

Oracle8i

CORBA 開発者ガイド

リリース 8.1

2000 年 11 月

部品番号 : J02312-01

ORACLE®

Oracle8i CORBA 開発者ガイド, リリース 8.1

部品番号: J02312-01

原本名: CORBA Developer's Guide, Release 3 (8.1.7)

原本部品番号: A83722-01

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

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載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	
前提となる関連ドキュメント	1-2
用語	1-2
CORBA について	1-3
CORBA の機能	1-5
ORB について	1-6
JNDI と IIOP の使用	1-7
IIOP	1-7
詳細情報	1-8
書籍	1-8
URL	1-8
2 スタート・ガイド	
最初の CORBA アプリケーション	2-2
IDL でのインタフェースのコーディング	2-3
スタブとスケルトンの生成	2-4
サーバー・オブジェクトの実装部のコーディング	2-6
クライアント・コードの作成	2-8
Java ソースのコンパイル	2-10
データベースへのクラスのロード	2-11
オブジェクト名の公開	2-12
例の実行	2-13

インタフェース定義言語 (IDL)	2-14
IDL の使用	2-15
IDL のタイプ	2-18
例外	2-21
IDL を使用しない開発	2-23
ORB とサーバー・オブジェクトの活性化	2-23
クライアント側	2-23
サーバー側	2-23
オブジェクトの活性化について	2-24
CORBA インタセプタ	2-24
デバッグ手法	2-24
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	2-25

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	3-11
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 JNDI 接続とセッション IIOP サービス

JNDI 接続の基本事項	4-3
ネームスペース	4-4
データベース・オブジェクトに対する実行権	4-5
URL 構文	4-6
URL のコンポーネントとクラス	4-7
JNDI を使用したバインド済みオブジェクトへのアクセス	4-8
JNDI サポート・クラスのインポート	4-10
JNDI InitialContext の取得	4-10

セッション IIOP サービス	4-14
セッション IIOP サービスの概要	4-14
セッション管理	4-16
サービス・コンテキスト・クラス (ServiceCtx)	4-17
セッション・コンテキスト・クラス (SessionCtx)	4-18
セッション管理の使用例	4-19
セッション・タイムアウトの設定	4-27
Oracle8i JVM のバージョン・ナンバーの検索	4-28
非 IIOP プレゼンテーションからのセッション中の CORBA オブジェクトの活性化	4-29
JNDI を使用しない CORBA オブジェクトへのアクセス	4-29
NameService の初期参照の取得	4-29
ORBDefaultInitRef からの初期参照の取得	4-33

5 高度な CORBA プログラミング

SQLJ の使用	5-2
SQLJ トランスレータの実行方法	5-3
SQLJ の完全な例	5-3
CORBA コールバックの実装	5-3
IDL	5-4
クライアント・コード	5-4
コールバック・サーバーの実装	5-5
コールバック・クライアント / サーバーの実装	5-6
IFR を使用したインタフェースの取得	5-6
IDL インタフェースの公開	5-7
インタフェースの暗黙的取得	5-8
インタフェースの明示的取得	5-8
CORBA の Tie メカニズムの使用	5-11
JDK 1.1 から Java 2 への移行	5-11
アプレットからの CORBA オブジェクトの起動	5-16
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	5-16
アプレット内のオブジェクト検索の実行	5-16
CORBA オブジェクトにアクセスするアプレット用の HTML の修正	5-18
非 Oracle ORB との相互運用性	5-21
Oracle ORB を使用する Java クライアント	5-22
非 Oracle ORB を使用する Java クライアント	5-22

C++ クライアントの相互運用性	5-22
IIOP トランスポート・プロトコル	5-24

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	7-3
トランザクションのデマーケート	7-3
トランザクション・コンテキストの伝播	7-4
リソースの確保	7-5
2 フェーズ・コミット	7-5
JTA の制限事項	7-6
JTA のサーバー側デマーケーション	7-7
JTA のクライアント側デマーケーション	7-9
2 フェーズ・コミット・エンジンの構成	7-15
DataSource オブジェクトの動的作成	7-17
トランザクションのタイムアウト設定	7-18

Java Transaction Service	7-18
JTS のクライアント側デマーケーション	7-19
JTS のサーバー側デマーケーション	7-21
JTS の制限事項	7-23
トランザクション・サービス・インタフェース	7-24
TransactionService	7-24
Java Transaction Service の使用	7-25
JTS の詳細情報	7-29
JDBC の制限事項	7-29

A コード例 : CORBA

基本例	A-2
README	A-2
Bank.IDL	A-2
サーバー	A-3
Client.java	A-5
IFR の例	A-6
Bank.IDL	A-6
サーバー	A-6
クライアント	A-9
コールバックの例	A-16
README	A-16
IDL ファイル	A-19
サーバー	A-20
クライアント	A-20
Tie の例	A-22
README	A-22
Hello.IDL	A-24
サーバー・コード - HelloImpl.java	A-24
Client.java	A-24
純粋な CORBA クライアント	A-25
README	A-25
Bank.IDL	A-29
サーバー・コード	A-29
Client.java	A-31

JTA の例	A-33
単一フェーズ・コミットの JTA トランザクションの例	A-33
Employee.IDL	A-33
Client.java	A-34
EmployeeServer.sqlj	A-37
2 フェーズ・コミットの JTA トランザクションの例	A-39
Employee.IDL	A-39
Client.java	A-39
サーバー	A-41
JTS トランザクションの例	A-46
README	A-46
Employee.IDL	A-50
Client.java	A-50
サーバー	A-51
SSL の例	A-53
クライアント側の認証	A-53
README	A-53
Hello.IDL	A-57
Client.java	A-57
サーバー	A-58
サーバー側の認証	A-58
README	A-58
Hello.IDL	A-62
Client.java	A-62
サーバー	A-64
セッションの例	A-65
README	A-65
Hello.IDL	A-68
Client.java	A-68
サーバー	A-69
アプレットの例	A-70
JDK および JInitiator のアプレット	A-70
README	A-70
JDK 1.1 の HTML	A-70
JDK 1.2 の HTML	A-71
Oracle JInitiator の HTML	A-72
アプレット・クライアント	A-72

Visigenic アプレット	A-73
README	A-73
Visigenic クライアント・アプレットの HTML	A-74
Visigenic クライアント・アプレット	A-74

B Oracle8i JVM ORB と VisiBroker VBJ ORB の比較

オブジェクト参照はセッションの存続期間のみ有効	B-2
データベース・サーバーが実装のメインライン	B-2
サーバー・オブジェクトの実装はロードと公開により配置	B-2
継承による実装はほとんど同一	B-3
委譲による実装は異なる	B-3
クライアントは JNDI でオブジェクト名を検索	B-4
実装リポジトリは存在しない	B-4
Aurora と VBJ の銀行の例	B-5
銀行 IDL モジュール	B-5
Aurora クライアント	B-5
VBJ クライアント	B-6
Aurora アカウントの実装	B-7
VBJ アカウントの実装	B-7
Aurora アカウント・マネージャの実装	B-8
VBJ アカウント・マネージャの実装	B-9
VBJ サーバーのメインライン	B-9

C 略称と頭字語

索引

はじめに

このマニュアルは、Oracle8i 用の CORBA アプリケーション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 CORBA アプリケーションの開発に役立ちます。プログラマ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。このマニュアルは、CORBA に関してある程度の経験があることを前提としています。CORBA の概念の詳細は、xi ページの「[参考資料](#)」を参照してください。

構成

このマニュアルは、次の章と付録から構成されています。

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

[第 2 章「スタート・ガイド」](#) では、Oracle8i データベース・サーバーで動作する CORBA サーバー・オブジェクトを開発する方法を説明します。

[第 3 章「IIOP アプリケーションの構成」](#) では、CORBA アプリケーションの構成方法を説明します。

[第 4 章「JNDI 接続とセッション IIOP サービス」](#) では、CORBA アプリケーションでの JNDI およびセッションの使用方法を説明します。

[第 5 章「高度な CORBA プログラミング」](#) では、CORBA アプリケーションのプログラミングの方法を、第 2 章に記載されている単純な例よりもさらに詳細に説明します。

[第 6 章「IIOP のセキュリティ」](#) では、CORBA アプリケーションにセキュリティを実装する方法を説明します。

[第 7 章「トランザクション操作」](#) では、CORBA アプリケーションを開発する際に使用できるトランザクション・インタフェースを説明します。

[付録 A「コード例:CORBA」](#) には、CORBA アプリケーションの例が記載されています。各例には、Java および IDL のソース・コードが記載されています。

[付録 B「Oracle8i JVM ORB と VisiBroker VBJ ORB の比較」](#) では、VisiBroker での CORBA アプリケーション開発と Oracle8i JVM での CORBA アプリケーション開発の基本的な違いをいくつか説明します。

[付録 C「略称と頭字語」](#) には、頭字語のリストが記載されています。

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省略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)、および組み込み 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。

参考資料

『Programming with VisiBroker』(D. Pedrick 他著、John Wiley and Sons、1998 年) では、VisiBroker の観点から CORBA 開発をわかりやすく紹介しています。

『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。

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>

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>

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』

1

概要

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。CORBA による開発における Oracle8i JVM に固有の特性に焦点を当て、標準的な CORBA 開発モデルの概要を述べます。

この章では、次のトピックを説明します。

- [前提となる関連ドキュメント](#)
- [用語](#)
- [CORBA について](#)
- [JNDI と IIOP の使用](#)
- [詳細情報](#)

前提となる関連ドキュメント

このマニュアルを読む前に、『Oracle8i Java 開発者ガイド』を読む必要があります。データベース・サーバーにおける Java を理解するのに必要な技術的背景を説明しています。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 Java 仮想マシンの基本事項と Oracle8i JVM に付属するツールの技術面の概要も述べます。

さらに、『Oracle8i Java 開発者ガイド』では、CORBA で実装される分散コンポーネント開発モデルの利点も説明します。

用語

この章で使用する基本用語のいくつかを定義します。Java および分散オブジェクト・コンピューティングで使用する一般的な頭字語のリストは、[付録 C「略称と頭字語」](#)も参照してください。

クライアント

クライアントは、サーバー・オブジェクトに対してリクエストを発行するオブジェクト、アプリケーションまたはアプレットです。クライアントは、ワークステーションまたはネットワーク・コンピュータ上で実行される Java アプリケーションや、Web ブラウザでダウンロードされたアプレットである必要はありません。サーバー・オブジェクトは、別のサーバー・オブジェクトのクライアントであってかまいません。クライアントとはリクエスト発行者とサーバーの関係において、リクエスト発行者の立場にあることを示しており、物理的な場所やコンピュータ・システムのタイプを指しているわけではありません。

マーシャリング

分散オブジェクト・コンピューティングでは、マーシャリングとは、ORB がクライアント・オブジェクトとサーバー・オブジェクトとの間でリクエストとデータをやり取りする過程のことです。

オブジェクト・アダプタ

それぞれの CORBA ORB では、ORB とメッセージ受渡しオブジェクトとの間のインタフェースであるオブジェクト・アダプタ (OA) が実装されています。CORBA 2.0 では、基本オブジェクト・アダプタ (BOA) が存在する必要があると規定していますが、そのインタフェースの詳細のほとんどは個々の CORBA ベンダーに任されています。将来の CORBA 規格では、ベンダー中立のポータブル・オブジェクト・アダプタ (POA) が必要になります。Oracle では、将来のリリリースで POA をサポートする予定です。

リクエスト

リクエストとは、メソッドの起動のことです。メソッド・コール、メッセージなどとも呼ばれます。

サーバー・オブジェクト

CORBA サーバー・オブジェクトは、通常、クライアントからの最初のリクエストに基づきサーバーにより活性化される Java オブジェクトです。

セッション

セッションとは、常に、データベース・セッションのことを意味します。概念上は、SQL*Plus などのツールが Oracle に接続するときに確立されるセッションと同じものですが、CORBA の場合は次のような相違点があります。

- CORBA クライアントからのデータベース・セッションは IIOP プロトコルを使用して確立されますが、SQL*Plus セッションは Net8 TTC プロトコルを使用して確立されます。
- IIOP セッションは、データベース・サーバー内で実行される Java 仮想マシン (JVM) により制御されます。

注意： CORBA を Oracle8i とともに使用するには、リスナーが TTC リクエストに加えて IIOP リクエストを認識できるようにデータベースを設定する必要があります。DBA およびシステム管理者は、データベースと、IIOP リクエストを受け入れるリスナーの設定に関して第 3 章「IIOP アプリケーションの構成」を参照してください。

セッションの詳細は、4-14 ページの「セッション IIOP サービス」を参照してください。

CORBA について

CORBA とは、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) の頭字語です。CORBA で共通 (Common) と呼ばれているのは、複数の提案者からのアイデアを統合した仕様である点です。CORBA は、1 つの大企業の主導でできたものではなく、慎重にベンダー中立性を実現したものです。CORBA アーキテクチャでは、ブローカというソフトウェア・コンポーネントが定義されています。ブローカは、オブジェクトにリクエストを仲介し、送信するものです。リクエストの送信先のオブジェクトは 1 つのネットワークまたは複数のネットワークに分散しており、リクエストとは異なる言語で書かれていたり、リクエストとはまったく異なるハードウェア・アーキテクチャ上で稼動している場合があります (また実際、それがふつうです)。

注意： この項では CORBA を簡単に紹介し、Oracle8i Server 環境での CORBA の使用方法を説明します。CORBA のすべてを紹介するのは、このマニュアルの範囲を超えています。さらに詳しい内容の文献は、1-8 ページの「詳細情報」のリファレンスを参照することをお勧めします。この項では、CORBA の概要を簡単に説明します。

CORBA により、アプリケーションは様々な出所のコンポーネントを結び付けられます。また、通常のクライアント / サーバー・アプリケーションとは異なり、CORBA アプリケーションは必ずしも同期が必要なわけではありません。CORBA リクエスタ (クライアント) がサーバー・コンポーネント上のメソッドを起動し、結果を待つのは必ずしも一般的ではありません。非同期メソッド起動、イベント・インタフェース、およびサーバー・オブジェクトからクライアント ORB へのコールバックを使用することで、アプリケーションの構成を、互いに連係している多くのオブジェクトとリンクし、1 つまたは多数のデータ・ソースおよびその他のリソースにトランザクション制御でアクセスする精密なものにできます。

CORBA は、クロス・プラットフォーム、クロス・ランゲージ環境での開発のための国際標準を提供します。CORBA は、アプリケーション・オブジェクトが公開するインタフェースの仕様を作成するための中立言語である **Interface Definition Language** (インタフェース定義言語: IDL) を規定するという形でクロス・ランゲージ開発をサポートします。

CORBA では、まったく異なるハードウェア上で稼働する異なるオペレーティング・システムの相互運用のためのトランスポート・メカニズムである IIOP を規定することでクロス・プラットフォーム開発をサポートします。IIOP は、各システム上で動作する ORB とともに使用し、アプリケーション開発者に対しデータおよび要求の転送の透過性を実現する共通のソフトウェア・バスを提供するものです。

CORBA 規格は Java が現れる直前に開発されて普及した、世代や洗練度の異なるシステム言語を取り込んで行う異種アプリケーション開発環境におけるコンポーネント開発を重視した規格ですが、Java のみで CORBA アプリケーションを開発することは可能です。CORBA と Java は非常に相性がよいといえます。

CORBA 開発者向けに、Oracle8i JVM では次のサービスとツールを提供します。

- **OMG Object Transaction Service (OTS) への Java Transaction Service (JTS) インタフェース。**
- **CosNaming サービスの実装。Oracle8i データベースへのオブジェクトの公開、オブジェクト参照の取得、およびオブジェクトの活性化で使用します。**
- **Oracle8i JVM セッションベースの ORB をサポートするバージョンの IIOP プロトコル。標準 IIOP と互換性があります。**
- **CORBA アプリケーションの開発を支援する様々なツール。これらのツールには次の機能があります。**
 - データベースへの Java クラスおよびリソース・ファイルのロード
 - ロードされたクラスの削除
 - CosNaming サービスへのオブジェクトの公開
 - セッション・ネームスペースの管理

CORBA の機能

CORBA は、次のような方法で柔軟性を実現します。

- CORBA ではインタフェース定義言語（Interface Definition Language: IDL）の仕様を定めており、これによりオブジェクトへのインタフェースを定義できます。IDL オブジェクト・インタフェースでは、特に次のものを記述します。
 - オブジェクト中の public なデータ。
 - オブジェクトが応答できるオペレーションの完全なシグネチャ。CORBA オペレーションは Java メソッドにマップされ、IDL データ型で記述されたオペレーション・パラメータは Java データ型にマップされます。
 - オブジェクトで発生する可能性のある例外。IDL 例外もまた Java 例外にマップされ、このマッピングは非常に直感的です。

CORBA では、COBOL、C などの非オブジェクト指向言語や Smalltalk、Java などのオブジェクト指向言語をはじめとする、様々なプログラミング言語に対するバインドが提供されます。

- すべての CORBA 製品の実装では、アプリケーション開発者にとって非常に透過的な方法でオブジェクト・リクエストのルーティングを処理するオブジェクト・リクエスト・ブローカ（Object Request Broker: ORB）が提供されます。たとえば、クライアント・コードに記述されるリモート・オブジェクトへのリクエスト（メソッド起動）はローカルのメソッド起動にきわめて類似しています。パラメータおよびリターン・データのマーシャリングなどのリモート・コール機能は、プログラマにかかわって ORB が実行します。
- CORBA では、ネットワーク・プロトコルとして、Internet Inter-ORB Protocol (IIOP) を規定しています。これは、広く利用されているインターネット標準のトランスポート・プロトコルである TCP/IP を介して ORB リクエストおよびデータを送信するためのプロトコルです。
- 開発者が毎回最初から作成し直さなくて済むようにすることで、アプリケーション開発の負担を軽減するために、よく使用される一連のサービスの仕様が規定されています。そのようなサービスのうちのいくつかを次に挙げます。
 - ネーミング。CORBA サーバー・オブジェクトに結び付けられている名前をもとに、オブジェクト参照を取得するための 1 つ以上のサービス。
 - トランザクション。柔軟かつ移植性のある方法でデータ・リソースのトランザクション制御を管理するためのサービス。
 - イベント。

CORBA では、12 種類以上のサービスが定義されます。その大半は、CORBA ORB ベンダーではまだ実装されていません。

この項の後半で、Oracle8i JVM CORBA アプリケーションの重要なコンポーネントのいくつかを紹介し、それには次のようなものがあります。

- ORB — リモート・オブジェクトとの通信方法
- IDL — 移植性のあるインタフェースの書き方
- ネーミング・サービス（および JNDI） — 永続オブジェクトの見つけ方
- オブジェクト・アダプタ — 一時オブジェクトの登録方法

注意： この章で使用する Java コードの例はオンラインで入手できます。完成例（付録 A 「コード例：CORBA」を参照してください）を使用して理解を深め、それをコンパイルして実行し、その後で、自分の用途にあわせて修正を加えることができます。例はすべて

ORACLE_HOME/javavm/demo/demo.zip ファイルに格納されています。

ORB について

Object Request Broker (ORB) は、CORBA の製品の基礎となる部分です。ORB を利用すると、クライアントからサーバーにメッセージを送信し、サーバーからクライアントに値を戻すことができます。ORB は、クライアントとサーバー・オブジェクトとの間のすべての通信を取り扱います。

Oracle8i JVM ORB は、Inprise の VisiBroker 3.4 for Java のコードに準拠しています。サーバー側で実行される ORB は、様々な Oracle8i オブジェクトのロケーションや起動モデルに対応するために、VisiBroker コードから少し変更されています。クライアント側 ORB の変更は、ごくわずかです。

注意： Oracle8i JVM に付属する VisiBroker ORB の機能については、Oracle8i サーバーへのアクセスの用途に限ってライセンスが提供されています。

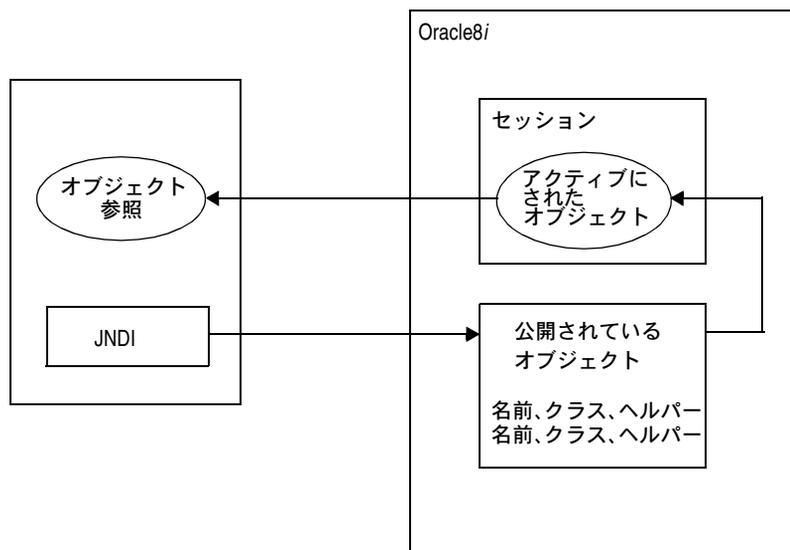
CORBA 製品の実装形式によっては、アプリケーション・プログラマとサーバー・オブジェクト開発者がクライアントとサーバー上で ORB を起動する方法の詳細を認識し、ORB を起動してオブジェクトを活性化するコードをオブジェクトに含める必要があります。これに対して、Oracle8i ORB では、こうした詳細がアプリケーション開発者にとってほぼ透過的になっています。この章と付録 A の Java コード例を見るとわかるように、開発者が ORB を直接制御する必要があるケースは限られています。そのケースとは、たとえば、コールバック・メカニズムをコーディングする場合や、一時オブジェクトを基本オブジェクト・アダプタに登録する必要がある場合などです。

JNDI と IIOP の使用

Oracle データベース内の CORBA オブジェクトを公開するには、OMG CosNaming サービスを使用します。また、これらのオブジェクトには、CosNaming に対する Oracle の JNDI インタフェースを使用してアクセスできます。

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図解したものです。

図 1-1 リモート・オブジェクト・アクセス



IIOP

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 と CORBA のプログラミング・モデルが実装されます。

詳細情報

この項には、CORBA の詳細および Java を使用した CORBA アプリケーション開発の詳細について参照できる情報源をまとめてあります。

書籍

Oracle8i JVM に付属する ORB と CORBA サービスの一部は、Inprise からライセンスが提供されている Java コード用の VisiBroker に準拠しています。『Programming with VisiBroker』(D. Pedrick 他著、John Wiley and Sons、1998 年) では、VisiBroker の観点から CORBA 開発を紹介し、VisiBroker CORBA 環境の詳細を説明しています。

『Client/Server Programming with Java and CORBA』(R. Orfali、D. Harkey 著、John Wiley and Sons、1998 年) では、Java による CORBA 開発を取り扱っています。このマニュアルではまた、例として VisiBroker 実装を使用します。

これらの書物の中で公開されている例を Oracle8i ORB で実行するためには、一部を修正する必要があることに注意してください。手始めにこのマニュアルの付録にある例を使用してみることをお薦めします。引用した書物の中の例よりも多岐にわたっており、Oracle8i CORBA のすべての機能が説明されています。Oracle8i 実装との主要な相違点は、[付録 B 「Oracle8i JVM ORB と VisiBroker VBJ ORB の比較」](#) も参照してください。

URL

次の Web サイトに用意されているリンクから、CORBA 2.0 の仕様書と CORBA サービスの仕様書をダウンロードできます。

<http://www.omg.org/>

次の Web サイトから、Inprise の VisiBroker for Java 製品のマニュアルをダウンロードできます。

<http://www.inprise.com/techpubs/visibroker/visibroker33/>

スタート・ガイド

この章では、Oracle8i 用の CORBA アプリケーションの基本的な作成手順について説明します。この章では、Oracle8i CORBA アプリケーション開発の基礎に重点が置かれています。CORBA アプリケーションの高度なプログラミング技法とその他のヒントについては、[第 5 章「高度な CORBA プログラミング」](#)を参照してください。

この章では、次のトピックを説明します。

- [最初の CORBA アプリケーション](#)
- [インタフェース定義言語 \(IDL\)](#)
- [ORB とサーバー・オブジェクトの活性化](#)
- [デバッグ手法](#)

最初の CORBA アプリケーション

この項では、Oracle8i JVM CORBA アプリケーションの開発プロセスを紹介します。クライアント・システム上で動作する単純ではあっても有用なプログラムを書き、IIOP を使用して Oracle に接続し、Oracle8i Java VM 内で活性化されて実行される CORBA サーバー・オブジェクトのメソッドを起動する方法を説明します。

図 2-1 CORBA アプリケーションのコンポーネント

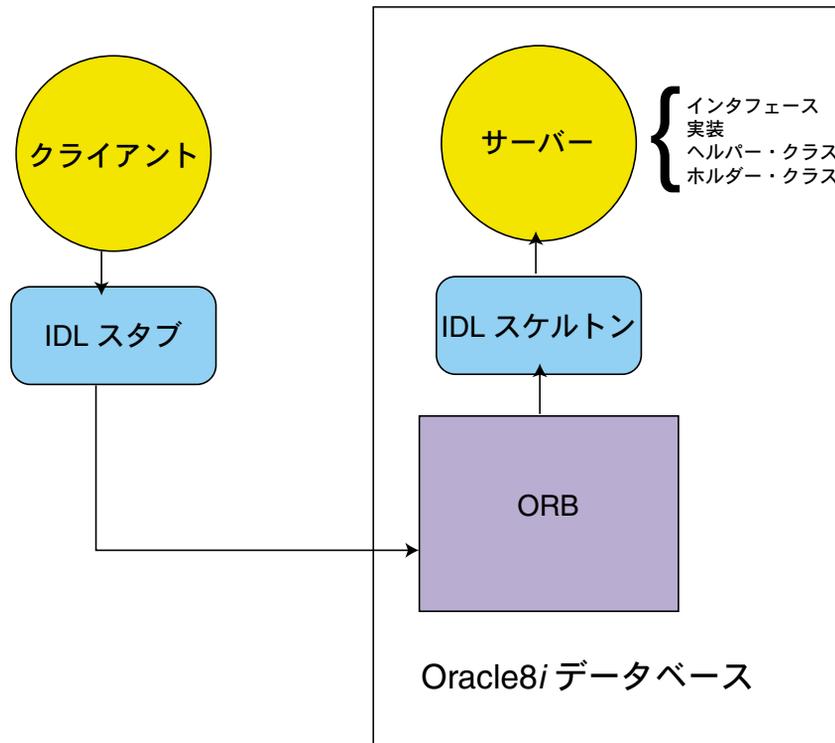


図 2-1 のように、CORBA アプリケーションではクライアント実装、サーバーのインタフェースと実装、および IDL スタブとスケルトンが必要です。各コンポーネントの作成手順は、次のとおりです。

1. IDL でオブジェクト・インタフェースを設計し、コーディングします。
2. スタブ、スケルトンおよびヘルパーとホルダーのサポート・クラスを生成します。
3. サーバー・オブジェクトの実装部をコーディングします。

4. クライアントの実装部をコーディングします。このコードは、Oracle8i データ・サーバーの外のワークステーションまたはパーソナル・コンピュータ上で実行されます。
5. クライアント側の Java コンパイラを使用して、Java サーバーの実装部をコンパイルします。また、IDL コンパイラにより生成された Java クラスをすべてコンパイルします。これらのクラスおよびその他の必要なリソース・ファイルを収めた JAR ファイルを生成します。
6. JDK Java コンパイラを使用してクライアント・コードをコンパイルします。
7. loadjava ツールを使用し、引数として JAR ファイルを指定して、コンパイルされたクラスを Oracle8i データベースにロードします。スタブやスケルトンなど、生成されたすべてのクラスを含めるようにします。クライアント・スタブがサーバーで必要になるのは、サーバー・オブジェクトが別の CORBA オブジェクトに対するクライアントとして動作する場合のみです。
8. CosNaming サービスで直接アクセス可能なオブジェクトの名前を公開し、クライアント・プログラムからアクセスできるようにします。

この章で使用する例では、EMP 表内の従業員番号を尋ねると、従業員の姓と現在の給与額が戻されます。その ID 番号を持つ従業員がデータベースに登録されていない場合は、例外が発生します。

IDL でのインタフェースのコーディング

サーバー・アプリケーションをコーディングするときには、インタフェース定義言語 (IDL) ファイルを作成してサーバーのインタフェースを定義する必要があります。インタフェースは、CORBA オブジェクトを定義するテンプレートです。一般的なオブジェクト指向言語でのオブジェクトと同様に、メソッドと、読み込みや設定のできるデータ要素が含まれています。ただし、インタフェースはあくまでも定義であり、オブジェクトが存在する場合のインタフェースの内容を定義するだけのものです。IDL ファイル中では、各インタフェースが 1 つのオブジェクトと、そのオブジェクトに対してクライアントが実行できる操作を記述しています。

注意： IDL の詳細は、2-14 ページの「[インタフェース定義言語 \(IDL\)](#)」を参照してください。

この例には、単一のサーバー側メソッドのみを含むファイル `employee.idl` が含まれています。`getEmployee` メソッドは、ID 番号をパラメータに取り、従業員の氏名と給与に関してデータベースに問合せを実行します。

このインタフェースでは、次の3つの要素が定義されます。

- データベースに問合せを実行して情報を戻す `getEmployee` メソッド
- 戻される情報を保持する `EmployeeInfo` データ構造
- 従業員が見つからない場合に発生する `SQLException` 例外

`employee.idl` ファイルの内容は、次のようになります。

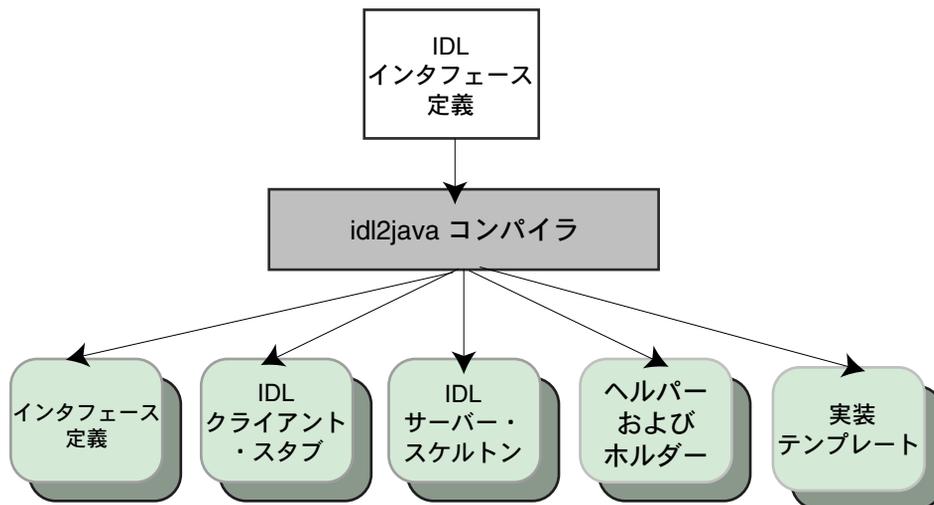
```
module employee {  
  
    struct EmployeeInfo {  
        wstring name;  
        long number;  
        double salary;  
    };  
  
    exception SQLException {  
        wstring message;  
    };  
  
    interface Employee {  
        EmployeeInfo getEmployee (in long ID) raises (SQLException);  
    };  
};
```

スタブとスケルトンの生成

`idl2java` コンパイラを使用して、インタフェース定義をコンパイルします。図 2-2 のように、コンパイラは、IDL ファイルに記述されている3つのオブジェクトに対してインタフェース・クラス、実装テンプレート・クラス、ヘルパー・クラスおよびホルダー・クラスを生成し、さらに `Employee` インタフェースに対してスタブおよびスケルトン・クラスを生成します。各クラスの詳細は 2-15 ページの「IDL の使用」、`idl2java` コンパイラの詳細は『Oracle8i Java Tools リファレンス』を参照してください。

注意： この例では `Tie` のメカニズムは役に立たないので、コンパイラを起動するときに `-no_tie` オプションを付けます。これによって生成されるクラスの数が増えることとなります。

図 2-2 IDL のコンパイルによるサポート・ファイルの生成



IDL を次のようにしてコンパイルします。

```
% idl2java -no_tie -no_comments employee.idl
```

注意： CORBA アプリケーションの開発では、コンパイル、ロード、公開のステップを何回も繰り返すため、コマンドライン指向の環境で作業している場合には、必ず Make ファイルやバッチ・ファイルを使用してプロセスを制御することをお勧めします。あるいは、Oracle の JDeveloper などの IDE 製品を使用してプロセスを制御できます。

employee.idl ファイルのコンパイル中に、次のファイルが生成されます。

ファイル名	ファイル・タイプ
_example_Employee.java	サーバー・オブジェクトの実装テンプレート
Employee.java	Employee インタフェース定義
EmployeeInfo.java	EmployeeInfo インタフェース定義
SQLException.java	SQLException インタフェース定義
_st_Employee.java	IDL クライアント・スタブ
_EmployeeImplBase.java	IDL サーバー・スケルトン

