Oracle8*i*

Java 開発者ガイド

リリース 8.1

2000 年 11 月

部品番号：J02315-01

ORACLE®

Oracle8*i* Java 開発者ガイド , リリース 8.1

部品番号：J02315-01

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

原本部品番号：A83728-01

原本著者：Sheryl Maring

原本協力者 : Steve Harris, Ellen Barnes, Peter Benson, Greg Colvin, Bill Courington, Matthieu Devin, Jim Haungs, Hal Hildebrand, Mark Jungerman, Susan Kraft, Thomas Kurian, Scott Meyer, Tom Portfolio, Dave Rosenberg, Jerry Schwarz, Harlan Sexton, Tim Smith, David Unietis, Brian Wright.

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

Printed in Japan.

制限付権利の説明

プログラム（ソフトウェアおよびドキュメントを含む）の使用、複製または開示は、オラクル社との契約に記された制約条件に従うものとします。著作権、特許権およびその他の知的財産権に関する法律により保護されています。

当プログラムのリバース・エンジニアリング等は禁止されております。

このドキュメントの情報は、予告なしに変更されることがあります。オラクル社は本ドキュメントの無謬性を保証しません。

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

危険な用途への使用について

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

Restricted Rights Notice

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

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

目次

はじめに vii

1 Oracle8*i* での Java の紹介

内容 1-2
Java の概要 1-2
Java およびオブジェクト指向プログラミングの用語 1-2
クラス 1-2
属性 1-3
メソッド 1-4
クラス階層 1-4
インタフェース 1-5
ポリモフィズム 1-6
Java 仮想マシン（JVM） 1-7
Java 言語のキーとなる機能 1-10
Oracle8*i* で Java を使用する理由 1-11
マルチスレッド 1-11
自動記憶域管理 1-12
フットプリント 1-13
パフォーマンス 1-13
ネイティブ・コンパイラによるパフォーマンス向上の仕組み 1-14
クラスの動的ロード 1-14
Oracle の Java アプリケーションの方針 1-15
Java ストアド・プロシージャ 1-16

PL/SQL の統合と Oracle RDBMS 機能 1-16
JDBC ドライバ 1-17
SQLJ（Java 埋込み SQL） 1-17
Java を起動する動的 HTML 1-18
サーブレット 1-18
JavaServer Pages 1-18
分散アプリケーションの開発 1-19
EJB コンポーネントの使用 1-19
開発ツール 1-20
Oracle8*i* Java マニュアルの概要 1-20

2 Oracle8*i* での Java アプリケーションの作成

概要 2-2
用語 2-2
Java アプリケーションにインポーズされるデータベース・セッション 2-3
セッションの存続期間 2-5
Java をサポートする API 2-5
実行制御 2-6
JDK 1.1 から Java 2 への移行 2-7
開発環境 2-7
JDBC 2.0 2-8
JDBC 2.0 を使用するサーバー・アプリケーション 2-9
JDBC 2.0 を使用するクライアント 2-9
Java 2 セキュリティ 2-10
Java 2 ORB の API 2-10
JNDI ルックアップ 2-11
Aurora ORB インタフェース 2-11
CORBA ORB インタフェース 2-12
8.1.5 CORBA と EJB アプリケーションの下位互換性 2-13
Java のコード、バイナリおよびリソースの格納 2-13
Java クラス・メソッドの実行準備 2-14
Java クラスのコンパイル 2-14
javac によるソースのコンパイル 2-15
loadjava によるソースのコンパイル 2-15
ランタイムでのソースのコンパイル 2-15
コンパイラ・オプションの指定 2-16
自動再コンパイル 2-18

クラス依存関係の解決 2-19
存在しないクラスへの参照のパーミッション 2-20
ByteCode ベリファイヤ 2-21
クラスのロード 2-22
同じクラスの 2 つの定義 2-24
必要なデータベース権限および JVM のパーミッション 2-25
JAR ファイルまたは ZIP ファイルのロード 2-25
実行権限を付与する方法 2-26
Java アップロードのチェック 2-27
オブジェクト名とタイプ 2-27
Status 2-28
公開 2-29
サーバー上のユーザー・インタフェース 2-29
クラスの短縮名 2-30
Oracle8i JVM の Class.forName() 2-31
Class.forName への ClassLoader の提供 2-32
classForNameAndSchema へのクラス名およびスキーマ名の提供 2-33
lookupClass へのクラス名およびスキーマ名の提供 2-33
シリアル化を行う際のクラス名およびスキーマ名の提供 2-34
Class.forName の例 2-34
オペレーティング・システム・リソースの管理 2-35
オペレーティング・システム・リソースの概要 2-36
オペレーティング・システム・リソースへのアクセス 2-36
オペレーティング・システム・リソースの存続期間 2-36
ガベージ・コレクションとオペレーティング・システム・リソース 2-37
コール間で影響を受けるオペレーティング・システム・リソース 2-38
ソケット 2-40
Oracle8i JVM におけるスレッド 2-41
スレッドのライフ・サイクル 2-42

3 データベースでの Java のコール

概要 3-2
Java メソッドのコール 3-3
Java ストアド・プロシージャの利用 3-3

CORBA と EJB を使用した分散オブジェクトの使用 3-6
IIOP 移送 3-7
ネーミング 3-7
Enterprise JavaBeans の作成と配置 3-8
EJB の使用方法 3-9
セッション・シェル 3-9
Remote Method Invocation（RMI）の使用 3-10
Java ネイティブ・インタフェース（JNI）サポートの使用 3-10
SQLJ および JDBC を使用したデータベースの問合せ 3-10
JDBC 3-11
SQLJ 3-11
JDBC と SQLJ の比較例 3-12
完全な SQLJ の例 3-13
SQLJ の厳密な型指定パラダイム 3-15
SQLJ プログラムの変換 3-15
サーバーでの SQLJ プログラムの実行 3-15
クライアント・アプリケーションの変換とサーバーでの実行 3-16
PL/SQL との対話 3-16
サーバー・アプリケーションのデバッグ 3-17
1. コードのデバッグ準備 3-18
2. デバッグ・プロキシの起動 3-19
3. デバッグ・エージェントの起動、停止および再起動 3-20
OracleAgent クラス 3-21
4. デバッガの接続 3-21
サーバー上での実行を見分ける方法 3-22
サーバー上の出力のリダイレクト 3-23

4 Java のインストールと構成

Java 対応データベースの初期化 4-2
手動インストール 4-2
要件 4-3
パッケージ DBMS_JAVA 4-3
Oracle8i JVM の構成 4-5
Java ストアド・プロシージャの構成 4-7
Enterprise JavaBeans と CORBA の構成 4-8

Java クライアントの使用可能化 4-8
1. クライアントへの JDK のインストール 4-8
2. CLASSPATH の設定 4-9
基本として組み込む JAR ファイル 4-9
JDK 1.1 クライアントに必要な JAR ファイル 4-9
Java 2 クライアントに必要な JAR ファイル 4-10
SQLJ を使用するクライアントに組み込む JAR ファイル 4-10
JSP を使用するクライアントに組み込む JAR ファイル 4-10
3. ポート /SID の検証 4-11
4. サンプルを使用したインストールのテスト 4-11

5 Oracle8*i* Java アプリケーションのセキュリティ

ネットワーク接続のセキュリティ 5-2
データベースの内容と JVM セキュリティ 5-2
Java 2 セキュリティ 5-3
パーミッションの設定 5-6
各パーミッションのきめ細かい定義 5-7
ポリシー表を更新するための管理のパーミッションの取得 5-12
パーミッションの作成 5-14
パーミッションの使用可能または使用禁止 5-17
パーミッションのタイプ 5-19
初期のパーミッションの付与 5-20
ロールに割り当てられる一般的なパーミッション定義 5-23
デバッグのパーミッション 5-24
クラスをロードするためのパーミッション 5-25

6 Oracle8*i* Java アプリケーションのパフォーマンス

ネイティブ・コンパイル・コード 6-2
Oracle8i JVM Accelerator の概要 6-3
Oracle8i JVM の Java コア・クラス・ライブラリ 6-4
Java アプリケーションのクラス・ライブラリのネイティブ・コンパイル 6-4
インストールの要件 6-4
Oracle8i JVM Accelerator の実行 6-6

Java のメモリー使用量 6-7
メモリー初期化パラメータの構成 6-7
Java プール・メモリー 6-8
Java プール・メモリー使用量の表示 6-9
メモリー不足エラーの修正 6-10
コンパイル時のメモリー不足 6-10
ロード時のメモリー不足 6-10
コール終了時の移行 6-11
コール終了時の最適化に対する Oracle 固有のサポート 6-12
メモリー・プロファイリング・ユーティリティ 6-16
MemStat の機能 6-17
MemStat の使用 6-17
MemStat パーミッション 6-19
MemStat レポート形式 6-19
サンプル出力 6-21

用語集

索引

はじめに

対象読者

このマニュアルの対象読者は次のとおりです。

- 管理者－ Oracle8*i* を購入した目的がデータベースでの Java 開発ではない場合もあります。しかし、今後の管理を見通す上で Oracle8*i* の Java 機能に関心のある方は、1-20 ページの「Oracle8i Java マニュアルの概要」を参照してください。
- 非 Java 開発者－ Oracle データベース・プログラミングは、PL/SQL およびその他の非 Java プログラミングで構成されます。PL/SQL 開発はよく理解しているが、まだ Java の経験が浅い方は、第 1 章「Oracle8i での Java の紹介」の冒頭にある Java の概要とオブジェクト指向を参照してください。Java の詳細情報の入手については、この「はじめに」の最後の「Java 関連情報の参照先」を参考にしてください。
- Java 開発者－ Java の開発者は、Sun Microsystems の仕様に準拠した Java 環境には慣れています。しかし、Java がデータベースに組み込まれると、Java とデータベースの概念が融合されます。したがって、Oracle8*i* の Java 環境は、データベースに関する事柄も含むように拡張されています。このマニュアルでは、主にデータベースで Java を実行するために理解しておく必要のある、様々な相違点を紹介しています。この融合から生じる 2 つの観点のアウトラインを次に示します。
    - Java 環境 － Oracle8*i* は Java 対応の実装を提供します。つまり、100% pure Java コードを実行できます。Oracle8*i* JVM は、Java 開発におけるクラスおよびクラスのある環境の管理方法に影響があります。たとえば、クラスはデータベースにロードする必要があります。また、Oracle8*i* のモデルではクライアントとサーバーが明確に区別されます。
    - データベース環境 － Java オブジェクトを管理するためにデータベースの概念を認識しておく必要があります。このマニュアルでは、Oracle8*i* データベースと Java 環境という、2 つの分野がどのように適合するかを広く紹介します。たとえば、セキュリティ・ポリシーを決める際には、総合的なセキュリティ・ポリシーのために、データベース・セキュリティと Java セキュリティの両方を考慮する必要があることなどです。

Java API プログラミング・モデル

Java 開発者が Oracle8*i* で使用する構成要素は次のとおりです。

- Java ストアド・プロシージャ － データベースに格納される Java アプリケーションを開発できます。これらのプロシージャはいったんロードすれば、SQL、PL/SQL から起動できる他、トリガーとして起動できます。詳細は、『Oracle8*i* Java ストアド・プロシージャ開発者ガイド』を参照してください。
- JDBC と SQLJ － クライアントから、またはサーバー上で直接 SQL データにアクセスする Java アプリケーションを作成できます。
- 分散 Java による CORBA または EJB アプリケーション － データベースにロードされ、起動される EJB または CORBA アプリケーションを開発できます。

