・サーバー内のストアド・プロシージャを置き換えることで、Oracle8i Java サーバー特有の拡張性を活用できます。

エンティティ Bean は、多くの場合、データとその計算を伴うビジネス・サービスを容易にするために使用されます。たとえば、アプリケーション開発者は、発注に含まれる品目の検索と計算を行うエンティティ Bean を実装できます。エンティティ Bean では、必要なタスクの実行中に複数の依存永続オブジェクトを管理できます。

永続性

セッション Bean は本質的に永続性 (Persistence) があるわけではありません。Persistence という言葉は、Bean の特性 (エンティティ Bean には永続性があり、セッション Bean には本質的な永続性はありません) を指す場合もあれば、データを将来のインスタンス化で取り出せるように Bean が保存するデータを指す場合もあります。永続データはデータベース内に保存されます。

したがって、セッション Bean ではその状態を Oracle8i データベースに保存できますが、ビジネス・データが直接表現されるわけではありません。エンティティ Bean は、ビジネス・データを自動的に (コンテナ管理の永続エンティティ Bean 内に) 保持するか、または JDBC や SQLJ を使用するメソッドにより保持し、Bean に (Bean 管理の永続エンティティ Bean 内で) コード化されます。

同期インタフェースを実装すると、セッション Bean のデータの格納および取出しを自動化できます。

セッション Bean

セッション Bean はクライアントで作成され、通常は、そのクライアントに固有となります。Oracle8i では、複数のクライアントが 1 つのセッション Bean を共有できます。

セッション Bean は、サーバーのクラッシュやネットワーク障害の後には存続しないという意味で一時的なものです。以前に存在していた Bean をクラッシュ後にインスタンス化しても、前のインスタンスの状態はリストアされません。状態は、エンティティ Bean にのみリストアできます。

ステートフル・セッション Bean

ステートフル・セッション Bean では、メソッド・コールから次のメソッド・コールまでの間、状態が維持されます。たとえば、セッション Bean の 1 つのインスタンスで JDBC データベース接続をオープンし、この接続を使用してデータベースから初期のデータを取り出すことができます。たとえば、ショッピング・カート・アプリケーション Bean は、活性化されると同時に、顧客プロフィールをデータベースからロードできます。これにより、そのプロフィールは Bean 内のメソッドから使用できるようになります。

代表的なステートフル・セッション EJB は、比較的多目的のオブジェクトです。1 つの Bean でほとんど常に複数のメソッドを含んでおり、それらのメソッドにより統一された論理的なサービスが提供されます。たとえば、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側を実装するセッション EJB には、購入可能な商品のリストを戻すメソッド、顧客のショッピング・カートに商品を入れるメソッド、注文するメソッド、顧客のプロフィールを変更するメソッドなどが実装されるはずです。

セッション Bean が維持している状態のことを、Bean の「カンパセショナル・ステート」と呼びます。これは、Bean が、電話による会話（カンパセション）のように、1 つのクライアントとの接続を維持しているためです。

Bean の状態は Bean 自体においては依然として一時的なデータであることに注意してください。クライアントから Bean への接続が切れると、その状態は失われる場合があります。これはクライアントがタイムアウト前に再接続できるかどうかによって依存します。

エンティティ Bean

エンティティ Bean は、サーバーのクラッシュやネットワーク障害の後にも存続するという意味で永続的なものです。エンティティ Bean が再びインスタンス化されると、前のインスタンスの状態が自動的にリストアされます。エンティティ Bean の詳細は、4-2 ページの「[エンティティ Bean の定義](#)」を参照してください。

Enterprise JavaBeans

この章では、Oracle8i Server 環境で Enterprise JavaBeans を開発し、配置する方法を説明します。EJB と EJB アーキテクチャに関する完全なチュートリアルではありませんが、この章の説明を読めば、EJB アプリケーションの開発を開始できます。

この章では、次のトピックを説明します。

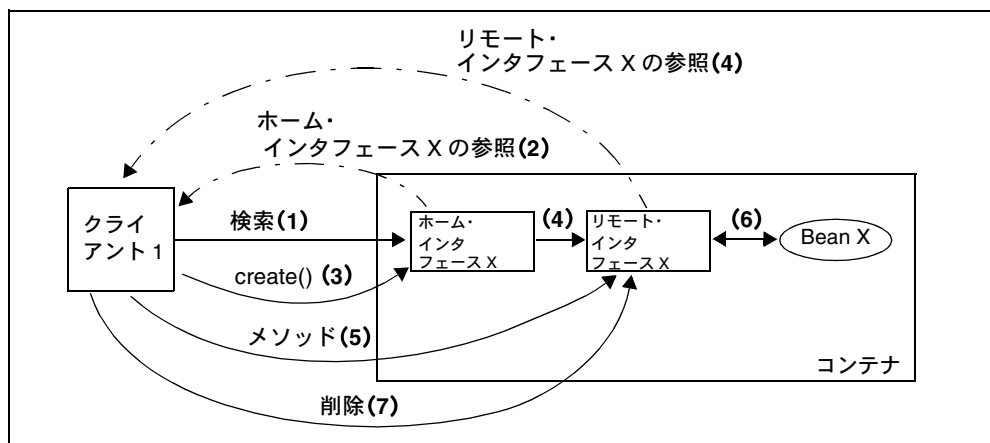
- [Enterprise JavaBeans の起動](#)
- [Enterprise JavaBeans の作成](#)
- [Bean の実装](#)
- [クライアント・アプリケーションの開発](#)
- [EJB の配置](#)
- [プログラミングの制限事項](#)
- [デバッグ手法](#)

Enterprise JavaBeans の起動

Enterprise JavaBean には、リモート・インタフェースおよびホーム・インタフェースという 2 つのクライアント・インタフェースがあります。リモート・インタフェースでは、オブジェクトのクライアントで起動できるメソッドを指定します。ホーム・インタフェースは、クライアントでのオブジェクトの作成方法を定義し、そのオブジェクトの参照を戻します。クライアントでは、Bean に対するメソッドの起動時に、両方のインタフェースが使用されます。

次の図とステップでは、クライアントが Bean 内のメソッドを起動するときに発生するイベントについて説明します。

図 2-1 セッション Bean のライフ・サイクルにおけるイベントの発生順序



図中の番号は、次の各ステップの番号に対応しています。

1. クライアント 1 が、Bean X のホーム・インタフェースを検索します。
JNDI 検索中に、データベースにより要求のサーバー側セッションが作成されます。
2. クライアント 1 に、ホーム・インタフェース X の参照が戻されます。
3. クライアント 1 が、ホーム・インタフェース X で create メソッドを起動します。
Bean インスタンスは、JNDI 検索時に確立されたセッション内で作成されます。
4. ホーム・インタフェース X が、リモート・インタフェース X をインスタンス化します。
コンテナはこのクライアント用の Bean をインスタンス化し、クライアントが削除メソッドを起動するときのみ破棄されます。
クライアント 1 に、リモート・インタフェース X のオブジェクト参照が戻されます。

5. クライアント 1 が、リモート・インタフェース X を使用して、Bean インスタンス X 上のメソッドを起動します。
6. リモート・インタフェース X が、Bean にコールを移譲します。
7. クライアント 1 は、Bean インタフェースでの終了後に、リモート・インタフェース X 上の `remove` メソッドを起動します。これにより、リモート・インタフェースと Bean インスタンスが破棄されます。

Enterprise JavaBeans の作成

EJB を作成するには、次のステップを実行する必要があります。

1. Bean 用のリモート・インタフェースを作成します。リモート・インタフェースでは、クライアントで起動できるメソッドを宣言します。このインタフェースでは、`javax.ejb.EJBObject` を拡張 (`extends`) する必要があります。
2. Bean 用のホーム・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、Bean 用の `create` メソッドも定義します。
3. Bean を実装します。これには、次の要素が含まれます。
 - a. リモート・インタフェースで宣言されたメソッドの実装。
 - b. `javax.ejb.SessionBean` または `javax.ejb.EntityBean` インタフェース内で定義されたメソッド。各種 Bean の違いについては、4-2 ページの「[エンティティ Bean の定義](#)」を参照してください。
 - c. ホーム・インタフェースで定義済みの `create` メソッドと一致するパラメータを持つ `ejbCreate` メソッド。
4. Bean ディプロイメント・ディスクリプタを作成します。このディプロイメント・ディスクリプタでは、Bean のプロパティを指定します。4-23 ページの「[エンティティ Bean の配置](#)」を参照してください。
5. Bean、リモート・インタフェースとホーム・インタフェースおよびディプロイメント・ディスクリプタを含む `ejb-jar` ファイルを作成します。この `ejb-jar` ファイルでは、アプリケーション内の Bean をすべて定義する必要があります。詳細は、4-23 ページの「[エンティティ Bean の配置](#)」を参照してください。

リモート・インタフェースとホーム・インタフェースの実装要件

要件	説明
RMI への準拠	<p>javax.ejb.EJBObject および javax.ejb.EJBHome インタフェースにより、java.rmi.Remote インタフェースが拡張されるため、この 2 つのインタフェースは Remote Method Invocation (RMI) 仕様に準拠する必要があります。これは、それぞれのメソッドで使用できるのは RMI で許されるデータ型のみであることと、どちらのインタフェース内のメソッドでも java.rmi.RemoteException 例外が発生する必要があることを意味します。</p> <p>RMI 仕様書は、JavaSoft のサイト、http://www.javasoft.com からアクセスできます。</p>
ネーミング規則	<p>これらのインタフェース内で定義するインタフェース名、メソッド名および定数は、アンダースコア (_) で始めることはできず、ドル記号 (\$) は使用できません。また、アプリケーション名と Bean 名には、スラッシュ記号 (/) を使用できます。</p>

リモート・インタフェースの作成

Bean のリモート・インタフェースは、クライアントで起動されるメソッド用のインタフェースを提供します。つまり、リモート・インタフェースでは、リモート・アクセス用に実装するメソッドを定義します。

1. Bean のリモート・インタフェースでは、javax.ejb.EJBObject インタフェースを拡張する必要があります。その定義は次のとおりです。

```
public interface javax.ejb.EJBObject extends java.rmi.Remote {
    public abstract EJBHome getEJBHome()
        throws java.rmi.RemoteException // returns reference to home
    // interface for this bean
    public abstract Handle getHandle()
        throws java.rmi.RemoteException // returns serializeable handle
    public abstract Object getPrimaryKey()
        throws java.rmi.RemoteException // returns key to an entity bean
    public abstract boolean isIdentical(EJBObject obj)
        throws java.rmi.RemoteException
    public abstract void remove()
        throws java.rmi.RemoteException, RemoveException //remove EJB object
}
```

EJBObject インタフェース内のメソッドを実装する必要はありません。これらのメソッドはコンテナにより実装されます。

機能	説明
<code>getEJBHome()</code>	特定の Bean に対応付けられたホーム・インタフェースのオブジェクト参照を取得します。 戻されるオブジェクトはホーム・インタフェースの型に型キャストできないため注意する必要があります。詳細は、2-13 ページの「 getEJBHome メソッドの使用 」を参照してください。
<code>getHandle()</code>	リモート・インタフェースの <code>getHandle</code> メソッドを使用すると、EJB オブジェクト参照のシリアライズ可能な Java 表現を取得できます。ハンドルをシリアライズし、同じオブジェクトへの接続の再確立に使用できます。セッション Bean では、その Bean インスタンスがアクティブである限り、接続が再確立されます。エンティティ Bean では、Bean がアクティブかどうかに関係なく接続が再確立されます。 Handle クラス内で <code>getEJBObject</code> メソッドを使用すると、Bean インスタンスを取得できます。
<code>getPrimaryKey()</code>	<code>getPrimaryKey</code> メソッドは、EJB エンティティ Bean に対応付けられた主キーを取得します。
<code>isIdentical()</code>	このメソッドをコールするオブジェクトと、引数に指定されたオブジェクトが同一かどうかを検査します（コンテナが重要な場合）。これにより、両方のオブジェクトがすべての用途に同一であることが識別されます。
<code>remove()</code>	EJB Bean を非活性化します。これにより、セッション Bean インスタンスが破棄されます（ステートフルの場合）。

- リモート・インタフェース内の各メソッドのシグネチャは、Bean 実装におけるシグネチャと一致している必要があります。
- リモート・インタフェースは `public` として宣言する必要があります。
- リモート・インタフェースではパブリック変数を宣言しません。宣言するのはパブリック・メソッドのみです。
- 例外は、シリアライズ可能である限り、クライアントに通知することができます。ランタイム例外は、リモート・ランタイム例外としてクライアントに転送されます。

例

次のサンプル・コードは、Bean に実装される `getEmployee` メソッドを宣言するリモート・インタフェース **Employee** を示しています。

```
package employee;

import employee.EmpRecord;
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Employee extends EJBObject {
    public EmpRecord getEmployee (int empNumber)
        throws java.sql.SQLException, EmpException, RemoteException;
}
```

ホーム・インタフェースの作成

ホーム・インタフェースでは、Bean 用に適切な `create` メソッドを定義する必要があります。このインタフェースでは1つ以上の `create` メソッドを指定します。`create` メソッドごとに、対応する `ejbCreate` メソッドをリモート・インタフェース内で定義する必要があります。`create` メソッドはいずれも Bean の型を戻しますが、`ejbCreate` メソッドはいずれも `void` を戻します。

クライアントは、ホーム・インタフェース内で宣言された `create` メソッドを起動します。コンテナは Bean 実装に対し、適切なパラメータ・シグネチャを持つ `ejbCreate` メソッドをコールします。パラメータ引数を使用すると、新規 EJB オブジェクトの状態を初期化できます。

1. 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。その定義は次のとおりです。

```
public interface javax.ejb.EJBHome extends java.rmi.Remote {
    public abstract EJBMetaData getEJBMetaData();
    public abstract void remove(Handle handle);
    public abstract void remove(Object primaryKey);
}
```

`EJBHome` インタフェース内のメソッドは、コンテナにより実装されます。クライアントは、ホーム・インタフェースまたはリモート・インタフェース内で定義されている `remove` メソッドを使用して、EJB オブジェクトを削除できます。

2. また、Bean のホーム・インタフェースを使用すると、特定のハンドルを指定し、`javax.ejb.EJBMetaData` インタフェースを介して Bean のメタデータ情報を取得したり、Bean インスタンスを削除できます。

3. すべての create メソッドでは、次の例外が発生する必要があります。

- javax.ejb.CreateException
- java.rmi.RemoteException または javax.ejb.EJBException

注意： deployejb ツールでは、データベース内のホーム・オブジェクトへの参照が公開されます。deployejb の詳細は、『Oracle8i Java Tools リファレンス』を参照してください。

例

次のサンプル・コードは、**EmployeeHome** というホーム・インタフェースを示しています。**create** メソッドには、引数は含まれていません。

```
package employee;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface EmployeeHome extends EJBHome {
    public Employee create()
        throws CreateException, RemoteException;
}
```

例外クラスの作成

Employee クラスの一部のメソッドでは、EmpException 例外が発生することがあります。オブジェクトからクライアントに送信される例外用に、クラスを定義する必要があります。

次の例は例外クラスを定義するコードで、**EmpException.java** に含まれています。

```
package employee;

public class EmpException extends RemoteException
{
    public EmpException(String msg)
    {
        super(msg);
    }
}
```

Bean の実装

Bean には、そのビジネス・ロジックが含まれています。実装されるメソッドは、次のとおりです。

- リモート・インタフェース内で宣言された Bean メソッド。
アプリケーション例の Bean は、従業員情報を取得するクラス `EmployeeBean` のみからなっています。
- `SessionBean` または `EntityBean` インタフェース内で宣言されたメソッド。
- ホーム・インタフェース内で宣言された `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、コンテナはそれに対応する `ejbCreate` メソッドを起動します。

実装されるインタフェース

メソッドは、Bean によって `SessionBean` または `EntityBean` インタフェース内で実装されます。次の例では、`SessionBean` インタフェースが実装されます。基本的に、セッション Bean は、プロセス指向の Bean、つまりあるタスクを達成するための Bean に使用されます。エンティティ Bean は、永続データの周囲に編成される複雑なリモート・オブジェクトです。2 種類の Bean の違いの詳細は、4-2 ページの「[エンティティ Bean の定義](#)」を参照してください。

セッション Bean では、`javax.ejb.SessionBean` インタフェースが実装されます。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、最低でも次のメソッドを実装する必要があります。

<code>ejbActivate()</code>	これは、カレント・リリースの EJB サーバーではコールされないため、NULL メソッドとして実装します。
<code>ejbPassivate()</code>	これは、カレント・リリースのサーバーではコールされないため、NULL メソッドとして実装します。
<code>ejbRemove()</code>	コンテナで、セッション・オブジェクトの存続を終了する前にこのメソッドがコールされます。このメソッドでは、必要なクリーン・アップが実行されます。たとえば、ファイル・ハンドルなどの外部リソースがクローズされます。

```
setSessionContext
(SessionContext ctx)
```

Bean のインスタンスをコンテキスト情報に対応付けます。コンテナでは、Bean の作成後にこのメソッドがコールされます。エンタープライズ Bean では、トランザクション管理に使用するコンテキスト・オブジェクトへの参照がインスタンス変数に格納できます。自分のトランザクションを管理する Bean では、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

このメソッドは、セッション Bean インスタンスでそのコンテキストの参照を保存するために使用されます。セッション Bean には、セッション・コンテキストというものがあり、これはコンテナが維持し、コンテナによって Bean に使用可能になります。Bean は、セッション・コンテキスト内のメソッドを使用して、コンテナへ要求をコールバックできます。

コンテナは、最初に Bean をインスタンス化してから setSessionContext メソッドを起動し、Bean でセッション・コンテキストを取得できるようにします。コンテナが、このメソッドをトランザクション・コンテキストからコールすることはありません。Bean がこの時点でセッション・コンテキストを保存しなければ、それ以後はセッション・コンテキストにアクセスできません。

コンテナは、このメソッドをコールするときに、SessionContext オブジェクトの参照を Bean に渡します。Bean は、その参照を後で使用できるように格納できます。次の例は、セッション・コンテキストを **sessctx** 変数に保存する Bean を示しています。

```
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 インタフェースには、次の定義があります。

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

また、javax.ejb.EJBContext インタフェースには、次の定義があります。

```
public interface EJBContext {
    public abstract Properties      getEnvironment();
    public abstract UserTransaction getUserTransaction();
    public abstract boolean        getRollbackOnly();
    public abstract void            setRollbackOnly();
    public abstract boolean        isCallerInRole(Identity);

                                // not supported
    public abstract Identity        getCallerIdentity(); // not supported
    public abstract EJBHome        getEJBHome();
}
```

Bean で表 2-1 に示す操作を実行する場合は、セッション・コンテキストが必要です。

表 2-1 SessionContext の操作

メソッド	説明
getEnvironment()	Bean のプロパティ 値を取得します。
getUserTransaction()	トランザクションをプログラムでデマークートできるように、トランザクション・コンテキストを取得します。これは、トランザクションを使用するように設計された Bean にのみ有効です。
setRollbackOnly()	現行トランザクションをコミットできないように設定します。
getRollbackOnly()	現行トランザクションにロールバック専用マークが付いているかどうかをチェックします。
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。

Bean 実装の例

次のコードでは、セッション Bean **EmployeeBean** のメソッドが実装されます。**SessionBean** インタフェースのメソッドは、リモート・インタフェースで宣言されているパブリック・メソッドとともに実装されます。

JDBC コードはデフォルトの接続をオープンします。これは、Oracle8i Server 上で実行される JDBC コードがサーバー側でのデータベース接続をオープンするときの標準的な方法です。JDBC のプリコンパイル済みの SQL 文を使用して、WHERE 句を持つ問合せを準備します。次に、setInt メソッドを使用して、getEmployee メソッドの empNumber 入力パラメータをプリコンパイルされた SQL 文問合せの「?」プレースホルダに対応付けます。これは、クライアント・アプリケーションで作成する JDBC コードと同じです。

```
package employeeServer;

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

public class EmployeeBean implements SessionBean {
    SessionContext ctx;

    //implement the bean method, getEmployee
    public EmpRecord getEmployee (int empNumber)
        throws SQLException, RemoteException {

        //create a new employee record
        EmpRecord empRec = new EmpRecord();

        //establish a connection to the database using JDBC
        Connection conn =
            new oracle.jdbc.driver.OracleDriver().defaultConnection();

        //retrieve the employee's information from the database
        PreparedStatement ps =
            conn.prepareStatement("select ename, sal from emp where empno = ?");
        ps.setInt(1, empNumber);
        ResultSet rset = ps.executeQuery();
        if (!rset.next())
            throw new RemoteException("no employee with ID " + empNumber);
        empRec.ename = rset.getString(1);
        empRec.sal = rset.getFloat(2);
        empRec.empno = empNumber;
        ps.close();
        return empRec;
    }

    //implement the SessionBean methods: ejbCreate, ejbActivate,
    // ejbPassivate, ejbRemove and setSessionContext

    //implement ejbCreate, which is called by the container when
    //the Home create is invoked by the client.
    public void ejbCreate() throws CreateException, RemoteException {
        //you can do any initialization for the bean at this point.
        //this particular example does not require any initialization or
        //environment variable retrieval.
    }

    //ejbActivate and ejbPassivate are never called in this release.
    //Both methods should be declared, but be null methods.
}
```

```
public void ejbActivate() {  
}  
public void ejbPassivate() {  
}  
  
//implement anything that needs to be done before the  
//bean is destroyed. this would include closing any open  
//resources. however, for this example, no open resources need  
//to be closed. thus, the method is empty.  
public void ejbRemove() {  
}  
  
//retrieve the session context  
public void setSessionContext(SessionContext ctx) {  
    this.ctx = ctx;  
}  
}
```

クライアント・アプリケーションの開発

すべての EJB クライアントは、次の操作を実行して Bean をインスタンス化し、そのメソッドを起動し、Bean を破棄します。

1. Bean のホーム・インタフェースを取得します。ホーム・インタフェースは、Bean 配置プロセスの一部をととして Oracle8i データベース上に公開されています。Java Naming and Directory Interface (JNDI) を使用して、ホーム・インタフェースを得ます。
2. 得られたホーム・インタフェースを介して、サーバー内に Bean のインスタンスを作成します。ホーム・インタフェース上で create メソッドを起動すると、新規 Bean がインスタンス化されます。これにより、Bean のリモート・インタフェースに Bean 参照が戻されます。
3. リモート・インタフェース内で定義されたメソッドを起動します。コンテナにより、インスタンス化された Bean に要求が転送されます。
4. Bean が不要になった場合は、remove メソッドを起動して Bean を破棄します。

これらのステップは、[図 2-1](#) の例に示されています。

単純な例として、EmployeeHome を、Employee という Bean のホーム・インタフェースから取得した参照であるとします。Employee ホーム・インタフェースには、Bean をインスタンス化できるように、最低 1 つは create メソッドが必要です。次のようにコーディングして、リモート・サーバー上にこの Bean の新しいインスタンスを作成します。

```
Context ic = new InitialContext(env);  
  
EmployeeHome home =  
    (EmployeeHome) ic.lookup(serviceURL + objectName); // lookup the bean  
Employee testBean = home.create(); // create a bean instance
```

次に、通常の構文を使用して `Employee` のメソッドを起動します。

```
testBean.getEmployee(empNumber);
```

getEJBHome メソッドの使用

`getEJBHome` メソッドを使用し、オブジェクト参照を指定してホーム・インタフェースを取得する場合、戻されたオブジェクトをホーム・インタフェースの型にキャストできません。かわりに、戻されるオブジェクトは `org.omg.CORBA.Object` 型となります。取得後のオブジェクトは、`Helper.narrow` メソッドを介して適切なホーム・インタフェースの型にキャストされます。次の例は、JNDI を使用して `Hello` のホーム・インタフェースを取得し、リモート・インタフェースを作成し、後で `getEJBHome` インタフェースを使用してホーム・インタフェースを再度取得する `Hello` の例を示しています。ホーム・インタフェースを適切に型キャストするために、`HelloHomeHelper.narrow` メソッドが使用されていることに注意してください。

```
HelloHome hello_home = (HelloHome)ic.lookup (serviceURL + objectName);
Hello hello = hello_home.create ();
System.out.println (hello.helloWorld ());

org.omg.CORBA.Object newHome = (org.omg.CORBA.Object) hello.getEJBHome();
HelloHome newHello = HelloHomeHelper.narrow(newHome);
```

パラメータの受渡し

EJB の実装時、あるいは EJB メソッドをコールするクライアント・コードの作成時には、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や Bean メソッドから受け取る戻り値には、シリアル化可能なすべての Java 型が使用できます。Java の基本型 (`int`、`double`) はシリアル化可能です。 `java.io.Serializable` インタフェースを実装している非リモート・オブジェクトを渡すことができます。Bean にパラメータとして渡されたり、Bean から戻された非リモート・オブジェクトは、参照ではなく、値で渡されます。したがって、たとえば、Bean メソッドを次のようにしてコールすると、

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

Bean の `method1()` は `theNumber` のコピーを受け取ります。Bean によりサーバー上で `theNumber` オブジェクトの値が変更された場合、値渡し方法であるため、この変更はクライアントに反映されません。

非リモート・オブジェクトが複雑な場合、たとえば、クラスに複数のフィールドがある場合には、非 `static` および非 `transient` フィールドのみがコピーされます。

リモート・オブジェクトをパラメータとして渡す場合には、そのリモート・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 へのパラメータ受渡しと戻り値としてのリモート・オブジェクトを説明します。

パラメータ・オブジェクト

`EmployeeBean` の `getEmployee` メソッドでは、`EmpRecord` オブジェクトが戻されるため、アプリケーションのどこかでこのオブジェクトが定義されている必要があります。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。

このクラスは、**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渡しで戻せるように、`java.io.Serializable` インタフェースが実装される必要があります。宣言は次のとおりです。

```
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、メソッドが指定されません。単に、クラスがシリアライズ可能であることが示されます。したがって、`EmpRecord` クラスで特別にメソッドを実装する必要はありません。

クライアント・コード

この項では、前述の Bean の例にメッセージを送信し、Bean から戻される結果を取得して出力するためのクライアント・コードを示します。このクライアント・コードは、クライアントの次のような動作を説明するためのものです。

- Bean ホーム・インタフェースなどのリモート・オブジェクトを見つける動作
- サーバーに対し自分自身の認証を行う動作
- Bean のインスタンスを活性化する動作
- Bean 上のメソッドを起動する動作

リモート・オブジェクトのロケーティング

RMI、EJB または CORBA のどれであっても、リモート・オブジェクトの実装の最初のステップは、リモート・オブジェクトを見つける方法を調べる作業です。リモート・オブジェクト参照を取得するには、次の情報が必要です。

- オブジェクトの名前
- ネーム・サーバーが置かれている場所

EJB では、初期オブジェクト名は EJB ホーム・インタフェースの名前です。これは、Java Naming and Directory Interface (JNDI) を使用して見つけます。EJB の仕様では、EJB 実装はロケーティングの手段として JNDI インタフェースを使用する必要があります。

JNDI について

JNDI は、ネーミング・サービスおよびディレクトリ・サービスとのインタフェースです。たとえば、JNDI をファイル・システムへのインタフェースとして使用し、ディレクトリやディレクトリに置かれているファイルの検索に利用できます。あるいは、JNDI をネーミング・サービスまたはディレクトリ・サービスへのインタフェースとして使用できます。たとえば、LDAP などのディレクトリ・プロトコルです。

この項では、JNDI を簡単に説明します。EJB 仕様では、リモート・オブジェクトの名前によるロケーティングのために JNDI を使用するよう規定しています。

この項では、Oracle8i の EJB アプリケーションを作成するために知っておく必要のある JNDI に関する情報の一部を説明します。JNDI API（および SPI）仕様書を取得するには、<http://www.javasoft.com/products/jndi> にアクセスします。

Sun Microsystems は JNDI を javax.naming パッケージで提供します。このため、クライアント・コードで次のように指定し、これらのクラスをインポートする必要があります。

```
import javax.naming.*;
```

Oracle8i EJB サーバーでは、JNDI は OMG CosNaming サービスへのインタフェース（SPI ドライバ）として使用されます。しかし、Oracle8i Server 用に EJB を作成して配置するだけであれば、CosNaming をすべて知っている必要はなく、また JNDI もすべて知っている必要はありません。作業を開始するために必要な情報は、永続的に格納されるホーム・インタフェース・オブジェクトへのアクセスに使用する JNDI メソッドの使用法と、JNDI Context オブジェクトの環境の設定方法のみです。

この JNDI の項の残りで、EJB オブジェクトにアクセスするために必要な javax.naming パッケージのデータ構造とメソッドを説明します。

初期コンテキストの取得

JNDI を使用して、Context オブジェクトを取得します。取得する最初の Context オブジェクトは、Oracle8i が公開するコンテキストのルート・ネーミング・コンテキストにバインドされています。EJB ホーム・インタフェースはデータベース内に公開され、ファイル・システムに類似した階層構造で配列されます。publish ツールの詳細は、『Oracle8i Java Tools リファレンス』を参照してください。

ルート・ネーミング・コンテキストを取得するには、次のように、新しい JNDI InitialContext を作成します。

```
Context initialContext = new InitialContext(environment);
```

environment パラメータは Java のハッシュ表です。表 2-2 に、このハッシュ表内で設定できる、javax.naming.Context に渡される 6 つのプロパティを示します。

表 2-2 コンテキストのプロパティ

プロパティ	目的
javax.naming.Context. URL_PKG_PREFIXES	URL コンテキスト・ファクトリ内にロードするときに使用するパッケージ接頭辞のリストを指定する環境プロパティ。このプロパティには "oracle.aurora.jndi" という値を使用する必要があります。
javax.naming.Context. SECURITY_AUTHENTICATION	データベース接続のセキュリティの種類。可能な値は次のとおりです。 <ul style="list-style-type: none">■ oracle.aurora.sess_iiop.ServiceCtx. NON_SSL_LOGIN■ oracle.aurora.sess_iiop.ServiceCtx. SSL_CREDENTIAL■ oracle.aurora.sess_iiop.ServiceCtx. SSL_LOGIN■ oracle.aurora.sess_iiop.ServiceCtx. SSL_CLIENT_AUTH
javax.naming.Context. SECURITY_PRINCIPAL	Oracle8i ユーザー名。(例: "SCOTT")
javax.naming.Context. SECURITY_CREDENTIALS	ユーザー名に対するパスワード。(例: "TIGER")
oracle.aurora.sess_iiop. ServiceCtx.SECURITY_ROLE	接続のためのデータベース・ロールを指定するオプションのプロパティ。たとえば、"CLERK" という文字列を使用して、CLERK ロールで接続します。
oracle.aurora.sess_iiop. ServiceCtx.SSL_VERSION	クライアント側の SSL バージョン・ナンバー。

JNDI および Oracle8i インスタンスへの接続の詳細は、[第 5 章「JNDI 接続とセッション IIOP サービス」](#)を参照してください。

ホーム・インタフェース・オブジェクトの取得

初期参照コンテキストを取得した後、そのメソッドを起動して、EJB ホーム・インタフェースへの参照を取得できます。そのためには、オブジェクトの公開されているフルパス名、オブジェクトが置かれているホスト・システム、そのシステム上のリスナーの IIOP ポート、およびデータベース識別子 (SID) がわかっている必要があります。これらの情報を EJB 実行者などから取得した後、次の構文を使用して URL を構成します。

```
<service_name>://<hostname>:<iiop_listener_port>:<SID>/<published_obj_name>
```

たとえば、TCP/IP ホスト名が myHost、リスナー IIOP ポートが 2481、システム識別子 (SID) が ORCL であるシステム上で /test/myEmployee として公開されている Bean のホーム・インタフェースへの参照を取得するには、URL を次のように構成します。

```
sess_iiop://myHost:2481:ORCL/test/myEmployee
```

IIOP リクエストのリスナー・ポートは、listener.ora ファイルで設定されています。Oracle8i のデフォルト値は 2481 です。IIOP 構成情報の詳細は『Oracle8i Net8 管理者ガイド』を参照してください。IIOP 接続の詳細は、[第 5 章「JNDI 接続とセッション IIOP サービス」](#)も参照してください。

ホーム・インタフェースを取得するには、初期コンテキスト上で lookup メソッドを使用し、パラメータとして URL を渡します。たとえば、ホーム・インタフェースの公開された名前が /test/myEmployee の場合、次のようにコーディングします。

```
...
String ejbURL = "sess_iiop://localhost:2481:ORCL/test/myEmployee";
Hashtable env = new Hashtable();
env.put(javax.naming.Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
// Tell sess_iiop who the user is
env.put(Context.SECURITY_PRINCIPAL, "SCOTT");
// Tell sess_iiop what the password is
env.put(Context.SECURITY_CREDENTIALS, "TIGER");
// Tell sess_iiop to use non-SSL login authentication
env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
// Lookup the URL
EmployeeHome home = null;
Context ic = new InitialContext(env);
home = (EmployeeHome) ic.lookup(ejbURL);
...
```

EJB メソッドの起動

Bean のホーム・インタフェースを取得後に、Bean の create メソッドの 1 つを起動して Bean をインスタンス化できます。実行権限を付与する方法は、[第 5 章「JNDI 接続とセッション IIOP サービス」](#) を参照してください。たとえば、次のようになります。

```
Employee testBean = home.create();
```

次に、EJB のメソッドを通常の方法で起動します。

```
int empNumber = 7499;  
EmpRecord empRec = testBean.getEmployee(empNumber);
```

クライアント・アプリケーションの完全なコードを次に示します。

```
import employee.Employee;  
import employee.EmployeeHome;  
import employee.EmpRecord;  
  
import oracle.aurora.jndi.sess_iiop.ServiceCtx;  
  
import javax.naming.Context;  
import javax.naming.InitialContext;  
import java.util.Hashtable;  
  
public class Client {  
  
    public static void main (String [] args) throws Exception {  
  
        String serviceURL = "sess_iiop://localhost:2481:ORCL";  
        String objectName = "/test/myEmployee";  
        int empNumber = 7499;    // ALLEN  
        Hashtable env = new Hashtable();  
  
        env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");  
        env.put(Context.SECURITY_PRINCIPAL, "scott");  
        env.put(Context.SECURITY_CREDENTIALS, "tiger");  
        env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);  
  
        Context ic = new InitialContext(env);  
  
        EmployeeHome home =  
            (EmployeeHome) ic.lookup(serviceURL + objectName); // lookup the bean  
        Employee testBean = home.create(); // create a bean instance  
        EmpRecord empRec =  
            testBean.getEmployee(empNumber); // get the data and print it
```

```
        System.out.println("Employee name is " + empRec.ename);  
        System.out.println("Employee sal is " + empRec.sal);  
    }  
}
```

EJB の配置

EJB 配置プロセスは、次のステップから成り立っています。

1. EJB 開発者から Bean を取得します。通常の場合、Bean をコンパイルし、ホーム・インタフェース、リモート・インタフェース、および Bean に依存するクラスなど、Bean とその関連クラスを JAR ファイル（1 つの Bean に対し 1 つの JAR ファイル）にまとめます。
2. Bean ごとに 1 つの EJB ディプロイメント・ディスクリプタを作成します。
3. Oracle 固有のディプロイメント・ディスクリプタを作成します。
4. `deployejb` ツールを実行します。次の作業が実行されます。
 - a. ディプロイメント・ディスクリプタと Bean JAR ファイルを読み込みます。
 - b. EJB ディプロイメント・ディスクリプタに定義されている論理名を、既存の JNDI 名およびデータベース表にマップします。
 - c. Bean クラスを Oracle8i データベースにロードします。
 - d. Bean ホーム・インタフェースを公開します。
5. アプリケーション開発者に対し Bean リモート・インタフェースおよび公開されている Bean の名前に関する必要な情報を知らせます。

注意： 配置プロセスは、*iAS* の場合も Oracle8i JVM の場合と同じです。配置用に適切なサーバーを指定するだけです。中間層に配置するか、データベース・バックエンドに配置するかを選択できます。中間層に配置する場合は、その層に *iCache* または Oracle8i JVM をインストールする必要があります。その後、インストール済み *iCache* の URL または JVM の URL をコマンドライン・ツールに渡します。

配置の手順

EJB をパッケージ化するためのフォーマットが EJB 仕様により定義されています。この項では、EJB 開発者と EJB 実行者が EJB のコンパイル、パッケージ化および配置を実行するステップを説明します。Oracle8i には、配置ツールとして `deployejb` が用意されており、このツールにより EJB の配置に必要なステップの大部分が自動的に実行されます。`deployejb` ツールは、Bean を一度に 1 つしか配置しません。このツールについては、『Oracle8i Java Tools リファレンス』を参照してください。

EJB を配置するには、次の 4 つのステップに従ってください。

1. Bean のコードをコンパイルします。次のものが含まれます。

- ホーム・インタフェース
- リモート・インタフェース
- Bean 実装
- Bean 実装クラスが依存しているすべての Java ソース・ファイル（この依存性は通常、Java コンパイラにより検出されます）

標準のクライアント側コンパイラを使用して、Bean のソース・ファイルをコンパイルします。Bean は通常、1 つ以上の Java ソース・ファイルから成り立っており、また関連するリソース・ファイルが付属する場合があります。

Oracle8i では、Sun Microsystems の Java Developer's Kit バージョン 1.1.6 または 1.2 のコンパイラをサポートします。また、JCK テスト済みの他の Java コンパイラを使用して、Oracle8i Server で実行する EJB を作成できる場合もあります。

2. EJB 用の XML ディプロイメント・ディスクリプタを作成します。ディプロイメント・ディスクリプタの作成方法の詳細は、2-27 ページの「[プログラミングの制限事項](#)」を参照してください。
3. Oracle の配置マッピング・ファイルを作成します。
4. Bean のインタフェース部および実装部のクラス・ファイル（ホーム・インタフェース、リモート・インタフェースおよび Bean 実装）で構成される JAR ファイルを作成します。他にも依存クラスおよびリソース・ファイルがある場合には、それらのクラスおよびファイルに対して別途 JAR ファイルを作成することをお勧めします。この JAR ファイルは、`deployejb` ツールで入力ファイルとして使用されます。
5. `deployejb` コマンドライン・ツールを実行して、JAR ファイルにまとめられた Bean をロードし、公開します（`deployejb` の詳細は、『Oracle8i Java Tools リファレンス』を参照してください）。

ディプロイメント・ディスクリプタの作成

EJB 1.1 では、ディプロイメント・ディスクリプタの定義に XML を使用します。Sun Microsystems は、Bean とアプリケーションの定義に必要なエントリを記述する DTD ファイルを提供しています。ディプロイメント・ディスクリプタは、論理名を含むように設計されています。論理名は、Oracle8i JVM にロードされているオブジェクトの実際の名前と一致するとは限りません。これらの論理名は、Oracle 用の配置マッピング・ファイルであるもう 1 つの配置ファイルを介して既存の名前にマップされます。Oracle の配置マッピング・ファイルでは、論理 Bean 名を既存の JNDI 名にマップし、コンテナ管理のエンティティ Bean のフィールドを既存のデータベース列にマップできます。

また、XML 配置ファイル内で実際の JNDI 名やデータベース列名を使用すると、`deployejb` により Oracle 配置マップ・ファイルが自動的に作成されます。

注意： この章の説明はセッション Bean を対象としているため、ここではセッション Bean に関連するフィールドについてのみ説明します。XML 要素の詳細は、付録 A 「XML ディプロイメント・ディスクリプタ」、または <http://www.javasoft.com> にある EJB 1.1 仕様書の「Deployment Descriptor」を参照してください。

次の例は、Employee の例に必要なセクションを示しています。この例では、Oracle 配置マップ・ファイル内の JNDI 名にマップする、XML ディプロイメント・ディスクリプタ内の論理名を使用しています。

例 2-1 Employee Bean の XML ディプロイメント・ディスクリプタ

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems Inc.//DTD Enterprise JavaBeans 1.1
//EN" "ejb-jar.dtd">
<ejb-jar>
  <enterprise-beans>
    <session>
      <description>Session Bean Employee Example</description>
      <ejb-name>Employee</ejb-name>
      <home>employee.EmployeeHome</home>
      <remote>employee.Employee</remote>
      <ejb-class>employee.EmployeeBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
    <security-role>
      <description>Public</description>
      <role-name>PUBLIC</role-name>
```

```

</security-role>
<method-permission>
  <description>public methods</description>
  <role-name>PUBLIC</role-name>
  <method>
    <ejb-name>EmployeeBean</ejb-name>
    <method-name>*</method-name>
  </method>
</method-permission>
<container-transaction>
  <description>no description</description>
  <method>
    <ejb-name>EmployeeBean</ejb-name>
    <method-name>*</method-name>
  </method>
  <trans-attribute>Supports</trans-attribute>
</container-transaction>
</assembly-descriptor>
</ejb-jar>

```

次の項では、XML ディプロイメント・ディスクリプタの各部分について説明します。

XML バージョン・ナンバー

```
<?xml version="1.0"?>
```

DTD ファイル名

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems Inc.//DTD Enterprise JavaBeans 1.1
//EN" "ejb-jar.dtd">
```

JAR ファイル

最初に宣言される要素は、<ejb-jar> 要素です。この要素内で、<enterprise-beans> および <assembly-descriptor> という 2 つのセクションを定義します。

<enterprise-beans> セクションでは、Bean を定義します。<assembly-descriptor> セクションでは、アプリケーションのセキュリティ属性とトランザクション属性を定義します。

```

<ejb-jar>                                //Start of JAR file descriptor
  <enterprise-beans>                       //EJB Descriptor section
  ...                                     //Bean definition
</enterprise-beans>
  <assembly-descriptor>                   //Application Descriptor section
  ...                                     //Transaction and security definition
</assembly-descriptor>
</ejb-jar>

```

Enterprise JavaBeans の要素

Bean は、`<enterprise-beans>` 要素内で記述されます。この要素には、Bean の種類、ホーム・インタフェース名、リモート・インタフェース名および Bean のクラス名などの情報が含まれています。

後述のセグメントの内容は、次のとおりです。

- Bean はセッション Bean で、`<session>` 要素で示されています。
- この Bean の論理名は `Employee` で、`<ejb-name>` 要素内で定義されています。
- ホーム、リモートおよび Bean クラス名は `employee` パッケージ内にあり、それぞれ `EmployeeHome`、`Employee` および `EmployeeBean` です。
- Oracle8i JVM 内のセッション Bean は、すべてステートフルです。
- トランザクションは、Bean ではなくコンテナにより管理されます。

```
<enterprise-beans>
  <session>
    <description>Session Bean Employee Example</description>
    <ejb-name>Employee</ejb-name>
    <home>employee.EmployeeHome</home>
    <remote>employee.Employee</remote>
    <ejb-class>employee.EmployeeBean</ejb-class>
    <session-type>Stateful</session-type>
    <transaction-type>Container</transaction-type>
  </session>
</enterprise-beans>
```

アセンブリ・ディスクリプタの要素

`<assembly-descriptor>` 要素では、アプリケーションのセキュリティ属性とトランザクション属性を記述します。

- セキュリティのために、このアプリケーションで使用するロールを `<security-role>` 要素内で定義する必要があります。これらのロールは、`<method-permission>` 要素内で Bean の特定のメソッドに割り当てます。この例では、`Employee Bean` により `PUBLIC` 内ですべてのメソッドにアクセスできるようになります。
- トランザクションの場合は、`<container-transaction>` 要素内で、各メソッドに必要なトランザクション・サポートのタイプを定義します。この例では、すべてのメソッドに `Supports` トランザクション属性を必須としています。

```
<assembly-descriptor>
  <security-role>
    <description>Public</description>
    <role-name>PUBLIC</role-name>
  </security-role>
  <method-permission>
    <description>public methods</description>
```

```

    <role-name>PUBLIC</role-name>
    <method>
      <ejb-name>EmployeeBean</ejb-name>
      <method-name>*</method-name>
    </method>
  </method-permission>
</container-transaction>
  <description>no description</description>
  <method>
    <ejb-name>EmployeeBean</ejb-name>
    <method-name>*</method-name>
  </method>
  <trans-attribute>Supports</trans-attribute>
</container-transaction>
</assembly-descriptor>

```

Oracle 配置マップ・ファイルの作成

実際の JNDI 名を <ejb-name> 内で宣言していれば、Oracle 配置マップ・ファイルを作成する必要はありません。この例では、XML 配置ファイル内で論理名 `Employee` を使用しました。また、`Employee` の JNDI 名は `/test/EmployeeBean` です。

Oracle 配置マップ・ファイルの構造は、次のとおりです。

1. XML のバージョンと DTD ファイルを表すヘッダー
2. JNDI 名にマップされる EJB 論理名

この例では、<ejb-name> 内で定義されている論理名 `Employee` は、<jndi-name> 要素内の `/test/EmployeeBean` にマップされます。

3. この Bean を実行するアイデンティティ
 - a. <run-as> 要素では、<mode> 要素内で、Bean を実行するアイデンティティを定義しています。可能な値は、次のとおりです。
 - CLIENT_IDENTITY — Bean はクライアントのユーザーとして実行されます。
 - SPECIFIED_IDENTITY — Bean はここで、指定したアイデンティティとして実行されます。このアイデンティティは、<security-role> 要素内で定義されます。詳細は、[付録 A「XML デプロイメント・ディスクリプタ」](#)を参照してください。
 - SYSTEM_IDENTITY — Bean はサーバーに対する DBA または ROOT で実行されます。Oracle8i Server の場合、Bean は SYS で実行されます。
 - b. <run-as> 要素の下位の <method> 要素では、<mode> アイデンティティに基づいて実行されるメソッドが定義されます。

この例では、`EmployeeBean` 内のメソッドがすべてクライアントのユーザーとして実行されます。

例 2-2 Employee 用の Oracle 配置マップ・ファイル

```
<?xml version="1.0"?>
<!DOCTYPE oracle-descriptor PUBLIC "-//Oracle Corporation.//DTD Oracle 1.1//EN"
"oracle-ejb-jar.dtd">
<oracle-descriptor>
  <mappings>
    <ejb-mapping>
      <ejb-name>Employee</ejb-name>
      <jndi-name>/test/EmployeeBean</jndi-name>
    </ejb-mapping>
  </mappings>
  <run-as>
    <description>no description</description>
    <mode>CLIENT_IDENTITY</mode>
    <method>
      <ejb-name>EmployeeBean</ejb-name>
      <method-name>*</method-name>
    </method>
  </run-as>
</oracle-descriptor>
```

JAR ファイルの作成

deployejb コマンドライン・ツールで、Bean にアクセスする際にクライアント側で使用する JAR ファイルを作成します。

```
deployejb -user scott -password tiger -service sess_iiop://dbserver:2481:orcl \
-descriptor employee.xml -oracledescriptor oracle_employee.xml \
-temp /tmp/ejb -generated empClient.jar employee.jar
```

deployejb ツールで実行されるタスクの 1 つは、必要なスタブとスケルトンを含む JAR ファイルを作成することです。このファイルは、クライアントにより実行時に使用されます。XML ディプロイメント・ディスクリプタ内の <ejb-client-jar> 要素で JAR ファイル名を指定しなければ、この JAR ファイルのデフォルト名は server_generated.jar となります。この JAR ファイルは、クライアントで実行できるように CLASSPATH に挿入する必要があります。

注意： アプリケーションで Bean 間にメソッドの循環参照を使用する場合は、まず loadjava を使用して参照される Bean をロードしてから、JAR ファイルに対して deployejb を起動する必要があります。メソッドの循環参照とは、Bean A のメソッドが Bean B のメソッドを参照し、Bean B のメソッドが Bean A のメソッドを参照する場合のことです。両者が異なるメソッドであっても、データベースにロード済みの参照先 Bean が見つからなければ、ロードの解決が失敗します。loadjava を使用して Bean B をロードすると、ロード時に正常に解決されます。

ホーム・インタフェースの公開

Bean 提供者は、クライアントが Bean を見付け出して活性化できるようにするため、Bean のホーム・インタフェースを JNDI lookup から利用可能にする必要があります。ただし、`deployejb` コマンドライン・ツールを使用して JAR ファイルを作成すると、このツールにより、JNDI ネームスペースにある Bean は Oracle 配置マッピング・ファイル内で指定した `<jndi-name>` 要素名で公開されます。

EJB の削除

データベースから EJB を削除するには、次のようにします。

- `dropjava` ツールを実行して、データベースからクラスを削除します。Bean のクラス・ファイルが収められている元の Bean JAR ファイルを指定します。
- セッション・シェル・ツールを使用して、オブジェクト名前空間から Bean のホーム・インタフェース名を削除します。

`dropjava` およびセッション・シェル・ツールの詳細は、『Oracle8i Java Tools リファレンス』を参照してください。

例の実行

この例を実行するには、クライアント側の JVM を使用してクライアント・クラスを実行します。この例では、`java` コマンドに次のものが組み込まれるよう、`CLASSPATH` を設定する必要があります。

- 標準 Java ライブラリ・アーカイブ (`classes.zip`)
- Java の `vbjapp.jar` および `vbjorb.jar` の場合は VisiBroker のものなど、クライアントの ORB で使用するクラス・ファイル
- Oracle8i に付属の JAR ファイル、`mts.jar` および `aurora_client.jar`

JDBC を使用している場合は、次の JAR ファイルのどちらか一方を含めます。

- JDBC 1.1 サポート用の `classes111.zip`
- JDBC 1.2 サポート用の `classes12.zip`

SSL を使用している場合は、次の JAR ファイルのどちらか一方を含めます。

- SSL 1.1 サポート用の `javax-ssl-1_1.jar` および `jssl-1_1.jar`
- SSL 1.2 サポート用の `javax-ssl-1_2.jar` および `jssl-1_2.jar`

これらのライブラリは、インストールした Oracle ホーム・ディレクトリの下の `lib` および `jlib` ディレクトリにあります。

JDK の `java` コマンドを次のように起動すると、この例が実行されます。

注意： UNIX シェル変数 \$ORACLE_HOME は、Windows NT では %ORACLE_HOME% と表される場合があります。JDK_HOME は、Java Development Kit (JDK) のインストール先ディレクトリです。

```
% java -classpath .:$ (ORACLE_HOME) /lib/aurora_client.jar
:$(ORACLE_HOME)/lib/mts.jar
:$(ORACLE_HOME) /jdbc/lib/classes111.zip:
$(ORACLE_HOME) /lib/vbjorb.jar:
$(ORACLE_HOME) /lib/vbjapp.jar:$(JDK_HOME) /lib/classes.zip
Client
sess_iiop://localhost:2481:ORCL
/test/myEmployee
scott tiger
```

プログラミングの制限事項

EJB 1.1 仕様には、次のプログラミング上の制限事項があります。EJB クラスのメソッドを実装するときにはこれらの事項に従う必要があります。

- EJB は、新しいスレッドの開始も、実行中のスレッドの終了もしないようする必要があります。カレント・リリースでは、EJB が新しいスレッドを開始した場合に例外は発生しませんが、ORB 内のローカル・スレッド・オブジェクトとの相互作用により、アプリケーションの動作は予測できないものになります。
- EJB では、スレッド同期の基本機能は使用できません。
- EJB でトランザクションのデマークेटに使用できるのは、`javax.Transaction.UserTransaction` インタフェースのみです。
- EJB では、`java.security.Identity` の変更はできません。変更しようすると、`java.security.SecurityException` が発生します。
- EJB では、JDBC のコミット・メソッドおよびロールバック・メソッドを使用することも、SQLJ や JDBC を使用して直接 SQL コミット・コマンドまたはロールバック・コマンドを発行することもできません。

EJB1.1 仕様では、「EJB では、読取りおよび書込み両用の `static` フィールドを使用できません。読取り専用の `static` フィールドのみが使用できます。したがって、すべての `static` フィールドは `final` と宣言する必要があります。」と規定されています。これは、Oracle8i 特有の制限事項ではありません。

デバッグ手法

Java IDE および JVM がリモート・デバッグをサポートするようになるまで、CORBA のクライアントおよびサーバー・コードのデバッグに、いくつかの代替手法を使用できます。

1. Java アプリケーションのデバッグには JDeveloper を使用します。JDeveloper には、Oracle8i JVM のデバッグ機能を使用するユーザー・インタフェースが用意されています。JDeveloper のデバッガを使用すると、データベースにロードされたオブジェクトを正常にデバッグできます。手順については、JDeveloper のドキュメントを参照してください。
2. サーバー上で実行されるオブジェクトのデバッグには、事前に公開された DebugAgent オブジェクトを使用します。詳細は、2-29 ページの「[デバッグ・エージェントを使用したサーバー・アプリケーションのデバッグ](#)」を参照してください。
3. 単一のマシン上で ORB トレース機能を使用して、スタンドアロン ORB デバッグを実行します。

クライアントとサーバーの両方を単一プロセスの単一アドレス空間に配置してデバッグします。クライアントまたはサーバーのデバッグに IDE を使用するかどうかは任意ですが、使用することをお勧めします。

4. Oracle8i トレース・ファイルを使用します。

クライアントでは、`System.out.println()` の出力が画面に出力されます。ただし、Oracle8i ORB では、すべてのメッセージはサーバーのトレース・ファイルに出力されます。トレース・ファイルのディレクトリは、データベース初期化ファイル内でパラメータとして指定されています。製品がデフォルトで `$ORACLE_BASE` という名前のディレクトリにインストールされている場合、トレース・ファイルは次のように示されます。

```
${ORACLE_BASE}/admin/<SID>/bdump/ORCL_s000x_xxx.trc
```

ただし、ORCL は SID であり、x_xxx はプロセス ID 番号を表します。Oracle インスタンスを起動した後にトレース・ファイルを削除しないでください。削除すると、トレース・ファイルには何も出力されません。トレース・ファイルを削除する場合には、いったん停止し、それからサーバーを再起動してください。

5. 単一の Oracle MTS サーバーの使用。

デバッグの場合に限り、`INITSID.ORA` ファイル内の `MTS_SERVERS` パラメータを `MTS_SERVERS = 1` に設定し、`MTS_MAX_SERVERS` を 1 に設定してください。複数の MTS サーバーをアクティブにしておくと、サーバー・プロセスごとにトレース・ファイルが開かれるので、複数のセッションでオブジェクトが活性化されると複数のトレース・ファイルにメッセージが分散します。

6. `printback` の例を使用して `System.out` をリダイレクトします。この例は、デモ・ディレクトリ `demo/examples/corba/basic/printback` にあります。

デバッグ・エージェントを使用したサーバー・アプリケーションのデバッグ

デバッグ環境の設定手順の詳細は、『Oracle8i Java 開発者ガイド』を参照してください。ただし、DBMS_JAVA プロシージャを使用してデバッグ・エージェントを起動する方法が記載されています。CORBA アプリケーションでは、デバッグ・エージェントの起動、停止および再起動に `oracle.aurora.debug.DebugAgent` クラスのメソッドを使用できます。これらのメソッドの機能は、DBMS_JAVA の同等のメソッドとまったく同じです。

```
public void start( java.lang.String host, int port, long timeout_seconds)
                throws DebugAgentError
public void stop() throws DebugAgentError
public void restart(long timeout) throws DebugAgentError
```

例 2-3 サーバー上での DebugAgent の起動

次の例は、サーバー上に存在するオブジェクトのデバッグ方法を示しています。最初に、`debugproxy` コマンドライン・ツールによってデバッグ・プロキシを起動する必要があります。この例では、`debugproxy` に対して、デバッグ・エージェントによる接続時に `jdb` デバッガを起動するように通知します。

このコマンドの実行後に、デバッグ対象のオブジェクトを検索するクライアントを起動し、「`/etc/debugagent`」として事前に公開されている `DebugAgent` を検索し、`DebugAgent` を起動します。

`DebugAgent` の起動後は、`debugproxy` により `jdb` デバッガが起動し、ブレーク・ポイントを設定できます。`DebugAgent` がタイムアウトになるまでの期間を指定しているため、最初にすべてのスレッドを一時停止する必要があります。次に、すべてのブレーク・ポイントを設定してから再開します。これにより、実行する準備ができるまでタイムアウトが一時停止されます。

tstHost 上のプロキシ・ウィンドウ

```
% debugproxy -port 2286 start jdb -password
. (wait until a debug agent starts up and
.  contact this proxy... when it does, jdb
.  starts up automatically and you can set
.  breakpoints and debug the object, as follows:)
> suspend
> load SCOTT:Bank
> stop in Bank:updateAccount
> resume
> ...
```

クライアント・コード

```
main( ... )
{
    //retrieve the object that you want to debug
    Bank b = (Bank)ic.lookup(sessURL + "/test/Bank");
    //lookup debugagent from JNDI
    DebugAgent dbagt = (DebugAgent)ic.lookup(svcURL + "/etc/debugagent");
    //start the debug agent and give the proxy host, port, and a timeout
    dbagt.start("tstHost",2286,30);
    ...
    //execute methods within Bank)
    ...
    //stop the agent when you want to
    dbagt.stop();
    //restart the agent when you want to
    dbagt.restart(30);
}
```

IIOP アプリケーションの構成

IIOP ベース・アプリケーションを構成するには、それが EJB アプリケーションか CORBA アプリケーションかに関係なく、セッション・ベースの IIOP 通信用に適切なリスナーと MTS サーバーを構成する必要があります。IIOP ベース・アプリケーションの構成処理には、データベースとネットワークの構成が含まれます。この章では、これらの要素について説明します。

- 概要
- Oracle8i 標準インストールまたは最小インストール
- Oracle8i カスタム・インストール
- 手動インストールおよび構成
- 高度な構成に関するオプション

概要

クライアントは、Internet Inter-ORB Protocol (IIOP) 接続を介して、データベース内の EJB アプリケーションおよび CORBA アプリケーションにアクセスします。IIOP は、TCP/IP を介した General Inter-ORB Protocol (GIOP) の実装です。データベースと通信する CORBA クライアントまたは EJB クライアントとの IIOP 接続では、次の場合を除き、必ずデータベース上および Net8 リスナー内に IIOP を構成する必要があります。

- 接続するリスナーがデータベースと同じノード上にある場合
- 使用するデータベースが汎用ディスパッチャとして構成されている場合
- 3-15 ページの「[動的なリスナー・エンドポイント登録](#)」で説明する動的登録ツールを使用して、リスナーで IIOP 要求を管理できるようにする場合

それ以外の場合は、データベースとリスナーを次のように構成する必要があります。

エンティティ	説明	構成ツール
データベース	IIOP 接続をサポートするには、GIOP 用のデータベースを MTS モードで構成する必要があります。	Database Configuration Assistant を使用して、IIOP 用にデータベースの MTS デスパッチャを構成します。このツールは、Oracle8i を通常インストールまたはカスタム・インストールすると、同時に起動されます。
Net8 リスナー	IIOP 接続をサポートするには、定義済のポート 2481 または 2482 で IIOP 接続を受信するように Net8 リスナーを構成する必要があります。	Net8 Assistant を使用して、IIOP 用に Net8 リスナーを構成します。

データベースでは、プレゼンテーションを介して、受信したリクエストがサポートされます。プレゼンテーション・プロトコルには、アプリケーション層およびセッション層が対応できる形式でデータが表現されていることを確認する役割があります。リスナーとディスパッチャの両方で、構成されるプレゼンテーションに基づいて、受信ネットワーク・リクエストが受け付けられます。IIOP の場合は、GIOP プレゼンテーションが構成されます。

注意： セキュリティ上の理由から、IIOP 接続でセキュア・ソケット・レイヤー (SSL) を使用するかどうかを決定する必要があります。

- SSL の情報は、6-3 ページの「[セキュア・ソケット・レイヤー \(SSL\) の使用](#)」を参照してください。
- SSL の構成方法の詳細は、3-17 ページの「[EJB および CORBA 用の SSL の構成](#)」を参照してください。

IIOP 接続用の構成は、次の 3 つの方法のいずれかで実行できます。

- **Oracle8i 標準インストールまたは最小インストール** — Oracle8i を標準インストールまたは最小インストールすると、データベースおよびリスナーの両方に対して、セッション・ベースの非 SSL IIOP 接続が構成されます。
- **Oracle8i カスタム・インストール** — Oracle8i のカスタム・インストール時に Oracle8i JVM オプションを選択すると、データベースに対してセッション・ベースの非 SSL IIOP 接続が構成されます。リスナーに対して IIOP 接続を構成するには、Net8 Assistant を起動する必要があります。
- **手動インストールおよび構成** — `initjvm.sql` スクリプトを起動して Oracle8i JVM をインストールする場合、IIOP 接続を手動で構成する必要があります。どの構成も、Database Configuration Assistant および Net8 Assistant を直接起動するか、または様々な初期化パラメータ・ファイルを編集して手動で実行できます。

Oracle8i 標準インストールまたは最小インストール

サーバーの標準インストール時に、Oracle8i JVM がインストールされ、構成されます。非 SSL TCP/IP を使用したリスナーを介して、セッション・ベースの IIOP 接続を行う MTS データベースの構成が自動的に受信されます。

標準インストールが完了すると、データベース初期化ファイルに次の行が追加されます。

```
mts_dispatchers="(protocol=tcp) (presentation=oracle.aurora.server.SGiopServer) "
```

Advanced Security Option をインストールしており、SSL ベースの TCP/IP 接続を構成する場合は、データベース初期化ファイルを編集して、次の行のハッシュ記号 (#) を削除します。

```
mts_dispatchers="(protocol=tcps) (presentation=oracle.aurora.server.SGiopServer) "
```

注意： (protocol=tcps) 属性は、接続が SSL を使用できると識別します。

さらに、IIOP 用にリスナーが構成されます。次のコードが、`listener.ora` ファイルに追加されます。

```
listener=
  (description_list =
    (description=
      (address=(...))
      (protocol_stack=
```

```
        (presentation=GIOP)
        (session=RAW)
    )
)
)
```

接続の構成後、ホスト名、ポート番号等が組み込まれた URL を使用してクライアントからリクエストが送信されます。URL では、リスナーを特定する情報に加えて、SID またはデータベース・サービス名が指定されます。URL の構文は次のとおりです。

```
session_iiop://<hostname>/:<portnumber>/:<SID | service_name>
```

Oracle8i カスタム・インストール

Database Configuration Assistant でカスタム・インストールを実行するときに Oracle8i JVM オプションを選択すると（[図 3-1](#) を参照してください）、非 SSL TCP/IP を使用してセッション・ベースの IIOP 接続を行う MTS データベースが構成されます。

注意： カスタム・インストールで標準インストールまたは最小インストール時と同じオプションを選択した場合は、3-3 ページの「[Oracle8i 標準インストールまたは最小インストール](#)」に定義されているものと同じ構成になります。

図 3-1 Oracle8i JVM オプションの選択



この画面のように選択すると、データベース初期化ファイルに次の行が追加されます。

```
mts_dispatchers=" (protocol=tcp) (presentation=oracle.aurora.server.SGiopServer) "
```