ファイル名	ファイル・タイプ
EmployeeHelper.java	Employee のヘルパー・クラス。このクラスで最も重要なメソッドは、戻されるオブジェクトを Employee オブジェクトに型キャストする narrow メソッドと、インタフェースの識別子を戻す id メソッドです。
EmployeeHolder.java	Employee のホルダー・クラス。ホルダー・クラスにより、Java オブジェクトで値をクライアントに渡すことができます。
EmployeeInfoHelper.java	EmployeeInfo のヘルパー・クラス
EmployeeInfoHolder.java	EmployeeInfo 構造体のホルダー・クラス
SQLExceptionHelper.java	SQLException のヘルパー・クラス
SQLExceptionHolder.java	SQLException 例外のホルダー・クラス

アプリケーション開発者が手を加えるファイルは、`_example_Employee.java` ファイルのみです。`_example_Employee.java` ファイルを、`EmployeeImpl.java` のようにより適切な名前に変更する必要があります。改名後に、このファイルに変更を加えてサーバーの実装部を追加します。`EmployeeImpl.java` ファイルにより、IDL サーバー・スケルトン `_EmployeeImplBase.java` が拡張 (extends) されます。`Employee.java` インタフェースに定義されている `getEmployee` メソッドを追加して実装します。さらに、これらのメソッドを適切に起動するクライアント・アプリケーションを作成する必要があります。`EmployeeImpl.java` 内で `Employee` のサーバー実装部を作成する方法の具体例は、2-6 ページの「サーバー・オブジェクトの実装部のコーディング」を参照してください。

サーバー・オブジェクトの実装部のコーディング

実装とは、インタフェースが実際にインスタンス化されたものです。つまり、IDL インタフェースに定義されたすべてのファンクションとデータ要素を実装するコードです。`Employee` インタフェースの実装方法は、次のとおりです。

- 元は `_example_Employee.java` ファイルであった `EmployeeImpl.java` ファイルを変更し、サーバー実装部を追加します。**EmployeeImpl** は、IDL で生成されたスケルトン `_EmployeeImplBase` を拡張 (extends) しています。

図 2-1 のように、`_EmployeeImplBase` の IDL スケルトンは ORB とサーバー・アプリケーションの間に存在するため、サーバー・アプリケーションの起動はこの IDL スケルトンを介して実行されます。スケルトンによりパラメータが準備され、サーバー・メソッドが起動され、戻り値、出力パラメータまたは入力パラメータが受けとられます。

2. `getEmployee` メソッドを実装し、従業員に関してデータベースに問合せを実行し、`EmployeeInfo` で適切な氏名と給与を戻します。

```
package employeeServer;

import employee.*;
import java.sql.*;

public class EmployeeImpl extends _EmployeeImplBase {

    /*constructor*/
    public EmployeeImpl() {
    }

    /*getEmployee method queries database for employee info*/
    public EmployeeInfo getEmployee (int ID) throws SQLException {
        try {
            /*create a JDBC connection*/
            Connection conn =
                new oracle.jdbc.driver.OracleDriver().defaultConnection ();

            /*Create a SQL statement for the database query*/
            PreparedStatement ps =
                conn.prepareStatement ("select ename, sal from emp where empno = ?");
            /*set the employee identifier and execute query. return the
            result in an EmployeeInfo structure */
            try {
                ps.setInt (1, ID);
                ResultSet rset = ps.executeQuery ();
                if (!rset.next ())
                    throw new SQLException ("no employee with ID " + ID);
                return new EmployeeInfo (rset.getString (1), ID, rset.getFloat (2));
            } finally {
                ps.close ();
            }
        }
        /*If a problem occurs, throw the SQLException exception*/
        catch (SQLException e) {
            throw new SQLException (e.getMessage ());
        }
    }
}
```

このコードでは、データベース問合せの実行に JDBC API を使用します。この実装では、問合せの WHERE 句に変数を入れるために、コンパイル済みの SQL 文を使用しています。Oracle8i JDBC 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。SQL 文が静的な場合は、JDBC のかわりに SQLJ を使用できます。

Oracle8i サーバー・アプリケーションと他の ORB アプリケーションの比較

一般的に、ORB アプリケーションは、サーバー実装をインスタンス化し、このインスタンスを CORBA オブジェクト・アダプタに登録するサーバー・アプリケーションを提供する必要があります。しかし、Oracle8i では、Oracle8i JVM により実装がインスタンス化され、そのインスタンスが必要に応じて登録されます。したがって、ORB の初期化、実装のインスタンス化およびインスタンスの登録のためのコードを提供する必要はありません。提供するサーバー・コードは、実際のサーバー実装部のみです。クライアントは ORB を介して、アクティブなサーバー実装インスタンスを検索できません。これは、サーバー実装インスタンスは呼び出されるまでインスタンス化されないためです。実装オブジェクトは CosNaming サービスで公開する必要があります。クライアントは、JNDI lookup を介してネーム・サービスからオブジェクトを取得します。取得後に、クライアントはオブジェクトのインスタンスを初期化する activate メソッドを起動します。この時点で、クライアントはオブジェクト上でメソッドを起動できます。

クライアント・コードの作成

サーバーをコーディングした後に、クライアントを作成する必要があります。サーバー・オブジェクトにアクセスするには、名前をもとに、オブジェクト参照を入手します。クライアントからサーバー・オブジェクトにアクセスするには、Oracle8i データベース内にサーバー・オブジェクトを公開します。クライアント・コードは公開されている名前を検索し、同時にサーバー・オブジェクトを活性化します。サーバー・オブジェクトの検索には、JNDI または CosNaming を使用できます。後述の例は、JNDI を利用する場合を示しています。JNDI と CosNaming の詳細は、4-3 ページの「[JNDI 接続の基本事項](#)」を参照してください。

JNDI lookup を実行すると、サーバー側の ORB が起動され、初期コンテキスト・オブジェクトを作成するときに用意された環境プロパティを使用してクライアントの認証が行われます。6-1 ページの「[IIOP のセキュリティ](#)」を参照してください。

ネーム・サービスからオブジェクトを取得するには、次を指定する必要があります。

- オブジェクト名
- IIOP サービス名
- クライアント認証情報

オブジェクト名

オブジェクト名では、公開されているオブジェクトの完全なパス名を指定します。たとえば、/test/myServer などとします。

lookup() メソッドの詳細は、4-10 ページの「[JNDI InitialContext の取得](#)」を参照してください。

IIOP サービス名

サービス名は、IIOP プレゼンテーションにより管理されるサービスを指定し、データベース・インスタンスを示します。サービス URL の形式は、4-29 ページの「[JNDI を使用しない CORBA オブジェクトへのアクセス](#)」で説明します。簡単に言うと、サービス名は次のコンポーネントを指定します。

- サービスの URL 接頭辞
- サービス・プレゼンテーションを管理するホストの名前
- そのホスト上のターゲット・データベース・インスタンスのリスナーのポート番号
- ホスト上のデータベース・インスタンスに対するシステム識別子 (SID)

サービス名の代表的な例は、`sess_iiop://localhost:2481:ORCL` です。`sess_iiop` はサービスの URL 接頭辞、`localhost` はデフォルトでローカル・データベースのホスト、`2481` は IIOP 接続のデフォルトのリスナー・ポート、`ORCL` は SID です。

クライアント認証情報

接続するたびに、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。認証情報のタイプは、認証方法、つまり、ユーザー名 / パスワードの組合せを使用するか、または SSL 証明書を使用するかによって異なります。詳細は、6-1 ページの「[IIOP のセキュリティ](#)」を参照してください。

クライアントの例

クライアントは、次のステップに従って `getEmployee` メソッドを起動します。

1. JNDI の `InitialContext` オブジェクトをインスタンス化し、認証情報など必要な接続プロパティで初期化します。4-3 ページの「[JNDI 接続の基本事項](#)」を参照してください。
2. サービス名と検索するオブジェクトの名前を指定するパラメータとして URL を使用し、初期コンテキストで `lookup()` メソッドを起動します。`lookup()` メソッドは、Employee CORBA サーバー・オブジェクトのオブジェクト参照を戻します。詳細は、4-8 ページの「[JNDI を使用したバインド済みオブジェクトへのアクセス](#)」を参照してください。
3. `lookup()` メソッドが戻すオブジェクト参照を使用して、サーバー内のオブジェクトに対して `getEmployee()` メソッドを起動します。このメソッドは、IDL の `EmployeeInfo` 構造体に基づいた `EmployeeInfo` クラスを戻します。例を簡単にするため、従業員 ID 番号はこのメソッド起動のパラメータとしてハードコーディングされています。

4. EmployeeInfo クラスの getEmployee () が戻した値を表示します。

```
import employee.*;
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";

// Step 1: Populate the JNDI properties with connect and authentication
// information
        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);

// Step 2: Lookup the object providing the service URL and object name
        Employee employee = (Employee)ic.lookup (serviceURL + objectName);

// Step 3 (using SCOTT's employee ID number): Invoke getEmployee
        EmployeeInfo info = employee.getEmployee (7788);

// Step 4: Print out the returned values.
        System.out.println (info.name + " " + info.number + " " + info.salary);
    }
}
```

クライアント・コードが実行されると、クライアント・コンソールに次のように出力されます。

```
SCOTT 7788 3000.0
```

Java ソースのコンパイル

クライアント側の Java バイト・コード・コンパイラ `javac` を実行し、作成した Java ソースをすべてコンパイルします。Java ソースには、クライアントとサーバーのオブジェクト実装、および IDL コンパイラにより生成された Java クラスが含まれます。

Employee の例では、次のファイルをコンパイルします。

- employee/Employee.java
- employee/EmployeeHelper.java
- employee/EmployeeHolder.java
- employee/EmployeeInfo.java
- employee/EmployeeInfoHelper.java
- employee/EmployeeInfoHolder.java
- employee/SQLException.java
- employee/SQLExceptionHelper.java
- employee/SQLExceptionHolder.java
- employee/_EmployeeImplBase.java
- employee/_st_Employee.java
- EmployeeImpl.java
- Client.java

Java コンパイラが使用する依存関係に従って、生成されたその他の Java ファイルがコンパイルされます。

Oracle8i JVM は、Java JDK コンパイラ、リリース 1.1.6 または 1.2 をサポートしています。また、IDE 組込みのコンパイラなど、他の Java コンパイラを使用することもできます。

データベースへのクラスのロード

この例のために作成された EmployeeImpl オブジェクトなどの CORBA サーバー・オブジェクトは、Oracle8i データベース・サーバー内で実行されます。すべてのクラスは、必要に応じて ORB が活性化できるように、loadjava コマンドライン・ツールを介してサーバーにロードする必要があります。また、IDL で生成されたホルダー・クラスやヘルパー・クラスなどのすべての依存クラス、およびこの例の EmployeeInfo クラスなどのサーバー・オブジェクトにより使用されるクラスもロードします。

loadjava ツールを使用して、各サーバー・クラスを Oracle8i データベースにロードします。Employee の例では、次の方法で loadjava コマンドを発行します。

```
% loadjava -resolve -user scott/tiger
employee/Employee.class employee/EmployeeHolder.class
employee/EmployeeHelper.class employee/EmployeeInfo.class
employee/EmployeeInfoHolder.class employee/EmployeeInfoHelper.class
employee/SQLException.class employee/SQLExceptionHolder.class
employee/SQLExceptionHelper.class employee/_st_Employee.class
employee/_EmployeeImplBase.class employeeServer/EmployeeImpl.class
```

注意： クライアントを実装するクラスや、サーバー側で使用していないその他のクラスはロードしません。

サーバー・クラスを1つのJARファイルにまとめておき、loadjava コマンドに対する引数としてそのJARファイルを使用すると便利な場合もあります。この例では、次のコマンドを実行します。

```
% jar -cf0 myJar.jar employee/Employee.class employee/EmployeeHolder.class \  
employee/EmployeeHelper.class employee/EmployeeInfo.class \  
employee/EmployeeInfoHolder.class employee/EmployeeInfoHelper.class \  
employee/SQLException.class employee/SQLExceptionHolder.class \  
employee/SQLExceptionHelper.class employee/_st_Employee.class \  
employee/_EmployeeImplBase.class employeeServer/EmployeeImpl.class
```

loadjava コマンドを次のように実行します。

```
% loadjava -resolve -user scott/tiger myJar.jar
```

オブジェクト名の公開

アプリケーションを準備する最終ステップとして、Oracle8i データベース内の CORBA サーバー・オブジェクトの名前を公開します。オブジェクトを公開する方法は、4-4 ページの「[ネームスペース](#)」および『Oracle8i Java Tools リファレンス』の「[publish](#)」を参照してください。

たとえば、この項では、次のように publish コマンドを使用してサーバー・オブジェクトを公開できます。

```
% publish -republish -user scott -password tiger -schema scott  
-service sess_iiop://localhost:2481:ORCL  
/test/myEmployee employeeServer.EmployeeImpl employee.EmployeeHelper
```

このコマンドでは、次のものを指定します。

- `publish` — `publish` コマンドを実行します。
- `-republish` — 公開されている同じ名前のオブジェクトを上書きします。
- `-user scott` — `scott` は公開を行うスキーマに対するユーザー名です。
- `-password tiger` — Scott のパスワード。
- `-schema scott` — クラス名を解決するスキーマの名前。
- `-service sess_iiop://localhost:2481:ORCL` — サービス名を指定します (4-17 ページの「[サービス・コンテキスト・クラス \(ServiceCtx\)](#)」も参照してください)。
- `/test/myEmployee` — 公開されているオブジェクトの名前。

- `employeeServer.EmployeeImpl` — サーバー・オブジェクトを実装する、データベース内にロードされているクラスの名前。
- `employee.EmployeeHelper` — ヘルパー・クラスの名前。

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に、`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`)
- クライアントの ORB で使用するクラス・ファイル: VisiBroker for Java の `vbjapp.jar` および `vbjorb.jar` など
- 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・ディレクトリの下に `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:
$(ORACLE_HOME)/sqlj/lib/translator.zip:$(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
```

この例は、コマンドラインで次の引数を指定してクライアントを起動することを前提としています。

- CLASSPATH ライブラリ
- クライアント・オブジェクト
- サービス名
- 活性化する対象の公開されているオブジェクトの名前
- ユーザー名
- パスワード

注意： 前述のような長い java コマンドに対しては Make ファイルやバッチ・ファイルの使用が有用です。

インタフェース定義言語 (IDL)

CORBA はプログラミング言語からの独立性を提供します。ある言語で書かれた CORBA オブジェクトから、別の言語で書かれたオブジェクトにリクエストを送信できます。Java や Smalltalk などのオブジェクト指向言語で実装されたオブジェクトから C や COBOL で書かれたオブジェクトに通信したり、あるいはその逆を行うことができます。

プログラミング言語からの独立性は、IDL と呼ばれるメタ言語の使用により実現されています。このメタ言語は、オブジェクト（またはラッパーを使用してオブジェクトのようにしたレガシー・コード）が外部に示すインタフェースの定義に使用されます。他のオブジェクト指向システムの場合と同様、CORBA オブジェクトは独自のプライベート・データと独自のメソッドを持つことができます。パブリック・データおよびパブリック・メソッドが、オブジェクトにより外部に示されるインタフェースです。

IDL は、CORBA がそのオブジェクトを定義するために使用する言語です。IDL では実際のコードを書きません。つまり、IDL はデータ、メソッドおよび例外の指定にのみ使用されます。

IDL 定義を特定の言語に変換するコンパイラが、各 CORBA ベンダーから提供されています。Oracle8i JVM では、Inprise の idl2java コンパイラが採用されています。idl2java コンパイラにより、IDL インタフェース仕様が Java クラスに変換されます。このツールの詳細は、『Oracle8i Java Tools リファレンス』を参照してください。

注意： idl2java コンパイラは、ASCII 文字にのみ対応します。ISO Latin-1 やその他の非 ASCII NLS 文字を IDL ファイルで使用しないでください。

IDL の使用

次の例は、HelloWorld の例の IDL を示しています。この例の詳細については、A-2 ページの「基本例」を参照してください。

```
module hello {  
    interface Hello {  
        wstring helloWorld();  
    };  
};
```

この IDL は、1 つのモジュールで構成されています。モジュールは関連するオブジェクト・インタフェース群から成り立っています。IDL コンパイラでは、生成後に Java クラスを配置するディレクトリの名前としてモジュール名が使用されます。また、モジュール名は、生成されるクラスの Java パッケージの名前にも使用されます。

このモジュールでは、Hello という 1 つのインタフェースが定義されます。Hello インタフェースでは、helloWorld という 1 つのオペレーションが定義されます。helloWorld はパラメータを取らず、wstring (Java の String にマップされるワイド文字列) を戻します。

注意： 前述の wstring などの IDL データ型および例外タイプの定義については、このマニュアルでは説明しません。このマニュアルでは IDL から Java へのバインドのいくつかを一覧表にして掲載しますが (たとえば、2-18 ページの「IDL のタイプ」を参照してください)、CORBA 開発者は、IDL および IDL のタイプについては OMG 仕様書を読む必要があります。1-8 ページの「詳細情報」を参照してください。

モジュール名とインタフェース名は、Java の有効な識別名であることと、オペレーティング・システムに対して有効なファイル名であることが必要です。インタフェースおよびモジュールに名前を付ける場合は、Java オブジェクトと CORBA オブジェクトの両方が移植可能であること、および大 / 小文字区別のオペレーティング・システムもそうでないオペレーティング・システムもあることに注意して、プロジェクトにおいて一意の名前になるようにしてください。

ネストッド・モジュール

モジュールをネストできます。たとえば、次のモジュールを指定する IDL ファイルは、Java パッケージ階層 package org.omg.CORBA にマップされます。

```
module org {  
    module omg {  
        module CORBA {  
            ...  
        };  
    };  
};
```

```
};
...
};
```

IDL コンパイラの実行

HelloWorld オブジェクトの IDL が `hello.idl` というファイルに保存されているとします。`idl2java` を実行して `hello` モジュールをコンパイルすると、8 個の Java クラス・ファイルが生成され、IDL ファイルと同じディレクトリの `hello` という名前のサブディレクトリに格納されます。

```
% idl2java hello.idl
Traversing hello.idl
Creating: hello/Hello.java
Creating: hello/HelloHolder.java
Creating: hello/HelloHelper.java
Creating: hello/_st_Hello.java
Creating: hello/_HelloImplBase.java
Creating: hello/HelloOperations.java
Creating: hello/_tie_Hello.java
Creating: hello/_example_Hello.java
```

ORB ではこれらの Java クラスを使用してリモート・オブジェクトを起動し、パラメータを渡して結果を戻すなど、様々なことを実行します。生成されるファイル、ファイルの格納場所、および IDL コンパイル作業のその他の側面 (IDL コンパイラが Java ファイル内にコメントを生成するかどうかなど) を制御できます。`idl2java` コンパイラの詳細は、『Oracle8i Java Tools リファアレンス』を参照してください。

生成される各ファイルは、次のとおりです。

Hello このファイルでは、Hello オブジェクトへのインタフェースの内容が Java で記述されます。この場合、インタフェース は次のようになります。

```
package hello;
public interface Hello extends org.omg.CORBA.Object {
    public java.lang.String helloWorld();
}
```

ファイルは `hello` ディレクトリに置かれるので、パッケージ名もその名前になります。すべての CORBA インタフェース・クラスは、直接的あるいは間接的に `org.omg.CORBA.Object` のサブクラスになります。

このインタフェースにメソッドを実装する必要があります。`hello.java` インタフェースの実装クラス名には `helloImpl` を使用することをお勧めしますが、このネーミング規則は必須ではありません。

HelloHolder	インタフェースに定義されたオペレーション内のパラメータが <code>out</code> または <code>inout</code> というタイプの場合、アプリケーションでホルダー・クラスが使用されます。ORB では Java パラメータは値渡しされるので、パラメータの戻り値を指定するために特別なホルダー・クラスが必要です。
HelloHelper	ヘルパー・クラスには、ストリームへの読取り / 書込みを行うメソッドや、オブジェクトを基本クラスのタイプとの間でキャストするメソッドが含まれます。たとえば、ヘルパー・クラスには、次のコードのようにオブジェクトを適切なタイプにキャストするために使用される <code>narrow()</code> メソッドがあります。 <pre style="text-align: center;">LoginServer lserver = LoginServerHelper.narrow (orb.string_to_object (loginIOR));</pre> <p>JNDI <code>InitialContext lookup()</code> メソッドを使用してオブジェクト参照を取得する場合は、ヘルパー <code>narrow()</code> メソッドをコールする必要はないことに注意してください。ORB により自動的にコールされます。</p>
_st_Hello	<code>_st_</code> 接頭辞がインタフェース名に付いている生成ファイルはスタブ・ファイル、つまりクライアント・プロキシ・オブジェクトです (<code>_st_</code> は <code>VisiBroker</code> 固有の接頭辞です)。 これらのクラスは、リモート・オブジェクトをコールするクライアントにインストールされます。クライアントがリモート・オブジェクト上のメソッドをコールする場合、実際にはスタブをコールしており、コールされたスタブではリモート・メソッドを起動するのに必要なオペレーションが実行されます。たとえば、リモート・ホストへのトランスポートのためにパラメータ・データのマーシャリングが行われます。
_HelloImplBase	<code><interfaceName>ImplBase</code> という形式で生成されたソース・ファイルは、スケルトン・ファイルです。スケルトン・ファイルはサーバー上にインストールされ、クライアント上のスタブ・ファイルと通信します。クライアントからのメッセージを ORB をとおして受け付け、サーバーへアップコールします。スケルトン・ファイルはまた、パラメータや戻り値をクライアントに戻します。
HelloOperations _tie_Hello	サーバーでは、サーバー・オブジェクトの <code>Tie</code> 実装のためにこれらの 2 つのクラスが使用されます。 <code>Tie</code> クラスの詳細は、5-11 ページの「CORBA の <code>Tie</code> メカニズムの使用」を参照してください。
_example_Hello	<code>_example_<interfaceName></code> クラスは、サーバー・オブジェクト実装用のテンプレートを提供します。 <code>Hello</code> サーバー・オブジェクトを実装するディレクトリにこのコード例をコピーし、改名して (このマニュアルの例では <code>HelloImpl.java</code> を使用します)、メソッドを実装します。

IDL インタフェース本体

IDL インタフェース本体には、次のような種類の宣言が含まれています。

定数	インタフェースにより公開される定数値
型	型定義
例外	インタフェースにより公開される例外構造体
属性	インタフェースにより公開される関連属性
オペレーション	インタフェースによりサポートされるメソッド

IDL のタイプ

この項では、IDL のデータ型と、Java データ型へのマッピングを簡単に説明します。このマニュアルで取り扱っていない IDL の型の詳細は、CORBA 仕様書および 1-8 ページの「[詳細情報](#)」で紹介している書籍を参照してください。

基本型

IDL の基本型および Java の基本型との間のマッピングは直接的です。マッピングの一覧を [表 2-1](#) に示します。変換時に発生する可能性のある CORBA 例外も表に示してあります。

表 2-1 IDL データ型から Java データ型へのマッピング

CORBA IDL データ型	Java データ型	例外
boolean	boolean	
char	char	CORBA::DATA_CONVERSION
wchar	char	
octet	byte	
string	java.lang.String	CORBA::MARSHAL CORBA::DATA_CONVERSION
wstring	java.lang.String	CORBA::MARSHAL
short	short	
unsigned short	short	
long	int	
unsigned long	int	
long long	long	
unsigned long long	long	

表 2-1 IDL データ型から Java データ型へのマッピング (続き)

CORBA IDL データ型	Java データ型	例外
float	float	
double	double	

IDL のキャラクタ・タイプ `char` は 8 ビット・タイプであり、ISO Latin-1 文字を表します。これは、Unicode 文字を表す 16 ビット符号なし要素である、Java `char` 型にマップされます。パラメータのマーシャリングでは、Java の `char` を IDL の `char` にマップできない場合、CORBA `DATA_CONVERSION` 例外が発生します。

IDL `string` 型には IDL の文字列が入ります。Java `String` と IDL `string` との間の変換では、CORBA `DATA_CONVERSION` 例外が発生する場合があります。Java 文字列と IDL `string` および `wstring` との間の変換では、Java `String` が大きすぎて IDL 文字列に収まらない場合に、CORBA `MARSHALS` 例外が発生することがあります。

構造化された型

Java 開発者にとって最も利用することの多い IDL の構造化された型は `struct` でしょう。IDL コンパイラでは、IDL の `struct` が Java クラスに変換されます。たとえば、IDL 定義が次のようになっていると、

```
module employee {
    struct EmployeeInfo {
        long empno;
        wstring ename;
        double sal;
    };
    ...
}
```

IDL コンパイラでは `EmployeeInfo` クラスに対して独立した Java ソース・ファイルが生成されます。これは次のようになります。

```
package employee;
final public class EmployeeInfo {
    public int empno;
    public java.lang.String ename;
    public double sal;
    public EmployeeInfo() {
    }
    public EmployeeInfo(
        int empno,
        java.lang.String ename,
        double sal
    ) {
        this.empno = empno;
    }
}
```

```
        this.ename = ename;
        this.sal = sal;
    }
    ...
```

クラスには、struct 内の各フィールドに対してパラメータを持つパブリック・コンストラクタが含まれます。フィールドの値は、オブジェクトが作成されるときにインスタンス変数内に保存されます。通常、これらは CORBA オブジェクトに値渡しされます。

コレクション型

CORBA には、シーケンスおよび配列という 2 種類の順序付きコレクション型があります。IDL のシーケンスは、同じ名前の Java 配列にマッピングされます。IDL の配列は、各次元のサイズがコンパイル時に確定する多次元集合です。

ORB は、Java データをシーケンスまたは配列に変換したときにシーケンスまたは配列の範囲を超えると、実行時に CORBA MARSHAL 例外が発生します。

IDL ではまた、シーケンスのホルダー・クラスも生成されます。ホルダー・クラス名は、シーケンスのマッピングされた Java クラス名に `Holder` を付加したものです。

次の IDL のコードは、部門内の従業員に関する情報を表すために struct のシーケンスを使用する方法を示したものです。

```
module employee {
    struct EmployeeInfo {
        long empno;
        wstring ename;
        double sal;
    };

    typedef sequence <EmployeeInfo> employeeInfos;

    struct DepartmentInfo {
        long deptno;
        wstring dname;
        wstring loc;
        EmployeeInfos employees;
    };
};
```

IDL コンパイラが `DepartmentInfo` クラスに対して生成する Java クラス・コードは次のとおりです。

```
package employee;
final public class DepartmentInfo {
    public int deptno;
    public java.lang.String dname;
    public java.lang.String loc;
    public employee.EmployeeInfo[] employees;
};
```

```

public DepartmentInfo() {
}
public DepartmentInfo(
    int deptno,
    java.lang.String dname,
    java.lang.String loc,
    employee.EmployeeInfo[] employees
) {
    this.deptno = deptno;
    this.dname = dname;
    this.loc = loc;
    this.employees = employees;
}

```

シーケンス `employeeInfos` は、Java の配列 `EmployeeInfo[]` として生成されることに注意してください。

IDL における配列を次のように指定します。

```

const long ArrayBound = 12;
typedef long larray [ArrayBound];

```

IDL コンパイラではこれが次のように生成されます。

```

public int[] larray;

```

IDL の構造化された型およびコレクション型をアプリケーションで使用する場合、生成された `.java` ファイルをコンパイルし、クラスがサーバー・オブジェクトであるときには、コンパイルしたファイルを Oracle8i データベースにロードする必要があります。生成された `.java` ファイルをスキャンして、必要なファイルがすべてコンパイルされ、ロードされていることを確認してください。これらのタイプを定義している CORBA の例の `Makefile` (UNIX) または `makeit.bat` バッチ・ファイル (Windows NT) を見ると、IDL 生成クラスがどのようにコンパイルされ、データ・サーバーにロードされるかがわかります。

例外

`exception` キーワードを使用して IDL の新しいユーザー例外クラスを作成できます。たとえば 次のようになります。

```

exception SQLException {
    wstring message;
};

```

IDL では、オペレーションによってユーザー定義例外が発生する可能性があると宣言できません。たとえば、次のようになります。