これらの各モデルは第 1 章「Oracle8i での Java の紹介」で簡単に説明します。また、第 3 章「データベースでの Java のコール」の例も参照してください。この 2 つの章は、アプリケーションに使用するモデルを決める上で役立ちます。適切なモデルを決定した後、該当する開発者ガイドで各モデルの詳細を参照してください。たとえば、Java ストアド・プロシージャを使用する場合は、『Oracle8*i* Java ストアド・プロシージャ開発者ガイド』を参照してください。

Java 関連情報の参照先

次の表は、Java プログラミング・マニュアルのセットで取り上げている最新情報の参照先です。

場所	説明
`http://www.oracle.com/java`	Oracle8*i* データベースの Java 向けの最新製品、アップデートおよび最新情報。このサイトでは、よくある質問（FAQ）、JDBC ドライバのアップデート、SQLJ 参照の実装、および Java アプリケーション開発を詳説するホワイトペーパを提供しています。また、このサイトからは試供用の Java ツールをダウンロードできます。
`http://java.sun.com/`	Java に関する情報の中心である Sun Microsystems の Web サイト。このサイトでは、Java 製品や、チュートリアル、推奨文献、Java Developer's Kit（JDK）などの情報を提供しています。JDK は `http://java.sun.com/products` から入手できます。
`http://java.sun.com/docs/books/jls` `http://java.sun.com/docs/books/vmspec`	Oracle8*i* JVM は Java 言語（JLS）および Java 仮想マシン（JVM）仕様に基づいています。
`comp.lang.java.programmer` `comp.lang.java.databases`	インターネット・ニュースグループは、他の Java 開発者からの Java に関する有益な情報源です。左記の 2 つのニュースグループには常に注目しておくことをお薦めします。 **注意：**オラクル社ではこれらのニュースグループの一部の活動をモニターし、Oracle 固有の問題に対する答えを掲示しています。

全国の書店またはオンライン書店でも Java の参考資料を入手できます。また、その他の初心者にも役立つ、一般的なリファレンスとしても利用できる資料リストが『Oracle8*i* Java ストアド・プロシージャ開発者ガイド』に掲載されています。

1

Oracle8*i* での Java の紹介

Java アプリケーションは Oracle8*i* データベースにおいてサポートされます。Java アプリケーションには、簡単なスタンドアロンのアプリケーションから、EJB や CORBA を使用した大規模なエンタープライズ・ソリューションまでありますが、サポートされるすべての Java API を 1 冊のマニュアルで説明するのは困難なため、Oracle8*i* における Java の全サポートは数冊のマニュアルで解説しています。このマニュアルでは、Java アプリケーションをデータベースにロードして実行するためのプログラミング方法を概説します。また、開発する Java アプリケーションのタイプを選択する際に必要な知識を提供し、そのタイプに対応する詳細な情報を提供するマニュアルも紹介します。

この章では、次の情報を取り上げます。

- Oracle データベース・プログラマのために Java 言語の基本を解説します。Oracle PL/SQL 開発者は、SQL データが密接に統合されたサーバー側アプリケーションの開発をよく理解しているので、Oracle データベースの拡張性とパフォーマンスを生かした、Java のサーバー側アプリケーションを開発できます。Java について知るには、1-2 ページの「Java の概要」を参照してください。
- Oracle8*i* データベース内で、Java を使用する必要性を考察します。1-11 ページの「Oracle8i で Java を使用する理由」を参照してください。また、Oracle8*i* でサポートされる Java アプリケーション開発用インタフェースも簡単に説明します。これには、SQLJ、JDBC、Java ストアド・プロシージャ、EJB および CORBA が含まれます。1-15 ページの「Oracle の Java アプリケーションの方針」を参照してください。
- Oracle8*i* Java マニュアル へのロードマップを提供します。Oracle8*i* では複数のタイプの Java アプリケーションがサポートされています。このマニュアルではこれらのタイプを概説するにとどめ、詳細は各タイプ別のマニュアルで説明します。1-20 ページの「Oracle8i Java マニュアルの概要」に各タイプの詳細の参照先をまとめてあります。

内容

- Java の概要
- Oracle8i で Java を使用する理由
- Oracle の Java アプリケーションの方針
- Oracle8i Java マニュアルの概要

Java の概要

Sun Microsystems で開発された Java は、優れたオブジェクト指向のプログラミング言語として数年前から台頭してきました。Java には、次のような概念が組み込まれています。

- プラットフォームからの独立性を実現するための基盤を提供する、Java 仮想マシン（JVM）
- ガベージ・コレクションに代表される自動記憶域管理技法
- C から流用した、厳密な型指定を施行する言語構文

この結果、既存の C プログラマが簡単に習得できる言語でありながら、アプリケーション・レベルのプログラムにオブジェクト指向と能率の良さを提供する言語となっています。

Java およびオブジェクト指向プログラミングの用語

ここでは、Oracle8*i* 環境における Java アプリケーション開発の説明で使用する、基本的な用語を説明します。ここで説明する用語は、Java プログラマにはすでに浸透しているものです。オブジェクト指向プログラミング全般および Java 言語の詳細な説明は、このマニュアルの対象外です。様々な書籍や詳細な言語仕様書は、書店またはインターネットで入手できます。参考資料およびインターネットでの Java 関連の情報入手先の詳細は、「はじめに」の「Java 関連情報の参照先」と、『Oracle8*i* Java ストアド・プロシージャ開発者ガイド』の「参考資料」を参照してください。

クラス

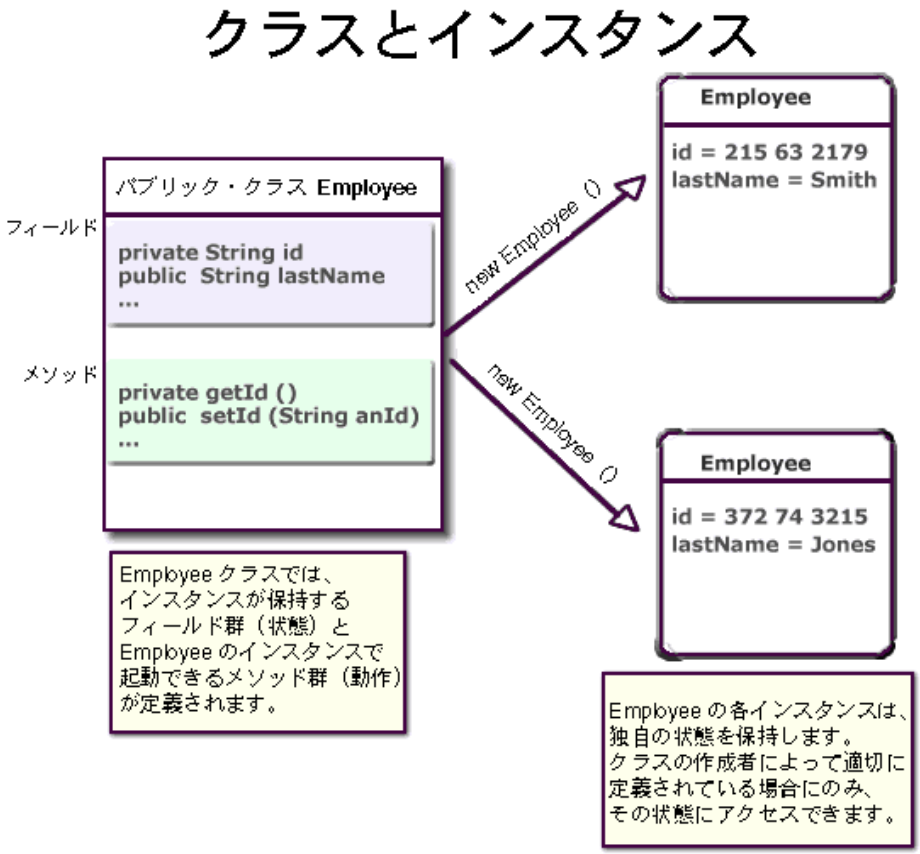
クラスの概念は、どのオブジェクト指向プログラミング言語でもサポートされています。クラスは、表定義と同じように、共通の特性を持つオブジェクトのテンプレートを提供します。各クラスには、次のものが含まれています。

- 属性 — 特定のクラスの各オブジェクトが持つ静的変数またはインスタンス変数。
- メソッド — クラスによって定義されたメソッド、またはクラスから拡張されたクラスのいずれかにより継承されたメソッドをコールできます。

クラスからオブジェクトを作成すると、そのクラスのインスタンスが生成されます。インスタンスには、オブジェクトのフィールドが含まれており、オブジェクトのデータつまり状態

を表します。図 1-1 は、姓（lastName）と従業員 ID（ID）の 2 つの属性で定義される Employee クラスの一例です。

図 1-1　クラスとインスタンス



インスタンスを作成すると、その従業員のみに関連する個別の情報が属性に格納されます。つまり、1 つの Employee インスタンスに含まれている情報は、その 1 人の従業員の情報のみです。図 1-1 の例は、Smith と Jones の 2 人の従業員のインスタンスを示しています。各インスタンスには、それぞれの従業員に関する情報が個人別に入っています。

属性

インスタンス内の属性はフィールドで示されます。インスタンス・フィールドは、リレーショナル表の行のフィールドに似ています。クラスでは、フィールドを定義する他、各

フィールドの型も定義します。Java では、フィールドを静的、パブリック、プライベート、プロテクトまたはデフォルト・アクセスとして宣言できます。

- パブリック、プライベート、プロテクトまたはデフォルト・アクセス・フィールドは各インスタンスの中に作成されます。
- 静的フィールドは、情報が Employee クラスのすべてのインスタンスに対して使用可能である点がグローバル変数と似ています。

言語仕様では、すべてのフィールドに対するデータの可視性の規則が定義されています。可視性規則では、これらのフィールドのデータにどのような場合にアクセスできるかが規定されています。

メソッド

クラスでは、そのクラスのインスタンスで起動できるメソッドも定義されます。メソッドは Java で書かれ、オブジェクトの動作を定義するものです。このように状態と動作を一体化することが、あらゆるオブジェクト指向プログラミング言語の機能であるカプセル化の本質です。各従業員の `id` をプライベート・フィールドと宣言して `Employee` クラスを定義すると、メソッドがフィールドを返さなければ、他のオブジェクトはそのプライベート・フィールドにアクセスできません。この例では、オブジェクトは `Employee.getId()` メソッドを起動することで、従業員の ID を取り出すことができます。