Advanced Security Option をインストールしており、SSL ベースの TCP/IP 接続を構成する場合は、データベース初期化ファイルを編集して、次の行のハッシュ記号 (#) を削除します。

```
mts_dispatchers=" (protocol=tcps) (presentation=oracle.aurora.server.SGiopServer) "
```

注意： (protocol=tcps) 属性は、接続が SSL を使用できると識別します。

インストールの完了後、Net8 Assistant を起動して、IIOP 接続用のリスナーを構成する必要があります。

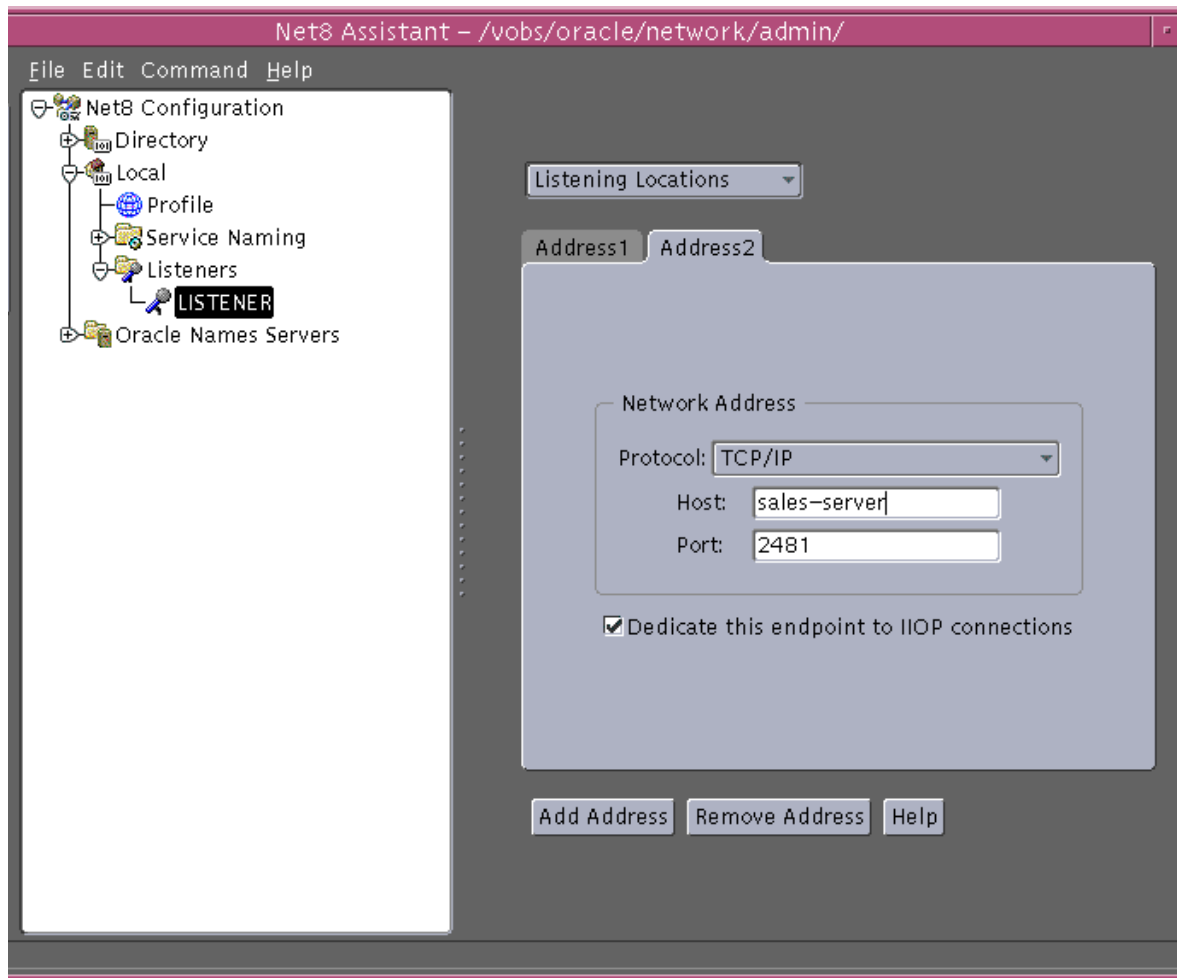
Net8 Assistant

Net8 Assistant では、リスナーの設定を変更できます。Net8 Assistant を使用してリスナーを構成する手順を、簡単に説明します。詳細な説明は、『Oracle8i Net8 管理者ガイド』を参照してください。

1. Net8 Assistant を起動します。
 - UNIX では、`$ORACLE_HOME/bin` で `netasst` を実行します。
 - Windows NT では、「スタート」>「プログラム」>「Oracle - *HOME_NAME*」>「Network Administration」>「Net8 Assistant」を選択します。
2. ナビゲータ・シートで、「Local」>「Listeners」を開きます。

リスナー・ロケーションのパネルが表示されます（[図 3-2](#) を参照してください）。

図 3-2 IIOP リスニング・ポートの構成



3. 「LISTENER」を選択します。
4. 右ペインのリストで、「Listening Locations」を選択します。
5. 「Protocol」プルダウン・リストから「TCP/IP」または「TCP/IP with SSL」を選択します。
6. 「Host」フィールドにデータベースのホスト名を入力します。

7. プロトコルに「TCP/IP」を選択した場合は、「Port」フィールドに 2481 と入力します。「TCP/IP with SSL」を選択した場合は、2482 と入力します。
8. 「Dedicate this endpoint to IIOP connections」チェック・ボックスを選択します。
9. 「File」>「Save Network Configuration」を選択します。

次のコードが、`listener.ora` ファイルに追加されます。

```
listener=
  (description_list =
    (description=
      (address=(protocol=tcp) (host=sales-server) (port=2481)))
      (protocol_stack=
        (presentation=GIOP)
        (session=RAW)
      )
    )
  )
```

接続の構成後、ホスト名、ポート番号等が組み込まれた URL を使用してクライアントからリクエストが送信されます。URL では、リスナーを特定する情報に加えて、SID またはデータベース・サービス名が指定されます。URL の構文は次のとおりです。

```
session_iiop://<hostname>/:<portnumber>/:<SID | service_name>
```

手動インストールおよび構成

通常インストールまたはカスタム・インストール時のオプションを使用して Oracle8i JVM をインストールしたのであれば、`initjvm.sql` スクリプトを使用して、Oracle8i JVM を既存のデータベースに追加できます。このスクリプトの詳細は、『Oracle8i Java 開発者ガイド』を参照してください。

Oracle8i JVM をインストールすると、Database Configuration Assistant または Net8 Assistant を使用するか、初期化ファイルを手動で編集して、IIOP 接続を構成できます。

ツールを使用して構成する場合

1. Database Configuration Assistant を使用して IIOP 用のデータベースを構成します。Database Configuration Assistant を起動するには、次の操作を実行します。
 - UNIX では、`$ORACLE_HOME/bin` で `dbassist` を実行します。
 - Windows NT では、「スタート」>「プログラム」>「Oracle - HOME_NAME」>「Database Administration」>「Database Configuration Assistant」を選択します。

Database Configuration Assistant の起動後、Oracle8i JVM オプションを選択します。この処理による初期化ファイルへの影響は、3-4 ページの「[Oracle8i カスタム・インストール](#)」を参照してください。

2. Net8 Assistant を使用して IIOP 用のリスナーを構成します。このステップは、3-6 ページの「[Net8 Assistant](#)」で説明しています。

初期化ファイルを編集して構成する場合

データベースのプレゼンテーション層では、クライアントからデータベースにアクセスする際に使用する接続のタイプが識別されます。GIOP プレゼンテーションを識別するには、セッション・ベースの IIOP 接続用の構成である `oracle.aurora.server.SGiopServer` を使用します。EJB アプリケーションおよび CORBA アプリケーションは、複数のセッション内でオブジェクトを活性化できます。そのクライアントが開始した単一のセッション内のオブジェクトに限定されません。これらの接続は、セッション IIOP、標準 IIOP 両方のセマンティクスで識別されます。

IIOP 接続を構成するには、次の初期化ファイルで GIOP プレゼンテーションを指定します。

1. データベース初期化ファイルで IIOP 接続を構成します。MTS_DISPATCHERS パラメータの PRESENTATION 属性を設定します。

この項では、MTS_DISPATCHERS パラメータの PRESENTATION 属性のみを説明します。MTS 構成の詳細は、『Oracle8i Net8 管理者ガイド』を参照してください。

2. IIOP 接続用の Net8 リスナーを構成します。

次に、この 2 つのステップを詳しく説明します。

1. データベース初期化ファイルを介した IIOP 接続の構成

データベース内で IIOP 接続を構成するのに、データベース初期化ファイルを手動で編集できます。

MTS_DISPATCHERS パラメータの構文は次のとおりです。

```
mts_dispatchers="(protocol=tcp | tcps)
                 (presentation=oracle.aurora.server.SGiopServer) "
```

MTS_DISPATCHER の属性は次のとおりです。

属性	説明
PROTOCOL (PRO または PROT)	TCP/IP または SSL を使用する TCP/IP を指定します。 このプロトコルに対してディスパッチャにより、リスニング・エンドポイントが生成されます。 有効な値: TCP (TCP/IP の場合) または TCPS (SSL を使用する TCP/IP の場合)
PRESENTATION (PRE または PRES)	GIOP のサポートを使用可能にします。GIOP プレゼンテーションには次の値を指定します。 <ul style="list-style-type: none">■ セッション・ベースの GIOP 接続の場合、<code>oracle.aurora.server.SGiopServer</code> です。このプレゼンテーションは、TCP/IP および SSL を使用する TCP/IP で有効です。

注意： データベース初期化ファイル内に複数の MTS_DISPATCHERS を構成する場合、各 MTS 定義をファイル中のまとまった位置に置く必要があります。複数の MTS_DISPATCHER 定義の間に別の構成パラメータを定義しないでください。

たとえば、非 SSL TCP/IP を使用し、リスナーを介してセッション・ベースの IIOP 接続を行う MTS を構成するには、データベース初期化ファイルに次の行を追加します。

```
mts_dispatchers="(protocol=tcp) (presentation=oracle.aurora.server.SGiopServer) "
```

2. 着信接続要求用のリスナーの構成

各リスナーは予約済みのポート番号上でリスニングするように構成され、クライアントはこのポート番号を使用してリスナーと通信します。CORBA および EJB をサポートするには、ポート 2481 または 2482 のいずれかで IIOP クライアントをリスニングするようにリスナーを構成する必要があります。

リスナーは、Net8 Assistant を使用して構成するか、または listener.ora ファイルで手動で構成します。Net8 Assistant を使用する方法をお勧めします。Net8 Assistant の詳細は、3-6 ページの「[Net8 Assistant](#)」を参照してください。

リスナーを手動で構成するには、listener.ora ファイルのリスナーの DESCRIPTION パラメータを変更する必要があります。

LISTENER.ORA DESCRIPTION パラメータの変更 GIOP リスニング・アドレスを使用するリスナーを構成する必要があります。次の例では、ポート番号 2481 を使用する非 SSL TCP/IP 用の GIOP プレゼンテーションの構成を示します。SSL を使用しない場合はポート 2481 を使用し、SSL を使用する場合はポート 2482 を使用します。

GIOP の場合、sales-server に対する IIOP 接続を構成するときに、PROTOCOL_STACK パラメータが DESCRIPTION に追加されます。

```
listener=
  (description_list=
    (description=
      (address=(protocol=tcp) (host=sales-server) (port=2481))
      (protocol_stack=
        (presentation=giop)
        (session=raw))))
```

次の表は、各 GIOP パラメータの定義を示します。

属性	説明
PROTOCOL_STACK	接続に関するプレゼンテーション層およびセッション層の情報を識別します。
(PRESENTATION=GIOP)	IIOP クライアント用の GIOP プレゼンテーションを識別します。GIOP は TCP/IP を使用して oracle.aurora.server.SGiopServer をサポートします。
(SESSION=RAW)	セッション層を識別します。IIOP クライアントの場合、特定のセッション層はありません。
(ADDRESS=...)	ポート 2481（SSL を使用しない場合）またはポート 2482（SSL を使用する場合）のいずれかに、TCP/IP を使用するリスニング・アドレスを指定します。SSL を使用しない場合はプロトコルに TCP を、SSL を使用する場合は TCPS を定義します。

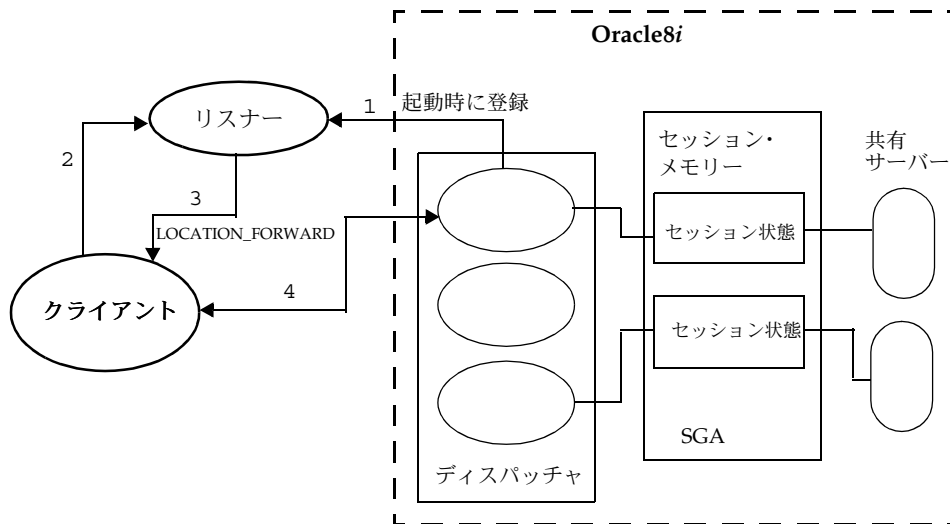
高度な構成に関するオプション

- データベースのリスナーとディスパッチャ
- 動的なリスナー・エンドポイント登録
- ディスパッチャへの直接接続
- EJB および CORBA 用の SSL の構成

データベースのリスナーとディスパッチャ

図 3-3 は、リスナーとディスパッチャの対話の仕組みと、Oracle8i ORB セッションが起動される仕組みを示しています。

図 3-3 リスナーとディスパッチャの対話



1. データベースの起動時に、ディスパッチャはそれ自体をリスナーに登録します。
2. クライアントが、宛先としてリスナーの URL アドレスを指定してメソッドを起動します。
3. リスナーが、ディスパッチャのアドレスを通知する LOCATION_FORWARD 応答を、クライアントの ORB レイヤーに送信します。これにより、要求が適切なディスパッチャにリダイレクトされます。

注意： クライアントは、クライアントのために行なわれる、ORB ランタイム・レイヤーで実行されるリダイレクションのロジックを認識しません。

4. 基礎となる ORB ランタイム・レイヤーは、初期要求をディスパッチャに再送します。それ以後のすべてのメソッドの起動は、ディスパッチャに向けられます。リスナーは通信に関与しなくなります。

受信した要求が共有サーバー・サービスにより検査され、既存セッションに関するものかどうかを確認されます。既存セッションに関する要求の場合は、指定されたセッションに転送されます。そうでない場合は、サービスにより要求のための新規データベース・セッションが作成され、セッション中に ORB が活性化されます。このセッションは、Net8 着信接続要求に対して作成されたデータベース・セッションにきわめて類似しています。このセッションで ORB では、受信 IIOP メッセージの読込み、クライアント認証、対応するサーバー側オブジェクトの検索と活性化処理が実行され、接続したクライアントにリプライするために必要に応じて IIOP メッセージが送信されます。クライアントからの後続のメッセージは、既存のセッションに向けられます。

リスナーの構成時には、Net8 接続と IIOP 接続のリスニング・エンドポイントとして個別のポートを構成する必要があります。同様に、エンドポイントにセキュア・ソケット・レイヤー (SSL) を使用する場合は、SSL を使用できる IIOP エンドポイント用の別個のエンドポイントも必要です。IIOP と SSL を使用する接続の詳細は、6-3 ページの「[セキュア・ソケット・レイヤー \(SSL\) の使用](#)」を参照してください。

受信した要求の処理

データベース管理者は、GIOP を使用できるディスパッチャを使用して MTS サーバーを構成します。また、管理者はこのディスパッチャがデータベース起動時に登録するリスナーを構成します。

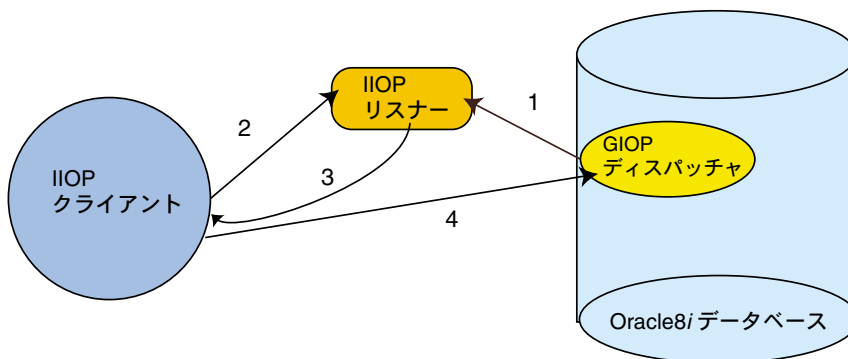
データベースの起動時に、すべてのディスパッチャは同じデータベース初期化ファイル内で構成されているリスナーすべてに登録します。ただし、IIOP クライアントが要求を起動すると、リスナーは GIOP ディスパッチャへの要求のリダイレクトまたは汎用ディスパッチャへの接続の受渡しのみを行います。

この動作の詳細は、次の項を参照してください。

- [GIOP ディスパッチャへのリダイレクト](#)
- [汎用ディスパッチャへの接続の受渡し](#)

GIOP ディスパッチャへのリダイレクト リスナーでは、IIOP プロトコルが認識され、要求が登録済みの GIOP ディスパッチャにリダイレクトされます。

図 3-4 IIOP リスナーから GIOP ディスパッチャへのリダイレクト



1. GIOP ディスパッチャが、それ自体をリスナーに登録します。
2. IIOP クライアント、つまり EJB または CORBA クライアントが、リスナーのアドレスを指定してメソッドを起動します。リダイレクションを発生させるには、IIOP 要求を受信するようにリスナーを静的に構成する必要があります。
3. リスナーが、GIOP ディスパッチャのアドレスを通知する応答を、クライアントに送信します。

GIOP ディスパッチャが構成されている場合は、リスナーがリダイレクトします。GIOP ディスパッチャが構成されていない場合は、リスナーは要求を汎用ディスパッチャに渡すことができます。詳細は、3-14 ページの「[汎用ディスパッチャへの接続の受渡し](#)」を参照してください。

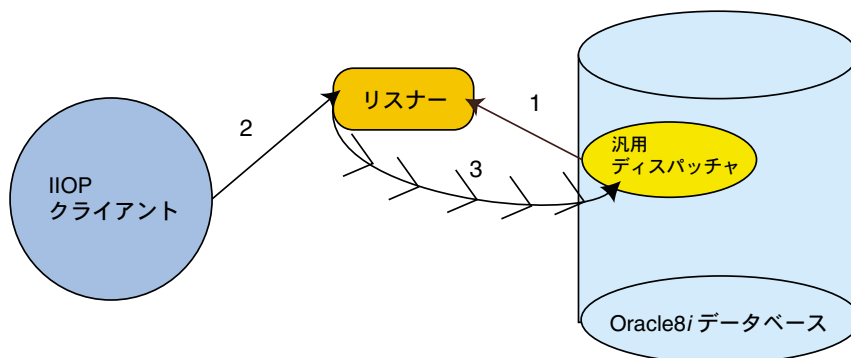
4. クライアント上の基礎となる ORB ランタイム・レイヤーが、初期要求を GIOP ディスパッチャに再送します。それ以後のすべてのメソッドの起動は、ディスパッチャに向けられます。リスナーは通信に関与しなくなります。

汎用ディスパッチャへの接続の受渡し 静的に構成された GIOP ディスパッチャがなく、汎用ディスパッチャが構成されている場合、リスナーは要求をこのディスパッチャに渡すことができます。制限事項は、接続の受渡しを発生させるにはリスナーとディスパッチャが同じノード上に存在する必要があることと、リスナーは IIOP 要求を受信するように静的または動的に構成する必要があることのみです。

接続の受渡しによって、リスナーが保持していたクライアントからのソケットはディスパッチャに移されます。受渡しは単一ノード内でのみ可能です。

図 3-5 は、受渡し環境におけるディスパッチャとリスナーの組合せを示しています。

図 3-5 ディスパッチャへの接続の受渡し



1. データベースの起動時に、汎用ディスパッチャがそれ自体を構成済みリスナーに登録します。

注意： リスナーは、IIOP 要求を受信するように構成する必要があります。この場合、リスナーは、Net8 構成を介して静的に構成する方法と、セッション・シェル (sess_sh) の動的登録コマンド `regep` を介して動的に構成する方法があります。詳細は、3-15 ページの「[動的なリスナー・エンドポイント登録](#)」を参照してください。

2. クライアントがリスナーに要求を送信します。
3. リスナーが要求を汎用ディスパッチャに渡します。リスナーは、別個のチャンネルで汎用ディスパッチャとネゴシエートします。このチャンネル上で、ソケットはオペレーティング・システムのメカニズムを介してディスパッチャに渡されます。

クライアントは、この時点からはディスパッチャと直接通信します。クライアントには、ソケットが渡されたことは認識されません。

動的なリスナー・エンドポイント登録

3-14 ページの「[汎用ディスパッチャへの接続の受渡し](#)」で説明したように、リスナーはソケットを既存の汎用ディスパッチャに渡します。IIOP 受信要求の受渡しを発生させるには、リスナーに登録済みの IIOP エンドポイントが必要です。リスニング・エンドポイントは、次のどちらかを介して登録できます。

- 静的構成 - Net8 構成ツールで構成します。
- 動的構成 - セッション・シェル (sess_sh) の動的登録コマンド `regep` で登録します。

セッション・シェル (sess_sh) の動的登録コマンド `regep` では、任意のタイプのリスニング・エンドポイントがリスナーに追加されます。これには、IIOP リスニング・エンドポイントが含まれます。IIOP リスニング・エンドポイントに動的登録ツールを使用する方法については、後述します。

この使用例の制限事項は、次のとおりです。

- リスナーと汎用ディスパッチャの両方が、常に同じノードに存在すること
- GIOP 構成済みディスパッチャが存在しないこと

注意： GIOP 構成済みのディスパッチャが存在すると、リスナーは構成済みディスパッチャに要求を渡すのではなくリダイレクトします。

リスナー・エンドポイントを動的に登録すると、このリスナーで IIOP を使用できるようにデータベースを再起動する必要がないという利点があります。リスニング・エンドポイントは、即時にアクティブになります。

`regep` ツールの詳細は、『Oracle8i Java Tools リファレンス』を参照してください。

例 3-1 ポート 2241 でのリスナーの動的登録

次の行では、エンドポイント・ポート番号 2241 で SUNDB ホスト上のリスナーを動的に登録しています。このツールは、SUNDB ホストにログオンします。

```
regep -pres oracle.aurora.server.SGiopServer -host sundb -port 2241
```

ディスパッチャへの直接接続

リスナーをバイパスし、クライアントからディスパッチャに直接接続するには、クライアントをディスパッチャのポート番号に接続します。ディスパッチャのポート番号を検出するには、次のいずれかの操作を実行します。

- ポート番号を含む ADDRESS パラメータを追加して、ディスパッチャ用にポート番号を構成します。
- `lsnrctl service` を起動して、ディスパッチャに割り当てられているポートを検出します。

ポート番号を構成する場合、構文は次のようになります。

```
mts_dispatchers="(address=(protocol=tcp | tcps)
                  (host=<server_host>) (port=<port>))
                  (presentation=oracle.aurora.server.SGiopServer)"
```

属性は次のとおりです。

属性	説明
ADDRESS (ADD または ADDR)	ディスパッチャがリスニングするネットワーク・アドレスを指定します。ネットワーク・アドレスには、TCP/IP (TCP) または SSL を使用する TCP/IP (TCPS) のいずれかのプロトコル、サーバーのホスト名および GIOP リスニング・ポートを組み込むことができます。GIOP リスニング・ポートには、使用されていない任意のポートを指定できます。
PRESENTATION (PRE または PRES)	GIOP のサポートを使用可能にします。GIOP プレゼンテーションには次の値を指定します。 <ul style="list-style-type: none"> ■ セッション・ベースの GIOP 接続の場合、 oracle.aurora.server.SGiopServer です。このプレゼンテーションは、TCP/IP および SSL を使用する TCP/IP で有効です。

クライアントは、次のように URL でポート番号を指定します。

```
session_iiop://<hostname>/:<portnumber>
```

URL では SID またはサービス名は除外されます。SID インスタンスまたはサービス名は送信済のリクエストなので、ディスパッチャでは必要ありません。

EJB および CORBA 用の SSL の構成

Oracle8i では、証明書や秘密鍵など、GIOP と組み合わせて SSL を使用するために必要な認証データのサポートされます。GIOP とともに SSL を使用できるようにトランスポートを構成するには、次の操作を実行します。

1. SSL を使用できるように MTS_DISPATCHERS パラメータを設定します。
2. リスナーとデータベースを構成するときに SSL Wallet を使用するように指定します。
3. リスナーが SSL を受け付けるように構成します。

次の項では、この 3 つのステップの実行方法を詳しく説明します。

SSL 用の MTS_DISPATCHERS の有効化

SSL を使用できるディスパッチャを追加するには、データベースの初期化ファイルを編集する必要があります。データベース初期化ファイルの、TCPS ポートを定義している MTS_DISPATCHERS パラメータをコメント解除します。インストール時に、Database Configuration Assistant は常に SSL TCP/IP 用の行をコメントにして組み込みます。この行は次のようになります。

```
mts_dispatchers="(protocol=tcps) (presentation=oracle.aurora.server.SGiopServer) "
```

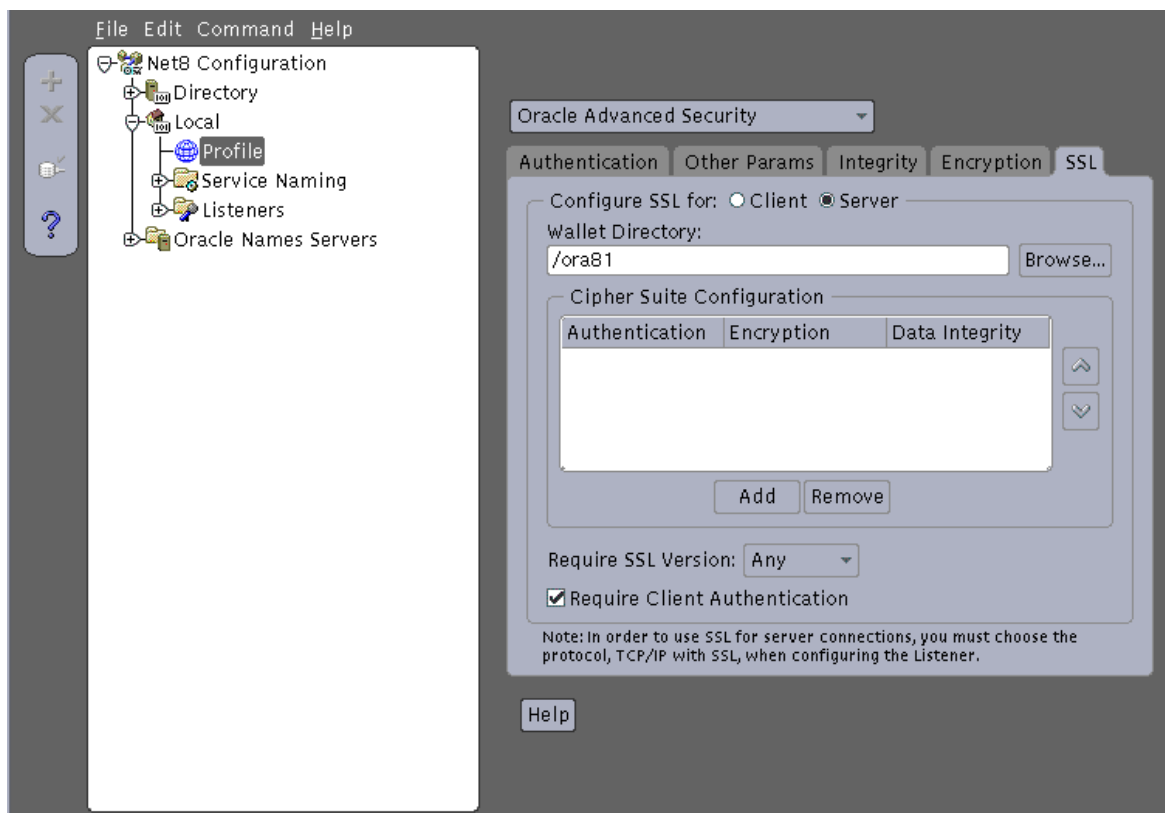
Net8 Assistant を介した Wallet の位置の設定

ポート 2482 で SSL リクエストを受信するようにリスナーを変更します。

1. Net8 Assistant を起動します。
 - UNIX では、`$ORACLE_HOME/bin` にある `netasst` を実行します。
 - Windows NT では、「スタート」>「プログラム」>「Oracle - *HOME_NAME*」>「Network Administration」>「Net8 Assistant」を選択します。
2. ナビゲータ・シートで、「Local」>「Profile」を開きます。
3. プルダウン・リストから、「Oracle Advanced Security」を選択し、「SSL」タブを選択します。

リスニング・ポートのパネルが表示されます（図 3-6 を参照してください）。

図 3-6 IIOP リスニング・ポートの構成



4. 「Configure SSL for」で、「Server」ラジオ・ボタンを選択します。
5. 「Wallet Directory」フィールドに、Wallet の位置を入力します。
6. 特定のバージョンの SSL を使用する場合は、SSL のバージョンのプルダウン・リストから適切なバージョンを選択します。
7. クライアントが証明書を提供して自分自身を認証させるには、「Require Client Authentication」チェック・ボックスを選択します。
8. 「File」>「Save Network Configuration」を選択します。

これらのステップを実行すると、Wallet と SSL の構成情報がリスナーとデータベースの両方の構成ファイルに追加されます。SSL Wallet の位置は、リスナーとデータベースの両方の構成ファイルで指定する必要があります。どちらのファイルでも、証明書のハンドシェイクを実行できるように Wallet の位置を指定する必要があります。

listener.ora ファイル:

```
ssl_client_authentication=false
ssl_version=undetermined
```

デフォルトでは、データベースがクライアントを認証するように設定されています。リスナーがクライアントを認証するように設定するには、ssl_client_authentication パラメータを TRUE に変更します。

データベースの sqlnet.ora ファイル:

```
ssl_client_authentication=true
ssl_version=0
sqlnet.crypto_seed=<seed_info>
```

クライアント認証をリクエストしなかった場合は、ssl_client_authentication パラメータが FALSE になります。デフォルトでは、クライアント認証は TRUE になっています。また、ssl_version に SSL の特定のバージョン番号 (3.0 など) を指定できます。ssl_version 値が 0 の場合、バージョンは未定で、ハンドシェイク時に決定されます。

リスナーの listener.ora ファイルとデータベースの sqlnet.ora ファイルの両方で、次のように Wallet の位置を指定します。

```
oss.source.my_wallet=
  (source=
    (method=file)
    (method_data=
      (directory=wallet_location)))
```

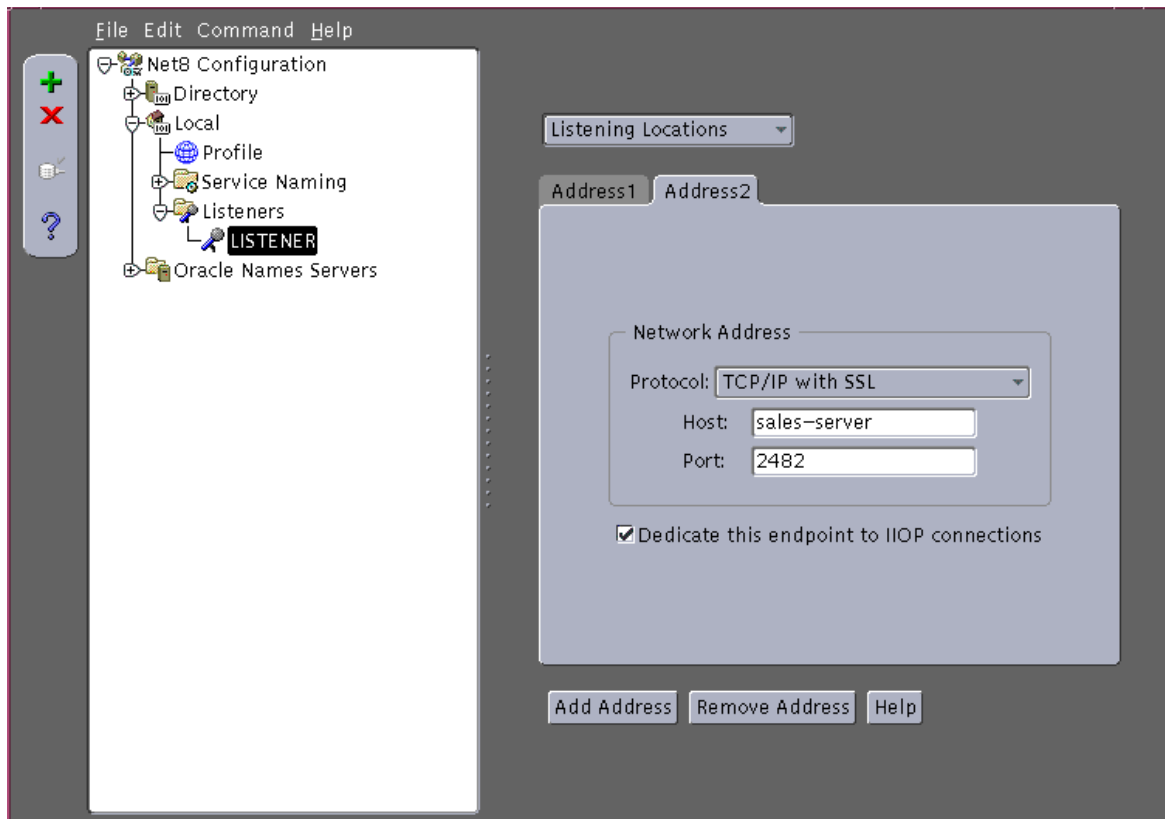
適切な証明書を持つ SSL Wallet の設定方法は、『Oracle8i Advanced Security 管理者ガイド』を参照してください。

Net8 Assistant を介して SSL を使用できるリスナーの構成

1. ナビゲータ・シートで、「Local」 > 「Listeners」を開きます。

リスナー・ロケーションのパネルが表示されます（図 3-7 を参照してください）。

図 3-7 IIOP リスニング・ポートの構成



2. 「LISTENER」を選択します。
3. 右ペインのリストで、「Listening Locations」を選択します。SSL リスニング・アドレス用に使用できるリスニング・アドレスがない場合は、「Add Address」ボタンをクリックするとアドレスを追加できます。
4. 「Protocol」プルダウン・リストから「TCP/IP with SSL」を選択します。
5. 「Host」フィールドにデータベースのホスト名を入力します。

6. 「Port」 フィールドに 2482 と入力します。
7. 「Dedicate this endpoint to IIOP connections」 チェック・ボックスを選択します。
8. 「File」 > 「Save Network Configuration」 を選択します。

次のコードが `listener.ora` ファイルに追加されます。これによりリスナーは、ポート番号 2482 を使用するプロトコルとして（TCP ではなく）TCPS を指定するように変更されます。次のコードは、`sales-server` ホスト上の、SSL を使用できるリスナーの例を示しています。

```
listener=
  (description_list=
    (description=
      (address= (protocol=tcps) (host=sales-server) (port=2482))))
    (protocol_stack=
      (presentation=giop)
      (session=raw)))
```

エンティティ Bean

この章では、エンティティ Bean の定義、作成方法およびセッション Bean との違いについて説明します。

- [エンティティ Bean の定義](#)
- [セッション Bean とエンティティ Bean の違い](#)
- [コールバック・メソッドの実装](#)
- [エンティティ Bean の作成](#)
- [Bean 管理の永続性とコンテナ管理の永続性の違い](#)
- [EJB の参照と JDBC の DataSources へのアクセス](#)

エンティティ Bean の定義

エンティティ Bean は、永続データを管理し、複雑なビジネス・ロジックを実行するリモート・オブジェクトで、依存関係のある Java オブジェクトを複数使用する場合もあり、主キーで一意に識別できます。通常、エンティティ Bean は、複数のファイン・グ레인（細かな）永続 Java オブジェクトに格納されている永続データを使用するという点で、大まかなロジックを実装した永続オブジェクトです。

注意： 通常、ファイン・グ레인永続 Java オブジェクトでは、データと表の列の間に 1 対 1 のマッピングを持つ永続データが管理されます。大まかなロジックを実装した永続 Java オブジェクトでは、複数のファイン・グ레인永続オブジェクトに格納されている永続データが使用または管理されます。

永続データの管理

エンティティ Bean では、そのデータの永続性は、`javax.ejb.EntityBean` インタフェース内で定義されているコールバック・メソッドを介して管理されます。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で指定される各コールバック関数を開発します。永続性には、Bean 管理の永続性とコンテナ管理の永続性の 2 種類があります。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。つまり、コンテナとエンティティ Bean の間には、コールバック・メソッドの起動順序と Bean の永続データをどちらが管理するかが指定されます。

主キーによる一意な識別

各エンティティ Bean には、永続的な識別情報が対応付けられています。つまり、エンティティ Bean には、主キーがあれば取得できる一意の識別情報が含まれています。主キーがあれば、クライアントはエンティティ Bean を取得できます。Bean 利用可能な状態でなければ、コンテナはその Bean をインスタンス化し、永続データを Bean に再度移入します。

一意キーの型は、Bean プロバイダにより定義されます。

依存オブジェクトを伴う複雑なロジックの実行

EJB アプリケーションの設計時には、各種オブジェクトの様々な側面に注意する必要があります。

- 通常、セッション Bean は、リモート・クライアントに対する単純なタスクの実行に使用します。
- 通常、エンティティ Bean は、リモート・クライアントに対して大まかなロジックを実装した永続性を伴う複雑なタスクを実行するために使用します。

- Java オブジェクトは、永続的かどうかに関係なく、ローカル・クライアントに対する単純なタスクに使用します。

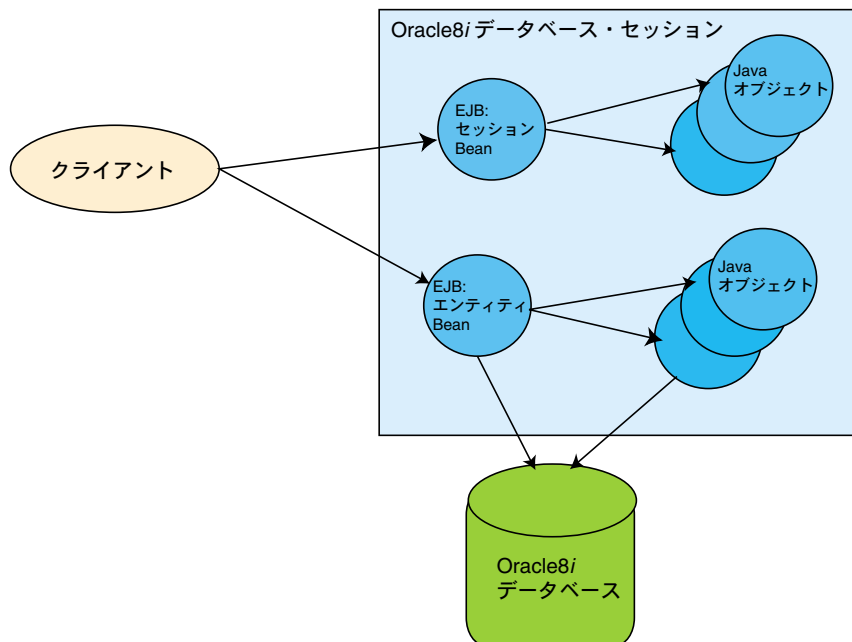
Enterprise JavaBeans はリモート・オブジェクトであり、ネットワークを介したクライアントとの相互作用に使用します。リモート・オブジェクトを使用すると、セキュリティおよびトランザクション情報の検証に関するオーバーヘッドが大きくなります。したがって、アプリケーションの設計時には、エンティティ Bean またはセッション Bean をクライアントと相互に作用させるのみでなく、Enterprise JavaBean で、依存関係のある他の Java オブジェクトを起動し、タスクの実行や永続データの管理を行ってもよいでしょう。

通常、エンティティ Bean は、リモート・クライアント用の複雑な大まかなロジックを実装した永続データの管理に使用します。エンティティ Bean と永続オブジェクトの違いを把握するように注意してください。エンティティ Bean は単なる永続オブジェクトではなく、複雑なデータを管理して戻し、データ管理にリモート・オブジェクトを使用する理由付けを行う必要があります。

アプリケーションでは、1 つ以上の依存オブジェクトをコールするエンティティ Bean を使用できます。エンティティ Bean はリモート・オブジェクトであるため、その主な機能はネットワークを介してクライアントと相互に作用することです。エンティティ Bean は、ネットワーク上の同一ノードにある別のエンティティ Bean を起動しないようにする必要があります。アプリケーション内で複数のオブジェクトを設計する必要がある場合は、エンティティ Bean によりクライアントと他の Java オブジェクト間の通信とデータ管理が容易になるように、アプリケーションを設計してください。

図 4-1 は、クライアントとセッション Bean またはエンティティ Bean との相互作用を示しています。セッション Bean またはエンティティ Bean により、クライアント用のアプリケーションはそのアプリケーション内の他の Java オブジェクトを使用して管理されます。アプリケーションのバックエンドを構成する Java オブジェクトは、永続オブジェクトでもかまいません。また、この図は、エンティティ Bean と永続 Java オブジェクトを永続的なものにして、データベース内でデータを格納する方法を示しています。

図 4-1 Enterprise JavaBeans と Java オブジェクトの関係



たとえば、オンライン書店のショッピング・カートを管理する場合は、次の要件があります。

- ショッピング・カートを使用する顧客を識別します。
- 顧客のショッピング・カートに品目を追加します。
- 値引、販売品目および出荷コストを考慮して、全品目の価格を計算します。
- 品目を顧客に出荷します。

この使用例の場合、エンティティ Bean では次のことができます。

- 依存永続オブジェクトから顧客情報を取得します。
- 品目の依存永続オブジェクトから品目情報を取得します。
- 各品目の品目数を注文書に記録します。
- 値引と出荷コストを取得します。
- 指定された顧客住所への出荷を調整します。

したがって、このエンティティ Bean では、他のオブジェクトから永続情報が取得されるのみでなく、独自の永続データもメンテナンスされ、複雑な計算が実行されます。

セッション Bean とエンティティ Bean の違い

セッション Bean とエンティティ Bean の重要な違いは、エンティティ Bean には、永続データ管理、永続的アイデンティティおよび複雑なビジネス・ロジックのためのフレームワークが必要であることです。エンティティ Bean のインタフェース要件では、永続データの管理が必要な場合や、Bean をそのアイデンティティに基づいて取得する必要がある場合に、コンテナがコールするコールバック関数が提供されます。

エンティティ Bean では、各コールバック・メソッドが適切なタイミングでコールされるように、インタフェースが設計されています。たとえば、トランザクションがコミットされる直前には、常に `ejbStore` メソッドが起動します。これにより、エンティティ Bean は、その永続データすべてをトランザクションの完了前に保存できます。これらの各コールバック・メソッドの詳細は、4-5 ページの「[コールバック・メソッドの実装](#)」を参照してください。

次の表は、セッション Bean とエンティティ Bean の各種インタフェースを示しています。Bean クラスと主キーでは、2 種類の EJB に違いがあることに注意してください。永続データの管理は、すべて Bean クラスのメソッド内で実行されます。

	エンティティ Bean	セッション Bean
リモート・インタフェース	<code>javax.ejb.EJBObject</code> を拡張 (extends) します。	<code>javax.ejb.EJBObject</code> を拡張 します。
ホーム・インタフェース	<code>javax.ejb.EJBHome</code> を拡張 します。	<code>javax.ejb.EJBHome</code> を拡張 します。
Bean クラス	<code>javax.ejb.EntityBean</code> を拡張 します。	<code>javax.ejb.SessionBean</code> を拡張 します。
主キー	特定の Bean インスタンスの識別と取得に使用します。	セッション Bean には使用しません。

コールバック・メソッドの実装

エンティティ Bean は、そのデータの永続性が、`javax.ejb.EntityBean` インタフェース内で定義されているコールバック・メソッドを介して管理されるリモート・オブジェクトです。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で指定される各コールバック関数を開発します。永続性には、Bean 管理の永続性とコンテナ管理の永続性の 2 種類があります。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コールし、Bean とその永続データを管理します。つまり、コンテナとエンティティ Bean 間では、コールバック・メソッドの起動順序と Bean の永続データをどちらが管理するかが指定されます。

Bean クラスでは、`EntityBean` インタフェースのメソッドが実装されます。`javax.ejb.EntityBean` インタフェースには、次の定義があります。

```
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 void unsetEntityContext();
}
```

コンテナは、これらのメソッドに次の機能があるものと想定します。

- **ejbCreate** ホーム・インタフェースで宣言されている create メソッドの 1 つに対応する 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、コンテナは最初に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てから、対応する ejbCreate メソッドを起動します。ejbCreate メソッドにより、次の操作が実行されます。
 - データベース行などのデータ用に永続な記憶域が作成されます。
 - 一意の主キーが初期化され、戻されます。
- **ejbPostCreate** コンテナは、環境が設定された後にこのメソッドを起動します。ejbCreate メソッドごとに、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、エンティティ・コンテキスト内またはエンティティ・コンテキストからのパラメータを初期化できます。
- **ejbRemove** コンテナで、セッション・オブジェクトの存続を終了する前に、このメソッドがコールされます。このメソッドでは、必要なクリーン・アップが実行されます。たとえば、ファイル・ハンドルなどの外部リソースがクローズされます。
- **ejbStore** コンテナは、トランザクションがコミットされる直前に、このメソッドを起動します。永続データは、データベースなどの外部リソースに保存されます。
- **ejbLoad** コンテナは、データをデータベースから初期化しなおす必要がある場合に、このメソッドをトランザクション内で起動します。通常、これはトランザクションの開始後に発生します。

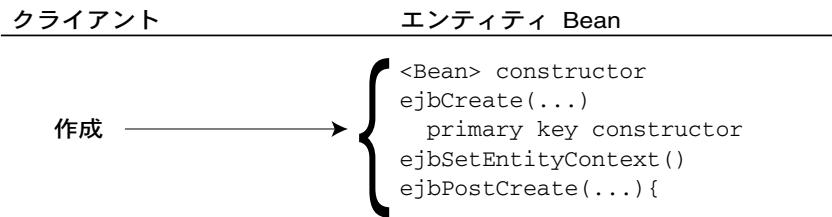
- `setEntityContext`
Bean インスタンスをコンテキスト情報に対応付けます。コンテナでは、Bean の作成後にこのメソッドがコールされます。エンタープライズ Bean では、トランザクション管理に使用するコンテキスト・オブジェクトへの参照がインスタンス変数に格納できます。自分のトランザクションを管理する Bean では、セッション・コンテキストを使用して、トランザクション・コンテキストを取得できます。

また、このメソッド内で Bean の存続期間中だけ存在するリソースを割り当てることもできます。これらのリソースは、`unsetEntityContext` で解放する必要があります。
- `unsetEntityContext`
対応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`
これは、カレント・リリースのサーバーではコールされないため、空のメソッドとして実装します。
- `ejbPassivate`
これは、カレント・リリースのサーバーではコールされないため、空のメソッドとして実装します。

ejbCreate および ejbPostCreate の使用

エンティティ Bean は、`ejbCreate` など、特定のコールバック・メソッドが指定のタイミングで起動されるという点では、セッション Bean に似ています。エンティティ Bean では、その永続データ、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されます。次の図は、エンティティ Bean の作成時にコールされるメソッドを示しています。

図 4-2 エンティティ Bean の作成



setEntityContext の使用

このメソッドは、エンティティ Bean インスタンスでそのコンテキストの参照を保存するために使用されます。エンティティ Bean には、コンテナによりメンテナンスされ、Bean に使用可能になるコンテキストがあります。Bean では、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、トランザクション・ロールなど、その Bean に関する情報を取得できます。Bean に関してコンテキストから取得できる情報の詳細は、Enterprise JavaBeans 1.1 の仕様書を参照してください。

コンテナは、最初に Bean をインスタンス化してから setEntityContext メソッドを起動し、Bean でコンテキストを取得できるようにします。コンテナが、このメソッドをトランザクション・コンテキスト内でコールすることはありません。Bean がこの時点でコンテキストを保存しなければ、それ以後はそのコンテキストにはアクセスできません。

注意： setEntityContext および unsetEntityContext メソッドを使用して、インスタンスの存続期間だけ存在するリソースを割り当てたり破棄することもできます。

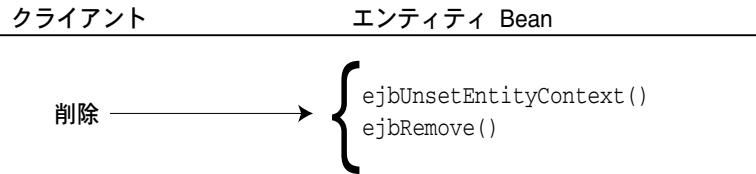
コンテナは、このメソッドをコールするときに、EntityContext オブジェクトの参照を Bean に渡します。Bean は、その参照を後で使用できるように格納できます。次の例は、コンテキストを `this.ctx` 変数に保存する Bean を示しています。

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

ejbRemove の使用

クライアントが remove メソッドを起動すると、コンテナが次のメソッドを起動します。

図 4-3 エンティティ Bean の削除



ejbStore および ejbLoad の使用

永続データを管理するために、ejbStore および ejbLoad メソッドもコールされます。この2つは、Bean 管理の永続 Bean に使用する最も重要なコールバック・メソッドです。

- ejbStore メソッドは、トランザクションの終了に近づくたびにコンテナによりコールされます。その目的は、永続データをデータベースなどの外部リソースに保存することです。
- ejbLoad メソッドは、トランザクションが開始されるたびに、またはエンティティ Bean がインスタンス化されるときに、コンテナによりコールされます。その目的は、この Bean のインスタンスについて存在する永続データをリストアすることです。

エンティティ Bean の作成

エンティティ Bean の作成手順は、セッション Bean の場合と同じです。ただし、Bean クラスのメソッドとデータには違いがあります。エンティティ Bean には、Bean 管理の永続 Bean とコンテナ管理の永続 Bean の2種類があります。この項では、Bean 管理の永続 Bean について説明します。コンテナ管理の永続 Bean の例については、4-27 ページの「[コンテナ管理の永続性](#)」を参照してください。

エンティティ Bean を作成するは、次のステップを実行します。

1. Bean 用のリモート・インタフェースを作成します。リモート・インタフェースでは、クライアントで起動できるメソッドを宣言します。このインタフェースでは、`javax.ejb.EJBObject` を拡張 (extends) する必要があります。
2. Bean 用のホーム・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。このインタフェースでは、Bean の `create` メソッドと `findByPrimaryKey` などのファインダ・メソッドを定義します。
3. Bean の主キーを定義します。主キーでは、各エンティティ Bean インスタンスが識別されます。主キーは、`java.lang.String` のような定式クラスを指定するか、または独自のクラス内で定義する必要があります。
4. Bean を実装します。これには、次の要素が含まれます。
 - a. リモート・インタフェースで宣言されたメソッドの実装。
 - b. Bean の空のコンストラクタ。
 - c. `javax.ejb.EntityBean` インタフェース内で定義されたメソッド。

- d. ホーム・インタフェース内で宣言されているメソッドと一致するメソッド。これには、次の要素が含まれます。
 - * ホーム・インタフェースで定義済みの create メソッドと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・インタフェース内で定義されている他のファインダ・メソッド。
5. 永続データがデータベースに保存されるか、データベースからリストアされる場合は、Bean 用に適切な表が存在することを確認する必要があります。
6. Bean ディプロイメント・ディスクリプタを作成します。このディプロイメント・ディスクリプタでは、XML のプロパティを介して Bean のプロパティを指定します。詳細は、2-19 ページの「[EJB の配置](#)」を参照してください。
7. Bean、リモート・インタフェースとホーム・インタフェースおよびディプロイメント・ディスクリプタを含む、ejb-jar ファイルを作成します。この ejb-jar ファイルでは、アプリケーション内の Bean をすべて定義する必要があります。詳細は、2-25 ページの「[JAR ファイルの作成](#)」を参照してください。

ホーム・インタフェース

セッション Bean と同様に、エンティティ Bean のホーム・インタフェースには、create メソッドを含める必要があります。クライアントは、このメソッドを起動して Bean インスタンスを作成します。各 create メソッドには、異なるシグネチャを使用できます。

エンティティ Bean 用に、findByPrimaryKey メソッドを開発する必要があります。永続データはインスタンスに対応付けられているため、各エンティティ Bean のインスタンスは主キーにより一意に識別されます。一意キーの型は、開発者が定義します。たとえば、顧客 Bean の主キーとして顧客番号を使用したり、発注の主キーとして発注番号を使用します。主キーには、一意の値であれば何でも使用できます。

エンティティ Bean を初めて作成すると、その Bean を識別する主キーが ejbCreate メソッドにより作成されます。Bean クラスの ejbCreate メソッド内で一意の主キーが作成され、初期化されます。これ以降、この Bean はこの主キーに対応付けられていることになります。したがって、findByPrimaryKey メソッドに主キー・オブジェクトを指定すると、この Bean を取得できます。

必要であれば、Bean を検索するために他のファインダ・メソッドを開発できます。これらのメソッドの名前は、find<name> となります。

注意： ホーム・インタフェースにあるすべてのファインダ・メソッドの戻り型は、エンティティ Bean のリモート・インタフェース、またはそれを実装するオブジェクトの Enumeration である必要があります。Collection を戻す動作はサポートされません。

Bean クラス内で実装されるすべてのファインダ・メソッドの戻り型は、主キー、またはその Enumeration を戻します。コンテナは、ejbFind<name> メソッドで戻される主キーごとに、適切なエンティティ Bean のリモート・インタフェースを取得します。

例 4-1 発注のホーム・インタフェース

エンティティ Bean の具体例を示すために、発注を管理する Bean を作成するとします。エンティティ Bean には、顧客が発注する品目のリストが含まれています。

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 purchase;

import javax.ejb.*;
import java.rmi.RemoteException;
import java.sql.SQLException;

public interface PurchaseOrderHome extends EJBHome
{
    // Create a new PO
    public PurchaseOrder create() throws CreateException, RemoteException;

    // Find an existing one
    public PurchaseOrder findByPrimaryKey (String POnumber)
        throws FinderException, RemoteException;
}
```

リモート・インタフェース

エンティティ Bean のリモート・インタフェースは、顧客に表示され、メソッドを起動するインタフェースです。このインタフェースにより javax.ejb.EJBObject が拡張され、ビジネス・ロジックのメソッドが定義されます。発注エンティティ Bean の場合、リモート・インタフェースには、発注に品目を追加するメソッド、発注に含まれる全品目のリストを取得するメソッド、発注価格合計を計算するメソッドが含まれています。

```
package purchase;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;
```

```
import java.sql.SQLException;
import java.util.Vector;

public interface PurchaseOrder extends EJBObject {
    // Price the purchase order
    public float price() throws RemoteException;

    // getContents returns a Vector of LineItem objects in the purchase
    public Vector getContents() throws RemoteException;

    //Add items to the purchase
    public void addItem (int sku, int count) throws RemoteException;
}
```

主キー

エンティティ Bean のインスタンスごとに、それを他のインスタンスから一意に識別する主キーがあります。主キーを定義するには、次の 2 通りの方法があります。

- 主キーの型を、`java.lang.String` のような定式型に定義します。主キーが定式データ型の場合は、型をディプロイメント・ディスクリプタ内の `<prim-key-class>` で定義します。
- 主キーの型を、`<name>PK` クラスの中に、シリアル化可能オブジェクトとして定義します。主キーが複合データ型の場合は、シリアル化可能なクラス内で定義します。このクラスは、ディプロイメント・ディスクリプタの `<prim-key-class>` 要素内で宣言します。

定式型としての主キーの定義

主キーを定式型として定義するには、主キーのデータ型をディプロイメント・ディスクリプタ内で定義します。

発注の例では、主キーを `java.lang.String` として定義しています。

```
<enterprise-beans>
  <entity>
    <ejb-name>test/purchase</ejb-name>
    <home>purchase.PurchaseOrderHome</home>
    <remote>purchase.PurchaseOrder</remote>
    <ejb-class>purchaseServer.PurchaseOrderBean</ejb-class>
    <persistence-type>Bean</persistence-type>
    <prim-key-class>java.lang.String</prim-key-class>
    <reentrant>False</reentrant>
  </entity>
  ...
</enterprise-beans>
```

クラス内での主キーの定義

主キーが単純データ型より複雑な場合は、`<name>PK` という名前でシリアライズ可能なクラスにする必要があります。このクラス内で、この主キーに固有の実装用に提供する `equals` および `hashCode` メソッドを実装する必要があります。

顧客の例では、主キーである顧客識別子を `PurchaseOrderPK.java` 内で宣言しています。

```
package purchase;

public class PurchaseOrderPK implements java.io.Serializable
{
    public int orderid;

    public boolean equals(Object obj) {
        if ((obj instanceof PurchaseOrderPK) &&
            (((PurchaseOrderPK)obj).orderid == this.orderid))
            return true;
        return false;
    }

    public int hashCode() {
        return orderid;
    }
}
```

主キーを定義するクラスは、ディプロイメント・ディスクリプタ内で次のように宣言されています。

```
<enterprise-beans>
  <entity>
    <ejb-name>test/purchase</ejb-name>
    <home>purchase.PurchaseOrderHome</home>
    <remote>purchase.PurchaseOrder</remote>
    <ejb-class>purchaseServer.PurchaseOrderBean</ejb-class>
    <persistence-type>Bean</persistence-type>
    <prim-key-class>purchase.PurchaseOrderPK</prim-key-class>
    <reentrant>False</reentrant>
  </entity>
  ...
</enterprise-beans>
```

主キーの管理

`ejbCreate` メソッドは、主として主キーの作成を受け持ちます。これには、主キーの作成、キーの永続データ表現の作成、一意の値へのキーの初期化、および実行者へのこのキーの戻しが含まれます。`ejbFindByPrimaryKey` メソッドは、主キーが一意かどうかを検証し、もう一度コンテナに戻す操作を受け持ちます。

発注の例では、前述のメソッドにより次の操作が実行されます。

- `ejbCreate` メソッドでは、主キー `ponumber` が発注番号の順序内で次に使用可能な番号に初期化されます。
- `ejbFindByPrimaryKey` メソッドは、発注番号が有効かどうかをチェックし、この番号をコンテナに戻すために実装されます。

```
// The create methods takes care of generating a new PO and returns
// its primary key
public String ejbCreate () throws CreateException, RemoteException
{
    String ponumber = null;
    try {
        //retrieve the next available purchase order number
        #sql { select ponumber.nextval into :ponumber from dual };
        //assign this number as this instance's identification number
        #sql { insert into pos (ponumber, status) values (:ponumber, 'OPEN') };
    } catch (SQLException e) {
        throw new PurchaseException (this, "create", e);
    }
    return ponumber;
}

// The ejbFindByPrimaryKey method verifies that the POnumber exists. This
// method must return the primary key to the container.. which in turn
// retrieves the instance based on the primary key. So.. this method must
// only verify that the primary key is valid.
public String ejbFindByPrimaryKey (String ponumber)
    throws FinderException, RemoteException
{
    try {
        int count;
        #sql { select count (ponumber) into :count from pos
              where ponumber = :ponumber };

        // There has to be one
        if (count != 1)
            throw new FinderException ("Inexistent PO: " + ponumber);
    } catch (SQLException e) {
        throw new PurchaseException (this, "findByPrimaryKey", e);
    }
    // The ponumber is the primary key
    return ponumber;
}
```

エンティティ Bean クラス

エンティティ Bean クラスでは、次のメソッドが実装されます。

- Bean インスタンス作成用の空のコンストラクタ。
- `ejbCreate` メソッドや、`ejbFindByPrimaryKey` のようなファインダ・メソッドなど、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。
- リモート・インタフェースで宣言されているビジネス・ロジックのメソッド。
- `EntityBean` インタフェースで宣言されているメソッド。

次のコードでは、`PurchaseOrderBean` という名のエンティティ Bean のメソッドが実装されます。

1. 変数の宣言

発注 Bean では、顧客のショッピング・カートにある全品目を格納する `Vector` が宣言されています。また、エンティティ Bean の環境情報を取得するために、エンティティ・コンテキストが定義されています。

```
#sql iterator ItemsIter (int skunumber, int count, String description,
                        float price);

public class PurchaseOrderBean implements EntityBean {
    EntityContext ctx;
    Vector items;           // The items in the PO (instances of LineItem)
```

2. リモート・インタフェース・メソッドの実装

次の例は、リモート・インタフェースで宣言されている Bean メソッド `price`、`getContents` および `addItem` の実装を示しています。

```
public float price() throws RemoteException {
    float price = 0;
    Enumeration e = items.elements ();
    while (e.hasMoreElements ()) {
        LineItem item = (LineItem)e.nextElement ();
        price += item.quantity * item.price;
    }

    // 5% discount if buying more than 10 items
    if (items.size () > 10)
        price -= price * 0.05;

    // Shipping is a constant plus function of the number of items
    price += 10 + (items.size () * 2);

    return price;
}
```

```
}

// The getContents methods has to load the descriptions
public Vector getContents() throws RemoteException {
    return items;
}