```

interface employee {
    attribute name;
}

```

```

exception invalidID {
    wstring reason;
};
...
wstring getEmp(long ID)
    raises(invalidID);
};
};

```

CORBA システム例外

OMG CORBA システム例外とその Java 形式との間のマッピングもきわめて直接的です。マッピングの一覧を表 2-2 に示します。

表 2-2 CORBA と Java の例外

OMG CORBA の例外	Java の例外
CORBA::PERSIST_STORE	org.omg.CORBA.PERSIST_STORE
CORBA::BAD_INV_ORDER	org.omg.CORBA.BAD_INV_ORDER
CORBA::TRANSIENT	org.omg.CORBA.TRANSIENT
CORBA::FREE_MEM	org.omg.CORBA.FREE_MEM
CORBA::INV_IDENT	org.omg.CORBA.INV_IDENT
CORBA::INV_FLAG	org.omg.CORBA.INV_FLAG
CORBA::INTF_REPOS	org.omg.CORBA.INTF_REPOS
CORBA::BAD_CONTEXT	org.omg.CORBA.BAD_CONTEXT
CORBA::OBJ_ADAPTER	org.omg.CORBA.OBJ_ADAPTER
CORBA::DATA_CONVERSION	org.omg.CORBA.DATA_CONVERSION
CORBA::OBJECT_NOT_EXIST	org.omg.CORBA.OBJECT_NOT_EXIST
CORBA::TRANSACTIONREQUIRED	org.omg.CORBA.TRANSACTIONREQUIRED
CORBA::TRANSACTIONROLLEDBACK	org.omg.CORBA.TRANSACTIONROLLEDBACK
CORBA::INVALIDTRANSACTION	org.omg.CORBA.INVALIDTRANSACTION

IDL を使用しない開発

Oracle8i JVM 開発環境は、Inprise Caffeine ツールを備えており、CORBA モデルに基づく Java 分散アプリケーションを開発できます。インタフェース仕様を Java で書き、java2iiop ツールを使用して CORBA 互換 Java スタブおよびスケルトンを生成することができます。

開発者は java2idl ツールを使用して純粋な Java でコーディングできますが、Java をサポートしていない CORBA サーバーを使用する顧客向けに出荷できる IDL を手に入れることもできます。このツールは Java インタフェース仕様から IDL を生成します。

java2iiop および java2idl の詳細は、『Oracle8i Java Tools リファレンス』を参照してください。

ORB とサーバー・オブジェクトの活性化

CORBA アプリケーションでは、クライアントのシステムとサーバーが動作しているシステムの両方で ORB が起動している必要があります。この章のこれまでの例を見ると、ORB がどのようにして起動されるかは、クライアントでもサーバーでも明確ではありません。この項では、活性化のトピックを詳しく説明します。

クライアント側

クライアント側の ORB は、通常は次のどちらかの方法で初期化されます。

- クライアントが JNDI InitialContext オブジェクトに対して JNDI lookup() メソッドを介してサーバー・オブジェクトをインスタンス化するとき、ORB が暗黙的にインスタンス化されます。
- JNDI を使用しない純粋な CORBA クライアントが CORBA ORB init メソッドを起動するとき、ORB が明示的にインスタンス化されます。init メソッドの詳細は、5-12 ページの「[Aurora ORB インタフェース](#)」を参照してください。

注意： 前述の場合を除き、クライアント側で ORB.init メソッドを介して ORB を明示的に初期化するのは、コールバック・シナリオの場合のみです。コールバックの詳細は、5-3 ページの「[CORBA コールバックの実装](#)」を参照してください。サーバーによりコールバックされるオブジェクト内での ORB の初期化を示す例も含まれています。

サーバー側

IIOP リクエストを管理するプレゼンテーションにより、セッションが作成されたときにサーバー上で ORB が起動されます。ORB インスタンスを取得する場合は、CORBA の oracle.aurora.jndi.orb_dep.Orb.init メソッドを使用します。このメソッドの詳細は、5-12 ページの「[Aurora ORB インタフェース](#)」を参照してください。

オブジェクトの活性化について

オブジェクトは要求に応じて活性化されます。クライアントがオブジェクトを検索するとき、ORB によりオブジェクトがメモリー内にロードされ、キャッシュに格納されます。ORB では、オブジェクトを活性化するために、公開されたオブジェクトの完全修飾クラス名によりクラスが検索されます。クラス名は、実行者のスキーマではなく、公開時に定義されているスキーマで解決されます。詳細は、『Oracle8i Java Tools リファアレンス』のコマンドライン・ツール `publish` の説明を参照してください。

クラスが特定されると、ORB は `newInstance()` を使用してそのクラスの新しいインスタンスを作成します。このため、永続オブジェクト・クラスの引数なしのコンストラクタは**パブリック**であることが必要です。クラスで `oracle.aurora.AuroraServices.ActivableObject` インタフェースを実装している場合、`_initializeAuroraObject()` メッセージがインスタンスに送られます。`(_initializeAuroraObject())` を必要とする例は、5-11 ページの「[CORBA の Tie メカニズムの使用](#)」を参照してください。

サーバーの実装で `boa.obj_is_ready()` コールを使用して公開オブジェクトをオブジェクト・アダプタに登録する必要はありません。Oracle8i JVM ORB により登録は自動的に実行されます。

永続公開オブジェクトなど、その他のオブジェクトにより生成された一時オブジェクトは、`obj_is_ready()` を使用して BOA に登録します。例として、製品 CD の `examples/corba/basic/factory` ディレクトリ内の `factory` デモを参照してください。

CORBA インタセプタ

Visibroker を使用すると、インタセプタを実装できます。インタセプタの作成方法の詳細は、Visibroker のドキュメントを参照してください。

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、2-25 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_HOME` 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_HOME}/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、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。そこには、`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 2-1 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・ツールを介してデバッグ・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設定しています。

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` は特定の時間でタイムアウトになるため、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スレッドを再開します。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プロセスには、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せん。

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。

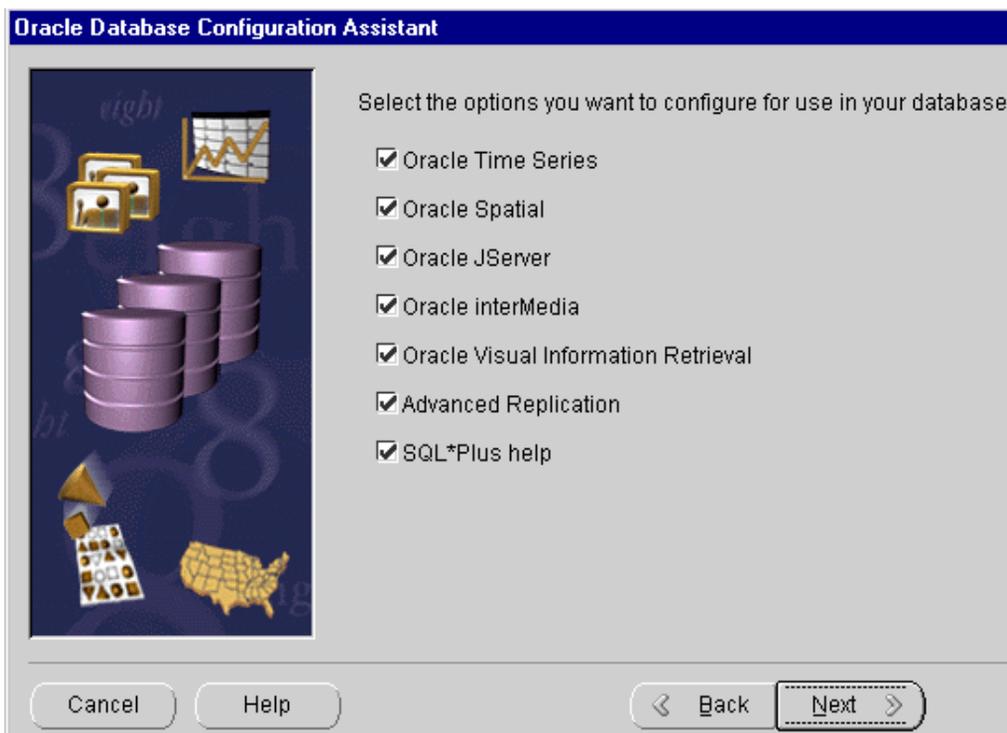
```
session_iiop://<hostname>/:<portnumber>/:<SID | service_name>
```

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。

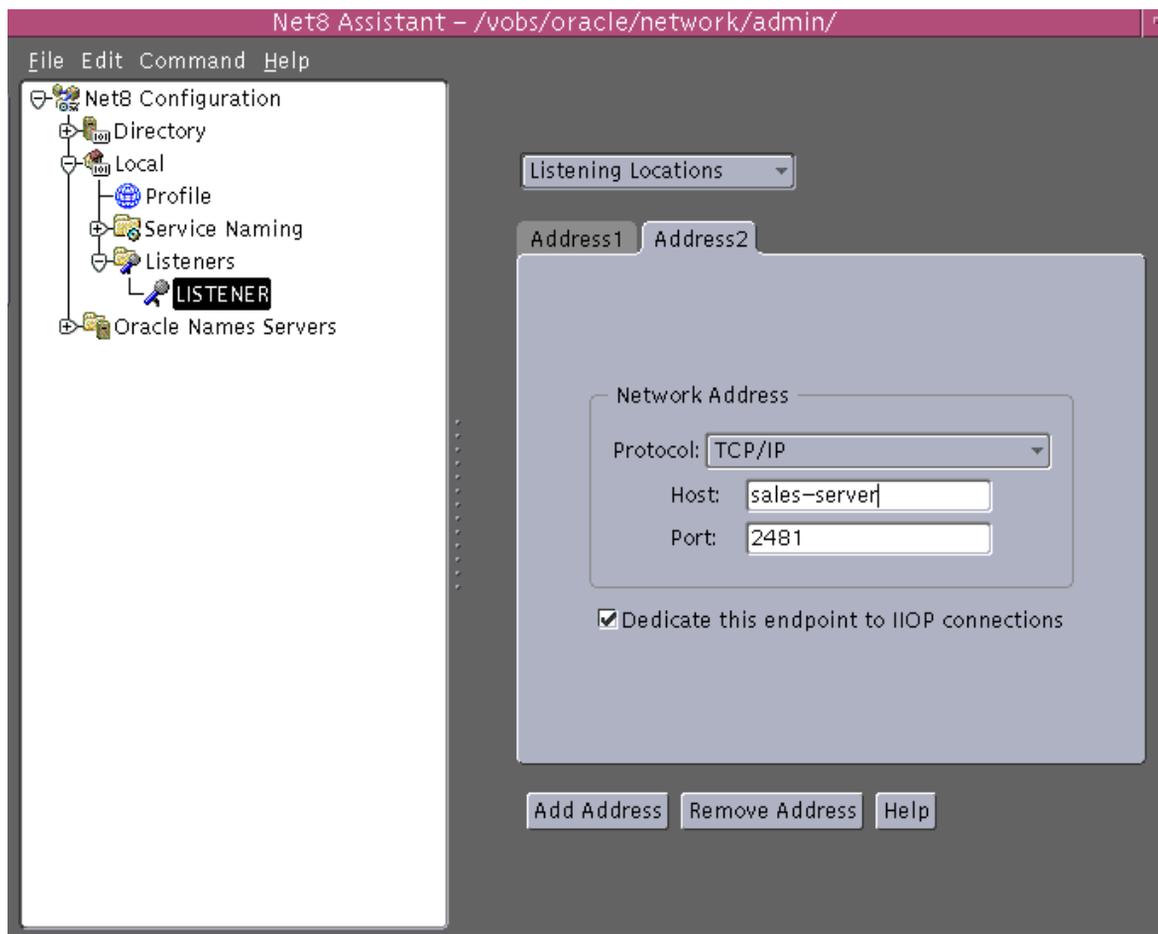
Net8 Assistant

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。セッション・ベースの IIOP を使用すると、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、セッションと、標準 IIOP のセマンティクスの両方によって識別されます。

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)	IIOIP 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 <code>oracle.aurora.server.SGiopServer</code> をサポートします。
(SESSION=RAW)	セッション層を識別します。IIOIP 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。

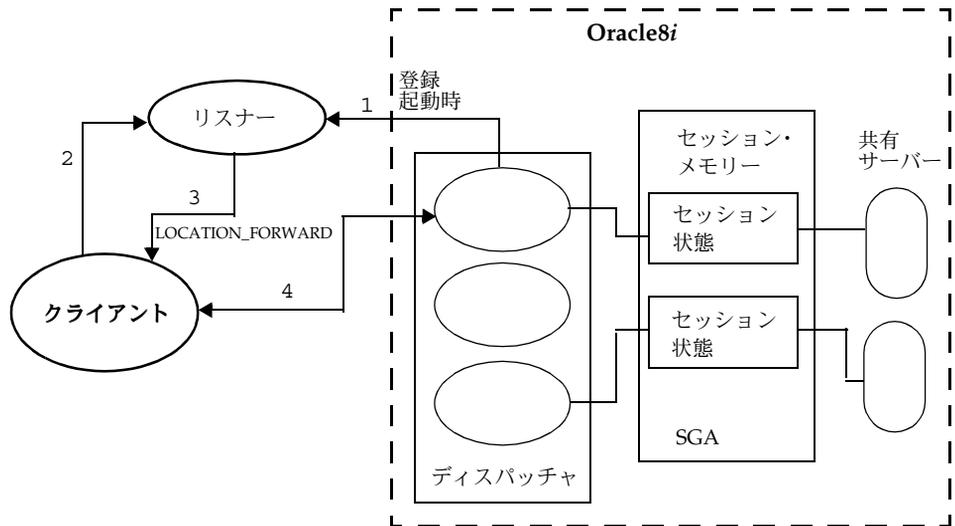
高度な構成に関するオプション

- 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、クライアントをサポートする 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、このセッション中に 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

に必要な応じて 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、既存のセッションで処理されます。

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。

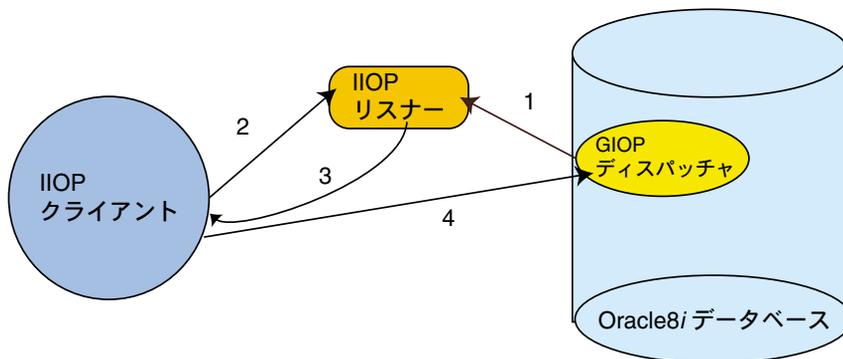
データベースの起動時に、すべてのディスパッチャは同じデータベース初期化ファイル内で構成されているリスナーすべてに登録します。ただし、IIOP クライアントが要求を起動すると、リスナーは GIOP ディスパッチャへの要求のリダイレクト、または汎用ディスパッチャへの接続の受渡しのみを行います。

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。

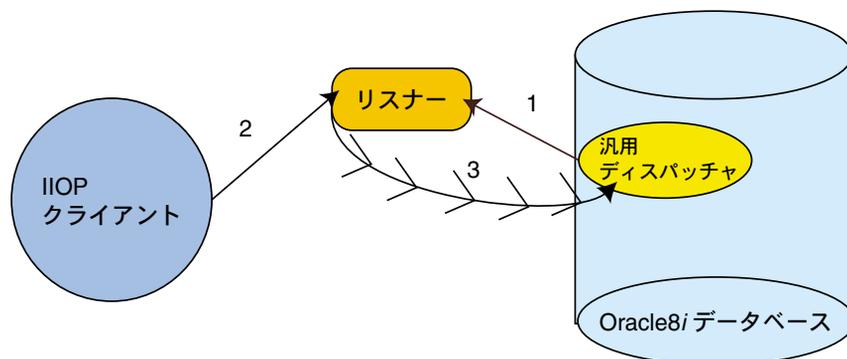
4. クライアント上の基礎となる ORB ランタイム・レイヤーが、初期要求を GIOP ディスパッチャに再送します。それ以後のすべてのメソッドの起動は、ディスパッチャに向けられます。リスナーは通信に関与しなくなります。

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。唯一の制限事項は、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、Net8 構成を介して静的に構成する方法と、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やり取りします。このチャンネルで、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 - 動的登録コマンド `regep` で登録します。

`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・エンドポイントをリスナーに追加できます。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。

このシナリオの制限事項は、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するため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 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。

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。このコマンドは、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、<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。

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使用もサポートされます。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) "
```

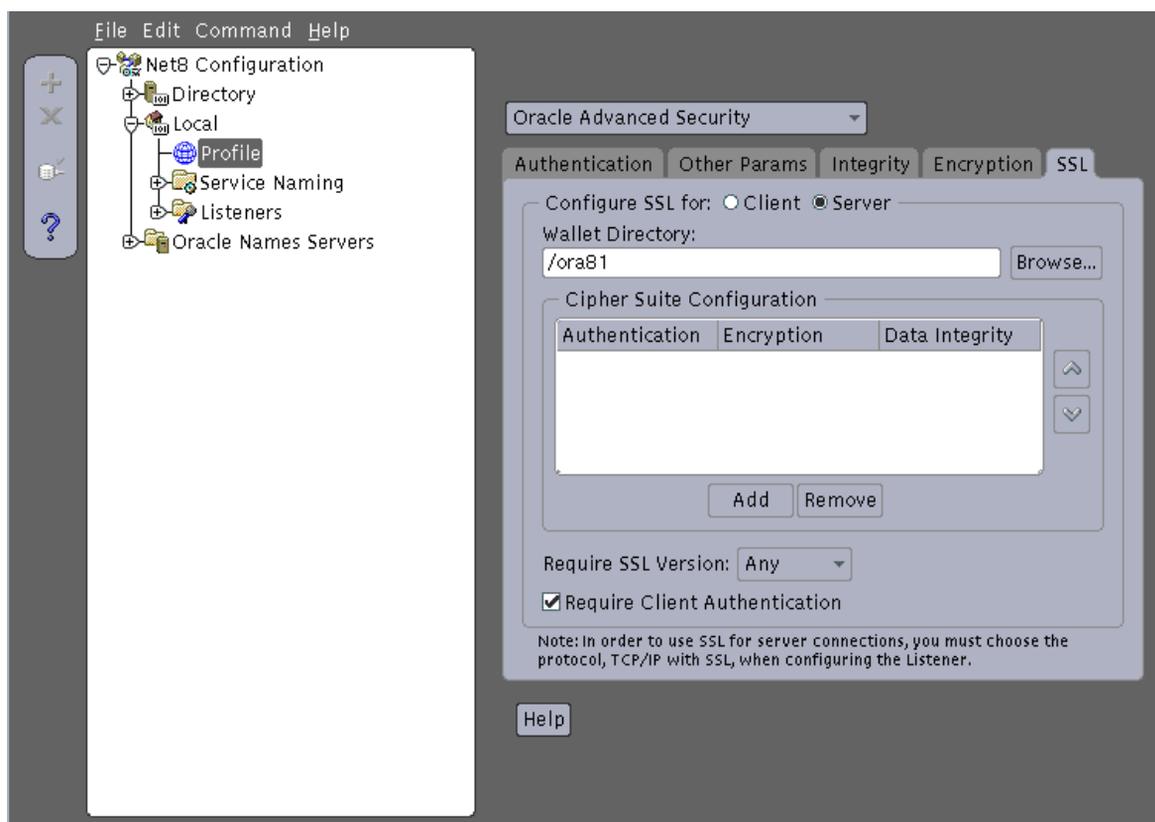
Net8 Assistant を介した Wallet の位置の設定

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 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」 > 「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)))
```

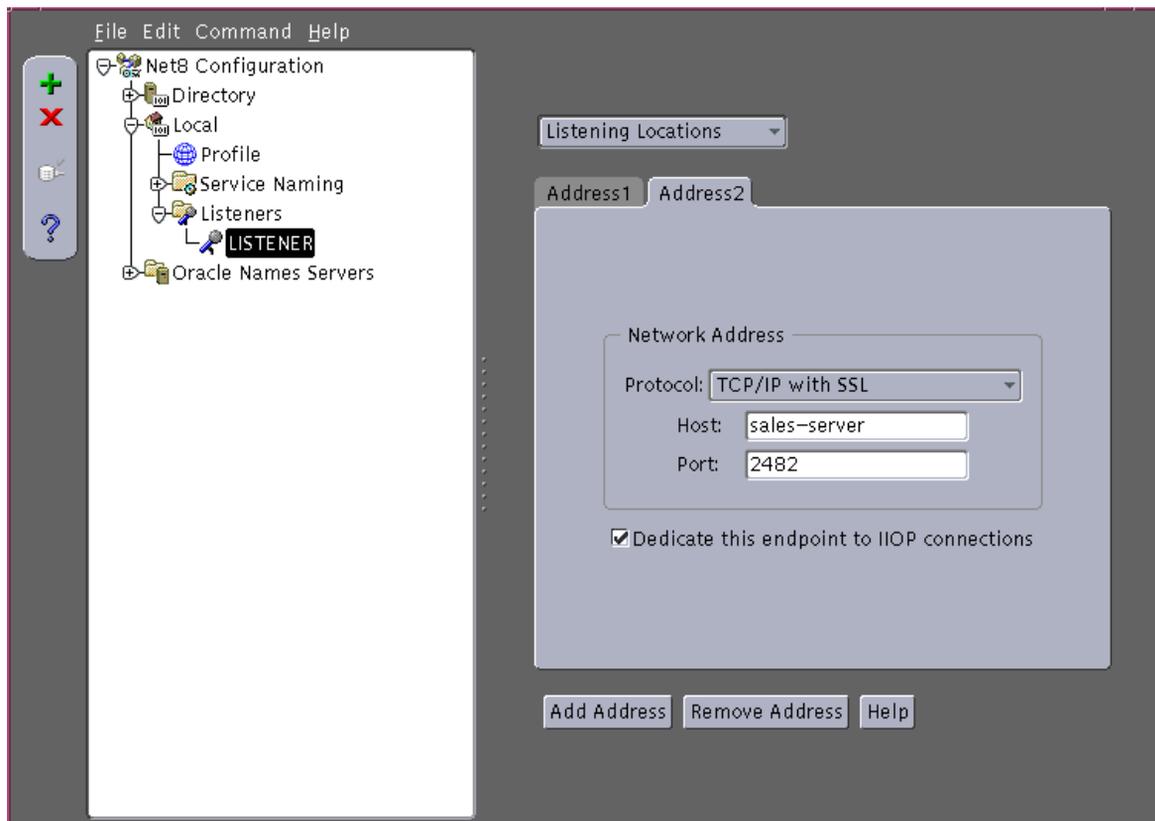
適切な証明書を持つ SSL Wallet の設定方法は、『Oracle8i Advanced Security 管理者ガイド』を参照してください。

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」>「Listener」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 3-7 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 でプロトコルとして (TCP ではなく) TCPS を使用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))
```

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。

CORBA オブジェクトを実行するには、最初に CORBA CosNaming サービスを使用して、これらのオブジェクトを Oracle8i データベース・インスタンス内で公開する必要があります。これにより、URL ベースの JNDI インタフェースを介して CosNaming から、または直接 CosNaming サービスから、オブジェクト参照を取得することができます。Java でコーディングされたクライアントでは、公開されているオブジェクトをより簡単に検索して活性化できるため、JNDI の使用をお勧めします。

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](#)
- [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 CORBA オブジェクトの活性化
 - JNDI を使用しない CORBA オブジェクトへのアクセス

JNDI 接続の基本事項

第2章のクライアントの例では、単一の URL 指定を使用して、Oracle に接続し、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. myHello hello =
   (myHello) ic.lookup("sess_iiop://localhost:2481:ORCL/test/myHello");
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、4-6 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により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はできますが、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 読み権限の保有者は、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 コンテキストの書き権限の保有者は、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・セルの `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

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の制御には、次のツールを使用できます。

- ロード時には、`-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権を付与するには、`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権を持っていても、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権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` でセッション・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。

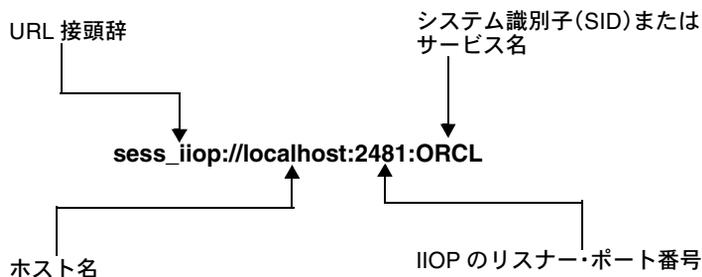
- ネーム・サービス
- InitialReferences オブジェクト (ブート・サービス)
- Login オブジェクト

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えます。サービス URL の例を図 4-1 に示します。

図 4-1 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ため、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、4-8 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` オブジェクトの取得時には、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得するには、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。

4-6 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ことが必要です。

- 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得する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得する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
sess_iiop:// <hostname>:<port>:<SID>	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、ServiceCtx オブジェクトが戻されます。セッション IIOP サービスは、IIOP 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得されます。詳細は、4-14 ページの「 セッション IIOP サービス 」を参照してください。
jdbc_access:	オブジェクトの lookup に JDBC 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 JTA の UserTransaction および DataSource オブジェクトを取得するために使用します。
java:	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 EJB 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、データベース内に複数の異なる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、4-14 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 lookup を行う場合は、4-6 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できる必要があります。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できる必要があります。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・インスタンスを取得したり、新規サービス・コンテキストを作成します。

- オブジェクト・インスタンスを取得するには、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` インスタンスを取得するには、次の操作を行います。

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

サービス名は `sess_iiop://localhost:2481:ORCL`、`Hello` 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得するには、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、4-19 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、アプリケーションで、1つ以上のデータベース・セッ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 (ServiceCtx)
- セッション・コンテキスト・クラス (SessionCtx)
- 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、CORBA 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、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 CORBA サーバー・オブジェクトを起動する必要があります。CORBA サーバー・オブジェクト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 CORBA サーバー・オブジェクト・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、4-19 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 4-2 TTC サービスおよび IIOP サービス

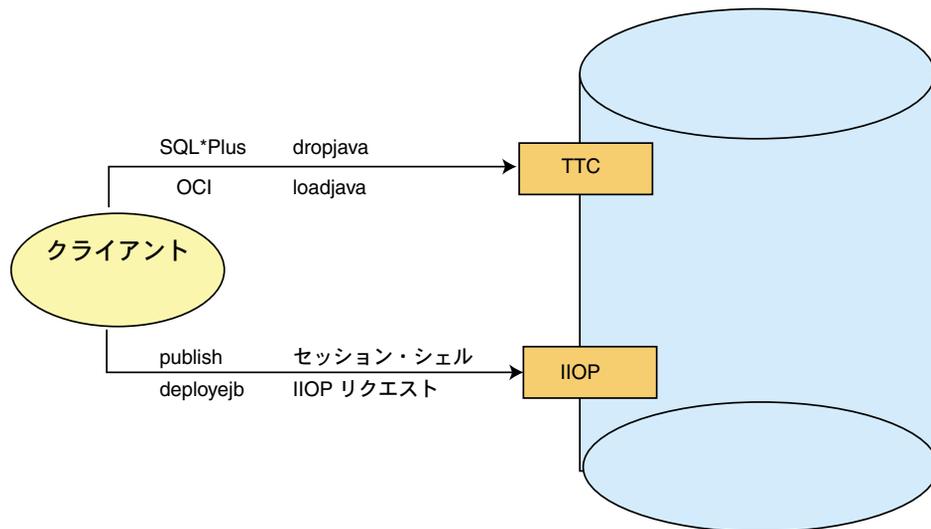


図 4-2 は、どのツールおよびリクエストで TTC が使用され、どのツールで IIOP データベース・ポートが使用されるかを示しています。TTC に対するデフォルトのポート番号は 1521 で、IIOP に対するデフォルトのポート番号は 2481 です。

- 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 (SessionCtx)** — 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 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 (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。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 4-1 を参照してください。

表 4-1 ServiceCtx Public 変数

文字列値の名前	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"
整数値の名前	整数値のコンストラクタ
SESS_IOP	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。

demo/examples/corba/basic ディレクトリにインストールされているデモの「sharedsession」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 (SessionCtx)

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、バインド済み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);
SomeObject myObj =
    (SomeObject) ic.lookup("sess_iiop://localhost:5521:ORCL/test/myObj");
```

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。上記の様にして 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。

```
SomeObject myObj =
    (SomeObject) ic.lookup("sess_iiop://localhost:5521:ORCL/test/myHello");
```

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。

```
Hello hello = (Hello)session.activate (objectName);
```

```
System.out.println (hello.helloWorld ());
```

例 4-1 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
Hello hello1 = (Hello)session1.activate (objectName);
Hello hello2 = (Hello)session2.activate (objectName);

// 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得され、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 4-2 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. セッションが作成され、URL を指定した `lookup` メソッドによってオブジェクトがインスタンス化されます。
3. `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得でき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
Hello hello = (Hello)ic.lookup (serviceURL + objectName);
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
String obj_hndl = orb.object_to_string (hello);
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` オブジェクトを取得します。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・システムから取得しています。

2. `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);
Hello hello = (Hello) orb.string_to_object(istream.readObject());
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得することができます。

```
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得しようとすると、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 を取得します。SessionCtx は、次のどちらかの方法で ServiceCtx から取得することができます。

- 次のように、最初に ServiceCtx を取得し、次に ServiceCtx から SessionCtx を取得します。

```
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 回の検索で取得します。

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

- 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。

```
MyObject myObj = (MyObject) sessCtx.activate("myObject");
// 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得する `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得することもできます。

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` を介して設定できます。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 lookup によって `Timeout` オブジェクトを取得します。
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返します。Version オブジェクトを取得するには、JNDI lookup で `/etc/version` を指定します。次の例では、バージョン・ナンバーを取得しています。

```
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 CORBA 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 CORBA 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 CORBA 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 CORBA 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。

JNDI を使用しない CORBA オブジェクトへのアクセス

クライアントは、この章の他の項で説明している JNDI クラスを使用せずにサーバー・オブジェクトにアクセスできます。この場合、クライアントは、CosNaming メソッドを使用して Oracle Server に接続します。

NameService の初期参照の取得

CORBA サーバー・オブジェクトのメソッドを使用するには、最初にネーミング・サービス・オブジェクトを取得する必要があります。Oracle8i では、ORB の `resolve_initial_references` メソッドを介して取得できる NameService オブジェクトが事前に公開されています。

CORBA では、NameService の初期参照を取得するために、ORBInitRef または ORBDefaultInitRef を使用する 2 通りの方法が用意されています。現時点で使用できるのは、ORBDefaultInitRef を使用する手法のみです。

ホスト、ポートおよび SID の形式で、ORBDefaultInitRef にサービス URL を指定する必要があります。または、ホスト、ポート、サービス名を使用してサービス URL を指定する方法もあります。さらに、次のようなオプション・プロパティも指定できます。

- 接続に SSL を使用する場合、ORBUseSSL プロパティを true に設定します。

```
System.setProperty("ORBUseSSL", "true");
```
- トランスポート・タイプには、sess_iiop または iiop を指定できます。TRANSPORT_TYPE プロパティを次のように設定します。

```
System.setProperty("TRANSPORT_TYPE", "sess_iiop");
```
- 先に BootService にアクセスせずに NameService を取得する場合は、次のように下位互換性プロパティ (ORBNameServiceBackCompat) を false に設定します。

```
System.setProperty("ORBNameServiceBackCompat", "false");
```
- サービス URL に SID のかわりにサービス名を使用します。次のように、USE_SERVICE_NAME プロパティを true に設定します。

```
System.setProperty("USE_SERVICE_NAME", "true");
```

注意： ORBDefaultInitRef または個々のプロパティ ORBBootHost、ORBBootPort および ORACLE_SID を介して、サーバー URL を初期化します。

例 4-3 CosNaming を使用したサーバー・オブジェクトの取得

次の例は、NameService オブジェクトを取得する方法を示しています。このオブジェクトからログインが実行され、サーバー・オブジェクトが取得されます。

```
import java.lang.Exception;

import org.omg.CORBA.Object;
import org.omg.CORBA.SystemException;
import org.omg.CosNaming.NameComponent;

import oracle.aurora.client.Login;
import oracle.aurora.AuroraServices.LoginServer;
import oracle.aurora.AuroraServices.LoginServerHelper;
import oracle.aurora.AuroraServices.PublishedObject;
import oracle.aurora.AuroraServices.PublishingContext;
import oracle.aurora.AuroraServices.PublishedObjectHelper;
```

```
import oracle.aurora.AuroraServices.PublishingContextHelper;

import Bank.Account;
import Bank.AccountManager;
import Bank.AccountManagerHelper;

public class Client {
    public static void main(String args[]) throws Exception {
        // Parse the args
        if (args.length < 4 || args.length > 5 ) {
            System.out.println ("usage: Client host port username password <sid>");
            System.exit(1);
        }
        String host = args[0];
        String port = args[1];
        String username = args[2];
        String password = args[3];
        String sid = null;
        if(args.length == 5)
            sid = args[4];

        // Declarations for an account and manager
        Account account = null;
        AccountManager manager = null;
        PublishingContext rootCtx = null;

        // access the Aurora Names Service
        try {
            // Initialize the ORB
            // The service URL for the server is provided in a string
            // that is prefixed with 'iioploc://' and includes either
            // host, port, sid or, if the USE_SERVICE_NAME is set to true,
            // host, port, service_name. This example uses host, port, sid
            // and sets it in the ORBDefaultInitRef.
            String initref;
            initref = (sid == null) ? "iioploc://" + host + ":" + port :
                "iioploc://" + host + ":" + port + ":" + sid;
            System.setProperty("ORBDefaultInitRef", initref);

            /*
             * Alternatively, you can set the host, port, sid or service in the
             * following individual properties. If set, these properties
             * take precedence over the URL set within the ORBDefaultInitRef property
            */
            System.setProperty("ORBBootHost", host);
            System.setProperty("ORBBootPort", port);
            if (sid != null)
```

```
    //set the SID. alternatively, if the USE_SERVICE_NAME property is
    //true, this should contain the service name instead of the sid.
        System.setProperty("ORACLE_SID", sid);
    */

/*
 * Some of the other properties that you can set
 * include the backwards compatibility flag, the service name
 * indicator, the SSL protocol definition, and the transport type.
System.setProperty("ORBNameServiceBackCompat", "false");
System.setProperty("USE_SERVICE_NAME", "true");
System.setProperty("ORBUseSSL", "true");
//transport type can be either sess_iiop or iiop
System.setProperty("TRANSPORT_TYPE", "sess_iiop");
*/

//initialize the ORB
com.visigenic.vbroker.orb.ORB orb =
    oracle.aurora.jndi.orb_dep.Orb.init();

// Get the Name service Object reference with the
// resolve_initial_references method
rootCtx = PublishingContextHelper.narrow(orb.resolve_initial_references(
    "NameService"));

//After retrieving the NameService initial reference, you must perform
// the login, as follows:
// Get the pre-published login object reference
PublishedObject loginPubObj = null;
LoginServer serv = null;
NameComponent [] nameComponent = new NameComponent [2];
nameComponent [0] = new NameComponent ("etc", "");
nameComponent [1] = new NameComponent ("login", "");

// Lookup this object in the Name service
Object loginCorbaObj = rootCtx.resolve (nameComponent);

// Make sure it is a published object
loginPubObj = PublishedObjectHelper.narrow (loginCorbaObj);

// create and activate this object (non-standard call)
loginCorbaObj = loginPubObj.activate_no_helper ();
serv = LoginServerHelper.narrow (loginCorbaObj);

// Create a client login proxy object and authenticate to the DB
Login login = new Login (serv);
login.authenticate (username, password, null);
```

```

// Now create and get the bank object reference
PublishedObject bankPubObj = null;
nameComponent [0] = new NameComponent ("test", "");
nameComponent [1] = new NameComponent ("bank", "");

// Lookup this object in the name service
Object bankCorbaObj = rootCtx.resolve (nameComponent);

// Make sure it is a published object
bankPubObj = PublishedObjectHelper.narrow (bankCorbaObj);

// create and activate this object (non-standard call)
bankCorbaObj = bankPubObj.activate_no_helper ();
manager = AccountManagerHelper.narrow (bankCorbaObj);

account = manager.open ("Jack.B.Quick");

float balance = account.balance ();
System.out.println ("The balance in Jack.B.Quick's account is $"
    + balance);
} catch (SystemException e) {
    System.out.println ("Caught System Exception: " + e);
    e.printStackTrace ();
} catch (Exception e) {
    System.out.println ("Caught Unknown Exception: " + e);
    e.printStackTrace ();
}
}
}

```

LoginServer、Login および LogoutServer オブジェクトの詳細は、4-20 ページの「セッションの終了」を参照してください。

ORBDefaultInitRef からの初期参照の取得

CORBA 2.3 Interoperable Name Service では、初期参照の作成と取得に関して ORBInitRef と ORBDefaultInitRef の両方の手法がサポートされます。現時点で、Oracle8i でサポートされるのは、ORBDefaultInitRef に IIOP URL を指定する方法のみです。4-29 ページの「NameService の初期参照の取得」を参照してください。初期参照の検索時には、ホスト、ポート、SID の組合せ、またはホスト、ポート、サービス名の組合せに接頭辞「iioploc://」を付けたもののみを、ORBDefaultInitRef に対して指定できます。このロケーションで、サービスは活性化済みである必要があります。指定したロケーションで活性化されたサービスは、resolve_initial_references メソッドでそのオブジェクト・キーを指定して取得することができます。オブジェクト・キーは、活性化の時点で定義されます。

たとえば、ORBDefaultInitRef を次のサーバー URL に設定するとします。

```
System.setProperty("ORBDefaultInitRef", "iioploc://myHost:myPort:mySID");
```

次のように ORB を初期化してサービスを取得します。

```
//initialize the ORB
com.visigenic.vbroker.orb.ORB orb = oracle.aurora.jndi.orb_dep.Orb.init();