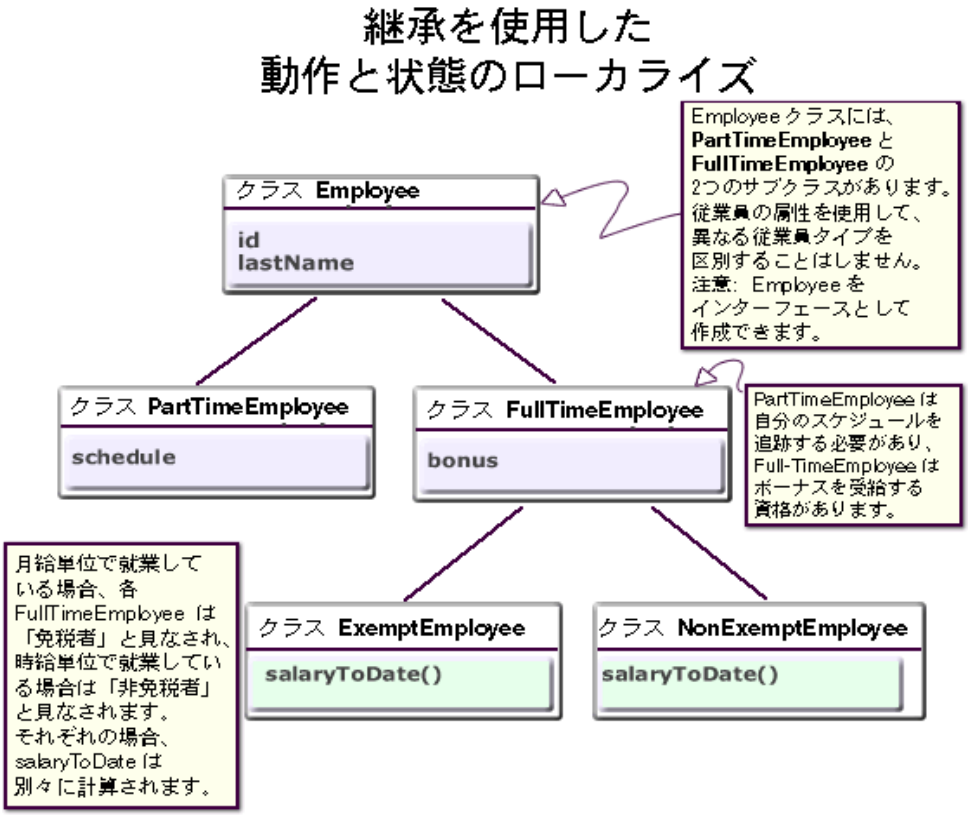
さらに、カプセル化では、`Employee.getId()` メソッドをプライベートと宣言したり、`Employee.getId()` メソッドを書かないことも可能です。カプセル化は、再利用が可能で、かつ誤用されないプログラムの作成に役立ちます。カプセル化により、パブリックと宣言されたオブジェクトの機能のみを公開し、他のフィールドとメソッドをすべてプライベートとして定義できます。プライベート・フィールドおよびメソッドは内部オブジェクト処理に使用できます。

クラス階層

Java では、クラスは大きなクラス階層の中に定義されます。階層の最上部に `Object` クラスがあります。Java のすべてのクラスは、スーパークラスの継承をさかのぼっていくと、必ずどこかのレベルで `Object` クラスを継承しています。クラス B がクラス A から継承しているとすると、クラス B の各インスタンスにはクラス B に定義されたすべてのフィールドとともに、クラス A に定義されたすべてのフィールドが存在します。たとえば、図 1-2 では、`FullTimeEmployee` クラスは `Employee` クラスを継承しているので、このクラスには `Employee` クラスに定義された `id` フィールドと `lastName` フィールドが含まれます。また、`FullTimeEmployee` クラスには、`FullTimeEmployee` にのみ存在するフィールドである `bonus` も追加されます。

クラス B のインスタンスでは、クラス A または B のどちらで定義されたインスタンスのメソッドでも起動できます。Employee クラスの例では、`FullTimeEmployee` インスタンスが起動できるのは、そのクラス内で定義されたメソッド、または `Employee` クラスで定義されたメソッドのみです。

図 1-2 継承階層



クラス B のインスタンスは、クラス A のインスタンスに置き換えることができます。このように継承は、コードの再利用性を向上させる、オブジェクト指向言語の強力な構造体の 1 つです。クラス・ライブラリの既存の機能を利用しながらも、階層内において効果的な動作と状態を定義する新しいクラスを作成できます。

インタフェース

Java では、単一継承のみ可能で、各クラスが継承するクラスは 1 つのみです。複数のソースから継承する必要がある場合は、Java では、インタフェースを通じて複数継承と同等の継承が行えます。その際、複数継承に伴う複雑さや混乱も避けることができます。インタフェースは、クラスに似ていますが、実装ではなく、メソッドのシグネチャの定義のみを行います。メソッドは、インタフェースの実装を宣言したクラスで実装されます。1 つのクラスで同時に多数のインタフェースをサポートする場合に、複数継承が行われます。

ポリモフィズム

Employee の例で、日給を従業員種別ごとに算出する場合を想定します。給与は従業員種別ごとに算出方法が異なります。

- FullTimeEmployees は賞与の支払対象です。
- 残業手当ての支払対象は NonExemptEmployees です。

従来のプロシージャ型言語では、考えられるケースをそれぞれ定義する長い switch 文を記述する必要がありました。

```
switch: (employee.type) {
          case: Employee
            return employee.salaryToDate;
          case: FullTimeEmployee
            return employee.salaryToDate + employee.bonusToDate
         ...
```

新規に Employee の種類を追加する場合は、switch 文を更新する必要がありました。データ構造を変更する場合は、そのデータ構造を使用しているすべての switch 文を変更する必要がありました。Java などのオブジェクト指向言語では、 Employee クラスで定義されていない特殊な処理が必要な、Employee クラスの各サブクラスで、メソッド compensationToDate() を実装します。たとえば、NonExemptEmployee の compensationToDate() メソッドは次のように実装できます。

```
private float compensationToDate() {
        return super.compensationToDate() + this.overtimeToDate();
}
```

また、FullTimeEmployee のメソッドは次のように実装します。

```
private float compensationToDate() {
        return super.compensationToDate() + this.bonusToDate();
}
```

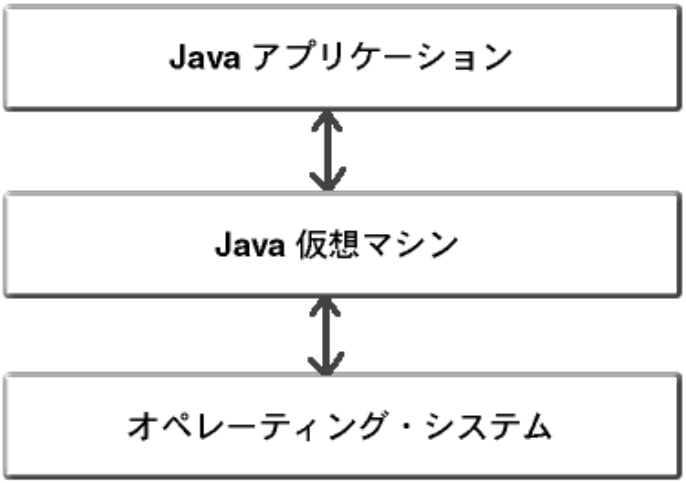
通常、compensationToDate() メソッドを使用すると、異なるクラスで同一のメソッドを起動して、異なる結果を得ることができます。その際、従業員の種類を気にする必要はありません。FullTimeEmployees と PartTimeEmployees を処理するために特別なメソッドを作成する必要はありません。このように異なるオブジェクトが同一メッセージに対してそれぞれ異なる方法で応答することをポリモフィズムと言います。

また、Employee からなにも継承しない新しいクラス Contractor を作成し、このクラスに compensationToDate() メソッドを実装することもできます。現在までの給与合計を計算するプログラムは、フルタイムか、パートタイムか、契約社員かに関係なく、全従業員の給与に対して反復処理を行って、従業員ごとに compensationToDate() メソッドを起動し、その戻り値を合計します。個々の compensationToDate() メソッドに対して変更を行っても、メソッドのコール元はそのままで正しく動作します。たとえば、既存のクラスに新しいフィールドを追加しても問題はありません。

Java 仮想マシン（JVM）

他の高級コンピュータ言語と同様に、Java ソースはマシン命令にコンパイルされます。Java では、これらの命令をバイトコードと呼びます。各命令が必ず 1 バイトの記憶域を占有するためです。C など、他の大半の言語は、マシン固有の命令、つまり Intel Pentium または HP プロセッサ固有の命令にコンパイルされます。Java ソースは、Java 仮想マシン（JVM）と対話する、プラットフォームに依存しない標準のバイトコード群にコンパイルされます。JVM は、Java コードを実行する特定のプラットフォーム向けに最適化された独立したプログラムです。図 1-3 は、Java がプラットフォームからの独立性を保持する仕組みを示します。Java ソースは、プラットフォームに依存しないバイトコードにコンパイルされます。各プラットフォームには、それぞれのオペレーティング・システムに固有の JVM がインストールされています。ソースから生成された Java バイトコードは、JVM を介して、適切なプラットフォーム固有の動作に解釈されます。

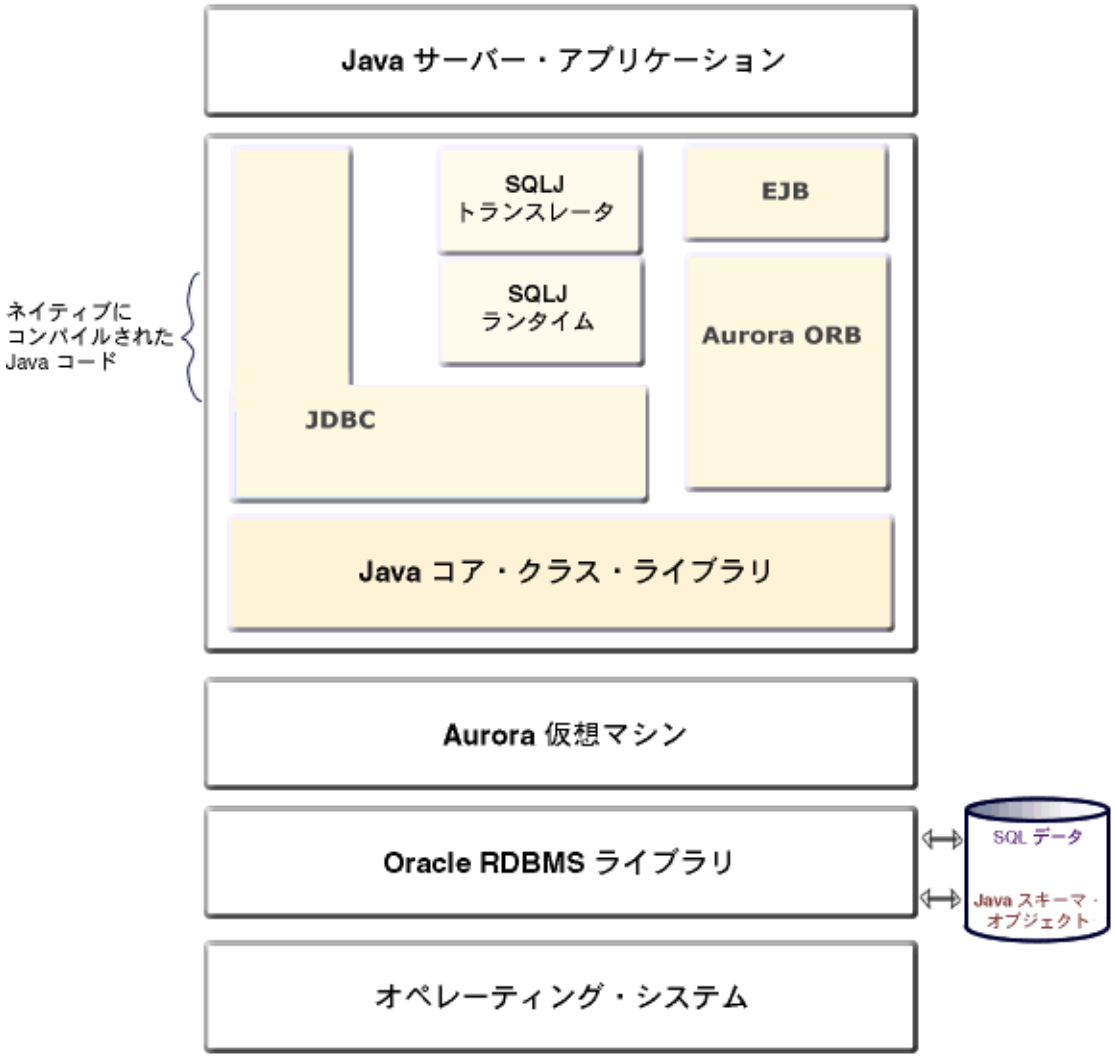
図 1-3　Java コンポーネント構造



Java プログラムの開発時は、Java 言語で書かれている事前定義済みのコア・クラス・ライブラリを使用します。Java のコア・クラス・ライブラリは、複数のパッケージに分割されています。これらのパッケージには、基本言語サポート（`java.lang`）、入出力（`java.io`）、ネットワーク・アクセス（`java.net`）など、共通に使用される機能がまとめられています。JVM とコア・クラス・ライブラリを備えたプラットフォームで作成した Java プログラムは、Java をサポートするあらゆるハードウェアおよびオペレーティング・システムで実行できます。これが、「一度作成すれば、後は実行環境を問わない（write once, run anywhere）」という Java の概念です。