// The add Item method gets the price and description
public void addItem (int sku, int count) throws RemoteException {
    try {
        String description;
        float price;
        #sql { select price, description into :price, :description
                from skus where skunumber = :sku };
        items.addElement (new LineItem (sku, count, description, price));
    } catch (SQLException e) {
        throw new PurchaseException (this, "addItem", e);
    }
}
```

3. EntityBean インタフェースのメソッドの実装

ビジネス・ロジックのメソッドを実装後に、次の要素も提供する必要があります。

- Bean インスタンスのパブリック・コンストラクタ。このコンストラクタは引数を取りません。コンテナは、コンストラクタを起動してエンティティ Bean クラスのインスタンスを作成します。コンストラクタは、空の実装にすることもできます。
- ホーム・インタフェースで定義されている各 create メソッドの ejbCreate メソッド。
- ホーム・インタフェースで定義されている各 find<name> メソッドの ejbFind<name> メソッド。これには、少なくとも、コンテナに主キーを戻す ejbFindByPrimaryKey メソッドが含まれます。
- EntityBean メソッドの実装。これらのメソッドは、コンテナにより必要に応じてコールされるコールバック・メソッドです。ほとんどのコールバック・メソッドは、エンティティ Bean のデータの永続性管理に関係しています。

パブリック・コンストラクタ パブリック・コンストラクタは、Bean インスタンスを作成するためにコンテナによりコールされます。ejbCreate および ejbPostCreate メソッドは、このインスタンスを初期化するために起動されます。次の例は、発注用のコンストラクタを示しています。

```
//provide an empty constructor for the creating the instance
public void PurchaseOrderBean() {}
```

create メソッド : ejbCreate および ejbPostCreate 図 4-2 のように、ejbCreate および ejbPostCreate メソッドは、対応する create メソッドの起動時に起動されます。すべてのメソッドは、同じ引数を取ります。通常、ejbCreate メソッドではすべての永続データが初期化され、ejbPostCreate ではエンティティのコンテキストを伴う初期化が実行されます。コンテキスト情報は、ejbCreate の起動時には使用できませんが、ejbPostCreate の起動時に使用可能になります。

次の例は、発注の例の ejbCreate と ejbPostCreate を示しています。ejbCreate メソッドでは、発注番号である主キーが初期化され、このキーが実行者に戻されます。発注明細項目の Vector は、ejbPostCreate 内で初期化されます。

```
// The create methods takes care of generating a new PO and returns
// its primary key
public String ejbCreate () throws CreateException, RemoteException
{
    String ponumber = null;
    try {
        //retrieve the next available purchase order number
        #sql { select ponumber.nextval into :ponumber from dual };
        //assign this number as this instance's identification number
        #sql { insert into pos (ponumber, status) values (:ponumber, 'OPEN') };
    } catch (SQLException e) {
        throw new PurchaseException (this, "create", e);
    }
    return ponumber;
}

// create a vector to contain the purchase order line items. since this
// is performed only once and needed for the lifetime of the object, it is
// appropriate to create the vector in either ejbCreate or ejbPostCreate.
public void ejbPostCreate () {
    items = new Vector ();
}
```

ファインダ・メソッド すべてのエンティティ Bean では、ejbFindByPrimaryKey メソッドを提供する必要があります。また、他のタイプのファインダ・メソッドも使用できます。開発者はホーム・インタフェースで宣言されるファインダ・メソッドを実装する必要があるため、この種のメソッドを使用できる数に制限はありません。唯一の制限事項は、ejbFindByPrimaryKey メソッドを除くファインダ・メソッドはいずれも、リモート・インタフェースの参照、またはリモート・インタフェースの複数の参照を含む Enumeration を戻す必要があることです。ejbFindByPrimaryKey メソッドでは、主キーを戻す必要があります。

注意： Collection は現時点ではサポートされていないため、戻り型には使用できません。

他のファインダ・メソッドを提供するには、次の操作が必要です。

1. そのメソッドをホーム・インタフェースで `find<name>` として宣言します。
2. そのメソッドを Bean クラスで `ejbFind<name>` として実装します。

次のコードは、発注の例の `ejbFindByPrimaryKey` メソッドを示しています。このメソッドでは、主キーが有効かどうかを検証されます。有効であれば、そのキーがコンテナに戻されます。コンテナは、このキーの適切な Bean インスタンスを取得して、クライアントに参照を戻します。

```
// The findByPrimaryKey method verifies that the POnumber exists. This
// method must return the primary key to the container.. which in turn
// retrieves the instance based on the primary key. So.. this method must
// only verify that the primary key is valid.
public String ejbFindByPrimaryKey (String ponumber)
    throws FinderException, RemoteException
{
    try {
        int count;
        #sql { select count (ponumber) into :count from pos
              where ponumber = :ponumber };

        // There has to be one
        if (count != 1)
            throw new FinderException ("Inexistent PO: " + ponumber);
    } catch (SQLException e) {
        throw new PurchaseException (this, "findByPrimaryKey", e);
    }
    // The ponumber is the primary key
    return ponumber;
}
```

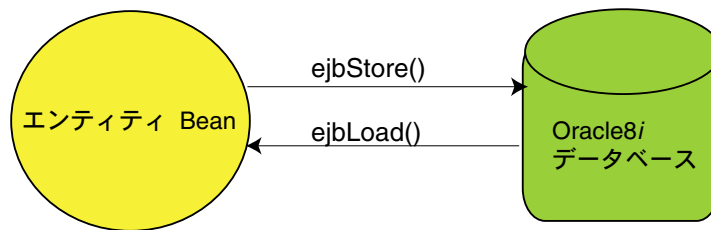
EntityBean メソッド：ロードと格納 エンティティ Bean とセッション Bean の主な違いは、エンティティ Bean では管理対象の永続データが保持されることです。データが永続データとして定義されている場合は、絶えずデータベースやファイルなどのリソースに保存し、そこからリストアップする必要があります。Bean が破棄された場合は、永続データをそのままリストアップできます。

すべてのエンティティ Bean で実装される `EntityBean` インタフェースでは、永続データを管理する次のコールバック・メソッドを定義します。

- `ejbStore` — データを永続記憶域に保存します。
コンテナは、トランザクションがコミットされる直前、または Bean が削除される直前に、常に `ejbStore` を起動して永続データの既存の値を保存します。
- `ejbLoad` — 永続記憶域に保存されたデータを Bean にロードします。
コンテナは、Bean がインスタンス化された直後、またはトランザクションの開始直後に、`ejbLoad` を起動します。

図 4-4 は、`ejbStore` を使用してエンティティ Bean 内の永続データをデータベースに保存する方法を示しています。また、データは、`ejbLoad` によってデータベースからリストアされます。

図 4-4 永続データの管理



次のコードは、発注の例のメソッドを示しています。`ejbStore` メソッドでは、発注品目がデータベースに保存されます。`ejbLoad` メソッドでは、発注品目がデータベースからリストアされます。

```
// The store method replaces all entries in the lineitems table with the
// new entries from the bean
public void ejbStore() throws RemoteException {
    // Get the purchase order number
    String ponumber = (String)ctx.getPrimaryKey();

    try {
        // Delete old entries in the database
        #sql { delete from lineitems where ponumber = :ponumber };

        // Insert new entries from the vector in the bean. Crude, but effective.
        Enumeration e = items.elements ();
        while (e.hasMoreElements ()) {
            LineItem item = (LineItem)e.nextElement ();
            #sql { insert into lineitems (ponumber, skunumber, count)
                    values (:ponumber, :(item.sku), :(item.quantity))
        }
    }
}
```

```

        };
    }
    } catch (SQLException e) {
        throw new PurchaseException (this, "store", e);
    }
}

// The load method populates the items array with all the saved
// line items
public void ejbLoad() throws RemoteException {
    // Get the purchase order number
    String ponumber = (String)ctx.getPrimaryKey();

    // Load all line items into a new vector.
    try {
        items = new Vector ();
        ItemsIter iter = null;
        try {
            #sql iter = {
                select lineitems.skunumber, lineitems.count,
                    skus.description, skus.price
                from lineitems, skus
                where ponumber = :ponumber and lineitems.skunumber = skus.skunumber
            };

            while (iter.next ()) {
                LineItem item =
                    new LineItem (iter.skunumber(), iter.count(), iter.description(),
                                iter.price());
                items.addElement (item);
            }
        } finally {
            if (iter != null) iter.close ();
        }
    } catch (SQLException e) {
        throw new PurchaseException (this, "load", e);
    }
}

```

EntityBean メソッド: 削除 クライアントが削除メソッドを起動すると、`ejbRemove` メソッドが起動されます。Bean 管理の永続 Bean の場合は、このメソッド内で Bean に対応付けられたデータベースからデータを削除する必要があります。

次の例は、発注明細項目をデータベースから削除する方法を示しています。

```
// The remove method deletes all line items belonging to the purchase order
public void ejbRemove() throws RemoteException {
    // Get the purchase order number from the session context
    String ponumber = (String)ctx.getPrimaryKey();
    try {
        //delete the line item vector for the purchase order
        #sql { delete from lineitems where ponumber = :ponumber };
        //delete the row associated with the purchase order
        #sql { delete from pos where ponumber = :ponumber };
    } catch (SQLException e) {
        throw new PurchaseException (this, "remove", e);
    }
}
```

EntityBean メソッド: コンテキストの設定 アプリケーションの存続期間中に、コンテキスト内の情報にアクセスする必要がある場合は、そのコンテキストを `setEntityContext` 内で保存する必要があります。

```
//Set the provided context to this.ctx
public void setEntityContext(EntityContext ctx)
{
    this.ctx = ctx;
}

//reinitialize the context to null
public void unsetEntityContext()
{
    this.ctx = null;
}
```

EntityBean メソッド: 活性化と受動化 現在、Oracle では活性化と受動化はサポートされていません。ただし、これらのメソッド用に空の実装を引き続き提供する必要があります。

```
//There are no requirements for ejbActivate for this bean
public void ejbActivate()
{
}

//There are no requirements for ejbPassivate for this bean
public void ejbPassivate()
{
}
```

4. LineItem クラス

発注書アプリケーションでは、個々の注文が永続 Java オブジェクト LineItem を使用して永続的に格納されます。つまり、エンティティ Bean は各発注項目の管理を EJB 以外の Java オブジェクトに移譲します。

```
package purchase;

public class LineItem implements java.io.Serializable {
    public int sku;
    public int quantity;
    public String description;
    public float price;

    //Persistently manage each line item within the purchase order.
    public LineItem (int sku, int quantity, String description, float price) {
        //Each line item has the following information: SKU number, quantity,
        // description, and price.
        this.sku = sku;
        this.quantity = quantity;
        this.description = description;
        this.price = price;
    }
}
```

エンティティ・データ用のデータベース表および列の作成

エンティティ Bean で永続データがデータベースに格納される場合は、そのエンティティ Bean に必要な列を含む適切な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してください。

発注の例では、次の表を作成する必要があります。

表	列	説明
SKUS	■ skunumber: 品目番号	この表には、倉庫内の各品目が記述されます。
	■ description: 品目摘要	
	■ price: 品目の価格	
POS	■ ponumber: 発注番号	顧客発注の状態を管理する表。発注の状態が含まれます。
	■ status: オープン、実施済みまたは出荷済み	
LINEITEMS	■ ponumber: 発注番号	顧客が発注した個別品目をすべて含む表。
	■ skunumber: 品目番号	
	■ count: 発注品目数	

次の例は、これらのフィールドを作成する SQL コマンドを示しています。

```
-- This sql scripts create the SQL tables used by the PurchaseOrder bean

-- The sku table lists all the items available for purchase
create table skus (skunumber number constraint pk_skus primary key,
                  description varchar2(2000),
                  price number);

-- The pos table stores information about purchase orders
-- The status column is 'OPEN', 'EXECUTED' or 'SHIPPED'
create table pos (ponumber number constraint pk_pos primary key,
                 status varchar2(30));

-- The ponumber sequence is used to generate PO ids
create sequence ponumber;

-- The lineitems table stores the contents of a po
-- The skunumber is a reference into the skus table
-- The ponumber is a reference into the pos table
create table lineitems (ponumber number constraint fk_pos references pos,
                       skunumber number constraint fk_skus references skus,
                       count number);

commit;

exit;
```

エンティティ Bean の配置

エンティティ Bean の配置方法は、セッション Bean の場合と同じです。詳細は、2-19 ページの「[EJB の配置](#)」を参照してください。セッション Bean の場合と同じ方法で、Bean の XML デプロイメント・ディスクリプタを作成し、Bean のすべてのファイルを含む JAR ファイルを作成し、`deployejb` ツールを使用して Bean をデータベースにロードし、公開する必要があります。

XML デプロイメント・ディスクリプタの詳細は、[付録 A「XML デプロイメント・ディスクリプタ」](#)を参照してください。付録 A には、エンティティ Bean の永続性を定義する方法の詳細も記載されています。完全性を期すために、ここでは発注の例のデプロイメント・ディスクリプタの編成方法を示します。発注書では、マップする論理名が定義されておらず、`<run-as>` オプションを使用していないため、必要なのは EJB デプロイメント・ディスクリプタのみです。

例 4-2 発注の EJB デプロイメント・ディスクリプタ

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems Inc.//DTD Enterprise JavaBeans 1.1
//EN" "ejb-jar.dtd">
```

```
<ejb-jar>
  <enterprise-beans>
    <entity>
      <description>no description</description>
      <ejb-name>test/purchase</ejb-name>
      <home>purchase.PurchaseOrderHome</home>
      <remote>purchase.PurchaseOrder</remote>
      <ejb-class>purchaseServer.PurchaseOrderBean</ejb-class>
      <persistence-type>Bean</persistence-type>
      <prim-key-class>java.lang.String</prim-key-class>
      <reentrant>False</reentrant>
    </entity>
  </enterprise-beans>
</assembly-descriptor>
  <security-role>
    <description>no description</description>
    <role-name>PUBLIC</role-name>
  </security-role>
  <method-permission>
    <description>no description</description>
    <role-name>PUBLIC</role-name>
    <method>
      <ejb-name>test/purchase</ejb-name>
      <method-name>*</method-name>
    </method>
  </method-permission>
  <container-transaction>
    <description>no description</description>
    <method>
      <ejb-name>test/purchase</ejb-name>
      <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
</assembly-descriptor>
</ejb-jar>
```

クライアントから配置済みエンティティ Bean へのアクセス

配置済みのエンティティ Bean にアクセスするには、クライアントで次のどちらかを実行します。

- [新規エンティティ Bean の作成](#)
- [既存のエンティティ Bean へのアクセス](#)

新規エンティティ Bean の作成

エンティティ Bean にアクセスする場合は、最初に Bean のホーム・インタフェースの位置を特定する必要があります。ホーム・インタフェースは、JNDI を介してネームスペースから取得します。URL には、次の構文を使用する必要があります。

```
<service_name>://<hostname>:<iiop_listener_port>:<SID>/<published_obj_name>
```

この構文の詳細は、2-17 ページの「[ホーム・インタフェース・オブジェクトの取得](#)」を参照してください。

例 4-3 JNDI ネームスペースからのホーム・インタフェースの取得

次の例では、/test/purchase 内にあって公開されている EJB のホーム・インタフェースを取得しています。ホスト、ポートおよび SID は、それぞれ localhost、2471 および ORCL です。

```
String serviceURL = "sess_iiop://localhost:2471:ORCL";
String objectName = "/test/purchase";

Hashtable env = new Hashtable();
env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put(Context.SECURITY_PRINCIPAL, user);
env.put(Context.SECURITY_CREDENTIALS, password);
env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
Context ic = new InitialContext (env);

CustomerHome ch = (CustomerHome)ic.lookup (serviceURL + objectName);
Customer myCust = (Customer) ch.create();
```

注意： 検索での型キャストには、`narrow` メソッドが不要であることに注意してください。Oracle8i の `lookup` メソッドでは、適切な `narrowing` 機能が自動的に実行されます。ただし、戻されるオブジェクトをキャストする型は指定する必要があります。

既存のエンティティ Bean へのアクセス

クライアントでは、次のいずれかの方法で既存のエンティティ Bean にアクセスできます。

- メソッドのコールで戻されるパラメータとしてなど、他者から参照を受け取ります。
- エンティティ Bean の主キーを指定し、そのリモート参照を戻す findByPrimaryKey メソッドを起動します。
- Bean の Handle オブジェクトを指定し、Handle オブジェクトに対して getEJBObject メソッドを起動します。このハンドルは、getHandle メソッドを使用してエンティティ Bean オブジェクトから作成されています。ハンドルは、そのまま別のオブジェクトに渡しても、後で使用できるようにシリアルライズして格納してもかまいません。

Bean 管理の永続性とコンテナ管理の永続性の違い

エンティティ Bean 内の永続データを管理するには、Bean 管理の永続性とコンテナ管理の永続性という 2 つの方法があります。Bean 管理の永続 Bean とコンテナ管理の永続 Bean の主な違いは、エンティティ Bean のデータの永続性を管理する担当者により定義されます。

次の表は、実際の意味で、両者の定義と両者におけるプログラムおよび宣言の違いの概要を示しています。

	Bean 管理の永続性	コンテナ管理の永続性
永続性の管理	EntityBean の ejbStore および ejbLoad メソッド内に永続性管理を実装する必要があります。この 2 つのメソッドには、永続データの保存とリストアに関するロジックを含めます。 たとえば、ejbStore メソッドには、エンティティ Bean のデータを適切なデータベースに格納するロジックを含める必要があります。このロジックがなければ、データが失われることがあります。Bean 管理の永続性の実装例については、4-16 ページの「3. EntityBean インタフェースのメソッドの実装」を参照してください。	永続データの管理は自動的に実行されます。つまり、コンテナが Bean にかわって永続性マネージャを起動します。 コミット前にデータを準備したり、データベースからリフレッシュされた後でデータを操作するには、ejbStore および ejbLoad を使用します。コンテナは、常にコミット直前に ejbStore メソッドを起動します。また、データベースからの CMP データをもう一度インスタンス化した直後に、常に ejbLoad メソッドを起動します。
ファインダ・メソッドの使用可否	findByPrimaryKey メソッドと、他に実装するファインダ・メソッドを使用できます。	使用できるのは、findByPrimaryKey メソッドと、WHERE 句のファインダ・メソッドのみです。
CMP フィールドの定義	該当なし	EJB ディプロイメント・ディスクリプタに必須です。主キーも CMP フィールドとして宣言する必要があります。

Bean 管理の永続性	コンテナ管理の永続性
CMP フィールドからリソース宛先へのマッピング	該当なし
永続性マネージャの定義	該当なし
	必須。永続性マネージャに依存します。
	Oracle 固有のディプロイメント・ディスクリプタに必須です。永続性マネージャについては、次項を参照してください。

コンテナ管理の永続性

Bean の永続データをコンテナで管理するように選択できます。コンテナにより永続データがデータベースに格納され、そこから再ロードされるため、開発および管理作業が軽減されます。

コンテナ管理の永続性を使用すると、コンテナにより、永続性管理のビジネス・ロジックを提供する永続性マネージャ・クラスが起動されます。このリリースでサポートされている永続性マネージャは、Oracle Persistence Service Interface Reference Implementation (PSI-RI) のみです。

コンテナによる永続データの管理を有効化するには、次の操作が必要です。

1. 適切な Bean クラスのコールバック・メソッドを変更します。
2. 主キーを定義します。
3. ディプロイメント・ディスクリプタ内でコンテナ管理の永続フィールドを宣言します。
4. 永続性マネージャ・クラスを宣言します。
5. コンテナ管理の永続フィールドをデータベースにマップします。

Bean クラスのコールバック・メソッドの変更

永続データの管理が不要であれば、コンテナによる Bean 管理を選択します。これは、なんらかのコールバック・メソッドをコンテナとして実装する必要がなく、永続性マネージャにより永続性と主キーの管理が実行されることを意味します。これらのメソッドは引き続きコンテナによりコールされるため、他の目的を持つロジックを追加できます。従来どおり、すべてのコールバック・メソッド用に、少なくとも空の実装を提供する必要があります。

次の表は、Bean クラスのコールバック関数の実装要件の詳細を示しています。

コールバック・メソッド	必要な機能
<code>ejbCreate</code>	Bean 管理の永続 Bean と同じ機能。主キーなど、コンテナ管理の永続フィールドをすべて初期化する必要があります。
<code>ejbPostCreate</code>	Bean 管理の永続 Bean と同じ機能。エンティティ・コンテキストを伴うものなど、他の初期化を提供するかどうかを選択できます。
<code>ejbRemove</code>	外部リソースから永続データを削除する機能は不要です。エンティティ Bean に対応付けられた永続データは、すべて永続性マネージャによりデータベースから削除されます。少なくともコールバック用の空の実装を提供する必要があります。これは、必要なクリーン・アップ機能を実行するロジックを追加できることを意味します。
<code>ejbFindByPrimaryKey</code>	主キーをコンテナに戻すための機能は不要です。主キーは、 <code>ejbCreate</code> メソッドで初期化された後は、コンテナにより管理されます。したがって、通常、このメソッドに必要な機能はコンテナにより実行されます。このメソッド用に空の実装を提供する必要があります。
<code>ejbStore</code>	このメソッド内で永続データを保存するための機能は不要です。すべての永続データは、永続性マネージャによりデータベースに保存されます。ただし、コンテナは永続性マネージャを起動する前に <code>ejbStore</code> メソッドを起動するため、少なくとも空の実装を提供する必要があります。これにより、永続データの保存前にデータ管理やクリーン・アップを実行できます。
<code>ejbLoad</code>	このメソッド内で永続データをリストアするための機能は不要です。すべての永続データは、永続性マネージャによりリストアされます。ただし、コンテナは永続性マネージャを起動した後に <code>ejbLoad</code> メソッドを起動するため、少なくとも空の実装を提供する必要があります。これにより、Bean にリストアされた永続データを操作するロジックを実行できます。
<code>setEntityContext</code>	<p>Bean インスタンスをコンテキスト情報に対応付けます。コンテナでは、Bean の作成後にこのメソッドがコールされます。エンタープライズ Bean では、トランザクション管理に使用するコンテキスト・オブジェクトへの参照がインスタンス変数に格納できます。自分のトランザクションを管理する Bean では、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。</p> <p>また、このメソッド内で Bean の存続期間中だけ存在するリソースを割り当てることもできます。これらのリソースは、<code>unsetEntityContext</code> で解放する必要があります。</p>
<code>unsetEntityContext</code>	対応付けられたエンティティ・コンテキストの設定を解除し、 <code>setEntityContext</code> で割り当てられたリソースを解放します。

WHERE 句のファインダ・メソッド Oracle では、find<name> ネーミング構文を使用し、CMP 専用のファインダ・メソッドを介して、永続データ表に対する SQL 問合せを実行できます。このメソッドは、SQL 問合せの where 句を示す String を取ります。したがって、String には、「select * from <table>」を除く文全体を挿入します。空の文字列を指定すると、この表からすべての値が選択されます。

ホーム・インタフェースでは、このようなファインダ・メソッドを定義する必要があります。コンテナは、ファインダ・メソッドの WHERE 句を満たす実装を提供します。

たとえば、次の例では、ホーム・インタフェース内で2つのファインダ・メソッドを定義しています。一方ではすべての顧客が検索され、他方では1人の顧客が検索されます。表中のすべての顧客を検索する findAllCustomers メソッドで、Enumeration がどのように戻されるかに注意してください。

注意： 通常、ファインダ・メソッドの複数項目の戻り型は Collection となります。ただし、このリリースでは、Collection はサポートされていません。ファインダ・メソッドから複数項目を受け取るには、Enumeration 型を使用する必要があります。

```
public Customer findByWhere(String whereString)
    throws RemoteException, FinderException;

public java.util.Enumeration findMultipleCustomers(String whereString)
    throws RemoteException, FinderException;
```

単一の顧客を検索する場合は、顧客名を指定して次の SQL 文を使用します。

```
select * from customer where name = "Smith, John";
```

findByWhere ファインダ・メソッドには、「select * from customer」を除く文全体を挿入します。次のように、永続性表に対する選択が必要であるとします。

```
Customer find_customer = findByWhere ("where name = ", + custname);
```

特定の条件に基づいて従業員から数行を選択する場合、SQL 文は次のようになります。

```
select * from customer where item_bought = treadmill order by name;
```

find<name> メソッドは、この例では findByWhere で、「select * from employee」以外の文全体が含まれます。永続性表に対する一致をすべて選択する必要があるとします。この場合は次のようになります。

```
Enumeration customer_list = findMultipleCustomers(
    "where item_bought = treadmill order by name");
```

また、顧客リスト全体が必要な場合は、空の文字列を指定します。コンテナにより「select * from customer」が起動され、すべてのレコードが検索されます。

```
Enumeration customer_list = findMultipleCustomers();
```

主キーの定義

Bean 管理とコンテナ管理における永続主キーの定義の主な違いは、キー内のフィールドをディプロイメント・ディスクリプタ内でコンテナ管理の永続フィールドとして宣言する必要があります。主キーのすべてのフィールドは、SQL の型にマップできるシリアル化可能なプリミティブ型に制限されます。詳細は、A-28 ページの「[永続フィールド](#)」を参照してください。

主キーを定義するには、次の 2 通りの方法があります。

- [主キーとしての単一オブジェクトの定義](#)
- [複雑な主キー・クラスの定義](#)

主キーとしての単一オブジェクトの定義 主キーを、コンテナ管理の永続フィールドおよびその型として、ディプロイメント・ディスクリプタ内で定義します。次の例は、`<cmp-field>` および `<primkey-field>` として宣言され、その型が `<prim-key-class>` 内で宣言されている主キー `custid` を示しています。

```
<enterprise-beans>
  <entity>
    <description>customer bean</description>
    <ejb-name>/test/customer</ejb-name>
    <home>customer.CustomerHome</home>
    <remote>customer.Customer</remote>
    <ejb-class>customerServer.CustomerBean</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.String</prim-key-class>
    <reentrant>False</reentrant>
    <cmp-field><field-name>custid</field-name></cmp-field>
    <cmp-field><field-name>name</field-name></cmp-field>
    <cmp-field><field-name>addr</field-name></cmp-field>
    <primkey-field>custid</primkey-field>
  </entity>
</enterprise-beans>
```

Bean クラス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。

これは、SQLJ を介して SQL の型にマップできる Java の型にする必要があります。また、このオブジェクトは、シリアル化可能である必要があります。詳細は、A-6 ページの「[エンティティ Bean の要素](#)」を参照してください。

複雑な主キー・クラスの定義 主キーが単純なデータ型より複雑な場合は、主キーを構成するフィールドを、ディプロイメント・ディスクリプタ内でコンテナ管理のフィールドとして定義します。次に、各フィールドをクラス内で主キーとして宣言します。このクラスは、シリアル化可能である必要があります。また、Bean クラス内で宣言される主キー変数は、いずれも SQL の型にマップできる型にする必要があります。

Bean クラス内では、主キー変数を `public` として宣言する必要があります。空の主キー・インスタンスを作成するために、引数を取らないコンストラクタも提供します。

シリアル化可能な主キー・クラス内で、この主キーに固有の実装用に提供する `equals` および `hashCode` メソッドを実装します。

次の例は、各船室を船名、デッキ名および船室番号で識別する、クルーズ客船の船室 Bean の主キーを示しています。

```
package cruise;

public class CabinPK implements java.io.Serializable
{
    //Ship names { Castaway, LightFantastic, FantasyRide }
    public String ship;

    //Deck names { Upper Promenade, Promenade, Lower Promenade, Main Deck,
                  Lower Deck }
    public String deck;

    //Cabin numbers A100-N300
    public String cabin;

    //empty constructor
    public CabinPK ( ) { }

    public boolean equals(Object obj) {
        if ((obj instanceof CabinPK) &&
            ((CabinPK)obj).ship == this.ship) &&
            ((CabinPK)obj).deck == this.deck) &&
            ((CabinPK)obj).cabin == this.cabin))
            return true;
        return false;
    }

    public int hashCode() {
        return ((ship + deck + cabin).hash);
    }
}
```

主キーを定義するクラスは、XML デプロイメント・ディスクリプタ内で次のように宣言されています。

```
<enterprise-beans>
  <entity>
    <ejb-name>CabinBean</ejb-name>
    <home>cruise.CabinHome</home>
    <remote>cruise.Cabin</remote>
    <ejb-class>cruiseServer.CabinBean</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>cruise.CabinPK</prim-key-class>
    <reentrant>False</reentrant>
    <cmp-field><field-name>deck</field-name></cmp-field>
    <cmp-field><field-name>cabin</field-name></cmp-field>
  ...
</enterprise-beans>
```

デプロイメント・ディスクリプタの定義の詳細は、A-6 ページの「[エンティティ Bean の要素](#)」を参照してください。

主キーの管理 主キーを含め、すべての CMP フィールドは `ejbCreate` メソッド内で初期化する必要があります。発注の例の `ejbCreate` メソッドでは、主キー `ponumber` が発注番号の順序内で次に使用可能な番号に初期化されます。

```
// The create methods takes care of generating a new PO and returns
// its primary key
public String ejbCreate () throws CreateException, RemoteException
{
    String ponumber = null;
    try {
        //retrieve the next available purchase order number
        #sql { select ponumber.nextval into :ponumber from dual };
        //assign this number as this instance's identification number
        #sql { insert into pos (ponumber, status) values (:ponumber, 'OPEN') };
    } catch (SQLException e) {
        throw new PurchaseException (this, "create", e);
    }
    return ponumber;
}
```

永続フィールドの宣言

コンテナ管理の永続フィールドは、すべて Bean クラス内で `public` として宣言する必要があります。一次フィールドとしては宣言できません。また、これらのフィールドは、SQL の型にマップできるシリアライズ可能なプリミティブ型に制限されます。詳細は、A-29 ページの表 A-2「永続変数に関してサポートされない Java の型」を参照してください。

永続性プロバイダの宣言

コンテナは、CMP Bean を管理するために永続性プロバイダを起動します。このリリースでは、Oracle Persistence Service Interface Reference Implementation (PSI-RI) がサポートされます。プロバイダの定義の詳細は、A-27 ページの「コンテナ管理の永続性の定義」を参照してください。

注意： 永続性マネージャを含むクラスがデータベースにロードされていることを確認してください。

次の例は、Oracle 固有のデプロイメント・ディスクリプタのうち、永続性マネージャを定義する部分を示しています。

```
...
<persistence-provider>
  <description> specifies a type of persistence manager </description>
  <persistence-name>psi-ri</persistence-name>
  <persistence-deployer>oracle.aurora.ejb.persistence.ocmp.OcmpEntityDeployer</p
  <persistence-deployer>
</persistence-provider>

<persistence-descriptor>
  <description> This specifies a particular type of persistence manager to be us
  ed for a bean.  param is where you would put bean specific persistence info in t
  he format of params.  The deployment process just passes what's in the param to
  the persistence deployer.  For the baby persistence, we do parse the persistence
  -mapping but for other persistence backend we don't do anything with the params
  </description>
  <ejb-name>customerbean</ejb-name>
  <persistence-name>psi-ri</persistence-name>
  <psi-ri>
    <schema>SCOTT</schema>
    <table>customers</table>
    <attr-mapping>
      <field-name>custid</field-name>
      <column-name>cust_id</column-name>
    </attr-mapping>
    <attr-mapping>
      <field-name>name</field-name>
```

```
        <column-name>cust_name</column-name>
    </attr-mapping>
    <attr-mapping>
        <field-name>addr</field-name>
        <column-name>cust_addr</column-name>
    </attr-mapping>
</psi-ri>
</persistence-descriptor>
</oracle-descriptor>
```

コンテナ管理の永続フィールドのマッピング

Bean 内で定義される CMP データ・フィールドは、すべてディプロイメント・ディスクリプタの <cmp-field> 要素内で宣言する必要があります。詳細は、A-6 ページの「[エンティティ Bean の要素](#)」を参照してください。また、これらのフィールドは、意図したデータベース表とそれぞれの列にマッピングする必要があります。詳細は、A-28 ページの「[永続フィールド](#)」を参照してください。

次の例は、Oracle 固有のディプロイメント・ディスクリプタのうち、顧客の例（単一の主キー・フィールドを使用）にある主キーとコンテナ管理の永続フィールドを、永続性プロパティにより各フィールドの値が格納されるデータベース表および列にマッピングする部分を示しています。特に、コンテナ・マネージャでは、永続フィールドが SCOTT のスキーマの顧客表に格納されます。永続フィールドは、次のようにマッピングされます。

永続フィールド	表列
custid（主キー）	customers 表の cust_id 列
name	customers 表の cust_name 列
addr	customers 表の cust_addr 列

```
<psi-ri>
    <schema>SCOTT</schema>
    <table>customers</table>
    <attr-mapping>
        <field-name>custid</field-name>
        <column-name>cust_id</column-name>
    </attr-mapping>
    <attr-mapping>
        <field-name>name</field-name>
        <column-name>cust_name</column-name>
    </attr-mapping>
    <attr-mapping>
        <field-name>addr</field-name>
```

```

        <column-name>cust_addr</column-name>
      </attr-mapping>
    </psi-ri>

```

EJB の参照と JDBC の DataSources へのアクセス

EJB の参照と JDBC の DataSources には、Bean の環境を介してアクセスできます。

- [EJB の参照](#)
- [JDBC の DataSources](#)

EJB の参照

エンティティ Bean では、ディプロイメント・ディスクリプタ内でターゲット Bean 用の環境参照を作成できます。EJB 環境参照の URL の構文は、次のとおりです。

```
"java:comp/env/ejb/"<ejb-ref-name>
```

"java:comp/env/ejb" 接頭辞は、JNDI に対して、ディプロイメント・ディスクリプタに定義されている EJB 参照内で EJB 参照の位置を特定するように指示するサブコンテキストです。<ejb-ref-name> は、ディプロイメント・ディスクリプタに定義されている EJB 参照の論理環境名です。次の例は、Bean が配置環境内で他の Bean の参照を検索する方法を示しています。

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

```
CustomerHome ch = (CustomerHome)ic.lookup ("java:comp/env/ejb/PurchaseOrder");
```

EJB 環境参照の詳細は、A-9 ページの「[他の Enterprise JavaBeans の環境参照](#)」を参照してください。

JDBC の DataSources

エンティティ Bean には、JNDI 内でバインドされている JDBC DataSources の環境参照を作成するオプションがあります。これらは、ディプロイメント・ディスクリプタ内で Bean に対して宣言されています。serviceURL および objectName が定義されている場合、それぞれに次の構文を持つ EJB 環境参照の URL が含まれます。

```
"java:comp/env/jdbc/"<resource-ref-name>
```

"java:comp/env/jdbc" 接頭辞は、JNDI に対して、ディプロイメント・ディスクリプタに定義されている <resource-ref> 要素内で JDBC の DataSource をローケーティングするように指示するサブコンテキストです。ディプロイメント・ディスクリプタでは、実際には論理環境名が定義されています。

次の例は、発注用の EJB ディプロイメント・ディスクリプタでの、JDBC DataSource の定義を示しています。

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

注意： このリリースでは、<res-auth> 用にサポートされるのは Application オプションのみです。

次の例は、発注の例での JNDI コンテキストの初期化を示しています。

```
#import javax.sql.*

String dbURL = "java:comp/env/jdbc/EmployeeAppDB";

DataSource dbRes = (DataSource)ic.lookup (dbURL);
Connection conn = dbRes.getConnection;
```

JDBC DataSource 変数の詳細は、A-12 ページの「[リソース・マネージャの接続ファクトリ参照への環境参照](#)」を参照してください。

JNDI 接続とセッション IIOP サービス

この章では、クライアントが Oracle8i Server セッションに接続する方法、およびクライアントがサーバーに対して認証を実行する方式を詳しく説明します。この章で使用するクライアントという用語には、ネットワーク接続されているパーソナル・コンピュータやワークステーション上で稼働しているクライアント・アプリケーションとアプレット、および他の分散サーバー・オブジェクトをコールし、それらのオブジェクトに対してクライアントとして動作する EJB および CORBA サーバー・オブジェクトなどの分散オブジェクトが含まれます。

この章では、認証の他に、データベース内のオブジェクトに対するアクセス制御という意味のセキュリティも説明します。データ・サーバー内の公開されているオブジェクトには一連のパーミッションがあり、誰がオブジェクトにアクセスして変更できるかを設定できます。また、データ・サーバーにロードされているクラスは特定のスキーマ内にロードされ、そのクラスの配置者が、誰がクラスを使用できるかを制御できます。

この章では、次のトピックを説明します。

- JNDI 接続の基本事項
- ネームスペース
- データベース・オブジェクトに対する実行権限
- URL 構文
- JNDI を使用したバインド済みオブジェクトへのアクセス
- セッション IIOP サービス
- Oracle8i JVM のバージョン・ナンバーの検索
- 非 IIOP プレゼンテーションからのセッション中の EJB オブジェクトの活性化
- アプレットからの EJB オブジェクトの起動

JNDI 接続の基本事項

第2章のクライアントの例では、単一 URL 指定を使用して、Oracle にし、データベース・サーバー・セッションを開始し、オブジェクトを活性化する方法を示しました。これらの操作は、次のステップで実行されました。

```
1. Hashtable env = new Hashtable();
2. env.put(javax.naming.Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
3. env.put(javax.naming.Context.SECURITY_PRINCIPAL, username);
4. env.put(javax.naming.Context.SECURITY_CREDENTIALS, password);
5. env.put(javax.naming.Context.SECURITY_AUTHENTICATION,
ServiceCtx.NON_SSL_LOGIN);
6. Context ic = new InitialContext(env);
7. HelloHome hello_home =
    (HelloHome) ic.lookup("sess_iiop://localhost:2481:ORCL/test/myHello");
    myHello hello = hello_home.create ();
8. System.out.println(hello.helloWorld());
```

この例には、4つの基本操作があります。

- 1～5行目で、JNDI 初期コンテキストの環境をセットアップします。
- 6行目では、JNDI 初期コンテキストを作成します。
- 7行目では、公開されているオブジェクトを検索します。（URL 構文の詳細は、5-5 ページの「[URL 構文](#)」を参照してください。）
- 8行目では、オブジェクト上でメソッドを起動します。

クライアントが JNDI lookup メソッドを介してオブジェクトを検索するときに、クライアントとサーバーの両方で次のロジックが自動的に実行されます。

- ローカル・ホスト・データベースの ORCL インスタンスへのセッション IIOP 接続が確立されます。
- サーバーによりデータベース・セッションが確立されます。
- クライアントに対する認証が NON_SSL_LOGIN プロトコルを使用して行われます。ユーザー名とパスワードは初期コンテキスト環境で指定されたものが使用されます。
- 公開されているオブジェクト /test/myHello がセッション・ネームスペース内で検索され、その参照がクライアントに戻されます。

戻された参照に対して、クライアントにより helloWorld() などのメソッドが起動されると、サーバーによりそのオブジェクトがサーバー上で活性化されます。

ネームスペース

データベース内のネームスペースは、通常のファイル・システムにきわめて類似しています。セッション・シェル・ツールを使用して、公開するネームスペース内のオブジェクトを調べたり、操作できます。セッション・シェルの詳細は、『Oracle8i Java Tools リファレンス』の「sess_sh」ツールを参照してください。

ルート・ディレクトリがあり、これはスラッシュ（「/」）で示されます。ルート・ディレクトリは、bin、etc および test という他の3つのディレクトリを含むように作成されています。/test ディレクトリは、プログラム例中のほとんどのオブジェクトが公開されている場所です。ルートの下に新しいディレクトリを作成し、個別のプロジェクトに対するオブジェクトを保持するようにできますが、ルートの下に新しいディレクトリを作成するにはデータベース・ユーザーを SYS としてアクセスする必要があります。

ディレクトリのネストの深さには実質的な制限はありません。

注意： 公開するネームスペース内の初期値は、Oracle8i JVM 製品をインストールするときに設定されます。

/etc ディレクトリには、ORB で使用するオブジェクトが置かれます。/etc ディレクトリ内のオブジェクトを削除しないでください。/etc 内のオブジェクトは次のとおりです。

deployejb execute loadjava login transactionFactory

ネームスペース内のエントリは、次のクラスのインスタンスであるオブジェクトで表されます。

- oracle.aurora.AuroraServices.PublishingContext — 他のオブジェクト（ディレクトリ）を含むことができるクラスを表します。
- oracle.aurora.AuroraServices.PublishedObject — ツリーのリーフに使用されるオブジェクト名自体です。

これらのクラスは、製品 CD に収録されている JavaDoc に記載されています。

オブジェクトの公開されている名前は、データベース内の表に格納されます。公開されている各オブジェクトにも、関連するパーミッションのセットがあります。それぞれのクラスまたはリソース・ファイルには、次のパーミッションの組合せを持たせることができます。

読み込み 読み込み権限の保有者は、クラスまたはクラスの属性（名前、ヘルパー・クラス、所有者など）の一覧を表示できます。

書き込み コンテキストの書き込み権限の保有者は、新しいオブジェクト名をコンテキストにバインドできます。オブジェクト（ツリーのリーフ・ノード）の場合は、書き込み権限の保有者はこの名前別のオブジェクトを再公開できます。

実行 コンテキストまたは公開されているオブジェクト名により表されるオブジェクトを解決して活性化するには、実行権限が必要です。

オブジェクト権限を表示および変更するには、セッション・シェル・ツールの `chmod` コマンドを使用します。

データベース・オブジェクトに対する実行権限

Oracle8i では、認証およびプライバシーの他に、CORBA オブジェクトおよび EJB オブジェクトを構成するクラスへのアクセス制御もサポートされています。データベースに格納されているオブジェクトの Java クラスに対する実行権限を付与されているユーザーまたはロールのみが、オブジェクトを活性化し、そのオブジェクト上でメソッドを起動できます。

Java クラスに対する実行権限の制御には、次のツールを使用できます。

- ロード時は、`-grant` 引数を付けて `loadjava` を実行できます。`loadjava` およびデータベース内の Java クラスに対する実行権限の詳細は、『Oracle8i Java 開発者ガイド』を参照してください。
- SQL 文を使用できます。データベース内にロードされている Java クラスに対する実行のパーミッションを付与するには、`GRANT EXECUTE DDL` 文を使用します。たとえば、SCOTT がクラス `Hello` をすでにロードしている場合は、SCOTT（または SYS）は次の SQL 文を発行して、そのクラスに対する実行権限を別のユーザー（たとえば OTTO）に付与できます。

```
SQL> GRANT EXECUTE ON "Hello" TO OTTO;
```

ロードされている Java クラスからユーザーの実行権限を削除するには、SQL 文 `REVOKE EXECUTE` を使用します。

- 公開時は、公開されているオブジェクトは特定のスキーマに制限されません。インスタンス内のすべてのユーザーが使用可能です。公開されているオブジェクトにはパーミッションがあり、これは基礎となるクラスのパーミッションとは異なる場合があります。たとえば、ユーザー SCOTT が公開されているオブジェクト名に対する実行のパーミッションを持っていたとしても、公開されているオブジェクトが表しているクラスの実行のパーミッションは持っていない場合、SCOTT はオブジェクトを活性化できません。

公開されているオブジェクトに対するパーミッションは、次の 2 つの方法で制御できます。

1. `publish` ツールで `-grant` オプションを指定できます。
2. セッション・シェル内で `chmod` コマンドおよび `chown` コマンドを実行できます。`chown` コマンドを実行するには、ユーザー SYS としてセッション・シェルに接続する必要があります。

セッション・シェルで `ls -l` コマンドを実行すると、パーミッション（EXECUTE、READ および WRITE）および公開されているオブジェクトの所有者が表示されます。

認証なしにクライアントがアクセスできる組込みサーバー・オブジェクトが3つがあります。

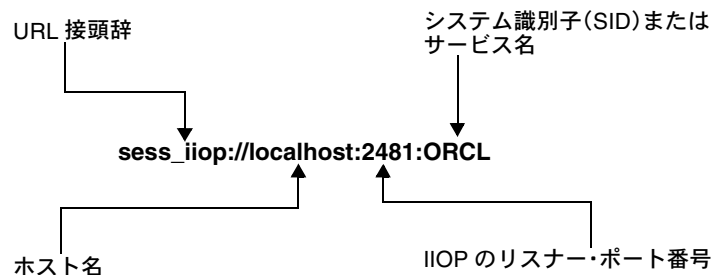
- ネーム・サービス
- InitialReferences オブジェクト（ブート・サービス）
- Login オブジェクト

これらのオブジェクトは、認証がなくても `serviceCtx.lookup()` を使用して活性化できます。Login オブジェクトに明示的にアクセスする例は、6-11 ページの「[Oracle8i JVM セッションへのログインとログアウト](#)」を参照してください。

URL 構文

Oracle8i では、サービスとセッションのアクセスに Universal Resource Locator (URL) 構文を使用します。URL では、JNDI リクエストを使用して、サービスとセッションの起動や、データベース・インスタンス内で公開されているコンポーネントへのアクセスを行います。サービス URL の例を図 5-1 に示します。

図 5-1 サービス URL



サービス URL は、次の4つの部分から構成されます。

1. URL 接頭辞。コロンと2つのスラッシュが後ろに付きます。セッション IIOP リクエストの場合、`sess_iiop://` となります。
2. システム名（ホスト名）。たとえば、`myPC-1` などです。`localhost` やホストの IP アドレスを使用できます。
3. IIOP サービスのリスナー・ポート番号。デフォルトは 2481 です。
4. システム識別子（SID）またはサービス名。たとえば、`ORCL` や `mySID.myDomain` などです。
 - SID — システム識別子は、データベース初期化ファイルで `db_name` として定義されています。システム識別子により、接続しているデータベース・インスタンスを

識別できます。サービス URL に SID を追加すると、リスナーにより受信されたリクエストがデータベース・インスタンスの複数のディスパッチャに対してロード・バランス処理されます。

- サービス名 — サービス名は、データベース初期化ファイルに定義されている `service_name` パラメータまたは `db_name.db_domain` パラメータの値と同じになります。サービス URL 内でサービス名を使用すると、リスナーにより受信されたリクエストが複数のデータベース・インスタンス（つまり、リスナーに登録されているすべてのデータベース・インスタンス）に対してロード・バランス処理されます。このオプションは、パラレル・サーバーを使用している場合に適しています。

注意： URL でサービス名を使用する場合は、どのツールでも `-useServiceName` フラグを指定する必要があります。このフラグを指定しない場合、ツールは最後の文字列が SID であると想定します。

ホスト名、ポートおよび SID またはサービス名の間は、必ずコロンで区切ってください。

注意： リスナー・ポートのかわりにディスパッチャ・ポートを指定し、さらに SID を指定すると、サーバーで `ObjectNotFound` 例外が発生します。ディスパッチャ・ポートに直接接続するアプリケーションはあまりスケールしないので、Oracle ではディスパッチャとの直接接続はお勧めしません。

URL のコンポーネントとクラス

Oracle に接続し、JNDI を使用して公開されているオブジェクトを検索する場合、サービス（サービス名、ホスト、ポートおよび SID）と、検索、活性化する公開オブジェクトの名前を含む URL を使用します。たとえば、完全な URL は次のようになります。

```
sess_iiop://localhost:2481:ORCL/:default/projectAurora/Plans816/getPlans
```

`sess_iiop://localhost:2481:ORCL` にはサービス名を指定します。`:default` はデフォルトのセッションを示します（セッションがすでに確立されている場合）。`/projectAurora/Plans816` にはネームスペース内のディレクトリ・パスを指定します。また、`getPlans` は検索する公開オブジェクトの名前です。

注意： 接続に対してセッションがまだ確立されていない場合は、セッション名を指定しないでください。つまり、最初の検索時には活性化されているセッションがまだないので、セッション名としての `:default` は無意味です。また、`:default` は暗黙的なため、URL は、このセッション名なしでも使用できます。

URL の各コンポーネントは Java クラスを表します。たとえば、サービス名は `ServiceCtx` クラスのインスタンスで表され、セッションは `SessionCtx` インスタンスで表されます。URL 内のサービス名とセッション名の詳細は、5-7 ページ以降の「[JNDI を使用したバインド済みオブジェクトへのアクセス](#)」および「[セッション IIOP サービス](#)」を参照してください。

JNDI 名に関する CosNaming の制限事項

公開されているオブジェクトの、JNDI にバインドされている名前には、JNDI 構文規則を使用する必要があります。Oracle8i JVM JNDI で使用される基礎となるネーミング・サービスは `CosNaming` です。したがって、ある名前にドット (.) が含まれていると、次のように通常の `CosNaming` 規則とは異なる動作が発生します。

- ドットの前の部分文字列は、`CosNaming NameComponent ID` として処理されます。
- ドットの後の部分文字列は、`CosNaming NameComponent` の種類として処理されます。
- ID と種類の両方が連結されて、フル JNDI 名となります。

通常、`CosNaming` オブジェクトの取出し時には、ID と種類を別個のエンティティとして指定します。Oracle8i JVM の実装では、ID と種類の両方が連結されます。したがって、アプリケーションでオブジェクトを取り出すには、ドットがセパレータではなく JNDI 名の一部として含まれているフルネームを使用します。

JNDI を使用したバインド済みオブジェクトへのアクセス

クライアントは、Java Naming and Directory Interface (JNDI) インタフェースを使用して、Oracle8i JVM ネームスペース内の公開されているオブジェクトを検索します。JNDI は Sun Microsystems が提供するインタフェースで、これを使用すると Java アプリケーション開発者はネーム・サービスおよびディレクトリ・サービスにアクセスできます。この項では、公開されているオブジェクトを検索し、活性化するために必要な JNDI API を説明します。JNDI の完全なドキュメントを入手するには、<http://java.sun.com/products/jndi/index.html> にアクセスしてください。

注意： JNDI を使用せずにセッション・ネームスペースにアクセスすることもできます。JNDI のかわりに、`CosNaming` のメソッドを使用できます。

5-5 ページの「[URL 構文](#)」で説明したように、Oracle8i JVM ネームスペース内にバインドされている名前へのアクセスを行う JNDI URL は、次の 2 つの構成要素からなる複合名である必要があります。

- サービス名 — サービス名は、Oracle8i JVM で適切なネームスペースへのハンドルを取り出すために使用されます。

ネットワークには、複数のネームスペースが存在します。サービスでは、JNDI でバインドされているオブジェクトを取り出すネームスペースが指定されます。サービス名には、次のいずれか 1 つを使用できます。

サービス名	説明
<code>sess_iiop://</code> <code><hostname>:<port>:<SID></code>	ネームスペースが置かれているホスト、ポートおよび SID を指定します。このサービス名のみを指定してセッション名を指定しなければ、 <code>ServiceCtx</code> オブジェクトが戻されます。セッション <code>IIOP</code> サービスは、 <code>IIOP</code> アプリケーションで使用されるメイン・サービスです。このサービスにより、様々なデータベース・ホスト上の JNDI ネームスペースにバインドされているオブジェクトとオブジェクト参照が取り出されます。詳細は、5-12 ページの「 セッション IIOP サービス 」を参照してください。
<code>jdbc_access:</code>	オブジェクトの <code>lookup</code> に <code>JDBC</code> 接続を使用することを示します。主として、ネームスペースから <code>JTA</code> の <code>UserTransaction</code> および <code>DataSource</code> オブジェクトを取り出すために使用します。
<code>java:</code>	バインドされている名前が、実際にはディプロイメント・ディスクリプタ内で指定された <code>EJB</code> 環境変数であることを示すのに使用されます。

- セッション名 — 指定したネームスペース内のオブジェクトの、実際に JNDI にバインドされている名前です。この構文は、UNIX ファイル・システムの構文に似ています。セッション名は、`SessionCtx` オブジェクトで表すことができます。

サービスとセッションのコンテキストを利用して、データベース内に異なる複数のセッションをオープンしたり、単一セッション中のオブジェクトに複数のクライアントからアクセスするなど、いくつかの高度な手法を実行できます。この手法の詳細は、5-12 ページの「[セッション IIOP サービス](#)」を参照してください。ただし、単純な JNDI 検索を行う場合は、5-5 ページの「[URL 構文](#)」に示した URL 構文を使用するべきです。

JNDI サポート・クラスのインポート

クライアントまたはサーバーのオブジェクトの実装で JNDI を使用する場合は、次のインポート文を各ソース・ファイルに組み込んでください。

```
import javax.naming.Context;    // the JNDI Context interface
import javax.naming.InitialContext;
import oracle.aurora.jndi.sess_iiop.ServiceCtx; // JNDI property constants
import java.util.Hashtable;     // hashtable for the initial context environment
```

JNDI InitialContext の取得

Context は、InitialContext の取得に使用される javax.naming パッケージ内のインタフェースです。Oracle8i の EJB および CORBA クライアントでは、いずれも JNDI lookup() に InitialContext が使用されます。JNDI lookup() を実行する前に、認証情報などの環境変数を Context に対し設定します。ハッシュ表または環境のプロパティ・リストを使用できます。この情報は、lookup() の実行時にネーミング・サービスに対して使用されます。このマニュアルの例では、次のように常に Java の Hashtable クラスを使用します。

```
Hashtable environment = new Hashtable();
```

次に、ハッシュ表内にプロパティを設定します。クライアント側とサーバー側のどちらで操作するかに関係なく、常に Context の URL_PKG_PREFIXES プロパティを設定する必要があります。残りのプロパティは、主としてユーザー認証に使用されます。

- javax.naming.Context.URL_PKG_PREFIXES
- javax.naming.Context.SECURITY_PRINCIPAL
- javax.naming.Context.SECURITY_CREDENTIALS
- javax.naming.Context.SECURITY_ROLE
- javax.naming.Context.SECURITY_AUTHENTICATION
- USE_SERVICE_NAME

URL_PKG_PREFIXES

Context.URL_PKG_PREFIXES には、URL コンテキスト・ファクトリ内にロードするとき使用するパッケージ接頭辞のリストを指定するための環境プロパティが保持されます。このプロパティの値は、URL コンテキスト・ファクトリを作成するファクトリ・クラスのクラス名に対するパッケージ接頭辞をコロンで区切って並べたリストであることが必要です。

現行の実装では、常にこのプロパティを Context 環境で指定し、文字列「oracle.aurora.jndi」に設定する必要があります。

SECURITY_PRINCIPAL

Context.SECURITY_PRINCIPAL には、データベースのユーザー名が保持されます。

SECURITY_CREDENTIALS

Context.SECURITY_CREDENTIAL には、クリアテキスト・パスワードが保持されます。これは、SECURITY_PRINCIPAL（データベース・ユーザー）に対する Oracle データベースのパスワードです。次の「SECURITY_AUTHENTICATION」で述べている 3 つの認証方式では、パスワードは暗号化されてサーバーに送信されます。

SECURITY_ROLE

Context.SECURITY_ROLE には、ユーザーが接続するときに使用する Oracle8i データベース・ロールが保持されます。たとえば、「CLERK」や「MANAGER」です。

SECURITY_AUTHENTICATION

Context.SECURITY_AUTHENTICATION には、使用する認証のタイプを指定する環境プロパティの名前が保持されます。このプロパティの値により、Oracle8i でサポートされている認証のタイプが決まります。このプロパティには、次の 4 つの値を指定できます。これらの値は、ServiceCtx クラスで定義されています。

- ServiceCtx.NON_SSL_LOGIN - クライアントは、(セキュア・ソケット・レイヤー接続ではなく) 標準の TCP/IP 接続でログイン・プロトコルを使用して、サーバーに対してユーザー名とパスワードによる認証を行います。パスワードは、クライアントからサーバーへの送信時にログイン・プロトコルにより暗号化されます。サーバーは、クライアントに資格証明を提供して自分自身を認証します。このプロトコルの詳細は、6-9 ページの「[ユーザー名とパスワードを使用したクライアント側の認証](#)」を参照してください。
- ServiceCtx.SSL_CREDENTIAL - クライアントは、セキュア・ソケット・レイヤー (SSL) 接続で暗号化されたユーザー名とパスワードを提供して、サーバーに対して自分自身を認証します。サーバーは、クライアントに対して自分自身を認証しません。
- SSL_LOGIN - クライアントは、SSL 接続でログイン・プロトコルを使用して、サーバーに対してユーザー名とパスワードによる自分自身を認証します。サーバーは、クライアントに資格証明を提供して自分自身を認証します。
- SSL_CLIENT_AUTH - クライアントとサーバーの両方が、SSL 接続でそれぞれの証明書を提供して相互に自分自身を認証します。

注意： SSL 接続を使用するには、SSL ポートが構成されているリスナーにアクセスする必要があります。また、リスナーは SSL を使用できるデータベース IIOP ポートにリクエストをリダイレクトする必要があります。さらに、アプリケーションをコンパイルし作成するときに、次の JAR ファイルを組み込む必要があります。

- クライアントで JDK 1.1 を使用している場合は、jssl-1_1.jar と javax-ssl-1_1.jar をインポートします。
 - クライアントで Java 2 を使用している場合は、jssl-1_2.jar と javax-ssl-1_2.jar をインポートします。
-

USE_SERVICE_NAME

URL に SID ではなくサービス名を使用している場合は、このプロパティを TRUE に設定します。それ以外の場合は、URL の最後の文字列で SID を指定する必要があります。変数 env 内の Hashtable に、URL 内で SID ではなくサービス名を使用することを指定するには次のようにします。

```
Hashtable env = new Hashtable();
env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put(Context.SECURITY_PRINCIPAL, "scott");
env.put(Context.SECURITY_CREDENTIALS, "tiger");
env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
env.put("USE_SERVICE_NAME", "true");
Context ic = new InitialContext(env);
```

デフォルトは FALSE です。

lookup で指定する URL には、SID ではなくサービス名を含める必要があります。次の URL には、サービス名 orasun12 が含まれています。

```
myHello hello =
    (myHello) ic.lookup("sess_iiop://localhost:2481:orasun12/test/myHello");
```

JNDI InitialContext メソッド

InitialContext は、Context インタフェースを実装する javax.naming パッケージ内のクラスです。すべてのネーミング操作はコンテキストに関するものです。InitialContext は、Context インタフェースを実装しており、名前解決の出発点となるものです。

コンストラクタ

新しい初期コンテキストを作成するには、次のコンストラクタを使用します。

```
public InitialContext(Hashtable environment)
```

「[JNDI InitialContext の取得](#)」で前述したように環境情報を含む入力パラメータとして、Hashtable が必要です。次のコードで、標準的なクライアント用の環境を設定し、新しい初期コンテキストを作成します。

```
Hashtable env = new Hashtable();
env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put(Context.SECURITY_PRINCIPAL, "scott");
env.put(Context.SECURITY_CREDENTIALS, "tiger");
env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
Context ic = new InitialContext(env);
```

lookup

CORBA アプリケーションや EJB アプリケーションの開発者が使用する最も一般的なクラスのメソッドは次のものです。

```
public Object lookup(String URL)
```

lookup () を使用して、オブジェクト・インスタンスを取り出したり、新規サービス・コンテキストを作成します。

- オブジェクト・インスタンスを取り出すには、サービス名と、JNDI にバインドされている名前（とセッション名）をつなげた URL を指定します。戻される結果は、予期されるオブジェクト型にキャストする必要があります。たとえば、Hello インタフェースを取り出すには、次の操作を行います。

```
HelloHome hello_home =  
    (HelloHome) ic.lookup("sess_iiop://localhost:2481:ORCL/test/myHello");  
myHello hello = hello_home.create ();
```

サービス名は「sess_iiop://localhost:2481:ORCL」、Hello のホーム・インタフェースの JNDI にバインドされる名前は「/test/myHello」です。

- 特定のネームスペースへのハンドルを取り出すには、必要なサービス・コンテキストを指定します。戻される結果は、新しいサービス・コンテキストを作成する際に ServiceCtx にキャストする必要があります。たとえば、initContext が JNDI 初期コンテキストであれば、次の文により新しいサービス・コンテキストが作成されます。

```
ServiceCtx service =  
    (ServiceCtx) initContext.lookup("sess_iiop://localhost:2481:ORCL");
```

EJB または CORBA アプリケーションでの JNDI lookup メソッドの使用例については、5-17 ページの「[セッション管理の使用例](#)」を参照してください。

セッション IIOP サービス

すべてのクライアント / サーバー・ネットワーク通信では、両方のエンティティ間で受け入れられるプロトコルを介して要求がルーティングされます。Oracle8i データベースへのほとんどのネットワーク通信は、2 タスク共通 (TTC) レイヤーを介してルーティングされます。これは、データベースの SQL サービスに関する受信 Net8 要求を処理するサービスです。ただし、データベースへの Java の追加により、Oracle8i JVM ではクライアントがデータベース・セッションを意識した IIOP トランスポートを介してサーバーと通信するように要求しています。これは、セッション IIOP サービスを介して実行されます。

セッション IIOP サービスは、CORBA および EJB アプリケーションなど、IIOP アプリケーションに対する要求を容易にするために使用されます。この後の項では、1 つ以上のデータベース・セッションでアプリケーションを管理する方法について説明します。

- [セッション IIOP サービスの概要](#)
- [セッション管理](#)
- [サービス・コンテキスト・クラス](#)
- [セッション・コンテキスト・クラス](#)
- [セッション管理の使用例](#)
- [セッション・タイムアウトの設定](#)

セッション IIOP サービスの概要

『Oracle8i Java 開発者ガイド』に記載されているように、EJB はデータベースにロードされるため、クライアント・アプリケーションはデータベース・セッションのコンテキスト内で EJB を起動する必要があります。Bean はセッション内で活性化されるため、各クライアントは別のセッションへのハンドルが与えられない限り、そちらのセッションでアクティブな Bean インスタンスを参照できません。また、オブジェクトは、既存のセッションまたは別のセッション内で活性化できます。

セッション IIOP サービスのセッション・コンポーネント・タグ TAG_SESSION_IIOp は、IIOP プロファイル SessionIIOP 内に存在します。この Oracle セッション IIOP コンポーネント・タグの値は 0x4f524100 で、オブジェクトが活性化されたセッションを一意に識別する情報が含まれます。クライアント ORB ランタイムでは、この情報を使用して特定のセッション内のオブジェクトにリクエストが送信されます。

Oracle8i セッション IIOP サービスは、セッション ID 情報を含むという意味から標準 IIOP プロトコルを強化するサービスといえますが、ワイヤー上のデータ送信プロトコルとしては標準 IIOP との違いはありません。

クライアントの要件

クライアントでは、ORB の実装でセッション IIOP をサポートし、同じプログラム内から異なるセッションのオブジェクトへの同時アクセスや、同じセッションに対する切断および再接続を行えることが必要です。Oracle8i とともに出荷されるバージョンの Visigenic ORB は、セッション IIOP をサポートするように拡張されています。

セッション・ルーティング

クライアントがデータベースに対して IIOP 接続を実行するときに、Oracle8i では、リクエストを処理する新しいセッションを開始するか、またはリクエストを既存のセッションにルートするかを決定します。クライアントが (InitialContext.lookup() メソッドを使用して) 接続リクエストを新たに送信したときに、その接続に対してアクティブなセッションがない場合には、新しいセッションが自動的に開始されます。クライアント用にセッショ

ンがすでに活性化されている場合は、セッション識別子とそのオブジェクトのオブジェクト・キー中にコード化されます。この情報により、セッション IIOP サービスはリクエストを適切なセッションにルートできます。また、このセッション識別子を使用して、単一クライアントに複数セッションへのアクセスを許可することもできます。詳細は、5-17 ページの「[セッション管理の使用例](#)」を参照してください。