// Get the myService service Object reference with resolve_initial_references
rootCtx = PublishingContextHelper.narrow(orb.resolve_initial_references(
    "myService"));
```

この例ではサービスの取得に使用するオブジェクト・キーは、「myService」です。このキーを持つオブジェクトが `resolve_initial_references` メソッドで戻されます。

ORB の起動時に活性化される Oracle8i サービスは、NameService、BootService、AuroraSSLCurrent および AuroraSSLCertificateManager です。

Oracle8i の ORB の起動時にサービスを活性化したい場合は、活性化したいサービスのカンマ区切りのリストを含む文字列を、UserORBServices プロパティに指定します。各サービスは、ORBServiceInit クラスを拡張するクラスのパッケージ名を含む完全修飾名にする必要があります。Oracle8i の ORB にサービスをインストールするには、このクラスを拡張する必要があります。

高度な CORBA プログラミング

この章では、サーバーからクライアントへのコールバックなど、高度な CORBA プログラミングの技法について説明します。セキュリティとトランザクションに関する高度なプログラミングについては、別個の章で説明します。この章では、次のトピックを説明します。

- [SQLJ の使用](#)
- [CORBA コールバックの実装](#)
- [IFR を使用したインタフェースの取得](#)
- [CORBA の Tie メカニズムの使用](#)
- [JDK 1.1 から Java 2 への移行](#)
- [アプレットからの CORBA オブジェクトの起動](#)
- [非 Oracle ORB との相互運用性](#)

SQLJ の使用

Oracle8i SQLJ を使用して静的 SQL オペレーションを実行することにより、CORBA サーバー・オブジェクトの実装が簡単になる場合がよくあります。SQLJ 文を使用すると、相当する JDBC コールに比べてコード量が少なくなり、実装が理解しやすく、デバッグしやすいものになります。この項では、2-2 ページの「[最初の CORBA アプリケーション](#)」で最初に示した例を取り上げますが、データベース・アクセスに JDBC ではなく SQLJ を使用します。SQLJ の詳細は、『Oracle8i SQLJ 開発者ガイドおよびリファレンス』を参照してください。

SQLJ を使用するに当たって、変更が必要な唯一のファイルは Employee オブジェクトを実装している EmployeeImpl.java です。SQLJ による実装では、EmployeeImpl.sqlj というファイル名になりますが、次のとおりです。2-6 ページの「[サーバー・オブジェクトの実装部のコーディング](#)」の同じオブジェクトの JDBC 実装と対比してみてください。

```
package employeeServer;

import employee.*;
import java.sql.*;

public class EmployeeImpl extends _EmployeeImplBase {
    public EmployeeInfo getEmployee (int ID) throws SQLException {
        try {
            String name = null;
            double salary = 0.0;
            #sql { select ename, sal into :name, :salary from emp
                where empno = :ID };
            return new EmployeeInfo (name, empno, (float)salary);
        } catch (SQLException e) {
            throw new SQLException (e.getMessage ());
        }
    }
}
```

SQLJ によるコードは、JDBC によるコードに比べてかなり短くなっています。Oracle では、一般に、静的な SQL 文のみ使用するアプリケーションでは SQLJ を、動的な SQL 文を使用する必要があるアプリケーションでは、JDBC、または JDBC と SQLJ の組合せを使用することをお勧めします。

SQLJ トランスレータの実行方法

EmployeeImpl.sqlj ファイルをコンパイルするには、次の SQLJ コマンドを発行します。

```
% sqlj -J-classpath
.:$(ORACLE_HOME)/lib/aurora_client.jar:$(ORACLE_HOME)/jdbc/lib/classes111.zip:
$(ORACLE_HOME)/sqlj/lib/translator.zip:$(ORACLE_HOME)/lib/vbjobr.jar:
$(ORACLE_HOME)/lib/vbjapp.jar:$(JDK_HOME)/lib/classes.zip -ser2class
employeeServer/EmployeeImpl.sqlj
```

このコマンドは、次の作業を行います。

- SQLJ コードの Java ファイルへの変換
- 変換後の .java ソースのコンパイルと、.class ファイルの出力
- -ser2class オプションによる SER ファイルの .class ファイルへの変換

SQLJ の変換で、さらに次の 2 つのクラス・ファイルが生成されます。

```
employeeServer/EmployeeImpl_SJProfile0
employeeServer/EmployeeImpl_SJProfileKeys
```

これらも、loadjava コマンドを実行するときにデータベースにロードする必要があります。

SQLJ の完全な例

この例は、examples/corba/basic example ディレクトリに完全な形で収められています。Make ファイルまたは Windows NT バッチ・ファイルが付属しているので、コード例をコンパイルしてロードする方法がわかります。

CORBA コールバックの実装

この項では、CORBA サーバー・オブジェクトがクライアントにコールバックする方法を説明します。この例で示す基本的な手法は次のとおりです。

- クライアント側で実行され、コールバック先のオブジェクトを稼動させるメソッドを含むクライアント・オブジェクトを記述します。
- クライアントのコールバック・オブジェクトへの参照をパラメータとして取るメソッドを持つサーバー・オブジェクトを実装します。

- クライアント・コードでは、
 - クライアントのコールバック・オブジェクトをインスタンス化します。
 - それを BOA に登録します。
 - それをコールするサーバー・オブジェクトにその参照を渡します。
- サーバー・オブジェクトの実装では、クライアントへのコールバックが実行されます。

注意： SSL 環境でのコールバックの使用例は、6-22 ページの「[セキュリティを使用したコールバック](#)」を参照してください。

IDL

この例の IDL を次に示します。次のように、`client.idl` および `server.idl` という 2 種類の IDL ファイルがあります。

```
/* client.idl */
module client {
    interface Client {
        wstring helloBack ();
    };
};

/* server.idl */
#include <client.idl>

module server {
    interface Server {
        wstring hello (in client::Client object);
    };
};
```

サーバー・インタフェースには、`client.idl` で定義されているインタフェースが含まれることに注意してください。

クライアント・コード

この例のクライアント・コードでは、サーバーからアクセスできるように、クライアント側コールバック・オブジェクトをインスタンス化して BOA に登録する必要があります。このコードはそのために次のステップを実行します。

- ORB 疑似オブジェクト上で、パラメータを付けずに `init()` メソッドを起動します。既存のクライアント側 ORB への参照が戻されます。
- ORB 参照を使用して、BOA を初期化します。

- 新しいクライアント・オブジェクトをインスタンス化します。
- クライアント・オブジェクトをクライアント側 BOA に登録します。

このステップを実行するコードは次のとおりです。

```
com.visigenic.vbroker.orb.ORB orb = oracle.aurora.jndi.orb_dep.Orb.init ();
org.omg.CORBA.BOA boa = orb.BOA_init ();
ClientImpl client = new ClientImpl ();
boa.obj_is_ready (client);
```

最後に、クライアント・コードは次のようにして、サーバー・オブジェクトをコールし、登録されているクライアント側コールバック・オブジェクトへの参照を渡し、戻り値を表示します。

```
System.out.println (server.hello (client));
```

コールバック・サーバーの実装

サーバー側オブジェクトの実装は次のように非常に単純です。クライアント側のコールバック・オブジェクトを受け取って、このオブジェクトからメソッドを起動します。この例では、サーバーはクライアント側の `helloBack` メソッドを起動します。

```
package serverServer;

import server.*;
import client.*;
import oracle.aurora.AuroraServices.ActivatableObject;

public class ServerImpl extends _ServerImplBase implements ActivatableObject
{
    public String hello (Client client) {
        return "I Called back and got: " + client.helloBack ();
    }

    public org.omg.CORBA.Object _initializeAuroraObject () {
        return this;
    }
}
```

サーバーでは単に、コールバックから戻された文字列値を含む文字列が戻されます。

コールバック・クライアント/サーバーの実装

クライアント側のコールバック・サーバーでは、必要なコールバック・メソッドが実装されます。次の例では、helloBack メソッドが実装されます。

```
package clientServer;

import client.*;
import oracle.aurora.AuroraServices.ActivatableObject;

public class ClientImpl extends _ClientImplBase implements ActivatableObject
{
    public String helloBack () {
        return "Hello Client World!";
    }

    public org.omg.CORBA.Object _initializeAuroraObject () {
        return this;
    }
}
```

クライアント側オブジェクトは、他のサーバー・オブジェクトにきわめて類似しています。しかし、このコールバックの例では、クライアント・システム上で実行できるクライアント ORB で実行されていますが、必ずしも Oracle8i データベース・サーバー内で実行されるわけではありません。

IFR を使用したインタフェースの取得

OMG により規定されたインタフェース・リポジトリ (IFR) では、インタフェース定義の格納方法と取得方法が定義されています。インタフェースに含まれる情報は、ORB 内部でオブジェクト参照に関する情報を取得したり、リクエストのシグネチャのタイプ・チェックを行うために使用されます。また、DII/DSI アプリケーションが DII/DSI を介して動的にオブジェクトをインスタンス化する場合にも、IFR の情報が利用できます。

IDL インタフェース定義は、Oracle8i JVM の publish コマンドを介して IFR に格納します。publish コマンドにより、インタフェースはフラット・ファイル AuroraIFR.idl に格納されます。

注意： 通常、このファイルは \$ORACLE_HOME/javavm/admin に自動的に書き込まれます。ただし、このディレクトリが書き込み可能になっていない場合は、次のように、modifyprops ツールを介して "aurora.ifr.file" システム・プロパティ内で別の完全修飾ファイル名を指定できます。

```
modifyprops -user scott/tiger@dbhost:5521:orcl
            "aurora.ifr.file" "/private/ifr/myIFRfile"
```

格納後のインタフェース定義は、`_get_interface_def` メソッドを介して暗黙的に取得したり、IFR の `Repository` オブジェクトを明示的に検索し、標準メソッドを起動し、リポジトリを探索して取得することができます。

次の項では、IDL インタフェース情報を公開し、取得する方法を詳しく説明します。

- IDL インタフェースの公開
- インタフェースの暗黙的取得
- インタフェースの明示的取得

IDL インタフェースの公開

IDL インタフェース定義は、Oracle8i JVM の `publish` コマンドを介して IFR に格納します。このコマンドには、IDL インタフェース定義を IFR に格納するための次の 2 つのオプションが含まれています。

- | | |
|--------------------------|--|
| <code>-idl</code> | IDL インタフェース定義を IFR にロードします。IDL インタフェース定義をロードできるのは、 <code>sess_iiop</code> の URL を使用するときのみです。 |
| <code>-replaceIDL</code> | IDL インタフェース定義がすでに IFR に存在している場合は、上書きします。このオプションを指定しなければ、 <code>publish</code> コマンドでは IFR 内の既存のインタフェースを上書きしません。 |

注意： `-replaceIDL` フラグを指定すると、最初に別のユーザーにより格納された場合でも、IFR 内で同じ名前を持つインタフェースがすべて置換されます。したがって、様々なユーザーが他のユーザーのインタフェースを意図せずに上書きする恐れがあります。

次の `publish` コマンドでは、IFR に `Bank.idl` インタフェースがロードされます。このコマンドは、SCOTT スキーマのセキュリティのパーミッションで実行されます。既存のインタフェースがある場合は、`-replaceIDL` オプションにより、このバージョンの `Bank.idl` に置換するように指定しています。

```
publish -republish -user SCOTT -password TIGER -schema SCOTT \
  -service sess_iiop://dlsun164:2481:orcl \
  /test/myBank bankServer.AccountManagerImpl \
  Bank.AccountManagerHelper -idl Bank.idl -replaceIDL
```

IFR から IDL インタフェースを削除するには、`sess_sh` の `remove` コマンドに `-idl` オプションを使用します。

インタフェースの暗黙的取得

インタフェース定義は、`org.omg.CORBA.Object._get_interface_def` メソッドを介して暗黙的に取得できます。戻されるオブジェクトは、`InterfaceDef` にキャストする必要があります。次のコードでは、`Bank.Account` の `InterfaceDef` オブジェクトを取得します。

```
AccountManager manager =
    (AccountManager)ic.lookup (serviceURL + objectName);

Bank.Account account = manager.open (name);

org.omg.CORBA.InterfaceDef intf = (org.omg.CORBA.InterfaceDef)
    account._get_interface_def();
```

取得後は、任意の `InterfaceDef` メソッドを実行してインタフェース情報を取得できます。

インタフェースの明示的取得

IFR には、すべての定義済みインタフェースが階層形式で格納されます。階層の最上位レベルは `Repository` オブジェクトで、これは `Container` オブジェクトでもあります。`Repository` オブジェクトの下位のオブジェクトは、すべて `Contained` オブジェクトです。必要なインタフェース定義が見つかるまで `Contained` オブジェクトを確認しながら、`Container` オブジェクトを下位へと探索できます。

注意： ユーザーが参照できるのは、自分が読取り権限を持っているオブジェクトのみです。

`Repository` オブジェクトは、`/etc/ifr` という名前で事前に公開されています。`Repository` オブジェクトは、次の処理を実行して取得できます。

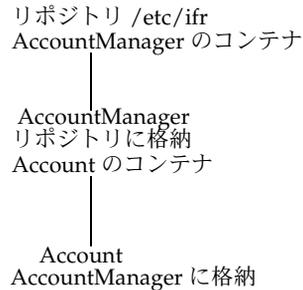
次のように `/etc/ifr` オブジェクトを検索し、事前に公開されている IFR `Repository` オブジェクトを取得します。

```
Repository rep = (Repository)ic.lookup(serviceURL + "/etc/ifr");
```

`Repository` オブジェクトの取得後は、必要なオブジェクトに達するまで階層を探索できます。各オブジェクト型のメソッド、`InterfaceDef` および他のメソッドは、OMG CORBA 仕様に詳細に記述されています。

図 5-1 のように、`Account` インタフェースは `AccountManager` に含まれており、`AccountManager` は `Repository` オブジェクト内のコンテナです。

図 5-1 Account インタフェースの IFR 階層



例 5-1 print メソッド内での IFR リポジトリの探索

IFR オブジェクトの取得後は、IFR に格納されているすべての定義を探索できます。例 5-1 の print メソッドでは、IFR にあるすべての格納済み定義が出力されます。

```

public void print( ) throws org.omg.CORBA.UserException {

    //retrieve the repository as a container... as the top level container
    org.omg.CORBA.Container container =
        (Container)ic.lookup(serviceURL + "/etc/ifr");

    //All objects in the IFR are Contained, except for the Repository.
    //Retrieve the contents of the Repository, which would be all objects that
    //it contains.
    org.omg.CORBA.Contained[] contained =
        container.contents(org.omg.CORBA.DefinitionKind.dk_all, true);

    //The length is equal to the number of objects contained within the IFR
    for(int i = 0; i < contained.length; i++) {
        {
            //Each Contained object has a description.
            org.omg.CORBA.ContainedPackage.Description description =
                contained[i].describe();

            //Each object is of a certain type, which is retrieved by the value method.
            switch(contained[i].def_kind().value()) {
                case org.omg.CORBA.DefinitionKind._dk_Attribute:
                    printAttribute(org.omg.CORBA.AttributeDefHelper.narrow(contained[i]));
                    break;
                case org.omg.CORBA.DefinitionKind._dk_Constant:
                    printConstant(org.omg.CORBA.ConstantDefHelper.narrow(contained[i]));
                    break;
            }
        }
    }
}

```

```
case org.omg.CORBA.DefinitionKind._dk_Exception:
    printException(org.omg.CORBA.ExceptionDefHelper.narrow(contained[i]));
    break;
case org.omg.CORBA.DefinitionKind._dk_Interface:
    printInterface(org.omg.CORBA.InterfaceDefHelper.narrow(contained[i]));
    break;
case org.omg.CORBA.DefinitionKind._dk_Module:
    printModule(org.omg.CORBA.ModuleDefHelper.narrow(contained[i]));
    break;
case org.omg.CORBA.DefinitionKind._dk_Operation:
    printOperation(org.omg.CORBA.OperationDefHelper.narrow(contained[i]));
    break;
case org.omg.CORBA.DefinitionKind._dk_Alias:
    printAlias(org.omg.CORBA.AliasDefHelper.narrow(contained[i]));
    break;
case org.omg.CORBA.DefinitionKind._dk_Struct:
    printStruct(org.omg.CORBA.StructDefHelper.narrow(contained[i]));
    break;
case org.omg.CORBA.DefinitionKind._dk_Union:
    printUnion(org.omg.CORBA.UnionDefHelper.narrow(contained[i]));
    break;
case org.omg.CORBA.DefinitionKind._dk_Enum:
    printEnum(org.omg.CORBA.EnumDefHelper.narrow(contained[i]));
    break;
case org.omg.CORBA.DefinitionKind._dk_none:
case org.omg.CORBA.DefinitionKind._dk_all:
case org.omg.CORBA.DefinitionKind._dk_Typedef:
case org.omg.CORBA.DefinitionKind._dk_Primitive:
case org.omg.CORBA.DefinitionKind._dk_String:
case org.omg.CORBA.DefinitionKind._dk_Sequence:
case org.omg.CORBA.DefinitionKind._dk_Array:
default:
    break;
    }
}
}
```

CORBA の Tie メカニズムの使用

サーバー・オブジェクトの実装に、継承メカニズムではなく CORBA の Tie メカニズム、つまり委譲メカニズムを使用する場合に、特別な考慮事項が 1 つあります。Tie の場合は、`oracle.aurora.AuroraServices.ActivatableObject` インタフェースを実装する必要があります。このインタフェースには `_initializeAuroraObject()` という 1 つのメソッドがあります。

(Oracle8i ORB の旧リリースでは、すべてのサーバー・オブジェクトに対してこのメソッドを実装する必要があったことに注意してください。このリリースでは、実装は Tie オブジェクトに対してのみ必要です。)

Tie クラスの `_initializeAuroraObject()` の実装は、通常、次のとおりです。

```
import oracle.aurora.AuroraServices.ActivatableObject;
...
public org.omg.CORBA.Object _initializeAuroraObject () {
    return new _tie_Hello (this);
    ...
}
```

`_tie_<interface_name>` は IDL コンパイラにより生成される Tie クラスです。

また、実装するオブジェクトには、常に、パラメータのないパブリック・コンストラクタを含める必要があります。

Tie メカニズムの使用方法を示している完全な例は、CORBA の例に含まれている `tieimpl` の例を参照してください。コードは、A-22 ページの「[Tie の例](#)」も参照してください。

JDK 1.1 から Java 2 への移行

Oracle8i JVM では、ORB 実装が JDK 1.1 および Java2 と互換性のある Visibroker 3.4 に更新されています。

注意： 既存の CORBA アプリケーションをリリース 8.1.6 で使用する場
合、スタブとスケルトンは再生成する必要があります。IDL ファイルから
コードを再生成するときに、リリース 8.1.6 のツールを使用してください。

Sun Microsystems の Java 2 には、OMG CORBA 実装が含まれています。JDK 1.1 には OMG CORBA 実装は含まれていません。したがって、以前は、Inprise ライブラリをインポートして CORBA のメソッドを起動する場合、常に Visibroker 実装が起動されていました。Java 2 に含まれる実装では、次に示すように変更を加えないで CORBA メソッドを起動する場合、Sun Microsystems の CORBA 実装が起動されるため、予期しない結果が起こる可能性があります。

クライアント側で ORB を初期化する 3 つのメソッドと、Sun Microsystems の CORBA 実装を使用しないための推奨事項を次に示します。

- **JNDI lookup** — lookup メソッドの設定は、JDK 1.1 と Java 2 の場合で同一です。ただし、スタブとスケルトンは再生成する必要があります。
- **Aurora ORB インタフェース** — Aurora ORB では、ORB を初期化するためのインタフェースが提供されます。JNDI を使用しない場合、クライアントが自分のノードで ORB を初期化し、データベース内の ORB と通信します。このクラスを介して、クライアント側の Aurora ORB を使用できます。
- **CORBA ORB インタフェース** — OMG の CORBA ORB インタフェースを使用する場合は、いくつかのプロパティを設定して、正しい実装にアクセスする必要があります。クライアントで Aurora ORB を使用しない場合は、CORBA インタフェースを使用できます。ただし、コールが正しい実装に送信されるように環境を設定する必要があります。

JNDI lookup

クライアントで JNDI を使用して、サーバーに常駐する CORBA オブジェクトにアクセスしている場合は、コードを変更する必要はありません。ただし、CORBA のスタブとスケルトンは再生成する必要があります。

Aurora ORB インタフェース

クライアント環境で JDK 1.1 を使用している場合は、既存のコードを変更する必要はありません。ただし、スタブとスケルトンは再生成する必要があります。

クライアント環境が Java 2 にアップグレードされている場合は、`oracle.aurora.jndi.orb_dep.Orb.init` メソッドを介して、ORB を初期化できます。このメソッドを使用して ORB を初期化すると、1 つの ORB インスタンスのみが初期化されます。すなわち、Java 2 ORB インタフェースを使用すると、続けて `init` メソッドを 1 回起動するたびに、新しい ORB インスタンスが 1 つ戻されます。Aurora の `init` メソッドでは、1 つの ORB インスタンスが初期化されます。`init` をコールするたびに、既存の ORB インスタンスへのオブジェクト参照が戻されます。

また、Aurora ORB インタフェースでは、セッション・ベースの IIOP 接続が管理されます。

oracle.aurora.jndi.orb_dep.Orb クラス `init` メソッドにはいくつかの種類があり、そのパラメータ・リストはそれぞれ異なります。各 `init` メソッドの構文およびパラメータを次に示します。

注意： 戻されるクラスは、それぞれの `init` メソッドによって異なります。戻された `org.omg.CORBA.ORB` クラスは `com.visigenic.vbroker.orb.ORB` に安全にキャストできます。

パラメータなし

パラメータを指定せずに `ORB.init` メソッドを実行すると、次の処理が行われます。

- ORB インスタンスが存在しない場合は、ORB インスタンスが作成され、その参照が戻されます。
- ORB インスタンスが存在する場合は、その ORB 参照が戻されます。

構文

```
public com.visigenic.vbroker.orb.ORB init();
```

ORB プロパティの指定

ORB プロパティを唯一のパラメータとして指定して `ORB.init` メソッドを実行すると、次の処理が行われます。

- ORB インスタンスが存在しない場合は、プロパティの引数が考慮されて ORB インスタンスが作成され、参照が戻されます。
- ORB インスタンスが存在する場合は、その ORB 参照が戻されます。

構文

```
public org.omg.CORBA.ORB init(Properties props);
```

入力引数および ORB プロパティの指定

ORB プロパティおよび ORB コマンドライン引数を指定して `ORB.init` メソッドを実行すると、常に ORB インスタンスが作成され、その参照が戻されます。

構文

```
public org.omg.CORBA.ORB init(String[] args, Properties props);
```

パラメータ	説明
<code>Properties props</code>	ORB システム・プロパティ
<code>String[] args</code>	ORB インスタンスに渡される引数

例 5-2 Aurora ORB init メソッドの使用

次の例に、クライアントで Aurora Orb クラスを使用して、ORB をインスタンス化する方法を示します。

```
// Create the client object and publish it to the orb in the client
// Substitute Aurora's Orb.init for OMG ORB.init call
// old way: org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init ();
com.visigenic.vbroker.orb.ORB orb = oracle.aurora.jndi.orb_dep.Orb.init();
```

ユーザー名、パスワードおよびロールと ORB プロパティの指定

ORB プロパティ、ユーザー名、パスワードおよびロールをパラメータとして指定して ORB.init メソッドを実行すると、次の処理が行われます。

- ORB インスタンスが存在しない場合は、ORB インスタンスが作成され、その参照が戻されます。
- ORB インスタンスが存在する場合は、その ORB 参照が戻されます。

クライアントで JNDI を使用せずに ORB 初期化を実行し、別のクライアントから既存のオブジェクトへの参照を受け取る場合は、このメソッドを使用します。セッション内でアクティブなオブジェクトにアクセスするには、新しいクライアントは次のいずれかの方法で、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。

- SSL_CREDENTIALS が要求されている場合は、init メソッドのパラメータにユーザー名、パスワードおよびロールを指定します。提供されたオブジェクト参照に対してメソッドを起動すると、最初のメッセージでユーザー名、パスワードおよびロールが暗黙的に渡され、データベースに対するクライアントの認証が実行されます。
- SSL_LOGIN または NON_SSL_LOGIN のいずれかを介してログイン・プロトコルが要求されている場合は、最初のクライアントが、ログイン・オブジェクトと目的のオブジェクトの両方のオブジェクト参照を渡す必要があります。2 番目のクライアントは、ログイン・オブジェクトの authenticate メソッドでユーザー名、パスワードおよびロールを指定することで、自分自身を認証します。これにより、目的のオブジェクトに対して任意のメソッドを実行できます。

このメソッドは、2 番目のクライアントが、すでに確立されたセッション中でアクティブなオブジェクトを起動するためのものです。

構文

```
public org.omg.CORBA.ORB init(String un, String pw, String role,
                             boolean ssl, java.util.Properties props);
```

パラメータ	説明
String un	クライアント側認証用のユーザー名。
String pw	クライアント側認証用のパスワード。
String role	ログイン後に使用するロール。
Boolean ssl	TRUE の場合、接続用に SSL を使用可能です。FALSE の場合、非 SSL 接続が使用されます。
Properties props	ORB が使用するプロパティ。

CORBA ORB インタフェース

純粋な CORBA クライアントを実装している場合、すなわち JNDI を使用していない場合は、ORB 初期化コールの前に、次のプロパティを設定する必要があります。これらのプロパティを使用すると、Java 2 に含まれた CORBA 実装ではなく Aurora 実装に初期化コールが送信されます。これにより、目的の動作が保証されます。Visibroker で実行される動作は次のとおりです。

- ORB.init を 2 回以上起動しても、Oracle8i JVM により 1 つの ORB インスタンスのみが作成されます。これらのプロパティを設定しない場合は、ORB.init を起動するたびに新しい ORB インスタンスが作成されることに注意してください。
- セッション IIOP 接続が正しく管理されます。
- サーバーからのコールバックが正しく管理されます。

プロパティ	割当て値
org.omg.corba.ORBClass	com.visigenic.vbroker.orb
org.omg.corba.ORBSingletonClass	com.visigenic.vbroker.orb

例 5-3 OMG プロパティへの Visibroker 値の割当て

次の例に、OMG CORBA init メソッドを Visibroker 実装に送信するための、OMG プロパティの設定方法を示します。

```
System.getProperties().put("org.omg.CORBA.ORBClass",
                           "com.visigenic.vbroker.orb.ORB");
System.getProperties().put("org.omg.CORBA.ORBSingletonClass",
                           "com.visigenic.vbroker.orb.ORB");
```

また、コマンドラインでも次のようにプロパティを設定できます。

```
java -Dorg.omg.CORBA.ORBClass=com.visigenic.vbroker.orb.ORB
     -Dorg.omg.CORBA.ORBSingletonClass=com.visigenic.vbroker.orb.ORB
```

8.1.5 との下位互換性

publish など、Oracle8i で提供されているツールは、JDK 1.1 と Java 2 のいずれの環境でも使用できるように修正されています。ただし、リリース 8.1.5 のツールで生成またはロードされたコードは使用できなくなります。使用しているすべてのツールがリリース 8.1.6 であることを確認してください。この規則は CORBA のスタブとスケルトンに適用されます。すべてのスタブとスケルトンを 8.1.6 IDL コンパイラで再生成する必要があります。

アプレットからの CORBA 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。信用のおけ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 lookup を実行する方法は、初期コンテキスト内で次のプロパティを設定すること以外は他の Oracle Java クライアントの場合と同じです。

```
env.put (ServiceCtx.APPLLET_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 lookup 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;
}

```

CORBA 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、アプレットをロードする 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。各プラグインは、アプレット情報の設定に関して異なるシンタックスを持ちます。ただし、どのプラグインを使用しても HTML ページに含めるプロパティの設定は以下のとおりです。

- 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 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・ルールの関係で、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。上記の手順は、ORBdisableLocator、ORBClass および ORBSingletonClass プロパティを設定するためのものです。

次の項の例は、各プラグイン・タイプでの 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 5-4 「JDK 1.1 プラグインの HTML 定義」](#)
- [例 5-5 「Java 2 プラグインの HTML 定義」](#)
- [例 5-6 「JInitiator プラグインの HTML 定義」](#)

例 5-4 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 5-5 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">
</EMBED></COMMENT>
</NOEMBED></EMBED>
</OBJECT>

</center>
<hr>
</pre>
```

例 5-6 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>
```

非 Oracle ORB との相互運用性

他のベンダーの ORB を使用するクライアントから、Oracle8i との相互運用ができます。そのためには、ベンダーは、セッション・ベースの接続、拡張 CosNaming 機能およびログイン・プロトコルなど、Oracle8i で ORB をデータベースと統合するのに使用された機能を提供する必要があります。この機能を提供するために、ORB ベンダーはオラクル社の製品管理部門と協働して、ライブラリを提供する必要があります。

クライアント側の機能はすべて、`aurora_client.jar` にパッケージ化されています。この JAR ファイルは、利用する ORB ベンダーとの相互運用のために、次の 2 つの JAR ファイルに分割されます。

- `aurora_orbindep.jar` — JNDI など、ORB に依存しない機能が含まれます。
- `aurora_orbdep.jar` — セッション・ベースの通信、ログイン・プロトコル、セキュリティ・コンテキストなど、Oracle ORB に依存する機能が含まれます。

ORB ベンダーから、`aurora_orbdep.jar` ファイルの提供を受ける必要があります。したがって、ベンダーからの `aurora_orbdep.jar` ファイルと Oracle が提供する `aurora_orbindep.jar` ファイルを組み込んで、その `aurora_client.jar` を置き換えます。

注意： CLASSPATH から `aurora_client.jar` ファイルを削除しない場合は、ORB ベンダーのクラスのかわりに Oracle のクラスを使用します。

`aurora_orbdep.jar` には、次の機能が含まれます。

機能	説明
ログイン	ログイン・プロトコルでは、データベースに対してクライアントの認証を実行するための要求および応答が実行されます。詳細は、6-1 ページの「 IIOP のセキュリティ 」を参照してください。
ブートストラップ	ブート・サービスにより、CosNaming などの主要なサービスが取得されます。
拡張 CosNaming	Aurora ORB の拡張 CosNaming。最初の検索時にオブジェクトを自動的にインスタンス化します。
セッション IIOP	セッション IIOP は、1 つのクライアントが複数の IIOP セッションに同時に接続できるように実装されます。詳細は、 第 3 章「IIOP アプリケーションの構成」 を参照してください。
資格証明	資格証明認証用のセキュリティ・コンテキスト・インタセプタ。

Oracle ORB を使用する Java クライアント

Oracle が提供する ORB をクライアントで使用する場合は、次の操作を行います。

1. CLASSPATH に存在するディレクトリに `aurora_client.jar` を格納します。
2. CORBA アプリケーションをコンパイルして実行します。

非 Oracle ORB を使用する Java クライアント

Oracle 以外のベンダーの ORB をクライアントで使用する場合は、次の操作を行います。

1. CLASSPATH に存在するディレクトリに `aurora_orbindep.jar` を格納します。
2. ORB ベンダーに連絡して、そのベンダーの `aurora_orbdep.jar` を取得します。
3. CLASSPATH に存在するディレクトリに `aurora_orbdep.jar` を格納します。
4. CORBA アプリケーションをコンパイルして実行します。

注意： CLASSPATH から `aurora_client.jar` ファイルを削除しない場合は、ORB ベンダーのクラスのかわりに Oracle のクラスを使用します。

C++ クライアントの相互運用性

C++ クライアントの場合、ORB ベンダーは `aurora_client.jar` ファイルの機能を共有ライブラリの形で提供する必要があります。ベンダーは Oracle が提供する C++ ログイン・プロトコルを利用して認証を実行します。すべてのクライアントは、データベースに対して自身を認証する必要があります。認証用メソッドの 1 つは、ログイン・プロトコルを介して実行されます。

ログイン・プロトコルは Oracle 固有の設計で、ユーザー名とパスワードを提供してクライアントの認証を実行してデータベースにログインするためのものです。次に、Oracle8i に対して C++ CORBA クライアントを記述する方法の例を示します。この例では、クライアント側の ORB として Visigenics C++ ORB を使用します。

例 5-7 ログイン・プロトコルで認証を実行する C++ クライアント

次の C++ クライアントでは、クライアント側 ORB として Visigenics C++ ORB を使用します。使用する ORB の種類により、実際の実装は違ってきます。

```
#include <Login.h>
#include <oracle_orbdep.h>

// set up host, port, and SID
char *sid = NULL;
char *host = argv[1];
```

```
int port = atol(argv[2]);
if(argc == 4) sid = argv[3];

// set up username, password, and role
wchar_t *username = new wchar_t[6];
username[0] = 's';
username[1] = 'c';
username[2] = 'o';
username[3] = 't';
username[4] = 't';
username[5] = '\\0';

wchar_t *password = new wchar_t[6];
password[0] = 't';
password[1] = 'i';
password[2] = 'g';
password[3] = 'e';
password[4] = 'r';
password[5] = '\\0';

wchar_t *role = new wchar_t[1];
role[0] = '\\0';

// Get the Name service Object reference
AuroraServices::PublishingContext_ptr rootCtx = NULL;

// Contact Visibroker's boot service for initializing
rootCtx = VisiCppBootstrap::getNameService (host, port, sid);

// Get the pre-published login object reference
AuroraServices::PublishedObject_ptr loginPubObj = NULL;
AuroraServices::LoginServer_ptr serv = NULL;
CosNaming::NameComponent *nameComponent = new CosNaming::NameComponent[2];

nameComponent[0].id = (const char *) "etc";
nameComponent[0].kind = (const char *) "";
nameComponent[1].id = (const char *) "login";
nameComponent[1].kind = (const char *) "";