図 1-4 は、Oracle の Java アプリケーションが Java コア・クラス・ライブラリの上にあり、その Java コア・クラス・ライブラリが JVM の上にあることを図解するものです。Oracle の Java サポート・システムはデータベースの中に組み込まれているため、JVM は直接オペレーティング・システムと対話するのではなく、Oracle データベース・ライブラリと対話します。

図 1-4 Oracle8*i* JVM のコンポーネント構造



Sun Microsystems 社は、Java 言語と JVM の仕様を公開しています。Java 言語仕様（JLS）は、構文やセマンティクスなどの定義です。JVM の仕様は、バイトコードを実行するマシンに対する動作要件を定義したものです。また、Sun Microsystems 社では、JVM を実装する際に仕様との適合を判定するための互換性テスト・スイートを提供しています。このテスト・スイートを Java Compatibility Kit（JCK）と言います。Oracle の JVM 実装は JCK に完全に準拠しています。総合的な Java 計画の一環として、ベンダーがあらゆるプラットフォームで Java のサポートを統一できるように、オープン標準が規定されています。また、その標準に準拠しているかを検証する簡単な方法も含まれています。

Java 言語のキーとなる機能

Java 言語には、最適なサーバー・アプリケーション開発を支援するキーとなる機能があります。その機能は、次のとおりです。

- **簡潔性** — Java は、オブジェクト・モデルの使用が徹底されているので、サーバー・アプリケーションで使用される他の大半の言語より簡単に習得できます。豊富なクラス・ライブラリの標準セットには、あらゆるプラットフォームの Java 開発者が使用できる強力なツールが多数用意されています。
- **移植性** — Java はプラットフォーム間で移植可能です。Java では、プラットフォームに依存したコードでも、別のマシンにシームレスに移植できるようなプログラムでも簡単に書くことができます。Oracle Server アプリケーションは、稼動するプラットフォームで直接グラフィカル・ユーザー・インタフェースをサポートしないため、Java の抱えているわずかな移植性の問題をも避けることができます。
- **自動記憶域管理** —Java 仮想マシンは、プログラム実行中に、すべてのメモリーの割当てと解放を自動的に行います。Java プログラマはメモリーの割当ておよび解放を明示的に行えません。かわりに、JVM がこのブックキーピング処理を行います。JVM が、オブジェクトの新規作成時にメモリーを割り当て、オブジェクトが参照されなくなると、メモリーを解放します。後者の処理をガベージ・コレクションと言います。
- **厳密な型指定** — Java の変数を使用するには、その変数で保持するオブジェクトのクラスを宣言しておく必要があります。Java の厳密な型指定は、Java 言語と PL/SQL 言語間のコールに対する安全で妥当なソリューションであり、Java と SQL 間の統合を可能にします。
- **ポインタの不使用** — Java の構文は C 言語によく似ていますが、ダイレクト・ポインタやポインタ操作はサポートしていません。基本型以外のすべてのパラメータを、値ではなく参照で渡します（つまり、オブジェクト ID が保持されます）。Java は C 言語とは異なり、ポインタへの直接アクセスはサポートしていないため、メモリーの破壊やリークを回避できます。
- **例外処理** —Java 例外はオブジェクトです。Java では、開発者が、特定のクラスにおいてメソッドにより発生する例外を宣言する必要があります。
- **柔軟なネームスペース** —Java では、定義したクラスをインターネットのドメインのネームスペースをミラー化した階層構造に保持します。Java アプリケーションを分散しても名前の衝突は起こりません。Java Naming and Directory Interface (JNDI) などの Java 拡

張機能によって、フレームワークが複数のネーム・サービスを連係できます。Java のネームスペースは非常に柔軟なため、言語仕様に完全に準拠しながらも、Oracle はクラス名を解決するスキーマの概念を組み込むことでができます。

- **セキュリティ** —Java バイトコードと仮想マシンには、Java バイナリ・コードが改ざんされていないかを検証する機構が組み込まれています。Oracle8*i* と一緒にインストールされる Security Manager のインスタンスは、Oracle データベースのセキュリティと併用して、Java メソッドを起動できるユーザーを制限します。
- **リレーショナル・データベースへの接続基準** —JDBC と SQLJ により、Java コードのリレーショナル・データベース内のデータに対するアクセスや操作が可能になります。Oracle は、リレーショナル・データベースにアクセスするためのドライバを提供していますが、これらはベンダーに依存しない、移植性の高い Java コードを実現しています。

Oracle8*i* で Java を使用する理由

開発者が、Java アプリケーションを作成してデータベースにロードできるのは、Java が安全な言語であるためです。Java は、Java コードが常駐するオペレーティング・システムを改ざんから保護するように開発されています。C などの言語はデータベースで問題を引き起こす恐れがあります。Java はデータベース内で使用できる安全な言語として設計されています。

Java は開発者にとって利点の多い言語ですが、Java サーバー・アプリケーションをスケーラブルにサポートする JVM の実装は簡単ではありません。この JVM 実装について説明します。

- マルチスレッド
- 自動記憶域管理
- フットプリント
- パフォーマンス
- クラスの動的ロード

マルチスレッド

マルチスレッドのサポートは、Java 言語の主要なスケーラブルな機能の 1 つです。たしかに、Java 言語とクラス・ライブラリにより、Java では他の言語に比べ簡単にマルチスレッド・アプリケーションを作成できるようになりました。しかし、信頼性が高くスケーラブルなマルチスレッド・コードの作成は、依然としてどの言語でも簡単ではありません。

Oracle8*i* はデータベース・サーバーとして、数千人のユーザーの作業を効率的にスケジューリングします。Oracle8*i* の Java 仮想マシン Aurora は、RDBMS サーバーの機能を使用して、数千人のユーザーに対する Java の実行を同時にスケジューリングします。Oracle8*i* は、Java 言語仕様（JLS）および Java 互換性キット（JCK）に必要な Java 言語レベルのスレッドをサポートしていますが、データベースの有効範囲内でスレッドを使用しても拡張性が向上するわけではありません。データベースに組み込まれた拡張性を利用することで、マルチスレッ

ド対応の Java サーバーを作成する必要がなくなります。データベースの機能を使用して、シングル・スレッドの Java アプリケーションを作成してください。アプリケーション間のスケジューリング処理はデータベースが行うため、スレッドを管理することなく拡張性を実現できます。マルチスレッド対応の Java アプリケーションを作成することは可能ですが、複数の Java スレッドによってシステムのパフォーマンスが向上することはありません。

Java でマルチスレッド機能を利用する上での問題の 1 つに、スレッドと自動記憶域管理、つまりガベージ・コレクション機能との相互干渉があります。汎用 JVM で実行するガベージ・コレクタは、どの Java 言語スレッドが実行しているのか、基礎となるオペレーティング・システムがそれらのスレッドをどのようにスケジューリングしているのかは関知しません。

- 非 Oracle8*i* モデル — 単一ユーザーが単一の Java 言語レベル・スレッドにマッピングされます。つまり、1 つのガベージ・コレクタですべてのユーザーのガベージが処理されます。通常、オブジェクトの存続期間別およびサイズ別に、それぞれ別の手法でオブジェクトの割当てと収集が行われます。この結果、アプリケーションが過度にマルチスレッド化され、最悪の場合は、ネイティブ・スレッドのサポートをオペレーティング・システムに過度に頼るようになります。このようなアプリケーションは、信頼性が低く、拡張性が極度に制限されたものになります。このような実装で高度な拡張性を実際に達成できた例はありません。
- Oracle8*i* JVM モデル — 多くのユーザーがサーバーに接続し、同じ Java コードを実行しても、各ユーザーは自分専用の Java コードを自分専用の Java 仮想マシンで実行しているように感じます。Oracle8*i* JVM の役割は、Oracle RDBMS のスケーラブルなアプローチを使用してオペレーティング・システム・プロセスとスレッドを利用することです。このアプローチにおいて、JVM のガベージ・コレクタは、どの時点でも対象を 1 人のユーザーに限定するため、信頼性と効率性が向上します。Oracle8*i* JVM でのスレッド・モデルでの実装に関する詳細は、2-41 ページの「Oracle8i JVM におけるスレッド」を参照してください。

自動記憶域管理

ガベージ・コレクションは、Java の自動記憶域管理の主要な機能であり、これによって Java 開発者はメモリーの割当てと解放を明示的に行う必要がなくなります。これにより、C および C++ プログラムでよく発生するメモリー・リークの大きな原因を排除できます。ただし、ガベージ・コレクションにも問題はあります。ガベージ・コレクションを行うと、プログラムの実行速度とフットプリントにオーバーヘッドがかかります。ガベージ・コレクションの長所と短所を書いた論文は多数ありますが、ガベージ・コレクションに代わりうる他の機能に比べると、総合的にガベージ・コレクションの方が優れています。

拡張性が高く、高速な Java プラットフォームの提供を目標とするとき、ガベージ・コレクションは JVM の開発において 1 つの課題となります。Aurora JVM は、次のような方法でこのような課題に対応します。

- Aurora JVM は、多数のユーザーを効率的に管理できる Oracle8*i* スケジューリング機能を使用します。

- ガベージ・コレクションは単一セッション内の単一ユーザーのみを対象として実行されるため、ユーザー数が増えても高いパフォーマンスを維持できます。Oracle8*i* の Aurora JVM には、ユーザー数が増加しても、メモリー・マネージャの処理の負荷と複雑度が増加しないという大きな利点があります。メモリー・マネージャによる単一セッション内のオブジェクトの割当てと回収は、通常、単一ユーザーの活動に変換されます。
- Aurora JVM は、使用されるメモリーのタイプに応じて異なるガベージ・コレクション・テクニックを使用します。これらのテクニックにより、高い効率性と低いオーバーヘッドが実現します。

フットプリント

Java プログラムの実行時のフットプリントは、複数の要因によって決まります。

- プログラム自体のサイズ － クラスとメソッドの数およびそのコード量。
- プログラムの複雑度 － プログラム自体ではなく、プログラムの実行時に Aurora で使用されるコア・クラス・ライブラリの総数。
- Aurora が使用する状態の総数 － Aurora が割り当てるオブジェクトの数、オブジェクトのサイズおよび次のコールまで保持する必要があるオブジェクトの数。
- 実行中のプログラムの要求を処理するガベージ・コレクタとメモリー・マネージャの能力。これは多くの場合、決定的要因にはなりません。この要因の重要性は、オブジェクトの割当て速度と、他のオブジェクトによって保持される方法に影響されます。