Oracle8i JVM ツール

Oracle8i JVM ツールを使用する場合、特に EJB アプリケーションおよび CORBA アプリケーションを開発する場合は、TTC と IIOP の 2 つのネットワーク・サービスのプロトコル・タイプを区別することが非常に重要です。

図 5-2 TTC サービスおよび IIOP サービス

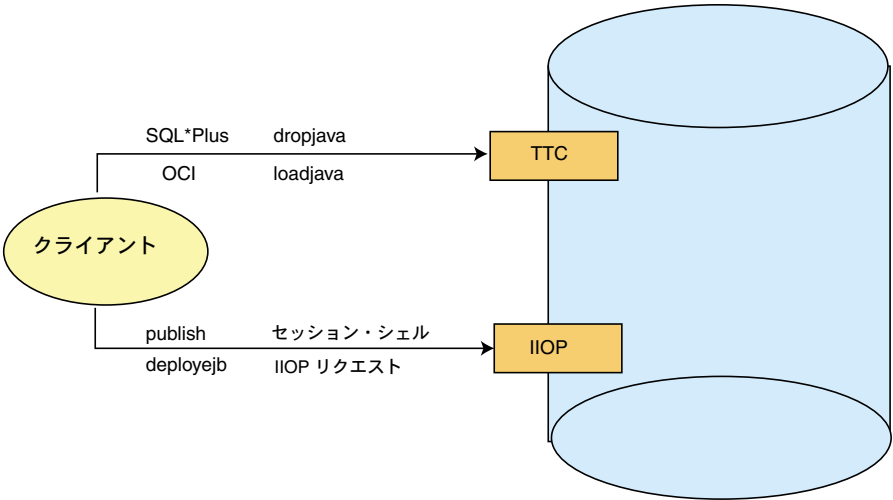


図 5-2 は、どのツールおよびリクエストで TTC が使用され、どのツールで IIOP データベース・ポートが使用されるかを示しています。TTC に対するデフォルトのポート番号は 1521 で、IIOP に対するデフォルトのポート番号は 2481 です。

- publish、deployejb およびセッション・シェルなどのツールは、IIOP オブジェクトにアクセスするために IIOP ポートを使用して接続する必要があります。また、EJB クライアントと CORBA クライアントでも、Oracle にリクエストを送信するときに IIOP ポートを使用する必要があります。
- loadjava や dropjava などのツールでは、接続に TTC ポートを使用します。

セッション管理

最も単純な場合、クライアント（またはクライアントとして動作するサーバー・オブジェクト）では、サーバー・オブジェクトの検索を実行するときに、新規サーバー・セッションが暗黙的に開始されます。Oracle8i では、セッション起動を明示的に制御することもできます。セッション IIOP サービス接続とデータベース内のセッションを制御できるように、Oracle 固有の 2 つのクラスが用意されています。

- **サービス・コンテキスト・クラス (ServiceCtx)** — データベースへのセッション IIOP サービス接続を制御します。そのデータベースへの URL を指定して、サービス・コンテキストを作成できます。このサービス・コンテキストから、データベース内で 1 つ以上の名前付きセッションをオープンできます。
- **セッション・コンテキスト・クラス** — サービス・コンテキストから作成された名前付きデータベース・セッションを制御します。作成後は、名前付きセッション・コンテキスト・オブジェクトを使用して、セッション内で CORBA または EJB のオブジェクトを活性化できます。

サービス・コンテキスト・クラス

データベースへのセッション IIOP サービス接続を制御します。そのデータベースへの URL を指定して、サービス・コンテキストを作成できます。このサービス・コンテキストから、データベース内で 1 つ以上の名前付きセッションをオープンできます。この Oracle 固有のクラスは、JNDI Context クラスを拡張 (extends) しています。

変数

ServiceCtx クラスには、環境プロパティやその他の変数の定義に使用できる多数の final public static 変数が定義されています。表 5-1 を参照してください。

表 5-1 ServiceCtx Public 変数

文字列の名前	値
NON_SSL_CREDENTIAL	"Credential"
NON_SSL_LOGIN	"Login"
SSL_CREDENTIAL	"SecureCredential"
SSL_LOGIN	"SecureLogin"
SSL_CLIENT_AUTH	"SslClientAuth"
SSL_30	"30"
SSL_20	"20"
SSL_30_WITH_20_HELLO	"30_WITH_20_HELLO"

表 5-1 ServiceCtx Public 変数 (続き)

整数値の名前	整数値のコンストラクタ
SESS_IIOP	new Integer(2)
IIOP	new Integer(1)

メソッド

このクラスのパブリック・メソッドのうち、CORBA アプリケーションや EJB アプリケーションの開発者が使用できるメソッドは次のとおりです。

`public Context createSubcontext(String name)`

このメソッドは、パラメータとして Java String を取り、データベースにおけるセッションを表す JNDI Context オブジェクトを戻します。このメソッドにより、新しい名前付きセッションが作成されます。パラメータは作成されるセッションの名前であり、コロン (:) で始める必要があります。

戻される結果は、SessionCtx オブジェクトにキャストする必要があります。

javax.naming.NamingException 例外が発生する可能性があります。

`public Context createSubcontext(Name name)`

(String をパラメータとする各メソッドには、Name をパラメータとした対応するメソッドがあります。機能は同じです。)

`public static org.omg.CORBA.ORB init(String username,
String password,
String role,
boolean ssl,
java.util.Properties props)`

検索を実行するときに、作成された ORB にアクセスします。SSL 認証の場合、ssl パラメータを true に設定します。JNDI を使用せずにサーバー・オブジェクトにアクセスするクライアントは、このメソッドを使用する必要があります。

使用例は、『Oracle8i CORBA 開発者ガイド』の第 4 章「JNDI 接続とセッション IIOP サービス」の「セッション IIOP サービス」を参照してください。

`public Object lookup(String string)`

lookup() は、サービス・コンテキストと関連するデータベース・インスタンス内の公開されているオブジェクトを検索し、活性化されたオブジェクトのインスタンスを戻します。または、javax.naming.NamingException 例外が発生する可能性があります。

セッション・コンテキスト・クラス

サービス・コンテキストから作成された名前付きデータベース・セッションを制御します。作成後は、名前付きセッション・コンテキスト・オブジェクトを使用して、セッション内で CORBA または EJB のオブジェクトを活性化できます。セッション・コンテキストはセッションを表しており、これにはセッションに対するクライアント認証や、オブジェクトを活性化するなどのセッション操作を実行するためのメソッドが含まれています。このクラスは、JNDI Context クラスを拡張 (extends) しています。

メソッド

クライアントは、次のセッション・コンテキスト・メソッドを使用します。

```
public synchronized boolean login()
```

login() は、ユーザー名、パスワードおよびロールという、InitialContext コンストラクタで渡された初期コンテキスト環境プロパティを使用してクライアントの認証を実行します。

```
public synchronized boolean login(String username,  
                                   String password,  
                                   String role)
```

login() は、パラメータとして指定されたユーザー名、パスワードおよびオプションのデータベース・ロールを使用してクライアントの認証を実行します。

```
public Object activate(String name)
```

name という名前を持つ公開されているオブジェクトを検索し、活性化します。

セッション管理の使用例

次の項では、データベース・セッション管理の様々な使用例について説明します。

- **クライアントが単一セッションにアクセスする場合** — クライアントにより :default セッション内でオブジェクトが活性化され、アクセスされます。
- **セッションの終了** — セッションを明示的に終了するメソッドについて説明します。
- **クライアントが名前付きセッションを起動する場合** — クライアントにより :default 以外のセッション内で1つ以上のオブジェクトが活性化され、アクセスされます。このセッションは、SessionCtx に対応する名前で識別されます。
- **2つのクライアントが同一セッションにアクセスする場合** — 複数のクライアントに目的のオブジェクトのハンドルおよび Login オブジェクトのハンドルを提供することにより、単一セッション内で活性化されたオブジェクトに複数のクライアントからアクセスできます。

- **同一セッション中の活性化** — クライアントとして動作するサーバー・オブジェクトにより、同じセッション内で別のオブジェクトが活性化されます。
- **JNDI コンテキストからのオブジェクトの検索** — JNDI 名の一部による検索および、バインドされているオブジェクトの活性化について説明します。

クライアントが単一セッションにアクセスする場合 一般に、URL、ホスト名およびポートを指定してクライアントから公開されているオブジェクトを検索する場合、オブジェクトは新しいセッションで活性化されます。たとえば、クライアントは次のようなコードを実行します。

```
Hashtable env = new Hashtable();
env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put (Context.SECURITY_PRINCIPAL, "scott");
env.put (Context.SECURITY_CREDENTIALS, "tiger");
env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
Context ic = new InitialContext (env);
SomeObjectHome myObj_home =
    (SomeObjectHome) ic.lookup ("sess_iiop://localhost:2481:ORCL/test/myObj");
SomeObject myObj = myObj_home.create ();
```

サーバー・オブジェクトから新しいセッションでオブジェクトを活性化するのも、アプリケーション・クライアントからセッションを開始するのと同じことです。上記のようにして lookup メソッドがサーバー・オブジェクト内で起動されると、元のセッションとは別個のセッションで第 2 のオブジェクト・インスタンスが活性化されます。

セッションの終了 通常、クライアントの終了時にセッションが終了します。ただし、セッションを明示的に終了したい場合には、次の 2 通りの方法があります。

endsession メソッドを使用したサーバー側からのセッション終了

サーバーでは、次のメソッドを実行してセッション終了を制御できます。

```
oracle.aurora.mts.session.Session.THIS_SESSION().endSession();
```

logout オブジェクトを使用したクライアント側からのセッション終了

クライアントからセッションを終了する場合は、LogoutServer オブジェクトの logout メソッドを実行できます。これは、「/etc/logout」として事前に公開されています。logout を実行できるのは、セッション所有者のみです。他の所有者が実行すると、NO_PERMISSION 例外が発生します。

LogoutServer オブジェクトは、「/etc/login」として事前に公開されている LoginServer オブジェクトに似ています。LoginServer オブジェクトを使用して、サーバーへの認証に使用する Login オブジェクトを生成します。JNDI lookup の際に暗黙的に Login オブジェクトを使用することもできます。

次の例では、クライアントで LoginServer オブジェクトを使用して認証し、LogoutServer オブジェクトを介してセッションを終了できることを示しています。

```

import oracle.aurora.AuroraServices.LoginServer;
import oracle.aurora.AuroraServices.LogoutServer;
...
// To log in using the LoginServer object
LoginServer loginServer = (LoginServer)ic.lookup(serviceURL + "/etc/login");
Login login = new Login(loginServer);
System.out.println("Logging in ..");
login.authenticate(user, password, null);
...
//To logout using the LogoutServer
LogoutServer logout = (LogoutServer)ic.lookup(serviceURL + "/etc/logout");
logout.logout();

```

クライアントが名前付きセッションを起動する場合 ServiceCtx および SessionCtx クラスに用意されている JNDI メソッドを利用して、データベース・インスタンス上で複数のセッションを明示的に作成できます。

次の lookup メソッドには、IIOP サービス URL である「sess_iiop://localhost:5521:ORCL」とデフォルトのセッション・コンテキストを定義する URL が含まれています。

```

SomeObjectHome myObj_home =
    (SomeObjectHome) ic.lookup("sess_iiop://localhost:2481:ORCL/test/myObj");
SomeObject myObj = myObj_home.create ();

```

この単純な例では、JNDI 初期コンテキストの lookup メソッドにより、暗黙的にセッションが開始されてクライアントが認証されます。このセッションは、名前「:default」で識別されるデフォルト・セッションとなります。すべてのセッションには名前が付いています。ただし、デフォルトの場合、すべての要求はこの単一セッションに送られるため、クライアントがセッション名を知る必要はありません。セッション名を指定しなければ、追加で活性化されるすべてのオブジェクトは、デフォルト・セッションで活性化されます。新しい JNDI 初期コンテキストを作成し、同じオブジェクトまたは新しいオブジェクトを検索する場合でも、オブジェクトは最初のオブジェクトと同じセッションでインスタンス化されます。

他のセッションでオブジェクトを活性化するには、名前付きセッションを作成する必要があります。サービス・コンテキストからセッション・コンテキストを作成すると、デフォルト・セッションのかわりまたは追加として他のセッションを作成できます。各セッションは名前付きセッションとなり、これを利用してデータベース内の異なるセッション中でオブジェクトを活性化できます。

1. サーバーに渡される環境プロパティ用に新しいハッシュ表をインスタンス化します。

```
Hashtable env = new Hashtable();  
env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
```

注意： URL_PKG_PREFIXES Context 変数のみに値を入力してください。その他の情報は、login.authenticate() メソッドのパラメータで提供します。

2. 新しい JNDI コンテキストを作成します。

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

3. 初期コンテキスト上で JNDI lookup メソッドを起動し、サービス URL を渡して、サービス・コンテキストを確立します。この例では、ホスト名、リスナー・ポートおよび SID からなるサービス URL を使用します。

注意： ホスト名、リスナー・ポートおよびデータベース SID からなるサービス URL のみを使用してください。必要なオブジェクトの JNDI 名を指定すると、デフォルト・セッションが作成されます。

```
ServiceCtx service =  
    (ServiceCtx) ic.lookup("sess_iiop://localhost:2481:ORCL");
```

4. サービス・コンテキスト・オブジェクト上で createSubcontext メソッドを起動して、セッションを作成します。createSubcontext メソッドのパラメータとしてセッション名を指定します。データベース内で新しいセッションが作成されます。

```
SessionCtx session = (SessionCtx) service.createSubcontext(":session1");
```

注意： 新しいセッションを作成するときは、そのセッションに名前を付ける必要があります。セッション名の先頭にはコロン (:) を付ける必要があります、スラッシュ (/) を含めることはできませんが、これ以外の制限はありません。

5. セッション・コンテキスト・オブジェクト上で login メソッドを起動し、データベースに対してクライアント・プログラムを認証します。

```
session.login("scott", "tiger", null);    // role is null
```

6. 名前付きセッション上で、バインドされている JNDI 名で識別されるオブジェクトを活性化します。

EJB の活性化により、Bean のホーム・インタフェースが戻されます。基本的に、ステップ 1～6 では、ホーム・インタフェースに対して JNDI lookup() を実行するのと同じことを実行します。単に、オブジェクトを活性化する特定の名前付きセッションを指定できます。

```
// Activate one Hello object in the session
HelloHome hello_home = (HelloHome)session.activate (objectName);
```

7. リモート・インタフェースを取得し、Hello Bean に対してメソッドを起動します。

```
Hello hello = hello_home.create ();
System.out.println (hello.helloWorld ());
```

例 5-1 名前付きセッションでのオブジェクトの活性化

次の例では、:session1 および :session2 という名前を持つ 2 つの名前付きセッションを作成します。各セッションでは、Hello オブジェクトを個別に検索します。クライアントは、各名前付きセッション上の Hello オブジェクトを両方とも起動します。

```
Hashtable env = new Hashtable ();
env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
Context ic = new InitialContext (env);

// Get a SessionCtx that represents a database instance
ServiceCtx service = (ServiceCtx) ic.lookup ("sess_iiop://localhost:2481:ORCL");

// Create and authenticate a first session in the instance.
SessionCtx session1 = (SessionCtx) service.createSubcontext (":session1");

// Authenticate
session1.login("scott", "tiger", null);

// Create and authenticate a second session in the instance.
SessionCtx session2 = (SessionCtx) service.createSubcontext (":session2");

// Authenticate using a login object (not required, just shown for example).
LoginServer login_server2 = (LoginServer)session2.activate ("/etc/login");
Login login2 = new Login (login_server2);
login2.authenticate ("scott", "tiger", null);

// Activate one Hello object in each session
HelloHome hello_home1 = (HelloHome)session1.activate (objectName);
HelloHome hello_home2 = (HelloHome)session2.activate (objectName);

//create the bean and retrieve the remote interface
Hello hello1 = hello_home1.create ();
Hello hello2 = hello_home2.create ();
```

```
// Verify that the objects are indeed different
System.out.println (hello1.helloWorld ());
System.out.println (hello2.helloWorld ());
```

2つのクライアントが同一セッションにアクセスする場合 クライアントが JNDI lookup メソッドを起動すると、Oracle8i JVM によりセッションが作成されます。2 番目のクライアントからこのセッションでインスタンス化されるオブジェクトにアクセスする場合は、次の操作が必要です。

1. 最初のクライアントにより、オブジェクト・インスタンス・ハンドルと Login オブジェクト参照の両方が保存されます。
2. 第2のクライアントにより、ハンドルと Login オブジェクト参照が取り出され、オブジェクト・インスタンスへのアクセスに使用されます。

例 5-2 2つのクライアントが単一インスタンスにアクセスする場合

1. 最初のクライアントにより、Login オブジェクトの `authenticate` メソッドを介してユーザー名とパスワードが指定され、クライアント自身がデータベースに対して認証されます。
2. セッションが、URL を指定して `lookup` メソッドによって作成されます。
3. Bean が、ホーム・インタフェースの `create` メソッドでインスタンス化されます。
4. `LoginServer` オブジェクトとサーバー・オブジェクトのインスタンス・ハンドルの両方が、第2のクライアントで取り出せるようにファイルに保存されます。

```
// Login to the 8i server
LoginServer lserver = (LoginServer)ic.lookup (serviceURL + "/etc/login");
new Login (lserver).authenticate (username, password, null);

// Activate a Hello in the 8i server
// This creates a first session in the server
HelloHome hello_home = (HelloHome)ic.lookup (serviceURL + objectName);
Hello hello = hello_home.create ();
hello.setMessage ("As created by Client1");
System.out.println ("Client1: " + hello.helloWorld ());

// save Login object into a file, loginFile, for Client2 to read
com.visigenic.vbroker.orb.ORB orb = oracle.aurora.jndi.orb_dep.Orb.init ();
String log = orb.object_to_string (lserver);
OutputStream os = new FileOutputStream (loginFile);
os.write (log.getBytes ());
os.close ();

// save object instance handle into a file, helloFile,
// for Client2 to read
Handle obj_hndl = testBean.getHandle ();
```

```
OutputStream os = new FileOutputStream (helloFile);
os.write (obj_hndl.getBytes ());
os.close ();
```

第2のクライアントが次の操作を実行し、アクティブ・セッション上の Hello オブジェクト・インスタンスにアクセスします。

1. オブジェクト・ハンドルと Login オブジェクトを取り出します。この例では、実装定義のメソッドである readHandle および readLogin を使用して、これらのオブジェクトをファイル・システムから取り出しています。
2. getEJBObject メソッドを介して EJB 参照を取り出します。
3. authenticate メソッドを介して、最初のクライアントと同じ Login オブジェクトでデータベース・セッションに対して認証します。Login オブジェクトは、Login コンストラクタを介して LoginServer オブジェクトから再作成できます。

```
FileInputStream finstream = new FileInputStream (hellofile);
ObjectInputStream istream = new ObjectInputStream (finstream);
javax.ejb.Handle handle = (javax.ejb.Handle)istream.readObject ();
Hello hello = (Hello)helloHandle.getEJBObject ();
finstream.close ();
```

```
// Authenticate with the login Object
LoginServer lserver = (LoginServer) readLogin(loginFile);

//Set the VisiBroker bind options to specify that the
//login is to not try recursively, which means that if it
//fails on the first try, return with the error immediately.
//See VisiBroker manuals for more information.
lserver._bind_options (new BindOptions (false, false));

Login login = new Login (lserver);
login.authenticate (username, password, null);
```

同一セッション中の活性化 サーバー・オブジェクト内で、そのオブジェクトが活性化しているのと同じセッションで、公開されたオブジェクトを新しく検索して活性化する場合は、次の操作を実行します。

```
Context ic = new InitialContext ( );
SomeObject myObj = (SomeObject) ic.lookup("/test/Hello");
```

認証情報のための環境設定や lookup に使用する URL 中のセッションの指定がないことに注意してください。セッションにログインするための認証は、すでに成功しています。また、オブジェクトはローカル・マシンに存在します。このため、同じセッション中では、認証情報やターゲットの sess_iiop URL アドレスを指定しなくても、他のオブジェクトの活性化を進行させることができます。

注意： この項で示すセッション中の活性化は、IIOP クライアントと非 IIOP クライアントの両方に有効です。

リリース 8.1.7 以前の同一セッション中の活性化 リリース 8.1.7 以前は、同一セッション中の活性化は URL の hostname:port:SID ではなく thisServer/:thisSession 表記を使用して行われていました。この表記法は、IIOP クライアントについてののみ引き続き有効です。

たとえば、同じセッションでオブジェクトを検索して活性化するには、次のようにします。

```
Hashtable env = new Hashtable();
env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
Context ic = new InitialContext (env);
SomeObject myObj =
    (SomeObject) ic.lookup("sess_iiop://thisServer/:thisSession/test/Hello");
```

この場合、myObj は、起動する側のオブジェクトが稼動しているセッションと同じセッションで活性化されます。クライアント（この場合はサーバー・オブジェクト）はすでに Oracle8i に対して認証されているので、ログイン認証情報を指定する必要はないことに注意してください。

クライアントはオブジェクトに対して認証されるのではなく、セッションに対して認証される必要があります。ただし、別のセッションを開始する場合は、なんらかの形式の認証（ログイン認証または SSL Credential 認証のいずれか）を実行する必要があります。

注意： thisServer 表記は、サーバー側（つまりサーバー・オブジェクト）でのみ使用できます。クライアント・プログラムでは使用できません。

JNDI コンテキストからのオブジェクトの検索 Sun Microsystems の JNDI では、「/test/myObject」の名前でオブジェクトがバインドされている場合、次のようなコードで Context からオブジェクトを取り出すことができます。

```
Context ctx = ic.lookup("/test");
MyObject myobj = ctx.lookup("myObject");
```

戻されるオブジェクトは活性化済みで、そこからメソッドを起動できます。

Oracle8i では、Context からオブジェクトを取り出そうとすると、アクティブでないオブジェクトが戻されます。アクティブなオブジェクトを得るには次のことを実行する必要があります。

1. Context のかわりに SessionCtx を取り出します。SessionCtx は、次のどちらかの方法で ServiceCtx から取り出すことができます。

- 次のように、最初に ServiceCtx を取り出し、次に ServiceCtx から SessionCtx を取り出します。

```
ServiceCtx service =
    (ServiceCtx) ic.lookup("sess_iiop://localhost:2481:ORCL");
//Retrieve the ServiceCtx subcontext
SessionCtx sess = (SessionCtx) service.lookup("/test");
```

- 次のように、ServiceCtx と SessionCtx を 1 回の検索で取り出します。

```
SessionCtx sess =
    (SessionCtx) ic.lookup("sess_iiop://localhost:2481:ORCL/test");
```

2. セッション中で活性化したいオブジェクトごとに、Oracle 固有の SessionCtx.activate メソッドを実行します。このメソッドにより、セッション中でオブジェクトが活性化され、オブジェクト参照が戻されます。オブジェクトの lookup のみの実行ではアクティブでないオブジェクトが戻されます。活性化済みのオブジェクトを得るには、次のように activate メソッドを実行します。

```
MyObjectHome myObj_home = (MyObjectHome) sess.activate("myObject");
```

3. 最後に、Bean インスタンスを作成して Bean のメソッドを起動します。

```
//create the bean and retrieve the remote interface
MyObject myObj = myObj_home.create ();

// Verify that the objects are indeed different
System.out.println (myObj.printMe ());
```

Oracle8i JVM の JNDI 実装には、Context オブジェクトの次の 2 つの実装が用意されています。

- ServiceCtx — sess_iiop URL を介してデータベース・インスタンスを識別します。
- SessionCtx — データベース内のデータベース・セッションを表します。

検索の実行時には、データベースを識別する ServiceCtx と、実際に JNDI にバインドされているオブジェクトを取り出す SessionCtx の両方を検索する必要があります。通常は、lookup メソッドに指定する JNDI URL 内で両方のオブジェクトの URL を指定します。ただし、前述のように各オブジェクトを個別に取り出すこともできます。

セッション・タイムアウトの設定

セッションとその状態は、通常は最後の接続とともに終了します。ただし、次のように、最後の接続が終了した後も、セッションとその状態を指定の時間間隔だけアイドル状態に保つ必要がある場合があります。

- 中間層レイヤーでは、接続の保持が高コストのためセッションへの接続はオープン状態にしておきたくないが、後続の受信クライアント要求に備えてセッションはオープン状態に保ちたいといった場合があります。
- 接続を異常終了させるようなネットワーク上の問題が発生したときに、接続を再確立できるようにセッションを指定の期間だけ維持したい場合があります。
- アプリケーションが接続終了前にセッション内の既存オブジェクトへのハンドルを他のクライアントに渡すとき、第2のクライアントはセッションにアクセスするための時間を必要とします。

タイムアウト・クロックは、セッションへの最後の接続の終了時に起動します。セッションへの他の接続が時間枠内に開始されると、タイムアウト・クロックがリセットされます。それ以外の場合は、セッションが終了します。

セッションのアイドル・タイムアウトは、クライアントから、またはサーバー・オブジェクト内から設定できます。

- [クライアントからのセッション・タイムアウトの設定](#)
- [サーバー・オブジェクトからのセッション・タイムアウトの設定](#)

クライアントからのセッション・タイムアウトの設定

クライアント上でのアイドル・タイムアウトは、事前に公開されているユーティリティ・オブジェクト `oracle.aurora.AuroraServices.Timeout` によって設定できます。このオブジェクトは、`"/etc/timeout"` という名前で事前に公開されています。このオブジェクトの `setTimeout` メソッドを使用します。

1. `"/etc/timeout"` の JNDI 検索によって `Timeout` オブジェクトを取り出します。
2. `setTimeout` メソッドでセッション・アイドル時間を秒単位で指定して、タイムアウトを設定します。

```
Timeout timeout = (Timeout)ic.lookup(serviceURL + "/etc/timeout");
System.out.println("Setting a timeout of 20 seconds ");
timeout.setTimeout(20);
```

サーバー・オブジェクトからのセッション・タイムアウトの設定

サーバー・オブジェクトでは、`sessionTimeout()` メソッドを含む `oracle.aurora.net.Presentation` オブジェクトを使用して、セッション・タイムアウトを制御できます。このメソッドは、パラメータとしてセッション・タイムアウト値（秒数）を取ります。たとえば、次のようになります。

```
int timeoutValue = 30;
...
// set the timeout to 30 seconds
oracle.aurora.net.Presentation.sessionTimeout(timeoutValue);
...
// set the timeout to a very long time
oracle.aurora.net.Presentation.sessionTimeout(Integer.MAX_INT);
```

注意： `sessionTimeout()` メソッドを使用するときは、CLASSPATH に `$(ORACLE_HOME)/javavm/lib/aurora.zip` を追加する必要があります。

Oracle8i JVM のバージョン・ナンバーの検索

事前に公開されている `oracle.aurora.AuroraServices.Version` オブジェクトを介して、データベースにインストールされている Oracle8i JVM のバージョンを検索できます。このオブジェクトは、JNDI ネームスペース内で `"/etc/version"` として公開されています。Version オブジェクトには `getVersion` メソッドが含まれており、このメソッドは「8.1.7」のようにバージョンを含む文字列を戻します。Version オブジェクトを取り出すには、JNDI 検索で `"/etc/version"` を指定します。次の例では、バージョン・ナンバーを取り出しています。

```
Version version = (Version)ic.lookup(serviceURL + "/etc/version");
System.out.println("The server version is : " + version.getVersion());
```

非 IIOP プレゼンテーションからのセッション中の EJB オブジェクトの活性化

HTTP や DCOM など、非 IIOP サーバー・リクエストでは、同一セッション中で EJB オブジェクトを活性化できます。

- HTTP HTTP クライアントは Oracle Servlet Engine と対話して、JSP またはサーブレットを実行します。これによって、実行中の同じセッションで EJB オブジェクトを活性化できます。
- DCOM DCOM クライアントでは、Oracle8i JVM へのアクセスに DCOM ブリッジが使用されます。DCOM ブリッジ・セッションは実行中の同じセッションで EJB オブジェクトを活性化できます。

非 IIOP サーバー・オブジェクトで、実行中の同じセッションで公開されたオブジェクトを新しく検索して活性化する場合は、次の操作を実行できます。

```
Context ic = new InitialContext();
SomeObject myObj = (SomeObject) ic.lookup("/test/Hello");
```

注意： このメソッドを介して IIOP オブジェクト参照を取得した場合、このオブジェクトをリモート・クライアントまたはサーバーに渡すことはできません。

認証情報のための環境設定や lookup に使用する URL 中のアドレスの指定がないことに注意してください。セッションにログインするための認証は、すでに成功しています。また、オブジェクトはローカル・マシンに存在します。このため、同一セッション中では、認証情報やターゲットの URL アドレスを指定しなくても、他のオブジェクトの活性化を進行させることができます。

アプレットからの EJB オブジェクトの起動

サーバー・オブジェクトをアプレットから起動する方法は、クライアントから起動する場合と同じです。ただし、次の相違点があります。

- アプレット規格に準拠する必要があります。
- Java プラグイン規格に準拠する必要があります。サポートされる Java プラグインは、JDK 1.1、Java2 および Oracle の JInitiator です。
- オブジェクト検索の前に、初期コンテキスト環境でプロパティ ORBdisableLocator、ORBClass および ORBSingletonClass を設定します。

署名付き JAR ファイルを使用したサンドボックス・セキュリティへの準拠

セキュリティ・サンドボックスにより、アプレットはローカル・ディスク上の要素へのアクセスや、アプレットのダウンロード元となったホスト以外のリモート・ホストへの接続が制限されます。信頼されている第三者として署名付き JAR ファイルを作成する場合は、サンドボックス・セキュリティの制限を受けません。アプレットのサンドボックス・セキュリティと署名付き JAR ファイルの詳細は、<http://java.sun.com> を参照してください。

アプレット内のオブジェクト検索の実行

アプレット内で JNDI 検索を実行する方法は、他の Oracle Java クライアントの場合と同じですが、初期コンテキスト内で次のプロパティを設定します。

```
env.put (ServiceCtx.APPLET_CLASS, this);
```

デフォルトでは、アプレットを実行するためにクライアントに JAR ファイルをインストールする必要はありません。ただし、Oracle JAR ファイルをクライアント・マシンにインストー

ルする場合は、次のように InitialContext 環境で ClassLoader プロパティを設定します。

```
env.put('ClassLoader', this.getClass().getClassLoader());
```

次の例は、Bank の例を起動するアプレット内の init メソッドを示しています。このアプレットでは、APPLET_CLASS プロパティの設定など、初期コンテキストが設定され、URL を指定して JNDI 検索が実行されます。

```
public void init() {
    // This GUI uses a 2 by 2 grid of widgets.
    setLayout(new GridLayout(2, 2, 5, 5));
    // Add the four widgets.
    add(new Label("Account Name"));
    add(_nameField = new TextField());
    add(_checkBalance = new Button("Check Balance"));
    add(_balanceField = new TextField());
    // make the balance text field non-editable.
    _balanceField.setEditable(false);
    try {
        // Initialize the ORB (using the Applet).
        Hashtable env = new Hashtable();
        env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
        env.put(Context.SECURITY_PRINCIPAL, "scott");
        env.put(Context.SECURITY_CREDENTIALS, "tiger");
        env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
        env.put(ServiceCtx.APPLET_CLASS, this);

        Context ic = new InitialContext(env);
        _manager = (AccountManager)ic.lookup
            ("sess_iiop://hostfunk:2222/test/myBank");
    } catch (Exception e) {
        System.out.println(e.getMessage());
        e.printStackTrace();
        throw new RuntimeException();
    }
}
```

このアプレットの action メソッド内で、取得されたオブジェクトから各メソッドが起動されます。この例では、取得された AccountManager オブジェクトの open メソッドが起動されています。

```
public boolean action(Event ev, Object arg) {
    if(ev.target == _checkBalance) {
        // Request the account manager to open a named account.
        // Get the account name from the name text widget.
        Bank.Account account = _manager.open(_nameField.getText());
    }
}
```

```
// Set the balance text widget to the account's balance.
_balanceField.setText(Float.toString(account.balance()));
return true;
}
return false;
}
```

EJB オブジェクトにアクセスするアプレット用の HTML の修正

Oracle8i では、アプレット内でロードされる HTML ページ用にサポートされる Java プラグインは、JDK 1.1、Java 2 および Oracle JInitiator のみです。各プラグインでの、アプレット情報を設定する構文はそれぞれ異なります。ただし、各 HTML ページに含めるプロパティの設定の内容は以下の 2 つです。

- ORBdisableLocator を TRUE に設定 — すべてのアプレットに必須です。
- ORBClass および ORBSingletonClass を定義 — Java 2 または JInitiator プラグインを使用するアプレットに必須です。

注意： サンドボックス・セキュリティ・ルールの制限により、システム・プロパティの設定や読み込みはできません。したがって、ORB ランタイムに渡す値は、アプレットのパラメータ内で設定する必要があります。これは、ORBdisableLocator、ORBClass および ORBSingletonClass プロパティの設定に使用されます。

次の項の例は、各プラグイン・タイプに則した HTML 定義の作成方法を示しています。各 HTML 定義では、アプレット bank の例を定義しています。

- [例 5-3 「JDK 1.1 プラグインの HTML 定義」](#)
- [例 5-4 「Java 2 プラグインの HTML 定義」](#)
- [例 5-5 「JInitiator プラグインの HTML 定義」](#)

例 5-3 JDK 1.1 プラグインの HTML 定義

```
<pre>
<html>
<title>Applet talking to 8i</title>
<h1>applet talking to 8i using java plug in 1.1  </h1>
<hr>
The bank example
Specify the plugin in codebase, the class within the CODE parameter, the JAR
files in the ARCHIVE parameter, the plugin version in the type parameter, and
set ORBdisableLocator to true.
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
        WIDTH = 500 HEIGHT = 50
```

```

codebase="http://java.sun.com/products/plugin/1.1/
jinstall-11-win32.cab#Version=1,1,0,0">
<PARAM NAME = CODE VALUE = OracleClientApplet.class >
<PARAM NAME = ARCHIVE VALUE = "oracleClient.jar,
aurora_client.jar,vbjorb.jar,vbjapp.jar" >
<PARAM NAME="type" VALUE="application/x-java-applet;version=1.1">
<PARAM NAME="ORBdisableLocator" VALUE="true">
<COMMENT>
Set the plugin version in the type, set ORBdisableLocator to true, the applet
class within the java_CODE tag, the JAR files in the java_ARCHIVE tag, and the
plug-in source site within the pluginspage tag.
<EMBED type="application/x-java-applet;version=1.1"
ORBdisableLocator="true"
java_CODE = OracleClientApplet.class
java_ARCHIVE = "oracleClient.jar,
aurora_client.jar,vbjorb.jar,vbjapp.jar"
WIDTH = 500 HEIGHT = 50
pluginspage="http://java.sun.com/products/plugin/1.1/plugin-install.html">
</NOEMBED></COMMENT>
</NOEMBED></EMBED>
</OBJECT>

</center>
<hr>
</pre>

```

例 5-4 Java 2 プラグインの HTML 定義

```

<pre>
<html>
<title>applet talking to 8i</title>
<h1>applet talking to 8i using Java plug in 1.2 </h1>
<hr>
The bank example
Specify the plugin in codebase, the class within the CODE parameter, the JAR
files in the ARCHIVE parameter, the plugin version in the type parameter, and
set ORBdisableLocator to true.
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
WIDTH = 500 HEIGHT = 50
codebase="http://java.sun.com/products/plugin/1.2/jinstall-11-win32.cab#
Version=1,1,0,0">
<PARAM NAME = CODE VALUE = OracleClientApplet.class >
<PARAM NAME = ARCHIVE VALUE = "oracleClient.jar,
aurora_client.jar,vbjorb.jar,vbjapp.jar" >
<PARAM NAME="type" VALUE="application/x-java-applet;version=1.1.2">
<PARAM NAME="ORBdisableLocator" VALUE="true">

```

```

<PARAM NAME="org.omg.CORBA.ORBClass" VALUE="com.visigenic.vbroker.orb.ORB">
<PARAM NAME="org.omg.CORBA.ORBSingletonClass"
    VALUE="com.visigenic.vbroker.orb.ORB">
<COMMENT>
Set the plugin version in the type, set ORBdisableLocator to true, the ORBClass
and ORBSingletonClass to the correct ORB class, the applet
class within the java_CODE tag, the JAR files in the java_ARCHIVE tag, and the
plug-in source site within the pluginspage tag.
<EMBED type="application/x-java-applet;version=1.1.2"
    ORBdisableLocator="true"
    org.omg.CORBA.ORBClass="com.visigenic.vbroker.orb.ORB"
    org.omg.CORBA.ORBSingletonClass="com.visigenic.vbroker.orb.ORB"
    java_CODE = OracleClientApplet.class
    java_ARCHIVE = "oracleClient.jar,
        aurora_client.jar,vbjorb.jar,vbjapp.jar"
    WIDTH = 500 HEIGHT = 50
pluginspage="http://java.sun.com/products/plugin/1.2/plugin-install.html">
<NOEMBED></COMMENT>
</NOEMBED></EMBED>
</OBJECT>

</center>
<hr>
</pre>

```

例 5-5 JInitiator プラグインの HTML 定義

```

<h1> applet talking to 8i using JInitiator 1.1.7.18</h1>
<COMMENT>
Set the plugin version in the type, set ORBdisableLocator to true, the
ORBClass and ORBSingletonClass to the correct ORB class, the applet
class within the java_CODE tag, the source of the applet in the java_CODEBASE
and the JAR files in the java_ARCHIVE tag.
<EMBED type="application/x-jinit-applet;version=1.1.7.18"
    java_CODE="OracleClientApplet"
    java_CODEBASE="http://hostfunk:8080/applets/bank"
    java_ARCHIVE="oracleClient.jar,aurora_client.jar,vbjorb.jar,vbjapp.jar"
    WIDTH=400
    HEIGHT=100
    ORBdisableLocator="true"
    org.omg.CORBA.ORBClass="com.visigenic.vbroker.orb.ORB"
    org.omg.CORBA.ORBSingletonClass="com.visigenic.vbroker.orb.ORB"
    serverHost="orasundb"
    serverPort=8080
<NOEMBED>
</COMMENT>

```

</NOEMBED>
</EMBED>

IIOP のセキュリティ

セキュリティは、データ整合性、認証および認可を含みます。

- データ整合性については、Oracle8i ではアプリケーションでセキュア・ソケット・レイヤー（SSL）上で IIOP を使用できます。
- 認証については、アプリケーションではユーザー名 / パスワードの組合せを指定するか、証明書を選択するかを選択できます。
- 認可については、受信クライアントが指定を要求されるトラスト・ポイントのレベルを選択できます。

次の項では、これらの事項について詳細に説明します。

- [概要](#)
- [データ整合性](#)
- [認証](#)
- [クライアント側の認証](#)
- [サーバー側の認証](#)
- [認可](#)

概要

『Oracle8i Java 開発者ガイド』で説明しているように、アプリケーションで考慮する必要のあるセキュリティ上の問題がいくつかあります。『Oracle8i Java 開発者ガイド』では、セキュリティの問題を、ネットワーク接続、データベースの内容および JVM に分けて別々に説明します。これらの問題はすべて IIOP と関係しています。ただし、次に説明するように、ネットワーク・セキュリティおよび JVM のセキュリティでは実装上の特有の問題があります。

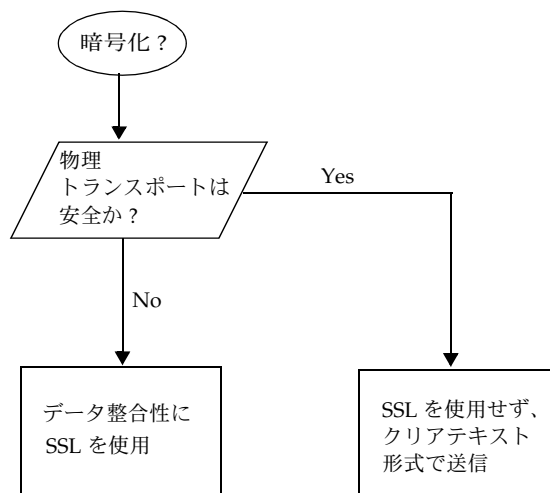
- JVM のセキュリティには、Java2 のパーミッションの利用と実行権限の付与が含まれます。IIOP の場合、次のいずれかの方法で実行権限を付与できます。
 - * CORBA — 所有者は、loadjava ツールでオプションを使用して CORBA オブジェクトに実行権限を付与します。CORBA のクラスをロードする際の実行権限付与の詳細は、『Oracle8i Java 開発者ガイド』の loadjava の説明を参照してください。
 - * EJB — 所有者は、EJB オブジェクトに実行権限を付与します。このとき、ディプロイメント・ディスクリプタ内のメソッドに対しても実行権限が付与される場合があります。
- ネットワーク接続のセキュリティには、次の問題が含まれます。
 - * **データ整合性** — 送信内容がネットワークの外部から直接読み取られるのを防止するために、すべての送信をコード化します。Oracle では、暗号化を行うためにセキュア・ソケット・レイヤー（SSL）をサポートします。
 - * **認証** — 権限のないユーザーが権限があるふりをしてネットワークに接続するのを防止するために、クライアントまたはサーバーは認証情報を提供します。この情報は、ユーザー名とパスワードを組み合わせた形式か、または証明書の形式で使用できます。
 - * **認可** — オブジェクトにアクセスできるユーザーであることを証明するために、次の 2 つのタイプの認可を実行します。
 - セッション認可 — セッションに対する許可をユーザーに付与します。この場合、クライアントは提供したユーザー名または証明書が有効であることを証明して、サーバーにアクセスする認可を取得します。
 - ユーザー認可 — クライアントまたはサーバーは、提供された証明書に対して認可を実行できます。ユーザー認可を実行できるのは、クライアントまたはサーバーが証明書を提供して自分自身を認証する場合のみです。

この項では、IIOP アプリケーションで考慮する必要がある、ネットワーク接続のセキュリティ上の問題を詳しく説明します。

データ整合性

トランスポート回線を暗号化する必要性、およびデータの整合性と機密性を保持する必要性について検討します。物理的な接続への干渉が予想される場合は、セキュア・ソケット・レイヤー（SSL）による暗号化テクノロジーを使用してすべての送信の暗号化を検討します。ただし、送信に暗号化を使用すると接続パフォーマンスに影響があるため、トランスポート層上にセキュリティ上の問題がない場合は、暗号化せずに送信することをお勧めします。

図 6-1 データ整合性と暗号化



セキュア・ソケット・レイヤー（SSL）の使用

Oracle8i JVM の CORBA および EJB 実装は、データ整合性を保ち、認証を実行するセキュア・ソケット・レイヤー（SSL）に依存します。SSL は、保護ネットワーク・プロトコルで、Netscape Communications, Inc. によって最初に規定されました。Oracle8i JVM では、ORB に使用される IIOP プロトコルの SSL 上での通信がサポートされます。

クライアントとサーバーの間で接続がリクエストされると、両側の SSL レイヤーによって接続ハンドシェイク時にネゴシエーションが行われ、接続を許可するかどうかを検証されます。接続の検証にはいくつかのレベルがあります。

1. トランスポートの際にデータ整合性が保証されるように、クライアントとサーバーの SSL のバージョンが一致している必要があります。
2. サーバー側の証明書を使用した認証を行う必要がある場合、サーバーで提供される証明書がクライアントにより SSL レイヤーで検証されます。これにより、接続の対象が正し

いサーバーであることが保証されます。つまり、サーバーを装う第三者には接続されません。

3. クライアント側の証明書を使用した認証を行う必要がある場合、クライアントで提供される証明書が SSL レイヤーで検証されます。サーバーは、クライアントから認証、認可のためにクライアントの証明書を受け取ります。

注意： 通常、クライアント側の認証は、クライアントが偽のクライアントではなく、信頼されているクライアントであるとサーバーが検証することのみを意味します。ただし、Oracle8i JVM で SSL_CLIENT_AUTH を指定すると、サーバー側とクライアント側の両方で認証が必要になります。

SSL レイヤーは、ピア間の認証を実行します。ハンドシェイクが終了すると、ピアが自分自身であると認証されたことが確認できます。さらに、証明書の連鎖で、このユーザーに対してアプリケーションへのアクセスを許可するための追加のテストを実行できます。認証後の作業方法は、6-26 ページの「認可」を参照してください。

注意： SSL を使用する場合、クライアントに次の JAR ファイルをインポートする必要があります。

- クライアントで JDK 1.1 を使用している場合は、jssl-1_1.jar と javax-ssl-1_1.jar をインポートします。
 - クライアントで Java 2 を使用している場合は、jssl-1_2.jar と javax-ssl-1_2.jar をインポートします。
-

SSL のバージョンのネゴシエーション

SSL は、クライアント側とサーバー側の SSL プロトコルのバージョン番号が一致していることを確認します。指定できる値は次のとおりです。

- Undetermined : SSL_UNDETERMINED (デフォルト)。
- 3.0 with 2.0 Hello: この設定はサポートされていません。
- 3.0: SSL_30。
- 2.0: この設定はサポートされていません。

データベースでは、デフォルトは Undetermined です。データベースは 2.0 も 3.0 with 2.0 Hello もサポートしません。したがって、クライアントで使用できるのは Undetermined または 3.0 のみです。

- サーバーのバージョンは、データベースの SQLNET.ORA ファイルで SSL_VERSION パラメータを使用して設定します。たとえば、SSL_VERSION = 3.0 と設定します。
- クライアントの場合、次のように、クライアントの JNDI 環境で SSL クライアントのバージョン・ナンバーを設定します。

```
environment.put("CLIENT_SSL_VERSION", ServiceCtx.SSL_30);
```

表 6-1 は、クライアントとサーバーの SSL のバージョン設定に応じて、ハンドシェイクが成功する場合と失敗する場合を示します。×印は、ハンドシェイクに失敗する場合を示します。

表 6-1 SSL バージョン・ナンバー

	サーバーの設定			
	Undetermined	3.0 W/2.0 Hello (サポートされない)	3.0	2.0 (サポート されない)
クライアントの設定				
Undetermined	3.0	X	X	X
3.0 W/2.0 Hello (サポートされない)	X	X	X	X
3.0	3.0	X	3.0	X
2.0 (サポートされ ない)	X	X	X	X

認証

認証は、認証を要求された側が要求側に対して自分自身を識別する情報を提供するプロセスです。この情報は、自分が偽者ではないことを保証するものです。クライアント / サーバー分散環境では、クライアントまたはサーバーから認証を要求できます。

- サーバー側の認証 — サーバーは、自分自身を認証する識別情報を送信します。クライアントは、この情報を使用して相手がサーバー自身であり偽者ではないことを検証します。SSL を要求すると、サーバーは常に証明書ベースの認証情報を送信します。
- クライアント側の認証 — 同じ理由からクライアントも、自分自身を認証するためにサーバーに識別情報を送信します。送信する情報には、ユーザー名とパスワードの組合せまたは証明書のいずれかが含まれます。クライアントはデータベースにログインするので、常にデータベースに対して自分自身を認証する必要があります。

- コールアウト認証 — サーバーが別のオブジェクトへのコールを行います。これにより、サーバーはクライアントとして動作します。このとき、サーバーは認証情報を提供して自分自身をクライアントとして認証する必要がありますが、データベース・サーバーとしての認証情報をこの目的に使用することはできません。
- コールバック認証 — サーバーには、クライアント上に存在するオブジェクトへコールバックできるように、CORBA IOR または EJB ハンドルが与えられます。この使用例では、サーバーはクライアントとして動作します。このとき、サーバーは認証情報を提供して自分自身をクライアントとして認証する必要がありますが、データベース・サーバーとしての認証情報をこの目的に使用することはできません。

クライアント側の認証

Oracle データ・サーバーはセキュリティを考慮したサーバーであるため、クライアント・アプリケーションはまずデータベース・サーバーで認証されていないと、データベースに格納されているデータにアクセスできません。Oracle8i CORBA サーバー・オブジェクトおよび Enterprise JavaBeans は、データベース・サーバーで実行されます。クライアントがこのオブジェクトを活性化し、そこでメソッドを起動するには、サーバーに対して自分自身を認証する必要があります。クライアントの認証は、CORBA オブジェクトまたは EJB オブジェクトが新しいセッションを開始するときに行います。各 IIOP クライアントは、たとえば次のようにしてデータベースに対して自分自身を認証する必要があります。

- クライアントが最初に新しいセッションを開始する場合は、そのクライアントがデータベースに対して自分自身を認証します。
- クライアントが 2 番目のクライアントにオブジェクト参照（CORBA IOR または EJB Bean ハンドル）を渡す場合は、2 番目のクライアントがオブジェクト参照で指定されたセッションに接続します。2 番目のクライアントは、サーバーに対して自分自身を認証します。

クライアントは、次のいずれかの方法で自分自身を認証します。

認証のタイプ	定義
証明書	ユーザー証明書、認証局の証明書（または、ユーザー証明書と認証局の証明書の両方とその他の識別証明書を含む証明書の連鎖）および秘密鍵を提供できます。
ユーザー名およびパスワードの組合せ	Credential またはログイン・プロトコルを使用して、ユーザー名とパスワードを提供できます。また、ユーザー名とパスワードとともに、データベース・ロールをサーバーに渡すことができます。

クライアント側の認証のタイプは、サーバーの構成により決まります。SQLNET.ORA ファイル内で SSL_CLIENT_AUTHENTICATION パラメータが TRUE に設定されている場合、クライアントは証明書ベースの認証を提供する必要があります。
SSL_CLIENT_AUTHENTICATION パラメータが FALSE に設定されている場合は、クライア

ントはユーザー名とパスワードの組合せを使用して自分自身を認証します。
SSL_CLIENT_AUTHENTICATION パラメータが TRUE に設定されている場合にクライアントがユーザー名とパスワードを提供すると、接続のハンドシェイクは失敗します。

次の表は、クライアントが認証に際して使用できるオプションの概要を示したものです。

- 表の列には、データ整合性を保つために SSL を使用する場合に選択可能なオプションを挙げています。
- 表の行には、ログイン・プロトコル、Credential および証明書という 3 つの認証方式を挙げています。
- 表内の項目では、クライアント側の認証タイプを実装する際に使用する必要がある様々なメソッドを詳しく説明します。

認証方式	非 SSL トランスポート	SSL トランスポート
ログイン・プロトコルを使用したユーザー名とパスワードの提供	<ul style="list-style-type: none">■ 暗黙的な方法: JNDI プロパティを NON_SSL_LOGIN に設定します。JNDI プロパティにユーザー名とパスワードを指定します。■ 明示的な方法: ユーザー名とパスワードを使用してログイン・オブジェクトを作成します。	<ul style="list-style-type: none">■ 暗黙的な方法: JNDI プロパティを SSL_LOGIN に設定します。JNDI プロパティにユーザー名とパスワードを指定します。■ 明示的な方法: ユーザー名とパスワードを使用してログイン・オブジェクトを作成します。
Credential を使用したユーザー名とパスワードの提供	パスワードはクリアテキストで送信されるので、サポートされません。	JNDI プロパティを SSL_CREDENTIAL に設定します。ユーザー名とパスワードはハンドシェイク時に暗黙的にサーバーに送信されます。
証明書の提供	証明書では SSL トランスポートが必要なので、サポートされません。	JNDI プロパティを SSL_CLIENT_AUTH に設定します。JNDI プロパティに、クライアント証明書、CA 証明書および秘密鍵を指定します。 CORBA オブジェクトは、AuroraCertificateManager クラスを使用して、証明書、CA 証明書および秘密鍵を指定します。

表に示すように、ほとんどの認証オプションで JNDI プロパティに適切な値を設定する必要があります。

認証のための JNDI の使用

JNDI を使用してクライアント側の認証を設定するには、

`javax.naming.Context.SECURITY_AUTHENTICATION` 属性を次のいずれかの値に設定します。

- `ServiceCtx.NON_SSL_LOGIN` — 通常の IIOP 接続を使用します。SSL を使用しないので、トランスポート層上に送信されるデータは一切暗号化されません。したがって、パスワードを保護するために、クライアントはログイン・プロトコルを使用して自分自身を認証します。また、サーバー自身を識別する SSL 証明書はクライアントに提供されません。
- `ServiceCtx.SSL_LOGIN` — SSL を使用できる IIOP 接続を使用します。トランスポート層上に送信されるデータはすべて暗号化されます。クライアント認証に証明書を使用しない場合、ログイン・プロトコルを使用してユーザー名とパスワードを提供します。

この接続は SSL 接続なので、サーバーにより証明書の識別情報がクライアントに送信されます。クライアントには、サーバー認証のために必要に応じてサーバー証明書を検証する役割があります。また、クライアントはサーバー証明書を検証するための Trustpoint を設定できます。

- `ServiceCtx.SSL_CREDENTIAL` — SSL を使用できる IIOP 接続を使用します。トランスポート層上に送信されるデータはすべて暗号化されます。クライアントは、ログイン・プロトコルを使用せずにユーザー名とパスワードを提供して、サーバーに対するクライアント認証を行います。ユーザー名とパスワードは、最初のメッセージを送信するときに、セキュリティ・コンテキストで自動的にサーバーに渡されます。

注意： クライアントのパスワードは SSL の場合のように暗号化されません。SSL 接続を介したパスワードの暗号化は冗長なので、`SSL_CREDENTIAL` を使用方法は、`SSL_LOGIN` を使用方法よりも多少効率的です。

サーバーにより、証明書の識別情報がクライアントに提供されます。クライアントには、サーバー認証のために必要に応じてサーバー証明書を検証する役割があります。

- `ServiceCtx.SSL_CLIENT_AUTH` — SSL を使用できる IIOP 接続を使用します。トランスポート層上に送信されるデータはすべて暗号化されます。クライアントは、クライアント側の認証のために適切な証明書をサーバーに提供します。また、サーバーにより証明書の識別情報がクライアントに提供されます。クライアントには、必要に応じてサーバー証明書を許可する役割があります。
- 何も指定しません。クライアントは、サーバー側のオブジェクトのメソッドを活性化して起動する前に、ログイン・プロトコルを明示的に活性化する必要があります。クライアントが既存のセッションに接続し、既存のオブジェクトのメソッドを起動する必要がある場合は、この方法を使用します。詳細は、`demo/examples/corba/session/sharedsession` の例を参照してください。初期

コンテキスト環境のユーザー名とパスワードは、ログイン・オブジェクトの `authenticate()` メソッドにパラメータとして自動的に渡されます。

これらの各オプションにおいて、次のいずれか 1 つまたは複数の処理を実行します。

クライアント認証	<ul style="list-style-type: none">■ ログイン・プロトコルを使用して、サーバーに対して自分自身を認証します。■ 通常のユーザー名とパスワードを使用して、サーバーに対して自分自身を認証します。■ SSL 証明書を使用して、サーバーに対して自分自身を認証します。
サーバー認証	<ul style="list-style-type: none">■ SSL 証明書を使用して、クライアントに対して自分自身を認証します。

クライアント認証またはサーバー認証を行うためにこれらのメソッドをそれぞれ実装する方法の詳細は、次の項を参照してください。

- [ユーザー名とパスワードを使用したクライアント側の認証](#)
- [クライアント認証のための証明書の使用](#)
- [サーバー側の認証](#)

ユーザー名とパスワードを使用したクライアント側の認証

クライアントは、ユーザー名とパスワードを使用するか、または適切な証明書を提供して、データベース・サーバーに対して自分自身を認証します。ユーザー名とパスワードは、SSL トランスポート接続で Oracle のログイン・プロトコルまたは `Credential` のいずれかを使用して提供されます。

- JNDI プロパティを設定してユーザー名とパスワードを提供します。JNDI プロパティを設定すると、ユーザー名とパスワードがログイン・プロトコルに暗黙的に設定されます。SECURITY_AUTHENTICATION を `ServiceCtx.SSL_LOGIN` または `ServiceCtx.NON_SSL_LOGIN` に設定します。
- `Credential` を使用してユーザー名とパスワードを提供します。ユーザー名とパスワードは暗黙的のうちに提供され、暗号化された SSL トランスポートによりサーバーに送信されます。SECURITY_AUTHENTICATION を `serviceCtx.SSL_CREDENTIAL` に設定します。
- 明示的に活性化したログイン・プロトコルでユーザー名とパスワードを提供します。

注意： Login クラスは、ログイン・ハンドシェイキング・プロトコルのクライアント側の実装およびサーバー・ログイン・オブジェクトをコールするためのプロキシ・オブジェクトとして使用されます。このコンポーネントは、aurora_client.jar ファイルにパッケージ化されています。すべての Oracle8i ORB アプリケーションは、このライブラリをインポートする必要があります。

JNDI プロトコルをログイン・プロトコルに設定したユーザー名の送信

クライアントは、ログイン・プロトコルを使用して、Oracle8i データ・サーバーに対して自分自身を認証できます。ログイン・プロトコルは、SSL による暗号化を行っていてもなくても使用できます。これは、セキュリティを考慮したハンドシェイキング暗号化プロトコルがログイン・プロトコルに組み込まれているためです。

アプリケーションでクライアント / サーバー間のデータ・セキュリティのために SSL 接続が必要な場合は、JNDI 初期コンテキストの取得時に渡される

SECURITY_AUTHENTICATION プロパティに **SSL_LOGIN** サービス・コンテキスト値を指定します。次の例は、ログイン・プロトコル用に SSL を使用可能にした接続を定義します。ユーザー名とパスワードが設定されていることに注意してください。

```
Hashtable env = new Hashtable();
env.put (javax.naming.Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put (javax.naming.Context.SECURITY_PRINCIPAL, username);
env.put (javax.naming.Context.SECURITY_CREDENTIALS, password);
env.put (javax.naming.Context.SECURITY_AUTHENTICATION, ServiceCtx.SSL_LOGIN);
Context ic = new InitialContext (env);
...
```

アプリケーションで SSL 接続を使用しない場合は、SECURITY_AUTHENTICATION パラメータの **NON_SSL_LOGIN** を次のように指定します。

注意： ログイン・ハンドシェイキングは暗号化により保護されますが、それ以外のクライアント / サーバー間の対話は保護されません。

```
env.put (javax.naming.Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
```

4 つすべての JNDI Context 変数 (URL_PKG_PREFIXES、SECURITY_PRINCIPAL、SECURITY_CREDENTIALS および SECURITY_AUTHENTICATION) に対して値を指定すると、Context.lookup() メソッドの最初の起動時にログインが自動的に実行されます。

接続を行うクライアントがすでに IOR を得ているために JNDI による検索を行わない場合、オブジェクトに対する IOR をこのクライアントに渡したユーザーは、活性化済みのオブジェクトと同じセッション内に存在するログイン・オブジェクトの IOR も渡しておく必要があります。活性化済みのオブジェクトのメソッドを起動する前に、ログイン・オブジェクトの authenticate メソッドにユーザー名とパスワードを指定する必要があります。

Oracle8i JVM セッションへのログインとログアウト セッション所有者がセッションを終了する場合、所有者は LogoutServer オブジェクトの logout メソッドを使用できます。このオブジェクトは、/etc/logout として事前に公開されています。セッションを終了するには、LogoutServer オブジェクトを使用します。logout を実行できるのは、セッション所有者のみです。他の所有者が実行すると、NO_PERMISSION 例外が発生します。

LogoutServer オブジェクトは、/etc/login として事前に公開されている LoginServer オブジェクトに似ています。LoginServer オブジェクトを使用して、サーバーへの認証に使用する Login オブジェクトを生成します。JNDI lookup の際に Login オブジェクトを暗黙的に使用することもできます。

次の例では、クライアントで LoginServer オブジェクトを使用して認証し、LogoutServer オブジェクトを介してセッションを終了しています。

```
import oracle.aurora.AuroraServices.LoginServer;
import oracle.aurora.AuroraServices.LogoutServer;
...
// To log in using the LoginServer object
LoginServer loginServer = (LoginServer)ic.lookup(serviceURL + "/etc/login");
Login login = new Login(loginServer);
System.out.println("Logging in ..");
login.authenticate(user, password, null);
...
//To logout using the LogoutServer
LogoutServer logout = (LogoutServer)ic.lookup(serviceURL + "/etc/logout");
logout.logout();
```

Credential を使用した暗黙的なユーザー名の送信

ServiceCtx.SSL_CREDENTIAL 認証タイプを使用すると、ユーザー名、パスワードおよびロール（指定されている場合）が最初のリクエストでサーバーに渡されます。この情報は SSL 接続で渡されるので、パスワードは転送プロトコルにより暗号化され、ログイン・プロトコルで使用するハンドシェイキングは必要ありません。このため、Credential を使用した送信はわずかに効率が良く、SSL 接続にお勧めできます。

ログイン・オブジェクトを明示的に活性化して行うユーザー名の送信

データベースにログインするためのログイン・オブジェクトを明示的に作成できます。通常、クライアントから複数のセッションを作成して使用する場合に、この方法を使用します。クライアントが2つの別々のセッションを作成してログインする例を次に示します。これを行うには、次のステップを実行します。

1. 初期コンテキストを作成します。
2. 接続先データベースの URL で検索を実行します。
3. このデータベース・サービス・コンテキストでは、2つのサブコンテキスト（各セッションにつき1つ）が作成されます。

4. ログイン・オブジェクトを使用して各セッションにログインします。このとき、ユーザー名とパスワードを提供します。

注意： 接続先データベースが同じなので、2つのセッションに対するユーザー名とパスワードは同じになります。クライアントが2つの別々のデータベースに接続する場合は、異なるユーザー名とパスワードでのログインが必要な場合もあります。

```
// Prepare a simplified Initial Context as we are going to do
// everything by hand
Hashtable env = new Hashtable ();
env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
Context ic = new InitialContext (env);

// Get a SessionCtx that represents a database instance
ServiceCtx service = (ServiceCtx)ic.lookup (serviceURL);

// Create and authenticate a first session in the instance.
SessionCtx session1 = (SessionCtx)service.createSubcontext (":session1");
LoginServer login_server1 = (LoginServer)session1.activate ("etc/login");
Login login1 = new Login (login_server1);
login1.authenticate (user, password, null);

// Create and authenticate a second session in the instance.
SessionCtx session2 = (SessionCtx)service.createSubcontext (":session2");
LoginServer login_server2 = (LoginServer)session2.activate ("etc/login");
Login login2 = new Login (login_server2);
login2.authenticate (user, password, null);

// Activate one Hello object in each session
Hello hello1 = (Hello)session1.activate (objectName);
Hello hello2 = (Hello)session2.activate (objectName);
```