CosNaming::Name *name1 = new CosNaming::Name(2, 2, nameComponent, 0);

// Lookup this object in the Name service
CORBA::Object_ptr loginCorbaObj = rootCtx->resolve (*name1);

// Make sure it is a published object
loginPubObj = AuroraServices::PublishedObject::_narrow (loginCorbaObj);
```

```
// create and activate this object (non-standard call)
loginCorbaObj = loginPubObj->activate_no_helper ();
serv = AuroraServices::LoginServer::_narrow (loginCorbaObj);

// Create a client login proxy object and authenticate to the DB
oracle_orbdep *_visi = new oracle_orbdep(serv);
Login login(_visi);
boolean res = login.authenticate(username, password, role);
```

IIOP トランスポート・プロトコル

他のベンダーの ORB の使用時に、その ORB ベンダーがセッションベースの IIOP をサポートしていない場合は、標準の IIOP ポートを使用します。標準 IIOP トランスポートを使用するクライアントは、複数のセッションにはアクセスできません。

非セッションベースの IIOP リスナーを構成する手順は次のとおりです。

1. MTS_DISPATCHERS パラメータを、`oracle.aurora.server.SGiopServer` のかわりに `oracle.aurora.server.GiopServer` に設定します。

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

2. クライアント・プログラムに標準の IIOP を使用するよう指定するには、`TRANSPORT_TYPE` プロパティを、次のように `ServiceCtx.IIOP` に設定します。

```
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);
env.put("TRANSPORT_TYPE", ServiceCtx.IIOP);
Context ic = new InitialContext(env);
```

注意： 任意のコマンドライン・ツールで、`-iiop` オプションを指定すると `TRANSPORT_TYPE` プロパティを設定するのと同じ効果が得られます。クライアントでリクエストがディスパッチャに直接接続される場合は、コマンドラインでサービス名に標準 IIOP ポートも指定する必要があります。

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、証明書を指定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せん。

- 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を付与します。CORBA のクラスをロードする際の実行権付与の詳細は、『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が付与される場合があります。ディプロイメント・ディスクリプタ内で実行権を定義する方法の詳細は、『Oracle8i Enterprise JavaBeans 開発者ガイドおよびリファレンス』の「アクセス制御」を参照してください。
- 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。

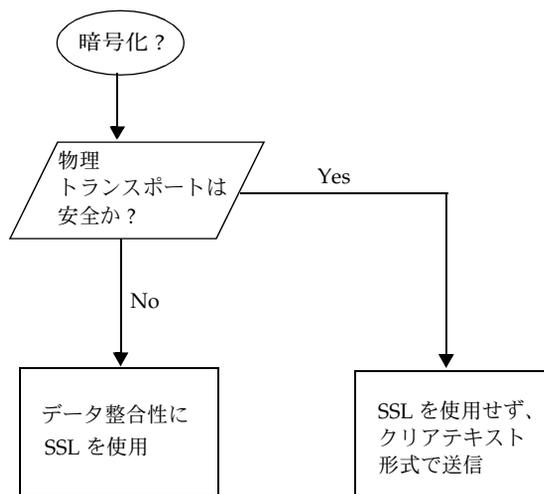
-ユーザー認可 — クライアントまたはサーバーは、提供された証明書に対して認可を実行できます。ユーザー認可を実行できるのは、クライアントまたはサーバーが証明書を提供して自分自身を認証する場合のみです。

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、Oracle[®]i 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

クライアントの設定	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)
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。	<p>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。</p> <p>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。</p>

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` を使用する方法は、`SSL_LOGIN` を使用する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	<code>SECURITY_PRINCIPAL</code>
秘密鍵を復号化するためのキー	<code>SECURITY_CREDENTIAL</code>

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 =
    "MIIECjCCAEoGAWIBAgICAmowDQYJKoZIhvcNAQEEBQAwazELMAkGA1UEBhMCMVWx" +
    "DzANBgNVBAoTBk9yYWNsZTEoMCMYGA1UECzMFRW50ZXJwcm1zZSBBChBsaWVhdGlv" +
    "biBTZXJ2aWVlc3EhMB8GA1UEAxMYRUFTTUUEgQ2VydG1maWVhdGU2VydMvYmB4X" +
    "DTk5MDgxNzE2MjIxMl0XDTAwMDIxMzE2MjIxMl0wGyUxCzAJBgNVBAYTA1VTRsw" +
    "GQYDVQQKEExJPcmFjbGUgQ29ycG9yYXRpb24xPDA6BgNVBAsUMyocIFN1Y3VyaXR5" +
    "IFRFRU1RJTtkcgQU5EIEVWQUxVQVRJT04gT05MWSB2ZXJzaW9uMiAqKjEkbmBkGA1UE" +
    "AxQScSdGVzdEB1cy5vcnFjbGUuY29tMHwwDQYJKoZIhvcNAQEEBQADawAwaAJhANG1" +
    "Kk2K7uOotI/UBYrmtE89LVRrG83Eb0/wY3xwGElkBeEUTw57a26u2M9LZAfmT91" +
    "e8Afkscq4qQW23Sjxyo4ObQK3Kth6y1NjgovBgfMu1YgtDHaSn2VEg8p58g+nwID" +
    "AQABozYwNDARBg1ghkgBhvCAQEEBAMCAAwHwYDVR0jBBgwFoAUDChwEuJfIFXD" +
    "a7tuYNO8bOw1EYwwDQYJKoZIhvcNAQEEBQADgYEARC5rWKge5trqgZ18onldinCg" +
    "Fof6D/qFT9b6Cex5JK3a2dBekg/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	<p>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。</p> <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` を取得します。
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 <code>AuroraCurrentManager</code> 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	<code>AuroraCurrentManager</code> 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。

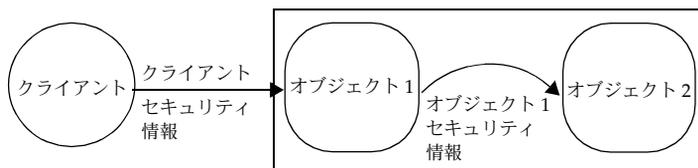
サーバー側の認証を行うには、Wallet Manager を使用して、適切な証明書を持つデータベース Wallet を設定する必要があります。Wallet の作成方法の詳細は、『Oracle8i Advanced Security 管理者ガイド』を参照してください。

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

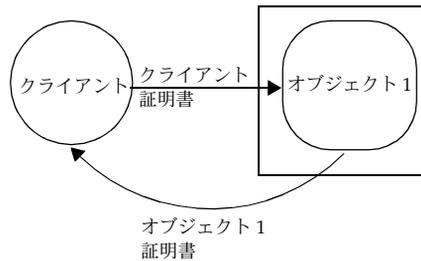


- ユーザー名 / パスワード: クライアントが認証用のユーザー名 / パスワードの組合せをデータベースに送信する場合、サーバー・オブジェクトも自身のユーザー名 / パスワードの組合せを第 2 のオブジェクトに送信する必要があります。サーバー・オブジェクトがクライアントのユーザー名 / パスワードの組合せを転送することはできず、自身の情報を提供する必要があります。ユーザー名 / パスワードの組合せは、クライアントと同じ方法で設定できます。詳細は、6-9 ページの「ユーザー名とパスワードを使用したクライアント側の認証」を参照してください。
- クライアントベース: また、クライアントが認証用の証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

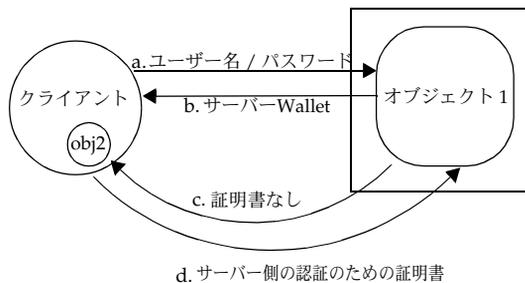
コールバックは、クライアントがサーバー・オブジェクトに、クライアント上に存在するオブジェクトのオブジェクト参照を渡す場合に発生します。図 6-3 のように、サーバー・オブジェクトはオブジェクト参照を受け取ってメソッドを起動します。これにより、サーバーから効率的にコールアウトすることができ、クライアントにあるオブジェクトがコールバックされます。詳細は、2-24 ページの「デバッグ手法」を参照してください。

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("AuroraSSLTSession", null);

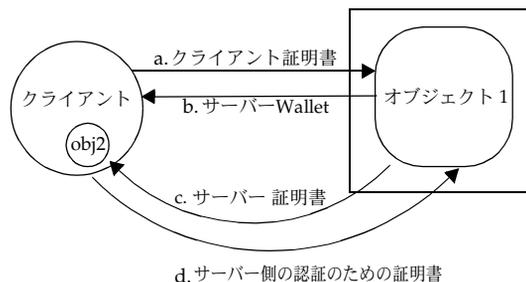
//Instantiate the client callback object
ClientImpl client = new ClientImpl ();

//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。

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、自身の証明書の設定を行います。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` を取得します。
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となります。また、サーバーは Oracle Wallet 内に Trustpoint を指定しているの、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、証明書を取得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取得することができました。現在では、発行者証明書の連鎖全体を取得します。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、証明書の連鎖を取得し、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得できます。

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。連鎖を取得した後、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**

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 情報の取得

次の例は、AuroraCurrent オブジェクトを使用して証明書情報を取得することによりピアに認可を与える方法を示します。

1. **AuroraCurrent** オブジェクトを取得するには、**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 情報を取得します。
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 xcert = cf.generateCertificate(bais);
        System.out.println(xcert);
        if(xcert instanceof X509Certificate)
        {
            X509Certificate x509Cert = (X509Certificate)xcert;
            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

この章では、CORBA アプリケーションのトランザクション管理について説明します。CORBA 開発者は、提供される次のトランザクション API のどちらか一方の使用を選択できます。

- Sun Microsystems の Java Transaction API (JTA) は、純粋な Java 環境でグローバル・トランザクションを作成するためのメソッドです。JTA は、単一または 2 フェーズ・コミット・トランザクションに使用できます。また、クライアントまたはサーバー・オブジェクトからデマーケートできます。
- Java Transaction Service (JTS) は、Oracle8i JVM と共に提供される OMG Object Transaction Service (OTS) API のサブセットのマッピングです。CORBA 開発者は、トランザクション・サービスを起動して、Java 環境または非 Java 環境で分散オブジェクトのトランザクション・プロパティを使用可能にします。JTS で使用できるのは、単一フェーズ・コミット・トランザクションのみです。

Oracle8i では、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。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ケーション
- JTA のクライアント側デマーケーション
- 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
- Java Transaction Service

- トランザクション・サービス・インタフェース
- 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 プロパティで定義されます。

- 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カー
- 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するときには、トランザクションを作成します。トランザクションが参加するデータベースは1つのみで、そのデータベースに行われたすべての更新が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参加し、トランザクションに参加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参加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。

分散オブジェクト・アプリケーションでは、トランザクションの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がどちらであるかによって、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が定義されます。詳細は、7-9 ページの「[JTA のクライアント側デマーケーション](#)」および 7-7 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。

クライアントまたはサーバー・オブジェクトでは、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中でデマーケートされます。明示的デマーケーションの詳細は、7-9 ページの「[JTA のクライアント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・オブジェクトへ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。

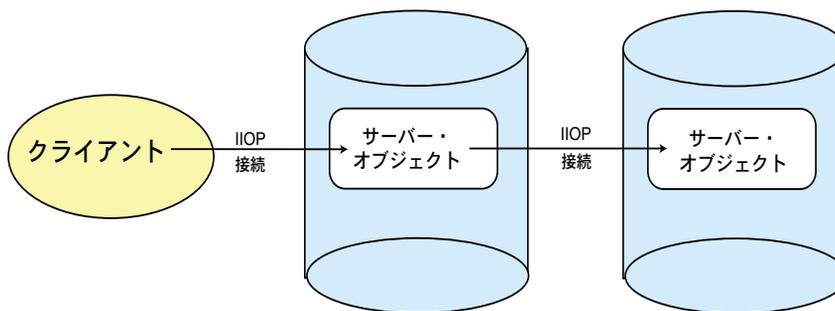
トランザクション・コンテキストの伝播

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参加する場合は、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。

図 7-1 IIOP を介したオブジェクトへの接続



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。

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 メソッドの後で、getConnection メソッドを起動する必要があります。

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、7-15 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。

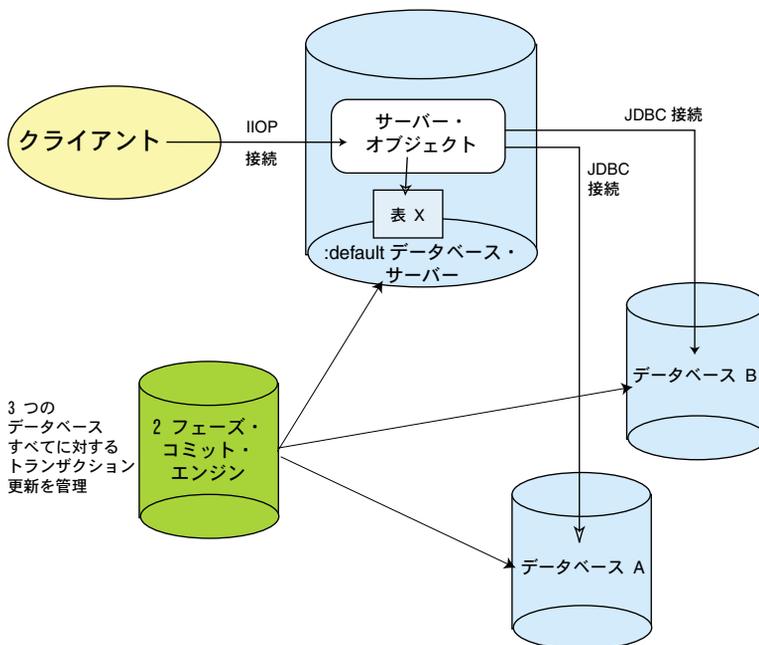
ただし、グローバル・トランザクションに複数のサーバー・オブジェクトが参加していても、データベース・リソースが 1 つのみであれば、2 フェーズ・コミット・エンジンを指定する必要はありません。2 フェーズ・コミット・エンジンは、複数データベースに対する変更を同期するためのみ必要です。データベースが 1 つのみであれば、トランザクション・マネージャで単一フェーズ・コミットを実行できます。

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、7-15 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確保されます。データベース確保の詳細は、7-5 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

サーバー側でオブジェクトまたはデータベース・リソースを取得する場合は、同一セッション中の活性化またはリモート取得を行います。

- 同一セッション中の活性化（ローカル取得）：サーバー・オブジェクトの取得はローカルでもリモートでも可能です。UserTransaction は常にローカルで取得します。DataSource オブジェクトはローカルまたはリモートで取得可能です。どのオブジェクトも、ローカルで取得する場合は同一セッション中で活性化します。ネームスペースは常にローカルであり、ローカル取得時の lookup に必要なのは JNDI 名のみです。また、環境プロパティを設定しなくても、初期コンテキストを作成できます。
- リモート取得：サーバー・オブジェクトや DataSource オブジェクトをリモート取得する場合は、クライアントの場合と同じ情報をすべて提供する必要があります。つまり、認証情報、ネームスペースの URL、jdbc_access:// 接頭辞の提供、および OracleDriver の登録が必要です。リモート取得の場合は、7-9 ページの「[JTA のクライアント側デマーケーション](#)」と同一の操作を実行します。

例 7-1 単一フェーズ・コミットのサーバー側デマーケーション

クライアントを起動する前に、まずネームスペース内で UserTransaction オブジェクトと DataSource オブジェクトをバインドしておく必要があります。

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。

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、7-15 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。

サーバー・アプリケーションの開発

次の例は、`UserTransaction` および `DataSource` オブジェクトを同一セッション中で検索するサーバー・オブジェクトを示しています。この例では、単一フェーズ・コミット・トランザクションを使用しています。

注意： この 2 フェーズ・コミットを変更するには、初期コンテキスト・コンストラクタに渡される環境内のユーザー名とパスワードを指定します。

```
ic = new InitialContext ( );

// lookup the usertransaction
UserTransaction ut = (UserTransaction)ic.lookup ("/test/myUT");
...
ut.begin ();

// Retrieve the DataSource
DataSource ds = (DataSource)ic.lookup ("/test/empDB");

// Get connection to the database through DataSource.getConnection
Connection conn = ds.getConnection ();
```

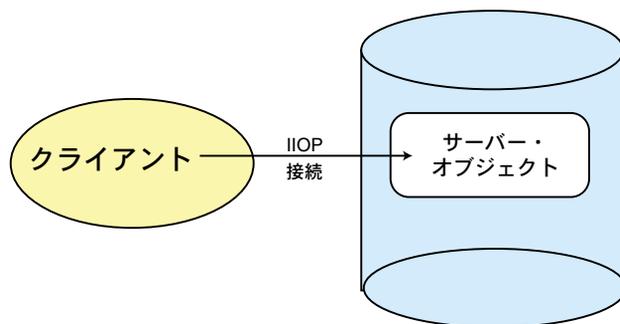
JTA のクライアント側デマーケーション

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しておく必要があります。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 を保持する `Hashtable` により環境を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。クライアントは、グローバル・トランザクションを起動してから、オブジェクトを起動します。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保持する、`Hashtable` によって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。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・オブジェクトを取得します。

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 7-2 は、トランザクション内でサーバー・オブジェクトを起動するクライアントを示しています。

例 7-2 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

`UserTransaction` オブジェクトのバインド後に、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` の登録および `jdbc_access://` 接頭辞が必要です。

```
import employee.*;
import java.sql.DriverManager;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.transaction.UserTransaction;
import java.sql.SQLException;
import javax.naming.NamingException;
import oracle.aurora.jndi.jdbc_access.jdbc_accessURLContextFactory;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

public class Client
{
    public static void main (String[] args) throws Exception
    {
        UserTransaction ut = null;
        EmployeeInfo info;
        String sessiiopURL = args [0];
        String objectName = args [1];

        //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";

        // lookup usertransaction object in the namespace
        try {
            //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, "SCOTT");
env.put (Context.SECURITY_CREDENTIALS, "TIGER");
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. Required for JDBC connection to retrieve
// UserTransaction from namespace.
DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver ());

//3. Retrieve the UserTransaction object from JNDI namespace
ut = (UserTransaction)ic.lookup ("jdbc_access://test/myUT");

//4. Start the transaction
ut.begin();

//5. Retrieve the server object reference
// lookup employee object in the namespace
Employee employee = (Employee)ic.lookup
    ("sess_iiop://myhost:1521:orcl/test/employee");

//6. Perform bean business logic.
// retrieve the info
info = employee.getEmployee ("SCOTT");
System.out.println ("Before Update: " + info.name + " " + info.salary);

// change the salary and update it
System.out.println ("Increase by 10%");
info.salary += (info.salary * 10) / 100;
employee.updateEmployee (info);

//7. End the transaction
//Commit the updated value
ut.commit ();
}

```

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起動時にオブジェクトへと伝播されます。

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す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

データベースを更新する必要がある場合、クライアントがアクセスするデータベースは、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。

クライアントから 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。

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。

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. オブジェクト参照を取得します。
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サーバー・オブジェクトを起動し、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。次の例は、7-11 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せん。

```
import employee.*;
import java.sql.DriverManager;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.transaction.UserTransaction;
import java.sql.SQLException;
import javax.naming.NamingException;
import oracle.aurora.jndi.jdbc_access.jdbc_accessURLContextFactory;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

public class Client
{
    public static void main (String[] args) throws Exception
    {
        UserTransaction ut = null;
        EmployeeInfo info;
        String sess_iiopURL = args [0];
        String objectName = args [1];
        String dsName = args [2];

        //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";

        // lookup usertransaction object in the namespace
        try {
            //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, "SCOTT");
            env.put (Context.SECURITY_CREDENTIALS, "TIGER");
            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. This is a requirement for retrieving
// the UserTransaction and DataSource objects from the namespace over
// a JDBC connection.
DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());

//3. Retrieve the UserTransaction object from JNDI namespace
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 server object reference
// lookup employee object in the namespace
Employee employee = (Employee)ic.lookup (sessiopURL + objectName);

//7 (a). Perform bean business logic.
// retrieve the info
info = employee.getEmployee ("SCOTT");
System.out.println ("Before Update: " + info.name + " " + info.salary);

// change the salary and update it
System.out.println ("Increase by 10%");
info.salary += (info.salary * 10) / 100;
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 ();
}

```

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、そのすべてのデータベースに対する変更を管理するために、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、binddds 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。

```

binddds /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。この例では、次のように実行します。

```

GRANT FORCE ANY TRANSACTION TO SCOTT

```

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 / パスワードを指定します。

次の例では、サーバー・オブジェクトが、同一セッション中での活性化により、ローカルにバインドされている UserTransaction および DataSource オブジェクトを取得しています。UserTransaction は、2 フェーズ・コミット・エンジンの URL、ユーザー名およびパスワードと共にバインドされています。DataSource オブジェクトは、すべて適切なデータベース・リンクと共にバインドされています。

```
//with the environment set, create the initial context.
InitialContext ic = new InitialContext ();
UserTransaction ut = (UserTransaction)ic.lookup ("/test/myUT");

//With the same username and password for the 2pc engine,
// lookup the local datasource and a remote database.
DataSource localDS = (DataSource)ic.lookup ("/test/localDS");

//remote lookup requires environment setup
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);
env.put (jdbc_accessURLContextFactory.CONNECTION_URL_PROP, namespaceURL);
Context ic = new InitialContext (env);

//Register a JDBC OracleDriver.
DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());
//retrieve the DataSource for the remote database
DataSource remoteDS = (DataSource)ic.lookup ("jdbc_access://test/NewYorkDS");

//retrieve connections to both local and remote databases
Connection localConn = localDS.getConnection ();
Connection remoteConn = remoteDS.getConnection ();
...
//close the connections
localConn.close();
remoteConn.close();

//end the transaction
ut.commit();
```

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 オブジェクトをバインドする場合は、次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 7-4 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.setTransactionTimeout (120);

//begin the transaction
ut.begin

//Update employee table with new employees
updateEmployees(emp, newEmp);

//end the transaction.
ut.commit ();
```

Java Transaction Service

JTS では、`TransactionService` オブジェクトから取得したトランザクション・コンテキストを介してトランザクションをデマークートします。トランザクション・コンテキストには、`begin`、`commit`、`rollback`、`suspend` および `resume` メソッドが含まれます。JTS のデメリットの 1 つは、2 フェーズ・コミット・エンジンを使用して複数データベースに対する変更を調整できないことです。JTS のメリットは、トランザクションを一時停止してから再開できることです。また、CORBA に基づいているため、アプリケーションに Java 言語も非 Java 言語も使用できます。

JTS の今回の実装では分散トランザクションをサポートしません。複数のデータベース・サーバー間に分散しているトランザクション制御は、必要な 2 フェーズ・コミット・プロトコルのサポートを含め、JTA でのみ使用可能です。

Oracle8i JVM に付属する JTS トランザクション API では、単一のリソース、つまり単一の Oracle8i データベース・セッションのみが管理されます。トランザクションは単一サーバー内でのみ終了するため、複数サーバーや単一サービスの複数のデータベース・セッションにまたがることはできません。トランザクション・コンテキストはサーバーの外部に伝播することはありません。サーバー・オブジェクトが別のサーバーへコールアウトする場合、現在のトランザクション・コンテキストと一緒に伝播することはありません。しかし、トランザクションに 1 つあるいは複数のオブジェクトを参加させることは可能です。トランザクションにそれらのオブジェクト上の 1 つまたは複数のメソッドを含めることもできます。

トランザクションをクライアント側とサーバー側のどちらでデマークートする場合も、次のステップが必要です。

1. TransactionService オブジェクトを初期化します。

Oracle8i は、任意のサーバー・オブジェクトに対し、このオブジェクトを自動的に初期化します。つまり、このオブジェクトを明示的に初期化する必要があるのは、クライアントのみです。この初期化には、AuroraTransactionService.initialize メソッドが使用されます。

2. 静的メソッド TS.getTS を介して TransactionService オブジェクトを取得します。
3. TransactionService.getCurrent メソッドを介してカレント・トランザクション・コンテキストを取得します。
4. トランザクション・コンテキスト (Current クラス) のメソッド begin、commit、rollback、rollback_only、suspend、resume を介してトランザクションを管理します。

JTS のクライアント側デマークーション

クライアント側とサーバー側のデマークーションの唯一の違いは、クライアントでは取得前に TransactionService オブジェクトを初期化する必要があることです。クライアントにより、目的のサーバーのために TransactionService オブジェクトが初期化されます。JTS で管理できるのは単一サーバー内のトランザクションのみのため、クライアントはその単一サーバー上に存在するサーバー・オブジェクトのみを起動する必要があります。また、データベースに対して実行される SQL 文も、同じサーバーに対してのみ適用する必要があります。

次の例は、クライアント側デマークーションに必要なステップを示しています。

1. TransactionService オブジェクトを初期化します。この初期化には、AuroraTransactionService.initialize メソッドが使用されます。
2. 静的メソッド TS.getTS を介して TransactionService オブジェクトを取得します。

3. `TransactionService.getCurrent` メソッドを介してカレント・トランザクション・コンテキストを取得します。
4. トランザクション・コンテキスト (Current クラス) のメソッド `begin`、`commit`、`rollback`、`rollback_only`、`suspend`、`resume` を介してトランザクションを管理します。

例 7-5 JTS のクライアント側デマーケーションの例

```
import employee.*;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.jts.client.AuroraTransactionService;
import oracle.aurora.jts.util.*;
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];

        //The environment must be setup with the correct authentication
        //and prefix information before you create 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);

        //provide the initial context and the service URL of the server
        AuroraTransactionService.initialize (ic, serviceURL);

        //Since JTS can only manage transactions on a single server, the
        //destination server object exists on the same server as the transaction
        //service. Thus, you use the same service URL to retrieve the object.
        Employee employee = (Employee)ic.lookup (serviceURL + objectName);
        EmployeeInfo info;
```

```

//Use the static method getTS to retrieve the TransactionService and the
//static method getCurrent to retrieve the current transaction context.
//Off of the Current object, you can start the transaction with the begin
//method. All three methods have been combined as follows:
TS.getTS ().getCurrent ().begin ();

//invoke a method on the retrieved server object. Since the object exists
//on the transaction server, it is included in the transaction.
info = employee.getEmployee ("SCOTT");
System.out.println (info.name + " " + " " + info.salary);
System.out.println ("Increase by 10%");
info.salary += (info.salary * 10) / 100;
employee.updateEmployee (info);
info = employee.getEmployee ("SCOTT");
System.out.println (info.name + " " + " " + info.salary);

//Finally, commit the transaction with the Current.commit method.
TS.getTS ().getCurrent ().commit (true);
}
}

```

JTS のサーバー側デマーケーション

Oracle8i は、任意のサーバー・オブジェクトのために TransactionService を初期化します。サーバーは、クライアントと同様に、同じサーバー上の他のサーバー・オブジェクトのみを起動する必要があります。SQL 文も、同じデータベースにのみ適用することが必要です。

次の例は、クライアント側デマーケーションに必要なステップを示しています。

1. 静的メソッド TS.getTS を介して TransactionService オブジェクトを取得します。
2. TransactionService.getCurrent メソッドを介してカレント・トランザクション・コンテキストを取得します。
3. トランザクション・コンテキスト (Current クラス) のメソッド begin、commit、rollback、rollback_only、suspend、resume を介してトランザクションを管理します。

例 7-6 JTS のサーバー側デマーケーションの例

```

package employeeServer;

import employee.*;
import java.sql.*;
import oracle.aurora.jts.util.*;
import org.omg.CosTransactions.*;

```

```
public class EmployeeImpl extends _EmployeeImplBase
{
    Control txn;

    public EmployeeInfo getEmployee (String name) throws SQLException {
        //When the client invokes the getEmployee method, the transaction is started
        //Retrieve the Transaction service through the static getTS method.
        //Retrieve the current transaction context through the getCurrent method.
        //And start the transaction with the Current.begin method. These have
        //been combined into one statement....
        TS.getTS ().getCurrent ().begin ();

        //Retrieve the employee information given the employee name.
        int empno = 0;
        double salary = 0.0;
        #sql { select empno, sal into :empno, :salary from emp
                where ename = :name };

        //At this point, we suspend the transaction to return the employee
        //information to the client.
        txn = TS.getTS().getCurrent().suspend();
        return new EmployeeInfo (name, empno, (float)salary);
    }

    public void updateEmployee (EmployeeInfo employee) throws SQLException {
        //After the client retrieves the employee info, it invokes the updateEmp
        //method to change any values.
        //The transaction is resumed in this method through the Current.resume,
        //which requires the Control object returned on the suspend method.
        TS.getTS().getCurrent().resume(txn);

        //update the employee's information.
        #sql { update emp set ename = :(employee.name), sal = :(employee.salary)
                where empno = :(employee.number) };

        //Once finished, complete the transaction with the Current.commit method.
        TS.getTS ().getCurrent ().commit (true);
    }
}
```

JTS の制限事項

Oracle8i のこのリリース用に提供される JTS 実装は、主にクライアント側トランザクション境界（デマーケーション）をサポートすることを目的としています。アプリケーションの設計時に注意する必要がある制限事項があります。

分散トランザクションの非サポート

JTS の今回の実装では分散トランザクションをサポートしません。複数のデータベース・サーバー間に分散しているトランザクション制御は、必要な 2 フェーズ・コミット・プロトコルのサポートを含め、JTA でのみ使用可能です。

リソース

Oracle8i JVM に付属する JTS トランザクション API では、単一のリソース、つまり Oracle8i データベース・セッションのみが管理されます。トランザクションは、1 つのサービスで複数のサーバーまたは複数のデータベース・セッションにまたがることはできません。

トランザクション・コンテキストはサーバーの外部に伝播することはありません。サーバー・オブジェクトが別のサーバーへコールアウトする場合、現在のトランザクション・コンテキストと一緒に運ばれることはありません。

しかし、トランザクション中に 1 つあるいは複数のオブジェクトを参加させることは可能です。トランザクションにそれらのオブジェクトの 1 つまたは複数のメソッドを含めることもできます。この場合、トランザクションの有効範囲は、参加しているオブジェクト全体が共有するトランザクション・コンテキストにより定義されます。たとえば、クライアントでは、単一セッションで同一サーバー上の 1 つ以上のオブジェクトを起動できます。

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と、トランザクション・サービスで `SubtransactionsUnavailable` 例外が発生します。

タイムアウト

`setTimeout()` などのトランザクション・タイムアウトをサポートする JTS のメソッドはこのリリースでは動作しません。作成したコードからそれらのメソッドを起動しても例外は発生しませんが、効果はありません。

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 OTS 実装との相互運用性はありません。

トランザクション・サービス・インタフェース

Oracle8i では、あるバージョンの JTS をサポートします。JTS は、OMG Object Transaction Service (OTS) の Java マッピングです。アプリケーション開発者が使用できるクラスが 2 種類あります。

- [TransactionService](#)
- UserTransaction インタフェース
oracle.aurora.jts.client.AuroraTransactionService が、これを実装しています。

TransactionService

TransactionService を使用して、クライアント上のトランザクション・コンテキストを初期化します。次の **import** 文を使用して Java クライアント・ソースに AuroraTransactionService パッケージを組み込みます。

```
import oracle.aurora.jts.client.AuroraTransactionService;  
import javax.jts.*;  
import oracle.aurora.jts.util.*;
```

これらのクラスは、ライブラリ・ファイル `aurora_client.jar` に含まれており、JTS を使用するすべてのソース・ファイルをコンパイルして、実行するときに CLASSPATH に含まれている必要があります。

コールできるメソッドはこのパッケージに 1 つしかありません。

```
public synchronized static void initialize(Context initialContext,  
                                           String serviceName)
```

このメソッドは、クライアント上のトランザクション・コンテキストを初期化します。パラメータは次のとおりです。

<code>initialContext</code>	JNDI Context コンストラクタにより戻される context オブジェクト。
<code>serviceName</code>	完全なサービス名。たとえば、 <code>sess_iiop://localhost:2481:ORCL。</code>

`initialize()` を使用する例を次に示します。

```
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 initialContext = new InitialContext(env);
```

```
AuroraTransactionService.initialize  
    (initialContext, "sess_iiop://localhost:2481:ORCL");
```

Java Transaction Service の使用

JTS には、クライアント側またはサーバー側オブジェクトでのトランザクション開始、トランザクションのコミットまたはロールバック、およびトランザクション・タイムアウトの設定などのユーティリティ機能の実行に使用するメソッドが含まれます。

次の項では、JTS の API について説明します。

- [必須の import 文](#)
- [Java Transaction Service のメソッド](#)
- [カレント・トランザクションのメソッド](#)

必須の import 文

JTS メソッドを使用するには、ソースに次の import 文を含めます。

```
import oracle.aurora.jts.util.TS;  
import javax.jts.util.*;  
import org.omg.CosTransactions.*;
```

oracle.aurora.jts.util パッケージは、ライブラリ・ファイル `aurora_client.jar` に含まれており、JTS を使用するすべての Java ソース用に CLASSPATH で指定されている必要があります。

TS クラス内の static メソッドを使用して、トランザクション・サービスを取得します。

Java Transaction Service のメソッド

JTS には次のメソッドが用意されています。