拡張性の面から考えると、同時に多数のクライアントをサポートするためのキーは、ユーザー当たりのセッション・フットプリントを最小限にすることです。Aurora では、ユーザーあたりのセッション・フットプリントを最小化する手段として、Java バイトコードなど、ユーザーに対するすべての読取り専用データを共有メモリーに格納します。コール・メモリーおよびセッション・メモリーには適切なガベージ・コレクション・アルゴリズムが適用され、ユーザー・セッションのフットプリントが抑えられます。Aurora では 3 種類のガベージ・コレクション・アルゴリズムを使用して、ユーザーのセッション・メモリーを制限します。

- 存続期間の短いオブジェクトについては世代別にメモリーを回収します。
- 単一コールの間のみ存続するオブジェクトには、Mark and lazy Sweep を使用します。
- 存続期間の長いオブジェクト（1 つのセッション内で複数のコールにわたって存続するオブジェクト）にはコピー・コレクタを使用します。

パフォーマンス

Oracle8*i* JVM のパフォーマンスは、ネイティブ・コンパイラを実装することによって向上します。

ネイティブ・コンパイラによるパフォーマンス向上の仕組み

Java は、JVM 上においてプラットフォームに依存しないバイトコードを実行します。そのかわりに、JVM がハードウェア・プラットフォーム別の処理を行います。ソフトウェアにレベルを追加するたびにパフォーマンスは低下していきます。Java はプラットフォームに依存しないバイトコードを直接解釈できないため、C 言語などのプラットフォーム固有の言語と比べると Java アプリケーションの効率は劣ります。このため、複数の JVM のサプライヤがネイティブ・コンパイラを提供しています。ネイティブ・コンパイラは Java バイトコードをプラットフォームに依存するネイティブ・コードに変換します。これにより、途中の解釈処理が不要になり、パフォーマンスが向上します。ネイティブ・コンパイルの 2 つの方法は次のとおりです。

コンパイラ	説明
Just In Time (JIT) コンパイル	JIT コンパイラは、実行時に Java バイトコードを即座にネイティブな（プラットフォーム固有の）マシン・コードにコンパイルします。この処理では、プラットフォーム上で実行するための実行可能ファイルは作成されず、そのかわり、変換後即座に実行できる、プラットフォームに依存しないコードが Java バイトコードから生成されます。実行頻度が高い Java コードにこの方法を使用すれば、C 言語並みの速度で実行できます。
静的コンパイル	静的コンパイルでは、実行前に Java バイトコードをプラットフォームに依存しない C コードに変換します。次に、標準の C コンパイラで C コードがターゲート・プラットフォームの実行可能ファイルにコンパイルされます。この方法は、更新頻度の低い Java アプリケーションに適しています。この方法は、最近の C コンパイラで採用されている成熟した効率的なプラットフォーム固有のコンパイル技術を利用するものです。

Oracle8*i* では、静的コンパイルを用いて、Java のコア・クラス・ライブラリ、Aurora/ORB および JDBC コードをネイティブ形式にコンパイルして配布しています。この方式が Oracle のサポートするすべてのプラットフォームに適用できるのに対して、JIT 方式では、プロセッサに依存した低水準コードをプラットフォームごとに作成および保守する必要があります。このネイティブ・コンパイル・テクノロジは、作成した Java コードと一緒に使用できます。詳細は、6-2 ページの「ネイティブ・コンパイル・コード」を参照してください。

クラスの動的ロード

Java のもう 1 つの強力な機能はクラスの動的ロードです。クラスがプログラム実行中使用されている場合に限り、クラス・ローダーはディスクからクラスをロードします（そして、解釈実行するために JVM 固有のメモリー構造に格納します）。クラス・ローダーはプログラムの実行中にクラスを `CLASSPATH` で検索してロードします。この方法はアプレットには向いていますが、サーバー環境では次のような問題をもたらします。

問題	説明	解決法
予測可能性	初回実行時は、クラスがロードされるので、処理速度が極端に低下します。単純なプログラムでも、そのプログラムで必要な多数のコア・クラスが Aurora によってロードされることがあります。Aurora によってロードされるクラスの数は、プログラマが簡単に予測または判定することはできません。	Aurora は、他の Java 仮想マシンと同じように、クラスを動的にロードします。1 回のクラスのロード速度は同じです。ただし、Aurora はクラスを共有メモリーにロードするので、他のユーザーは同じクラスを再度ロードする必要がありません。他のユーザーは、ロード済みのクラスを単に使用するだけです。
信頼性	クラスの動的ロードの利点は、プログラムの更新がサポートされることです。たとえば、サーバー上のクラスを更新すると、クライアントはプログラムの次回使用時にそのプログラムをダウンロードし動的ロードを行うことによって、更新されたクラスを使用できるようになります。サーバー・プログラムは信頼性が重要です。開発者は、指定したプログラム構成がすべてのクライアントによって実行されるようにする必要があります。開発者が意図していないクラスがクライアントによってロードされないようにします。	Oracle8*i* では、アップロードおよび解決処理がクラスのロード処理から切り離されており、実行時には行われません。作成した Java コードをサーバーにアップロードするには、`loadjava` ユーティリティを使用します。`CLASSPATH` を使用するかわりに、インストール時にリゾルバを指定します。リゾルバは `CLASSPATH` によく似ていますが、クラスが常駐するスキーマを指定できます。このように解決処理をクラスのロード処理から切り離すと、どのプログラムをユーザーが実行しているのかを常に知ることができます。loadjava とリゾルバの詳細は、『Oracle8*i* Java Tools リファレンス』を参照してください。

Oracle の Java アプリケーションの方針

Java の価値の 1 つは、広く使用されていることであり、Java アプリケーションの開発ができるプログラマの数が増加していることにあります。Oracle は、エンタープライズ・アプリケーション開発者に Java アプリケーションを作成、配置および管理するためのあらゆる規模の Java ソリューションを提供しています。つまり、クライアント側およびサーバー側のプログラム・インタフェース、Java 開発をサポートするツール群、および Oracle8*i* データベース・サーバーに統合された Java 仮想マシンなどです。これらの製品はすべて Java 標準に 100% 準拠しています。

Aurora JVM に加え、Oracle8*i* の Java プログラミング環境は次のもので構成されます。

- Java に匹敵し、PL/SQL と連携する Java ストアド・プロシージャ。Java ストアド・プロシージャは PL/SQL に密接に統合されています。Java ストアド・プロシージャは PL/SQL パッケージからコールでき、PL/SQL プロシージャは Java ストアド・プロシージャからコールできます。
- JDBC および SQLJ プログラミング・インタフェースを通じてアクセスできる SQL データ。
- Object Request Broker（Aurora/ORB）と Enterprise JavaBeans のサポートを通じた分散エンタープライズ・アプリケーション開発。
- サーブレットと JavaServer Pages を介した動的 HMTL ページ。

- 開発、クラスのロードおよびクラス管理に役立つツールとスクリプト。

使用する Java API の決定には、次の表を参考にしてください。

必要な機能のタイプ	使用する Java API
トリガーなどの Java プロシージャを SQL からコールする。	Java ストアド・プロシージャ
Java オブジェクトから既知の列名を持つ既知の表をもとに静的で単純な SQL 文をコールする。	SQLJ
Java オブジェクトから動的で複雑な SQL 文をコールする。	JDBC
HTML ページから Java を起動する。	サーブレットまたは JavaServer Pages
複数層 Java アプリケーションを作成する。	CORBA または EJB

Java ストアド・プロシージャ

PL/SQL プログラマが Java を利用する場合は、Java ストアド・プロシージャに関わることになります。Java ストアド・プロシージャは Java で作成され、PL/SQL ストアド・プロシージャと同じようにサーバーで実行されるプログラムです。Java ストアド・プロシージャは、SQL*Plus のような製品から直接コールすることも、Net8 クライアント（OCI、PRO*、JDBC または SQLJ）からトリガーで間接的にコールしてアクセスすることもできます。『Oracle8*i* Java ストアド・プロシージャ開発者ガイド』では、Java でのストアド・プロシージャの作成方法、PL/SQL からの Java ストアド・プロシージャへのアクセス方法、Java からの PL/SQL 機能へのアクセス方法を説明しています。

また、PL/SQL を使用せずに、Java のみを使用してプログラムを開発することもできます。Oracle8*i* によって、Java プログラミング言語および JVM に完全対応した実装が実現されます。

PL/SQL の統合と Oracle RDBMS 機能

既存の PL/SQL プログラムは Java から起動できます。また、Java プログラムは PL/SQL から起動できます。このソリューションは、既存の投資を損わずに利用しながら、Java ベースのインターネット・コンピューティングの利点と機会をもたらします。

Java 開発者は、Oracle の 2 種類のアプリケーション・プログラミング・インタフェース（API）を使用して、SQL データ（JDBC と SQLJ）にアクセスできます。クライアントとサーバーでこの 2 種類の API を利用できるので、同じコードをどちら側でも配置できます。

- JDBC ドライバ — クライアント / サーバー型の 2 階層アプリケーションの作成に使用。
- SQLJ（Java 埋込み SQL）— 静的 SQL へのアクセスに使用。列名が分かっていることが必要です。

JDBC ドライバ

JDBC はデータベース・アクセス・プロトコルであり、これを使用することによってデータベースに接続し、SQL 文を準備および実行できます。Java のコア・クラス・ライブラリには、JDBC API が 1 つのみあります。ただし、JDBC は、ベンダーが個々のデータベースに特化したドライバを提供できるように設計されています。Oracle8*i* JVM には、次のような 3 種類の JDBC ドライバが組み込まれています。

ドライバ	説明
JDBC Thin ドライバ	JDBC の thin ドライバを使用すると、Oracle SQL データにアクセスするアプリケーションとアプレットを 100%、Java で作成できます。JDBC の thin ドライバは、特に Web ブラウザ・ベースのアプリケーションとアプレットに向いています。他の Java アプレットと同じように、web ページから動的にダウンロードできるためです。
JDBC Oracle Call Interface ドライバ	JDBC Oracle Call Interface（OCI）ドライバを使用すると、クライアント層または中間層にある Oracle 固有のネイティブ・コード（つまり非 Java コード）ライブラリにアクセスできます。このドライバは、JDBC Thin ドライバに比べて、機能が豊富であり、高パフォーマンスですが、サイズが大きく、クライアント側でインストールが必要な点が難点です。
サーバー側 JDBC 内部ドライバ	Oracle8*i* は、サーバーで Java コードを実行するときに Oracle8*i* JVM のサーバー側内部ドライバを使用します。これによって、サーバーの Java 仮想マシンで実行している Java アプリケーションは、JDBC を使用してローカル（つまり、同一マシンおよび同一プロセス）に定義されているデータにアクセスできます。このドライバは Oracle RDBMS ライブラリを直接使用できるため、このドライバを使用することによりパフォーマンスがさらに向上します。それは、Java コードと SQL データ間のネットワーク接続のオーバーヘッドがかからないためです。Oracle 8*i* では、サーバー上で同じ Java-SQL インタフェースがサポートされるので、コードの配置時にコードを作成し直す必要がありません。

JDBC の詳細は、3-10 ページの「SQLJ および JDBC を使用したデータベースの問合せ」または『Oracle8*i* JDBC 開発者ガイドおよびリファレンス』の詳しい説明を参照してください。

SQLJ（Java 埋込み SQL）

JDBC は、Java から SQL データにアクセスするための API を提供します。JDBC によって、SQL に対応する Java クラスが導入されます。オラクル社は IBM 社、Tandem 社、Sybase 社、Sun Microsystems 社などのベンダーと協力して、SQL 文を Java プログラムに埋め込む標準的な方法、つまり SQLJ を開発しました。SQLJ は、JDBC より簡単で生産性の高いプログラミング API の新しい規格（ANSI x.3.135.10-1998）です。ユーザーはこの API に合せてアプリケーションを作成した後、標準プリプロセッサを使用して、そのプログラムを JDBC コールを使用した標準 Java ソースに変換できます。実行時、このプログラムは、標準 JDBC