クライアント認証のための証明書の使用

証明書を使用してクライアント認証を行うには、クライアントから証明書または証明書の連鎖をサーバーに送信する必要があります。サーバーでは、クライアントがクライアント自身であり信頼されているクライアントであることが検証されます。

注意： 証明書、Trustpoint および秘密鍵はすべて、base-64 でコード化する必要があります。

クライアントが証明書による認証を行うように設定するには、次のいずれかの方法を実行します。

- ファイルで証明書を指定
- 個々の JNDI プロパティで証明書を指定
- AuroraCertificateManager を使用して証明書を指定

ファイルで証明書を指定

ユーザー証明書、発行者証明書、証明書の完全な連鎖、暗号化された秘密鍵および Trustpoint を含むファイルを設定できます。ファイルを一度作成すると、クライアントが接続のハンドシェイク時にそのファイルを使用してクライアント認証を行うように指定できます。

1. クライアント証明書ファイルの作成 — このファイルは、Wallet Manager のエクスポート機能を使用して作成できます。Oracle Wallet Manager には、このファイルを作成するオプションがあります。ファイルの作成を要求する前に、Wallet Manager を使用して Wallet を作成する必要があります。

有効な Wallet を作成した後で、Wallet Manager を起動して次の操作を実行します。

- メニュー・バーのプルダウン・リストから、「Operations」>「Export Wallet」を選択します。
- ファイル名フィールドに、作成する証明書ファイルの名前を入力します。

これにより、追加したすべての証明書、鍵、および Trustpoint を含む、base-64 でコード化されたファイルが Wallet 内に作成されます。Wallet の作成方法の詳細は、『Oracle8i Advanced Security 管理者ガイド』を参照してください。

2. 接続用のクライアント証明書ファイルの指定 — クライアント・コード内で、SECURITY_AUTHENTICATION プロパティを ServiceCtx.SSL_CLIENT_AUTH に設定します。クライアント認証を行うサーバーに、適切な証明書と Trustpoint を提供します。次のように、JNDI プロパティにファイル名と復号化キーを指定します。

値	JNDI プロパティの設定
証明書ファイルの名前	SECURITY_PRINCIPAL
秘密鍵を復号化するためのキー	SECURITY_CREDENTIAL

次のコードは、クライアント証明書ファイルを定義するための JNDI プロパティの設定方法の例です。

```
Hashtable env = new Hashtable();
env.put(javax.naming.Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put(javax.naming.Context.SECURITY_PRINCIPAL, <filename>);
env.put(javax.naming.Context.SECURITY_CREDENTIAL, <decrypting_key>);
env.put(javax.naming.Context.SECURITY_AUTHENTICATION,
ServiceCtx.SSL_CLIENT_AUTH);
Context ic = new InitialContext(env);
...
たとえば、復号化キーが welcome12 で、証明書ファイルが credsFile の場合、次の
2 行で JNDI コンテキスト内にこれらの値を指定します。

env.put(Context.SECURITY_CREDENTIALS, "welcome12");
env.put(Context.SECURITY_PRINCIPAL, "credsFile");
```

個々の JNDI プロパティで証明書を指定

証明書、秘密鍵および Trustpoint を JNDI プロパティ内に個々に設定して、これらをプログラム上で提供できます。JNDI プロパティにユーザー証明書、発行者（認証局）の証明書、暗号化された秘密鍵および Trustpoint を設定すると、接続ハンドシェイク時にこれらが使用されて認証が行われます。クライアント側の認証を設定するには、SECURITY_AUTHENTICATION プロパティを serviceCtx.SSL_CLIENT_AUTH に設定します。

注意： JNDI プロパティで設定できるのは、1 つの発行者証明書のみです。

JNDI プロパティに証明書を設定する方法は何でもかまいません。認可情報の値はすべて、コンテキストを初期化する前に設定する必要があります。

次の例では、証明書を静的変数として宣言します。ただし、これは多くの選択肢のうちの 1 つにすぎません。証明書は base-64 でコード化する必要があります。たとえば、次のコードの `testCert_base64` は、base-64 でコード化され、静的変数として宣言されたクライアント証明書です。CA 証明書のその他の変数や秘密鍵などは示しませんが、同様に定義します。

注意： 個々の証明書を静的変数として設定する場合、Oracle8i 用の証明書にはセパレータが含まれません。ただし、Visigenic ORB の証明書を設定する場合（コールバック使用例におけるクライアントのコールバック・オブジェクトの場合と同様）、証明書は各行を識別する「BEGIN CERTIFICATE」と「END CERTIFICATE」で区切る必要があります。これらの文字列のフォーマットの詳細は、Visigenic のドキュメントを参照してください。

```
final private static String testCert_base64 =
    "MIIECejCCAEogAwIBAgICAmowDQYJKoZIhvcNAQEEBQAwazELMAkGA1UEBhMCVVMx" +
    "DzANBgNVBAoTBk9yYWNsZTEoMCYGA1UECzMFRW50ZXJwcm1zZSBBCHBsaWNhdGlv" +
    "biBTZXJ2aWNlczEhMB8GA1UEAxMYRUFTTUUEgQ2VydG1maWNhdGUgU2VydMvYMB4X" +
    "DTk5MDgxNzE2MjIxMloXDTAwMDIxMzE2MjIxMlowgYUxCzAJBgNVBAYTA1VTMRsw" +
    "GQYDVQQKEExJPcmFjbGUgQ29ycG9yYXRpb24xPDA6BgNVBAsUMyoqIFN1Y3VyaXR5" +
    "IFRFRU1RJTkcgQU5EIEVWQUxVQVRJT04gT05MWSB2ZXJzaW9uMiAqKjEhMBkGA1UE" +
    "AxQScSdGZvdEB1cy5vcnFjbGUuY29tMHwwDQYJKoZIhvcNAQEEBQADawAwaAJhANG1" +
    "Kk2K7uOotI/UBYrmTe89LVRrG83Eb0/wY3xwGelkBeEUTw57a26u2M9LZAfmT91" +
    "e8Afksqc4qQW23Sjxyo4ObQK3Kth6y1NJgovBgfMu1YgtDHaSn2VEg8p58g+nwID" +
    "AQABozYwNDARBg1ghkgBhvCAQEEBAMCAAwHwYDVR0jBBgwFoAUDChwEuJfIFXD" +
    "a7tuYNO8bOw1EYwwDQYJKoZIhvcNAQEEBQADgYEARC5rWKge5trqgZ18onldinCg" +
    "Fof6D/qFT9b6Cex5JK3a2dEekg/P/KqDINyifIZL0DV7z/XCK6PQDLwYcVqSSK/m" +
    "487qjdH+zM5X+1DaJ+ROhqOOX54UpiAhAleRmDLT5KuXV6AtAx6Q2mc8k9bzFzwq" +
    "eR3uI+i5Th0dKgxhCZU=\n";

Hashtable env = new Hashtable();
env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.SSL_CLIENT_AUTH);
//decrypting key
env.put(Context.SECURITY_CREDENTIALS, "welcome12");

// you may also set the certificates individually, as shown bellow.
//User certificate
env.put(ServiceCtx.SECURITY_USER_CERT, testCert_base64);
//Certificate Authority's certificate
env.put(ServiceCtx.SECURITY_CA_CERT, caCert_base64);
//Private key
env.put(ServiceCtx.SECURITY_ENCRYPTED_PKEY, encryptedPrivateKey_base64);
// setup the trust point
env.put(ServiceCtx.SECURITY_TRUSTED_CERT, trustedCert);

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

AuroraCertificateManager を使用して証明書を指定

JNDI を使用しない CORBA クライアントでは、AuroraCertificateManager を使用してユーザー証明書、発行者証明書、暗号化された秘密鍵および Trustpoint を設定できます。

AuroraCertificateManager は、アプリケーションの証明書を保持します。接続のための SSL ハンドシェイクで渡す証明書は、SSL 接続を実行する前に設定しておく必要があります。この方法で証明書を設定する必要があるのは、次の条件に該当する場合のみです。

- クライアント側の認証が必要な場合で、JNDI プロパティを使用しない場合、クライアントは AuroraCertificateManager を使用して証明書を設定します。
- サーバーは、コールアウトまたはコールバックを実行する場合は、AuroraCertificateManager によって証明書を設定します。通常のクライアント / サーバー間のハンドシェイクに使用される典型的なサーバー側認証は、データベース Wallet で処理されます。ただし、このサーバーがコールアウトまたはコールバックを実行してクライアントとして動作しようとする場合は、自分自身を識別する証明書を設定する必要がありますが Wallet に含まれているデータベース証明書はこの目的には使用できません。

AuroraCertificateManager クラス

このオブジェクトで提供されるメソッドを使用すると、次の処理を実行できます。

- SSL プロトコルのバージョンを設定できます。デフォルトは Undetermined です。
- 秘密鍵および証明書の連鎖を設定できます。
- クライアント・アプリケーションが証明書の連鎖を提示して自分自身を認証するように要求できます。このメソッドは、サーバーのみが使用できます。

SSLCertificateManager パラメータを指定して ORB.resolve_initial_references メソッドを起動すると、AuroraCertificateManager にナローイングすることができるオブジェクトが戻されます。[例 6-1](#) は、次のメソッドのコード例を示しています。

addTrustedCertificate

このメソッドでは、指定された証明書が信頼できる証明書として追加されます。証明書は DER 方式でコード化されている必要があります。クライアントは、サーバー側の認証のためにこのメソッドを使用して Trustpoint を追加します。

クライアントがサーバーを認証する必要がある場合、サーバーでは証明書の連鎖がクライアントに送信されます。連鎖内のすべての証明書をチェックする必要はありません。たとえば、認証局、企業、部署、グループおよびユーザーの各証明書から構成される連鎖があります。グループが信頼されている場合、階層の連鎖内のグループより上位の証明書は信頼されているので、ユーザーの証明書からチェックを始めて、グループの証明書が取得された時点で連鎖のチェックを停止できます。

構文

```
void addTrustedCertificate(byte[] derCert);
```

パラメータ	説明
derCert	証明書を含む、DER 方式でコード化されたバイト配列

requestClientCertificate

このメソッドは、クライアント・アプリケーションの証明書を必要とするサーバーにより起動されます。このメソッドは、クライアント・アプリケーションでの使用を目的とするものではありません。

注意： requestClientCertificate メソッドは現在のところ必要ありません。SQLNET.ORA および LISTENER.ORA の構成パラメータ SSL_CLIENT_AUTHENTICATION がこの機能を実行するためです。

構文

```
void requestClientCertificate(boolean need);
```

パラメータ	説明
need	TRUE の場合、クライアントは認証用の証明書を送信する必要があります。FALSE の場合は、クライアントから証明書を送信する必要はありません。

setCertificateChain

このメソッドは、クライアント・アプリケーションまたはサーバー・オブジェクトに証明書の連鎖を設定します。このメソッドはクライアントまたはサーバーから起動できます。証明書の連鎖の先頭は常に認証局の証明書です。次に続く各証明書は、先行する証明書の発行者に対する証明書です。連鎖の最後の証明書は、ユーザーまたはプロセス用の証明書です。

構文

```
void setCertificateChain(byte[] [] derCertChain)
```

パラメータ	説明
derCertChain	証明書の連鎖を含むバイト配列

setEncryptedPrivateKey

このメソッドは、クライアント・アプリケーションまたはサーバー・オブジェクトに秘密鍵を設定します。鍵は PKCS5 形式または PKCS8 形式で指定する必要があります。

構文

```
void setEncryptedPrivateKey(byte[] key, String password);
```

パラメータ	説明
key	暗号化された秘密鍵を含むバイト配列
パスワード	秘密鍵を復号化するためのパスワードを含む文字列

setProtocolVersion

このメソッドは、接続に使用する SSL プロトコルのバージョンを設定します。バージョン 2.0 のクライアントからバージョン 3.0 のサーバーに対して、またはバージョン 3.0 のサーバーからバージョン 2.0 のクライアントに対して SSL 接続を確立しようとするとう失敗します。Version_Undetermined を使用することをお勧めします。これを使用すると、使用しているプロトコルのバージョンを問わずピア間で SSL 接続を確立できます。デフォルト値は SSL_Version_Undetermined です。

構文

```
void setProtocolVersion(int protocolVersion);
```

パラメータ	説明
protocolVersion	指定されているプロトコルのバージョン。指定する値は、 <code>oracle.security.SSL.OracleSSLProtocolVersion</code> に定義されています。このクラスでは、次の値が定義されます。 <ul style="list-style-type: none">■ <code>SSL_Version_Undetermined</code>: バージョンは未定です。これは、SSL 2.0 のピアと SSL 3.0 のピアとの接続に使用します。これがデフォルトのバージョンです。■ <code>SSL_Version_3_0_With_2_0_Hello</code>: サポートされていません。■ <code>SSL_Version_3_0</code>: バージョン 3.0 のピアとの接続にのみ使用します。■ <code>SSL_Version_2_0</code>: サポートされていません。

例 6-1 AuroraCertificateManager を使用した SSL セキュリティ情報の設定

次の例では、次の処理を行っています。

1. **AuroraCertificateManager** を取り出します。
2. このクライアントの SSL 情報を初期化します。
 - a. **setCertificateChain** で証明書の連鎖を設定します。
 - b. **addTrustedCertificate** で Trustpoint を設定します。
 - c. **setEncryptedPrivateKey** で秘密鍵を設定します。

```
// Get the certificate manager
AuroraCertificateManager cm = AuroraCertificateManagerHelper.narrow(
orb.resolve_initial_references("AuroraSSLCertificateManager"));

BASE64Decoder decoder = new BASE64Decoder();
byte[] userCert = decoder.decodeBuffer(testCert_base64);
byte[] caCert = decoder.decodeBuffer(caCert_base64);

// Set my certificate chain, ordered from CA to user.
byte[][] certificates = {
    caCert, userCert
};
cm.setCertificateChain(certificates);
cm.addTrustedCertificate(caCert);

// Set my private key.
byte[] encryptedPrivateKey =
decoder.decodeBuffer(encryptedPrivateKey_base64);

cm.setEncryptedPrivateKey(encryptedPrivateKey, "welcome12");
```

サーバー側の認証

サーバーには、そのロールに応じて異なるタイプの認証が必要になります。標準的なクライアント / サーバー環境でデータベースをサーバーとして使用している場合は、サーバー側認証のためにデータベースの Wallet 内で設定されている証明書を使用します。ただし、サーバーを別のオブジェクトへのコールアウトやクライアント上のオブジェクトへのコールバックに使用している場合、そのサーバーはクライアントとして動作しているため、自分自身を識別する証明書が必要です。しかし、サーバーはデータベースのサーバー側認証用に生成された Wallet をこの目的には使用できません。

サーバー・アクティビティ 認証方式	
典型的なクライアント / サーバー	Oracle Wallet Manager により生成されたデータベース Wallet を使用します。
他のオブジェクトへのコールアウト	JNDI プロパティまたは AuroraCurrentManager クラスを使用して、識別証明書を設定します。
クライアント・オブジェクトへのコールバック	AuroraCurrentManager クラスを使用して識別証明書を設定します。

次の項では、これについて詳細に説明します。

- [典型的なクライアント / サーバー](#)
- [セキュリティを使用したコールアウト](#)
- [セキュリティを使用したコールバック](#)

典型的なクライアント / サーバー

サーバー側の認証は、認証のためにサーバーがクライアントに証明書を提供するときに行われます。認証が要求されると、サーバーではクライアントに証明書を提供して自分自身の認証が行われます（サーバー側の認証）。SSL レイヤーでは、接続ハンドシェイク時に両方のピアの認証が行われます。クライアントは、JNDI プロパティに任意の SSL_* 値を設定してサーバー側の認証を要求します。JNDI の値の詳細は、6-8 ページの「[認証のための JNDI の使用](#)」を参照してください。

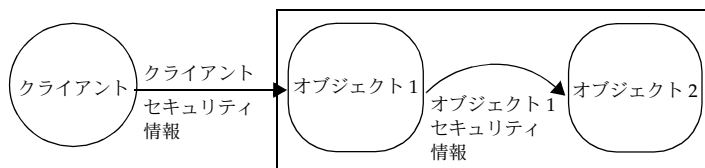
サーバー側の認証を行うには、Wallet Manager を使用して、適切な証明書を持つデータベース Wallet を設定する必要があります。Wallet の作成方法の詳細は、『Oracle8i Advanced Security 管理者ガイド』を参照してください。

注意： クライアントが Trustpoint と突きあわせてサーバーを検証する必要がある場合、またはサーバーを許可する必要がある場合は、クライアント側の責任で Trustpoint の設定や、認可を与えるためのサーバー証明書の解析を実行します。詳細は、6-26 ページの「認可」を参照してください。

セキュリティを使用したコールアウト

コールアウトは、データベースにロードされた Java オブジェクトが他の Java オブジェクト内のメソッドを起動するときに発生します。クライアントからの元のコールに一定レベルのセキュリティ、つまり、証明書またはユーザー名 / パスワードによるセキュリティが必要だった場合は、サーバー・オブジェクトも第 2 のサーバー・オブジェクト上でメソッドを起動する前に、自分自身に関して同一レベルのセキュリティ情報を提供する必要があります。

図 6-2 セキュリティが必要なサーバー・コールアウト

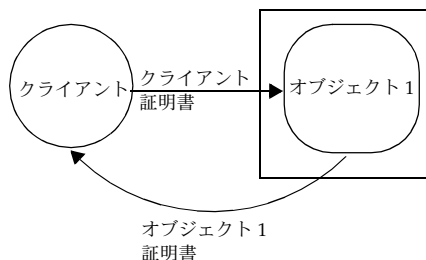


- ユーザー名 / パスワード: クライアントが認証用のユーザー名 / パスワードの組合せをデータベースに送信する場合、サーバー・オブジェクトも自身のユーザー名 / パスワードの組合せを第 2 のオブジェクトに送信する必要があります。サーバー・オブジェクトがクライアントのユーザー名 / パスワードの組合せを転送することはできず、自身の情報を提供する必要があります。ユーザー名 / パスワードの組合せは、クライアントと同じ方法で設定できます。詳細は、6-9 ページの「ユーザー名とパスワードを使用したクライアント側の認証」を参照してください。
- クライアントベース: また、クライアントが認証用の証明書を送信する場合は、サーバー・オブジェクトも同様に証明書を送信する必要があります。さらに、サーバーは、自分自身の証明書を作成して送信する必要があり、クライアントの証明書を認証用に転送することはできません。サーバー・オブジェクトの証明書は、適切な JNDI プロパティまたは `AuroraCertificateManager` を使用して設定できます。6-12 ページの「クライアント認証のための証明書の使用」を参照してください。

セキュリティを使用したコールバック

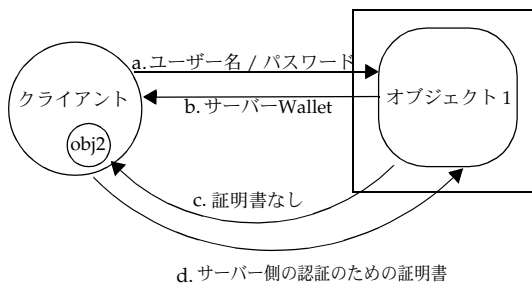
コールバックは、クライアントがサーバー・オブジェクトに、クライアント上に存在するオブジェクトのオブジェクト参照を渡す場合に発生します。図 6-3 のように、サーバー・オブジェクトはオブジェクト参照を受け取ってメソッドを起動します。これにより、サーバーから効率的にコールアウトすることができ、クライアントにあるオブジェクトがコールバックされます。詳細は、2-28 ページの「デバッグ手法」を参照してください。

図 6-3 セキュリティが必要なサーバー・コールアウト



コールバックに使用できるセキュリティのタイプは、SSL による証明書ベースのセキュリティです。SSL セキュリティをコールバックに追加する場合は、次の 2 つの状況のどちらかが考えられます。

1. サーバー側の認証のみ



- a. クライアントは、自分自身を証明書で認証する必要はありません。ただし、ユーザー名 / パスワードの組合せを使用して、引き続き自分自身をデータベースに対して認証する必要があります。
- b. SSL では常にサーバー側の認証が必要なため、サーバーはデータベース Wallet に含まれる証明書を提供し、自分自身をクライアントに対して認証します。

- c. サーバーがクライアントへコールバックする場合は、クライアントとして動作するため、認証用の証明書を提供する必要はありません。
- d. コール側オブジェクトはクライアントに含まれていますが、コールバック使用例ではサーバー・オブジェクトです。したがって、サーバー側の認証ルールが適用されるので、コールバック・オブジェクトは証明書を提供して自分自身を認証する必要があります。

例 6-2 サーバー側の認証のみを使用するコールバック・コード

次のコードは、前述のステップ (a) および (b) を実行するクライアント・コードを示しています。このクライアント・コードの前半では、自分自身をデータベースに対して認証するユーザー名とパスワードを設定しています。次に、サーバー・オブジェクトを取得します。ただし、サーバーのメソッドを起動する前に、コードの後半で証明書を設定し、BOA を初期化し、コールバック・オブジェクトをインスタンス化して、クライアントのコールバック・オブジェクトを設定しています。最後に、サーバー・メソッドが起動されます。

```
public static void main (String[] args) throws Exception {
    String serviceURL = args [0];
    String objectName = args [1];
    String user = args [2];
    String password = args [3];

    //set up username/password for authentication to database. Set up
    //security to be SSL_LOGIN - login authentication for client and server-side
    //authentication.
    Hashtable env = new Hashtable ();
    env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
    env.put (Context.SECURITY_PRINCIPAL, user);
    env.put (Context.SECURITY_CREDENTIALS, password);
    env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.SSL_LOGIN);
    Context ic = new InitialContext (env);

    // Get the server object before preparing the client object.
    // You have to do it in this order to get the ORB initialized correctly
    Server server = (Server)ic.lookup (serviceURL + objectName);

    // Create the client object and export it to the ORB in the client
    // First, set up the ORB properties for the callback object
    java.util.Properties props = new java.util.Properties();
    props.put("ORBservices", "oracle.aurora.ssl");

    // Initialize the ORB.
    com.visigenic.vbroker.orb.ORB orb = (com.visigenic.vbroker.orb.ORB)
        oracle.aurora.jndi.orb_dep.Orb.init(args, props);
}
```

```
// Get the certificate manager
AuroraCertificateManager certificateManager =
    AuroraCertificateManagerHelper.narrow(
        orb.resolve_initial_references("AuroraSSLCertificateManager"));

// Set up client callback certificate chain, ordered from user to CA.
byte[] testCert = new byte[testCert_base64.length()];
testCert_base64.getBytes(0, testCert_base64.length(), testCert, 0);

byte[] caCert = new byte[caCert_base64.length()];
caCert_base64.getBytes(0, caCert_base64.length(), caCert, 0);

// Set my certificate chain, ordered from user to CA.
byte[][] certificates = {
    testCert, caCert
};
certificateManager.setCertificateChain(certificates);

// Set client callback object's private key.
byte[] encryptedPrivateKey = new byte[encryptedPrivateKey_base64.length()];
encryptedPrivateKey_base64.getBytes(0, encryptedPrivateKey_base64.length(),
    encryptedPrivateKey, 0);

certificateManager.setEncryptedPrivateKey(encryptedPrivateKey, "welcome12");

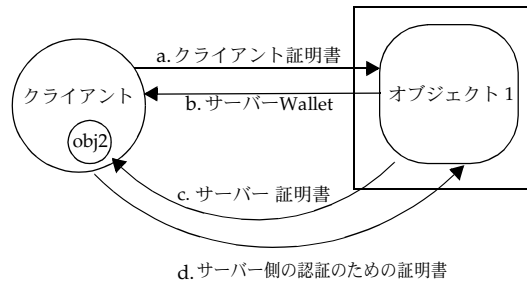
// Initialize the BOA with SSL
org.omg.CORBA.BOA boa = orb.BOA_init("AuroraSSLSession", null);

//Instantiate the client callback object
ClientImpl client = new ClientImpl ();

//register callback object with BOA
boa.obj_is_ready (client);

// Invoke the server method, passing the client to call us back
System.out.println (server.hello (client));
}
}
```

2. クライアント側とサーバー側の認証



- a. クライアントは、自分自身を証明書で認証する必要があります。
- b. SSL では常にサーバー側の認証が必要なため、サーバーはデータベース Wallet に含まれる証明書を提供し、自分自身をクライアントに対して認証します。
- c. サーバーがクライアントへコールバックする場合は、クライアントとして動作するため、認証用に自分自身の証明書を提供する必要があります。
- d. コール側オブジェクトはクライアントに含まれていますが、コールバック使用例ではサーバー・オブジェクトです。したがって、サーバー側の認証ルールが保持されるので、コールバック・オブジェクトは証明書を提供して自分自身を認証する必要があります。

この例は例 6-2 に示したクライアント・コードと同じですが、ただ 1 つ異なるのは、クライアントがユーザー名とパスワードではなく証明書を提供することです。

クライアント側の認証が要求されており、サーバーがクライアントとして動作しているため、サーバー・コードでは、コールバック・オブジェクトを起動する前に、自身の証明書を識別する設定を行います。サーバーは、自身の証明書を作成して送信する必要があります。クライアントの証明書を転送して認証することはできません。サーバー・オブジェクトの証明書は、6-12 ページの「[クライアント認証のための証明書の使用](#)」の記述にあるように、適切な JNDI プロパティまたは `AuroraCertificateManager` を使用して設定します。

例 6-3 クライアント側の認証を使用するコールバックでのサーバー・コード

後述のサーバー・コードでは、次が実行されます。

1. `init` メソッドを起動して `Oracle8i ORB` の参照を取得します。
2. `AuroraCertificateManager` を取り出します。
3. `AuroraCertificateManager` メソッドを介して証明書とキーを設定します。

4. クライアントのコールバック・メソッド hello を起動します。

```
public String hello (Client client) {
    BASE64Decoder decoder = new BASE64Decoder();
    com.visigenic.vbroker.orb.ORB orb = (com.visigenic.vbroker.orb.ORB)
        oracle.aurora.jndi.orb_dep.Orb.init();

    try {
        // Get the certificate manager
        AuroraCertificateManager cm = AuroraCertificateManagerHelper.narrow(
            orb.resolve_initial_references("AuroraSSLCertificateManager"));

        byte[] userCert = decoder.decodeBuffer(testCert_base64);
        byte[] caCert = decoder.decodeBuffer(caCert_base64);

        // Set my certificate chain, ordered from CA to user.
        byte[] [] certificates = { caCert, userCert };
        cm.setCertificateChain(certificates);

        // Set my private key.
        byte[] encryptedPrivateKey =
            decoder.decodeBuffer(encryptedPrivateKey_base64);

        cm.setEncryptedPrivateKey(encryptedPrivateKey, "welcome12");

    } catch (Exception e) {
        e.printStackTrace();
        throw new org.omg.CORBA.INITIALIZE( "Couldn't initialize SSL context");
    }

    return "I Called back and got: " + client.helloBack ();
}
```

認可

SSL レイヤーでは、接続ハンドシェイク時にピアの認証が実行されます。ハンドシェイクが終了すると、ピアが自分自身であると認証されたことになります。また、サーバーは Oracle Wallet 内に Trustpoint を指定しているので、サーバー上の SSL アダプタによりクライアントが許可されます。ただし、クライアントはサーバーに対して許可の範囲を指定できます。

- クライアントは、Trustpoint を設定して、サーバーを認可するように SSL レイヤーに指示できます。
- クライアントは、サーバー証明書の連鎖を抽出して解析して、サーバーを認可できます。

トラスト・ポイントの設定

サーバーには、インストールされている Oracle Wallet 内に Trustpoint が自動的に確立されています。Wallet 内の Trustpoint は、クライアント証明書の検証に使用します。ただし、クライアントがサーバー証明書を特定の Trustpoint と突きあわせて検証する必要がある場合は、次のようにクライアントの Trustpoint を設定できます。

- サーバー側オブジェクトでの認証が必要な場合、クライアントとなるサーバー側オブジェクトには証明書のセットがありません。したがって、サーバー証明書を検証するために、クライアントとなるサーバー側オブジェクトは JNDI を使用して 1 つの Trustpoint を設定するか、または純粋な CORBA アプリケーション（JNDI を使用しないアプリケーション）の場合は `AuroraCertificateManager.addTrustedCertificate` メソッドを使用して Trustpoint を追加します。JNDI を介して単一の Trustpoint を設定する方法は、[例 6-4](#) を参照してください。
- クライアント側での認証が必要な場合については、クライアントにはすでに証明書が設定されていることもあります。したがって、クライアントは、証明書を含むファイルに Trustpoint を追加するか、JNDI を使用して 1 つの Trustpoint を追加するか、または CORBA アプリケーション（JNDI を使用しないアプリケーション）の場合は `AuroraCertificateManager.addTrustedCertificate` メソッドを使用して Trustpoint を追加できます。

クライアントが Trustpoint を設定しない場合には、認可が拒否されることはありません。この場合、Oracle8i JVM はクライアントがサーバーを信頼していると認識します。

例 6-4 Trustpoint の検証

次の例は、クライアントが JNDI を使用して Trustpoint を設定する方法を示します。JNDI の `SECURITY_TRUSTED_CERT` プロパティには、1 つの証明書のみが指定できます。

```
// setup the trust point
env.put(ServiceCtx.SECURITY_TRUSTED_CERT, trustedCert);
```

サーバー証明書の連鎖の解析

クライアントは、証明書を取り出して認可チェックを実行します。以前は、1 つの発行者証明書をとり出すことができました。現在では、発行者証明書の連鎖全体を取り出します。必要な情報を取得するために、証明書の連鎖を解析する必要があります。連鎖の解析には、`AuroraCurrent` オブジェクトを使用できます。

注意： 第3章「IIOP アプリケーションの構成」で説明しているように、データベースとリスナーを、SSL を使用できるように構成する必要があります。

注意： JDK 1.1 の証明書クラスは、`javax.security.cert` に含まれていました。JDK 1.2 では、これらのクラスが `java.security.cert` に移動しています。

AuroraCurrent には、証明書の連鎖を取り出し、管理するための3つのメソッドがあります。証明書の連鎖の作成および解析には、X509Cert クラスのメソッドを使用できます。このクラスは、Sun Microsystems の JDK のドキュメントを参照してください。X509Cert クラスにおける証明書の連鎖の操作方法は、JDK 1.1 と Java 2 とでは異なります。

AuroraCurrent クラスのメソッドは次のとおりです。

- `getPeerDERCertChain` — ピアの証明書の連鎖を取得します。これにより、アプリケーションのメソッドへのアクセスを許可されているピアであることを検証できます。
- `getNegotiatedProtocolVersion` — バージョン番号を検証するために、接続で使用している SSL プロトコルのバージョンを取得します。
- `getNegotiatedCipherSuite` — 暗号化が目的を果たすために十分に効果があることを検証するために、接続を介して渡されるメッセージの暗号化に使用する cipher suite を取得します。

ハンドシェイクが発生すると、プロトコルのバージョンと使用する暗号化のタイプがネゴシエートされます。暗号化のタイプは完全な暗号化または制限された暗号化です。これは、米国の法律上の制限に準拠しています。ハンドシェイクが完了すると、AuroraCurrent はネゴシエーションで解決された内容を取り出せます。

AuroraCurrent クラス

次に、AuroraCurrent クラスに含まれるメソッドを説明します。これらのメソッドのコード例は、例 6-5 を参照してください。

getNegotiatedCipherSuite

このメソッドは、ピアとのハンドシェイクでネゴシエートされた暗号化のタイプを取得します。

構文

```
String getNegotiatedCipherSuite(org.omg.CORBA.Object peer);
```

パラメータ	説明
peer	このピアから、ネゴシエートされた暗号を取得します。

戻り値

次のいずれかの値を持つ文字列を戻します。

米国から輸出可能な暗号方式：

- SSL_RSA_EXPORT_WITH_RC4_40_MD5
- SSL_RSA_EXPORT_WITH_DES40_CBC_SHA
- SSL_DH_anon_EXPORT_WITH_RC4_40_MD5
- SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA
- SSL_RSA_WITH_NULL_SHA
- SSL_RSA_WITH_NULL_MD5

米国内でのみ使用可能な暗号方式：

- SSL_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_RSA_WITH_RC4_128_SHA
- SSL_RSA_WITH_RC4_128_MD5
- SSL_RSA_WITH_DES_CBC_SHA
- SSL_DH_anon_WITH_3DES_EDE_CBC_SHA
- SSL_DH_anon_WITH_RC4_128_MD5
- SSL_DH_anon_WITH_DES_CBC_SH

getPeerDERCertificateChain

このメソッドは、ピアの証明書の連鎖を取得します。連鎖を取り出した後、アプリケーションに対する許可をピアに与えるために連鎖内の証明書を解析できます。

構文

```
byte [] [] getPeerDERCertificateChain(org.omg.CORBA.Object peer);
```

パラメータ	説明
peer	このピアから、証明書の連鎖を取得します。

戻り値

証明書の連鎖を含むバイト配列

getNegotiatedProtocolVersion

このメソッドは、ピアの SSL プロトコルのネゴシエートされたバージョンを取得します。

構文

String getNegotiatedProtocolVersion(org.omg.CORBA.Object peer);

パラメータ	説明
peer	このピアから、ネゴシエートされたプロトコル・バージョンを取得します。

戻り値

次のいずれかの値を持つ文字列を戻します。

- SSL_Version_Undetermined
- SSL_Version_3_0

例 6-5 認可のためのピアの SSL 情報の取出し

次の例は、AuroraCurrent オブジェクトを使用して証明書情報を取り出すことによりピアに認可を与える方法を示します。

1. **AuroraCurrent** オブジェクトを取り出すには、**AuroraSSLCurrent** を引数に指定して ORB.resolve_initial_references メソッドを起動します。
2. **AuroraCurrent** のメソッド (**getNegotiatedCipherSuite**、**getNegotiatedProtocolVersion** および **getPeerDERCertChain**) を使用して、ピアから SSL 情報を取り出します。
3. ピアに認可を付与します。証明書の連鎖に基づいてピアに認可を与えることができます。

注意： この例では、証明書連鎖の解析に Java 2 に固有の x509Certificate クラスのメソッドを使用しています。Java 1.1 を使用している場合は、Java 1.1 に固有の x509Certificate クラスのメソッドを使用する必要があります。

```

static boolean verifyPeerCert(org.omg.CORBA.Object obj) throws Exception
{
    org.omg.CORBA.ORB orb = oracle.aurora.jndi.orb_dep.Orb.init();

    // Get the SSL current
    AuroraCurrent current = AuroraCurrentHelper.narrow
        (orb.resolve_initial_references("AuroraSSLCurrent"));

    // Check the cipher
    System.out.println("Negotiated Cipher: " +
        current.getNegotiatedCipherSuite(obj));
    // Check the protocol version
    System.out.println("Protocol Version: " +
        current.getNegotiatedProtocolVersion(obj));
    // Check the peer's certificate
    System.out.println("Peer's certificate chain : ");
    byte [] [] certChain = current.getPeerDERCertChain(obj);

    //Parse through the certificate chain using the X509Certificate methods
    System.out.println("length : " + certChain.length);
    System.out.println("Certificates: ");
    CertificateFactory cf = CertificateFactory.getInstance("X.509");

    //For each certificate in the chain
    for(int i = 0; i < certChain.length; i++) {
        ByteArrayInputStream bais = new ByteArrayInputStream(certChain[i]);
        Certificate xcrt = cf.generateCertificate(bais);
        System.out.println(xcrt);
        if(xcrt instanceof X509Certificate)
        {
            X509Certificate x509Cert = (X509Certificate)xcrt;
            String globalUser = x509Cert.getSubjectDN().getName();
            System.out.println("DN out of the cert : " + globalUser);
        }
    }

    return true;
}

```

注意： x509Certificate クラスは、Java 2 に固有のクラスです。詳細は、Sun Microsystems のドキュメントを参照してください。また、javax.net.ssl は、javadoc も参照してください。

トランザクション操作

Oracle8i では、Enterprise JavaBeans でのトランザクション管理に Java Transaction API (JTA) 1.0.1 が使用されます。JTA には、Bean 管理とコンテナ管理のトランザクション用に、次の機能が用意されています。

- Bean 管理のトランザクションは、Bean 実装内でプログラムによりデマークートされます。トランザクションは、アプリケーションにより全面的に制御されます。
- コンテナ管理のトランザクションは、コンテナにより制御されます。つまり、コンテナはクライアントのトランザクションに参加するか、ディプロイメント・ディスクリプタ内で定義されているアプリケーション用のトランザクションを開始し、Bean の完了時に終了します。実装では、トランザクション管理用のコードを提供する必要はありません。

この章では、JTA に関する実務知識があることを前提としています。大部分は、Sun Microsystems の JTA 仕様と Oracle の JTA 実装の違いについて、例を中心にして説明します。Sun Microsystems の JTA 仕様については、<http://www.javasoft.com> を参照してください。

- トランザクションの概要
- JTA のサーバー側デマークーション
- JTA のクライアント側デマークーション
- 2 フェーズ・コミット・エンジンの構成
- DataSource オブジェクトの動的作成
- トランザクションのタイムアウト設定
- JDBC の制限事項

トランザクションの概要

トランザクションにより、単一アプリケーション内で複数のデータベースに加えられる変更が1単位の作業として管理されます。つまり、1つ以上のデータベース内のデータを管理するアプリケーションがある場合に、すべてのデータベース内のすべての変更がトランザクション内で管理されていれば、同時にコミットされることを保証できます。

トランザクションは、次のような ACID プロパティで記述されます。

- アトミック (Atomic) : 変更に失敗した場合に、トランザクション内で行われたデータベースへの変更がすべてロールバックされます。
- 一貫性 (Consistent) : トランザクションの結果によりデータベースはある一貫した状態から他の一貫した状態に移行します。
- 孤立 (Isolated) : トランザクション内の中間ステップはデータベースの他のユーザーから参照できません。
- 持続性 (Durable) : トランザクションが完了 (コミットまたはロールバック) した後は、その結果がデータベース内で持続します。

Sun Microsystems が指定している JTA 実装は、JDBC 2.0 仕様と XA アーキテクチャに大きく依存しています。その結果、すべてのデータベース間でトランザクションが完全に管理されることを保証するために、アプリケーションには複雑な要件が課されます。Sun Microsystems は、Java Transaction API (JTA) 1.0.1 および JDBC 2.0 を <http://www.javasoft.com> で公開しています。

Oracle8i 環境で JTA を使用する場合は、次のとおりです。

- [グローバル・トランザクションとローカル・トランザクション](#)
- [トランザクションのデマークート](#)
- [トランザクション・コンテキストの伝播](#)
- [2 フェーズ・コミット](#)
- [リソースの確保](#)
- [JTA の制限事項](#)

グローバル・トランザクションとローカル・トランザクション

アプリケーションから JDBC などを使用してデータベースに接続されるときには、トランザクションを作成します。トランザクションには単一データベースのみが関与し、そのデータベースに対するすべての更新はその終了時にコミットされる場合、これは、ローカル・トランザクションと呼ばれます。

グローバル・トランザクションでは複雑な管理オブジェクトのセットが関与し、トランザクションに関与するすべてのオブジェクトとデータベースは管理オブジェクトにより追跡されます。このグローバル・トランザクション・オブジェクト、つまり TransactionManager

と Transaction により、グローバル・トランザクションに関与するオブジェクトとリソースがすべて追跡されます。トランザクションの終了時には、TransactionManager および Transaction オブジェクトにより、すべてのデータベース変更が同時に自動的にコミットされることが確認されます。

トランザクションのデマーケート

トランザクションは境界（デマーケーション）を持つものであると言われます。これは、明確な開始点と明確な終了点を持つことを意味します。たとえば、SQL*Plus などの対話式ツールでは、各 SQL DML 文がまだトランザクションの一部になっていなければ、暗黙的に新しいトランザクションが開始されます。COMMIT または ROLLBACK 文が発行されるとトランザクションは終了します。

EJB のトランザクションは、トランザクションの開始側と、明示的にデマーケートされるか暗黙的にデマーケートされるかに応じて分類されます。

- トランザクションの開始側 — 分散オブジェクト・アプリケーションでは、開始側がクライアントかサーバーかに応じて、トランザクションが異なる方法でデマーケートされます。トランザクションの開始側によって、そのトランザクションがクライアント側デマーケートなのかサーバー側デマーケートなのか決まります。詳細は、7-16 ページの「[JTA のクライアント側デマーケーション](#)」および 7-13 ページの「[JTA のサーバー側デマーケーション](#)」を参照してください。
- 明示的または暗黙的デマーケーション
 - * 明示的デマーケーションは、クライアントまたは Bean 管理のトランザクションの Bean により、適切な begin または commit メソッドが実行され、トランザクションがプログラムでデマーケートされることを意味します。明示的デマーケーションの詳細は、7-16 ページの「[JTA のクライアント側デマーケーション](#)」および 7-13 ページの「[JTA のサーバー側デマーケーション](#)」を参照してください。
 - * 暗黙的デマーケーションは、コンテナ管理のトランザクションの Bean のディプロイメント・ディスクリプタ内で指定されます。コンテナは、ディプロイメント・ディスクリプタ内で指定される Bean の構成に応じて、トランザクションを開始し、終了します。暗黙的デマーケーションの詳細は、7-13 ページの「[JTA のサーバー側デマーケーション](#)」を参照してください。

注意： また、トランザクションを開始するクライアントまたはオブジェクトは、トランザクションをコミットまたはロールバックで終了する必要があります。ただし、開始側でトランザクションを元のメソッドとは異なるメソッド中で終了することは可能です。たとえば、クライアントがトランザクションを開始し、サーバー・オブジェクトにコールアウトする場合は、起動したメソッドからの戻り後にトランザクションを終了する必要があります。起動されたサーバー・オブジェクトはトランザクションを終了できません。

コンテナ管理または Bean 管理のトランザクション

Enterprise JavaBeans では、Bean がそれ自体でトランザクションをデマークートして管理するか、またはコンテナがトランザクションのデマークートと管理を行う必要があるかを指定できます。

コンテナ管理のトランザクション

Bean がコンテナ管理のトランザクションの Bean として指定されている場合、その Bean のメソッドにトランザクションの実装は不要です。すべてのトランザクションのロジックは、EJB ディプロイメント・ディスクリプタに指定されたトランザクション属性に基づいて、コンテナにより実行されます。詳細は、次の項を参照してください。

- A-18 ページの「[トランザクションの定義](#)」。コンテナ管理のトランザクションの EJB ディプロイメント・ディスクリプタの仕様部について説明しています。
- 7-6 ページの「[コンテナ管理のトランザクションの Bean へのトランザクション・コンテキストの伝播](#)」。トランザクション属性に指定されたディプロイメント・ディスクリプタに応じて、コンテナにより実行される操作について説明しています。

Bean 管理のトランザクション

Bean が Bean 管理のトランザクションの Bean として指定されている場合は、次の操作を行う必要があります。

- Bean 自体ですべてのグローバル・トランザクションを開始します。他のオブジェクト、つまりクライアントやサーバーからトランザクション・コンテキストを受け入れることはできません。したがって、Bean 管理のトランザクションの Bean は、他のオブジェクトのグローバル・トランザクションには参加できません。起動側オブジェクトがトランザクションに参加する場合、そのトランザクションは、Bean 管理のトランザクションの Bean の実行中は一時停止されます。その後、Bean 管理のトランザクションの Bean から制御が戻されると、起動側オブジェクトのトランザクションが再開されます。
- Bean が、グローバル・トランザクションに参加する他のオブジェクトを必要とする場合は、伝播されるトランザクション・コンテキストを受け入れるトランザクション属性を指定して、コンテナ管理の Bean を起動する必要があります。
- Bean は、7-7 ページの「[リソースの確保](#)」の説明と同じ方法で、リソースを自動的に確保できます。

トランザクション・コンテキストの伝播

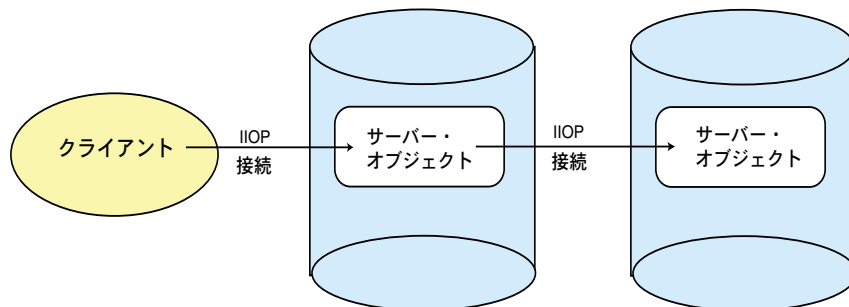
クライアントまたはサーバー・インスタンス内でトランザクションを開始すると、JTA によりトランザクション・マネージャ内でトランザクションの開始者が示されます。トランザクションに複数のオブジェクトやリソースが関与する場合は、これらのすべてのオブジェクトとリソースがトランザクション・マネージャによりトランザクション内で追跡され、各エンティティに対するトランザクションが管理されます。

オブジェクトが他のオブジェクトをコールすると、起動されたオブジェクトがトランザクションに含まれるように、JTA によりトランザクション・コンテキストが起動されたオブジェクトに伝播されます。起動されたオブジェクトをグローバル・トランザクションに含めるには、トランザクション・コンテキストの伝播が必要です。

図 7-1 のように、クライアントがグローバル・トランザクションを開始し、データベース内のサーバー・オブジェクトをコールすると、そのサーバー・オブジェクトへとトランザクション・コンテキストが伝播されます。サーバー・オブジェクトがトランザクションをサポートしていれば、このオブジェクトはグローバル・トランザクションに参加するものとしてトランザクション・マネージャに連結されます。このサーバー・オブジェクトが同じデータベースまたはリモート・データベース内で別のサーバー・オブジェクトを起動すると、そのオブジェクトにもトランザクション・コンテキストが伝播されます。これにより、グローバル・トランザクションへの参加がサポートされるオブジェクトすべてが、トランザクション・マネージャにより追跡されることが保証されます。

注意： すべてのサーバー・オブジェクトはデータベースにロードされるため、JTA では、サーバー・オブジェクトとともに、グローバル・トランザクションに含まれるリソースとして、データベースが自動的に確保されます。

図 7-1 IIOP を介したオブジェクトへの接続



コンテナ管理のトランザクションの Bean へのトランザクション・コンテキストの伝播

コンテナ管理のトランザクションの Bean の場合、EJB ディプロイメント・ディスクリプタ内のトランザクション属性の定義により、グローバル・トランザクション・コンテキストがサーバー・オブジェクトに伝播されるかどうかが決まります。次の表は、各トランザクション属性と、各種のサーバー・オブジェクトおよびリソースに発生する動作を示しています。

表 7-1 コンテナ管理のトランザクションの Bean に対するトランザクション属性の影響

配置トランザクション属性	クライアントのトランザクションのデマーケーション	ターゲット・サーバー・オブジェクトまたはリソース（データベース）の動作
NotSupported	トランザクションを開始しません。	トランザクションは開始されません。
	トランザクションを開始します。	実行者のトランザクションは、Bean の実行中は一時停止され、制御が実行者に戻ると再開されます。
Required	トランザクションを開始しません。	新規トランザクションが開始されます。
	トランザクションを開始します。	実行者のトランザクション・コンテキストが伝播されます。サーバー・オブジェクトとローカル・リソースがトランザクションに参加します。
Supports	トランザクションを開始しません。	トランザクションは開始されません。
	トランザクションを開始します。	実行者のトランザクション・コンテキストが伝播されます。サーバー・オブジェクトとローカル・リソースがトランザクションに参加します。
RequiresNew	トランザクションを開始しません。	新規トランザクションが開始されます。
	トランザクションを開始します。	実行者のトランザクションが一時停止されます。新規トランザクションが開始され、コミットされてから、制御が実行者に戻されます。制御が戻されると、実行者のトランザクションが再開されます。

表 7-1 コンテナ管理のトランザクションの Bean に対するトランザクション属性の影響（続き）

配置トランザクション属性	クライアントのトランザクションのデマーケーション	ターゲット・サーバー・オブジェクトまたはリソース（データベース）の動作
Mandatory	トランザクションを開始しません。	エラーが戻されます。このオブジェクトには、トランザクション・コンテキストが必要です。
	トランザクションを開始します。	実行者のトランザクション・コンテキストが伝播されます。サーバー・オブジェクトとローカル・リソースがトランザクションに参加します。
Never	トランザクションを開始しません。	トランザクションは開始されません。
	トランザクションを開始します。	エラーが戻されます。このオブジェクトは、クライアントやサーバーなど、トランザクションに参加するオブジェクトからはコールできません。

リソースの確保

データベースなど、グローバル・トランザクションで管理対象となる各リソースを確保する必要があります。Oracle8i の JTA 実装では、グローバル・トランザクションのコンテキスト内でデータベースへの JDBC 接続をオープンすると、すべてのデータベースが自動的に確保されます。トランザクション内でデータベースへの JDBC 接続をオープンする方法の詳細は、『Oracle8i 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。

JTA では、次のいずれかが発生するとデータベース・リソースが自動的に確保されます。

ローカル・データベースの確保

次の JDBC メソッドは、ローカル・データベースにアクセスできるように、ローカル・トランザクション内で使用されます。処理がグローバル・トランザクション内で行われるようにするには、グローバル・トランザクションの開始後に、これらのメソッドおよび後続する SQL 文を実行します。

表 7-2 データベースの確保に使用される JDBC のメソッド

取得メソッド	説明
<pre>OracleDriver(). defaultConnection()</pre>	ローカル接続を取得するための JDBC 2.0 以前のメソッド。データベース表の更新に #sqlj マクロを使用する場合は、このマクロにより defaultConnection メソッドが起動されますので注意してください。

表 7-2 データベースの確保に使用される JDBC のメソッド (続き)

取得メソッド	説明
<code>DriverManager.getConnection</code> (<code>"jdbc:oracle:kprb:"</code>)	ローカル接続を取得するための JDBC 2.0 以前のメソッド。
<code>DataSource.getConnection</code> (<code>"jdbc:oracle:kprb:"</code>)	ローカル・データベースへの接続を取得する JDBC 2.0 のメソッド。

注意： データベースへの接続を取得するには、JDBC 2.0 のメソッドまたは JDBC 2.0 以前のメソッドのどちらか一方のみを使用する必要があります。両方のメソッドを同じ Bean に混在させないでください。

この 2 つの自動確保メソッドの例の詳細は、7-25 ページの「[サーバー側でのリソースの確保](#)」を参照してください。

リモート・データベースの確保

クライアント・オブジェクトもサーバー・オブジェクトも、`DataSource` オブジェクト内の JDBC 2.0 メソッドを介してのみ、グローバル・トランザクション内でデータベースを確保できます。`UserTransaction` オブジェクトの `begin` メソッドの後に、`getConnection` メソッドが起動される必要があります。

トランザクションに複数のデータベースが参加する場合は、Oracle8i データベースを 2 フェーズ・コミット・エンジンとして指定する必要があります。詳細は、7-29 ページの「[2 フェーズ・コミット・エンジンの構成](#)」を参照してください。

2 フェーズ・コミット

グローバル・トランザクションの主な利点の 1 つは、複数のオブジェクトとデータベース・リソースをトランザクション内で 1 単位として管理できることです。グローバル・トランザクションに複数のデータベース・リソースが関与する場合は、2 フェーズ・コミット・エンジン、つまり、トランザクション内の全データベースの変更を管理するように設定された Oracle8i データベースを指定する必要があります。2 フェーズ・コミット・エンジンは、トランザクションの終了時に、すべてのデータベースに対する変更がすべてコミットされるか、すべてロールバックされることを保証します。

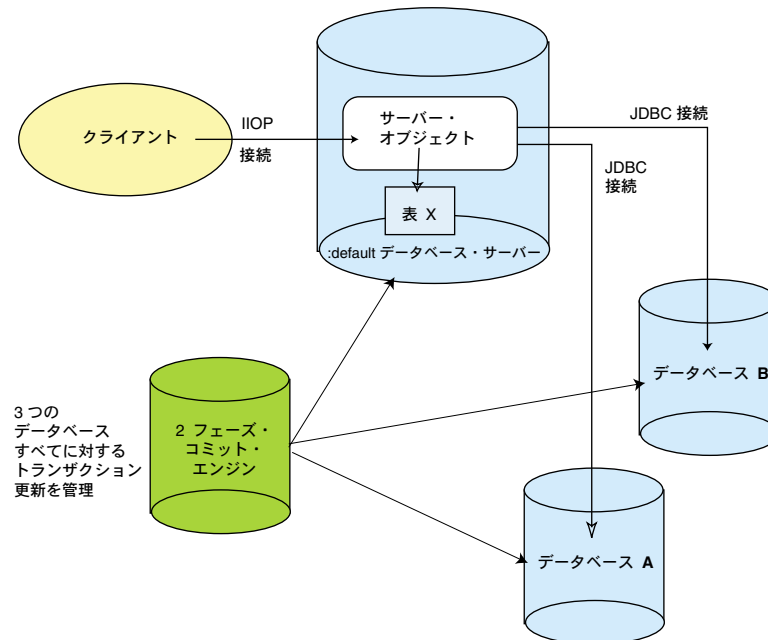
ただし、グローバル・トランザクションに複数のサーバー・オブジェクトが参加していても、データベース・リソースが 1 つのみであれば、2 フェーズ・コミット・エンジンを指定する必要はありません。2 フェーズ・コミット・エンジンは、複数データベースに対する変更を同期するためにのみ必要です。データベースが 1 つのみであれば、トランザクション・マネージャで単一フェーズ・コミットを実行できます。

注意： 2 フェーズ・コミット・エンジンは、どの Oracle8i データベースでもかまいません。サーバー・オブジェクトが存在するデータベースや、トランザクションにまったく参加しないデータベースでも使用できます。2 フェーズ・コミット・エンジンの設定の詳細は、7-29 ページの「[2 フェーズ・コミット・エンジンの構成](#)」を参照してください。

図 7-2 は、グローバル・トランザクションで確保される 3 つのデータベースと、2 フェーズ・コミット・エンジンとして指定された第 4 のデータベースを示しています。ローカル・データベースを含め、すべてのデータベースは、グローバル・トランザクションの開始後に JDBC 接続がオープンされた時点で自動的に確保されます。データベース確保の詳細は、7-7 ページの「[リソースの確保](#)」を参照してください。

グローバル・トランザクションの終了時に、2 フェーズ・コミット・エンジンにより、データベース A、B およびローカル・データベースに加えられた変更が、すべて同時にコミットまたはロールバックされることが保証されます。

図 7-2 グローバル・トランザクションの 2 フェーズ・コミット



JTA の概要

次の項では、トランザクションのデマークートと、トランザクション内でのデータベースの確保の概要がまとめられています。それぞれの詳細は、例を示して説明します。ただし、各表は参考となるポイントを示しています。

表 7-3 トランザクション・オブジェクト取得用の環境設定

ソース	修飾子	環境設定
		設定には、認証情報、ネームスペースの URL および OracleDriver 登録が含まれます。
クライアント	リモート・オブジェクトまたはリモート・データベース接続を取得します。	必要
サーバー	同一セッション内での活性化を使用します。ローカル・オブジェクトまたはローカル・データベース接続を取得します。	不要
	リモート・オブジェクトまたはリモート・データベース接続を取得します。	必要

表 7-4 JDBC 2.0 DataSource と 2.0 以前の JDBC ドライバの使用法の違い

	JDBC 2.0 DataSource	2.0 以前の JDBC ドライバ: OracleDriver または DriverManager
バインドの違い	bindds コマンドを使用して、DataSource をネームスペースにバインドする必要があります。	JDBC オブジェクトのバインドは不要です。
コードの違い 次のように JNDI 検索を実行します。 クライアント	<div>1. 次の環境設定を行います。環境の Hashtable に認証情報とネームスペースの URL を含め、OracleDriver を登録します。</div> <div>2. JNDI 検索を介して、「jdbc_access://」接頭辞を含む DataSource オブジェクトを取得します。</div>	クライアントでは使用できません。
サーバー	<div>次のどちらかを行います。</div> <div>■ 同一セッション内での活性化を使用して DataSource オブジェクトを取得します。環境設定や「jdbc_access://」接頭辞は不要です。</div> <div>■ 前もってディプロイメント・ディスクリプタに設定した環境変数を介してデータベースを検索します。これには、「java:comp/env」接頭辞を使用します。</div>	2.0 以前の JDBC ドライバ (OracleDriver および DriverManager) を使用できるのは、サーバー・オブジェクト内のみです。JDBC 接続の取得は、旧リリースと同じです。

表 7-5 コンテナ管理と Bean 管理のトランザクションの違い

		コンテナ管理のトランザクション	Bean 管理のトランザクション
単一フェーズ・コミット	ディプロイメント・ディスクリプタ	<ul style="list-style-type: none"> ■ <transaction-type> 要素内で、これがコンテナ管理であることを定義します。 ■ XML ディプロイメント・ディスクリプタの <container-transaction> 要素内で、コンテナ管理の属性の型を、表 7-1 の値の 1 つとして設定します。 	<ul style="list-style-type: none"> ■ <transaction-type> 要素内で、これが Bean 管理であることを定義します。
	バインド	<ul style="list-style-type: none"> ■ UserTransaction には、バインドは不要です。UserTransaction オブジェクトは自動的に作成されます。 ■ トランザクションに DataSource オブジェクトを使用する場合は、bindds コマンドを使用してバインドします。ただし、データベース・リンクは不要です。 	<ul style="list-style-type: none"> ■ UserTransaction には、バインドは不要です。UserTransaction オブジェクトは自動的に作成されます。 ■ トランザクションに DataSource オブジェクトを使用する場合は、bindds コマンドを使用してバインドします。ただし、データベース・リンクは不要です。
	実行時	<ul style="list-style-type: none"> ■ UserTransaction を取得しません。UserTransaction はコンテナにより管理されます。 ■ トランザクション内で SQL の DML 文の管理に DataSource オブジェクトを使用する場合は、DataSource を取得します。 	<ul style="list-style-type: none"> ■ SessionCtx の EJB 1.0 の getUserTransaction メソッド、または「java:comp/UserTransaction」文字列を指定した JNDI lookup の EJB 1.1 のメソッドを介して、UserTransaction を取得します。 ■ 実行時には、トランザクションの開始と終了を受け持ちます。 ■ トランザクション内で SQL の DML 文の管理に DataSource オブジェクトを使用する場合は、DataSource を取得します。

表 7-5 コンテナ管理と Bean 管理のトランザクションの違い（続き）

コンテナ管理のトランザクション		Bean 管理のトランザクション
2 フェーズ・コミット	ディプロイメント・ディスクリプタ	要件は、コンテナ管理のトランザクションと同じです。
バインド	<ul style="list-style-type: none">■ UserTransaction オブジェクトを、2 フェーズ・コミット・エンジンの完全修飾データベース・アドレス、そのユーザー名およびパスワードと共にバインドする必要があります。■ 2 フェーズ・コミット・エンジンからのデータベース・リンクに使用されているユーザーは、トランザクションをコミットするには権限が必要です。したがって、このユーザーに「FORCE ANY TRANSACTION」権限が付与されていることを確認してください。■ トランザクションに関与するデータベースごとに、DataSource オブジェクトを、2 フェーズ・コミット・エンジンからそれ自体へのデータベース・リンクとバインドする必要があります。	<ul style="list-style-type: none">■ UserTransaction オブジェクトを、2 フェーズ・コミット・エンジンの完全修飾データベース・アドレス、そのユーザー名およびパスワードと共にバインドする必要があります。■ トランザクションに関与するデータベースごとに、DataSource オブジェクトを、2 フェーズ・コミット・エンジンからそれ自体へのデータベース・リンクとバインドする必要があります。
実行時	<ul style="list-style-type: none">■ UserTransaction を取得しません。UserTransaction はコンテナにより管理されます。■ DataSource を取得します。	<ul style="list-style-type: none">■ SessionCtx の EJB 1.0 の getUserTransaction メソッド、または「java:comp/UserTransaction」文字列を指定した JNDI lookup の EJB 1.1 のメソッドを介して、UserTransaction を取得します。■ 実行時には、トランザクションの開始と終了を受け持ちます。■ トランザクション内で SQL の DML 文の管理に DataSource オブジェクトを使用する場合は、DataSource を取得します。

JTA の制限事項

JTA 仕様のうち、Oracle8i でサポートされない部分は次のとおりです。

ネストしたトランザクション

ネストしたトランザクションは、このリリースではサポートされません。既存のトランザクションをコミットまたはロールバックする前に新しいトランザクションを開始しようとする、トランザクション・サービスで `NotSupportedException` 例外が発生します。

相互運用性

このリリースで提供されるトランザクション・サービスには、他の JTA 実装との相互運用性はありません。

JTA のサーバー側デマーケーション

サーバー側デマーケーションは、コンテナ管理のトランザクションを介して暗黙的に、または Bean 管理のトランザクションを介して明示的に発生させることができます。コンテナ管理のトランザクションは、Bean の EJB ディプロイメント・ディスクリプタ内で指定します。ほとんどの EJB では、アプリケーション内でトランザクションの実装を必要としないため、コンテナ管理のトランザクションが使用されます。

データベースの確保については、コンテナ管理のトランザクションでも Bean 管理のトランザクションでも同じです。

- [コンテナ管理のトランザクション](#)
- [Bean 管理のトランザクション](#)
- [サーバー側でのリソースの確保](#)

コンテナ管理のトランザクション

EJB ディプロイメント・ディスクリプタ内で、Bean のトランザクションをコンテナで管理するように宣言できます。そのためには、`<transaction-type>` 要素内で「Container」を指定し、`<container-transaction>` 要素内でコンテナ管理のトランザクション属性（表 7-1 を参照）を指定する必要があります。詳細は、A-18 ページの「[トランザクションの定義](#)」を参照してください。

Bean インスタンス内のメソッドが起動されると、EJB ディプロイメント・ディスクリプタ内で指定したトランザクション属性に基づいて、コンテナによりグローバル・トランザクションが開始、コミットまたはロールバックされます。各 Bean は、様々なトランザクション属性を持つように指定できます。トランザクション属性では、トランザクションのデマーケーションの処理方法を指定します。これは、Bean は `UserTransaction` オブジェクトを取得せず、そのメソッドも起動しないことを意味します。この操作はコンテナで自動的に実行されます。

コンテナでは、データベース・リソースは確保されません。7-11 ページの表 7-5 に示すように、ネームスペース内に各 DataSource をバインドします。コードでは、7-7 ページの表 7-2 に示すメソッドの 1 つを介してデータベース接続を取得し、各データベースを確保します。コンテナ管理のトランザクションの Bean とデータベース確保の使用例の詳細は、7-29 ページの「[2 フェーズ・コミット・エンジンの構成](#)」を参照してください。

Bean 管理のトランザクション

Bean 管理のトランザクションを使用するかどうかを選択できるのは、セッション Bean のみです。これは、begin、commit および rollback メソッドでトランザクションをデマーケートできるのが、セッション Bean のみであることを意味します。このセッション Bean でトランザクションをプログラムでデマーケートするように指定するには、その EJB ディプロイメント・ディスクリプタ内で次のように指定する必要があります。