```
public static synchronized TransactionService getTS ()
```

1. `getTS` メソッドは、トランザクション・サービス・オブジェクトを戻します。
2. トランザクション・サービスが取得された後、static メソッド `getCurrent ()` を起動して、トランザクション・コンテキストである疑似オブジェクト `Current` を戻すことができます。
3. 最後に、`Current` 疑似 CORBA オブジェクト上でカレント・トランザクションの開始、一時停止、再開、コミットまたはロールバックを実行するメソッドを起動します。

次に、JNDI 初期コンテキストの取得から始めてクライアント上で新しいトランザクションを開始する例を示します。

```
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.jts.client.AuroraTransactionService;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;
...
Context ic = new InitialContext(env);
...
AuroraTransactionService.initialize(ic, serviceURL);
...
Employee employee = (Employee)ic.lookup (serviceURL + objectName);
EmployeeInfo info;
oracle.aurora.jts.util.TS.getTS().getCurrent().begin();
```

使用できるトランザクション・サービスがない場合、`getTS()` では `NoTransactionService` 例外が発生します。

カレント・トランザクションのメソッド

カレント・トランザクション・コンテキスト上でトランザクションを制御するためにコールできるメソッドは次のとおりです。

public void begin()

新しいトランザクションを開始します。

次の例外が発生します。

- `NoTransactionService`: トランザクション・コンテキストを初期化していない場合。
- `SubtransactionsUnavailable`: カレント・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 `begin()` を起動した場合。

初期化の詳細は、7-24 ページの「[TransactionService](#)」を参照してください。

public Control suspend()

そのセッションにおけるカレント・トランザクションを一時停止します。Control トランザクション・コンテキスト疑似オブジェクトが戻されます。後の `resume()` の起動で使用できるように、このオブジェクト参照を保存しておく必要があります。次の方法で `suspend()` を起動します。

```
org.omg.CosTransactions.Control c =
    oracle.aurora.jts.util.TS.getTS().getCurrent().suspend();
```

suspend() では、次の例外が発生する可能性があります。

- NoTransactionService: トランザクション・コンテキストを初期化していない場合。
- TransactionDoesNotExist: トランザクションがアクティブなトランザクション・コンテキストにない場合。これは、間に resume() が入らずに suspend() が前の suspend() に続く場合に発生する可能性があります。

トランザクション・コンテキストの外で suspend() を起動した場合には、NoTransactionService 例外が発生します。begin() を起動する前、または suspend() を起動した後に suspend() を起動した場合に、この例外が発生します。

```
public void resume(Control which)
```

一時停止しているトランザクションを再開します。指定されたトランザクション・コンテキストを再開するために、suspend() の後にこのメソッドを起動します。which パラメータは、同じセッションで一致する前回起動した suspend() により戻されたトランザクションの Control オブジェクトであることが必要です。たとえば、次のようになります。

```
org.omg.CosTransactions.Control c =
    oracle.aurora.jts.util.TS.getTS().getCurrent().suspend();
... // do some non-transactional work
oracle.aurora.jts.util.TS.getTS().getCurrent().resume(c);
```

resume() では、次の例外が発生することがあります。

- InvalidControl which パラメータが有効ではないか、または NULL の場合に発生します。

```
public void commit(boolean report_heuristics)
```

カレント・トランザクションをコミットします。report_heuristics パラメータを **FALSE** に設定します。

(2 フェーズ・コミットに関する付加情報を入手するには、report_heuristics パラメータを **TRUE** に設定します。このリリースの Oracle8i JVM では分散オブジェクトに対して 2 フェーズ・コミットをサポートしていないので、report_heuristics パラメータを使用しても意味はありません。これは、将来のリリースとの互換性のために含まれています。)

commit() では、次の例外が発生することがあります。

- HeuristicMixed report_heuristics が TRUE に設定され、2 フェーズ・コミットが進行中の場合。
- HeuristicHazard report_heuristics が TRUE に設定され、2 フェーズ・コミットが進行中の場合。

HeuristicMixed および HeuristicHazard 例外は、OTS 仕様の中で規定されています。アクティブなトランザクションがない場合、commit() では NoTransaction 例外が発生します。

```
public void rollback()
```

カレント・トランザクションの結果をロールバックします。

rollback() を起動すると、トランザクションが終了する結果となります。したがって、rollback() の後に begin() 以外の JTS メソッドを起動すると、NoTransaction 例外が発生します。

トランザクション・コンテキスト中ではない場合、rollback() は NoTransaction 例外が発生します。

```
public void rollback_only() throws NoTransaction {
```

rollback_only() は、カレント・トランザクションへの最終的な操作をロールバックに限定します。トランザクション・コンテキスト中ではない場合、rollback_only() では NoTransaction 例外が発生します。

```
public void set_timeout(int seconds)
```

このメソッドはサポートされていないので、起動しても何も生じません。デフォルトのタイムアウト値はどのような場合でも 60 秒です。

```
public Status get_status()
```

いつでも get_status() をコールして、カレント・トランザクションの状態を検出できます。戻り値は次のとおりです。

- javax.transaction.Status.StatusActive
- javax.transaction.Status.StatusMarkedRollback
- javax.transaction.Status.StatusNoTransaction

状態を表す int 値の完全なセットが javax.transaction.Status で定義されています。

```
public String get_transaction_name() {
```

get_transaction_name() を起動すると、String として戻されたトランザクションの名前が戻されます。begin() の前、rollback() の後、またはトランザクション・コンテキストの外でこのメソッドを起動すると、NULL 文字列が戻されます。

JTS の詳細情報

Java Transaction Service の詳細は、次の URL を参照してください。

<http://java.sun.com/products/jts/index.html>

Sun JTS 仕様は、次の URL を参照してください。

<http://java.sun.com/products/jta/index.html>

OTS 仕様は、CORBA サービス仕様の一部です。第 10 章（章別にダウンロード可能）に、OTS 仕様が記載されています。次の URL を参照してください。

<http://www.omg.org/library/csindx.html>

JDBC の制限事項

CORBA サーバー・オブジェクトで 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のに、JDBC connection オブジェクト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してください。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。オブジェクトは、`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してください。

コード例 : CORBA

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](#)
- [IFR の例](#)
- [コールバックの例](#)
- [Tie の例](#)
- [純粋な CORBA クライアント](#)
- [JTA の例](#)
- [JTS トランザクションの例](#)
- [SSL の例](#)
- [セッションの例](#)
- [アプレットの例](#)

基本例

次の銀行の例は、単純な CORBA アプリケーションのデモを示しています。README、IDL、サーバー・コードおよびクライアント・コードが含まれています。Make ファイルについては、demo/corba/basic ディレクトリを参照してください。

README

bank demonstrates:

This is an Oracle8i-compatible version of the VisiBroker Bank example. The major differences from the Vb example are:

- (1) There is no server main loop. For Oracle8i the "wait-for-activation" loop is part of the IIOP presentation (MTS server).
- (2) `_boa.connect(object)` is used instead of the less portable `_boa_obj_is_ready(object)` in the server object implementation to register the new Account objects.
- (3) The client program contains the code necessary to lookup the AccountManager object (published under /test/myBank) and activate it, and to authenticate the client to the server. (Note that object activation and authentication, via `NON_SSL_LOGIN`, happen "under the covers" so to speak on the `lookup()` method invocation.)
- (4) There is also a tie implementation of this example, with the server being `AccountManagerImplTie.java`.

Bank.IDL

```
// Bank.idl

module Bank {
    interface Account {
        float balance();
    };
    interface AccountManager {
        Account open(in string name);
    };
};
```

サーバー

サーバー・コードは、次のように実装されています。

AccountManagerImpl.java

```
package bankServer;

import java.util.*;

public class AccountManagerImpl
    extends Bank._AccountManagerImplBase {

    public synchronized Bank.Account open(String name) {

        // Lookup the account in the account dictionary.
        Bank.Account account = (Bank.Account) _accounts.get(name);

        // If there was no account in the dictionary, create one.
        if(account == null) {

            // Make up the account's balance, between 0 and 1000 dollars.
            float balance = Math.abs(_random.nextInt()) % 100000 / 100f;

            // Create the account implementation, given the balance.
            account = new AccountImpl(balance);

            _orb().connect(account);

            // Print out the new account.
            // This just goes to the system trace file for Oracle 8i.
            System.out.println("Created " + name + "'s account: " + account);

            // Save the account in the account dictionary.
            _accounts.put(name, account);
        }
        // Return the account.
        return account;
    }

    private Dictionary _accounts = new Hashtable();
    private Random _random = new Random();
}
```

AccountImpl.java

```
package bankServer;

public class AccountImpl extends Bank._AccountImplBase {
    public AccountImpl(float balance) {
        _balance = balance;
    }
    public float balance() {
        return _balance;
    }
    private float _balance;
}
```

AccountManagerImplTie.java

```
package bankServer;

import java.util.*;
import oracle.aurora.AuroraServices.ActivatableObject;

public class AccountManagerImplTie
    implements Bank.AccountManagerOperations,
    ActivatableObject {

    public synchronized Bank.Account open(String name) {

        // Lookup the account in the account dictionary.
        Bank.Account account = (Bank.Account) _accounts.get(name);

        // If there was no account in the dictionary, create one.
        if(account == null) {

            // Make up the account's balance, between 0 and 1000 dollars.
            float balance = Math.abs(_random.nextInt()) % 100000 / 100f;

            // Create the account implementation, given the balance.
            account = new AccountImpl(balance);

            org.omg.CORBA.ORB.init().BOA_init().obj_is_ready(account);

            // Print out the new account.
            // This just goes to the system trace file for Oracle 8i.
            System.out.println("Created " + name + "'s account: " + account);

            // Save the account in the account dictionary.
            _accounts.put(name, account);
        }
    }
}
```

```

        // Return the account.
        return account;
    }

    public org.omg.CORBA.Object _initializeAuroraObject () {
        return new Bank._tie_AccountManager(this);
    }

    private Dictionary _accounts = new Hashtable();
    private Random _random = new Random();
}

```

Client.java

```

// Client.java opens the account through the AccountManager class and manages
// the account through the Account class */

import bankServer.*;
import Bank.*;

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 objectName user password "
+ "accountName");
            System.exit(1);
        }
        String serviceURL = args [0];
        String objectName = args [1];
        String user = args [2];
        String password = args [3];
        String name = args [4];

        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);

AccountManager manager =
    (AccountManager)ic.lookup (serviceURL + objectName);

// Request the account manager to open a named account.
Bank.Account account = manager.open(name);

// Get the balance of the account.
float balance = account.balance();

// Print out the balance.
System.out.println
    ("The balance in " + name + "'s account is $" + balance);
}
}
```

IFR の例

IFR (インタフェース・リポジトリ) の使用方法の例を次に示します。デモは、<demo/corba/basic/bankWithIFR> にあります。

Bank.IDL

```
// Bank.idl

module Bank {
    interface Account {
        float balance();
    };
    interface AccountManager {
        Account open(in string name);
    };
};
```

サーバー

サーバー・コードは、AccountManager、Account および TIE クラスに実装されます。

AccountManagerImpl.java

```
package bankServer;

import java.util.*;

public class AccountManagerImpl
```

```

extends Bank._AccountManagerImplBase {

public synchronized Bank.Account open(String name) {

    // Lookup the account in the account dictionary.
    Bank.Account account = (Bank.Account) _accounts.get(name);

    // If there was no account in the dictionary, create one.
    if(account == null) {

        // Make up the account's balance, between 0 and 1000 dollars.
        float balance = Math.abs(_random.nextInt()) % 100000 / 100f;

        // Create the account implementation, given the balance.
        account = new AccountImpl(balance);

        _orb().connect(account);

        // Print out the new account.
        // This just goes to the system trace file for Oracle 8i.
        System.out.println("Created " + name + "'s account: " + account);

        // Save the account in the account dictionary.
        _accounts.put(name, account);
    }
    // Return the account.
    return account;
}

private Dictionary _accounts = new Hashtable();
private Random _random = new Random();

}

```

AccountImpl.java

```

package bankServer;

public class AccountImpl extends Bank._AccountImplBase {
    public AccountImpl(float balance) {
        _balance = balance;
    }
    public float balance() {
        return _balance;
    }
    private float _balance;
}

```

AccountManagerImplTie.java

```
package bankServer;

import java.util.*;
import oracle.aurora.AuroraServices.ActivableObject;

public class AccountManagerImplTie
    implements Bank.AccountManagerOperations,
    ActivatableObject {

    public synchronized Bank.Account open(String name) {

        // Lookup the account in the account dictionary.
        Bank.Account account = (Bank.Account) _accounts.get(name);

        // If there was no account in the dictionary, create one.
        if(account == null) {

            // Make up the account's balance, between 0 and 1000 dollars.
            float balance = Math.abs(_random.nextInt()) % 100000 / 100f;

            // Create the account implementation, given the balance.
            account = new AccountImpl(balance);

            org.omg.CORBA.ORB.init().BOA_init().obj_is_ready(account);

            // Print out the new account.
            // This just goes to the system trace file for Oracle 8i.
            System.out.println("Created " + name + "'s account: " + account);

            // Save the account in the account dictionary.
            _accounts.put(name, account);
        }
        // Return the account.
        return account;
    }

    public org.omg.CORBA.Object _initializeAuroraObject () {
        return new Bank._tie_AccountManager(this);
    }

    private Dictionary _accounts = new Hashtable();
    private Random _random = new Random();
}
```

クライアント

クライアント・コードは、次のように実装されています。

- [Client.java](#)
- [PrintIDL.java](#)

Client.java

```
import bankServer.*;
import Bank.*;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;
import org.omg.CORBA.Repository;

public class Client
{
    public static void main (String[] args) throws Exception {
        if (args.length != 5) {
            System.out.println("usage: Client serviceURL objectName user password "
+ "accountName");
            System.exit(1);
        }
        String serviceURL = args [0];
        String objectName = args [1];
        String user = args [2];
        String password = args [3];
        String name = args [4];

        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);

        AccountManager manager =
            (AccountManager)ic.lookup (serviceURL + objectName);

        // Request the account manager to open a named account.
        Bank.Account account = manager.open (name);
    }
}
```

```
// Get the balance of the account.
float balance = account.balance();

// Print out the balance.
System.out.println
    ("The balance in " + name + "'s account is $" + balance);

System.out.println("Calling the implicit method get_interface()");
org.omg.CORBA.InterfaceDef intf = (org.omg.CORBA.InterfaceDef)
    account._get_interface_def();
System.out.println("intf = " + intf.name());

System.out.println("Now explicitly looking up for IFR and printing the");
System.out.println("whole repository");
System.out.println("");

Repository rep = (Repository)ic.lookup(serviceURL + "/etc/ifr");

new PrintIDL(org.omg.CORBA.ORB.init()) .print(rep);

}
}
```

PrintIDL.java

```
import java.io.PrintStream;
import java.util.Vector;
import java.io.DataInputStream;
import org.omg.CORBA.Repository;

public class PrintIDL
{
    private static org.omg.CORBA.ORB _orb;
    private static PrintStream _out = System.out;
    private static int _indent;

    public PrintIDL (org.omg.CORBA.ORB orb) {
        _orb = orb;
    }
    private void println(Object o) {
        for(int i = 0; i < _indent; i++) {
            _out.print("  ");
        }
        _out.println(o);
    }
}
```

```

private String toIdl(org.omg.CORBA.IDLType idlType) {
    org.omg.CORBA.Contained contained =
org.omg.CORBA.ContainedHelper.narrow(idlType);
    return contained == null ?
        idlType.type().toString() :
        contained.absolute_name();
}

public void print(org.omg.CORBA.Container container) throws
org.omg.CORBA.UserException {
    org.omg.CORBA.Contained[] contained =
        container.contents(org.omg.CORBA.DefinitionKind.dk_all, true);
    for(int i = 0; i < contained.length; i++) {
        {
            org.omg.CORBA.ContainedPackage.Description description =
                contained[i].describe();
            org.omg.CORBA.portable.OutputStream output =
                _orb.create_output_stream();
            org.omg.CORBA.ContainedPackage.DescriptionHelper.write(output,
                description);
            org.omg.CORBA.portable.InputStream input =
                output.create_input_stream();
            org.omg.CORBA.ContainedPackage.Description description2 =
                org.omg.CORBA.ContainedPackage.DescriptionHelper.read(input);
            org.omg.CORBA.Any any1 = _orb.create_any();
            org.omg.CORBA.ContainedPackage.DescriptionHelper.insert(any1,
                description);
            org.omg.CORBA.Any any2 = _orb.create_any();
            org.omg.CORBA.ContainedPackage.DescriptionHelper.insert(any2,
                description2);
            if(!any1.equals(any1) ||
                !any1.equals(any2) ||
                !any2.equals(any2) ||
                !any2.equals(any1)) {
                System.out.println("\n*** The descriptions were not equal (1) *** \n");
            }
            org.omg.CORBA.ContainedPackage.Description description3 =
                org.omg.CORBA.ContainedPackage.DescriptionHelper.extract(any2);
            if(description.kind != description2.kind ||
                !description.value.equals(description3.value)) {
                System.out.println("\n*** The descriptions were not equal (2) *** \n");
            }
        }
        switch(contained[i].def_kind().value()) {
        case org.omg.CORBA.DefinitionKind._dk_Attribute:
            printAttribute(org.omg.CORBA.AttributeDefHelper.narrow(contained[i]));
            break;

```

```
        case org.omg.CORBA.DefinitionKind._dk_Constant:
            printConstant(org.omg.CORBA.ConstantDefHelper.narrow(contained[i]));
            break;
        case org.omg.CORBA.DefinitionKind._dk_Exception:
            printException(org.omg.CORBA.ExceptionDefHelper.narrow(contained[i]));
            break;
        case org.omg.CORBA.DefinitionKind._dk_Interface:
            printInterface(org.omg.CORBA.InterfaceDefHelper.narrow(contained[i]));
            break;
        case org.omg.CORBA.DefinitionKind._dk_Module:
            printModule(org.omg.CORBA.ModuleDefHelper.narrow(contained[i]));
            break;
        case org.omg.CORBA.DefinitionKind._dk_Operation:
            printOperation(org.omg.CORBA.OperationDefHelper.narrow(contained[i]));
            break;
        case org.omg.CORBA.DefinitionKind._dk_Alias:
            printAlias(org.omg.CORBA.AliasDefHelper.narrow(contained[i]));
            break;
        case org.omg.CORBA.DefinitionKind._dk_Struct:
            printStruct(org.omg.CORBA.StructDefHelper.narrow(contained[i]));
            break;
        case org.omg.CORBA.DefinitionKind._dk_Union:
            printUnion(org.omg.CORBA.UnionDefHelper.narrow(contained[i]));
            break;
        case org.omg.CORBA.DefinitionKind._dk_Enum:
            printEnum(org.omg.CORBA.EnumDefHelper.narrow(contained[i]));
            break;
        case org.omg.CORBA.DefinitionKind._dk_none:
        case org.omg.CORBA.DefinitionKind._dk_all:
        case org.omg.CORBA.DefinitionKind._dk_Typedef:
        case org.omg.CORBA.DefinitionKind._dk_Primitive:
        case org.omg.CORBA.DefinitionKind._dk_String:
        case org.omg.CORBA.DefinitionKind._dk_Sequence:
        case org.omg.CORBA.DefinitionKind._dk_Array:
        default:
            break;
    }
}
}

private void printConstant(org.omg.CORBA.ConstantDef def) throws
    org.omg.CORBA.UserException {
    println("const " + toIdl(def.type_def()) + " " + def.name() + " = " +
        def.value() + ";");
}
}
```

```
private void printStruct(org.omg.CORBA.StructDef def) throws
    org.omg.CORBA.UserException {
    println("struct " + def.name() + " {"");
    _indent++;
    org.omg.CORBA.StructMember[] members = def.members();
    for(int j = 0; j < members.length; j++) {
        println(toIdl(members[j].type_def) + " " + members[j].name + ";");
    }
    _indent--;
    println("};");
}

private void printUnion(org.omg.CORBA.UnionDef def) throws
org.omg.CORBA.UserException {
    println("union " + def.name() + " switch(" + toIdl(def.discriminator_type_def())
+ ") {"");
    org.omg.CORBA.UnionMember[] members = def.members();
    int default_index = def.type().default_index();
    _indent++;
    for(int j = 0; j < members.length; j++) {
        if(j == default_index) {
            println("default:");
        }
        else {
            println("case " + members[j].label + ":");
        }
        _indent++;
        println(toIdl(members[j].type_def) + " " + members[j].name + ";");
        _indent--;
    }
    _indent--;
    println("};");
}

private void printException(org.omg.CORBA.ExceptionDef def) throws
    org.omg.CORBA.UserException {
    println("exception " + def.name() + " {"");
    _indent++;
    org.omg.CORBA.StructMember[] members = def.members();
    for(int j = 0; j < members.length; j++) {
        println(toIdl(members[j].type_def) + " " + members[j].name + ";");
    }
    _indent--;
    println("};");
}
```

```
private void printEnum(org.omg.CORBA.EnumDef def) throws
    org.omg.CORBA.UserException {
    org.omg.CORBA.TypeCode type = def.type();
    println("enum " + type.name() + " {"");
    _indent++;
    int count = type.member_count();
    for(int j = 0; j < count; j++) {
        println(type.member_name(j) + ((j == count - 1) ? "" : ","));
    }
    _indent--;
    println("};");
}

private void printAlias(org.omg.CORBA.AliasDef def) throws
    org.omg.CORBA.UserException {
    org.omg.CORBA.IDLType idlType = def.original_type_def();
    String arrayBounds = "";
    while(true) {
        // This is a little strange, since the syntax of typedef'ed
        // arrays is stupid.
        org.omg.CORBA.ArrayDef arrayDef =
            org.omg.CORBA.ArrayDefHelper.narrow(idlType);
        if(arrayDef == null) {
            break;
        }
        arrayBounds += "[" + arrayDef.length() + "]";
        idlType = arrayDef.element_type_def();
    }
    println("typedef " + toIdl(idlType) + " " + def.name() + arrayBounds + ";");
}

private void printAttribute(org.omg.CORBA.AttributeDef def) throws
    org.omg.CORBA.UserException {
    String readonly = def.mode() == org.omg.CORBA.AttributeMode.ATTR_READONLY ?
        "readonly " : "";
    println(readonly + "attribute " + toIdl(def.type_def()) + " " + def.name() +
        ";");
}

private void printOperation(org.omg.CORBA.OperationDef def) throws
    org.omg.CORBA.UserException {
    String oneway = def.mode() == org.omg.CORBA.OperationMode.OP_ONeway ?
        "oneway " : "";
    println(oneway + toIdl(def.result_def()) + " " + def.name() + "(");
    _indent++;
    org.omg.CORBA.ParameterDescription[] parameters = def.params();
    for(int k = 0; k < parameters.length; k++) {
```

```

String[] mode = { "in", "out", "inout" };
String comma = k == parameters.length - 1 ? "" : ",";
println(mode[parameters[k].mode.value()] + " " +
        toIdl(parameters[k].type_def) + " " +
        parameters[k].name + comma);
}
_indent--;
org.omg.CORBA.ExceptionDef[] exceptions = def.exceptions();
if(exceptions.length > 0) {
    println(" raises (");
    _indent++;
    for(int k = 0; k < exceptions.length; k++) {
        String comma = k == exceptions.length - 1 ? "" : ",";
        println(exceptions[k].absolute_name() + comma);
    }
    _indent--;
}
println(");");
}

private void printInterface(org.omg.CORBA.InterfaceDef idef) throws
    org.omg.CORBA.UserException {
    String superList = "";
    {
        org.omg.CORBA.InterfaceDef[] base_interfaces = idef.base_interfaces();
        if(base_interfaces.length > 0) {
            superList += " :";
            for(int j = 0; j < base_interfaces.length; j++) {
                String comma = j == base_interfaces.length - 1 ? "" : ",";
                superList += " " + base_interfaces[j].absolute_name() + comma;
            }
        }
    }
    println("interface " + idef.name() + superList + " {");
    _indent++;
    print(idef);
    _indent--;
    println("};");
}

private void printModule(org.omg.CORBA.ModuleDef def) throws
    org.omg.CORBA.UserException {
    println("module " + def.name() + " {");
    _indent++;
    print(def);
}

```

```
        _indent--;  
        println("}");  
    }  
}
```

コールバックの例

コールバックの例は、`demo/examples/corba/basic/callback`にあります。

README

Overview

=====

callback shows a CORBA server object that calls back to the client-side object. It works by activating a new object in the client-side ORB, using the Basic Object Adapter (BOA), and `boa.obj_is_ready()`, and sending a reference to that object to the CORBA server object.

Source files

=====

client.idl

The CORBA IDL that defines the client-side object, that will be called from the server.

```
interface Client  
    wstring helloBack()
```

server.idl

The CORBA IDL that defines the server-side object, that will be called from the client, and that will in turn call back to the client.

```
interface Server  
    wstring hello (in client::Client object)
```

Since the object is registered on the client side, and is not published in the database, to perform a callback the server object must have a reference to the client-side object. In this example, the server is called with a reference to the object that has been

registered with the client-side Basic Object Adapter (BOA) as a parameter.

Client.java

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 classes11.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 CORBA 'Server' object to find and activate it
- starts up the ORB on the client system (ORB.init())
- gets the basic object adapter object (BOA)
- instantiates a new client callback object (new ClientImpl()), and registers it with the object adapter (boa.obj_is_ready(client))
- invokes the hello() method on the server object, passing it the reference to the client callback object

It is important to do the lookup() before initializing the ORB on the Client side: The lookup call initializes the ORB in a way that's compatible with Oracle 8i. The following org.omg.CORBA.ORB.init() call does not initialize a new ORB instance but just returns the orb that was initialized by the lookup call.

The client prints:

```
I Called back and got: Hello Client World!
```

which is the concatenation of the strings returned by the server object, and the called-back client-side object.

```
serverServer/ServerImpl.java
```

```
-----
```

This class implements the server interface. The code has one method, `hello()`, which returns its own String ("I called back and got: ") plus the String that it gets as the return from the callback to the client.

```
clientServer/ClientImpl.java
```

```
-----
```

This class implements the client interface. It has a public constructor, which is required, and a single method, `helloBack()`, which simply returns the String "Hello Client World!" to the client that called it (the server object 'server' in this case).

```
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`.

IDL ファイル

Client.IDL

```
module client {
    interface Client {
        wstring helloBack ();
    };
};
```

Server.IDL

```
#include <client.idl>

module server {
    interface Server {
        wstring hello (in client::Client object);
    };
};
```

サーバー

ServerImpl.java

```
/* $Header: ServerImpl.java 14-mar-00.13:48:31 ielayyan Exp $ */
/* Copyright (c) Oracle Corporation 2000. All Rights Reserved. */
package serverServer;

import server.*;
import client.*;
import oracle.aurora.AuroraServices.ActivatableObject;

public class ServerImpl extends _ServerImplBase implements ActivatableObject
{
    public String hello (Client client) {
        return "I Called back and got: " + client.helloBack ();
    }

    public org.omg.CORBA.Object _initializeAuroraObject () {
        return this;
    }
}
```

クライアント

クライアントはサーバー・オブジェクトを起動し、サーバー・オブジェクトはクライアント側の他のオブジェクトへとコールバックします。開始側クライアントは、[Client.java](#) に実装されます。クライアント側コールバック・オブジェクトは、[ClientImpl.java](#) に実装されます。

Client.java

```
import server.*;
import client.*;
import clientServer.*;

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);

    // Get the server object before preparing the client object
    // You have to do it in that order to get the ORB initialized correctly
    Server server = (Server)ic.lookup (serviceURL + objectName);

    // Create the client object and publish it to the orb in the client
    //org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init ();
    com.visigenic.vbroker.orb.ORB orb = oracle.aurora.jndi.orb_dep.Orb.init ();
    org.omg.CORBA.BOA boa = orb.BOA_init ();
    ClientImpl client = new ClientImpl ();
    boa.obj_is_ready (client);

    // Pass the client to the server that will call us back
    System.out.println (server.hello (client));
}
}
```

ClientImpl.java

```
/* $Header: ClientImpl.java 14-mar-00.13:48:25 ielayyan Exp $ */
/* Copyright (c) Oracle Corporation 2000. All Rights Reserved. */
package clientServer;

import client.*;
import oracle.aurora.AuroraServices.ActivableObject;

public class ClientImpl extends _ClientImplBase implements ActivableObject
{
    public String helloBack () {
        return "Hello Client World!";
    }
}
```

```
public org.omg.CORBA.Object _initializeAuroraObject () {
    return this;
}
}
```

Tie の例

この例は、Tie メカニズムの使用方を示しています。

README

Overview
=====

This is a CORBA TIE (delegation) implementation of the helloworld example. See the readme for that example for more information. It uses the `_initializeAuroraObject()` method to return a class delegate, rather than the object itself.

Source files
=====

hello.idl

(See the helloworld example readme file.)

Client.java

(See the helloworld example readme file.)

helloServer/HelloImpl.java

Implements the IDL-specified Hello interface. The interface has one method, `helloWorld()`, that returns a String to the caller.

Note that the class definition *implements* the IDL-generated HelloOperations interface, rather than extending `_HelloImplBase`, as in the helloworld example.

The class also implements the Aurora ActivateableObject interface. ActivateableObject has only one method: `_initializeAuroraObject()`, which returns the class to be activated by the BOA.

This class performs no database access.

Client-side output

=====

The client prints the returned String "Hello World!" and then exits immediately.

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.