ドライバを使用して、多様なベンダーのデータベースとやり取りできます。SQLJ は、Java からデータベースにアクセスするクライアント側アプリケーションおよび中間層アプリケーションを開発するための、簡単かつ強力な手段です。SQLJ は、Oracle8*i* JVM 環境におけるストアド・プロシージャ、トリガー、メソッドにおいて、また EJB と CORBA とともに使用できます。また、SQLJ と JDBC を組み合せることもできます。

SQLJ トランスレータは、Java ソース・コードの中の埋込み SQL を純粋な JDBC ベースの Java コードに変換する Java プログラムです。Oracle8*i* JVM によって完全な Java 環境が提供されるので、Oracle8*i* JVM で実行する SQLJ プログラムをクライアント上でコンパイルできるばかりでなく、サーバー上で直接コンパイルすることもできます。Oracle8*i* は、インターネット標準に準拠しているため、開発スタイルは自由に選択できます。

SQLJ の詳細は、3-10 ページの「SQLJ および JDBC を使用したデータベースの問合せ」または 『SQLJ 開発者ガイドおよびリファレンス』の詳しい説明を参照してください。

Java を起動する動的 HTML

Java は、サーブレットまたは JavaServer Pages などの動的 HTML プロトコル内から起動できます。

サーブレット

サーブレットは、特に Web サーバー・コンテキスト内で、HTML と Java アプリケーション間のリクエストを管理する場合に便利です。サーブレットは、サーバーに対して、ブラウザに対するアプレットのような役割を果たします。Java と HTML ベースのプロトコルはプラットフォームに依存しないので、サーブレットは複数の環境で使用できます。

詳細は、『Oracle8*i* Oracle Servlet Engine ユーザーズ・ガイド』を参照してください。

JavaServer Pages

JavaServer Pages は、Web サーバー上で実行されるアプリケーションなどの Web アプリケーションによって出力されるページで動的な内容を生成する方法です。この方法によって、Web ページの HTML コード（または XML などのマークアップ・コード）内から外部の Java コンポーネントのコールや Java コードを挿入できます。JavaServer Pages は、JavaBeans および Enterprise JavaBeans（EJB）のビジネス・ロジックと動的な機能のフロントエンドとして効率的に動作します。また、標準の HTML ページに挿入できるものはすべて、JSP ページにも挿入できます。

たとえば、JSP ページでは、JavaBean または Enterprise JavaBean などのコンポーネントをコールできます。Bean は通常、JDBC または SQLJ を介してデータベースに直接または間接的にアクセスします。

JSP ページは、実行される前に Java サーブレットに変換されます。変換された Java サーブレットにより、HTTP リクエストが処理され、他のサーブレットと同様に応答が生成されます。JSP ページは、サーブレットと完全に相互運用できます。JSP ページには、サーブレット

からの出力またはサーブレットへの転送を組み込むことができ、サーブレットには、JSP ページからの出力または JSP ページへの転送を組み込むことができます。

詳細は、『Oracle8*i* JavaServer Pages 開発者ガイドおよびリファレンス』を参照してください。

分散アプリケーションの開発

Oracle8*i* JVM は、従来の RDBMS ストアド・プロシージャをサポートするばかりでなく、CORBA 2.0 ORB が組み込まれていますし、Enterprise JavaBeans（EJB）もサポートします。CORBA と EJB を併用すると、Java コンポーネントとアプリケーション・ロジックをクライアント層、中間層およびデータベース・サーバー層に分散できます。

OMG CORBA ORB	ORB によって、どの言語で書かれたプログラムでも、Object Management Group（OMG）によって定義された標準ワイヤー・プロトコルである Internet Inter-ORB Protocol（IIOP）を介して、Oracle8*i* データベースとやり取りできます。
Enterprise JavaBeans（EJB）	100%pure Java アプリケーションの場合は、EJB を標準フレームワークとして、コンポーネント・ベースの安全なトランザクション・アプリケーションを Oracle8*i* JVM で配置できます。

EJB コンポーネントの使用

1-2 ページの「Java およびオブジェクト指向プログラミングの用語」では、オブジェクト指向プログラミングの大きな特徴の 1 つとしてカプセル化を説明しています。各オブジェクトは固有のプライベートな状態を保持し、メソッドとして実装した一連の動作をサポートします。Java では、JavaBeans を使用したコンポーネントの定義形式が決まっています。JavaBean コンポーネントは、再利用可能な単一オブジェクトまたはオブジェクト・グループ（厳密には、オブジェクト・グラフ）であり、なんらかのビルダー・ツールで操作できます。JDeveloper などの統合開発環境（IDE）には、JavaBeans を使用するためのユーザー・インタフェースを構築し、JavaBean コンポーネントを作成するためのツールが用意されています。Bean ごとに、操作可能なパブリック・インタフェースとプロパティを指定します。JavaBeans は、ビジュアル指向コンポーネントでなくてもかまいません。実質的には、どのような Java プログラム抽象化でも Bean として扱い、操作することができます。

大規模なコンポーネント・ライブラリを利用することによって、事前に作成され、テストされた構成要素からアプリケーションを作成できます。ただし、Bean ではトランザクション・ロジックを含む複雑なビジネス・アプリケーションの作成に制限があります。この制限に対処するために、オラクル社、Sun Microsystems 社、IBM 社などが協力して、Enterprise JavaBeans（EJB）仕様を策定しました。EJB によって、コンポーネントでのトランザクションとセキュリティの扱い方を指定するための宣言方式が導入されています。Oracle8*i* での EJB コンポーネントの使用の詳細は、『Oracle8*i* Enterprise JavaBeans 開発者ガイドおよびリファレンス』または『Oracle8*i* CORBA 開発者ガイド』のマニュアルを参照してください。

JavaBeans および Enterprise JavaBeans に相当するコンポーネント・モデルとして、Microsoft の COM および COM+ モデルが有名です。既存の Microsoft の COM に対応した

アプリケーションがある場合は、ブリッジ製品を介して、JavaBeans や EJB などのオープン・インターネット標準製品と連係できます。ブリッジ製品は複数のベンダーから入手できます。

開発ツール

Java を Oracle8*i* Server に導入することによって、各種の Java 統合開発環境を使用できるようになります。Oracle8*i* JVM は Java との互換性や、オープン・インターネット標準およびプロトコルに準拠しているため、Java プログラムは Oracle8*i* JVM 上に配置することにより実行できます。Oracle8*i* JVM には、Java サーバー・アプリケーションの開発と配置を簡単にするためのツールまたはユーティリティが多数あり、すべて Java で書かれています。またオラクル社の JDeveloper には、Java ストアド・プロシージャと Enterprise JavaBeans の配置を容易にするための特別な機能が多数あります。

Oracle8*i* Java マニュアルの概要

このマニュアルは、Oracle8*i* Java 開発者向けの入門書です。このマニュアルでは、Aurora JVM の特徴も含め、Oracle8*i* における Java プログラミングの特徴的な機能について、その利点のプログラミングでの活用方法とともにいくつか紹介します。

Oracle8*i* データベースでの Java 開発の基本が理解できると、Java アプリケーションの実装に使用するプロトコルの情報が必要になります。次のリストは、マニュアル・セットの中から、このマニュアル以外の、アプリケーション開発に役立つ書籍をリストしたものです。

プロトコル	説明	書籍名
JDBC	Oracle8*i* Java 開発者は、オラクル社の製品 Java データベース・コネクティビティ（JDBC）を使いこなせるようになる必要があります。それは、JDBC がこの Java 標準に対する Oracle 固有の拡張であるということのみでなく、Java プログラムから SQL データにアクセスするための基礎でもあるためです。JDBC は業界標準です。	『Oracle8*i* JDBC 開発者ガイドおよびリファレンス』
SQLJ	Java 埋込み SQL（SQLJ）を使用すると、SQL データにアクセスする Java プログラムの開発が容易になります。SQLJ は、Java で書かれたプリプロセッサを使用して、埋込み SQL 文を標準 JDBC スタイルのプログラムに変換します。SQLJ は業界標準です。	『Oracle8*i* SQLJ 開発者ガイドおよびリファレンス』
JPublisher	JPublisher は、既存の Oracle リレーショナル・データベース表にアクセスする Java プログラムを作成するための、使いやすい便利なツールを提供します。	『Oracle8*i* JPublisher ユーザーズ・ガイド』

プロトコル	説明	書籍名
Java ストアド・プロシージャ	PL/SQL プログラマが Java を利用する場合は、Java ストアド・プロシージャが役立ちます。Java ストアド・プロシージャは Java で作成され、PL/SQL ストアド・プロシージャと同じようにサーバーで実行されるプログラムです。Java ストアド・プロシージャは、SQL*Plus のような製品から直接コールすることも、Net8 クライアント（OCI、PRO*、JDBC または SQLJ）からトリガーで間接的にコールしてアクセスすることもできます。『Oracle8*i* Java ストアド・プロシージャ開発者ガイド』には、ストアド・プロシージャを Java で書く方法、PL/SQL から Java ストアド・プロシージャにアクセスする方法、および Java から PL/SQL 機能にアクセスする方法が説明されています。 また、PL/SQL を使用せずに、Java のみを使用してプログラムを開発することもできます。Oracle8*i* は、Java プログラミング言語および JVM に完全対応した実装を提供します。	『Oracle8*i* Java ストアド・プロシージャ開発者ガイド』
JavaServer Pages とサーブレット	データベース上にあるか、または Web サーバーと対話するバックエンド Java を起動する動的 HTML プロトコル	『Oracle8*i* Oracle Servlet Engine ユーザーズ・ガイド』 『Oracle8*i* JavaServer Pages 開発者ガイドおよびリファレンス』
EJB と CORBA	分散アプリケーションには、ORB または EJB テクノロジが活用されます。Object Request Broker（Aurora/ORB）と Enterprise JavaBeans（EJB）には Oracle のオープンな分散オブジェクト・テクノロジが組み込まれています。Aurora/ORB と EJB によって、業界標準に準拠した強力なフレームワークとツールが提供されるため、イントラネットまたはインターネットにおいて、Oracle データに透過的なトランザクション・アクセスをするスケーラブルな Java アプリケーションを作成できます。	『Oracle8*i* Enterprise JavaBeans 開発者ガイドおよびリファレンス』 『Oracle8*i* CORBA 開発者ガイド』

2

Oracle8*i* での Java アプリケーションの作成

Oracle8*i* JVM では標準の Java アプリケーションを実行できます。ただし、データベース・サーバーの中に Java クラスが組み込まれているため、一般的な Java 開発環境とは異なる点があります。この章では、Oracle8*i* における Java アプリケーションの作成、インストールおよび配置の基本的な違いを説明します。

- 概要
- Java アプリケーションにインポーズされるデータベース・セッション
- 実行制御
- JDK 1.1 から Java 2 への移行
- Java のコード、バイナリおよびリソースの格納
- Java クラス・メソッドの実行準備
- サーバー上のユーザー・インタフェース
- クラスの短縮名
- Oracle8i JVM の Class.forName()
- オペレーティング・システム・リソースの管理
- Oracle8i JVM におけるスレッド

注意： Oracle8*i* JVM の異なった API の利用方法を十分に理解するために、詳細情報をドキュメントで参照してください。この章では、総合的な観点から Java API 間の連係、および Oracle8*i* JVM 環境での API の使用方法を詳細に説明します。