```
<enterprise-beans>
  <session>
    . . .
    <transaction-type>Bean</transaction-type>
  </session>
</enterprise-beans>
```

注意： EJB ディプロイメント・ディスクリプタ内でトランザクション管理を定義する方法の詳細は、A-18 ページの「[トランザクションの定義](#)」を参照してください。

セッション Bean の実装中では、クライアントの場合と同じ方法でトランザクションがデマーケートされます。UserTransaction オブジェクトから、begin、commit および rollback メソッドが起動されます。唯一の違いは、UserTransaction オブジェクトの取得方法です。Bean 管理のトランザクションの Bean 内でサーバー側の UserTransaction オブジェクトを取得するには、次の 2 つの方法があります。

- [SessionContext getUserTransaction メソッド](#)
- [JNDI lookup](#)

SessionContext getUserTransaction メソッド

Bean は、setSessionContext メソッドで設定された SessionContext オブジェクトから UserTransaction オブジェクトを取得します。これは、UserTransaction オブジェクトを取得するための EJB 1.0 の方法であり、使用できるのは単一フェーズ・コミット環境のみです。

コンテナにより、`UserTransaction` オブジェクトはすでに取得されています。次の例は、セッション・コンテキストが `ctx` 変数に保存されることを示しています。

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

Bean 実装内で、セッション・コンテキストから `UserTransaction` を取得し、トランザクションを開始します。

```
UserTransaction ut = ctx.getUserTransaction();  
ut.begin();  
...  
ut.commit();
```

JNDI lookup

EJB 1.1 の方法では、JNDI 名「`java:comp/UserTransaction`」を指定して同一セッション内での検索を実行し、`UserTransaction` オブジェクトを取得します。単一フェーズ・コミット環境では、コンテナにより `UserTransaction` オブジェクトが自動的に作成されます。2 フェーズ・コミット環境では、あらかじめ 2 フェーズ・コミット情報（ユーザー名、パスワードおよび 2 フェーズ・コミット URL）を指定して、`UserTransaction` オブジェクトをネームスペースにバインドする必要があります。取得後は、このオブジェクトを使用してグローバル・トランザクションをデマーケートします。

次の例は、ネームスペースから `UserTransaction` オブジェクトを取得する方法を示しています。

```
ic = new InitialContext ( );  
  
// lookup the usertransaction  
UserTransaction ut = (UserTransaction)ic.lookup ("java:comp/UserTransaction");  
ut.begin ( );  
...  
ut.commit();
```

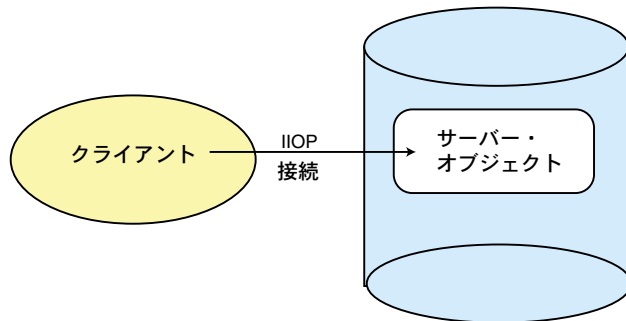
JTA のクライアント側デマーケーション

JTA の場合、クライアント側でデマーケートされるトランザクションは、UserTransaction オブジェクトを介してプログラムで明示的にデマーケートされます。bindut コマンドを使用して、このオブジェクトをネームスペースにバインドする必要があります。クライアント側トランザクションのデマーケーションでは、トランザクションがクライアントにより制御されます。クライアントは UserTransaction begin メソッドを起動してグローバル・トランザクションを開始し、commit または rollback メソッドを起動してトランザクションを終了します。

さらに、クライアントは、認証情報とネームスペースのロケーション URL を使用して、Hashtable を含む環境を常に設定する必要があります。ネームスペースからトランザクション・オブジェクトを取得するときには、OracleDriver も登録する必要があります。

図 7-3 は、サーバー・オブジェクトを起動するクライアントを示しています。この例では、サーバー・オブジェクトはコンテナ管理のトランザクションの Bean です。クライアントは、グローバル・トランザクションを開始してから、Bean を起動します。この Bean はコンテナ管理のトランザクションの Bean であり、トランザクション属性「Supports」が指定されているため、トランザクション・コンテキストはサーバー・オブジェクトを含むように伝播されます。

図 7-3 クライアント側でデマーケートされるグローバル・トランザクション



クライアントがトランザクションをデマーケートするには、次のステップが必要です。

1. ネームスペース・アドレスと認証情報を使用して、Hashtable による環境変数を初期化します。
2. OracleDriver を登録します。
3. クライアントのロジック内でネームスペースから UserTransaction オブジェクトを取得します。どのクライアントから UserTransaction オブジェクトを取得する場合も、URL では JNDI 名の前に「jdbc_access://」接頭辞を付ける必要があります。

4. `UserTransaction.begin()` を使用して、クライアント内でグローバル・トランザクションを開始します。
5. サーバー Bean を取得します。
6. トランザクションに含めるオブジェクト・メソッドを起動します。
7. `UserTransaction.commit()` または `UserTransaction.rollback()` を介してトランザクションを終了します。

例 7-1 は、トランザクション内でサーバー Bean を起動するクライアントを示しています。

例 7-1 クライアント側でデマーケートされるトランザクションによる Employee アプリケーションのクライアント・コード

クライアントを起動する前に、まずネームスペース内に `UserTransaction` オブジェクトをバインドする必要があります。

ネームスペースへの `UserTransaction` オブジェクトのバインド

ネームスペースに `UserTransaction` オブジェクトをバインドするには、`sess_sh` ツールの `bindut` コマンドを使用します。`UserTransaction` オブジェクトを `nsHost` にあるネームスペース内の「`/test/myUT`」という名前にバインドするには、次のように実行します。

```
sess_sh -service jdbc:oracle:thin:@nsHost:5521:ORCL -user SCOTT -password TIGER
& bindut /test/myUT
```

注意： クライアントが `UserTransaction` を取得するには、`bindut` コマンドに指定したのと同じ情報が必要です。

クライアント・アプリケーションの開発

`UserTransaction` オブジェクトのバインド後は、クライアント・コードで `UserTransaction` オブジェクトを取得し、グローバル・トランザクションを開始できます。クライアントはリモート・サイトから `UserTransaction` オブジェクトを取得するため、検索には認証情報、ネームスペースのロケーション、`OracleDriver` 登録および「`jdbc_access://`」接頭辞が必要です。

```
import employee.Employee;
import employee.EmployeeHome;
import employee.EmployeeInfo;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;
import javax.transaction.*;
```

```
public class Client
{
    public static void main (String[] args) throws Exception {

        //Set up the service URL to where the UserTransaction object
        //is bound. Since from the client, the connection to the database
        //where the namespace is located can be communicated with over either
        //a Thin or OCI8 JDBC driver. This example uses a Thin JDBC driver.
        String namespaceURL = "jdbc:oracle:thin:@nsHost:1521:ORCL";

        //User and password are case sensitive.
        String user = "SCOTT";
        String password = "TIGER";

        //1.(a) Authenticate to the database.
        // create InitialContext and initialize for authenticating client
        Hashtable env = new Hashtable ();
        env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
        env.put (Context.SECURITY_PRINCIPAL, user);
        env.put (Context.SECURITY_CREDENTIALS, password);
        env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
        //1.(b) Specify the location of the namespace where the transaction objects
        // are bound.
        env.put (jdbc_accessURLContextFactory.CONNECTION_URL_PROP, namespaceURL);
        Context ic = new InitialContext (env);

        //2. Register a JDBC OracleDriver. Requirement for opening JDBC connection
        // to retrieve UserTransaction object from namespace
        DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());

        //3. Retrieve the UserTransaction object from JNDI namespace
        UserTransaction ut = (UserTransaction)ic.lookup ("jdbc_access://test/myUT");

        //4. Start the transaction
        ut.begin();

        //5. Retrieve the EJB
        // get an handle to the employee_home object
        EmployeeHome employee_home =
            (EmployeeHome)ic.lookup ("sess_iiop://myhost:1521:orcl/test/employee");

        // get an handle to the remote bean
        Employee employee = employee_home.create ();

        //6. Perform bean business logic.
        // get an info of an employee
```

```

EmployeeInfo info = employee.getEmployee ("SCOTT");
System.out.println ("Beginning salary = " + info.salary);

// do work on the info-object
info.salary += (info.salary * 10) / 100;

// call update on the server-side
employee.updateEmployee (info);

//7. End the transaction
//Commit the updated value
ut.commit();
}
}

```

例 7-2 Supports 属性を持つコンテナ管理のトランザクションの Bean

EJB ディプロイメント・ディスクリプタ内で、コンテナ管理のトランザクションのセッション Bean を Supports 属性で指定します。これらのトランザクション仕様部は、`<transaction-type>` および `<container-transaction>` 要素で定義されます。

XML ディプロイメント・ディスクリプタ

```

<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Oracle Corporation//DTD Enterprise JavaBeans 1.1
//EN" "ejb-jar.dtd">
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>test/myEmployee</ejb-name>
      <home>employee.EmployeeHome</home>
      <remote>employee.Employee</remote>
      <ejb-class>employeeServer.EmployeeBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>EmployeeBean</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Supports</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>

```

このセッション Bean では、トランザクションの管理はコンテナにまかされ、Bean は既存のトランザクションをサポートするように指定されているため、トランザクション・コンテキストはクライアントによるメソッドの起動時に Bean に伝播します。Bean 自体には、トランザクションのロジックは含まれておらず、次のように Bean の実装コードのみが含まれています。

注意： この Bean が配置されているデータベースは、自動的に確保されます。したがって、SQLJ 文がデータベースに対して実行され、その変更はグローバル・トランザクションの一部となります。ローカル・データベースの自動確保の詳細は、7-25 ページの「[ローカル・データベースの確保](#)」を参照してください。

```
package employeeServer;

import employee.*;

import javax.ejb.SessionBean;
import javax.ejb.CreateException;
import javax.ejb.SessionContext;
import java.rmi.RemoteException;

import java.sql.SQLException;

public class EmployeeBean implements SessionBean
{
    // Methods of the Employee interface
    public EmployeeInfo getEmployee (String name)
        throws RemoteException, SQLException

    {
        int empno = 0;
        double salary = 0.0;
        #sql { select empno, sal into :empno, :salary from emp
                where ename = :name };

        return new EmployeeInfo (name, empno, salary);
    }

    public void updateEmployee (EmployeeInfo employee)
        throws RemoteException, SQLException
    {
        #sql { update emp set ename = :(employee.name),
                sal = :(employee.salary)
                where empno = :(employee.number) };
        return;
    }
}
```

```

    }

    // Methods of the SessionBean
    public void ejbCreate () throws RemoteException, CreateException {}
    public void ejbRemove () {}
    public void setSessionContext (SessionContext ctx) {}
    public void ejbActivate () {}
    public void ejbPassivate () {}
}

```

データベースを含む JTA のクライアント側デマーケーション

前述の例は、JTA グローバル・トランザクション内でトランザクション・コンテキストがクライアントからサーバー・オブジェクトへとどのように伝播されるかを示していました。サーバー・オブジェクトの実行時には、トランザクションは IIOP トランスポート・レイヤーを介して伝播します。IIOP サーバー・オブジェクトを起動するのみでなく、JDBC 接続を介してデータベースを更新する必要があることもあります。この項では、IIOP サーバー・オブジェクト伝播を使用すると同時に JDBC 接続を使用してデータベースを確保する方法について説明します。

クライアントからのトランザクションにリモート・データベースを含めるには、DataSource オブジェクトを使用する必要があります。このオブジェクトは、JTA の DataSource としてネームスペースにバインドされています。次に、トランザクションの開始後に DataSource オブジェクトの getConnection メソッドを起動すると、データベースがグローバル・トランザクションに含まれます。詳細は、7-7 ページの「[リソースの確保](#)」を参照してください。

クライアント・ランタイムがトランザクションをデマーケートするには、次のステップが必要です。

1. ネームスペース・アドレスと認証情報を使用して、Hashtable 環境を初期化します。
2. OracleDriver を登録します。
3. クライアントのロジック内でネームスペースから UserTransaction オブジェクトを取得します。クライアントから UserTransaction オブジェクトを取得する場合は、URL では JNDI 名の前に「jdbc_access://」接頭辞を付ける必要があります。
4. UserTransaction.begin() を使用して、クライアント内でグローバル・トランザクションを開始します。
5. 次のように、指定したデータベースへの接続をオープンし、トランザクションに含めるデータベース・リソースを確保します。
 - a. クライアントのロジック内でネームスペースから DataSource オブジェクトを取得します。どのクライアントから DataSource オブジェクトを取得する場合も、URL では JNDI 名の前に「jdbc_access://」接頭辞を付ける必要があります。
 - b. DataSource.getConnection メソッドを介してデータベースへの接続をオープンします。

6. Bean 参照を取得します。
7. トランザクションに含めるオブジェクト・メソッドを起動します。
8. 確保されたデータベースに対する SQL DML 文を起動します。
9. `UserTransaction.commit()` または `UserTransaction.rollback()` を介してトランザクションを終了します。

例 7-3 は、トランザクション内で Bean を起動し、単一データベースを確保するクライアントを示しています。

例 7-3 クライアント側でデマーケートされるトランザクションによる Employee アプリケーションのクライアント・コード

クライアントを起動する前に、まずネームスペースに `UserTransaction` オブジェクトと `DataSource` オブジェクトをバインドする必要があります。`UserTransaction` オブジェクトのバインド方法については、7-17 ページの「[ネームスペースへの UserTransaction オブジェクトのバインド](#)」を参照してください。

ネームスペースへの DataSource オブジェクトのバインド

ネームスペースに `DataSource` オブジェクトをバインドするには、`sess_sh` ツールの `bindds` コマンドを使用します。このコマンドの詳細は、『Oracle8i Java Tools リファレンス』を参照してください。

`empHost` データベースでの単一フェーズ・コミット・トランザクション用の `DataSource` オブジェクトを、`nsHost` にあるネームスペース内の「`/test/empDatabase`」という名前にバインドするには、次のように実行します。

```
sess_sh -service jdbc:oracle:thin:@nsHost:5521:ORCL -user SCOTT -password TIGER
& bindds /test/empDatabase -url jdbc:oracle:thin:@empHost:5521:ORCL -dstype jta
```

ネームスペースに `DataSource` オブジェクトをバインドした後は、サーバーはグローバル・トランザクション内でデータベースを確保できます。

注意： 複数のデータベースを使用する場合は、2 フェーズ・コミット用に設定する必要があります。詳細は、7-29 ページの「[2 フェーズ・コミット・エンジンの構成](#)」を参照してください。

クライアント・アプリケーションの開発

次の例は、7-21 ページの「データベースを含む JTA のクライアント側デマーケーション」に示したステップをたどっています。

```
import employee.Employee;
import employee.EmployeeHome;
import employee.EmployeeInfo;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;
import javax.transaction.*;

public class Client
{
    public static void main (String[] args) throws Exception {

        //Set up the service URL to where the UserTransaction object
        //is bound. Since from the client, the connection to the database
        //where the namespace is located can be communicated with over either
        //a Thin or OCI8 JDBC driver. This example uses a Thin JDBC driver.
        String namespaceURL = "jdbc:oracle:thin:@nsHost:1521:ORCL";

        //User and password are case sensitive.
        String user = "SCOTT";
        String password = "TIGER";

        //1.(a) Authenticate to the database.
        // create InitialContext and initialize for authenticating client
        Hashtable env = new Hashtable ();
        env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
        env.put (Context.SECURITY_PRINCIPAL, user);
        env.put (Context.SECURITY_CREDENTIALS, password);
        env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
        //1.(b) Specify the location of the namespace where the transaction objects
        // are bound.
        env.put (JdbcAccessURLContextFactory.CONNECTION_URL_PROP, namespaceURL);
        Context ic = new InitialContext (env);

        //2. Register a JDBC OracleDriver. Required for retrieving UserTransaction
        // and the DataSource objects from namespace over JDBC connection.
        DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());

        //3. Retrieve the UserTransaction object from JNDI namespace
        UserTransaction ut;
        ut = (UserTransaction)ic.lookup ("jdbc_access://test/myUT");
```

```
//4. Start the transaction
ut.begin();

//5.(a) Retrieve the DataSource (that was previously bound with bindds in
// the namespace. After retrieving the DataSource...
// get a connection to a database. You need to provide authentication info
// for a remote database lookup, similar to what you would do from a client.
// In addition, if this was a two-phase commit transaction, you must provide
// the username and password.
DataSource ds = (DataSource)ic.lookup ("jdbc_access://test/empDB");

// 5.(b) Get connection to the database through DataSource.getConnection
// in this case, the database requires the same username and password as
// set in the environment.
Connection conn = ds.getConnection ();

//6. Retrieve the EJB
// get an handle to the employee_home object
EmployeeHome employee_home =
    (EmployeeHome)ic.lookup (serviceURL + objectName);

// get an handle to the remote bean
Employee employee = employee_home.create ();

//7. (a) Perform bean business logic.
// get an info of an employee
EmployeeInfo info = employee.getEmployee ("SCOTT");
System.out.println ("Beginning salary = " + info.salary);

// do work on the info-object
info.salary += (info.salary * 10) / 100;

// call update on the server-side
employee.updateEmployee (info);

//7. (b) Execute SQL statements against the enlisted database.
Statement stmt = conn.createStatement ();
int cnt = stmt.executeUpdate ("insert into my_tab values (39304)");

//8. Close the database connection.
conn.close ();

//9. End the transaction
//Commit the updated value
ut.commit();
    }
}
```

ディプロイメント・ディスクリプタと Bean の実装コードは、[例 7-2](#) の場合と同じです。

サーバー側でのリソースの確保

Bean インスタンスで Bean 管理のトランザクションを使用するか、コンテナ管理のトランザクションを使用するかに関係なく、Bean がアクセスするデータベースを、グローバル・トランザクションに含まれるように確保する必要があります。詳細は、7-7 ページの「[リソースの確保](#)」および 7-22 ページの「[ネームスペースへの DataSource オブジェクトのバインド](#)」を参照してください。

バインド後は、トランザクションの開始後にデータベースを確保できます。

- [ローカル・データベースの確保](#)
- [リモート Oracle8i データベースの確保](#)

ローカル・データベースの確保

Bean は Oracle8i データベースに配置されているため、このデータベースがトランザクション内でコンテナにより自動的に確保されます。このデータベースは、ローカル・データベースと呼ばれます。したがって、このデータベースに対して SQL 文を実行すると、結果はトランザクションと同時にコミットされます。

- **SQLJ** 『Oracle8i SQLJ 開発者ガイドおよびリファレンス』に記載されているように、オブジェクトが実行中のデータベースへの暗黙的なローカル接続が提供されます。SQLJ 文中で実行される文は、すべてローカル・データベースに対して実行されます。ただし、文の結果をトランザクションの一部にするには、オープンしているグローバル・トランザクション内で SQLJ 文を実行する必要があります。つまり、`UserTransaction.begin` メソッドの起動後に SQLJ 文を実行します。ただし、SQLJ 文中ではトランザクションをコミットしないでください。トランザクションのコミットは、`UserTransaction.commit` メソッドでのみ発生させる必要があります。
- **JDBC** [表 7-6](#) に示すメソッドを実行すると、ローカル接続を作成できます。

表 7-6 ローカル・データベース取得メソッド

取得メソッド	説明
<code>OracleDriver().defaultConnection()</code>	例 7-6 を参照してください。
<code>DriverManager.getConnection("jdbc:oracle:kprb:")</code>	例 7-5 を参照してください。
<code>DataSource.getConnection("jdbc:oracle:kprb:")</code>	例 7-4 を参照してください。

前述のメソッドと後続の SQL 文は、常にグローバル・トランザクションの開始後に実行してください。

例 7-4 DataSource.getConnection メソッドの例

これらの例は、RequiresNew 属性を持つコンテナ管理のトランザクションの Bean を示しています。そのため、この Bean が最初に起動されるときに、グローバル・トランザクションが初期化されます。ローカル接続は、トランザクション内でデータベースを確保する DataSource.getConnection メソッドを介して取得されます。SQL 文は、ローカル・データベースに対して実行されます。これらの文は、Bean 終了時にコンテナによりグローバル・トランザクションがコミットされると、それと同時にコミットされます。

```
public EmpRecord query (int empNumber) throws SQLException, RemoteException
{
    //Retrieving the UserTransaction and DataSource using in-session activation
    Context ic = new InitialContext ( );

    //Retrieve the DataSource using in-session activation
    DataSource ds = (DataSource) ic.lookup("/test/myDB");

    //Retrieve the local connection object to the local database
    Connection conn = ds.getConnection ();

    //prepare and execute a sql statement against the local database.
    PreparedStatement ps =
        conn.prepareStatement ("select ename, sal from emp where empno = ?");
    try {
        ps.setInt (1, empNumber);
        ResultSet rset = ps.executeQuery ();
        if (!rset.next ())
            throw new RemoteException ("no employee with ID " + empNumber);
        return new EmpRecord (rset.getString (1), empNumber, rset.getFloat (2));
    } finally {
        ps.close();
    }

    //close the database connection
    conn.close();
}
```

例 7-5 DriverManager.getConnection メソッドの例

次の例は例 7-4 と同じですが、データベース取得メソッドが異なります。次のようにして接続を取得します。

```
//Retrieve the local connection object to the local database
Connection conn =
    DriverManager.getConnection ("jdbc:oracle:kprb:");
```

例 7-6 OracleDriver().defaultConnection メソッドの例

次の例は例 7-4 と同じですが、データベース取得メソッドが異なります。次のようにして接続を取得します。

```
//Retrieve the local connection object to the local database
Connection conn =
    new oracle.jdbc.driver.OracleDriver().defaultConnection ();
```

リモート Oracle8i データベースの確保

トランザクションに含める必要のあるサーバーから、リモートの Oracle8i データベースにアクセスする場合は、グローバル・トランザクションの開始後にそのデータベースへの接続をオープンする必要があります。

注意： 現時点で、Oracle JTA 実装では、グローバル・トランザクションに Oracle 以外のデータベースを含めることはサポートされていません。

例 7-7 単一フェーズ・トランザクションでのデータベースの確保

次の例では、Bean 管理のトランザクションの Bean 内でグローバル・トランザクションにデータベースを確保しています。

次の例はコンテナ管理のトランザクションの Bean のため、UserTransaction を取得しません。ただし、JDBC 2.0 接続を開始するために DataSource を取得します。

```
package employeeServer;

import employee.*;

import javax.ejb.SessionBean;
import javax.ejb.CreateException;
import javax.ejb.SessionContext;
import java.rmi.RemoteException;
import java.sql.SQLException;
import javax.transaction.*;

public class EmployeeBean implements SessionBean
```

```
{
    SessionContext ctx;

    // Methods of the Employee interface
    public EmployeeInfo getEmployee (String name)
        throws RemoteException, SQLException
    {
        // get a connection to the local database. If this was a two-phase commit
        // transaction, you would provide the username and password
        // for the 2pc engine
        DataSource ds = (DataSource)ic.lookup ("/test/myDS");

        // get connection to the local database through DataSource.getConnection
        Connection conn = ds.getConnection ();

        //perform your SQL against the database.
        //prepare and execute a sql statement.
        //retrieve the employee's selected benefits
        PreparedStatement ps =
            conn.prepareStatement ("update emp set ename = :(employee.name),
                                   sal = :(employee.salary) where empno = :(employee.number)");
        try {
            ps.setInt (1, empNumber);
            ResultSet rset = ps.executeQuery ();
            if (!rset.next ())
                throw new RemoteException ("no employee with ID " + empNumber);
            return new EmpRecord (rset.getString (1), empNumber, rset.getFloat (2));
        } finally {
            ps.close();
        }

        //close the connection
        conn.close();

        return new EmployeeInfo (name, empno, salary);
    }

    // Methods of the SessionBean
    public void ejbCreate () throws RemoteException, CreateException {}
    public void ejbRemove() {}
    public void setSessionContext (SessionContext ctx) {
        this.ctx = ctx;
    }
    public void ejbActivate () {}
    public void ejbPassivate () {}
}
```

2 フェーズ・コミット・エンジンの構成

トランザクションに複数のデータベースが参加する場合は、その全データベースに対する変更をすべて管理するために、2 フェーズ・コミット・エンジンを指定する必要があります。2 フェーズ・コミット・エンジンは、トランザクションの終了時にすべてのデータベースに接続し、含まれる全データベースに対するすべての更新のコミットまたはロールバックを管理します。したがって、2 フェーズ・コミット・エンジンには、トランザクションに含まれる各データベースへのデータベース・リンクに対するアクセス権が必要です。

2 フェーズ・コミット用に構成するには、システム管理者が次の操作を行う必要があります。

1. Oracle8i データベースを 2 フェーズ・コミット・エンジンとして指定します。
2. 2 フェーズ・コミット・エンジンから、グローバル・トランザクションに参加する可能性のある各データベースへのデータベース・リンクを構成します。この操作が必要なのは、2 フェーズ・コミット・エンジンがトランザクションの終了時に各データベースと通信するためです。
3. 個々のデータベースの DataSource をネームスペースにバインドするときに、bindds の -dblink オプションでそのデータベースへのデータベース・リンク名を指定します。

```
bindds /test/empDatabase -url jdbc:oracle:thin:@empHost:5521:ORCL  
-dstype jta -dblink 2pcToEmp.oracle.com
```

4. UserTransaction をネームスペースにバインドするときに、2 フェーズ・コミット・エンジンの完全修飾データベース・アドレス、ユーザー名およびパスワードを指定します。

```
bindut /test/myUT -url sess_iiop://dbsun.mycompany.com:2481:ORCL  
-user SCOTT -password TIGER
```

注意： 2 フェーズ・コミット・エンジンからのデータベース・リンクに使用されるユーザーが、アプリケーションからの接続に使用されるユーザーのトランザクションをコミットできるように、FORCE ANY TRANSACTION 権限を持っていることを確かめてください。

```
GRANT FORCE ANY TRANSACTION TO SCOTT
```

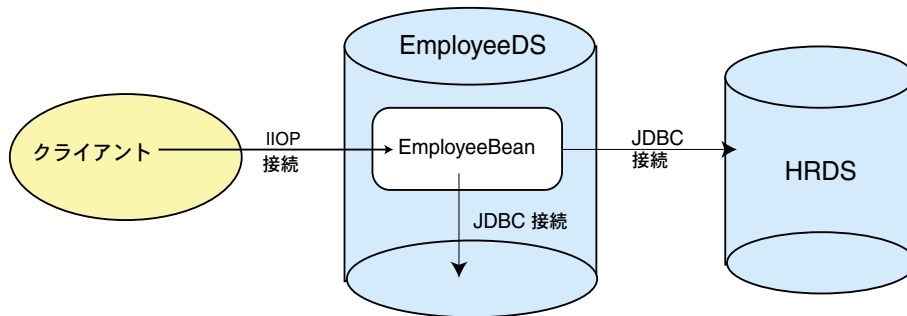
5. Oracle 固有のディプロイメント・ディスクリプタ中に、2 フェーズ・コミット・エンジンの JNDI 名を記述します。この名前は、<transaction-manager> 要素に挿入する必要があります。この要素は、トランザクションが開始される Bean 内で定義するだけで済みます。詳細は、A-26 ページの「[トランザクションの 2 フェーズ・コミット・エンジンの定義](#)」を参照してください。

この構成がすべて完了すると、アプリケーションは次の点で単一フェーズ・コミットの使用例とは異なるものになります。

- クライアントからトランザクションをデマークートする場合に、UserTransaction のバインドでユーザー名とパスワードを指定しないという選択ができます。その場合は、かわりに、UserTransaction の検索時に使用する InitialContext の Hashtable 内でユーザー名とパスワードを指定します。
- DataSource.getConnection を介してのみデータベース接続を取得できます。このオブジェクトはデータベース・リンクと共にバインドされており、2 フェーズ・コミット・エンジンではデータベース内の変更の管理にデータベース・リンクが使用されるため、データベース・リソースの確保には、DataSource を使用する必要があります。

図 7-4 は、EmployeeBean を起動するクライアントを示しています。この Bean により、そのローカル・データベースへの接続とリモート・データベースへの接続がオープンされます。

図 7-4 グローバル・トランザクションへのリモート Oracle8i データベースの挿入



例 7-8 クライアント・コード

このクライアントと例 7-1 のクライアントの唯一の違いは、このクライアントではトランザクションを開始しないことです。クライアントが EmployeeBean を起動すると、コンテナによりトランザクションが開始されます。

```
package client;
import common.*;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

public class Client
{
    public static void main (String[] args) throws Exception
```

```
{
    if (args.length != 6)
    {
        System.out.println ("usage: Client serviceURL jdbcURL objectName " +
            "user password");
        System.exit (1);
    }
    String serviceURL = args [0];
    String jdbcURL = args [1];
    String objectName = args [2];
    String user = args [3];
    String password = args [4];

    // set up the initial context
    Hashtable env = new Hashtable ();
    env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
    env.put (Context.SECURITY_PRINCIPAL, user);
    env.put (Context.SECURITY_CREDENTIALS, password);
    env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
    Context ic = new InitialContext (env);

    // lookup the home and remote interfaces fro the employee
    EmployeeHome home = (EmployeeHome)ic.lookup (serviceURL + objectName);
    EmployeeRemote remote = home.create ();
    Employee employee = null;

    // retrieve info about this employee in this session
    employee = remote.getEmployeeForUpdate ("SCOTT");
    System.out.println ("Beginning salary for " + employee.name + " is " +
        employee.salary);

    // increase salary
    // employee.salary += 0.1 * employee.salary;
    employee.salary += 100;

    // update the infomation in the transaction
    remote.updateEmployee (employee);

    // Get and print the info in the transaction
    employee = remote.getEmployee ("SCOTT");
    System.out.println ("End salary for " + employee.name + " is " +
        employee.salary);
}
}
```

例 7-9 EmployeeBean の EJB ディプロイメント・ディスクリプタ

EmployeeBean は、RequiresNew 属性を持つコンテナ管理のトランザクションの Bean として指定されています。また、環境変数として JDBC 2.0 および 2.0 以前のオブジェクトが定義されています。

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems Inc.//DTD Enterprise JavaBeans 1.1
//EN" "ejb-jar.dtd">
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>test/EmployeeBean</ejb-name>
      <home>common.EmployeeHome</home>
      <remote>common.EmployeeRemote</remote>
      <ejb-class>server.EmployeeBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
      <env-entry>
        <env-entry-name>EmployeeBean.KPRB_URL</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>jdbc:oracle:kprb:</env-entry-value>
      </env-entry>
      <env-entry>
        <env-entry-name>EmployeeBean.JDBC_DRIVER_NAME</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>oracle.jdbc.driver.OracleDriver</env-entry-value>
      </env-entry>
      <env-entry>
        <env-entry-name>EmployeeBean.JDBC_URL</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>
          JDBC_URL=jdbc:oracle:thin:@localhost:5521:jis1
        </env-entry-value>
      </env-entry>
      <env-entry>
        <env-entry-name>EmployeeBean.DB_LINK</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>
          LOOP2.REGRESS.RDBMS.DEV.US.ORACLE.COM
        </env-entry-value>
      </env-entry>
      <resource-ref>
        <res-ref-name>jdbc/EmployeeDS</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Application</res-auth>
      </resource-ref>
    </session>
  </enterprise-beans>
</ejb-jar>
```

```

        <res-ref-name>jdbc/HRDS</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Application</res-auth>
    </resource-ref>
</session>
</enterprise-beans>
<assembly-descriptor>
    <security-role>
        <description>no description</description>
        <role-name>PUBLIC</role-name>
    </security-role>
    <method-permission>
        <description>no description</description>
        <role-name>PUBLIC</role-name>
        <method>
            <ejb-name>test/EmployeeBean</ejb-name>
            <method-name>*</method-name>
        </method>
    </method-permission>
    <container-transaction>
        <description>no description</description>
        <method>
            <ejb-name>test/EmployeeBean</ejb-name>
            <method-name>*</method-name>
        </method>
        <trans-attribute>RequiresNew</trans-attribute>
    </container-transaction>
</assembly-descriptor>
</ejb-jar>

```

例 7-10 EmployeeBean の Oracle 固有のディプロイメント・ディスクリプタ

JDBC オブジェクトを表す環境変数は、JNDI バインド名にマップされます。2 フェーズ・コミット・エンジンの URL でバインドされている UserTransaction は、<transaction-manager> 要素で指定されています。

```

<?xml version="1.0"?>
<!DOCTYPE oracle-descriptor PUBLIC "-//Sun Microsystems Inc.//DTD Enterprise JavaBeans 1.1//EN" "oracle-ejb-jar.dtd">
<oracle-descriptor>
    <mappings>
        <ejb-mapping>
            <ejb-name>EmployeeBean</ejb-name>
            <jndi-name>test/EmployeeBean</jndi-name>
        </ejb-mapping>
        <security-role-mapping>
            <security-role>

```

```
<description>just a role</description>
<role-name>SECURITY_CLERK</role-name>
</security-role>
<oracle-role>CLERK</oracle-role>
</security-role-mapping>
<resource-ref-mapping>
  <res-ref-name>jdbc/EmployeeDS</res-ref-name>
  <jndi-name>test/DataSource/empDS</jndi-name>
</resource-ref-mapping>
<resource-ref-mapping>
  <res-ref-name>jdbc/HRDS</res-ref-name>
  <jndi-name>test/DataSource/hrDS</jndi-name>
</resource-ref-mapping>
<transaction-manager>
  <jndi-name>test/UserTransaction/testut</jndi-name>
</transaction-manager>
</mappings>
</oracle-descriptor>
```

例 7-11 Bean 管理のトランザクションを使用する EmployeeBean

コンテナ管理のトランザクションの EmployeeBean 中で、ローカルとリモートの Oracle8i データベースへの接続を取得します。

```
package server;

import common.*;
import java.sql.*;
import java.util.Hashtable;
import java.rmi.RemoteException;
import javax.sql.*;
import javax.naming.*;
import javax.ejb.*;
import oracle.aurora.jndi.sess_iiop.*;
import sqlj.runtime.ref.DefaultContext;
import javax.transaction.*;

import oracle.aurora.transaction.xa.OracleJTADDataSource;

public class EmployeeBean implements SessionBean
{
    SessionContext ctx;
    Context ic = null; // inSession Lookup Context
    Connection localConn = null;
    Connection remoteConn = null;
    DefaultContext remoteCtx = null;
    Employee remoteEmployee = null;
    String remoteEmpName = "SMITH";
```

```

private void setupDSConnections ()
    throws SQLException, RemoteException
{
    try {
        if (ic == null)
            ic = new InitialContext ();

        //retrieve the remote connection
        remoteConn = getRemoteDSConnection ();
        if (remoteConn == null)
            System.out.println ("remote connection is NULL");
        else
            System.out.println ("got remote connection");

        //setup the context for issuing SQLJ against the remote database
        remoteCtx = new DefaultContext (remoteConn);

        //retrieve the local connection
        localConn = getLocalDSConnection ();
        if (localConn == null)
            System.out.println ("local connection is NULL");
        else
            System.out.println ("got local connection");
    } catch (NamingException e) {
        e.printStackTrace ();
        throw new SQLException ("setupDSConnection failed:" + e.toString ());
    } catch (SQLException e) {
        e.printStackTrace ();
        throw new SQLException ("setupDSConnection failed:" + e.toString ());
    }
}

private Connection getLocalDSConnection ()
    throws SQLException, SQLException
{
    try {
        System.out.println ("looking up EmployeeDS in JNDI");
        // get a connection to the local DB using an environment variable
        //specified in the deployment descriptor
        DataSource localDS = (DataSource)ic.lookup
            ("java:comp/env/jdbc/EmployeeDS");
        System.out.println ("getLocalDSConnection: " +
            ((OracleJTADDataSource)localDS).getURL());

        // get a connectoin to the local DB
        return localDS.getConnection ();
    }
}

```

```
    } catch (NamingException e) {
        e.printStackTrace ();
        throw new SQLException ("getLocalDataSource failed:" + e.toString ());
    }
}

private Connection getRemoteDataSourceConnection ()
    throws SQLException, SQLException
{
    try {
        // get a connection to the remote DB
        System.out.println ("looking up HRDS in JNDI");
        //retrieve the remote database DataSource HRDS using the environment
        //variable specified in the deployment descriptor
        DataSource remoteDS = (DataSource)ic.lookup ("java:comp/env/jdbc/HRDS");
        System.out.println ("getRemoteDataSourceConnection: "+
            ((OracleJTADDataSource)remoteDS).getURL());

        // get a connection to the remote DB passing in the username and
        // password for this database. (Otherwise, it would have had to be
        // specified in the Context environment.
        return remoteDS.getConnection ("scott", "tiger");
    } catch (NamingException e) {
        e.printStackTrace ();
        throw new SQLException ("getRemoteDataSourceConnection failed:" + e.toString ());
    }
}

public void updateEmployee (Employee employee)
    throws SQLException, RemoteException
{
    try {
        setupDataSources ();

        //issue SQL DML statements against the local database
        #sql { update emp set ename = :(employee.name), sal = :(employee.salary)
            where empno = :(employee.number) };

        remoteEmployee.salary += 200;

        //issue SQL DML statements against the remote database
        #sql [remoteCtx] { update emp set ename = :(remoteEmployee.name),
            sal = :(remoteEmployee.salary)
            where empno = :(remoteEmployee.number) };

        //close both database connections
    }
}
```

```

        localConn.close();
        remoteConn.close ();
    } catch (SQLException e) {
        e.printStackTrace ();
        throw new SQLError ("updateEmployee failed: " + e.toString ());
    }
}

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

public void ejbCreate() throws CreateException, RemoteException {}
public void ejbActivate() {}
public void ejbPassivate() {}
public void ejbRemove() {}
}

```

DataSource オブジェクトの動的作成

不特定のデータベース・リソースに対して使用可能なネームスペース内の単一の DataSource オブジェクトのみをバインドする場合は、次の処理を行う必要があります。

1. URL、ホスト、ポート、SID またはドライバ・タイプを指定せずに DataSource をバインドします。この場合、次のように、`-dstype jta` オプションのみを指定して `bindds` ツールを実行します。

```

sess_sh -service jdbc:oracle:thin:@nsHost:5521:ORCL -user SCOTT -password
TIGER
& bindds /test/empDatabase -dstype jta

```

2. コード内で DataSource を検索します。検索の実行時には、戻されたオブジェクトを DataSource ではなく OracleJTADDataSource にキャストする必要があります。DataSource クラスの Oracle 固有のバージョンである OracleJTADDataSource クラスには、DataSource のプロパティを設定するメソッドが含まれています。
3. 次のプロパティを設定します。
 - OracleJTADDataSource.setURL メソッドを使用して URL を設定します。
 - URL を設定しなかった場合は、OracleJTADDataSource のメソッドである setURL、setDatabaseName、setPortNumber および setDriverType を使用して、ホスト、ポート、SID およびドライバのタイプを設定します。

- 2 フェーズ・コミット・エンジンを使用する場合は、
OracleJTADDataSource.setDBLink メソッドでデータベース・リンクを設定します。
 - 2 フェーズ・コミット・エンジン用に認証情報を提供する必要がある場合は、ユーザー名とパスワードを設定します。この情報は、初期コンテキスト環境で指定することもできます。また、getConnection メソッドで指定することもできます。また、OracleJTADDataSource メソッドで設定する場合は、setUser および setPassword メソッドを使用することもできます。
4. 他の例で示したように OracleJTADDataSource.getConnection メソッドを介して接続を取得します。

例 7-12 汎用 DataSource の取得

次の例では、同一セッション内検索を使用してネームスペースから汎用にバインドされた DataSource を取得し、すべての関連フィールドを初期化しています。

```
//retrieve an in-session generic DataSource object
OracleJTADDataSource ds = (OracleJTADDataSource)ic.lookup ("/test/genericDS");

//set all relevant properties for my database
//URL is for a local database so use the KPRB URL
ds.setURL ("jdbc:oracle:kprb:");
//Used in two-phase commit, so provide the fully qualified database link that
//was created from the two-phase commit engine to this database
ds.setDBLink("localDB.oracle.com");

//Finally, retrieve a connection to the local database using the DataSource
Connection conn = ds.getConnection ();
```

トランザクションのタイムアウト設定

グローバル・トランザクションの場合、アイドル・タイムアウトは自動的に 60 秒に設定されます。トランザクションに連結されているオブジェクトがタイムアウト制限を超えてもアイドル状態になっていると、そのトランザクションはロールバックされます。デフォルト値以外のタイムアウトを初期化するには、トランザクションの開始前に setTransactionTimeout メソッドによってタイムアウト値を秒単位で設定します。トランザクションの開始後にタイムアウト値を変更しても、そのトランザクション自身には影響しません。次の例では、トランザクションの開始前にタイムアウトを 2 分（120 秒）に設定しています。

```
//create the initial context
InitialContext ic = new InitialContext ( );

//retrieve the UserTransaction object
ut = (UserTransaction)ic.lookup ("/test/myUT");
```

```
//set the timeout value to 2 minutes
ut.setTimeout (120);

//begin the transaction
ut.begin

//Update employee table with new employees
updateEmployees(emp, newEmp);

//end the transaction.
ut.commit ();
```

セッション同期インタフェースの使用

セッション Bean は、コンテナにより Bean のトランザクション状態が通知されるようにしたい場合、オプションでセッション同期インタフェースを実装できます。
javax.ejb.SessionSynchronization インタフェースに次のメソッドが定義されています。

afterBegin

```
public abstract void afterBegin() throws RemoteException
```

afterBegin() メソッドは、新しいトランザクションが開始したこと、およびインスタンス上の後続のメソッドは、そのトランザクションのコンテキスト内で起動されることをセッション Bean インスタンスに通知します。

Bean はこのメソッドを使用してデータベースからデータを読み込み、Bean のフィールドにデータをキャッシュするといったことができます。

このメソッドは、そのトランザクション・コンテキスト内で実行されます。

beforeCompletion

```
public abstract void beforeCompletion() throws RemoteException
```

コンテナは、beforeCompletion() メソッドをコールして、トランザクションを直後にコミットする予定であることをセッション Bean に通知します。たとえば、このメソッドを実装して、キャッシュされているデータをデータベースに書き込むことができます。

afterCompletion

```
public abstract void afterCompletion(boolean committed) throws RemoteException
```

コンテナは、afterCompletion() をコールして、トランザクション・コミット・プロトコルが完了したことをセッション Bean に通知します。パラメータを使用して、トランザクションがコミットされたか、あるいはロールバックされたかが Bean に知らされます。

このメソッドは、トランザクション・コンテキストなしで実行されます。

例 7-13 SessionSynchron の例

コンテナが各トランザクションの前後に Bean で実装したロジックを起動するには、Bean は SessionSynchron インタフェースを実装する必要があります。

```
package employeeServer;

import employee.*;

import javax.ejb.SessionBean;
import javax.ejb.CreateException;
import javax.ejb.SessionContext;
import java.rmi.RemoteException;

import java.sql.SQLException;

public class EmployeeBean implements SessionBean implements SessionSynchron
{
    // Methods of the Employee interface
    public EmployeeInfo getEmployee (String name)
        throws RemoteException, SQLException
    {
        int empno = 0;
        double salary = 0.0;
        #sql { select empno, sal into :empno, :salary from emp
              where ename = :name };

        return new EmployeeInfo (name, empno, salary);
    }

    public void updateEmployee (EmployeeInfo employee)
        throws RemoteException, SQLException
    {
        #sql { update emp set ename = :(employee.name),
              sal = :(employee.salary)
              where empno = :(employee.number) };
        return;
    }
}
```

```
// Methods of the SessionBean
public void ejbCreate () throws RemoteException, CreateException {}
public void ejbRemove () {}
public void setSessionContext (SessionContext ctx) {}
public void ejbActivate () {}
public void ejbPassivate () {}

public void beforeBegin()
{
    ... perform work ...
}
public void afterCompletion()
{
    ... perform work ...
}
}
```

JDBC の制限事項

Bean で JDBC コールを使用してデータベースを更新し、アクティブなトランザクション・コンテキストがある場合は、トランザクション・サービスを実行するときに、JDBC 接続のメソッドをコールする方法を使用しないでください。JDBC のトランザクション管理メソッドをコーディングしないでください。たとえば、次のようになります。

```
Connection conn = ...
...
conn.commit(); // DO NOT DO THIS!!
```

このようにすると、`SQLException` が発生します。このような操作のかわりに、グローバル・トランザクションを処理するために取得した `UserTransaction` オブジェクトを使用してコミットする必要があります。JDBC 接続を使用してコミットする場合は、グローバル・トランザクションではなくローカル・トランザクションに対してコミットを指示することになります。接続がグローバル・トランザクションに参加しているにもかかわらず、グローバル・トランザクション内でローカル・トランザクションをコミットしようとするエラーが発生します。

JDBC を介して直接 SQL 文によるコミットまたはロールバックを実行することも避ける必要があります。Bean を、`UserTransaction` インタフェースを使用してトランザクションを直接操作するようにコーディングするか、または Bean のコンテナに Bean のトランザクションを管理させるようにします。

XML デプロイメント・ディスクリプタ

Enterprise JavaBean コンポーネントをデータベースに配置するには、適切なデプロイメント・ディスクリプタを提供する必要があります。デプロイメント・ディスクリプタには2種類があり、どちらも XML 表記法を使用して記述できます。

注意： どちらのデプロイメント・ディスクリプタに入力される値にも、すべて大 / 小文字区別があります。これには、ユーザー名とパスワードも含まれます。

次の表は、両方のデプロイメント・ディスクリプタを示しています。

ディプロイメント・ ディスクリプタ	必須/ オプション	説明
Enterprise JavaBean のディプロイメン ト・ディスクリプタ	必須	このディプロイメント・ディスクリプタは、Sun Microsystems の Enterprise JavaBeans 1.1 仕様で定義されています。Bean の名前および機能情報が含まれています。
Oracle 固有のディプ ロイメント・ディス クリプタ	オプション	<p>Oracle 固有のディプロイメント・ディスクリプタには、Oracle8i データベースに配置される Bean に固有の情報が含まれています。次のいずれかに該当する場合にのみ必須です。</p> <ul style="list-style-type: none"> ■ XML ディプロイメント・ディスクリプタで指定された名前が、実際の名前ではなく論理名でもかまわない場合。 ■ <run-as> 要素を指定する必要がある場合。これは、Oracle 固有のディプロイメント・ディスクリプタ内でサポートされている EJB 1.0 の機能です。 ■ Bean にコンテナ管理の永続性を使用する場合は、永続性マネージャとコンテナ管理のフィールドを Oracle 固有のディプロイメント・ディスクリプタで定義します。

この付録では、両方のディプロイメント・ディスクリプタの各セクションの概要を説明します。Sun Microsystems の XML ディプロイメント・ディスクリプタの詳細は、<http://www.javasoft.com> にある Enterprise JavaBeans 1.1 仕様書の「Deployment Descriptor」の章を参照してください。Oracle 固有のディプロイメント・ディスクリプタの詳細は、A-32 ページの「Oracle 固有のディプロイメント・ディスクリプタの DTD」を参照してください。

Enterprise JavaBean のディプロイメント・ディスクリプタ

これは、Bean 識別、セキュリティ・ロール、トランザクション境界（デマーケーション）およびオプションの環境定義など、Bean に関するほとんどの情報を含む主要なディプロイメント・ディスクリプタです。

- [ヘッダー](#)
- [JAR ファイル](#)
- [Enterprise JavaBeans の記述子](#)
- [アプリケーション・アセンブラ・セクション](#)
- [EJB クライアントの JAR セクション](#)

ヘッダー

次の 2 つのヘッダーは、すべての Oracle8i EJB ディプロイメント・ディスクリプタに必須です。XML のバージョンと、ディプロイメント・ディスクリプタに必要な構文の詳細を示す XML の DTD ファイルが記述されます。

XML バージョン・ナンバー

```
<?xml version="1.0"?>
```

DTD ファイル名

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems Inc.//DTD Enterprise JavaBeans 1.1//EN" "ejb-jar.dtd">
```

JAR ファイル

最初に宣言される要素は、<ejb-jar> 要素です。この要素内で、<enterprise-beans> および <assembly-descriptor> という 2 つのセクションを定義します。

配置セクション	説明
<enterprise-beans>	このセクションでは、各 Bean とその環境を定義します。
<assembly-descriptor>	このセクションでは、Bean のセキュリティ・ロールとトランザクション属性を定義します。デフォルトを使用する場合、このセクションはオプションです。

EJB ディプロイメント・ディスクリプタの構造全体に、次の構文が適用されます。

```
<ejb-jar>                                //Start of JAR file descriptor
<description> </description>            //Description of JAR file
<enterprise-beans>                       //EJB Descriptor section
. . .
</enterprise-beans>
<assembly-descriptor>                   //Application Descriptor section
. . .
</assembly-descriptor>
<ejb-client-jar>                         //specify output JAR file for
...                                       //client stubs
</ejb-client-jar>
</ejb-jar>
```

Enterprise JavaBeans の記述子

Bean は、<enterprise-beans> セクションで記述されます。このセクションには、Bean のタイプ（エンティティまたはセッション）、ホーム・インタフェース名、リモート・インタフェース名、Bean クラス名および永続性のタイプ（コンテナ管理または Bean 管理）などの情報が含まれます。次の例は、<enterprise-beans> セクションに含まれる要素を示しています。

```
<enterprise-beans>                       //beginning of the EJB descriptor
<entity> or <session>                   //define EJB type: entity or session
<description> </description>           //text display description
<ejb-name> </ejb-name>                 //logical name for the bean
<home> </home>                         //home interface name
<remote> </remote>                     //remote interface name
<ejb-class> </ejb-class>               //bean class name
<persistence-type> </persistence-type> //For entity beans: container or
                                         //bean-managed?

<prim-key-class> </prim-key-class>     //For entity beans: primary key class
<primkey-field> </prim-key-field>     //For entity beans: primary key field
<reentrant> </reentrant>               //Reentrant boolean: True or False
<cmp-field> </cmp-field>               //Container-managed
                                         //fields for entity beans

<transaction-type> </transaction-type> //transaction information for bean
<env-entry> </env-entry>               //bean environment definition
<ejb-ref> </ejb-ref>                   //EJB environment definition
<resource-ref> </resource-ref>         //database resource environment
<security-role-ref> </security-role-ref> //security role for bean
</session> or </entity>
</enterprise-beans>
```

Bean のタイプ

最初に、Bean がセッション Bean であるかエンティティ Bean であるかを定義する必要があります。そのためには、<entity> または <session> 要素を使用します。

Bean 名

記述子内で Bean を定義するには、次の 4 つの名前が必要です。

- ホーム・インタフェースは、<home> 要素内で定義します。
- リモート・インタフェースは、<remote> 要素内で定義します。
- Bean クラスは、<ejb-class> 要素内で定義します。
- JAR ファイル内の論理名は、<ejb-name> 要素内で定義します。この要素では、2 つの操作のどちらかを実行できます。つまり、この要素内で実際の JNDI 名を宣言するか、ディプロイメント・ディスクリプタ内で Bean を識別する論理名を定義できます。最終的には、<ejb-name> を Bean の JNDI バインド名に解決する必要があります。したがって、論理名を使用する場合は、その名前を Oracle 固有のディプロイメント・ディスクリプタ内で JNDI 名にマップする必要があります。

例 A-1 発注 Bean の記述子

この例では、次のコンポーネントを使用して、発注 Bean をエンティティ Bean として定義しています。

- ホーム・インタフェース — purchase.PurchaseOrderHome
- リモート・インタフェース — purchase.PurchaseOrder
- Bean 実装 — purchaseServer.PurchaseOrderBean
- 記述子内の Bean の論理名 — /test/purchase

注意： この例では、<ejb-name> 要素で JNDI 名「/test/purchase」を使用しています。PurchaseOrder などの論理名を使用した場合は、PurchaseOrder を Oracle 固有のディプロイメント・ディスクリプタの <mappings> 要素内で「/test/purchase」にマップする必要があります。詳細は、A-23 ページの「[マッピングの定義](#)」を参照してください。

```
<enterprise-beans>
  <entity>
    <description>no description</description>
    <ejb-name>test/purchase</ejb-name>
    <home>purchase.PurchaseOrderHome</home>
    <remote>purchase.PurchaseOrder</remote>
    <ejb-class>purchaseServer.PurchaseOrderBean</ejb-class>
```

```
<persistence-type>Container</persistence-type>
<prim-key-class>java.lang.String</prim-key-class>
<reentrant>False</reentrant>
</entity>
</enterprise-beans>
```

エンティティ Bean の要素

エンティティ Bean にのみ関連する項目は、次のように特定の要素で定義します。

- <persistence-type> 要素によって、Bean がコンテナ管理か Bean 管理かを定義します。値「Container」または「Bean」を使用できます。
- <cmp-field> 要素では、コンテナ管理の永続（CMP）エンティティ Bean 用のデータ・フィールドを定義します。各データ・フィールドは、独自の <cmp-field> <field-name> セクションにリストします。
- エンティティ Bean の主キーを定義します。この操作が必須なのは CMP Bean の場合のみであることに注意してください。

主キーは、java.lang.String のように SQL の型と整合性のある Java の型の単一フィールドとして宣言するか、複数のコンテナ管理フィールドの組合せとして宣言できます。ただし、主キーには、データベース内の Long Raw 列にマップするバイト配列としては定義できないという制限事項があります。

主キーの宣言	必要な方法論
SQL の型と整合性のある Java の型	<div>1. 主キーのデータ型を <prim-key-class> 要素内で宣言します。</div> <div>2. 主キーとなるフィールドを <cmp-field> 要素内で CMP として定義します。</div> <div>3. CMP フィールドを <primkey-field> 要素内で定義して指定します。</div>
主キー用のコンテナ管理の永続フィールドの組合せ	<div>1. 主キーのすべてのフィールドを、<cmp-field> 要素内で CMP として定義します。</div> <div>2. 主キーにある <cmp-field> 要素の名前を含むシリアライズ可能なクラスを、<bean_name>PK という名前で定義します。</div> <div>3. <prim-key-class> 要素内で主キーのクラスをリストします。</div>

例 A-2 CMP エンティティ Bean

この例では、顧客 Bean は CMP エンティティ Bean であり、主キーとしての顧客番号と、顧客名および住所という 2 つの永続フィールドを持っています。

- この Bean は、<persistence-type> 要素内でコンテナ管理の永続 Bean として定義されています。
- 主キーは顧客 ID である custid で、java.lang.String として宣言されています。
 1. custid は、<cmp-field> 内で定義されています。
 2. custid が <primkey-field> 内で主キーにマップされます。
 3. custid のデータ型が <prim-key-class> 要素内で宣言されます。
- もう 1 つのコンテナ管理の永続フィールドは、<cmp-field> 要素内で name および addr として定義されています。

注意： コンテナ管理フィールドを実際の永続記憶域に変換するメソッドは、選択する永続性マネージャに応じて異なります。詳細は、A-27 ページの「[コンテナ管理の永続性の定義](#)」を参照してください。

```
<enterprise-beans>
  <entity>
    <description>customer bean</description>
    <ejb-name>/test/customer</ejb-name>
    <home>customer.CustomerHome</home>
    <remote>customer.Customer</remote>
    <ejb-class>customerServer.CustomerBean</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.String</prim-key-class>
    <reentrant>False</reentrant>
    <cmp-field><field-name>custid</field-name></cmp-field>
    <cmp-field><field-name>name</field-name></cmp-field>
    <cmp-field><field-name>addr</field-name></cmp-field>
    <primkey-field>custid</primkey-field>
  </entity>
</enterprise-beans>
```

例 A-3 複合主キーを持つ CMP エンティティ Bean

この例では、顧客 Bean は CMP エンティティ Bean であり、その複合主キーが独自のシリアル化可能クラスで定義されています。また、この CMP Bean では、顧客名および住所という 2 つの永続フィールドが宣言されています。

- この Bean は、<persistence-type> 要素内でコンテナ管理の永続 Bean として定義されています。

- 主キーは、customer.CustomerPK クラス内で定義されています。
 1. 主キーの要素は、顧客の姓 (custname) と生年月日 (dobirth) です。どちらのフィールドも、<cmp-field> 内で定義されています。
 2. custid のデータ型が <prim-key-class> 要素内で宣言されます。
- もう1つのコンテナ管理の永続フィールドは、<cmp-field> 要素内で name および addr として定義されています。

注意： コンテナ管理フィールドを実際の永続記憶域に変換するメソッドは、選択する永続性マネージャに応じて異なります。詳細は、A-27 ページの「[コンテナ管理の永続性の定義](#)」を参照してください。

```
<enterprise-beans>
  <entity>
    <description>customer bean</description>
    <ejb-name>/test/customer</ejb-name>
    <home>customer.CustomerHome</home>
    <remote>customer.Customer</remote>
    <ejb-class>customerServer.CustomerBean</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>customer.CustomerPK</prim-key-class>
    <reentrant>False</reentrant>
    <cmp-field><field-name>custname</field-name></cmp-field>
    <cmp-field><field-name>dobirth</field-name></cmp-field>
    <cmp-field><field-name>name</field-name></cmp-field>
    <cmp-field><field-name>addr</field-name></cmp-field>
  </entity>
</enterprise-beans>
```

環境要素

実行時に Bean でアクセスできるように、環境変数、EJB 参照およびリソース・マネージャ (JDBC の DataSource) という3種類の環境要素を作成できます。これらの環境要素は静的であり、Bean では変更できません。

通常、ISV (独立系ソフトウェア・ベンダー) は、EJB コンテナに依存しない EJB を開発します。Bean 実装をコンテナ固有のものから分離するために、定義済みの変数、エンティティ Bean またはリソース・マネージャのうち、いずれか1つにマップする環境要素を作成できます。この間接性により、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、ディプロイメント・ディスクリプタ内で定義され、Oracle 固有のディプロイメント・ディスクリプタ内の実際の名前にリンクされます。

環境変数 Bean で InitialContext の検索を介してアクセス可能な環境変数を作成できます。これらの変数は <env-entry> 要素内で定義でき、型として String、Integer、Boolean、Double、Byte、Short、Long および Float のいずれかを指定できます。環境変数名は <env-entry-name> 内、その型は <env-entry-type> 内、初期化済みの値は <env-entry-value> 内で定義します。<env-entry-name> は、「java:comp/env」コンテキストと関連しています。

たとえば、次の 2 つの環境変数は、java:comp/env/minBalance および java:comp/env/maxCreditBalance の XML ディプロイメント・ディスクリプタ内で宣言されます。

```
<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>
```

この Bean のコード内で、これらの環境変数には次のように InitialContext を介してアクセスします。

```
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、各環境要素に接頭辞「java:comp/env/」を使用する必要があります。これは、コンテナにより環境変数が格納された位置を示します。

他の Enterprise JavaBeans の環境参照 ディプロイメント・ディスクリプタ内で、EJB の参照を定義できます。Bean で他の Bean をコール・アウトすることがわかっている場合は、この参照をディプロイメント・ディスクリプタ内で定義できます。deployejb ツールでは、EJB 参照が Bean のホーム・インタフェースにバインドされ、開始側 Bean ではターゲットの正確な JNDI 名を知らなくても、ターゲット Bean を検索できます。いずれにせよ、ターゲット Bean のホーム・インタフェースを取得するには、JNDI を使用します。取得後は、戻された EJB 参照で参照される Bean が、同じセッションでインスタンス化されます。

ターゲット Bean を環境の EJB 参照として宣言すると一定レベルの間接性が得られるため、開始側 Bean ではターゲット Bean を論理名で参照できます。これは、同じネームスペースにあるターゲット Bean の JNDI 名が、オペレーティング環境に応じて異なる可能性がある場合に役立ちます。

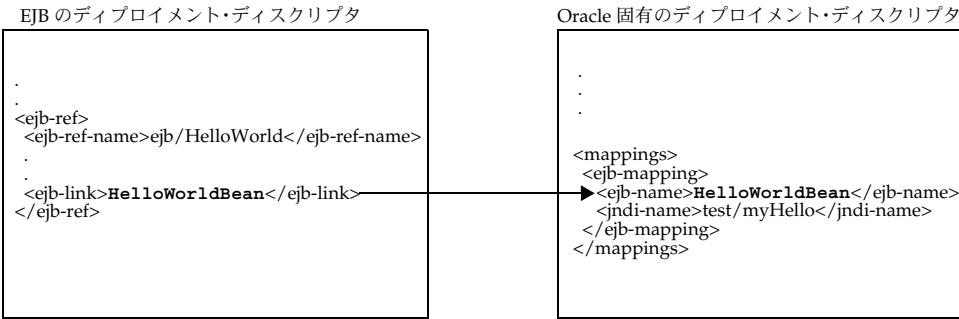
環境内で EJB を定義するには、次の値を指定します。

- 1. `ejb-ref-name` — ターゲット Bean の論理名を指定します。この名前は、Bean がターゲット Bean にアクセスするために JNDI の URL に使用されます。この名前は「`ejb/myEmployee`」のように「`ejb/`」で始める必要があり、「`java:comp/env/ejb`」コンテキスト内で使用可能になります。
- 2. `ejb-ref-type` — セッション Bean であるかエンティティ Bean であるかを定義します。値「`Session`」または「`Entity`」を指定する必要があります。
- 3. `home` — ホーム・インタフェースの完全修飾名を指定します。
- 4. `remote` — リモート・インタフェースの完全修飾名を指定します。
- 5. `ejb-link` — この EJB 参照を実際の JNDI URL でリンクする名前を指定します。

図 A-1 のように、Bean の論理名は、EJB ディプロイメント・ディスクリプタの `<ejb-link>` 要素と Oracle 固有のディプロイメント・ディスクリプタの `<ejb-mapping>` 要素内の `<ejb-name>` に、同じリンク名「`HelloWorldBean`」を指定することで、JNDI 名にマップされます。

注意： `<ejb-link>` フィールドを使用して、Oracle 固有のディプロイメント・ディスクリプタ内で定義されている JNDI URL に論理名をマップする場合、実装は Enterprise JavaBeans 1.1 仕様書に規定されているものとは異なるものになります。