Hello.IDL

```
module hello {
    interface Hello {
        wstring helloWorld ();
    };
};
```

サーバー・コード - HelloImpl.java

```
package helloServer;

import hello.*;
import oracle.aurora.AuroraServices.ActivatableObject;

public class HelloImpl implements HelloOperations, ActivatableObject
{
    public String helloWorld () {
        return "Hello World!";
    }

    public org.omg.CORBA.Object _initializeAuroraObject () {
        return new _tie_Hello (this);
    }
}
```

Client.java

```
import hello.Hello;

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);

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

純粋な CORBA クライアント

この例は、CORBA ネーミング・サービスを使用して、JNDI のかわりにオブジェクトを取得しています。

README

Overview

=====

This example is a variant of the VisiBroker for Java "bank" example, which simply creates and publishes a factory CORBA object AccountManager that generates Account objects with some random balance. The Account object has a method, balance(), that returns the account "balance".

This example differs from the other basic CORBA examples in this set in that it does not use JNDI to lookup and activate the published

object. It uses the CosNaming name service instead, along with the classes that are part of the oracle.aurora.AuroraServices package.

Source files

=====

bank.idl

The CORBA IDL for the example. Defines a single interface Bank with two interfaces: Account and AccountManager. The AccountManager has a single method, open(), that creates an Account object, and returns it to the caller. The Account class has a single method, balance(), that returns the random balance in the account.

bankServer.AccountManagerImpl.java

The Java code that implements the AccountManager class.

bankServer.AccountImpl.java

The Java code that implements the Account class.

Client.java

You invoke the client program from a command prompt, and pass it five arguments. For example:

```
% java -classpath LIBS Client localhost 2481 ORCL 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/vbjobr.jar  
$ORACLE_HOME/lib/vbjapp.jar  
$JAVA_HOME/lib/classes.zip
```

(Note: for NT users, the environment variables might be %ORACLE_HOME% and %JAVA_HOME%.)

The client program looks up and accesses the published AccountManager using the following steps:

- (1) Get argument values from the invocation that set:
 - (a) the hostname of the server machine
 - (b) the GIOP listener port on that server
 - (c) the database SID for the Oracle server
 - (d) the username in the instance
 - (e) the password

- (2) Uses the standard `resolve_initial_references()` method on the ORB to obtain the NameService of 8i. There are various properties that control how this initial name service is obtained, as explained in the doc and in the example code. The Oracle name service object is an instance of a PublishingContext, which is an Oracle-specific extension to the CosNaming NamingContext interface. But you can resolve arbitrary references to any service also using `resolve_initial_references()`.

(The PublishingContext class, along with the other Oracle-specific classes used in this example, is documented in the JavaDoc that accompanies this EJB/CORBA product.)

(3) Gets and uses a server login object, which is published under the standard name `/etc/login` in all Java-enabled Oracle8i databases. This is done in the following steps:

- (a) Set the `/etc` directory and the login object name as members of a CosNaming NameComponent array.
- (b) Using this array, resolve the component as a Java Object.
- (c) Narrow it to be a published object type.
- (d) Activate and get the object using the Oracle-specific `activate_no_helper()` method.
- (e) Narrow it to an Oracle LoginServer object.
- (f) Create a client login proxy object.
- (g) Authenticate the client using the login object.

(4) Lookup and activate an AccountManager class, which is published (by the Makefile) as `/test/bank`. The steps are the same as those for the login object.

(You can appreciate now how much the Oracle JNDI `lookup()` method is doing for you, as it performs steps (2), (3), and (4) in one invocation.)

(5) Get a new Account object, call the `balance()` method on it to get the "balance", and print the value to the client console.

Output
=====

The printed output is something like:

The balance in Jack.B.Quick's account is \$786.68

The actual balance amount is a random number, and will be different each time you run this program.

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`.

Bank.IDL

```
// Bank.idl

module Bank {
    interface Account { float balance(); };
    interface AccountManager { Account open(in string name); };
};
```

サーバー・コード

AccountManagerImpl.java

```
package bankServer;

// import the idl-generated classes
import Bank.*;

import java.util.Dictionary;
import java.util.Random;
import java.util.Hashtable;

// Corba specific imports
import org.omg.CORBA.Object;

// Aurora-orb specific imports
import oracle.aurora.AuroraServices.ActivatableObject;

public class AccountManagerImpl
    extends _AccountManagerImplBase
    implements ActivatableObject
{
    private Dictionary _accounts = new Hashtable ();
    private Random _random = new Random ();
```

```
// Constructors
public AccountManagerImpl () { super (); }
public AccountManagerImpl (String name) { super (name); }

public Object _initializeAuroraObject () {
    return new AccountManagerImpl ("BankManager");
}

public synchronized Account open (String name) {
    // Lookup the account in the account dictionary.
    Account account = (Account) _accounts.get (name);

    // If there was no account in the dictionary, create one.
    if (account == null) {
        // Make up the account's balance, between 0 and 1000 dollars.
        float balance = Math.abs (_random.nextInt ()) % 100000 / 100f;

        // Create the account implementation, given the balance.
        account = new AccountImpl (balance);

        // Make the object available to the ORB.
        _orb ().connect (account);

        // Print out the new account.
        System.out.println ("Created " + name + "'s account: " + account);

        // Save the account in the account dictionary.
        _accounts.put (name, account);
    }

    // Return the account.
    return account;
}
}
```

AccountImpl.java

```
package bankServer;

import Bank.*;

public class AccountImpl extends _AccountImplBase {
    private float _balance;
```

```
public AccountImpl () { _balance = (float) 100000.00; }
public AccountImpl (float balance) { _balance = balance; }
public float balance () { return _balance; }
}
```

Client.java

```
import java.lang.Exception;

import org.omg.CORBA.Object;
import org.omg.CORBA.SystemException;
import org.omg.CosNaming.NameComponent;

import oracle.aurora.client.Login;
import oracle.aurora.AuroraServices.LoginServer;
import oracle.aurora.AuroraServices.LoginServerHelper;
import oracle.aurora.AuroraServices.PublishedObject;
import oracle.aurora.AuroraServices.PublishingContext;
import oracle.aurora.AuroraServices.PublishedObjectHelper;
import oracle.aurora.AuroraServices.PublishingContextHelper;

import Bank.Account;
import Bank.AccountManager;
import Bank.AccountManagerHelper;

public class Client {
    public static void main(String args[]) throws Exception {
        // Parse the args
        if (args.length < 4 || args.length > 5 ) {
            System.out.println ("usage: Client host port username password <sid>");
            System.exit(1);
        }
        String host = args[0];
        String port = args[1];
        String username = args[2];
        String password = args[3];
        String sid = null;
        if(args.length == 5)
            sid = args[4];

        // Declarations for an account and manager
        Account account = null;
        AccountManager manager = null;
        com.visigenic.vbroker.orb.ORB orb;
        PublishingContext rootCtx = null;
    }
}
```

```

// access the Aurora Names Service
try {
    // Initialize the ORB
    String initref;
    initref = (sid == null) ? "iioploc://" + host + ":" + port :
"iioploc://" + host + ":" + port + ":" + sid;
    System.setProperty("ORBDefaultInitRef", initref);

    /*
     * Alternatively the following individual properties can be set
     * which take precedence over the URL above
     System.setProperty("ORBBootHost", host);
     System.setProperty("ORBBootPort", port);
     if(sid != null)
System.setProperty("ORACLE_SID", sid);
     */

    /*
     * Some of the other properties that you can set
     System.setProperty("ORBNameServiceBackCompat", "false");
     System.setProperty("USE_SERVICE_NAME", "true");
     System.setProperty("ORBUseSSL", "true");
     System.setProperty("TRANSPORT_TYPE", "sess_iiop");
     */

    orb = oracle.aurora.jndi.orb_dep.Orb.init();
    // Get the Name service Object reference
    rootCtx = PublishingContextHelper.narrow(orb.resolve_initial_references(
"NameService"));
    // Get the pre-published login object reference
    PublishedObject loginPubObj = null;
    LoginServer serv = null;
    NameComponent[] nameComponent = new NameComponent[2];
    nameComponent[0] = new NameComponent ("etc", "");
    nameComponent[1] = new NameComponent ("login", "");

    // Lookup this object in the Name service
    Object loginCorbaObj = rootCtx.resolve (nameComponent);

    // Make sure it is a published object
    loginPubObj = PublishedObjectHelper.narrow (loginCorbaObj);

    // create and activate this object (non-standard call)
    loginCorbaObj = loginPubObj.activate_no_helper ();
    serv = LoginServerHelper.narrow (loginCorbaObj);

```

```

// Create a client login proxy object and authenticate to the DB
Login login = new Login (serv);
login.authenticate (username, password, null);

// Now create and get the bank object reference
PublishedObject bankPubObj = null;
nameComponent [0] = new NameComponent ("test", "");
nameComponent [1] = new NameComponent ("bank", "");

// Lookup this object in the name service
Object bankCorbaObj = rootCtx.resolve (nameComponent);

// Make sure it is a published object
bankPubObj = PublishedObjectHelper.narrow (bankCorbaObj);

// create and activate this object (non-standard call)
bankCorbaObj = bankPubObj.activate_no_helper ();
manager = AccountManagerHelper.narrow (bankCorbaObj);

account = manager.open ("Jack.B.Quick");

float balance = account.balance ();
System.out.println ("The balance in Jack.B.Quick's account is $"
                    + balance);
} catch (SystemException e) {
    System.out.println ("Caught System Exception: " + e);
    e.printStackTrace ();
} catch (Exception e) {
    System.out.println ("Caught Unknown Exception: " + e);
    e.printStackTrace ();
}
}
}
}

```

JTA の例

単一フェーズ・コミットの JTA トランザクションの例

Employee.IDL

```

module employee {
    struct EmployeeInfo {
        wstring name;
        long number;
    };
};

```

```
        double salary;
    };

    exception SQLException {
        wstring message;
    };

    interface Employee {
        void setUpDSCConnection (in wstring dsName) raises (SQLException);
        EmployeeInfo getEmployee (in wstring name) raises (SQLException);
        void updateEmployee (in EmployeeInfo name) raises (SQLException);
    };
};
```

Client.java

```
import employee.*;

import java.sql.DriverManager;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.transaction.UserTransaction;

import java.sql.SQLException;
import javax.naming.NamingException;

import oracle.aurora.jndi.jdbc_access.jdbc_accessURLContextFactory;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;

public class Client
{
    public static void main (String[] args) throws Exception
    {
        if (args.length != 7)
        {
            System.out.println ("usage: Client sessiiopURL jdbcURL objectName " +
                "user password userTxnName dataSrcName");
            System.exit (1);
        }
        String sessiiopURL = args [0];
        String jdbcURL = args [1];
        String objectName = args [2];
        String user = args [3];
        String password = args [4];
        String utName = args [5];
        String dsName = args [6];
```

```
// lookup usertransaction object in the namespace
UserTransaction ut = lookupUserTransaction (user, password,
                                           jdbcURL, utName);

// lookup employee object in the namespace
Employee employee = lookupObject (user, password, sessiioURL, objectName);
EmployeeInfo info;

// for (int ii = 0; ii < 10; ii++)
// {
// start a transaction
ut.begin ();

// set up the DS on the server
employee.setUpDSCConnection (dsName);

// retrieve the info
info = employee.getEmployee ("SCOTT");
System.out.println ("Before Update: " + info.name + " " + info.salary);

// change the salary and update it
System.out.println ("Increase by 10%");
info.salary += (info.salary * 10) / 100;
employee.updateEmployee (info);

// commit the changes
ut.commit ();

// NOTE: you can do this before the commit of the previous transaction
// (without starting a txn) then it becomes part of the first
// global transaction.
// start another transaction to retrieve the updated info
ut.begin ();

// Since, you started a new transaction, the DS needs to be
// enlisted with the 'new' transaction. Hence, setup the DS on the server
employee.setUpDSCConnection (dsName);

// try to retrieve the updated info
info = employee.getEmployee ("SCOTT");
System.out.println ("After Update: " + info.name + " " + info.salary);

// commit the second transaction
ut.commit ();
}
```

```

private static UserTransaction lookupUserTransaction (String user,
    String password,
    String jdbcURL,
    String utName)
{
    UserTransaction ut = null;
    try {
        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 (jdbc_accessURLContextFactory.CONNECTION_URL_PROP, jdbcURL);
        Context ic = new InitialContext (env);

        DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver ());

        ut = (UserTransaction)ic.lookup ("jdbc_access://" + utName);
    } catch (NamingException e) {
        e.printStackTrace ();
    } catch (SQLException e) {
        e.printStackTrace ();
    }
    return ut;
}

private static Employee lookupObject (String user, String password,
    String sessiopURL, String objectName)
{
    Employee emp = null;
    try {
        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);

        System.out.println ("Trying to lookup: " + sessiopURL + objectName);
        emp = (Employee)ic.lookup (sessiopURL + objectName);
    } catch (NamingException e) {
        e.printStackTrace ();
    }
    return emp;
}
}

```

EmployeeServer.sqlj

```
package employeeServer;

import employee.*;

import java.sql.Connection;
import java.sql.DataSource;
import java.sql.SQLException;
import java.util.Hashtable;

import javax.naming.Context;
import javax.naming.InitialContext;

import javax.naming.NamingException;

public class EmployeeImpl
    extends _EmployeeImplBase
{
    Context ic = null;
    DataSource ds = null;
    Connection conn = null;

    private void setInSessionLookupContext ()
        throws NamingException
    {
        Hashtable env = new Hashtable ();
        env.put (Context.INITIAL_CONTEXT_FACTORY,
            "oracle.aurora.namespace.InitialContextFactoryImpl");
        ic = new InitialContext (env);
        // ic = new InitialContext ();
    }

    public void setUpDSConnection (String dsName)
        throws SQLException
    {
        try {
            if (ic == null)
                setInSessionLookupContext ();

            // get a connection to the local DB
            ds = (DataSource)ic.lookup (dsName);

            // get a connectoin to the local DB
            conn = ds.getConnection ();
        } catch (NamingException e) {
            e.printStackTrace ();
        }
    }
}
```

```
        throw new SQLException ("setUpDSCConnection failed:" + e.toString ());
    } catch (SQLException e) {
        e.printStackTrace ();
        throw new SQLException ("setUpDSCConnection failed:" + e.toString ());
    }
}

public EmployeeInfo getEmployee (String name)
    throws SQLException
{
    try {
        if (conn == null)
            throw new SQLException ("getEmployee: conn is null");

        int empno = 0;
        double salary = 0.0;
        #sql { select empno, sal into :empno, :salary from emp
                where ename = :name };
        return new EmployeeInfo (name, empno, (float)salary);
    } catch (SQLException e) {
        throw new SQLException (e.getMessage ());
    }
}

public void updateEmployee (EmployeeInfo employee)
    throws SQLException
{
    if (conn == null)
        throw new SQLException ("updateEmployee: conn is null");

    try {
        #sql { update emp set ename = :(employee.name), sal = :(employee.salary)
                where empno = :(employee.number) };
    } catch (SQLException e) {
        throw new SQLException (e.getMessage ());
    }
}
}
```

2 フェーズ・コミットの JTA トランザクションの例

Employee.IDL

```
module employee {
    struct EmployeeInfo {
        wstring name;
        long number;
        double salary;
    };

    exception SQLError {
        wstring message;
    };

    interface Employee {
        void initialize (in wstring user, in wstring password,
            in wstring serviceURL, in wstring objectName,
            in wstring utName, in wstring localDSName,
            in wstring remoteDSName) raises (SQLError);

        EmployeeInfo getEmployee (in wstring empName) raises (SQLError);
        void updateEmployee (in EmployeeInfo empInfo) raises (SQLError);

        EmployeeInfo getRemoteEmpInfo (in wstring name) raises (SQLError);
        void updateRemoteEmployee (in EmployeeInfo empInfo) raises (SQLError);
    };
};
```

Client.java

```
import employee.*;

import java.sql.DriverManager;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.transaction.UserTransaction;

import java.sql.SQLException;
import javax.naming.NamingException;

import oracle.aurora.jndi.jdbc_access.jdbc_accessURLContextFactory;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
```

```
public class Client
{
    public static void main (String[] args) throws Exception
    {
        if (args.length != 7)
        {
            System.out.println ("usage: Client sessiopURL objectName user password"
                + " userTxnName localDataSrcName remoteDataSrcName");
            System.exit (1);
        }
        String sessiopURL = args [0];
        String objectName = args [1];
        String user = args [2];
        String password = args [3];
        String utName = args [4];
        String localDSName = args [5];
        String remoteDSName = args [6];

        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);

        Employee employee = null;
        EmployeeInfo info;

        try {
            employee = (Employee)ic.lookup (sessiopURL + objectName);
            employee.initialize (user, password, sessiopURL, objectName, utName,
                localDSName, remoteDSName);

            info = employee.getEmployee ("SCOTT");
            System.out.println (info.name + " " + " " + info.salary);
            System.out.print ("Increase by 10% to ");
            info.salary += (info.salary * 10) / 100;
            System.out.println (info.salary);
            employee.updateEmployee (info);
        } catch (SQLException e) {
            System.out.println (" Got SQLException: " + e.toString ());
        }
    }
}
```

サーバー

```
package employeeServer;

import employee.*;

import java.sql.Connection;
import java.sql.DataSource;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.Hashtable;
import java.sql.SQLException;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.transaction.UserTransaction;
import javax.naming.NamingException;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.transaction.xa.OracleJTADDataSource;

public class EmployeeImpl extends _EmployeeImplBase
{
    Context inSessionLookupctx = null;
    UserTransaction ut = null;
    DataSource localDS = null;
    DataSource remoteDS = null;
    Connection localConn = null;
    Connection remoteConn = null;
    String utName = null;
    String localDSName = null;
    String remoteDSName = null;
    String user = null;
    String pwd = null;
    String serviceURL = null;
    String objectName = null;
    EmployeeInfo localEmpInfo = null;

    private void setInSessionLookupContext ()
        throws NamingException
    {
        // NOTE: here we need to set env as 2-phase coord needs
        //       user/pwd to be set (branches is optional)
        Hashtable env = new Hashtable ();
        env.put (Context.SECURITY_PRINCIPAL, user);
        env.put (Context.SECURITY_CREDENTIALS, pwd);
        inSessionLookupctx = new InitialContext (env);
    }
}
```

```
}

public void initialize (String user, String password, String serviceURL,
    String objectName, String utName, String localDSName,
    String remoteDSName)
    throws SQLException
{
    try {
        // set the local variables
        this.user = user;
        this.pwd = password;
        this.objectName = objectName;
        this.utName = utName;
        this.localDSName = localDSName;
        this.remoteDSName = remoteDSName;
        this.serviceURL = serviceURL;

        // se up a ctx to lookup the local/in-session objects
        if (inSessionLookupctx == null)
            setInSessionLookupContext ();

        // lookup the usertransaction
        ut = (UserTransaction)inSessionLookupctx.lookup (utName);

        // get a connection to the local DB
        localDS = (OracleJTADDataSource)inSessionLookupctx.lookup (localDSName);

        // get a connection to the local DB
        remoteDS = (OracleJTADDataSource)inSessionLookupctx.lookup (remoteDSName);
    } catch (NamingException e) {
        e.printStackTrace ();
        throw new SQLException ("setUpDSCconnection failed:" + e.toString ());
    }
}

private void getConnections ()
    throws SQLException
{
    if (localDS == null)
        throw new SQLException ("local DataSource is NOT set correctly");
    if (remoteDS == null)
        throw new SQLException ("remote DataSource is NOT set correctly");
    if (user == null || pwd == null)
        throw new SQLException ("user/pwd is NOT set correctly");
}
```

```
localDS.setURL ("jdbc:oracle:kprb:");
localConn = localDS.getConnection ();
System.out.println ("remoteDS.getURL: " + remoteDS.getURL ());
remoteConn = remoteDS.getConnection (user, pwd);
}

private void startTrans ()
    throws SQLException
{
    try {
        if (ut == null)
            throw new SQLException ("startTrans: userTransaction is null");

        ut.begin ();
    } catch (Exception e) {
        throw new SQLException ("startTrans failed:" + e.toString ());
    }
}

private void commitTrans ()
    throws SQLException
{
    try {
        ut.commit ();
    } catch (Exception e) {
        throw new SQLException ("commitTrans failed:" + e.toString ());
    }
}

public EmployeeInfo getLocalEmpInfo (String name)
    throws SQLException
{
    try {
        if (localConn == null)
            throw new SQLException ("getLocalEmpInfo: localConn is null");
        int empno = 0;
        double salary = 0.0;
        #sql { select empno, sal into :empno, :salary from emp
                where ename = :name };
        System.out.println (" Local (" + name + ", " + salary + ")");
        return new EmployeeInfo (name, empno, (float)salary);
    } catch (SQLException e) {
        e.printStackTrace ();
        throw new SQLException ("getRemoteEmpInfo SQLException: " + e.toString ());
    }
}
```

```
public EmployeeInfo getRemoteEmpInfo (String name)
    throws SQLException
{
    try {
        if (remoteConn== null)
            throw new SQLException ("getRemoteEmpInfo: remoteConn is null");
        int empno = 0;
        double salary = 0.0;

        PreparedStatement ps = remoteConn.prepareStatement
            ("select empno, sal from emp where ename = ?");

        ps.setString (1, name);
        ResultSet rs = ps.executeQuery ();

        while (rs.next ())
        {
            empno = rs.getInt (1);
            salary = rs.getDouble (2);
        }
        System.out.println (" Remote (" + name + ", " + salary + ")");
        return new EmployeeInfo (name, empno, (float)salary);
    } catch (SQLException e) {
        e.printStackTrace ();
        throw new SQLException ("getRemoteEmpInfo SQLException: " + e.toString ());
    }
}

public EmployeeInfo getEmployee (String name)
    throws SQLException
{
    System.out.println ("getEmployee: begin");

    try {
        this.startTrans ();
        // get a connection to the local and remote DB
        this.getConnections ();
    } catch (SQLException e) {
        e.printStackTrace ();
        throw new SQLException ("getEmployee SQLException: " + e.toString ());
    }

    // get info for localEmployee = smith
    localEmpInfo = this.getLocalEmpInfo ("SMITH");
}
```

```
// get info for the remote employee
EmployeeInfo info = this.getRemoteEmpInfo (name);
System.out.println (" Remote (" + info.name + ", " + info.salary + ")");

this.commitTrans ();
System.out.println ("getEmployee: end");
return info;
}

public void updateEmployee (EmployeeInfo empInfo) throws SQLException
{
    System.out.println ("updateEmployee: begin");
    this.startTrans ();

    try {
        this.getConnections ();

        System.out.println (" Before updating: ");
        this.getLocalEmpInfo ("SMITH");
        this.getRemoteEmpInfo ("SCOTT");

        localEmpInfo.salary -= 0.1 * localEmpInfo.salary;
        this.updateLocalEmployee (localEmpInfo);

        System.out.println (" calling to update " + empInfo.name + " salary");
        updateRemoteEmployee (empInfo);

        System.out.println ("updateEmployee: After updating: ");
        this.getLocalEmpInfo ("SMITH");
        this.getRemoteEmpInfo ("SCOTT");
    } catch (SQLException e) {
        e.printStackTrace ();
        throw new SQLException ("updateEmployee SQLException: " + e.toString ());
    }

    this.commitTrans ();
    System.out.println ("updateEmployee: end");
}

public void updateLocalEmployee (EmployeeInfo empInfo)
    throws SQLException
{
    System.out.println ("updateLocalEmployee: begin");
    try {
        this.getLocalEmpInfo ("SMITH");
```

```
#sql { update emp set ename = :(empInfo.name), sal = :(empInfo.salary)
        where empno = :(empInfo.number) };

System.out.println (" after updating " + empInfo.name + " salary");
this.getLocalEmpInfo (empInfo.name);
} catch (SQLException e) {
    e.printStackTrace ();
    throw new SQLError (e.toString ());
}

System.out.println ("updateLocalEmployee: end");
}

public void updateRemoteEmployee (EmployeeInfo empInfo)
    throws SQLError
{
    System.out.println ("updateRemoteEmployee: begin");

    try {
        PreparedStatement ps = remoteConn.prepareStatement
            ("update emp set ename = ?, sal = ? where empno = ?");
        ps.setString (1, empInfo.name);
        ps.setDouble (2, empInfo.salary);
        ps.setInt (3, empInfo.number);
        ps.executeUpdate ();

        System.out.println (" after updating " + empInfo.name + " salary");
        this.getRemoteEmpInfo (empInfo.name);
    } catch (SQLException e) {
        e.printStackTrace ();
        throw new SQLError (e.toString ());
    }

    System.out.println ("updateRemoteEmployee: end");
}
}
```

JTS トランザクションの例

README

Overview
=====

The serversideJTS example shows how to do transaction management for

CORBA server objects from the server object, using the XA JTS methods.

Compare this example with the clientside example, in which all transaction management is done on the client.

This example also shows a server object that uses SQLJ in its methods.

Source files

=====

employee.idl

The CORBA IDL for the example. Defines:

An EmployeeInfo struct

A SQLException exception

An Employee interface, with

```
EmployeeInfo getEmployee(in wstring name)
EmployeeInfo getEmployeeForUpdate(in wstring name)
void updateEmployee(in EmployeeInfo name)
```

The SQLException exception is used so that SQLException messages can be passed back to the client.

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 server object 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/myEmployee 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 is almost exactly the same as the code in
../clientside/Client.java, but without the JTS transaction calls.

The client code performs the following steps:

- gets the arguments passed on the command line
- creates a new JNDI Context (InitialContext())
- initializes the Aurora transaction service
- looks up the myEmployee CORBA published object on the server
(this step also authenticates the client using NON_SSL_LOGIN and
activates the server object)
- gets and prints information about the employee SCOTT
- decreases SCOTT's salary by 10%
- updates the EMP table with the new salary by calling the updateEmployee()
method on the employee object
- gets and prints the new information

The printed output is:

```
Beginning salary = 3000.0
Decrease by 10%
Final Salary = 2700.0
```

Note that the starting value is taken from the EMP table when the
example starts to run, so you may see a different salary amount. The new
salary amount is written back to the database, and will be used as the
new starting amount if you run this example again.

employeeServer/EmployeeImpl.sqlj

Implements the Employee interface. This file implements the three
methods specified in the IDL: getEmployee(), getEmployeeForUpdate(),
and updateEmployee(), using SQLJ for ease of DML coding.

EmployeeImpl also adds two private methods, `commitTrans()` and `startTrans()`, that perform XA JTS transaction management from the server.

Note that on the server there is no need to call `AuroraTransactionService.initialize()` to initialize the transaction manager. This is done automatically by the server ORB.

If the SQLJ code throws a `SQLException`, it is caught, and a CORBA-defined `SQLException` is thrown. This in turn would be propagated back to the client, where it is handled.

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`.

Employee.IDL

```
module employee {
    struct EmployeeInfo {
        wstring name;
        long number;
        double salary;
    };

    exception SQLException {
        wstring message;
    };

    interface Employee {
        EmployeeInfo getEmployee (in wstring name) raises (SQLException);
        EmployeeInfo getEmployeeForUpdate (in wstring name) raises (SQLException);
        void updateEmployee (in EmployeeInfo name) raises (SQLException);
    };
};
```

Client.java

```
import employee.*;

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];

    // get the handle to the InitialContext
    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);

    // This is using Server-side TX services, specifically, JTS/XA TX:

    // get handle to the object and it's info
    Employee employee = (Employee)ic.lookup (serviceURL + objectName);

    // get the info about a specific employee
    EmployeeInfo info = employee.getEmployee ("SCOTT");
    System.out.println ("Beginning salary = " + info.salary);
    System.out.println ("Decrease by 10%");
    // do work on the object or it's info
    info.salary -= (info.salary * 10) / 100;

    // call update on the server-side
    employee.updateEmployee (info);

    System.out.println ("Final Salary = " + info.salary);
}
}

```

サーバー

```

package employeeServer;

import employee.*;
import java.sql.*;

import oracle.aurora.jts.util.*;
import org.omg.CosTransactions.*;

```

```
public class EmployeeImpl extends _EmployeeImplBase
{
    Control c;

    private void startTrans () throws SQLException {
        try {
            TS.getTS ().getCurrent ().begin ();
        } catch (Exception e) {
            throw new SQLException ("begin failed:" + e);
        }
    }

    private void commitTrans () throws SQLException {
        try {
            TS.getTS ().getCurrent ().commit (true);
        } catch (Exception e) {
            throw new SQLException ("commit failed:" + e);
        }
    }

    public EmployeeInfo getEmployee (String name) throws SQLException {
        try {
            startTrans ();

            int empno = 0;
            double salary = 0.0;
            #sql { select empno, sal into :empno, :salary from emp
                where ename = :name };
            c = TS.getTS().getCurrent().suspend();
            return new EmployeeInfo (name, empno, (float)salary);
        } catch (SQLException e) {
            throw new SQLException (e.getMessage ());
        } catch (Exception e) {
            throw new SQLException (e.getMessage());
        }
    }

    public EmployeeInfo getEmployeeForUpdate (String name) throws SQLException {
        try {
            startTrans ();

            int empno = 0;
            double salary = 0.0;
            #sql { select empno, sal into :empno, :salary from emp
                where ename = :name for update };
            return new EmployeeInfo (name, empno, (float)salary);
        } catch (SQLException e) {
```

```

        throw new SQLException (e.getMessage ());
    }
}

public void updateEmployee (EmployeeInfo employee) throws SQLException {
    try {
        TS.getTS().getCurrent().resume(c);

        #sql { update emp set ename = :(employee.name), sal = :(employee.salary)
                where empno = :(employee.number) };
        commitTrans ();
    } catch (SQLException e) {
        throw new SQLException (e.getMessage ());
    } catch (Exception e) {
        throw new SQLException (e.getMessage ());
    }
}
}
}

```

SSL の例

クライアント側の認証

README

Overview

=====

This is a very simple CORBA example using client side ssl for login. The helloWorld server object merely returns a greeting plus the Java VM version number to the client.

The purpose of the example is to show how to use ssl client side authentication for logins.

Setup required

You need to open the encrypted wallet(ewallet.der) provided in this directory using the wallet manager tool provided by Oracle. The password is welcome12. Copy the cleartext into TNS_ADMIN directory and restart the database and listeners.

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` before restarting the database and listeners

The setup also requires creation of global user `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.

The Makefile has `loadjava` that loads all the classes into `scott`'s schema whereas the client program is executed as the user `guest`. Hence `loadjava` has a `"-grant guest"` to grant the privileges to `guest`.

Source files
=====

hello.idl

The CORBA IDL for the example. Defines a single interface `Hello` with a single method `helloWorld()`. The interface is defined in the Module named 'hello', which determines the name of the directory in which the `idl2java` compiler places the generated files.

The `helloWorld()` method returns a CORBA `wstring`, which maps to a Java `String` type:

```
module hello
  interface Hello
    wstring helloWorld()
```

Client.java

You invoke the client program from a command prompt, and pass it three arguments, the

- service URL (service ID, hostname, port, and SID if port is a listener)
- name of the published bean to lookup and instantiate
- credentials file - the B64 encoded wallet for the user. This is a generated wallet at Oracle site.

The password for the wallet is hardcoded as "welcome12"

For example:

```
% java -classpath LIBs Client sess_iiop://localhost:2222 /test/myHello cert.txt
```

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
- puts the authentication type and values into env context
- creates a new JNDI Context (InitialContext())
- looks up the published CORBA server object to find and activate it
- 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.
```

```
helloServer/HelloImpl.java
```

```
-----
```

Implements the IDL-specified Hello interface. The interface has one method, helloWorld(), that returns a String to the caller.

helloWorld() invokes System.getProperty("oracle.server.version") to get the version number of the Java VM.

This object performs no database access.

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.

Hello.IDL

```
module hello {
    interface Hello {
        wstring helloWorld ();
    };
};
```

Client.java

```
import hello.Hello;
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 != 3) {
            System.out.println("usage: Client serviceURL objectName credsFile");
            System.exit(1);
        }
        String serviceURL = args [0];
        String objectName = args [1];
        String credsFile = args [2];

        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, "welcome12");

        // 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);

        /*
        // As an alternative, you may also set the credentials individually, as
        // shown bellow.
        env.put (ServiceCtx.SECURITY_USER_CERT, testCert_base64);
        env.put (ServiceCtx.SECURITY_CA_CERT, caCert_base64);
        env.put (ServiceCtx.SECURITY_ENCRYPTED_PKEY, encryptedPrivateKey_base64);
        //System.getProperties().put ("AURORA_CLIENT_SSL_DEBUG", "true");
        */
    }
}
```

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

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

サーバー

```
package helloServer;

import hello.*;

public class HelloImpl extends _HelloImplBase {
    public String helloWorld() {
        String v = System.getProperty("oracle.server.version");
        return "Hello client, your javavm version is " + v + ".";
    }
}
```

サーバー側の認証

この例には、Trustpoint の設定が含まれています。Trustpoint を使用しない場合は、コード中の Trustpoint を設定している部分を削除します。

README

Overview
=====

This is a very simple CORBA example using server side ssl for login. The helloWorld server object merely returns a greeting plus the Java VM version number to the client.

The purpose of the example is to show how to use ssl server side authentication for logins.

Setup required

You need to open the encrypted wallet (ewallet.der) provided in this directory using the wallet manager tool provided by Oracle. The password is welcome12. Copy the cleartext into TNS_ADMIN directory and restart the database and listeners.

The encrypted wallet (ewallet.der) is only valid for Solaris platforms. For other platforms, you should generate the wallet using Oracle's own tool. And if you generate the wallet, be sure to change the trust point provided in the client file Client.java too.

The parameter `SSL_CLIENT_AUTHENTICATION` in `$TNSADMIN/sqlnet.ora` should be set to false before restarting the database and listeners.

You may also generate a wallet using Oracle tool `owmgui`. The cleartext wallet that you will be using should be binary compatible with the machine you are using to run this sample.

There is also a hard coded trustpoint within Client.java. This trust point matches with the one in the server's wallet. You may replace with your trust point if needed. But this trust point should be matching with the one in the server's wallet.

This example runs with JDK1.1 libraries. If you are using JDK 1.2 libraries, then you should comment out JDK1.1 code and uncomment JDK1.2 code.

Source files

=====

hello.idl

The CORBA IDL for the example. Defines a single interface Hello with a single method `helloWorld()`. The interface is defined in the Module named 'hello', which determines the name of the directory in which the `idl2java` compiler places the generated files.