概要

第 1 章で説明したように、Oracle8*i* JVM プラットフォームは、100% pure Java アプリケーションを実行できる標準の Java 対応環境です。Oracle が実装したこのプラットフォームは、Java 言語仕様および Java 仮想マシン仕様に準拠しています。このプラットフォームは、標準 Java バイナリ形式と標準 Java API をサポートします。さらに、Oracle8*i* は、実行時のクラスの動的ロードなど、標準の Java 言語のセマンティクスに準拠しています。ただし、Oracle8*i* JVM は、他の Java 環境と異なり、Oracle8*i* RDBMS に埋め込まれているので、新しい概念が多数導入されています。ここでは、Sun Microsystems の JDK 環境と、Java を Oracle8*i* データベースに組み込んだ場合の環境の違いを概説します。

用語

用語	**定義**
Oracle8*i* JVM	Java 対応 Oracle8*i* データベース・サーバー
Aurora	Oracle8*i* JVM
セッション	Java コードを実行するユーザーは、サーバー上でセッションを確立する必要があります。ここでは、セッションという用語を Oracle（または他のデータベース・サーバー）での標準的な意味で使用しています。通常、1 人のユーザーがサーバーに接続している期間が 1 つのセッションになります。
コール	ユーザーがセッション内で Java コードを実行しようとすることを、コールと言います。コールの開始方法は数通りあります。 ■ SQL クライアント・プログラムからの、Java ストアド・プロシージャの実行 ■ トリガーからの Java ストアド・プロシージャの実行 ■ PL/SQL プログラムからの Java コードのコール ■ CORBA クライアントからの CORBA オブジェクトのメソッドの起動 ■ EJB クライアントからの EJB オブジェクトのメソッドの起動 どの場合でも、コールが開始し、Java コード、SQL コードまたは PL/SQL コードのいずれかの組合せが実行、完了され、コールが終了します。

標準の Java 環境では、`java <classname>` を実行することによって、インタプリタを通じて Java アプリケーションを実行します。これにより、アプリケーションはオペレーティング・システム上のプロセスで実行されます。

Aurora JVM を使用した場合、アプリケーションをデータベースにロードし、インタフェースを公開した後に、データベース・セッションでアプリケーションを実行する必要があります

す。このマニュアルでは、データベース内で Java アプリケーションを実行する方法を説明します。特に、データベース内で Java を使用する方法は次の項を参照してください。

- 実行前に Java アプリケーションをロードおよび公開する方法－ 2-13 ページから始まる「Java のコード、バイナリおよびリソースの格納」および「Java クラス・メソッドの実行準備」を参照してください。
- データベース・セッション内で実行する方法－ 2-3 ページの「Java アプリケーションにインポーズされるデータベース・セッション」を参照してください。

さらに、標準の Java に含まれる機能の中には、データベース・セッションでアプリケーションを実行した場合に変化する機能があります。これらの機能は、次の項で説明します。

- 実行制御
- サーバー上のユーザー・インタフェース
- クラスの短縮名
- Oracle8i JVM の Class.forName()
- オペレーティング・システム・リソースの管理
- Oracle8i JVM におけるスレッド

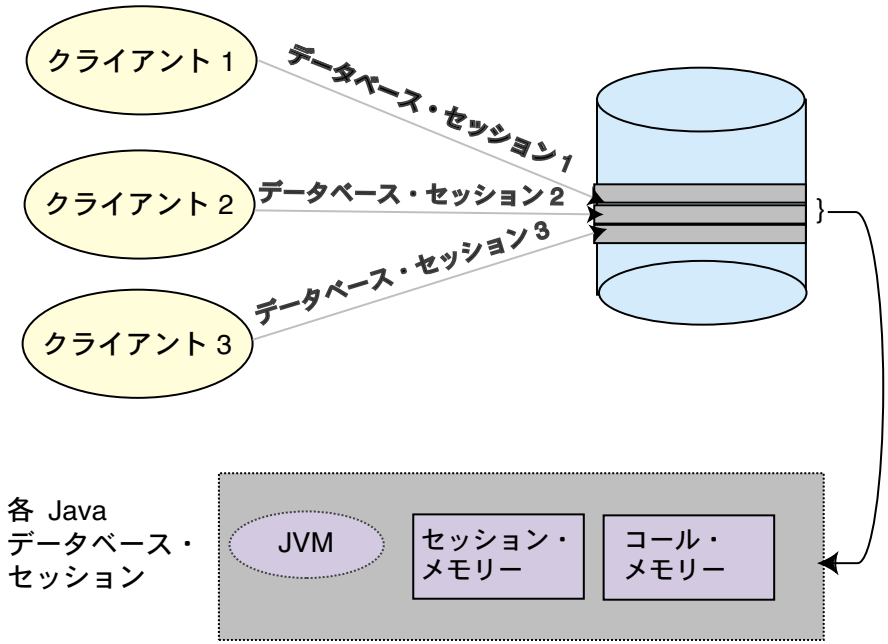
この章の内容を理解した後、第 3 章「データベースでの Java のコール」でクライアントの設定方法と、様々なタイプの Java アプリケーションを起動する例を参照してください。

Java アプリケーションにインポーズされるデータベース・セッション

Java を Oracle8*i* データベースに組み込むことで、Java 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に存在するようになります。Oracle8*i* JVM セッションは、従来の Oracle セッションとまったく同じ意味を持ちます。Oracle8*i* JVM セッションはそれぞれ、そのセッション内のコール全般にわたってクライアントの Java の状態を保持します。

図 2-1 で説明しているように、各 Java クライアントはデータベース内で Java を実行する環境としてデータベース・セッションを起動します。ガベージ・コレクション、セッション・メモリーおよびコール・メモリーは、そのセッション中のクライアントごとに存在します。

図 2-1 各データベース・セッション内の Java 環境



1 つのセッションのコンテキストで、クライアントは次のことを実行します。

1. データベースに接続してセッションを開きます。
2. データベース内で Java を実行します。これはコールとして実行されます。
3. 必要に応じてコールを実行しながら、そのセッション内で引き続き処理を続けます。
4. セッションを終了します。

1 つのセッション内において、クライアントは他のクライアント環境とは切り離された固有の Java 環境を持ちます。クライアントからは、セッションごとにそれぞれ個別の Java 仮想マシンが起動されるように見えますが、実際の実装は、はるかに効率的です。1 つのセッションで、Aurora JVM は開発者に代わってデータベース内の拡張性を管理します。1 つのクライアントから実行されたコールは、他のクライアントのコールとは別個にそれぞれのセッションで処理されます。Aurora JVM は、クライアント間における共有読取り専用データを最大化し、セッションごとの増分フットプリントを最小限に抑えることにより、複数クライアントのパフォーマンスを最大化します。

セッション、ネットワーク、状態およびその他の共有リソース管理の問題に関する詳細は、基礎となるサーバー環境によって、Java サーバー・コードから隠されます。静的変数はクライアントに対してすべてローカルです。セッション境界にまたがってメモリーを使用するこ

とができないため、クライアントは他のクライアントの静的変数にはアクセスできません。各クライアントは、それぞれのセッションでそのコールを実行するので、各クライアントのアクティビティは他のクライアントからは切り離されます。コール中は複数のクラスの静的フィールドにオブジェクトを格納でき、次のコールでもこの状態を利用できます。Java プログラムの状態は完全にプライベートで、セッションが終了するまで存続します。

Aurora JVM は、セッション内で次のものを管理します。

- Java の静的変数から参照されているすべてのオブジェクトや、これらのオブジェクトから参照されているすべてのオブジェクトなど（オブジェクトの推移閉包）
- 1 つのクライアントに対するガベージ・コレクション
- 静的変数のセッション・メモリーおよび複数のコール間の必要メモリー
- 1 つのコールで存続する変数のコール・メモリー

セッションの存続期間

Oracle8*i* に接続すると、データベース・セッションが開始します。次のイベントが発生すると、セッションが終了します。

1. ユーザーによる `oracle.aurora.mts.session.Session.THIS_SESSION().endSession()` メソッドのコール。
2. セッションのタイムアウト。これは CORBA または EJB セッションではオプションです。
3. Java コード外でのユーザー操作によるデータベース・セッションの終了。

Java をサポートする API

現行リリースの Oracle8*i* には、Java ストアド・プロシージャ、サーブレット、JavaServer Pages、CORBA 分散オブジェクトおよび Enterprise JavaBeans（EJB）の 5 つの API が用意されています。

API	存続期間
Java ストアド・プロシージャ	ストアド・プロシージャ・セッションの存続期間は、SQL セッションの存続期間と同じです。この概念は PL/SQL ユーザーにはよく知られています。Java で表される状態は、RDBMS セッションの存続期間中、透過的に存続します。それによって、Oracle の抽象データ型のストアド・プロシージャ、トリガーおよびメソッドの作成が簡単になります。セッション内における Java コードの個々の起動をコールと言います。たとえば、SQL コールでコールできます。
サーブレットとJavaServer Pages	サーブレットと JavaServer Pages は、HTML 内から Java を起動する動的な方法です。中間層キャッシュまたはデータベースのいずれかのリソースに格納された HTML と Java アプリケーション間の通信を管理するために使用されます。
CORBA と EJB	CORBA と EJB では、よりオブジェクト指向的にクライアントとサーバー間でメッセージをやり取りできます。クライアントはサーバーにおいて、セッションを暗黙的に、または明示的に確立する必要があります。クライアントからサーバー側オブジェクトにメッセージを送るたびに、コールされます。それぞれの詳細は、『Oracle8*i* Enterprise JavaBeans 開発者ガイドおよびリファレンス』または『Oracle8*i* CORBA 開発者ガイド』のマニュアルを参照してください。

注意： コールとセッションの概念は Oracle8*i* の使用時に必ず適用されます。

また、SQLJ または JDBC を通じて SQL データにアクセスすることも可能です。それぞれの Java API の例は、第 3 章「データベースでの Java のコール」を参照してください。

実行制御

Sun Microsystems の JDK 環境では、`main()` メソッドを使用して Java アプリケーションを作成します。このメソッドは、クラスの実行時にインタプリタによって呼び出されます。`main()` メソッドは、コマンドラインで `java` <*classname*> を実行するとコールされます。このコマンドは、Java インタプリタを起動して、実行対象のクラス名をインタプリタに渡します。Java インタプリタはそのクラスをロードし、`main()` をコールして実行を開始します。ただし、データベース内の Java アプリケーションの実行は `main()` メソッドから起動されるわけではありません。

データベースに Java アプリケーションをロードした後（2-22 ページの「クラスのロード」を参照）、ロードされたクラス内の任意の静的メソッドをコールして Java コードを実行できます。クラスまたはメソッドは、実行前に開発者に対し公開されている必要があります（2-29 ページの「公開」を参照）。一度公開されると、唯一のエントリ・ポイントが常に

main() であるとは限らなくなります。そのため、Java アプリケーションを実行するときには、ロードされたクラス内のメソッド名をエントリ・ポイントとして指定します。

たとえば、標準的な Java 環境では、次のようなコマンドを実行してサーバー上で Java オブジェクトを起動します。

```
java myprogram
```

このとき、myprogram は main() メソッドを持つクラスの名前にします。myprogram では、main() がただちに mymethod をコールして、受信した情報を処理します。

Oracle8*i* では、データベースに myprogram.class ファイルをロードし、エントリ・ポイントとして mymethod を公開します。これで、クライアントまたはトリガーにより明示的に mymethod がコールされます。