図 A-1 EJB 参照のマッピング



例 A-4 環境内での EJB 参照の定義

この例では、Hello Bean の参照を次のように定義しています。

- 1. 開始側 Bean 内でターゲット Bean に使用される論理名は、「`java:comp/env/ejb/HelloWorld`」です。

2. ターゲット Bean はセッション Bean です。
3. そのホーム・インタフェースは `hello.HelloHome`、リモート・インタフェースは `hello.Hello` です。
4. この Bean の JNDI URL へのリンクは、Oracle 固有のディプロイメント・ディスクリプタでは「`HelloWorldBean`」という名前で定義されています。

EJB
ディプロイメント・
ディスクリプタ

```
<ejb-ref>
  <description>Hello World Bean</description>
  <ejb-ref-name>ejb/HelloWorld</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>hello.HelloHome</home>
  <remote>hello.Hello</remote>
  <ejb-link>HelloWorldBean</ejb-link>
</ejb-ref>
```

図 A-1 のように、両方の要素に同じ論理名を指定することで、`<ejb-link>` は Oracle 固有のディプロイメント・ディスクリプタの `<ejb-mapping>` 要素内の `<ejb-name>` にマップされています。この Oracle 固有のディプロイメント・ディスクリプタでは、論理 Bean 名「`java:comp/env/ejb/HelloWorld`」を JNDI URL「`/test/myHello`」にマップするために、次のように定義されています。

Oracle 固有の
ディプロイメント・
ディスクリプタ

```
<mappings>
  <ejb-mapping>
    <ejb-name>HelloWorldBean</ejb-name>
    <jndi-name>/test/myHello</jndi-name>
  </ejb-mapping>
</mappings>
.
.
</ejb-ref>
```

注意： Oracle 固有のディプロイメント・ディスクリプタの詳細は、A-22 ページの「[Oracle 固有のディプロイメント・ディスクリプタ](#)」を参照してください。

この Bean を実装内から起動するには、ディプロイメント・ディスクリプタで定義されている `<ejb-ref-name>` を使用します。この名前には、接頭辞「`java:comp/env/ejb/`」を付けます。この接頭辞は、ディプロイメント・ディスクリプタに定義されている EJB 参照がコンテナにより配置される位置を示します。

```
InitialContext ic = new InitialContext();
HelloHome hh = (HelloHome)ic.lookup("java:comp/env/ejb/HelloWorld");
```

リソース・マネージャの接続ファクトリ参照への環境参照 リソース・マネージャの接続ファクトリ参照には、JMS、メール、XML および JDBC 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できます。このリリースの Oracle8i でサポートされるリソース・マネージャ接続ファクトリは、JDBC DataSource のみです。

EJB 参照と同様に、JDBC を介してデータベースにアクセスするには、従来のメソッドを使用する方法と、JDBC の DataSource 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、次の作業が必要です。

- 1. JDBC の DataSource を JNDI ネームスペース内でバインドします。
- 2. EJB ディプロイメント・ディスクリプタ内で論理名を作成します。この名前は、常に「java:comp/env/jdbc」で始める必要があります。
- 3. EJB ディプロイメント・ディスクリプタ内の論理名を、ステップ 1 で Oracle 固有のディプロイメント・ディスクリプタ内に作成した JNDI 名にマップします。

図 A-2 のように、JDBC の DataSource は JNDI 名「test/OrderDataSource」にバインドされています。Bean でこのリソースの認識に使用される論理名は、「jdbc/OrderDB」です。これらの名前は、Oracle 固有のディプロイメント・ディスクリプタ内でまとめてマップされます。したがって、Bean の実装内では、Bean は「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 A-2 JDBC のリソース・マネージャのマッピング

EJB のディプロイメント・ディスクリプタ

```
<enterprise-beans>
.
<resource-ref>
  <res-ref-name>jdbc/OrderDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Application</res-auth>
</resource-ref>
</enterprise-beans>
```

Oracle 固有のディプロイメント・ディスクリプタ

```
<mappings>
  <resource-ref-mapping>
    <res-ref-name>jdbc/OrderDB</res-ref-name>
    <jndi-name>test/OrderDataSource</jndi-name>
  </resource-ref-mapping>
</mappings>
```

JDBC の DataSource 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>	開始側 Bean 内で使用される JDBC の DataSource の論理名。この名前には、接頭辞「jdbc/」を付ける必要があります。この例では、注文管理データベースの論理名は「jdbc/OrderDB」です。
<res-type>	リソースの Java の型。JDBC の場合、これは <code>javax.sql.DataSource</code> です。

要素	説明
<res-auth>	データベースへのサイン・オンの担当者を定義します。この時点でサポートされている値は、「Application」のみです。アプリケーションは DataSource 内でユーザー名とパスワードを提供し、データベースの認証情報を提供します。

例 A-5 JDBC 接続用の環境要素の定義

環境要素を EJB ディプロイメント・ディスクリプタ内で定義するには、論理名「jdbc/OrderDB」、その型である javax.sql.DataSource および認証者である「Application」を指定します。

EJB
ディプロイメント・
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>
```

環境要素「jdbc/OrderDB」は、Oracle 固有のディプロイメント・ディスクリプタ内で接続の JNDI バインド名「test/OrderDataSource」にマップされます。

Oracle 固有の
ディプロイメント・
ディスクリプタ

```
<mappings>
  <resource-ref-mapping>
    <res-ref-name>jdbc/OrderDB</res-ref-name>
    <jndi-name>test/OrderDataSource</jndi-name>
  </resource-ref-mapping>
</mappings>
```

配置後は、アプリケーションで JDBC の DataSource を次のように取得できます。

```
javax.sql.DataSource db;
java.sql.Connection conn;
.
.
.
db = (javax.sql.DataSource) initCtx.lookup("java:comp/env/jdbc/OrderDB");
conn = db.getConnection();
```

Bean のサービス

Bean のトランザクション、セキュリティおよび再入可能性は、次の要素で定義されます。

- `<reentrant>` 要素によって、Bean の再入可能性を定義します。値「True」または「False」を指定する必要があります。
- `<transaction-type>` 要素によって、セッション Bean のトランザクション境界（デマーケーション）のタイプを定義します。値は、セッション Bean でトランザクション制御に Bean 境界（デマーケーション）を使用する場合は「Bean」、コンテナ境界（デマーケーション）を使用する場合は「Container」にする必要があります。

セッション Bean の場合もエンティティ Bean の場合も、トランザクション情報の詳細は `<assembly-descriptor>` セクション内で定義することに注意してください。詳細は、A-18 ページの「[トランザクションの定義](#)」を参照してください。

- `<security-role-ref>` 要素で、Bean のセキュリティ・ロールの論理名を定義します。これは、Bean 内で使用されるセキュリティ・ロール名を参照する論理名で、`<assembly-descriptor>` セクションで定義されている実際のセキュリティ・ロールにマップする必要があります。詳細は、A-15 ページの「[セキュリティの定義](#)」を参照してください。

例 A-6 再入可能性、トランザクション境界（デマーケーション）およびセキュリティ・ロール

次の例では、リエントラントでなく、Bean 境界（デマーケーション）を持つトランザクションを使用し、そのセキュリティ・ロールの参照時に Bean 実装内の論理名「CustRole」を使用する、顧客 Bean を定義しています。このロール名は SCOTT にマップされています。

```
<enterprise-beans>
  <session>
    <ejb-name>CustomerBean</ejb-name>
    <home>customer.CustomerHome</home>
    <remote>customer.Customer</remote>
    <ejb-class>customerServer.CustomerBean</ejb-class>
    <reentrant>False</reentrant>
    <transaction-type>Bean</transaction-type>
    <security-role-ref>
      <role-name>CustRole</role-name>
      <role-link>SCOTT</role-link>
    </security-role-ref>
  </session>
</enterprise-beans>
```

アプリケーション・アセンブラ・セクション

アプリケーション・アセンブラにより、この記述子に含まれるすべての Bean の一般情報が <assembly-descriptor> セクションに追加されます。このセクションの構造は、次のとおりです。

```
<assembly-descriptor>
  <security-role> </security-role>
  <method-permission> </method-permission>
  <container-transaction> </container-transaction>
</assembly-descriptor>
```

各セクションでは、セキュリティ属性とトランザクション属性が記述されています。

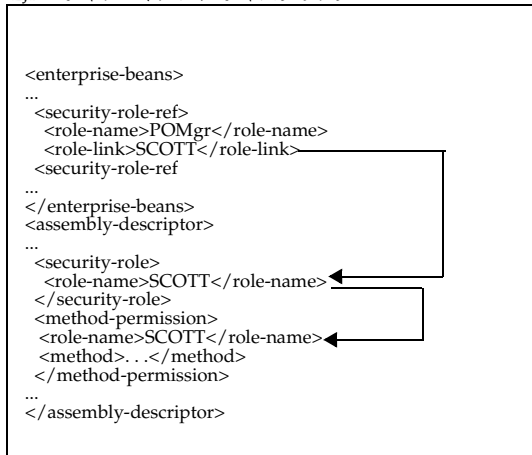
セキュリティの定義

ディプロイメント・ディスクリプタ内で、ユーザーとメソッドのパーミッションに関して、セキュリティの一部を管理できます。メソッドのパーミッションを指定しなければ、デフォルトではだれもアクセスを許可されません。

また、[図 A-3](#) のように、Bean 実装内でロールの論理名を使用し、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プできます。論理名からデータベース・ロールへのマッピングは、Oracle 固有のディプロイメント・ディスクリプタ内で指定します。詳細は、A-25 ページの「[セキュリティ・ロール](#)」を参照してください。

図 A-3 セキュリティのマッピング

EJB のディプロイメント・ディスクリプタ



isCallerInRole などのメソッドの Bean 実装内でデータベース・ロールに論理名を使用する場合は、次の手順で論理名を実際のデータベース・ロールにマップできます。

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、発注の例で使用しているロールを定義するには、Bean の実装内でコール側が発注を承認するための認可を持っているかどうかをチェックできます。したがって、コール側は正しいロールで承認される必要があります。注文を承認できるのは発注管理者のみのため、Bean でデータベース・ロールを認識する必要がないようにするために、POMgr などの論理名に関して isCallerInRole をチェックできます。そのため、次のように、<enterprise-beans> セクションの <security-role-ref><role-name> 要素内で、論理セキュリティ・ロール POMgr を定義します。

```
<enterprise-beans>
...
  <security-role-ref>
    <role-name>POMgr</role-name>
    <role-link>SCOTT</role-link>
  </security-role-ref>
</enterprise-beans>
```

<security-role-ref> 要素内の <role-link> 要素には、実際のデータベース・ロールを指定できます。このロールを <assembly-descriptor> セクション内でさらに定義します。また、他の論理名を指定することもできます。この場合も <assembly-descriptor> セクションで詳細に定義すると、Oracle 固有のディプロイメント・ディスクリプタ内で実際のデータベース・ロールにマップされます。

2. ロールと、そのロールに適用するメソッドを定義します。発注の例では、PurchaseOrder Bean 内で実行されるすべてのメソッドは、それ自体を SCOTT で許可する必要があります。PurchaseOrder が、<entity | session><ejb-name> 要素で宣言されている名前であることに注意してください。

したがって、次の例では、ロールとして SCOTT、EJB として PurchaseOrder、および * 記号ですべてのメソッドが定義されています。

注意： SCOTT は、<enterprise-beans> セクションの <role-link> 要素と同じです。これにより、論理名 POMgr が SCOTT の定義に連結されます。

```
<assembly-descriptor>
  <security-role>
    <description>Role needed purchase order authorization</description>
    <role-name>SCOTT</role-name>
  </security-role>
  <method-permission>
    <role-name>SCOTT</role-name>
```

```

<method>
  <ejb-name>PurchaseOrder</ejb-name>
  <method-name>*</method-name>
</method>
</method-permission>
...
</assembly-descriptor>

```

両方のステップを実行すると、Bean の実装内で POMgr を参照でき、POMgr はコンテナにより SCOTT に変換されます。

<method> 要素は、インタフェースまたは実装内で 1 つ以上のメソッドを定義するために使用されます。EJB 仕様によれば、この定義には次の 3 つの書式のいずれかを使用できます。

1. 次のように、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。

```

<method>
  <ejb-name>EJBNAME</ejb-name>
  <method-name>*</method-name>
</method>

```

2. Bean 内で一意に識別される特定のメソッドを定義します。次のように、適切なインタフェース名とメソッド名を使用します。

```

<method>
  <ejb-name>myBean</ejb-name>
  <method-name>myMethodInMyBean</method-name>
</method>

```

注意： 同じオーバーロード名を持つ複数のメソッドが存在する場合、このスタイルの要素では、その名前を持つすべてのメソッドが参照されます。

3. 次のように、多数のオーバーロード・バージョン間で特定のシグネチャを持つメソッドを定義します。

```

<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>

```

パラメータは、メソッドの入力パラメータの完全修飾による Java の型です。メソッドが入力引数を取らない場合、`<method-params>` 要素には要素は含まれません。配列は、`int[][]` のように、配列要素の型と後続の 1 組以上の大カッコで指定します。

4. 複数のインタフェース内で定義されている特定のメソッド名を定義します。Bean 実装とリモート・インタフェースまたはホーム・インタフェースの両方で、同じメソッド名が定義されている場合があります。この場合は、次のように `<method-intf>` 要素を使用して、どちらのメソッドを定義するのかを指定する必要があります。

```
<method>
  <ejb-name>EmployeeService</ejb-name>
  <method-intf>Remote</method-intf>
  <method-name>create</method-name>
  <method-params>
    <method-param>java.lang.String</method-param>
    <method-param>java.lang.String</method-param>
  </method-params>
</method>
```

`<method-intf>` 要素でリモート・インタフェースを定義すると、リモート・インタフェース内で定義されている `create(String, String)` メソッドは、ホーム・インタフェース内で定義されている `create(String, String)` メソッドと区別されます。定義は次のようになります。

```
<method>
  <ejb-name>EmployeeService</ejb-name>
  <method-intf>Home</method-intf>
  <method-name>create</method-name>
  <method-params>
    <method-param>java.lang.String</method-param>
    <method-param>java.lang.String</method-param>
  </method-params>
</method>
```

注意： `<method-intf>` 要素は、`<method-name>` 要素の前に定義する必要があります。

トランザクションの定義

エンティティ Bean で使用できるのはコンテナ管理のトランザクションのみですが、セッション Bean では、Bean 管理またはコンテナ管理のトランザクションを使用できます。セッション Bean の場合は、`<enterprise-beans>` セクションの `<transaction-type>` 要素内で、Bean 管理とコンテナ管理のうち、どちらのトランザクションを使用するかを定義します。値「Bean」または「Container」を指定する必要があります。たとえば、次の例では、セッション Bean を Bean 管理のトランザクションを行う Bean として定義しています。

```
<enterprise-beans>
  <session>
    . . .
    <transaction-type>Bean</transaction-type>
  </session>
</enterprise-beans>
```

コンテナ管理の Bean の場合は、トランザクション属性によって、コンテナで Bean のトランザクションをメンテナンスする方法を定義します。トランザクション属性では、新規トランザクションを開始する状況、既存のトランザクションを続行する状況などを定義します。Bean 管理の Bean については、これらの属性を定義しません。ただし、Bean 管理の Bean で別の Bean を起動する場合、ターゲット Bean は Bean 管理でもコンテナ管理でもかまいません。

トランザクション属性は <trans-attribute> 要素で指定します。表 A-1 は、その詳細を示しています。

表 A-1 トランザクション属性

トランザクション属性	説明
NotSupported	Bean はトランザクションに関与しません。クライアントがトランザクションに関与する間に Bean をコールすると、クライアントのトランザクションが一次停止されて Bean が実行され、Bean がクライアントに制御を戻すと、クライアントのトランザクションが再開されます。
Required	Bean はトランザクションに関与する必要があります。クライアントがトランザクションに関与している場合、Bean ではクライアントのトランザクションが使用されます。クライアントがトランザクションに関与していなければ、コンテナにより Bean の新規トランザクションが開始されます。
Supports	クライアントが関与するトランザクションの状態が Bean に使用されます。クライアントがトランザクションを開始すると、クライアントのトランザクション・コンテキストが Bean で使用されます。クライアントがトランザクションに関与しなければ、どちらも Bean で使用されません。
RequiresNew	クライアントがトランザクションに関与するかどうかに関係なく、この Bean には専用存在する新規トランザクションが必要です。クライアントがトランザクションに関与する間に Bean をコールすると、クライアントのトランザクションは Bean が完了するまで一時停止されます。

表 A-1 トランザクション属性 (続き)

トランザクション属性	説明
Mandatory	クライアントは、この Bean を起動する前にトランザクションに関与する必要があります。Bean は、クライアントのトランザクション・コンテキストを使用します。
Never	Bean はトランザクションに関与しません。また、クライアントは、Bean をコールしている間はトランザクションに関与できません。クライアントがトランザクションに関与すると、RemoteException が発生します。

トランザクション属性は、Bean 全体または Bean 内の個々のメソッドについて定義できます。それぞれの定義は、<container-transaction> 要素に含まれます。Bean 全体の属性を定義する場合は、<container-transaction> <method> <ejb-name> 要素内で Bean 名 (<session> または <entity> 要素内で定義した <ejb-name>) を指定します。

次の例では、次の要件が定義されています。

- PurchaseOrder Bean 内のすべてのメソッド ('*' 文字で指定) には、トランザクションが必要です (Required)。

注意： 各メソッドは、<method-permission> 要素内で定義されているとおり正確に指定できます。詳細は、A-15 ページの「[セキュリティの定義](#)」を参照してください。

- PurchaseOrder Bean 内の price メソッドには、常に新規トランザクションが必要です (RequiresNew)。

```
<assembly-descriptor>
...
  <container-transaction>
    <method>
      <ejb-name>PurchaseOrder</ejb-name>
      <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
  <container-transaction>
    <method>
      <ejb-name>PurchaseOrder</ejb-name>
      <method-name>price</method-name>
    </method>
```

```
    <trans-attribute>RequiresNew</trans-attribute>
  </container-transaction>
  . . .
</assembly-descriptor>
```

最後に、グローバル・トランザクションに 2 フェーズ・コミットを使用する場合は、Oracle 固有のディプロイメント・ディスクリプタ内の <transaction-manager> 要素に、UserTransaction オブジェクトの JNDI 名を指定する必要があります。詳細は、A-26 ページの「[トランザクションの 2 フェーズ・コミット・エンジンの定義](#)」を参照してください。

Oracle 固有のディプロイメント・ディスクリプタ

Oracle 固有のディプロイメント・ディスクリプタの用途は、次のとおりです。

- マッピングの定義
- トランザクションの 2 フェーズ・コミット・エンジンの定義
- run-as アイデンティティーの定義
- コンテナ管理の永続性の定義

このディプロイメント・ディスクリプタの構造は、次のとおりです。

```
<?xml version="1.0"?>
<!DOCTYPE oracle-descriptor PUBLIC "-//Oracle Corporation//DTD Enterprise JavaBeans
1.1//EN" "oracle-ejb-jar.dtd">

<oracle-descriptor>
  <mappings>
    <ejb-mapping> </ejb-mapping>
    <resource-ref-mapping> </resource-ref-mapping>
  </mappings>
  <run-as> </run-as>
  <persistence-provider> </persistence-provider>
  <persistence-descriptor> </persistence-descriptor>
</oracle-descriptor>
```

注意： DTD 全体については、A-32 ページの「[Oracle 固有のディプロイメント・ディスクリプタの DTD](#)」を参照してください。

ヘッダー

次の 2 つのヘッダーは、すべての Oracle8i EJB ディプロイメント・ディスクリプタに必須です。XML のバージョンと XML の DTD ファイルの詳細が記述されます。

XML バージョン・ナンバー

```
<?xml version="1.0"?>
```

DTD ファイル名

```
<!DOCTYPE oracle-descriptor PUBLIC "-//Oracle Corporation//DTD Oracle 1.1//EN"
"oracle-ejb-jar.dtd">
```

マッピングの定義

Oracle 固有のディプロイメント・ディスクリプタの主な用途は、EJB ディプロイメント・ディスクリプタ内で使用される論理名を、Oracle8i データベース内で使用される実際の名前にマップすることです。次のように、様々な名前をマップできます。

EJB の論理名	説明
<session entity> <ejb-name> 内で定義されている Bean 名	<ejb-name> 要素内で JNDI 名のかわりに論理名を使用している場合は、この名前を実際の JNDI 名にマップする必要があります。
EJB 参照の環境要素	<ejb-ref> 内でターゲット Bean の論理名を指定している場合は、この論理名を EJB の実際の JNDI 名にマップする必要があります。
JDBC DataSource の環境要素	環境要素として JDBC DataSource を定義している場合は、<res-ref-name> を実際の JNDI 名にマップする必要があります。
セキュリティ・ロール	<assembly-descriptor> 内の <security-role><role-name> 要素で論理名を使用している場合は、この論理名をデータベース内で使用する実際のユーザー名にマップする必要があります。
トランザクション・マネージャ	グローバル・トランザクションに 2 フェーズ・コミット・エンジンを使用する場合は、このデータベースを表す OracleJTADDataSource オブジェクトの JNDI 名を指定します。

Bean 名

A-5 ページの「[Bean 名](#)」で説明したように、Bean 名には論理名または JNDI 名を使用できます。次の例のように、論理名を定義した場合は、この名前を <mappings> セクション内で実際の JNDI 名にマップする必要があります。

例 A-7 EJB 名

次の例では、EJB ディプロイメント・ディスクリプタ内で、Bean の論理名 PurchaseOrder を定義しています。

```
<enterprise-beans>
  <entity>
    <description>no description</description>
    <ejb-name>PurchaseOrder</ejb-name>
    <home>purchase.PurchaseOrderHome</home>
    <remote>purchase.PurchaseOrder</remote>
    <ejb-class>purchaseServer.PurchaseOrderBean</ejb-class>
    <persistence-type>Bean</persistence-type>
```

```
<prim-key-class>java.lang.String</prim-key-class>
<reentrant>False</reentrant>
</entity>
</enterprise-beans>
```

PurchaseOrder 論理名は、Oracle 固有のディプロイメント・ディスクリプタの <mappings> セクション内で、その JNDI 名にマップされます。次の例は、Bean 名を <ejb-name> 要素内で定義し、対応する JNDI 名を <jndi-name> 要素内で定義する方法を示しています。

```
<oracle-descriptor>
  <mappings>
    <ejb-mapping>
      <ejb-name>PurchaseOrder</ejb-name>
      <jndi-name>/test/purchase</jndi-name>
    </ejb-mapping>
  </mappings>
  . . .
</oracle-descriptor>
```

EJB 参照

EJB 参照のマッピングについては、A-9 ページの「[他の Enterprise JavaBeans の環境参照](#)」を参照してください。マッピングの構造は、A-23 ページの「[Bean 名](#)」に示したように、Bean 名と同じです。

JDBC の DataSource

JDBC の DataSource の接続のマッピングについては、A-12 ページの「[リソース・マネージャの接続ファクトリ参照への環境参照](#)」を参照してください。JDBC の DataSource は、次のように <resource-ref-mapping> 要素でバインドされている JNDI 名にマップします。

```
<oracle-descriptor>
  <mappings>
    <resource-ref-mapping>
      <res-ref-name>jdbc/EmployeeAppDB</res-ref-name>
      <jndi-name>jdbc/OracleDataSource</jndi-name>
    </resource-ref-mapping>
  </mappings>
  . . .
</oracle-descriptor>
```

<res-ref-name> 要素には、EJB のディプロイメント・ディスクリプタ内で定義されている JDBC の DataSource が含まれています。<jndi-name> 要素には、ネームスペース内で JDBC の DataSource にバインドされている JNDI 名が含まれています。この例では、「jdbc:comp/env/jdbc/EmployeeAppDB」環境名が JNDI 名「jdbc/OracleDataSource」にマップされています。

セキュリティ・ロール

<assembly-descriptor> 内で <role-name> の論理名を定義した場合は、この論理名を Oracle 固有のディプロイメント・ディスクリプタ内の実際のデータベース・ロールまたはユーザーにマップする必要があります。

たとえば、発注許可に必要なロールの論理ロール POMGR を、<assembly-descriptor> 内で定義します。この認可を行うことが許可される実際のデータベース・ロールは、SCOTT です。次の例は、Oracle 固有のディプロイメント・ディスクリプタ内で必要なマッピングの定義を示しています。

```
<mappings>
  <security-role-mapping>
    <security-role>
      <description>POMGR role mapping</description>
      <role-name>POMGR</role-name>
    </security-role>
    <oracle-role>SCOTT</oracle-role>
  </security-role-mapping>
</mappings>
```

図 A-4 のように、<assembly-descriptor><security-role><role-name> で定義されている <role-name> の POMGR は、Oracle 固有のディプロイメント・ディスクリプタの <security-role-mapping> 要素内で SCOTT にマップされています。

図 A-4 セキュリティのマッピング

EJB のディプロイメント・ディスクリプタ

```
<enterprise-beans>
...
<security-role-ref>
  <role-name>POMGr</role-name>
  <role-link>POMGR</role-link>
</security-role-ref>
...
</enterprise-beans>
<assembly-descriptor>
...
  <security-role>
    <role-name>POMGR</role-name>
  </security-role>
  <method-permission>
    <role-name>POMGR</role-name>
  </method-permission>
...
</assembly-descriptor>
```

Oracle 固有のディプロイメント・ディスクリプタ

```
<mappings>
...
  <security-role-mapping>
    <security-role>
      <description>mapping POMGR</description>
      <role-name>POMGR</role-name>
      <oracle-role>SCOTT</oracle-role>
    </security-role>
  </security-role-mapping>
...
</mappings>
```

トランザクションの 2 フェーズ・コミット・エンジンの定義

グローバル・トランザクションに 2 フェーズ・コミットを使用する場合は、`<transaction-manager>` 要素に、`UserTransaction` オブジェクトの JNDI 名を指定する必要があります。次の例では、`UserTransaction` オブジェクト「`/test/myUTFor2pc`」を指定しています。

```
<mappings>
...
<transaction-manager>
  <jndi-name>/test/myUTFor2pc</jndi-name>
</transaction-manager>
</mappings>
```

run-as アイデンティティの定義

`<run-as>` 要素は、Oracle 固有のディプロイメント・ディスクリプタ内で引き続きサポートされている EJB 1.0 の要素です。`<run-as>` 要素では、特定の EJB を指定のアイデンティティで実行するように定義できます。Bean に指定できるアイデンティティには、次の 3 タイプがあります。

run-as アイデンティティ	説明
CLIENT_IDENTITY	Bean はクライアントのアイデンティティで実行されます。これはデフォルトです。
SYSTEM_IDENTITY	Bean は SYSTEM アイデンティティで実行されます。これは、Bean が実行される環境のタイプに固有です。たとえば、Oracle8i データベース上では、SYSTEM アイデンティティは SYS と等価です。
SPECIFIED_IDENTITY	起動時に Bean の特定のアイデンティティまたはロールを指定できます。このアイデンティティは、 <code><security-role></code> 要素内で指定します。

`<method>` 要素では、`<run-as>` の定義が適用されるメソッドを指定します。その定義は、次のとおりです。

- `<ejb-name>`: XML のディプロイメント・ディスクリプタ内で定義されている Bean の論理名。
- `<method-name>`: このモードが Bean 内で適用されるメソッド名 (* はすべてのメソッドを意味します)。
- `<method-params>`: この名前でオーバーロード・パラメータを持つ複数のメソッドがあり、その 1 つにのみ `<run-as>` を適用する場合は、この要素内でパラメータを指定します。

たとえば、次の例では、PurchaseOrder EJB 内で 2 つの String パラメータが必要な price メソッドを、起動時に常に SCOTT で実行するように定義しています。

```
<run-as>
  <description>no description</description>
  <mode>SPECIFIED_IDENTITY</mode>
  <security-role>
    <role-name>SCOTT</role-name>
  </security-role>
  <method>
    <ejb-name>PurchaseOrder</ejb-name>
    <method-name>price</method-name>
    <methodl-params>
      <method-param>java.lang.String<method-param>
      <method-param>java.lang.String<method-param>
    </method>
  </run-as>
```

注意： <security-role> 要素は、SPECIFIED_IDENTITY 値の場合にのみ指定します。CLIENT_IDENTITY 値と SYSTEM_IDENTITY 値の場合、<security-role> 要素は無視されます。

コンテナ管理の永続性の定義

エンティティ Bean の永続変数をコンテナで管理するように選択した場合は、永続性プロバイダ名前を指定し、各永続性プロバイダをコンテナ管理の永続 Bean にマップする必要があります。このバージョンでは、Oracle Persistence Service Interface Reference Implementation (PSI-RI) 永続性マネージャがサポートされます。このマネージャを使用して、永続フィールドからデータベースへのマッピングを定義する必要があります。

永続性プロバイダ 永続性プロバイダにより、すべてのコンテナ管理フィールドの管理が制御されます。ここでは、Bean 内のフィールドを対応する永続記憶域にマップする方法が定義されます。また、これらのフィールドを Bean から記憶域に保存して更新する方法も定義されます。

Oracle PSI-RI 永続性プロバイダは、<persistence-provider> 要素内で定義します。次の要素で永続性プロバイダを記述します。

永続性プロバイダの要素	説明
<description>	プロバイダの記述を指定します。
<persistence-name>	プロバイダの論理名を定義します。このリリースでサポートされるのは、Oracle PSI-RI のみです。
<persistence-deployer>	永続性のメンテナンスに使用する Java クラスを指定します。このクラスは、永続性が必要な場合に Oracle のコンテナによりコールされます。

次の例では、Oracle PSI-RI の永続性プロバイダを定義しています。論理名は「PSI-RI」として定義されており、永続性管理のためにコールされるクラスは `oracle.aurora.ejb.persistence.ocmp.OcmpEntityDeployer` です。

```
<persistence-provider>
  <description> specifies a type of persistence manager </description>
  <persistence-name>psi-ri</persistence-name>
  <persistence-deployer>oracle.aurora.ejb.persistence.ocmp.OcmpEntityDeployer
</persistence-deployer>
</persistence-provider>
```

永続フィールド 永続フィールドは、EJB ディプロイメント・ディスクリプタの `<cmp-field>` 要素内で定義されています。これらの各 `<cmp-field>` 要素は、プロバイダで永続的に管理する必要があります。`<persistence-descriptor>` 要素を介して、単一 Bean 内のすべての `<cmp-field>` 要素に対して単一のプロバイダを定義します。Bean のプロバイダは、`<persistence-descriptor>` 内で Bean とプロバイダの論理名の組合せを介して定義します。これらのフィールドは、次のとおりです。

永続性要素	説明
<description>	永続性要素のテキスト記述を指定します。
<ejb-name>	永続フィールドを含む EJB。これは、 <code><cmp-field></code> 要素を定義したエンティティ Bean の場合と同じ <code><ejb-name></code> です。
<persistence-name>	<code><persistence-provider></code> <code><persistence-name></code> 要素内で定義した永続性プロバイダの論理名。

たとえば、発注 Bean 内のすべての永続フィールドを Oracle PSI-RI で管理するように定義するには、次の各フィールドで論理 Bean 名 `PurchaseOrder` と論理永続性プロバイダ名 `Oracle PSI-RI` を指定します。

```
<persistence-descriptor>
  <ejb-name>PurchaseOrder</ejb-name>
  <persistence-name>psi-ri</persistence-name>
  . . .
</persistence-descriptor>
```

Oracle PSI-RI では、永続データ型がデータベース行にマップされます。SQL を使用して、Bean とデータベース間で永続フィールドが更新されます。各永続データ型は、『Oracle8i SQLJ 開発者ガイドおよびリファレンス』に記載されている Oracle SQLJ のデータ型にマップされます。データ型のマッピングの詳細は、5-2 ページの表 5-1 「ホスト式でサポートされている型の対応」を参照してください。唯一の制限事項は、次のデータ型がサポートされないことです。

表 A-2 永続変数に関してサポートされない Java の型

	Java の型	Oracle の型定義	Oracle のデータ型
SQLJ のストリーム・クラス	■ sqlj.runtime.BinaryStream	■ LONGVARBINARY	■ LONG RAW
	■ sqlj.runtime AsciiStream	■ LONGVARCHAR	■ LONG
	■ sqlj.runtime.UnicodeStream	■ LONGVARCHAR	■ LONG
Oracle の拡張機能	■ oracle.sql.ROWID	■ ROWID	■ ROWID
	■ oracle.sql.BLOB	■ BLOB	■ BLOB
	■ oracle.sql.CLOB	■ CLOB	■ CLOB
問合せ結果オブジェクト	■ java.sql.ResultSet	■ CURSOR	■ CURSOR
	■ SQLJ iterator objects	■ CURSOR	■ CURSOR

PSI-RI の使用を決定した場合は、<persistence-descriptor> にある <psi-ri> 要素内で次の要素を定義する必要があります。

永続性要素	説明
<schema>	データベース表が格納されているスキーマを定義します。
<table>	永続フィールドが保存される表を定義します。
<attr-mapping>	この要素内で各コンテナ管理フィールドおよび対応する表の行を定義します。
<field-name>	Bean 内で定義されているコンテナ管理フィールド名を定義します。
<column-name>	<field-name> が保存されるデータベース行を定義します。

注意： この Bean を配置する前に、表とその適切な行がスキーマに存在している必要があります。

たとえば、次の <psi-ri> では、顧客 Bean の永続フィールドである主キー、顧客名および住所を、SCOTT のスキーマ内の customers 表にマップしています。

```
<persistence-descriptor>
  <ejb-name>PurchaseOrder</ejb-name>
  <persistence-name>psi-ri</persistence-name>
  <psi-ri>
    <schema>SCOTT</schema>
    <table>customers</table>
    <attr-mapping>
      <field-name>custid</field-name>
      <column-name>cust_id</column-name>
    </attr-mapping>
    <attr-mapping>
      <field-name>name</field-name>
      <column-name>cust_name</column-name>
    </attr-mapping>
    <attr-mapping>
      <field-name>addr</field-name>
      <column-name>cust_addr</column-name>
    </attr-mapping>
  </psi-ri>
</persistence-descriptor>
```

定義する永続フィールドがオブジェクトであれば、Oracle PSI-RI に対して、永続として定義された一部またはすべてのフィールドを、単一のデータベース列にシリアライズできます。この列は、long Raw または Raw 型として定義する必要があります。

注意： すべての Raw 型の列には、4 KB という制限があります。

<serialize-mapping> 要素内で、各フィールドと宛先の列を定義します。永続フィールドは、<attr-mapping> または <serialize-mapping> 要素のどちらかでマップされます。両方ではマップできません。

次の例では、顧客 Bean の永続フィールドである顧客名と住所をシリアライズし、custinfo 列に保存しています。通常、この操作は、オブジェクトをシリアライズする必要がある場合に行います。

注意： 1つ制限事項があり、<serialize-mapping> 要素には主キーを挿入できません。挿入すると、findByPrimaryKey を介してオブジェクトを検索できなくなります。

```
<persistence-descriptor>
  <ejb-name>PurchaseOrder</ejb-name>
  <persistence-name>psi-ri</persistence-name>
  <psi-ri>
    <schema>SCOTT</schema>
    <table>customers</table>
    <serialize-mapping>
      <field-name>name</field-name>
      <field-name>addr</field-name>
      <column-name>custinfo</column-name>
    </serialize-mapping>
  </psi-ri>
</persistence-descriptor>
```

例 A-8 コンテナ管理の永続性

次の例は、顧客の例でコンテナ管理の永続性を定義する詳細を示しています。

```
<persistence-provider>
  <description>use the simple CMP provider</description>
  <persistence-name>psi-ri</persistence-name>
  <persistence-deployer>oracle.aurora.ejb.persistence.ocmp.OcmpEntityDeployer
  </persistence-deployer>
</persistence-provider>

<persistence-descriptor>
  <ejb-name>customerbean</ejb-name>
  <persistence-name>psi-ri</persistence-name>
  <psi-ri>
    <schema>SCOTT</schema>
    <table>customers</table>
    <attr-mapping>
      <field-name>custid</field-name>
      <column-name>cust_id</column-name>
    </attr-mapping>
    <attr-mapping>
      <field-name>name</field-name>
      <column-name>cust_name</column-name>
    </attr-mapping>
    <attr-mapping>
      <field-name>addr</field-name>
```

```
        <column-name>cust_addr</column-name>
    </attr-mapping>
</psi-ri>
</persistence-descriptor>
</oracle-descriptor>
```

EJB クライアントの JAR セクション

deployejb ツールで実行されるタスクの 1 つは、必要なスタブとスケルトンを含む JAR ファイルを作成することです。このファイルは、クライアントにより実行時に使用されます。この JAR ファイル名を XML ディプロイメント・ディスクリプタの `<ejb-client-jar>` 要素で指定しなければ、この JAR ファイルのデフォルト名は `server_generated.jar` となります。この JAR ファイルは、クライアントで実行できるように CLASSPATH に挿入する必要があります。

次の例では、deployejb で `myClient.jar` 内にクライアントのスタブとスケルトンを作成するように定義しています。

```
<ejb-client-jar>myClient.jar</ejb-client-jar>
```

Oracle 固有のディプロイメント・ディスクリプタの DTD

```
<!--
This is the XML DTD for the Oracle Specific EJB deployment descriptor
-->
<!--
The oracle-descriptor element is the root element of the Oracle-specific
deployment descriptor. It contains an optional description, optional structural
information about logical name mappings, optional definitions for run-as
beans and/or methods, and definitions of container-managed persistence. -->
<!ELEMENT oracle-descriptor (mappings*, run-as*, persistence-provider*,
persistence-descriptor*)>
<!--
The mappings section enables you to map logical names defined in the XML
deployment descriptor to actual names. Bean logical names are mapped to JNDI
names; security logical names are mapped to database roles or users. -->
<!ELEMENT mappings (ejb-mapping*, security-role-mapping*,
resource-ref-mapping*, transaction-manager*)>
<!--
The ejb-mapping element maps an EJB to its bound JNDI name -->
<!ELEMENT ejb-mapping (ejb-name, jndi-name)>
<!--
The security-role-mapping element maps security logical names to a database
role or user -->
<!ELEMENT security-role-mapping (security-role, oracle-role)>
<!--
The resource-ref-mapping element maps any environment variable defined in the
```

```

XML deployment descriptor to the JNDI name for the target object -->
<!--ELEMENT resource-ref-mapping (res-ref-name, jndi-name)>
<!--
The transaction manager defines the UserTransaction JNDI name that manages
the global transaction. This is only required for transactions that use two-phase
commit -->
<!--ELEMENT transaction-manager (description?, jndi-name)>
<!--
The jndi-name element specifies a JNDI name for a bound object -->
<!--ELEMENT jndi-name (#PCDATA)>
<!--
The run-as element enables you to specify a bean or certain methods within
a bean to run with an identity other than its own. The modes allowed are
CLIENT_IDENTITY (default), SYSTEM_IDENTITY, and SPECIFIED_IDENTITY. With the
SPECIFIED_IDENTITY mode, you must provide the identity within the <security-role>
element.
-->
<!--ELEMENT run-as (description?, mode, security-role, method)>
<!--
The mode element specifies the type of <run-as> identity. The values
can be one of the following:SYSTEM_IDENTITY, SPECIFIED_IDENTITY, CLIENT_IDENTITY
if mode is SPECIFIED_IDENTITY, security-role must be specified
if mode is SYSTEM_IDENTITY or CLIENT_IDENTITY and security-role is
specified, security-role is ignored
-->
<!--ELEMENT mode (#PCDATA)>
<!--
The security-role element specifies a database role -->
<!--ELEMENT security-role (description?, role-name)>
<!-- The role-name element specifies a database role or user -->
<!--ELEMENT role-name (#PCDATA)>
<!--
The method element defines a method by the bean's logical name, optionally
adding the interface name, the method name, and if overloading is present for
this method, the parameters of the method you are indicating.
-->
<!--ELEMENT method (description?, ejb-name, method-intf?, method-name,
method-params?)>
<!--
The ejb-name element defines the logical name for the bean that was used in the XML
deployment descriptor -->
<!--ELEMENT ejb-name (#PCDATA)>
<!--The method interface defines where the method is specified-->
<!--ELEMENT method-intf (#PCDATA)>
<!--
The method name element takes in the actual name of a method defined. -->
<!--ELEMENT method-name (#PCDATA)>

```

```
<!--
The method-params element specifies one or more parameters for a method. -->
<!ELEMENT method-params (method-param*)>
<!--
The method-param defines a single parameter for a method by its class type-->
<!ELEMENT method-param (#PCDATA)>
<!-- The oracle-role element specifies a database role or user -->
<!ELEMENT oracle-role (#PCDATA)>
<!-- The ejb-ref-name is the logical name for the EJB reference specified
in the XML deployment descriptor -->
<!ELEMENT ejb-ref-name (#PCDATA)>
<!-- The res-ref-name element is the logical name for the resource reference
specified in the XML deployment descriptor -->
<!ELEMENT res-ref-name (#PCDATA)>

<!---
persistence-provider describes the container managed persistence
-->
<!--
The persistence-provider element specifies the CMP provider that you are using.
At this time, only Oracle8i's PSI-RI is supported. -->
<!ELEMENT persistence-provider (description?, persistence-name,
persistence-deployer)>
<!ELEMENT description (#PCDATA)>
<!--
The persistence-name element defines the name of the provider. For Oracle8i,
this should be psi-ri -->
<!ELEMENT persistence-name (#PCDATA)>
<!-- The persistence-deployer is the class of the CMP provider. This should be
oracle.aurora.ejb.persistence.ocmp.OcmpEntityDeployer. -->
<!ELEMENT persistence-deployer (#PCDATA)>
<!-- The persistence-descriptor element defines the persistence fields in the bean
that must be managed by the CMP provider -->
<!ELEMENT persistence-descriptor (description?, ejb-name,
persistence-name, persistence-param*, psi-ri*)>
<!ELEMENT persistence-param (#PCDATA)>
<!-- The psi-ri element defines how the persistence fields in the beans are mapped
to database tables and columns. -->
<!ELEMENT psi-ri (schema, table, attr-mapping+, serialize-mapping?)>
<!-- The schema element specifies the schema where the table exists -->
<!ELEMENT schema (#PCDATA)>
<!-- The table element specifies the table where to store the persistent fields -->
<!ELEMENT table (#PCDATA)>
<!-- The attr-mapping element specifies how each persistent field is mapped to a
corresponding column in the table -->
<!ELEMENT attr-mapping (field-name, column-name)>
<!-- If you serialize all persistent fields into a single column, use the
```

```
serialize-mapping element -->
<!ELEMENT serialize-mapping (field-name+, column-name)>
<!-- The field-name element specifies the persistent variable in the bean -->
<!ELEMENT field-name (#PCDATA)>
<!-- The column-name element specifies the column for a single persistent field -->
<!ELEMENT column-name (#PCDATA)>
```

コード例 : EJB

Oracle8i JVM では、複数のサンプルが demo ディレクトリにインストールされています。この付録には、これらのサンプルの一部を記載しています。

demo ディレクトリにある例には、コンパイルして実行できるように、UNIX 用には Make ファイル、Windows NT 用にはバッチ・ファイルが付属しています。例を実行するには、標準の EMP および DEPT デモ表を含む、Java を使用できる Oracle8i データベースが必要です。

これらの簡単な例では、詳細な Java コーディング手法ではなく、ORB および CORBA の機能の説明に重点を置いています。各例には、例に含まれるファイルの一覧、実行内容およびコンパイル方法と実行方法が記載された README ファイルが付属しています。

- [基本例](#)
- [SQLJ の例](#)
- [Bean の継承例](#)
- [エンティティ Bean の例](#)
- [セッションの例](#)
- [SSL の例](#)

基本例

README

Overview

=====

This is the most basic program that you can create for the Oracle8i EJB server. One bean, HelloBean, is implemented. The bean and associated classes are loaded into the database, and the bean home interface is published as /test/myHello, as specified in the bean deployment descriptor hello.ejb.

The bean contains a single method: helloWorld, which simply returns a String containing the JavaVM version number to the client that invokes it.

This example shows the minimum number of files that you must provide to implement an EJB application: five. The five are:

- (1) the bean implementation: helloServer/HelloBean.java in this example
- (2) the bean remote interface: hello/Hello.java
- (3) the bean home interface: hello/HelloHome.java
- (4) the deployment descriptor: hello.ejb
- (5) a client app or applet: Client.java is the application in this example

Source Files

=====

Client.java

You invoke the client program from a command prompt, and pass it four arguments, the

- service URL (service ID, hostname, port, and SID if port is a listener)
- name of the published bean to lookup and instantiate
- username
- password that authenticates the client to the Oracle8i database server

For example:

```
% java -classpath LIBS Client sess_iiop://localhost:2222 /test/myHello scott tiger
```

where LIBs is the classpath that must include

```
$ORACLE_HOME/lib/aurora_client.jar
#If using Java 2, use classes12.zip instead of classes111.zip
$ORACLE_HOME/jdbc/lib/classes111.zip
$ORACLE_HOME/lib/vbjdbc.jar
$ORACLE_HOME/lib/vbjapp.jar
$JAVA_HOME/lib/classes.zip
```

(Note: for NT users, the environment variables would be %ORACLE_HOME% and %JAVA_HOME%.)

The client code performs the following steps:

- gets the arguments passed on the command line
- creates a new JNDI Context (InitialContext())
- looks up the published bean to find and activate its home interface
- using the home interface, instantiates through its create() method a new bean object, hello
- invokes the helloWorld() method on the hello object and prints the results

The printed output is:

Hello client, your javavm version is 8.1.5.

```
hello.ejb
-----
```

The bean deployment descriptor. This source file does the following:

- shows the class name of the bean implementation in the deployment name: helloServer.HelloBean
- names the published bean "/test/myHello"
- declares the remote interface implementation: hello.Hello
- declares the home interface: hello.HelloHome
- sets RunAsMode to the client's identity (SCOTT in this case)
- allows all members of the group PUBLIC to run the bean
- sets the transaction attribute to TX_SUPPORTS

The deployment descriptor is read by the deployejb tool, which uses it to load the required classes, and publish the bean home interface. (Deployejb does much else also. See the Tools chapter in the Oracle8i EJB and CORBA Developer's Guide for more information.)

helloServer/HelloBean.java

This is the EJB implementation. Note that the bean class is public, and that it implements the SessionBean interface, as required by the EJB specification.

The bean implements the one method specified in the remote interface: helloWorld(). This method gets the system property associated with "oracle.server.version" as a String, and returns a greeting plus the version number as a String to the invoking client.

The bean implementation also implements ejbCreate() with no parameters, following the specification of the create() method in hello/HelloHome.java.

Finally, the methods ejbRemove(), setSessionContext(), ejbActivate(), and ejbPassivate() are implemented as required by the SessionBean interface. In this simple case, the methods are implemented with null bodies.

(Note that ejbActivate() and ejbPassivate() are never called in the 8.1.5 release of the EJB server, but they must be implemented as required by the interface.)

hello/Hello.java

This is the bean remote interface. In this example, it specifies only one method: helloWorld(), which returns a String object. Note the two import statements, which are required, and that the helloWorld() method must be declared as throwing RemoteException. All bean methods must be capable of throwing this exception. If you omit the declaration, the deployejb tool will catch it and error when you try to deploy the bean.

hello/HelloHome.java

This is the bean home interface. In this example, a single create() method is declared. It returns a Hello object, as you saw in the Client.java code.

Note especially that the create() method must be declared as able to throw RemoteException and CreateException. These are required. If you do not declare these, the deployejb tool will catch it and error when

you try to deploy the bean.

Compiling and Running the Example =====

UNIX ----

Enter the command 'make all' or simply 'make' in the shell to compile, load, and deploy the objects, and run the client program. Other targets are 'run' and 'clean'.

Make sure that a shell environment variable ORACLE_HOME is set to point to the home location of the Oracle installation. This is operating system dependent, so see the Installation documentation that came with your system for the location. Also, review the README file for the Oracle database, and the README file for the CORBA/EJB server (the Oracle8i ORB), for additional up-to-date information.

Windows NT -----

On Windows NT, run the batch file makeit.bat from a DOS command prompt to compile, load, and deploy the objects. Run the batch file runit.bat to run the client program, and see the results.

Make sure that the environment variables %ORACLE_HOME%, %CLASSPATH%, and %SERVICE% are set appropriately for the DOS command window. You can set these as either user or system environment variables from the Control Panel. Double click on System in the Control Panel then on the Environment tab to set these variables. Start a new DOS window after setting environment variable values.

See the Installation documentation that came with your Oracle8i system for the values of these variables. Also, review the README file for the Oracle database, and the README file for the CORBA/EJB server (the Oracle8i ORB), for additional up-to-date information.

You can also set an environment variable %JAVA_HOME% to point to the root of your Java JDK. For example, SET JAVA_HOME=C:\JDK1.1.6.

クライアント

```
import hello.Hello;
import hello.HelloHome;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;

public class Client
{
    public static void main (String[] args) throws Exception {
        if (args.length != 4) {
            System.out.println ("usage: Client serviceURL objectName user password");
            System.exit (1);
        }
        String serviceURL = args [0];
        String objectName = args [1];
        String user = args [2];
        String password = args [3];

        Hashtable env = new Hashtable ();
        env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
        env.put (Context.SECURITY_PRINCIPAL, user);
        env.put (Context.SECURITY_CREDENTIALS, password);
        env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
        Context ic = new InitialContext (env);

        HelloHome hello_home = (HelloHome)ic.lookup (serviceURL + objectName);
        Hello hello = hello_home.create ();
        System.out.println (hello.helloWorld ());
    }
}
```

Hello のホーム・インタフェース

```
package hello;

import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import java.rmi.RemoteException;

public interface HelloHome extends EJBHome
```

```
{  
    public Hello create () throws RemoteException, CreateException;  
}
```

Hello のリモート・インタフェース

```
package hello;  
  
import javax.ejb.EJBObject;  
import java.rmi.RemoteException;  
  
public interface Hello extends EJBObject  
{  
    public String helloWorld () throws RemoteException;  
}
```

Hello の Bean 実装

```
package helloServer;  
  
import javax.ejb.SessionBean;  
import javax.ejb.CreateException;  
import javax.ejb.SessionContext;  
import java.rmi.RemoteException;  
  
public class HelloBean implements SessionBean  
{  
    // Methods of the Hello interface  
    public String helloWorld () throws RemoteException {  
        String v = System.getProperty("oracle.server.version");  
        return "Hello client, your javavm version is " + v + ".";  
    }  
  
    // Methods of the SessionBean  
    public void ejbCreate () throws RemoteException, CreateException {}  
    public void ejbRemove() {}  
    public void setSessionContext (SessionContext ctx) {}  
    public void ejbActivate () {}  
    public void ejbPassivate () {}  
}
```

SQLJ の例

README

Overview

=====

This example demonstrates doing a database query using SQLJ. pay attention to the makefile (UNIX) or the makeit.bat batch file (Windows NT), and note that the files that SQLJ generates (SER files converted to class files) must be loaded into the database with deployejb also.

Compare this example with the jdbcimpl basic EJB example, which uses JDBC instead of SQLJ to perform exactly the same query.

Source files

=====

Client.java

Invoke the client program from the command line, passing it four arguments:

- the name of the service URL, e.g. sess_iiop://localhost:2222
- the path and name of the published bean, e.g. /test/employeeBean
- the username for db authentication
- the password (you wouldn't do this in a production program, of course)

For example

```
% java Client -classpath LIBs sess_iiop://localhost:2222 /test/employeeBean  
    scott tiger
```

The client looks up and activates the bean, then invokes the query() method on the bean. query() returns an EmpRecord structure with the salary and the name of the employee whose ID number was passed to query().

There is no error checking in this code. See the User's Guide for more information about the appropriate kinds of error checking in this kind of client code.

The client prints:

```
Emp name is ALLEN
Emp sal  is 3100.0
```

```
employeeServer/employeeBean.sqlj
```

This class is the bean implementation. A SQLJ named iterator is declared to hold the results of the query. The `myIter.next();` statement is used as is to keep the code simple: after all the parameter passed in is a known valid primary key for the EMP table. (See what happens if you try an empno that is not in the table.)

The `EmpIter` getter methods are used to retrieve the query results into the `EmpRecord` object, which is then returned **by value**, as a serialized object, to the client.

```
employeeServer/EmpRecord.java
```

A class that is in essence a struct to contain the employee name and salary, as well as the ID number.

Note that the class **must** be defined as implementing the `java.rmi.Serializable` interface, to make it a valid serializable RMI object that can be passed from server to the client.

```
employee/employee.java
```

The bean remote interface.

```
employee/employeeHome.java
```

The bean home interface.

Compiling and Running the Example

=====

UNIX

Enter the command 'make all' or simply 'make' in the shell to compile, load, and deploy the objects, and run the client program. Other targets are 'run' and 'clean'.

Make sure that a shell environment variable ORACLE_HOME is set to point to the home location of the Oracle installation. This is operating system dependent, so see the Installation documentation that came with your system for the location. Also, review the README file for the Oracle database, and the README file for the CORBA/EJB server (the Oracle8i ORB), for additional up-to-date information.

Windows NT

On Windows NT, run the batch file makeit.bat from a DOS command prompt to compile, load, and deploy the objects. Run the batch file runit.bat to run the client program, and see the results.

Make sure that the environment variables %ORACLE_HOME%, %CLASSPATH%, and %SERVICE% are set appropriately for the DOS command window. You can set these as either user or system environment variables from the Control Panel. Double click on System in the Control Panel then on the Environment tab to set these variables. Start a new DOS window after setting environment variable values.

See the Installation documentation that came with your Oracle8i system for the values of these variables. Also, review the README file for the Oracle database, and the README file for the CORBA/EJB server (the Oracle8i ORB), for additional up-to-date information.

You can also set an environment variable %JAVA_HOME% to point to the root of your Java JDK. For example, SET JAVA_HOME=C:\JDK1.1.6.

クライアント

```
import employee.Employee;
import employee.EmployeeHome;
import employee.EmpRecord;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;

public class Client {
    public static void main (String [] args) throws Exception {
        if (args.length != 4) {
            System.out.println("usage: Client serviceURL objectName user password");
            System.exit(1);
        }
        String serviceURL = args [0];
        String objectName = args [1];
        String user = args [2];
        String password = args [3];

        Hashtable env = new Hashtable();
        env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
        env.put (Context.SECURITY_PRINCIPAL, user);
        env.put (Context.SECURITY_CREDENTIALS, password);
        env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
        Context ic = new InitialContext (env);

        EmployeeHome home = (EmployeeHome)ic.lookup (serviceURL + objectName);
        Employee testBean = home.create();
        EmpRecord empRec = empRec = testBean.query (7499);
        System.out.println ("Emp name is " + empRec.ename);
        System.out.println ("Emp sal is " + empRec.sal);
    }
}
```

ホーム・インタフェース

```
package employee;

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

public interface EmployeeHome extends EJBHome {
    public Employee create()
        throws CreateException, RemoteException;
}
```

リモート・インタフェース

```
package employee;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Employee extends EJBObject {
    public EmpRecord query (int empNumber)
        throws java.sql.SQLException, RemoteException;
}
```

Bean 実装

EmployeeBean.sqlj

```
package employeeServer;

import employee.EmpRecord;

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

public class EmployeeBean implements SessionBean {
    //SessionContext ctx;

    public void ejbCreate() throws CreateException, RemoteException {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbRemove() {}
    public void setSessionContext(SessionContext ctx) {
        //this.ctx = ctx;
    }
}
```

```
public EmpRecord query (int empNumber) throws SQLException, RemoteException
{
    String ename;
    double sal;

    #sql { select ename, sal into :ename, :sal from emp
          where empno = :empNumber };
    System.out.println ("ename = " + ename);
    System.out.println ("sal   = " + sal);

    return new EmpRecord (ename, empNumber, sal);
}
}
```

EmpRecord.java

```
package employee;

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

    public EmpRecord (String ename, int empno, double sal) {
        this.ename = ename;
        this.empno = empno;
        this.sal = sal;
    }
}
```

Bean の継承例

README

Overview
=====

This example show two beans: Foo and Bar. In the example, the Bar bean inherits from the Foo bean. The required coding and the effects of this bean inheritance are demonstrated in this example.

Source Files
=====

Client.java

You invoke the client program from a command prompt, and pass it four arguments, the

- service URL (service ID, hostname, port, and SID if port is a listener)
- name of the published bean to lookup and instantiate
- username
- password that authenticates the client to the Oracle8i database server

For example:

```
% java -classpath LIBs Client sess_iiop://localhost:2222 /test/myHello scott tiger
```

where LIBs is the classpath that must include

```
$ORACLE_HOME/lib/aurora_client.jar  
#If using Java 2, use classes12.zip instead of classes111.zip  
$ORACLE_HOME/jdbc/lib/classes111.zip  
$ORACLE_HOME/lib/vbjorb.jar  
$ORACLE_HOME/lib/vbjapp.jar  
$JAVA_HOME/lib/classes.zip
```

(Note: for NT users, the environment variables would be %ORACLE_HOME% and %JAVA_HOME%.)

The client code performs the following steps:

- gets the arguments passed on the command line
- creates a new JNDI Context (InitialContext())
- looks up the published bean to find and activate its home interface
- using the home interface, instantiates through its create() method a new bean object, hello
- invokes the helloWorld() method on the hello object and prints the results

The printed output is:

```
Hello World  
Hello World from bar  
Hello World 2 from bar  
Hello World from bar
```

foo.ejb

The Foo bean deployment descriptor. See ../helloworld/readme.txt for a more complete description of a typical example deployment descriptor.

bar.ejb

The bar bean deployment descriptor.

inheritance/FooHome.java

The Foo bean home interface. Specifies a single no-parameter create() method.

inheritance/Foo.java

The Foo remote interface. Note that only a single method, hello(), is specified.

inheritance/BarHome.java

The Bar bean home interface. Specifies a single no-parameter create() method.

inheritance/Bar.java

The Bar remote interface. Note that only a single method, hello2(), is specified.

inheritanceServer/FooBean.java

The Foo bean implementation. Implements the hello() method of inheritance/Foo.java, returning a String greeting.

inheritanceServer/BarBean.java

The Bar bean implementation. Implements both the hello() method inherited from FooBean, as well as the hello2() method specified in inheritance/Bar.java.

Note that this bean extends FooBean, so it does not implement SessionBean or any of its methods, such as ejbRemove(0, ejbActivate()), and so on, which is normally a requirement of a session bean. This is because BarBean inherits the implementation of these from FooBean.

Compiling and Running the Example

=====

UNIX

Enter the command 'make all' or simply 'make' in the shell to compile, load, and deploy the objects, and run the client program. Other targets are 'run' and 'clean'.

Make sure that a shell environment variable ORACLE_HOME is set to point to the home location of the Oracle installation. This is operating system dependent, so see the Installation documentation that came with your system for the location. Also, review the README file for the Oracle database, and the README file for the CORBA/EJB server (the Oracle8i ORB), for additional up-to-date information.

Windows NT

On Windows NT, run the batch file makeit.bat from a DOS command prompt to compile, load, and deploy the objects. Run the batch file runit.bat to run the client program, and see the results.

Make sure that the environment variables %ORACLE_HOME%, %CLASSPATH%, and %SERVICE% are set appropriately for the DOS command window. You can set these as either user or system environment variables from the Control Panel. Double click on System in the Control Panel then on the Environment tab to set these variables. Start a new DOS window after setting environment variable values.

See the Installation documentation that came with your Oracle8i system for the values of these variables. Also, review the README file for the Oracle database, and the README file for the CORBA/EJB server (the Oracle8i ORB), for additional up-to-date information.

You can also set an environment variable %JAVA_HOME% to point to the root of your Java JDK. For example, SET JAVA_HOME=C:\JDK1.1.6.

クライアント

```
import inheritanceServer.*;
import inheritance.*;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;

public class Client
{
    public static void main(String[] args) throws Exception {
        if (args.length != 5) {
            System.out.println("usage: Client serviceURL fooBeanName "
+ "barBeanName username password");
            System.exit(1);
        }

        String serviceURL = args [0];
        String fooBeanName = args [1];
        String barBeanName = args[2];
        String username = args[3];
        String password = args[4];

        Hashtable env = new Hashtable();
        env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
        env.put(Context.SECURITY_PRINCIPAL, username);
        env.put(Context.SECURITY_CREDENTIALS, password);
        env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
        Context ic = new InitialContext(env);

        // Get a foo object from a foo published bean
        FooHome home = (FooHome) ic.lookup(serviceURL + fooBeanName);
        Foo foo = home.create();
        System.out.println(foo.hello());
    }
}
```

```

        // Get a bar object from a bar published bean
        BarHome barHome = (BarHome) ic.lookup(serviceURL + barBeanName);
        Bar bar = barHome.create();
        System.out.println(bar.hello());
        System.out.println(bar.hello2());

        // Get a foo object from a bar published bean
        BarHome fooBarHome = (BarHome) ic.lookup(serviceURL + barBeanName);
        Foo fooBar = (Foo) fooBarHome.create();
        System.out.println(fooBar.hello());
    }
}

```

ホーム・インタフェース

BarHome.java

```

package inheritance;

import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import java.rmi.RemoteException;

public interface FooHome extends EJBHome
{
    public Foo create () throws RemoteException, CreateException;
}

```

FooHome.java

```

package inheritance;

import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import java.rmi.RemoteException;

public interface FooHome extends EJBHome
{
    public Foo create () throws RemoteException, CreateException;
}

```

リモート・インタフェース

Bar.java

```
package inheritance;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Foo extends EJBObject
{
    public String hello () throws RemoteException;
}
```

Foo.java

```
package inheritance;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Foo extends EJBObject
{
    public String hello () throws RemoteException;
}
```

Bean 実装

BarBean.java

```
package inheritanceServer;

import java.rmi.RemoteException;
import javax.ejb.*;
import oracle.aurora.jndi.sess_iiop.*;

public class FooBean implements SessionBean
{
    // Methods of the interface
    public String hello () throws RemoteException {
        return "Hello World";
    }

    // Methods of the SessionBean
    public void ejbCreate () throws RemoteException, CreateException {
    }
}
```

```

    public void ejbRemove() {
    }

    public void setSessionContext (SessionContext ctx) {
    }

    public void ejbActivate () {
    }

    public void ejbPassivate () {
    }
}

```

FooBean.java

```

package inheritanceServer;

import java.rmi.RemoteException;
import javax.ejb.*;
import oracle.aurora.jndi.sess_iiop.*;

public class FooBean implements SessionBean
{
    // Methods of the interface
    public String hello () throws RemoteException {
        return "Hello World";
    }

    // Methods of the SessionBean
    public void ejbCreate () throws RemoteException, CreateException {
    }

    public void ejbRemove() {
    }

    public void setSessionContext (SessionContext ctx) {
    }

    public void ejbActivate () {
    }

    public void ejbPassivate () {
    }
}

```

エンティティ Bean の例

次の 2 つの例は、Bean 管理またはコンテナ管理のオプションを使用して、エンティティ Bean を実装する方法を示しています。

- [Bean 管理のエンティティ Bean の例](#)
- [コンテナ管理のエンティティ Bean の例](#)

Bean 管理のエンティティ Bean の例

クライアント

```
import purchase.PurchaseOrder;
import purchase.PurchaseOrderHome;
import purchase.LineItem;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import java.sql.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.transaction.UserTransaction;
import java.util.*;

import oracle.jdbc.driver.*;
import oracle.aurora.jndi.jdbc_access.jdbc_accessURLContextFactory;

public class Client {
    public static void main (String [] args) throws Exception {
        System.out.println("Running client");
        if (args.length != 6) {
            System.out.println("usage: Client serviceURL jdbcURL objectName utName user
password");
            System.exit(1);
        }
        String serviceURL = args [0];
        String jdbcURL = args [1];
        String objectName = args [2];
        String utName = args [3];
        String user = args [4];
        String password = args [5];

        Hashtable env = new Hashtable();
        env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
        env.put (jdbc_accessURLContextFactory.CONNECTION_URL_PROP, jdbcURL);
        env.put (Context.SECURITY_PRINCIPAL, user);
```

```

env.put(Context.SECURITY_CREDENTIALS, password);
DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver ());
env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
Context ic = new InitialContext (env);

PurchaseOrderHome home =
    (PurchaseOrderHome)ic.lookup (serviceURL + objectName);

// Begin a transaction to create a new PO
UserTransaction ut;
ut = (UserTransaction)ic.lookup ("local://UserTransaction:iiop");
// ut = (UserTransaction)ic.lookup ("jdbc_access://" + utName);
ut.begin();

// Create a new PO and add items to it
PurchaseOrder po = home.create();
po.addItem (111111, 2);
po.addItem (333333, 4);

// Price the PO
System.out.println ("PO price $" + po.price ());

// Get the po number for future reference
String ponumber = (String)po.getPrimaryKey ();

// Commit the transaction
ut.commit();

// This is now the future:

// Start another transaction
ut.begin ();

// Retrieve the PO from its primary key
PurchaseOrder po2 = home.findByPrimaryKey(ponumber);

// Add another item
po.addItem (222222, 1);

// Check the PO contents
System.out.println ("Contents of the PO:");
Vector items = po.getContents ();
Enumeration e = items.elements ();
while (e.hasMoreElements ()) {
    LineItem item = (LineItem)e.nextElement ();
    System.out.println (item.quantity + " " + item.description + " at $"
+ (int)item.price + " each");
}

```

```

    }

    // Compute the price again
    System.out.println ("PO price $" + po.price ());

    // Rollback the change
    ut.rollback ();
}

}

```

ホーム・インタフェース

```

import javax.ejb.*;
import java.rmi.RemoteException;
import java.sql.SQLException;

public interface PurchaseOrderHome extends EJBHome {
    // Create a new PO
    public PurchaseOrder create() throws CreateException, RemoteException;

    // Find an existing one
    public PurchaseOrder findByPrimaryKey (String POnumber)
        throws FinderException, RemoteException;
}

```

リモート・インタフェース

```

package purchase;

/*
    PurchaseOrder is an entity bean.