The `helloWorld()` method returns a CORBA `wstring`, which maps to a Java `String` type:

```
module hello
  interface Hello
    wstring helloWorld()
```

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

For example:

```
% java -classpath LIBs Client sess_iiop://localhost:2222
/test/sslHelloServerAuthWithTP
```

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
- puts the authentication type and values into env context
- creates a new JNDI Context (InitialContext())
- looks up the published CORBA server object to find and activate it
- 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.
```

```
helloServer/HelloImpl.java
```

```
-----
```

Implements the IDL-specified Hello interface. The interface has one method, helloWorld(), that returns a String to the caller.

helloWorld() invokes System.getProperty("oracle.server.version") to get the version number of the Java VM.

This object performs no database access.

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.

Hello.IDL

```
module hello {
    interface Hello {
        wstring helloWorld ();
    };
};
```

Client.java

```
import hello.Hello;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;
import oracle.aurora.ssl.*;
import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.jndi.sess_iiop.SessionCtx;
import oracle.aurora.AuroraServices.LoginServer;
//import java.security.cert.*; // Needs JDK 1.2; won't compile in JDK 1.1
import javax.security.cert.*; // for JDK 1.1
import java.io.*;

public class Client
{
    private static String trustedCert =
    "MIIBNjCB4aADAgECAhEAv/poeUAh5DxtXZSkZAIunDANBgkqhkiG9w0BAQQFADAcMQswCQYDVQQG"+
    "EwJVUzENMAsGA1UEAxQEUEk9PVD AeFw05OTExMTcxODQ1NDNaFw0wMjAyMDIxODQ1NDNaMBwxCzAJ"+
    "BgNVBAYTALVIMQ0wCwYDVQQDFARST09UMFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBANEzeul7saeh"+
    "q60fGp4Ya0IZ4C2GkUFFmVxBIqqfvgXUyqifrz7ZsrnxoEEYmng+OWxhwToyk1LlGUYR4ngMgF78C"+
    "AwEAATANBgkqhkiG9w0BAQQFAANBAMzDmFK2/QAxgn085SLQ+bmYBatuji2YPVDgmMYa3ebhFgUe"+
    "I7CKLQTxFg1Y2bw71LFww0Mi9cxwrR+Lt9jhnes=";

    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("The account obj's certificate chain : ");
byte [] [] certChain = current.getPeerDERCertChain(obj);
System.out.println("length : " + certChain.length);
System.out.println("Certificates: ");

/*
// JDB 1.2 way
CertificateFactory cf = CertificateFactory.getInstance("X.509");
for(int i = 0; i < certChain.length; i++) {
    ByteArrayInputStream bais = new ByteArrayInputStream(certChain[i]);
    Certificate xcert = cf.generateCertificate(bais);
    System.out.println(xcert);
    if(xcert instanceof X509Certificate)
    {
        X509Certificate x509Cert = (X509Certificate)xcert;
        String globalUser = x509Cert.getSubjectDN().getName();
        System.out.println("DN out of the cert : " + globalUser);
    }
}
*/

// JDK 1.1 way

java.security.Security.setProperty("cert.provider.x509v1",
    "oracle.security.cert.X509CertificateImpl");

for(int i = 0; i < certChain.length; i++) {
    javax.security.cert.X509Certificate cert =
        javax.security.cert.X509Certificate.getInstance(certChain[i]);

    String globalUser = cert.getSubjectDN().getName();
    System.out.println("DN out of the cert : " + globalUser);
}

return true;
}

public static void main (String[] args) throws Exception {
    if (args.length != 2) {
        System.out.println("usage: Client serviceURL objectName");
        System.exit(1);
    }
}
```

```
String serviceURL = args [0];
String objectName = args [1];

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

// setup the trust point
env.put(ServiceCtx.SECURITY_TRUSTED_CERT, trustedCert);

Context ic = new InitialContext(env);

// Make an SSL connection to the server first. If the connection
// succeeds, then inspect the server's certificate, since we haven't
// specified a trust point.
// Get a SessionCtx that represents a database instance
ServiceCtx service = (ServiceCtx)ic.lookup (serviceURL);
SessionCtx session1 = (SessionCtx)service.createSubcontext (":session1");
// Lookup login object for the purpose of getting hold of some corba
// object needed for verifyPeerCert(). We should provide an extension
// to just getting the NS object, for this purpose.
LoginServer obj = (LoginServer) session1.activate("/etc/login");

if(!verifyPeerCert(obj))
    throw new org.omg.CORBA.COMM_FAILURE("Verification of Peer cert failed");

// Now that we trust the server, let's go ahead and do our business.
session1.login();
Hello hello = (Hello) session1.activate(objectName);
System.out.println(hello.helloWorld());
}
}
```

サーバー

```
package helloServer;

import hello.*;

public class HelloImpl extends _HelloImplBase {
    public String helloWorld() {
        String v = System.getProperty("oracle.server.version");
        return "Hello client, your javavm version is " + v + ".";
    }
}
```

セッションの例

セッションは様々な方法で管理できます。4-19 ページの「[セッション管理の使用例](#)」には、すべての方法が記載されています。ここに示す例は、単一クライアントから2つのセッションにアクセスする方法を示しています。

README

Overview

=====

Twosessions demonstrates a client that instantiates two separate sessions in the server, and calls methods on objects in each session. It also demos use of the login object for client authentication.

Compare this example to the `../examples/corba/session/clientserverserver` example, in which the client instantiates a server object, and that server object then instantiates a second server object in a different session.

Source files

=====

hello.idl

The CORBA IDL for the example. The IDL for the Hello object simply defines two methods:

```
interface Hello
    wstring helloWorld ();
    void setMessage (in wstring message);
```

which must be implemented by the `helloServer.HelloImpl.java` code.

Client.java

You invoke the client program from a command line prompt, and pass it four arguments: the service URL (service ID, hostname, and port), the name of the published object to lookup and instantiate, and a username and password that authenticate the client to the Oracle8i database server.

For example:

```
% java -classpath LIBs Client sess_iiop://localhost:2222 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 first obtains a service context in the normal way, by getting a JNDI Context object, and looking up the service context on it, using the service URL (e.g., sess_iiop://localhost:2222). The service context is then used to create new named sessions, :session1 and :session2. On each session, a login server object is instantiated, then a login client is obtained, and the authenticate() method on the login client is used to authenticate the client.

Note that this form of authentication is what happens automatically when a server object is instantiated, and the JNDI context is obtained by passing in the username, password, optional database role, and the value NON_SSL_LOGIN in the environmentg hashtable.

In this example, because the sessions are instantiated overtly, it is necessary to also do the authentication overtly.

After session instantiation and authentication, a Hello object is instantiated in each session, the helloWorld() method is invoked on each, and the returned String is printed on the console.

The printed output is:

```
Hello from Session1
Hello from Session2
```

```
helloServer/HelloImpl.java
-----
```

This source file implements the two methods specified in the hello.idl file: setMessage() to set the instance variable message, and helloWorld() to return the value set in message.

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`.

Hello.IDL

```
module hello {
    interface Hello {
        wstring helloWorld ();
        void setMessage (in wstring message);
    };
};
```

Client.java

```
import hello.Hello;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;
import oracle.aurora.jndi.sess_iiop.SessionCtx;
import oracle.aurora.AuroraServices.LoginServer;
import oracle.aurora.client.Login;

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];

        // 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);

// Verify that the objects are indeed different
hello1.setMessage ("Hello from Session1");
hello2.setMessage ("Hello from Session2");

System.out.println (hello1.helloWorld ());
System.out.println (hello2.helloWorld ());
}
}
```

サーバー

```
package helloServer;

import hello.*;
import oracle.aurora.AuroraServices.ActivableObject;

public class HelloImpl extends _HelloImplBase implements ActivableObject
{
    String message;

    public String helloWorld () {
        return message;
    }

    public void setMessage (String message) {
        this.message = message;
    }

    public org.omg.CORBA.Object _initializeAuroraObject () {
        return this;
    }
}
```

アプレットの例

JDK および JInitiator のアプレット

README

For all oracle clients, you need to set the following in the environment before creating InitialContext

```
env.put(ServiceCtx.APPLET_CLASS, this);// This refers to the applet class
```

In the HTML file itself, you need to set ORBdisableLocator=true, for all versions. In addition, if you are using JDK 1.2 (i.e, plug in 1.2), then you also need to set the following in the HTML file.

```
<PARAM NAME="org.omg.CORBA.ORBClass" VALUE="com.visigenic.vbroker.orb.ORB">
<PARAM NAME="org.omg.CORBA.ORBSingletonClass" VALUE="com.visigenic.vbroker.orb.ORB">
```

The syntax slightly differs from plugin to plugin. Look at the example applets in this directory.

Note that you don't need to copy any jars on to the client machine. But if you do wish to copy the jar files to the client machine, also set "ClassLoader" to the appletClassLoader (this.getClass().getClassLoader()) and things will work fine. You also don't need to change any files (like java.security etc.) on the client.

JDK 1.1 の HTML

```
<pre>
<html>
<title> CORBA Applet talking to 8i</title>
<h1> CORBA applet talking to 8i using java plug in 1.1 </h1>
<hr>
The good old bank example
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
WIDTH = 500 HEIGHT = 50 codebase="http://java.sun.com/products/plugin/1.1/jinst
all-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,vbj
app.jar" >
<PARAM NAME="type" VALUE="application/x-java-applet;version=1.1">
<PARAM NAME="ORBdisableLocator" VALUE="true">
<COMMENT>
```

```

<EMBED type="application/x-java-applet;version=1.1"
ORBdisableLocator="true" java_CODE = OracleClientApplet.class java_ARCHIVE = "or
acleClient.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>

```

JDK 1.2 の HTML

```

<pre>
<html>
<title> CORBA applet talking to 8i</title>
<h1> CORBA applet talking to 8i using Java plug in 1.2 </h1>
<hr>
The good old bank example
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
WIDTH = 500 HEIGHT = 50 codebase="http://java.sun.com/products/plugin/1.2/jinst
all-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,vbj
app.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.O
RB">
<COMMENT>
<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 = Orac
leClientApplet.class java_ARCHIVE = "oracleClient.jar,aurora_client.jar,vbjorb.j
ar,vbjapp.jar" WIDTH = 500 HEIGHT = 50  pluginspage="http://java.sun.com/produ
cts/plugin/1.2/plugin-install.html">
<NOEMBED></COMMENT>
</NOEMBED></EMBED>
</OBJECT>

</center>
<hr>
</pre>

```

Oracle JInitiator の HTML

```
<h1>CORBA applet talking to 8i using JInitiator 1.1.7.18</h1>
<COMMENT>
<EMBED type="application/x-jinit-applet;version=1.1.7.18"
  java_CODE="OracleClientApplet"
  java_CODEBASE="http://mysun: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="mysun"
  serverPort=8080
<NOEMBED>
</COMMENT>
</NOEMBED>
</EMBED>
```

アプレット・クライアント

```
// ClientApplet.java

import java.awt.*;

import oracle.aurora.jndi.sess_iiop.ServiceCtx;

import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;
import Bank.*;

public class OracleClientApplet extends java.applet.Applet {

  private TextField _nameField, _balanceField;
  private Button _checkBalance;
  private Bank.AccountManager _manager;

  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 {
    String serviceURL = "sess_iiop://mysun:2222";
    String objectName = "/test/myBank";

    // 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(serviceURL + objectName);
} catch (Exception e) {
    System.out.println(e.getMessage());
    e.printStackTrace();
    throw new RuntimeException();
}

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;
}
}
```

Visigenic アプレット

README

To run VisiClient applet, you need to do the following.

Start osagent and gatekeeper (with port 16000)
(for gate keeper, create a file called gatekeeper.properties and just

put this entry in there : exterior_port=16000)

Then start the Bank server (vbj Server &).

Your browser should have Jinitiator installed (use ojdk-pc.us.oracle.com for getting Jinitiator, jdk 1.1.7.18)
(Browser security doesn't have to be off, i.e, you may set it to AppletHost in Jinitiator)

Then simply connect to mysun:8080/applets/bank/VisiClient.html

Visigenic クライアント・アプレットの HTML

```
<h1>Visigenic Client applet</h1>
<COMMENT>
<EMBED type="application/x-jinit-applet;version=1.1.7.18"
  java_CODE="VisiClientApplet"
  java_CODEBASE="http://mysun:8080/applets/bank"
  java_ARCHIVE="visiClient.jar,vbjorb.jar,vbjapp.jar"
  WIDTH=400
  HEIGHT=100
  ORBgatekeeperIOR="http://mysun:16000/gatekeeper.ior"
  USE_ORB_LOCATOR="true"
  ORBbackCompat="true"
  serverHost="mysun"
  serverPort=8080
<NOEMBED>
</COMMENT>
</NOEMBED>
</EMBED>
```

Visigenic クライアント・アプレット

```
// ClientApplet.java

import java.awt.*;

public class VisiClientApplet extends java.applet.Applet {

  private TextField _nameField, _balanceField;
  private Button _checkBalance;
  private Bank.AccountManager _manager;

  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);
// Initialize the ORB (using the Applet).
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(this, null);
// Locate an account manager.
_manager = Bank.AccountManagerHelper.bind(orb, "BankManager");
}

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;
}
}
```

Oracle8i JVM ORB と VisiBroker VBJ ORB の比較

この付録は VisiBroker VBJ ORB をよく理解している開発者を対象としており、VBJ ORB と現在のバージョンの Oracle8i JVM ORB との主な違いを要約したものです。それぞれの ORB はいくつかの使用スタイルをサポートしますが、この付録では最もよく使用されるスタイルのみを比較します。例として、VBJ クライアントはヘルパーの `bind()` メソッドを使用して、名前オブジェクトを検索すると想定します、Oracle8i クライアントはこれと同じ目的に JNDI の `lookup()` メソッドを使用します。また、Oracle8i クライアントは Oracle のセッション IIOP を使用してサーバー・オブジェクトと通信しますが、Oracle8i JVM ORB では VBJ ORB で使用する標準 IIOP がサポートされると想定します。

ORB 間の違いは次の項目にまとめて説明してあります。

- オブジェクト参照はセッションの存続期間のみ有効
- データベース・サーバーが実装のメインライン
- サーバー・オブジェクトの実装はロードと公開により配置
- 継承による実装はほとんど同一
- 委譲による実装は異なる
- クライアントは JNDI でオブジェクト名を検索
- 実装リポジトリは存在しない

この付録の終わりに、VBJ 用の IDL と Aurora ORB 用の同じ IDL による同等のクライアント実装およびサーバー実装を比較のため掲載しました。

オブジェクト参照はセッションの存続期間のみ有効

Aurora ORB では、データベースのセッション内でオブジェクト・インスタンスが作成されます。セッションが消えると、そのセッションで作成されたオブジェクトへの参照は無効になります。使用しようとする、`object does not exist` という例外が発生します。セッションとの最後のクライアント接続がクローズするか、またはセッションのタイムアウト値に達すると、セッションは消えます。セッション内のオブジェクトは、次のメソッドでセッション・タイムアウト値を設定できます。

```
oracle.aurora.net.Presentation.sessionTimeout()
```

このメソッドへのクライアント・インタフェースを提供することもできます。クライアントは、セッションへのクライアント接続がクローズされた後もオブジェクトを存続させたい場合に、これをコールできます。

Oracle8i CORBA オブジェクトの存続期間中の典型的なシナリオは次のとおりです。

- クライアントは、実装が公開されているデータベースを指定して、JNDI でオブジェクトの実装の名前を検索します。
- Oracle ORB は、そのタイプのオブジェクトをインスタンス化して応答として、クライアントへ参照を戻します。
- クライアントはそのオブジェクト上でメソッドをコールし、その後同じオブジェクト上でメソッドをコールする可能性のある他のクライアントに参照を渡します。
- セッションが破棄されるとオブジェクトは消滅します。

データベース・サーバーが実装のメインライン

Oracle8i サーバー・オブジェクトの実装は、単一のクラスから成り立っています。データベース・サーバーがメインラインなので、開発者はメインライン・サーバーを作成しません。データベースが動作している場合、クライアントはそのデータベースで公開されているすべての実装を使用できます。データベース・サーバーは MTS スレッドを各実装に動的に割り当てます。Java スレッドを使用することで、各実装内での処理をマルチスレッド化できます。

サーバー・オブジェクトの実装はロードと公開により配置

`loadjava` ツールでオブジェクトの実装をデータベースにロードすると、その実装は、そのデータベースで実行中の ORB からアクセスできるようになります。`publish` ツールを使用して、ロードされた実装の名前をデータベースのセッション・ネームスペースに公開すると、その実装はクライアントから名前でもアクセスできるようになります。すべての CORBA オブジェクトの実装をロードする必要がありますが、公開する必要があるのはクライアントにより名前が検索されるもののみです。

継承による実装はほとんど同一

Oracle8i 上の仮想的なインタフェース Alpha を実装するためには、AlphaImplBase を継承して、IDL 操作を実装する Java メソッドを定義する AlphaImpl というクラスを作成します。また、新しいインスタンスの作成時に Oracle ORB がコールする `_initializeAuroraObject()` メソッドにインスタンス初期化コードを実装できます。

委譲による実装は異なる

委譲 (Tie) による Oracle8i の実装では、記述する実装クラスは、定義済みのクラスを継承し、2つの Oracle 定義のインタフェースを実装します。IDL インタフェース名に Operations を結びつけた名前を持つ最初のインタフェースでは、IDL 操作に対応するメソッドが定義されています。ActivatableObject という名前の 2 番目のインタフェースでは、`_initializeAuroraObject()` という名前の 1 つのメソッドが定義されます。このメソッドの実装中では、単にインスタンスを作成して戻します。簡単な例を次に示します。

```
// IDL
module hello {
    interface Hello {
        wstring helloWorld ();
    };
};

// Aurora tie implementation
package helloServer;

import hello.*;
import oracle.aurora.AuroraServices.ActivatableObject;

public class HelloImpl implements HelloOperations, ActivatableObject
//, extends <YourClass>
{
    public String helloWorld () {
        return "Hello World!";
    }

    public org.omg.CORBA.Object _initializeAuroraObject () {
        // create and initialize an instance and return it, for example ...
        return new _tie_Hello (this);
    }
}
```

クライアントは JNDI でオブジェクト名を検索

Oracle8i クライアントは、CORBA CosNaming またはより単純な JNDI (Java Naming and Directory Interface) を使用し、公開されているオブジェクトを名前を検索できます。

クライアントは、Java コンストラクタで特定のデータベースの JNDI 初期コンテキストを作成します。たとえば、次のようにします。

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

env パラメータでは、クライアントのログインに使用されるユーザー名とパスワードを指定します。オブジェクトの実装はデータベース・サーバー内で実行されるため、CORBA オブジェクトのユーザーは (クライアントを介して) 通常のデータベース操作と同様に、データベースに対して自分自身の識別名の提出および認証を行う必要があります。

公開されている実装のインスタンスを取得するために、クライアントは JNDI コンテキストの lookup() メソッドをコールし、ターゲット・データベースを指定する URL および必要なオブジェクトの実装の公開されている名前を渡します。lookup() コールでは、ターゲット・データベース内のインスタンスへの参照が戻されます。クライアントは他のクライアントに (多くの場合、文字列形式で) 参照を渡すこともあり、その参照は対応付けられているオブジェクトが作成されたセッションが存続する限り有効です。同じオブジェクト参照のコピーを使用するクライアントは、オブジェクトのデータベース・セッションを共有します。

クライアントが同じパラメータを使用して lookup() を 2 回続けて実行した場合、2 回目のオブジェクト参照は最初のものと同じです。つまり、最初の lookup() コールにより作成されたインスタンスが参照されます。しかし、クライアントが 2 番目のセッションを作成し、そのセッションで 2 回目の lookup() を実行した場合には、異なるインスタンスが作成され、参照が戻されます。

実装リポジトリは存在しない

現在のバージョンの Oracle8i ORB には、実装リポジトリは含まれていません。

Aurora と VBJ の銀行の例

次の項では、VBJ ドキュメンテーションで広く使用されている銀行の例の実装を比較します。クライアントとサーバーはいずれも、Oracle8i と VBJ で実装されるとおりに示します。すべての実装で継承が使用されます。

銀行 IDL モジュール

```
// Bank.idl

module Bank {
    interface Account {
        float balance();
    };
    interface AccountManager {
        Account open(in string name);
    };
};
```

Aurora クライアント

```
// Client.java

import bankServer.*;
import Bank.*;

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:2222";
        String objectName = "/test/myBank";
        String username = "scott";
        String password = "tiger";
```

```
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);

AccountManager manager =
    (AccountManager) ic.lookup(serviceURL + objectName);

// use args[0] as the account name, or a default.
String name = args.length == 1 ? args[0] : "Jack B. Quick";

// Request the account manager to open a named account.
Bank.Account account = manager.open(name);

// Get the balance of the account.
float balance = account.balance();

// Print out the balance.
System.out.println
    ("The balance in " + name + "'s account is $" + balance);
}
}
```

VBJ クライアント

```
// Client.java

public class Client {

    public static void main(String[] args) {
        // Initialize the ORB.
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
        // Locate an account manager.
        Bank.AccountManager manager =
            Bank.AccountManagerHelper.bind(orb, "BankManager");
        // use args[0] as the account name, or a default.
        String name = args.length > 0 ? args[0] : "Jack B. Quick";
        // Request the account manager to open a named account.
        Bank.Account account = manager.open(name);
        // Get the balance of the account.
        float balance = account.balance();
        // Print out the balance.
        System.out.println
```

```
        ("The balance in " + name + "'s account is $" + balance);
    }
}
```

Aurora アカウントの実装

```
// AccountImpl.java
package bankServer;

public class AccountImpl extends Bank._AccountImplBase {
    public AccountImpl(float balance) {
        _balance = balance;
    }
    public float balance() {
        return _balance;
    }
    private float _balance;
}
```

VBJ アカウントの実装

```
// AccountImpl.java

public class AccountImpl extends Bank._AccountImplBase {
    public AccountImpl(float balance) {
        _balance = balance;
    }
    public float balance() {
        return _balance;
    }
    private float _balance;
}
```

Aurora アカウント・マネージャの実装

```
// AccountManagerImpl.java
package bankServer;

import java.util.*;

public class AccountManagerImpl extends Bank._AccountManagerImplBase {

    public AccountManagerImpl() {
        super();
    }

    public AccountManagerImpl(String name) {
        super(name);
    }

    public synchronized Bank.Account open(String name) {
        // Lookup the account in the account dictionary.
        Bank.Account account = (Bank.Account) _accounts.get(name);
        // If there was no account in the dictionary, create one.
        if(account == null) {

            // Make up the account's balance, between 0 and 1000 dollars.
            float balance = Math.abs(_random.nextInt()) % 100000 / 100f;

            // Create the account implementation, given the balance.
            account = new AccountImpl(balance);

            _orb().connect (account);

            // Print out the new account.
            // This just goes to the system trace file for Aurora.
            System.out.println("Created " + name + "'s account: " + account);

            // Save the account in the account dictionary.
            _accounts.put(name, account);
        }
        // Return the account.
        return account;
    }

    private Dictionary _accounts = new Hashtable();
    private Random _random = new Random();
}
```

VBJ アカウント・マネージャの実装

```
// AccountManagerImpl.java

import java.util.*;

public class AccountManagerImpl extends Bank._AccountManagerImplBase {
    public AccountManagerImpl(String name) {
        super(name);
    }
    public synchronized Bank.Account open(String name) {
        // Lookup the account in the account dictionary.
        Bank.Account account = (Bank.Account) _accounts.get(name);
        // If there was no account in the dictionary, create one.
        if(account == null) {
            // Make up the account's balance, between 0 and 1000 dollars.
            float balance = Math.abs(_random.nextInt()) % 100000 / 100f;
            // Create the account implementation, given the balance.
            account = new AccountImpl(balance);
            // Make the object available to the ORB.
            _boa().obj_is_ready(account);
            // Print out the new account.
            System.out.println("Created " + name + "'s account: " + account);
            // Save the account in the account dictionary.
            _accounts.put(name, account);
        }
        // Return the account.
        return account;
    }
    private Dictionary _accounts = new Hashtable();
    private Random _random = new Random();
}
```

VBJ サーバーのメインライン

```
// Server.java

public class Server {

    public static void main(String[] args) {
        // Initialize the ORB.
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
    }
}
```

```
// Initialize the BOA.
org.omg.CORBA.BOA boa = orb.BOA_init();
// Create the account manager object.
Bank.AccountManager manager =
    new AccountManagerImpl("BankManager");
// Export the newly created object.
boa.obj_is_ready(manager);
System.out.println(manager + " is ready.");
// Wait for incoming requests
boa.impl_is_ready();
}
}
```

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。すべての頭字語を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・
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)
IIOIP	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)

索引

記号

`_get_interface_def` メソッド, 5-7, 5-8
`_initializeAuroraObject` メソッド, 2-24

数字

2 タスク共通, 「TTC」を参照
2 フェーズ・コミット, 7-15

A

ACID プロパティ, 7-2
ActivatableObject インタフェース, 2-24
 `_initializeAuroraObject` メソッド, 2-24
ADDRESS パラメータ, 3-11, 3-16
APPLET_CLASS プロパティ, 5-16
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
AuroraTransactionService クラス, 7-19
 `initialize` メソッド, 7-19, 7-24
aurora.zip, 4-28
authenticate メソッド, 4-20, 6-11

B

begin メソッド, 7-9, 7-12, 7-19, 7-20, 7-21, 7-26
bindds コマンド, 7-8, 7-17
bindut コマンド, 7-7, 7-15
BOA
 `obj_is_ready` メソッド, 2-24

C

ClassLoader プロパティ, 5-16
commit メソッド, 7-9, 7-10, 7-12, 7-19, 7-20, 7-21, 7-27
Contained オブジェクト, 5-8
Container オブジェクト, 5-8
CORBA
 Java 2 サポート, 5-11
 Java 2 を使用した純粋な CORBA, 5-15
 Tie メカニズム, 5-11
 コールバック, 5-3
 システム例外, 2-22
 スケルトン, 2-4
 スタブ, 2-4
 ドキュメントが置かれている Web サイト, 1-8
 ネームサービスの取得, 4-29
CosNaming サービス, 4-1, 4-3, 4-29
Current クラス
 begin メソッド, 7-19, 7-20, 7-21
 commit メソッド, 7-19, 7-20, 7-21
 resume メソッド, 7-19, 7-20, 7-21
 rollback_only メソッド, 7-19, 7-20, 7-21
 rollback メソッド, 7-19, 7-20, 7-21
 suspend メソッド, 7-19, 7-20, 7-21

D

Database Configuration Assistant, 3-8
DataSource オブジェクト
 動的作成, 7-17
 ネームスペースでのバインド, 7-8

DebugAgent クラス, 2-25
 restart メソッド, 2-25
 stop メソッド, 2-25
DESCRIPTION パラメータ, 3-10

E

endSession メソッド, 4-20

G

General Inter-ORB Protocol, 「GIOP」を参照
get_status メソッド, 7-28
get_transaction_name メソッド, 7-28
getCurrent メソッド, 7-19, 7-20, 7-21
getTS メソッド, 7-19, 7-21, 7-25
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

IDL, 1-4
 IFR, 5-7, 5-8
 インタフェース, 2-3
 言語マッピング, 2-3
 スケルトン, 2-6
idl2java ツール, 2-4
IFR, 5-7, 5-8
 Repository オブジェクト, 5-7
 オブジェクト階層, 5-8
 概要, 5-6
IIOP, 1-7, 3-2, 4-16
 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
 構成, 5-24
 プロファイル, 4-14
IIOP クライアント
 構成, 3-1, 3-19
InitialContext オブジェクト, 4-13
initialize メソッド, 7-19, 7-24
init メソッド, 2-23

Inprise, 1-8
 VisiBroker for Java, B-1
 サポートされていないバージョン, 5-11
InterfaceDef クラス, 5-8
Internet Inter-ORB Protocol, 「IIOP」を参照

J

Java 2
 JDK 1.1 からの移行, 5-11
Java Naming and Directory Interface, 「JNDI」を参照
Java Transaction API, 「JTA」を参照
Java Transaction Service, 「JTS」を参照
java2idl ツール, 2-23
java2iiop ツール, 2-23
javax-ssl-1_1.jar, 4-12, 6-4
javax-ssl-1_2.jar, 4-12, 6-4
JDeveloper
 デバッグ, 2-24
JNDI
 InitialContext コンストラクタ, 4-13
 lookup メソッド, 4-8, 4-13
 コンテキスト・オブジェクト, 4-10
 初期コンテキスト, 4-3
jssl-1_1.jar, 4-12, 6-4
jssl-1_2.jar, 4-12, 6-4
JTA
 2 フェーズ・コミット, 7-5, 7-15
 概要, 7-1, 7-2
 クライアント側デマーケーション, 7-9
 サーバー側デマーケーション, 7-7
 仕様の Web サイト, 7-1
 制限事項, 7-6
 タイムアウト, 7-18
 ネストしたトランザクション, 7-6
 リソースの確保, 7-5, 7-12
JTS, 7-1
 begin メソッド, 7-26
 commit メソッド, 7-27
 get_status メソッド, 7-28
 get_transaction_name メソッド, 7-28
 getTS メソッド, 7-25
 rollback_only メソッド, 7-28
 rollback メソッド, 7-28
 suspend メソッド, 7-26
 概要, 7-18
 起動ステップ, 7-19

クライアント側デマーケーション, 7-19
サーバー側デマーケーション, 7-21
初期化, 7-18
制限事項, 7-23

L

loadjava ツール, 2-11
LoginServer クラス, 6-11
 authenticate メソッド, 4-20, 6-11
Login クラス, 4-6, 6-11
LogoutServer クラス, 4-20, 6-11
logout メソッド, 4-20, 6-11
lookup メソッド, 4-12, 4-13

M

MTS_DISPATCHERS パラメータ
 ADDRESS 属性, 3-16
 PRESENTATION 属性, 3-9, 3-16
 PROTOCOL 属性, 3-9
 概要, 3-3
 構成, 5-24

N

NameService
 取得, 4-29
narrow メソッド, 2-6
Net8 Assistant
 IIOP クライアント用の構成, 3-6
 構成, 3-10
NON_SSL_LOGIN の値, 4-3, 4-11

O

obj_is_ready メソッド, 2-24
oracle.aurora.server.SGiopServer, 3-9
OracleJTADDataSource クラス, 7-17
ORB
 初期化, 2-23, 5-12, 6-24
ORBClass プロパティ, 5-15, 5-18
ORBDefaultInitRef, 4-33
ORBdisableLocator プロパティ, 5-18
ORBInitRef, 4-33
ORBSingletonClass プロパティ, 5-15, 5-18
OSS.SOURCE.MY_WALLET パラメータ, 3-19

P

PRESENTATION 属性, 3-9, 3-11, 3-16
PROTOCOL_STACK パラメータ, 3-11
PROTOCOL 属性, 3-9
publish コマンド
 IFR, 5-7
publish ツール, 2-12, 5-7

R

RAW セッション層, 3-11
regep ツール, 3-15, 3-16
Repository オブジェクト, 5-8
 IFR, 5-7
restart メソッド, 2-25
resume メソッド, 7-19, 7-20, 7-21, 7-27
rollback_only メソッド, 7-19, 7-20, 7-21, 7-28
rollback メソッド, 7-9, 7-10, 7-12, 7-19, 7-20,
 7-21, 7-28

S

SECURITY_AUTHENTICATION プロパティ, 4-11
SECURITY_CREDENTIALS プロパティ, 4-11
SECURITY_PRINCIPAL プロパティ, 4-11
SECURITY_ROLE プロパティ, 4-11
SESSION 属性, 3-11
setCertificateChain メソッド, 6-24
setEncryptedPrivateKey メソッド, 6-24
setTransactionTimeout メソッド, 7-18
SID, 4-7
SQLJ, 5-2
SSL, 6-20
 JAR ファイル, 4-12, 6-4
 構成, 3-17
 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, 4-11
SSL_CREDENTIAL の値, 4-11
SSL_LOGIN の値, 4-11
SSL_VERSION パラメータ, 3-19
SSL_VERSION プロパティ, 3-19
start メソッド, 2-25
stop メソッド, 2-25
suspend メソッド, 7-19, 7-20, 7-21, 7-26

T

Tie メカニズム, 5-11
TransactionManager クラス, 7-3
TransactionService クラス, 7-19, 7-24
 getCurrent メソッド, 7-19, 7-20, 7-21
Transaction クラス, 7-3
TRANSPORT_TYPE プロパティ, 5-24
TS クラス
 getTS メソッド, 7-19, 7-21
TTC, 4-14

U

URL
 構文, 4-6
URL_PKG_PREFIXES プロパティ, 4-10
USE_SERVICE_NAME プロパティ, 4-12
UserTransaction インタフェース, 7-24
UserTransaction オブジェクト
 begin メソッド, 7-9, 7-12
 commit メソッド, 7-9, 7-10, 7-12
 rollback メソッド, 7-9, 7-10, 7-12
 setTransactionTimeout メソッド, 7-18
 取得, 7-7
 ネームスペースでのバインド, 7-7
useServiceName フラグ, 4-7
 deployejb のオプション, 4-12

V

VisiBroker for Java, 1-8

W

Wallet, 6-20
Web サイト
 CORBA, 1-8

あ

アプレット
 サーバー・オブジェクトの起動, 5-16
 サンドボックス・セキュリティの制限事項, 5-16

い

インタセプタ, 2-24
インタフェース
 IFR, 5-7, 5-8
 IFR からの取得, 5-6
 定義済み, 2-3
インタフェース定義言語, 「IDL」を参照
インタフェース・リポジトリ, 「IFR」を参照

う

受渡し, 3-14

え

エンドポイント, 3-13
 登録, 3-15

お

オブジェクト・クラス
 _get_interface_def メソッド, 5-8
オブジェクトの活性化, 2-23, 2-24, 4-29
 セッション中, 4-25, 4-29

く

クライアント
 既存の Bean へのアクセス, 4-23
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, 4-5
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
コード例, A-1
コールアウト
 SSL の使用, 6-21
コールバック, 5-3
 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

コレクション
IDL, 2-20
コンテキスト
JNDI オブジェクト, 4-10

さ

サーバー側の認証, 6-5
サービス名, 2-9, 4-7, 4-12

し

システム識別子, 「SID」を参照
システム例外, 2-22
実装, 2-6
証明書, 6-20, 6-21, 6-24
マネージャ, 6-24

す

スケルトン, 2-6

せ

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, 4-20
ルーティング, 4-15
ログアウト, 4-20, 6-11
セッション中の活性化, 4-25

て

ディスパッチャ
概要, 3-11
構成, 3-11
直接接続, 3-11
データ整合性, 6-3
デバッグ手法, 2-24

と

頭字語, C-1
トランザクション
2 フェーズ・コミット, 7-5, 7-15
概要, 7-2
クライアント側デマーケーション, 7-9

グローバル, 7-3
コンテキストの伝播, 7-4
サーバー側デマーケーション, 7-7
制限事項, 7-6, 7-23
タイムアウト, 7-18
単一フェーズ・コミット
例, 7-8
デマーケーション, 7-3
リソースの確保, 7-5, 7-12
トレース・ファイル, 2-25

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, 4-20, 6-11

ね

ネームスペース, 4-4

は

バージョン
Visibroker, 5-11
パラメータの受渡し
値, 2-20

ふ

プレゼンテーション
GIOP, 3-2, 3-9
oracle.aurora.server.SGiopServer, 3-9
プロパティ
ORBClass, 5-15
ORBSingletonClass, 5-15

へ

ヘルパー・クラス
narrow メソッド, 2-6

ほ

ホルダー・クラス, 2-6

り

リスナー, 3-11

 エンドポイント登録, 3-15

 概要, 3-11

 構成, 3-13

 接続の受渡し, 3-14

 リダイレクション, 3-12, 3-13

リダイレクション, 3-12, 3-13, 3-15

れ

例外

 IDL, 2-21

ろ

ログイン

 非 JNDI ログイン, 4-20, 6-11