    It is a remote interface to a purchasing application.

    The remote interface lets you manage PO contents (getContents,
    addItem) and execute complex logic on the server side (price).

*/

import javax.ejb.EJBObject;
import java.rmi.RemoteException;
import java.sql.SQLException;
import java.util.Vector;

public interface PurchaseOrder extends EJBObject {

```

```
// Price the PO
public float price() throws RemoteException;

// Manage contents

// getContents returns a Vector of LineItem objects
public Vector getContents() throws RemoteException;

public void addItem (int sku, int count) throws RemoteException;
}
```

Bean 実装

PurchaseOrderBean.sqlj

```
package purchaseServer;

/*
   This is the PurchaseOrder Bean.

   The bean manages its own persistence. The PO line items are stored
   in the LINEITEMS table. This table references the SKUS table that
   contain individual pricing information.

*/

import purchase.*;
import java.sql.*;
import java.rmi.RemoteException;
import javax.ejb.*;
import java.util.*;

#sql iterator ItemsIter (int skunumber, int count, String description,
float price);

public class PurchaseOrderBean implements EntityBean {
    EntityContext ctx;

    Vector items;// The items in the PO (instances of LineItem)

    public void PurchaseOrderBean() {}

    // Bean Managed Persistence methods

    // The create methods takes care of generating a new PO and returns
    // its primary key
}
```

```
public String ejbCreate () throws CreateException, RemoteException
{
    String ponumber = null;
    try {
        #sql { select ponumber.nextval into :ponumber from dual };
        #sql { insert into pos (ponumber, status) values (:ponumber, 'OPEN') };
    } catch (SQLException e) {
        throw new PurchaseException (this, "create", e);
    }
    return ponumber;
}

// Nothing to do here
public void ejbPostCreate () {
    items = new Vector ();
}

// The remove method deletes all line items belonging to the PO
public void ejbRemove() throws RemoteException {
    // Get the PO number and delete
    String ponumber = (String)ctx.getPrimaryKey();
    try {
        #sql { delete from lineitems where ponumber = :ponumber };
        #sql { delete from pos where ponumber = :ponumber };
    } catch (SQLException e) {
        throw new PurchaseException (this, "remove", e);
    }
}

// The load method populates the items array with all the existing
// line items
public void ejbLoad() throws RemoteException {
    // Get the PO number
    String ponumber = (String)ctx.getPrimaryKey();

    // Load all line items.
    try {
        items = new Vector ();
        ItemsIter iter = null;
        try {
            #sql iter = {
select lineitems.skunumber, lineitems.count,
      skus.description, skus.price
  from lineitems, skus
 where ponumber = :ponumber and lineitems.skunumber = skus.skunumber
};
```

```
        while (iter.next ()) {
            LineItem item =
                new LineItem (iter.skunumber(), iter.count(), iter.description(),
                    iter.price());
            items.addElement (item);
        }
    } finally {
        if (iter != null) iter.close ();
    }
    } catch (SQLException e) {
        throw new PurchaseException (this, "load", e);
    }
}

// The store method replaces all entries in the lineitems table with the
// new entries from the bean
public void ejbStore() throws RemoteException {
    // Get the PO number
    String ponumber = (String)ctx.getPrimaryKey();

    try {
        // Delete old entries
        #sql { delete from lineitems where ponumber = :ponumber };

        // Insert new entries
        Enumeration e = items.elements ();
        while (e.hasMoreElements ()) {
            LineItem item = (LineItem)e.nextElement ();
            #sql { insert into lineitems (ponumber, skunumber, count)
                values (:ponumber, :(item.sku), :(item.quantity))
        };
        }
    } catch (SQLException e) {
        throw new PurchaseException (this, "store", e);
    }
}

// The findByPrimaryKey method verifies that the POnumber exists
public String ejbFindByPrimaryKey (String ponumber)
    throws FinderException, RemoteException
{
    try {
        int count;
        #sql { select count (ponumber) into :count from pos
            where ponumber = :ponumber };

        // There has to be one
    }
}
```

```
        if (count != 1)
throw new FinderException ("Inexistent PO: " + ponumber);
    } catch (SQLException e) {
        throw new PurchaseException (this, "findByPrimaryKey", e);
    }
    // The ponumber is the primary key
    return ponumber;
}

// Business Methods

// Price the PO
public float price() throws RemoteException {
    float price = 0;
    Enumeration e = items.elements ();
    while (e.hasMoreElements ()) {
        LineItem item = (LineItem)e.nextElement ();
        price += item.quantity * item.price;
    }

    // 5% discount if buying more than 10 items
    if (items.size () > 10)
        price -= price * 0.05;

    // Shipping is a constant plus function of the number of items
    price += 10 + (items.size () * 2);

    return price;
}

// The getContents methods has to load the descriptions
public Vector getContents() throws RemoteException {
    return items;
}

// The add Item method gets the price and description
public void addItem (int sku, int count) throws RemoteException {
    try {
        String description;
        float price;
        #sql { select price, description into :price, :description
            from skus where skunumber = :sku };
        items.addElement (new LineItem (sku, count, description, price));
    } catch (SQLException e) {
        throw new PurchaseException (this, "addItem", e);
    }
}
```

```
// EntityBean Methods
public void setEntityContext(EntityContext ctx) { this.ctx = ctx; }
public void unsetEntityContext() {}
public void ejbActivate() {}
public void ejbPassivate() {}
}
```

LineItem.java

```
package purchase;

/*

    This class represents one line item

*/

public class LineItem implements java.io.Serializable {
    public int sku;
    public int quantity;
    public String description;
    public float price;

    public LineItem (int sku, int quantity, String description, float price) {
        this.sku = sku;
        this.quantity = quantity;
        this.description = description;
        this.price = price;
    }
}
```

ディプロイメント・ディスクリプタ

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems Inc.//DTD Enterprise JavaBeans 1.1
//EN" "ejb-jar.dtd">
<ejb-jar>
  <enterprise-beans>
    <entity>
      <description>no description</description>
      <ejb-name>test/purchase</ejb-name>
      <home>purchase.PurchaseOrderHome</home>
      <remote>purchase.PurchaseOrder</remote>
      <ejb-class>purchaseServer.PurchaseOrderBean</ejb-class>
      <persistence-type>Bean</persistence-type>
      <prim-key-class>java.lang.String</prim-key-class>
```

```

        <reentrant>False</reentrant>
    </entity>
</enterprise-beans>
<assembly-descriptor>
    <security-role>
        <description>no description</description>
        <role-name>PUBLIC</role-name>
    </security-role>
    <method-permission>
        <description>no description</description>
        <role-name>PUBLIC</role-name>
        <method>
            <ejb-name>test/purchase</ejb-name>
            <method-name>*</method-name>
        </method>
    </method-permission>
    <container-transaction>
        <description>no description</description>
        <method>
            <ejb-name>test/purchase</ejb-name>
            <method-name>*</method-name>
        </method>
        <trans-attribute>Required</trans-attribute>
    </container-transaction>
</assembly-descriptor>
</ejb-jar>

```

コンテナ管理のエンティティ Bean の例

クライアント

```

import customer.Customer;
import customer.CustomerHome;

import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.rmi.RemoteException;
import javax.ejb.RemoveException;
import javax.ejb.CreateException;
import javax.ejb.FinderException;
import java.util.Enumeration;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import java.util.Hashtable;
import javax.naming.Context;

```

```

public class CustomerClient
{
    public static void main(String[] argv)
    {
        System.out.println("client is running");
        try
        {
            if (argv.length != 4) {
                System.out.println("usage: Client serviceURL
                                   objectName user password");
                System.exit(1);
            }
            String serviceURL = argv [0];
            String objectName = argv [1];
            String user = argv [2];
            String password = argv [3];

            Hashtable env = new Hashtable();
            env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
            env.put(Context.SECURITY_PRINCIPAL, user);
            env.put(Context.SECURITY_CREDENTIALS, password);
            env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
            Context ic = new InitialContext (env);
            CustomerHome ch = (CustomerHome)ic.lookup (serviceURL + objectName);

            Customer cust = ch.create("Jake Terwilliger", "Pine Drive");
            System.out.println("created " + cust.getName());
            System.out.println (" address = " + cust.getAddress());
            String pk = (String) cust.getPrimaryKey();
            System.out.println("Primarykey = " + pk);

            //imagine that time passes here, or this program is
            //finished, and a later program wants to use the
            //primary key
            Customer cust1 = ch.create("Al Smith", "Sesame Street");

            Customer cust2 = ch.create("Bob Davidson", "Elm Street");

            Customer cust3 = ch.create("Carol Fernandez", "Cedar Blvd");

            cust = null;
            cust = ch.findByPrimaryKey(pk);
            System.out.println("Found by primary key lookup");
            System.out.println (" name = " + cust.getName());
            System.out.println (" address = " + cust.getAddress());
            cust.remove();
        }
    }
}

```

```

        System.out.println("removed bean");

        cust = ch.findByWhere("where cust_addr = 'Elm Street'");
        System.out.println("Found by findByWhere");
        System.out.println (" name = " + cust.getName());
        System.out.println (" address = " + cust.getAddress());
        cust.remove();
        System.out.println("removed bean");

        Enumeration e = ch.findAllCustomers("");
        while(e.hasMoreElements())
        {
            cust = (Customer) e.nextElement();
            System.out.println (" name = " + cust.getName());
            System.out.println (" address = " + cust.getAddress());
        }

    }
    catch (RemoveException e)
    {
        System.out.println("RemoveException caught:" + e);
        e.printStackTrace();
    }
    catch (NamingException e)
    {
        System.out.println("NamingException caught:" + e);
        e.printStackTrace();
    }
    catch (FinderException e)
    {
        System.out.println("FinderException caught:" + e);
        e.printStackTrace();
    }

    catch (CreateException e)
    {
        System.out.println("CreateException caught:" + e);
        e.printStackTrace();
    }

    catch (RemoteException e)
    {
        System.out.println("RemoteException caught:" + e);
        e.printStackTrace();
    }
}
}

```

ホーム・インタフェース

```
package customer;
import javax.ejb.EJBHome;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.FinderException;

public interface CustomerHome extends EJBHome
{
    public Customer findByPrimaryKey(String pk) throws RemoteException,
        FinderException;
    public Customer findByWhere(String whereString) throws RemoteException,
        FinderException;
    public java.util.Enumeration findAllCustomers(String whereString) throws
        RemoteException, FinderException;
    public Customer create(String custname, String custaddr) throws
        RemoteException, CreateException;
}
```

リモート・インタフェース

```
package customer;
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Customer extends EJBObject
{
    public String getName() throws RemoteException;
    public String getAddress() throws RemoteException;
    public void setAddress(String addr) throws RemoteException;
}
```

Bean 実装

```
package customerServer;
import customer.*;
import javax.ejb.*;
import java.sql.*;
import java.util.*;
import java.rmi.RemoteException;
import java.io.Serializable;

public class CustomerBean implements EntityBean
{
    private transient EntityContext ctx;
    public String name;
    public String addr;
```

```

public String getName() throws RemoteException
{
    return name;
}
public void setName(String name) throws RemoteException
{
    this.name = name;
}
public String getAddress() throws RemoteException
{
    return addr;
}
public void setAddress(String addr) throws RemoteException
{
    this.addr = addr;
}
public void setEntityContext(EntityContext ctx)
{
    this.ctx = ctx;
    Properties props = ctx.getEnvironment();

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

public String ejbCreate(String custname, String custaddr) throws CreateException,
RemoteException
{
    try {
        setName(custname);
        setAddress(custaddr);
    } catch (java.rmi.RemoteException e) {
        throw new CreateException();
    }
    return null;
}

public String ejbFindByPrimaryKey(String pk) throws RemoteException, FinderException
{
    return null;
}

public void ejbPostCreate(String custname, String custaddr) throws CreateException

```

```
{
    // get primaryKey
    String pk = (String)ctx.getPrimaryKey();
}

public void ejbActivate()
{
}

public void ejbPassivate()
{
}

public void ejbRemove()
{
}

public void ejbLoad()
{
    // You can get to the primary key
    String pk = (String)ctx.getPrimaryKey();
}

public void ejbStore()
{
}
}
```

XML ディプロイメント・ディスクリプタ

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems Inc.//DTD Enterprise JavaBeans 1.1
//EN" "ejb-jar.dtd">
<ejb-jar>
  <enterprise-beans>
    <entity>
      <ejb-name>customerbean</ejb-name>
      <home>customer.CustomerHome</home>
      <remote>customer.Customer</remote>
      <ejb-class>customerServer.CustomerBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.String</prim-key-class>
      <reentrant>False</reentrant>
```

```

<cmp-field><field-name>name</field-name></cmp-field>
<cmp-field><field-name>addr</field-name></cmp-field>
<primkey-field>name</primkey-field>
<env-entry>
  <env-entry-name>realmName</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>my.realm</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>customerBean.databaseURL</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>jdbc:oracle:kprb:</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>customerBean.JDBCDriverName</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>oracle.jdbc.driver.OracleDriver</env-entry-value>
</env-entry>
</entity>
</enterprise-beans>
<assembly-descriptor>
  <security-role>
    <role-name>PUBLIC</role-name>
  </security-role>
  <method-permission>
    <role-name>PUBLIC</role-name>
    <method>
      <ejb-name>test/customer</ejb-name>
      <method-name>*</method-name>
    </method>
  </method-permission>
  <container-transaction>
    <method>
      <ejb-name>test/customer</ejb-name>
      <method-name>*</method-name>
    </method>
    <trans-attribute>RequiresNew</trans-attribute>
  </container-transaction>
</assembly-descriptor>
</ejb-jar>

```

Oracle 固有のディプロイメント・ディスクリプタ

```
<?xml version="1.0"?>
<!DOCTYPE oracle-descriptor PUBLIC "-//Oracle Corporation.//DTD Oracle 1.1//EN"
"oracle-ejb-jar.dtd">
<oracle-descriptor>
  <mappings>
    <ejb-mapping>
      <ejb-name>customerbean</ejb-name>
      <jndi-name>test/customer2</jndi-name>
    </ejb-mapping>
  </mappings>

  <persistence-provider>
    <persistence-name>psi-ri</persistence-name>
    <persistence-deployer>oracle.aurora.ejb.persistence.ocmp.OcmpEntityDeployer
    </persistence-deployer>
  </persistence-provider>

  <persistence-descriptor>
    <ejb-name>customerbean</ejb-name>
    <persistence-name>psi-ri</persistence-name>
    <psi-ri>
      <schema>SCOTT</schema>
      <table>customers</table>
      <attr-mapping>
        <field-name>name</field-name>
        <column-name>cust_name</column-name>
      </attr-mapping>
      <attr-mapping>
        <field-name>addr</field-name>
        <column-name>cust_addr</column-name>
      </attr-mapping>
    </psi-ri>
  </persistence-descriptor>
</oracle-descriptor>
```

データベース表の更新

```
connect scott/tiger

drop table CUSTOMERS;

create table CUSTOMERS (CUST_NAME VARCHAR(64) NOT NULL, CUST_ADDR VARCHAR(64),
                        CUST_SERIALIZE LONG RAW );
```

セッションの例

README

Overview
=====

This EJB example shows how you can create a second EJB in the same server, but in a different session. The same username and password are used to create the second object, and it accesses the same published EJB.

Source Files
=====

Client.java

You invoke the client program from a command prompt, and pass it four arguments, the

- service URL (service ID, hostname, port, and SID if port is a listener)
- name of the published bean to lookup and instantiate
- username
- password that authenticates the client to the Oracle8i database server

For example:

```
% java -classpath LIBs Client sess_iiop://localhost:2481:ORCL |  
    /test/myHello scott tiger
```

where LIBs is the classpath that must include

```
$ORACLE_HOME/lib/aurora_client.jar  
#If using Java 2, use classes12.zip instead of classes111.zip  
$ORACLE_HOME/jdbc/lib/classes111.zip  
$ORACLE_HOME/lib/vbjorb.jar  
$ORACLE_HOME/lib/vbjapp.jar  
$JAVA_HOME/lib/classes.zip
```

The client code performs the following steps:

- gets the arguments passed on the command line
- creates a new JNDI Context (InitialContext())
- looks up the published bean to find and activate its home interface
- using the home interface, instantiates through its create()

- method a new bean object, hello
- sets the hello bean's message to "Hello World!"
- asks the first hello bean to create another bean, by invoking the `getOtherHello()` method, passing it the authentication, service URL, and bean name parameters
- invokes `otherHelloWorld()` on the first bean, and printing its return value, which is derived from the second created bean

The printed output is:

```
Hello World!
Hello from the Other HelloBean Object
```

```
hello.ejb
-----
```

The bean deployment descriptor.

```
helloServer/HelloBean.java
-----
```

The EJB implementation.

```
hello/Hello.java
-----
```

The bean remote interface.

```
hello/HelloHome.java
-----
```

The bean's home interface.

Compiling and Running the Example

=====

Before running this example, the user 'scott' needs to have `javauserpriv`. This can be enabled by doing:

```
$ svrmgrl
SVRMGR> connect internal
SVRMGR> grant javauserpriv to scott;
```

```
SVRMGRL> quit
$
```

The configuration file INITSID.ORA must also specify that at least two MTS servers can be activated. That is, the parameters MTS_SERVERS and MTS_MAX_SERVERS must be set to at least the following:

```
mts_servers=2
mts_max_servers=2
```

UNIX

Enter the command 'make all' or simply 'make' in the shell to compile, load, and deploy the objects, and run the client program. Other targets are 'run' and 'clean'.

Make sure that a shell environment variable ORACLE_HOME is set to point to the home location of the Oracle installation. This is operating system dependent, so see the Installation documentation that came with your system for the location. Also, review the README file for the Oracle database, and the README file for the CORBA/EJB server (the Oracle8i ORB), for additional up-to-date information.

Windows NT

On Windows NT, run the batch file makeit.bat from a DOS command prompt to compile, load, and deploy the objects. Run the batch file runit.bat to run the client program, and see the results.

Make sure that the environment variables %ORACLE_HOME%, %CLASSPATH%, and %SERVICE% are set appropriately for the DOS command window. You can set these as either user or system environment variables from the Control Panel. Double click on System in the Control Panel then on the Environment tab to set these variables. Start a new DOS window after setting environment variable values.

See the Installation documentation that came with your Oracle8i system for the values of these variables. Also, review the README file for the Oracle database, and the README file for the CORBA/EJB server (the Oracle8i ORB), for additional up-to-date information.

You can also set an environment variable %JAVA_HOME% to point to the root of your Java JDK. For example, SET JAVA_HOME=C:\JDK1.1.6.

クライアント

```
import hello.Hello;
import hello.HelloHome;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;

public class Client
{
    public static void main (String[] args) throws Exception {
        if (args.length != 4) {
            System.out.println ("usage: Client serviceURL objectName user password");
            System.exit (1);
        }
        String serviceURL = args [0];
        String objectName = args [1];
        String user = args [2];
        String password = args [3];

        Hashtable env = new Hashtable ();
        env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
        env.put (Context.SECURITY_PRINCIPAL, user);
        env.put (Context.SECURITY_CREDENTIALS, password);
        env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
        Context ic = new InitialContext (env);

        // Activate a Hello in the 8i server
        // This creates a first session in the server
        HelloHome hello_home = (HelloHome)ic.lookup (serviceURL + objectName);
        Hello hello = hello_home.create ();
        hello.setMessage ("Hello World!");
        System.out.println (hello.helloWorld ());

        // Ask the first Hello to activate another Hello in the same server
        // This creates Another SESSION used by the first session
        hello.getOtherHello (user, password, serviceURL + objectName);
        System.out.println (hello.otherHelloWorld ());
    }
}
```

ホーム・インタフェース

```
package hello;

import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import java.rmi.RemoteException;

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

リモート・インタフェース

```
package hello;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;
import javax.ejb.CreateException;

public interface Hello extends EJBObject
{
    public String helloWorld () throws RemoteException;

    public void setMessage (String message) throws RemoteException;

    public void getOtherHello (String user, String password, String otherBeanURL)
        throws RemoteException, CreateException;

    public String otherHelloWorld () throws RemoteException;
}
```

Bean 実装

```
package helloServer;

import hello.*;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

import javax.ejb.CreateException;
import java.rmi.RemoteException;
import javax.naming.NamingException;
```

```
import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

public class HelloBean implements SessionBean
{
    String message;
    Hello otherHello;

    // Methods of the Hello interface
    public String helloWorld () throws RemoteException {
        return message;
    }

    public void setMessage (String message) throws RemoteException {
        this.message = message;
    }

    public void getOtherHello (String user, String password, String otherBeanURL)
        throws RemoteException, CreateException
    {
        try {
            // start a new session
            Hashtable env = new Hashtable ();
            env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
            env.put (Context.SECURITY_PRINCIPAL, user);
            env.put (Context.SECURITY_CREDENTIALS, password);
            env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
            Context ic = new InitialContext (env);

            // create the other Bean instance
            HelloHome other_HelloHome = (HelloHome)ic.lookup (otherBeanURL);
            otherHello = other_HelloHome.create ();
            otherHello.setMessage ("Hello from the Other HelloBean Object");
        } catch (NamingException e) {
            e.printStackTrace ();
        }
    }

    public String otherHelloWorld () throws RemoteException {
        if (otherHello != null)
            return otherHello.helloWorld ();
        else
            return "otherBean is not accessed yet";
    }
}
```

```
// Methods of the SessionBean
public void ejbCreate () throws RemoteException, CreateException {}
public void ejbRemove () {}
public void setSessionContext (SessionContext ctx) {}
public void ejbActivate () {}
public void ejbPassivate () {}
}
```

SSL の例

クライアント側の認証の例

README

Overview

=====

This is the exact same example as under examples/ejb/basic/helloworld, except that this example is using SSL client auth. So, except for the SSL details, please refer to the readme file under examples/ejb/basic/helloworld for other details.

The purpose of the example is to show how to use ssl client side authentication instead of username/password combination.

Setup required

You need to open the encrypted wallet (ewallet.der) provided in this directory using the wallet manager tool provided by Oracle, and save it as clear text wallet (cwallet.sso). The password is welcome12. Copy the generated cwallet.sso into TNS_ADMIN directory.

The encrypted wallet (ewallet.der) is only valid for Solaris platforms. For other platforms, you should generate the wallet using Oracle's own tool.

This test also requires B64 encoded wallet (cert.txt) which is already present in this directory. You can also generate your own using Oracle's owmgui tool and using export option in the tool.

The parameter SSL_CLIENT_AUTHENTICATION in \$TNSADMIN/sqlnet.ora should be set to TRUE.

Restart the database.

The setup also requires creation of a global user, say guest. The script to create global user is in this directory(create.sh). This script prompts for username and password of a privileged user as input to this script.

クライアント

```
import hello.Hello;
import hello.HelloHome;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;

public class Client
{
    public static void main (String[] args) throws Exception {
        if (args.length != 4) {
            System.out.println ("usage: Client serviceURL objectName credentials_file
password");
            System.exit (1);
        }
        String serviceURL = args [0];
        String objectName = args [1];
        String credsFile = args [2];
        String password = args [3];

        Hashtable env = new Hashtable ();
        env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
        env.put(Context.SECURITY_AUTHENTICATION, ServiceCtx.SSL_CLIENT_AUTH);
        env.put(Context.SECURITY_CREDENTIALS, password);
        // Simply specify a file that contains all the credential info. This is
        // the file generated by the wallet manager tool.
        env.put(Context.SECURITY_PRINCIPAL, credsFile);

        Context ic = new InitialContext (env);

        HelloHome hello_home = (HelloHome)ic.lookup (serviceURL + objectName);
        Hello hello = hello_home.create ();
        System.out.println (hello.helloWorld ());
    }
}
```

ホーム・インタフェース

```
package hello;

import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import java.rmi.RemoteException;

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

リモート・インタフェース

```
package hello;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

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

Bean 実装

```
package helloServer;

import javax.ejb.SessionBean;
import javax.ejb.CreateException;
import javax.ejb.SessionContext;
import java.rmi.RemoteException;

public class HelloBean implements SessionBean
{
    // Methods of the Hello interface
    public String helloWorld () throws RemoteException {
        String v = System.getProperty("oracle.server.version");
        return "Hello client, your javavm version is " + v + ".";
    }

    // Methods of the SessionBean
    public void ejbCreate () throws RemoteException, CreateException {}
    public void ejbRemove() {}
    public void setSessionContext (SessionContext ctx) {}
}
```

```
public void ejbActivate () {}  
public void ejbPassivate () {}  
}
```

サーバー側の認証の例

README

Overview

=====

This is the exact same example as under examples/ejb/basic/helloworld, except that this example is using SSL server side auth. So, except for the SSL details, please refer to the readme file under examples/ejb/basic/helloworld for other details.

The purpose of the example is to show how to use ssl server side authentication. Since the client doesn't have certificate in this case, it still passes username/password.

Setup required

You need to open the encrypted wallet(ewallet.der) provided in this directory using the wallet manager tool provided by Oracle, and save it as clear text wallet (cwallet.sso). The password is welcome12. Copy the generated cwallet.sso into TNS_ADMIN directory.

The encrypted wallet(ewallet.der) is only valid for Solaris platforms. For other platforms, you should generate the wallet using Oracle's own tool.

The parameter SSL_CLIENT_AUTHENTICATION in \$TNSADMIN/sqlnet.ora should be set to FALSE.

Restart the database.

クライアント

```
import hello.Hello;  
import hello.HelloHome;  
  
import oracle.aurora.jndi.sess_iiop.ServiceCtx;  
  
import javax.naming.Context;  
import javax.naming.InitialContext;
```

```
import java.util.Hashtable;

public class Client
{
    public static void main (String[] args) throws Exception {
        if (args.length != 4) {
            System.out.println ("usage: Client serviceURL objectName user password");
            System.exit (1);
        }
        String serviceURL = args [0];
        String objectName = args [1];
        String user = args [2];
        String password = args [3];

        Hashtable env = new Hashtable ();
        env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
        env.put (Context.SECURITY_PRINCIPAL, user);
        env.put (Context.SECURITY_CREDENTIALS, password);
        env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.SSL_CREDENTIAL);
        Context ic = new InitialContext (env);

        HelloHome hello_home = (HelloHome)ic.lookup (serviceURL + objectName);
        Hello hello = hello_home.create ();
        System.out.println (hello.helloWorld ());
    }
}
```

ホーム・インタフェース

```
package hello;

import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import java.rmi.RemoteException;

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

リモート・インタフェース

```
package hello;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

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

Bean 実装

```
package helloServer;

import javax.ejb.SessionBean;
import javax.ejb.CreateException;
import javax.ejb.SessionContext;
import java.rmi.RemoteException;

public class HelloBean implements SessionBean
{
    // Methods of the Hello interface
    public String helloWorld () throws RemoteException {
        String v = System.getProperty("oracle.server.version");
        return "Hello client, your javavm version is " + v + ".";
    }

    // Methods of the SessionBean
    public void ejbCreate () throws RemoteException, CreateException {}
    public void ejbRemove() {}
    public void setSessionContext (SessionContext ctx) {}
    public void ejbActivate () {}
    public void ejbPassivate () {}
}
```

略称と頭字語

この付録には、ネットワーク、分散オブジェクト開発および Java の分野で使用されるごく一般的な頭字語の一部が記載してあります。頭字語が指している製品または概念が、特定のグループ、企業または製品に関連している場合は、頭字語の元の綴りの後にそのグループ名、企業名または製品名を大カッコで囲んで示してあります。たとえば、次のようになります。CORBA ...[OMG]。

この頭字語のリストはあくまでも参考です。完全にリストしていること、またはリストが正確であるということは保証されません。

3GL	第三世代言語 (third generation language)
4GL	第四世代言語 (fourth generation language)
ACID	アトミック性、一貫性、孤立性、持続性 (atomicity, consistency, isolation, durability)
ACL	アクセス制御リスト (Access Control List)
ADT	抽象データ型 (abstract datatype)
AFC	Application Foundation Classes [Microsoft]
ANSI	米国規格協会 (American National Standards Institute)
API	アプリケーション・プログラミング・インタフェース (Application Programming Interface)
AQ	アドバンスド・キューイング (Advanced Queuing) [Oracle8]
ASCII	米国規格協会情報交換用標準コード (American Standard Code for Information Interchange)
ASP	Active Server Pages [Microsoft] アプリケーション・サービス・プロバイダ (application service provider)
AWT	抽象ウィンドウ操作ツールキット (Abstract Windowing Toolkit) [Java]

BDK	Bean Developer Kit [Java]
BLOB	バイナリ・ラージ・オブジェクト (binary large object)
BOA	基本オブジェクト・アダプタ (Basic Object Adapter) [CORBA]
BSD	Berkeley Software Distribution [UNIX]
C/S	クライアント / サーバー (client/server)
CGI	コモン・ゲートウェイ・インタフェース (Common Gateway Interface)
CICS	顧客情報管理システム (Customer Information Control System) [IBM]
CLI	コール・レベル・インタフェース (Call Level Interface) [SAG]
CLOB	キャラクタ・ラージ・オブジェクト (character large object)
COM	コンポーネント・オブジェクト・モデル (Component Object Model) [Microsoft]
COM+	拡張コンポーネント・オブジェクト・モデル (Component Object Model, extended) [Microsoft]
CORBA	共通オブジェクト・リクエスト・ブローカ・アーキテクチャ (Common Object Request Broker Architecture) [OMG]
DB	データベース (database)
DBA	データベース管理者、データベース管理 (database administrator, database administration)
DBMS	データベース管理システム (database management system)
DCE	分散コンピューティング環境 (Distributed Computing Environment) [OSF]
DCOM	分散コンポーネント・オブジェクト・モデル (Distributed Component Object Model) [Microsoft]
DDCF	Distributed Document Component Facility
DDE	ダイナミック・データ・エクスチェンジ (Dynamic Data Exchange) [Microsoft]
DDL	データ定義言語 (Data Definition Language) [SQL]
DLL	ダイナミック・リンク・ライブラリ (Dynamic Link Library) [Microsoft]
DLM	分散ロック・マネージャ (Distributed Lock Manager) [Oracle8]
DML	データ操作言語 (Data Manipulation Language) [SQL]
DOS	ディスク・オペレーティング・システム (Disk Operating System)

DSOM	分散システム・オブジェクト・モデル (Distributed System Object Model) [IBM]
DSS	意思決定支援システム (Decision Support System)
DTP	分散トランザクション処理 (Distributed Transaction Processing)
EBCDIC	拡張 2 進化 10 進コード (Extended Binary-Coded Decimal Interchange Code) [IBM]
EJB	Enterprise JavaBean
ERP	Enterprise Resource Planning
ESIOP	環境固有 ORB 間プロトコル (Environment-Specific Inter-ORB Protocol)
FTP	ファイル転送プロトコル (File Transfer Protocol)
GB	ギガバイト (gigabyte)
GIF	画像交換フォーマット (Graphics Interchange Format)
GIOP	汎用 ORB 間プロトコル (General Inter-Orb Protocol)
GUI	グラフィカル・ユーザー・インタフェース (Graphical User Interface)
GUID	Globally-Unique Identifier
HTML	ハイパーテキスト・マークアップ言語 (Hypertext Markup Language)
HTTP	ハイパーテキスト転送プロトコル (Hypertext Transfer Protocol)
IDE	統合開発環境 (Integrated Development Environment) 対話型開発環境 (Interactive Development Environment)
IDL	インタフェース定義言語 (Interface Definition Language)
IEEE	米国電気電子学会 (Institute of Electrical and Electronics Engineers)
IIOP	インターネット ORB 間プロトコル (Internet Inter-ORB Protocol)
IIS	Internet Information Server [Microsoft]
IP	インターネット・プロトコル (internet protocol)
IPC	プロセス間通信 (interprocess communication)
IS	情報サービス (information services)
ISAM	索引順次アクセス方式 (Indexed Sequential Access Method)
ISAPI	Internet Server API [Microsoft]
ISO	国際標準化機構 (International Standards Organization)

ISP	インターネット・サービス・プロバイダ (Internet Service Provider)
ISQL	対話型 SQL (Interactive SQL) [Interbase]
ISV	独立系ソフトウェア・ベンダー (Independent Software Vendor)
IT	情報技術 (Information Technology)
J2EE	Java 2 Enterprise Edition [Sun]
JAR	Java アーカイブ (tar の類推、tar を参照) (Java archive)
JCK	Java Compatibility Kit [Sun]
JDBC	Java database connectivity
JDK	Java Developer Kit
JFC	Java Foundation Classes
JIT	just in time
JLS	Java 言語仕様 (Java Language Specification)
JMF	Java Media Framework
JMS	Java Messaging Service
JNDI	Java Naming and Directory Interface
JNI	Java ネイティブ・インタフェース (Java Native Interface)
JOB	Java Objects for Business [Sun]
JPEG	Joint Photographic Experts Group
JRMP	Java Remote Message Protocol
JSP	Java Server Pages [Sun] Java ストアド・プロシージャ (Java Stored Procedure) [Oracle]
JTA	Java Transaction API
JTS	Java Transaction Service
JWS	Java Web Server [Sun]
KB	キロバイト (kilobyte)
LAN	ローカル・エリア・ネットワーク (Local Area Network)
LDAP	Lightweight Directory Access Protocol
LDIF	LDPA Data Interchange Format
LOB	ラージ・オブジェクト (large object)
MB	メガバイト (megabyte)
MIME	Multipurpose Internet Mail Extensions

MIS	Management Information Services
MOM	Message-Oriented Middleware
MPEG	Motion Picture Experts Group
MTS	マルチスレッド・サーバー (multi-threaded server) [Oracle]
MTS	Microsoft Transaction Server [Microsoft]
NCLOB	国内キャラクタ・ラージ・オブジェクト (national character large object)
NIC	ネットワーク情報センター (Network Information Center) [internet]
NNTP	Net News Transfer Protocol
NSAPI	Netscape Server Application Programming Interface
NSP	ネットワーク・サービス・プロバイダ (Network Service Provider)
NT	New Technology [Microsoft]
OCI	Oracle コール・インタフェース (Oracle Call Interface)
OCX	OLE Common Control [Microsoft]
ODBC	Open Database Connectivity [Microsoft]
ODBMS	オブジェクト・データベース管理システム (Object Database Management System)
ODL	Object Definition Language [Microsoft]
ODMG	オブジェクト・データベース・マネジメント・グループ (Object Database Management Group)
OEM	相手先商標製造会社 (Original Equipment Manufacturer)
OID	オブジェクト識別子 (Object Identifier)
OLE	オブジェクトのリンクと埋込み (Object Linking and Embedding)
OLTP	オンライン・トランザクション処理 (On Line Transaction Processing)
OMA	オブジェクト管理アーキテクチャ (Object Management Architecture) [OMG]
OMG	オブジェクト・マネージメント・グループ (Object Management Group)
OO	オブジェクト指向 (object-oriented, object orientation)
OODBMS	オブジェクト指向データベース管理システム (Object-oriented Database Management System)

OQL	オブジェクト照会言語 (Object Query Language)
ORB	オブジェクト・リクエスト・ブローカ (Object Request Broker)
ORDBMS	オブジェクト・リレーショナル・データベース管理システム (Object Relational Database Management System)
OS	オペレーティング・システム (Operating System)
OSF	オープン・ソフトウェア財団 (Open Software Foundation)
OSI	Open Systems Interconnect
OSQL	オブジェクト SQL (Object SQL)
OTM	Object Transaction Monitor
OTS	Object Transaction Service
OWS	Oracle Web Server
PB	ペタバイト (petabyte)
PDF	Portable Document Format [Adobe]
PGP	Pretty Good Privacy
PL/SQL	Procedural Language/SQL [Oracle]
POA	ポータブル・オブジェクト・アダプタ (Portable Object Adapter) [CORBA]
RAM	ランダム・アクセス・メモリー (Random Access Memory)
RAS	Remote Access Service [Microsoft]
RCS	リビジョン管理システム (Revision Control System)
RDBMS	リレーショナル・データベース管理システム (Relational Database Management System)
RFC	Request For Comments
RFP	Request For Proposal
RMI	Remote Method Invocation [Sun]
ROM	読出し専用メモリー (Read Only Memory)
RPC	リモート・プロシージャ・コール (Remote Procedure Call)
RTF	リッチ・テキスト・ファイル (Rich Text File)
SAF	Server Application Function [Netscape]
SAG	SQL アクセス・グループ (SQL Access Group)
SCSI	Small Computer System Interface
SDK	ソフトウェア開発者キット (Software Developer Kit)

SET	Secure Electronic Transaction
SGML	標準汎用マークアップ言語 (Standard Generalized Markup Language)
SID	システム識別子 (System Identifier) [Oracle]
SLAPD	Standalone LDAP Daemon
SMP	対称型マルチプロセッシング (Symmetric Multiprocessing)
SMTP	簡易メール転送プロトコル (Simple Mail Transfer Protocol)
SPI	Service Provider Interface
SQL	構造化照会言語 (Structured Query Language)
SQLJ	SQL for Java
SRAM	スタティック (または同期) ランダム・アクセス・メモリー (Static (or Synchronous) Random Access Memory)
SSL	セキュア・ソケット・レイヤー (Secure Socket Layer)
TB	テラバイト (terabyte)
TCPS	TCP for SSL
TCP/IP	伝送制御プロトコル / インターネット・プロトコル (Transmission Control Protocol/Internet Protocol)
TP	トランザクション処理 (Transaction Processing)
TPC	Transaction Processing Council
TPCW	TPC Web ベンチマーク (TPC Web benchmark)
TPF	トランザクション処理ファシリティ (Transaction Processing Facility)
TPM	トランザクション処理モニター (Transaction Processing Monitor)
UCS	汎用文字セット (Universal Character Set) [ISO 10646]
UDP	ユーザー・データグラム・プロトコル (User Datagram Protocol)
UI	ユーザー・インタフェース (user interface)
UML	Unified Modeling Language [Rational]
URI	Uniform Resource Identifier
URL	Universal Resource Locator
URN	Universal Resource Name
VAR	付加価値再販業者 (Value-Added Reseller)
VB	Visual Basic [Microsoft]

VRML	Virtual Reality Modeling Language
WAI	Web Application Interface [Netscape]
WAN	広域ネットワーク (Wide Area Network)
WIPS	Web Interactions Per Second [TPCW]
WWW	ワールド・ワイド・ウェブ (World Wide Web)
XA	eXtended Architecture [X/Open]
XML	eXtended Markup Language
jdb	Java デバッガ (Java debugger) [Sun]
tar	テープ・アーカイブ、テープ・アーカイバ (tape archive, tape archiver) [UNIX]
tps	トランザクション / 秒 (transactions per second)

記号

<assembly-descriptor> セクション, 2-22, A-3, A-14
<attr-mapping> 要素, A-29
<cmp-field> 要素, 4-30, 4-34, A-6, A-28
<column-name> 要素, A-29
<container-transaction> 要素, 2-23, A-20
<ejb-class> 要素, A-5
<ejb-client-jar> 要素, 2-25, A-32
<ejb-jar> 要素, 2-22, A-3
<ejb-link> 要素, A-10, A-11
<ejb-mapping> 要素, A-10, A-11, A-24
<ejb-name> 要素, 2-23, 2-24, A-5, A-10
<ejb-ref> 要素, A-10, A-11
<ejb-ref-name> 要素, 4-35, A-10, A-11
<ejb-ref-type> 要素, A-11
<enterprise-beans> セクション, A-3, A-4, 2-22, 2-23
<entity> 要素, A-5
<env-entry> 要素, A-9
<env-entry-name> 要素, A-9
<env-entry-type> 要素, A-9
<env-entry-value> 要素, A-9
<field-name> 要素, A-29
<home> 要素, A-5, A-11
<jndi-name> 要素, 2-26, A-10, A-11, A-12, A-24
<mapping> 要素, A-10, A-12
<mappings> 要素, A-23
<method> 要素, 2-24, A-20, A-27
 定義済み, A-17
<method-intf> 要素
 定義済み, A-18
<method-name> 要素, A-20, A-27
<method-permission> 要素, 2-23, A-15
<mode> 要素, 2-24, A-27
<oracle-descriptor> 要素, A-24

<persistence-deployer> 要素, A-28
<persistence-name> 要素, A-28
<persistence-provider> 要素, A-27
<persistence-type> 要素, A-6
<prim-key-class> 要素, 4-12, 4-30, A-6
<primkey-field> 要素, 4-30, A-6
<PSI-RI> 要素, A-29, A-30
<reentrant> 要素, A-14
<remote> 要素, A-5, A-11
<res-auth> 要素, 4-36, A-13
<resource-ref> 要素, 4-35
<resource-ref-mapping> 要素, A-12, A-24
<res-ref-name> 要素, 4-36, A-12, A-24
<res-type> 要素, 4-36, A-12
<role-link> 要素, A-14, A-15
<role-name> 要素, A-14, A-15, A-25, A-27
<run-as> 要素, 2-24, 4-23, A-2, A-26, A-27
<schema> 要素, A-29
<security-role> 要素, 2-23, A-15, A-25, A-26, A-27
<security-role-mapping> 要素, A-25
<security-role-ref> 要素, A-14, A-15
<serialize-mapping> 要素, A-30
<session> 要素, 2-23, A-5
<table> 要素, A-29
<transaction-manager> 要素, 7-29
 2pc エンジンの定義, A-21
<transaction-type> 要素, A-14, A-18
<trans-attribute> 要素, A-19

数字

2 タスク共通, 「TTC」を参照
2 フェーズ・コミット, 7-29

A

ACID プロパティ, 7-2
ADDRESS パラメータ, 3-11, 3-17
afterBegin メソッド, 7-39
afterCompletion メソッド, 7-40
APPLET_CLASS プロパティ, 5-28
aurora_client.jar ファイル, 6-10
AuroraCertificateManager クラス, 6-24, 6-25
 setCertificateChain メソッド, 6-24
 setEncryptedPrivateKey メソッド, 6-24
AuroraCurrentManager クラス, 6-20
aurora.zip, 5-27
authenticate メソッド, 5-19, 6-11

B

Bean
 インタフェース, 2-2
 エンティティ, 4-2
 環境, 2-10
 削除, 2-5
 作成, 2-3, 4-9
 参照の取得, 2-17
 セッション, 2-10, 4-2
 等価性の検査, 2-5
 ネーミング規則, 2-4
 配置, 2-20
 リモート・アクセス, 2-2
Bean 管理の永続性, 4-9, 4-18
beforeCompletion メソッド
 SessionSynchronization インタフェース, 7-39
begin メソッド, 7-16, 7-21
bindds コマンド, 7-22, 7-37
bindut コマンド, 7-17, 7-29

C

ClassLoader プロパティ, 5-29
CLIENT_IDENTITY プロパティ, 2-24, A-26
Collection, 4-11, 4-18
commit メソッド, 7-16, 7-17, 7-22
CosNaming サービス, 2-15, 5-2
CreateException, 2-7
create メソッド, 2-12, 2-18, 4-10, 4-11
 EJBHome インタフェース, 2-2, 2-3, 2-6

D

Database Configuration Assistant, 3-8
DataSource オブジェクト, A-13
 getConnection メソッド, 7-8, 7-25
 動的作成, 7-37
 ネームスペースでのバインド, 7-22
DebugAgent クラス, 2-29
 restart メソッド, 2-29
 stop メソッド, 2-29
deployejb ツール, 2-19, 2-20, 2-25
DESCRIPTION パラメータ, 3-11
DriverManager クラス
 getConnection メソッド, 7-8, 7-25
DTD ファイル, 2-21, 2-22, A-3, A-22

E

EJB
 Bean の作成, 2-3, 4-9
 アプリケーション開発者の役割, 1-3
 開発者の役割, 1-2
 基本概念, 1-2, 1-7
 コンテナ・ベンダーの役割, 1-3
 サーバー・ベンダーの役割, 1-2
 セキュリティ, 1-2
 セッションとエンティティの違い, 4-5
 ディプロイメント・ディスクリプタ, 1-2, A-1
 取り出すための URL, 2-17
 配置, 1-2, 2-19, 2-20
 パラメータの受渡し, 2-13
 プログラミングの制限事項, 2-27
 リモート・インタフェース, 2-4
ejbActivate メソッド, 2-8, 4-7, 4-21
EJBContext インタフェース, 2-10
ejbCreate メソッド, 2-3, 2-6, 4-6, 4-7, 4-10, 4-17, 4-28
 主キーの初期化, 4-13
EJBException, 2-7
ejbFindByPrimaryKey メソッド, 4-10, 4-13, 4-28
EJBHome インタフェース, 2-3, 2-4, 2-6, 4-9
 create メソッド, 4-10, 4-11
 findByPrimaryKey メソッド, 4-9, 4-10, 4-11
ejb-jar ファイル, 2-3, 4-10
ejbLoad メソッド, 4-6, 4-9, 4-19, 4-28
EJBMetaData インタフェース, 2-6
EJBObject インタフェース, 2-3, 2-4, 4-9, 4-11

ejbPassivate メソッド, 2-8, 4-7, 4-21
ejbPostCreate メソッド, 4-6, 4-10, 4-17, 4-28
ejbRemove メソッド, 2-8, 4-6, 4-8, 4-20, 4-28
ejbStore メソッド, 4-6, 4-9, 4-19, 4-28
endSession メソッド, 5-18
Enterprise JavaBeans, 「EJB」を参照
EntityBean インタフェース, 2-3, 2-8, 4-2, 4-5, 4-9, 4-28
 ejbActivate メソッド, 4-7, 4-21
 ejbCreate メソッド, 4-6, 4-7, 4-10, 4-28
 ejbFindByPrimaryKey メソッド, 4-10, 4-28
 ejbLoad メソッド, 4-6, 4-9, 4-19, 4-28
 ejbPassivate メソッド, 4-7, 4-21
 ejbPostCreate メソッド, 4-6
 ejbRemove メソッド, 4-6, 4-8, 4-20, 4-28
 ejbStore メソッド, 4-6, 4-9, 4-19, 4-28
 setEntityContext メソッド, 4-7, 4-8, 4-21, 4-28
 unsetEntityContext メソッド, 4-7
 実装, 4-16
Enumeration, 4-11

F

findByPrimaryKey メソッド, 4-9, 4-10

G

General Inter-ORB Protocol, 「GIOP」を参照
getEJBHome メソッド, 2-5, 2-10, 2-13
getEnvironment メソッド, 2-10
getHandle メソッド, 2-5
getPrimaryKey メソッド, 2-5
getRollbackOnly メソッド, 2-10
getUserTransaction メソッド, 2-10
GIOP
 oracle.aurora.server.SGiopServer, 3-9
 ディスパッチャ構成, 3-11
 プレゼンテーション, 3-2

I

iAS
 EJB の配置, 2-19
IIOP, 1-3, 1-4, 3-2, 5-14
 MTS_DISPATCHER, 3-3
 SSL サポート, 3-17
 クライアント

 セッション・ベース, 3-9
 ディスパッチャへの接続, 3-11
 プロファイル, 5-13
IIOP クライアント
 構成, 3-1, 3-19
InitialContext オブジェクト, 2-16, 5-11
Internet Inter-ORB Protocol, 「IIOP」を参照
isIdentical メソッド, 2-5

J

JAR ファイル, 2-3, 2-20, 4-10
Java Naming and Directory Interface, 「JNDI」を参照
javax-ssl-1_1.jar, 5-10, 6-4
javax-ssl-1_2.jar, 5-10, 6-4
JDeveloper
 デバッグ, 2-28
JNDI, 2-12
 EJB 参照の格納, 4-35
 EJB の検索, 4-25
 InitialContext コンストラクタ, 5-11
 JDBC DataSource の検索, 4-35
 lookup メソッド, 5-7, 5-12
 URL の構文, 4-25
 概要, 2-15
 コンテキスト・オブジェクト, 5-9
 参照の取得, 2-15
 初期コンテキスト, 5-2
jssl-1_1.jar, 5-10, 6-4
jssl-1_2.jar, 5-10, 6-4
JTA
 2 フェーズ・コミット, 7-8, 7-29
 Bean 管理, 7-14
 概要, 7-2
 クライアント側デマーケーション, 7-16
 コンテナ管理, 7-13
 仕様の Web サイト, 7-1
 制限事項, 7-13
 タイムアウト, 7-38
 ネストしたトランザクション, 7-13
 リソースの確保, 7-7, 7-25

L

LDAP, 2-15
LoginServer クラス, 6-11
 authenticate メソッド, 5-19, 6-11

Login クラス, 5-5, 6-11
LogoutServer クラス, 5-18, 6-11
logout メソッド, 5-18, 6-11
lookup メソッド, 2-17, 5-11, 5-12

M

Mandatory トランザクション属性, 7-6, A-19
MTS_DISPATCHERS パラメータ
 ADDRESS 属性, 3-17
 PRESENTATION 属性, 3-9, 3-10, 3-17
 PROTOCOL 属性, 3-10
 概要, 3-3

N

Net8 Assistant
 IIOP クライアント用の構成, 3-6, 3-10
Never トランザクション属性, 7-7, A-19
NON_SSL_LOGIN の値, 2-16, 5-2, 5-10
NotSupported トランザクション属性, 7-6, A-19

O

oracle.aurora.server.SGiopServer, 3-9
OracleDriver クラス
 defaultConnection メソッド, 7-7, 7-25
OracleJTADDataSource クラス, 7-38
ORB
 初期化, 6-24
ORBClass プロパティ, 5-30
ORBdisableLocator プロパティ, 5-30
ORBsingletonClass プロパティ, 5-30
OSS.SOURCE.MY_WALLET パラメータ, 3-19

P

Persistence Service Interface Reference
 Implementation, 「PSI-RI」を参照
PRESENTATION 属性, 3-9, 3-10, 3-11, 3-17
PROTOCOL_STACK パラメータ, 3-11
PROTOCOL 属性, 3-10
PSI-RI, 4-27, A-27

R

RAW セッション層, 3-11
regep ツール, 3-15, 3-16
Remote Method Invocation
 「RMI」を参照
RemoteException, 2-7
remove メソッド, 2-12
 EJBHome インタフェース, 2-3, 2-5
Required トランザクション属性, 7-6, A-19
RequiresNew トランザクション属性, 7-6, A-19
restart メソッド, 2-29
RMI, 2-4
rollback メソッド, 7-16, 7-17, 7-22

S

SECURITY_AUTHENTICATION プロパティ, 2-16, 5-10
SECURITY_CREDENTIALS プロパティ, 2-16, 5-9
SECURITY_PRINCIPAL プロパティ, 2-16, 5-9
SECURITY_ROLE プロパティ, 2-16, 5-10
Serializable インタフェース, 2-14
SessionBean インタフェース, 2-8
 EJB, 2-3, 2-8
 ejbActivate メソッド, 2-8, 4-7
 ejbPassivate メソッド, 2-8, 4-7
 ejbRemove メソッド, 2-8, 4-6
 setSessionContext メソッド, 2-9, 4-7
SessionContext
 インタフェース, 2-9
SessionSynchronization インタフェース, 7-39
 afterBegin メソッド, 7-39
 afterCompletion メソッド, 7-40
 beforeCompletion メソッド, 7-39
SESSION 属性, 3-11
setCertificateChain メソッド, 6-24
setEncryptedPrivateKey メソッド, 6-24
setEntityContext メソッド, 4-7, 4-8, 4-21, 4-28
setRollbackOnly メソッド, 2-10
setSessionContext メソッド, 2-9, 4-7, 4-8
setTransactionTimeout メソッド, 7-39
SID, 2-17, 5-5
SPECIFIED_IDENTITY プロパティ, 2-24, A-26
SSL, 6-20
 JAR ファイル, 5-10, 6-4
 構成, 3-17

接続のセキュリティ, 1-3
定義済み, 6-3
プロトコルのバージョン・ナンバー, 6-4
SSL_CLIENT_AUTHENTICATION パラメータ, 3-19
SSL_CLIENT_AUTH の値, 2-16, 5-10
SSL_CREDENTIAL の値, 2-16, 5-10
SSL_LOGIN の値, 2-16, 5-10
SSL_VERSION パラメータ, 3-19
SSL_VERSION プロパティ, 2-16, 3-19
start メソッド, 2-29
stop メソッド, 2-29
Supports トランザクション属性, 7-6, A-19
SYSTEM_IDENTITY プロパティ, 2-24, A-26

T

TransactionManager クラス, 7-3
Transaction クラス, 7-3
TTC, 5-12

U

unsetEntityContext メソッド, 4-7, 4-28
URL
 JNDI パラメータとして使用, 2-17
 構文, 5-5
URL_PKG_PREFIXES プロパティ, 2-16, 5-9
UserTransaction オブジェクト
 begin メソッド, 7-16, 7-21
 commit メソッド, 7-16, 7-17, 7-22
 rollback メソッド, 7-16, 7-17, 7-22
 setTransactionTimeout メソッド, 7-39
 取得, 7-14
useServiceName フラグ, 5-6
 deployejb のオプション, 5-11
USE_SERVICE_NAME プロパティ, 5-11

W

Wallet, 6-20

X

XML, 2-21
 ディプロイメント・ディスクリプタ, 4-10, A-1
 バージョン・ナンバー, 2-22, A-3, A-22

あ

値渡し, 2-13
アプレット
 サーバー・オブジェクトの起動, 5-28
 サンドボックス・セキュリティの制限事項, 5-28

う

受渡し, 3-14

え

永続性
 Bean 管理, 4-9
 PSI-RI, 4-27
 概要, 4-2
 管理, 4-10, 4-27
 コンテナ管理, 4-27, 4-33
 コンテナ管理と Bean 管理, 4-26
 ディプロイメント・ディスクリプタ, 4-33, A-2, A-6
 データ管理, 4-7
 データの初期化, 4-17
 データベース表の作成, 4-22
永続性プロバイダ, 4-33
エンティティ Bean
 Bean 管理の永続性, 4-18
 永続データ, 4-2, 4-9
 概要, 1-9, 4-2, 4-5
 活性化と受動化, 4-21
 クラス実装, 4-15, 4-16
 コンテキスト情報, 4-8, 4-21
 削除, 4-8
 作成, 4-7, 4-9, 4-10, 4-25
 参照の取得, 4-26
 主キー, 4-10
 配置, 4-23
 破棄, 4-20
 ファインダ・メソッド, 4-10, 4-13, 4-29
 ホーム・インタフェース, 4-10, 4-25
 リモート・インタフェース, 4-11, 4-15
エンドポイント, 3-13
 IIOP の登録, 3-15

お

オブジェクトの活性化, 5-27
セッション中, 5-23, 5-28

か

活性化, 2-8, 4-21
環境
 DataSource のロケーティング, 4-35
 EJB 参照の定義, 4-35
 取得, 2-10

く

クライアント
 既存の Bean へのアクセス, 5-22
クライアント側の認証, 6-5

こ

公開されているオブジェクト
 パーミッション, 5-4
構成, 3-1 ~ 3-19
 IIOP クライアント, 3-1, 3-19
 TCP/IP を介した SSL, 3-17
 ディスパッチャへの直接接続, 3-16
コールアウト
 SSL の使用, 6-21
コールバック
 SSL の使用, 6-22
 クライアント側の認証, 6-25
 サーバー側の認証, 6-22
コンテキスト
 JNDI オブジェクト, 5-9
 セッション, 2-10
 トランザクション, 2-10
コンテキスト・オブジェクト
 JNDI オブジェクト, 2-16
 JNDI コンテキスト, 2-16
コンテナ管理の永続性, 4-27
 主キーの管理, 4-32
 データ・フィールドの定義, 4-33
 配置, A-6, A-27

さ

サーバー側の認証, 6-5
サービス名, 5-6, 5-11
参照渡し, 2-13

し

システム識別子, 「SID」を参照
主キー, 4-9, 4-10
 エンティティ Bean, 4-30
 エンティティ Bean の識別, 4-10
 概要, 4-2, 4-12
 管理, 4-7
 作成, 4-13
 初期化, 4-32
 制限事項, A-6
 ファインダ・メソッド, 4-17
 複合, 4-31
受動化, 2-8, 4-21
証明書, 6-20, 6-21, 6-24
 マネージャ, 6-24

せ

制限事項, 2-27
セキュア・ソケット・レイヤー, 「SSL」を参照
セッション
 サーバー側からの終了, 5-18
 同期, 7-39
 ルーティング, 5-13
 ログアウト, 5-18, 6-11
セッション Bean
 IIOP, 3-9
 概要, 1-9, 4-2
 クラス実装, 2-8
 コンテキスト, 2-9
 削除, 2-8
 作成, 2-8, 2-18
 配置, 2-19, 2-20
 ホーム・インタフェース, 2-7
 例, 2-10, 4-15
セッション中の活性化, 5-23

て

デイスパッチャ

概要, 3-12

構成, 3-11

直接接続, 3-11

ディプロイメント・ディスクリプタ, 1-6, 2-3, 2-19, 4-10, A-1

Bean のアイデンティティ, 2-24

Bean のタイプ, A-5

Bean 名, A-5, A-23

EJB 参照, A-9

JDBC の DataSource, A-12, A-24

Oracle 固有の要素, A-22

run-as アイデンティティ, A-26

XML, 2-20

永続性, A-6

エンティティ Bean, 4-23

環境変数, A-8

再入可能性, A-14

セキュリティ, 2-23, A-14, A-15, A-25

トランザクション, 2-23, A-14, A-18

論理名のマッピング, A-23

データ整合性, 6-3

デバッグ手法, 2-28

と

頭字語, C-1

トランザクション

2 フェーズ・コミット, 7-8, 7-29

Bean 管理, 7-4, 7-14

概要, 1-2, 7-2

クライアント側デマーケーション, 7-16

グローバル, 7-3

コミット, 2-10

コンテキストの伝播, 2-10, 7-5

コンテナ管理, 7-4, 7-6, 7-13

ステータスの取得, 2-10

制限事項, 7-13

タイムアウト, 7-38

ディプロイメント・ディスクリプタ, A-19

デマーケーション, 7-3

リソースの確保, 7-7, 7-25

ロールバック, 2-10

トレース・ファイル, 2-28

に

認証

SSL の使用, 6-3

サーバー側, 6-20

定義済み, 6-5

ログアウト, 5-18, 6-11

ね

ネームスペース, 5-3

は

パラメータ

受渡し規則, 2-13

ハンドル

取得, 2-5

ふ

ファインダ・メソッド, 4-13, 4-29

ejbFindByPrimaryKey メソッド, 4-17

findByPrimaryKey メソッド, 4-10, 4-11

WHERE 句のファインダ・メソッド, 4-29

エンティティ Bean, 4-10

プレゼンテーション

GIOP, 3-2, 3-9

oracle.aurora.server.SGiopServer, 3-9

ほ

ホーム・インタフェース

getEJBHome メソッド, 2-13

lookup, 2-12

概要, 1-6

作成, 2-3, 4-9

取得, 2-5

要件, 2-4

例, 2-7

め

メタデータ, 2-6

り

- リスナー, 3-12
 - 受渡し, 3-14
 - エンドポイント, 3-13
 - エンドポイントの動的登録, 3-15
 - 概要, 3-12
 - リダイレクション, 3-12, 3-13
- リダイレクション, 3-12, 3-13, 3-16
- リモート・インタフェース, 2-12, 4-15
 - 概要, 1-6, 2-2
 - 作成, 2-3, 2-4, 4-9
 - 要件, 2-4
 - 例, 2-6
- リモート・オブジェクト
 - アクセス, 1-2
 - 定義, 4-3

れ

- 例外
 - 作成, 2-7

ろ

- ログイン
 - 非 JNDI ログイン, 5-18, 6-11