JDK 1.1 から Java 2 への移行

Java 2 は JDK 1.1 にほぼ対応しています。Sun Microsystems は、セキュリティ機能など一部の機能を Java 2 で変更しています。これらの変更は、次の Sun Microsystems 社の Web サイトを参照してください。

http://java.sun.com/products/jdk/1.2/compatibility.html

以降では、Java 2 での変更が Oracle8*i* に及ぼす影響を説明します。

- 開発環境
- JDBC 2.0
- Java 2 セキュリティ
- Java 2 ORB の API

開発環境

サーバーとの相互運用性は開発環境のレベルによって決まります。開発環境が Java 2 ベースであれば、そのシステムでコンパイルまたはデバッグしたコードをデータベースにロードして実行できます。ただし、クライアントでのアプリケーションの開発環境が JDK 1.1 の場合は、JDK 1.1 のクラスのみが使用できます。非常にまれですが、Java 2 データベース・サーバーで JDK 1.1 アプリケーションをロードして実行できる場合もあります。もちろん、どのような場合においても、Java 2 ベースのアプリケーションをシステム上でコーディングして、データベースにロードし、データベース内の Java 2 コンパイラを使用してコンパイルすることは可能です。

注意： コードが JDK 1.1 ベースの場合でも、Java 2 セキュリティを使用するための回避策はあります。PL/SQL パッケージで、セキュリティ API が提供されています。コードを実行する前にこのコール仕様を使用すれば、適切な Java 2 のパーミッションを可能にできます。

JDBC 2.0

リリース 8.1.5 は JDK 1.1 ベースでしたが、Oracle 固有のパッケージである `oracle.jdbc2` としてリリース 8.1.5 で JDBC 2.0 サポートが追加されていました。しかし、Oracle8*i* の現行バージョンは Java 2 をサポートしているため、JDBC 2.0 は本来のパッケージである `java.sql` に含まれています。JDBC 2.0 API 用の `oracle.jdbc2` パッケージを使用する JDBC プログラムがある場合、これらを JDBC 2.0 ドライバで実行する前にプログラムを変更しておく必要があります。

注意： Java アプリケーションで `oracle.jdbc2` パッケージ JDBC API が使用されていない場合（またはこのパッケージ内に存在する型のオブジェクトが戻された場合）、8.1.6 データベースに接続するために移行は必要ありません。

Oracle8*i* に Java 2 が追加されたことで、`java.sql` パッケージに JDBC 2.0 のサポートが含まれるようになりました。このパッケージは JDK コア・ライブラリに入っています。このため、アプリケーションに次のような影響があります。

環境	
JDBC 2.0 を使用する Oracle8*i* でのサーバー・アプリケーションの実行	`java.sql` パッケージを使用する必要があります。`oracle.jdbc2` を使用すると、アプリケーションはコンパイルされません。
Java 2 環境でのクライアントの実行	`java.sql` パッケージを使用する必要があります。oracle.jdbc2 を使用すると、クライアントはコンパイルされません。
JDK 1.1 環境でのクライアントの実行	classes 111.zip に対してコンパイルし、クライアントとしてサーバーにアクセスするときのみこのコードを使用する場合には、引き続き `oracle.jdbc2` API を使用できます。サーバーにこのコードをロードする場合には、`java.sql` パッケージに移行する必要があります。

JDBC 2.0 を使用するサーバー・アプリケーション

サーバー・アプリケーションは常に Java 2 環境のデータベースにロードされて実行されるため、次のようにアプリケーションをポートする必要があります。

1. プログラムに入っている `oracle.jdbc2` のすべてのインポートと他の記載部分を、`java.sql` で置き換えます。

   `oracle.jdbc2` パッケージには、Oracle が JDK 1.1D ドライバで実装した JDBC 2.0 実装が含まれています。これらのクラスとインタフェースは Java 2 でも共通なため、classes12.zip には `oracle.jdbc2` は含まれていません。

2. `Connection` の `getTypeMap()` メソッドの戻り型は、`java.util.Dictionary` から `java.util.Map` に変更されているので、これに応じてアプリケーションを変更してください。`Hashtable` では `java.util.Map` が実装されているので、`java.util.Hashtable` を使用している場合は変更は不要です。

3. Make ファイルの classes111.zip を classes12.zip に置き換えてください。

4. 実行可能ファイルを再コンパイルして再リンクしてください。

   `oracle.jdbc2` のインポートに使用されたすべてのクラスを再コンパイルする必要があります。さらに、クラスに戻されるオブジェクトの型が `oracle.jdbc2` パッケージ内に存在する場合は、オブジェクトが戻されるクラスを再コンパイルする必要があります。次の例を想定してください。

   ```
   #import java.sql.*;
   . . .
   jd = (java.sql.Connection) getDC();
   ```

   `getDC` メソッドによって `oracle.jdbc2` パッケージ内からオブジェクト型が戻された場合、`java.sql.*` の型キャストであるため、コンパイルでは問題になりません。かわりに、データベースにアプリケーションをロードしたときに次のようなエラー・メッセージが表示されます。

   `ORA-29521:` 参照名 `oracle/jdbc2/<`*`classname`*`>` が見つかりません。

   これらのすべてのクラスを classes12.zip に対してコンパイルすると、`java.sql.*` に戻されるオブジェクト型が訂正されます。

詳細は、『Oracle8*i* JDBC 開発者ガイドおよびリファレンス』の第 4 章を参照してください。

JDBC 2.0 を使用するクライアント

次のクライアント・アプリケーションは 8.1.6 と相互運用できます。

- JDK 1.1 に準拠し、`oracle.jdbc2` の JDBC 2.0 API を使用する Java クライアント。このアプリケーションは classes111.zip に入っている `oracle.jdbc2` パッケージをインポートします。

- Java 2 に準拠し、`java.sql` の JDBC 2.0 API を使用する Java クライアント。このアプリケーションは、`oracle.jdbc2` パッケージではなく、classes12.zip に入っている `java.sql` パッケージをインポートします。

Java 2 セキュリティ

Java 2 セキュリティはリリース 8.1.6 で実装されています。Oracle8*i* では、JDK 1.1 セキュリティ sandbox は適用されません。PL/SQL パッケージの DBMS_JAVA を使用してこれらのパーミッションを管理すれば、コードを変更せずに Java 2 セキュリティのパーミッションを使用できます。`doPrivileged` など、いずれかの Java 2 セキュリティ・メソッドを実行するには、Java アプリケーションを Java 2 環境でコンパイルする必要があります。JDK 1.2 リリースではパーミッションが新規のものになるためです。

Java 2 セキュリティに関する詳細は、5-3 ページの「Java 2 セキュリティ」を参照してください。

Java 2 ORB の API

Oracle8*i* JVM では ORB 実装が Visigenic 3.4 にアップデートされています。このバージョンは JDK 1.1 および Java 2 に準拠しています。

注意： 8.1.6 と動作するためには、すべての既存の CORBA アプリケーションはスタブおよびスケルトンを再生成する必要があります。アプリケーションの再生成には、8.1.6 のツールを使用してください。

Sun Microsystems の Java 2 には、ORB 実装が含まれていますが、JDK 1.1 には含まれていませんでした。したがって、Visigenic ライブラリをインポートして CORBA メソッドをコールすると、常に Visigenic の実装がコールされます。Java 2 の場合、次のような変更をせずに CORBA メソッドをコールすると、Sun Microsystems の CORBA の実装がコールされてしまい、予期せぬ結果が生じることがあります。

次に、クライアントから Oracle8*i* の CORBA サーバー・オブジェクトにアクセスするための 3 つの方法と、Sun Microsystems の CORBA 実装をバイパスするための推奨事項をリストします。

- JNDI ルックアップ — ルックアップ・メソッドのセットアップは JDK 1.1 と Java 2 ではまったく同じですが、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 は、ORB を初期化するためのインタフェースを提供します。JNDI を使用しないと、クライアントはそのノード上で ORB を初期化してデータベースの ORB と対話します。このクラスを通じて、クライアント上の Aurora ORB を使用することができます。
- 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には、適切な実装に確実にアクセスするために、いくつかのプロパティを設定する必要があり

ます。クライアントで Aurora ORB を使用しない場合は、純粋な CORBA インタフェースを使用できます。ただし、適切な実装にコールを送るには環境を設定する必要があります。

JNDI ルックアップ

クライアントで JNDI を使用してサーバー上の CORBA オブジェクトにアクセスする場合は、コードを変更する必要はありません。ただし、CORBA 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 にアップグレード済みであれば、`oracle.aurora.jndi.orb_dep.Orb.init` メソッドを通じて ORB を初期化できます。このメソッドにより、ORB を初期化した時に確実に ORB インスタンスが 1 つのみ初期化されます。すなわち、Java 2 ORB インタフェースを使用すると、`init` メソッドを起動するたびに新しい ORB インスタンスが返されます。Aurora の `init` メソッドは 1 つずつ ORB インスタンスを初期化します。以降、`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返されます。

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

oracle.aurora.jndi.orb_dep.Orb クラス `init` メソッドは複数あり、それぞれにパラメータ・リストは異なります。各 `init` メソッドの構文とパラメータは次のとおりです。

> **注意：** 返されたクラスは `init` メソッドによってそれぞれ異なります。`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。

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

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

例 2-1 Aurora ORB init メソッドの使用方法

次の例は、クライアントで Aurora Orb クラスを使用して ORB を初期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();
```

セッションの結合 あるセッションで作成されたオブジェクトへの参照をクライアントが受け取ると、そのセッションでそのオブジェクトのメソッドを起動できます。ただし、すべてのクライアントは認証される必要があるため、データベースにユーザー名とパスワードを指定する必要があります。サーバーが SSL_CREDENTIALS 形式のクライアント側の認証を必要としている場合には、`ORB.init` メソッドの中の接続ハンドシェイクに渡されるクライアントのユーザー名、パスワードおよびロールを指定できます。

```
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	クライアント側の認証のユーザー名。
String pw	クライアント側の認証のパスワード。
String role	ログオン後に使用するロール。
Boolean ssl	TRUE の場合、その接続で SSL が使用可能になります。FALSE の場合は、非 SSL 接続が使用されます。
Properties props	ORB システム・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た場合、すなわち JNDI を使用しない場合、ORB 初期化の前に次のような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は、Java 2 の実装ではなく、Aurora の実装を直接呼び出します。これにより、目的の動作を確保できます。Visigenic ORB で予期される動作は次のとおりです。

- `ORB.init` を複数回呼び出しても、作成される ORB は 1 つのみです。これらのプロパティを設定しないと、`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ので注意してください。
- セッション IIOP 接続は正しく処理されます。
- サーバーからのコールバックは正しく処理されます。

注意： Aurora CORBA 実装は Visibroker 3.4 に基づいています。

プロパティ	値の割当て
org.omg.CORBA.ORBClass	com.visigenic.vbroker.orb.ORB
org.omg.CORBA.ORBSingletonClass	com.visigenic.vbroker.orb.ORB

例 2-2 OMG プロパティへの Visigenic 値の割当て

次の例は、OMG CORBA `init` メソッドを Visigenic 実装に誘導する OMG プロパティのセットアップ方法を示します。

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

あるいは、次のようにコマンドラインにプロパティを設定することもできます。

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

8.1.5 CORBA と EJB アプリケーションの下位互換性

Oracle8*i* で提供される `publish` などのツールは、JDK 1.1 または Java 2 環境で動作するように変更されています。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 CORBA または EJB コードは正常に実行されません。すべてのツールに対して必ずリリース 8.1.6 を使用するようにしてください。この規則は CORBA スタブおよびスケルトンにも当てはまります。すべてのスタブとスケルトンは、リリース 8.1.6 IDL コンパイラで再生成する必要があります。

Java のコード、バイナリおよびリソースの格納

Sun Microsystems の Java 開発環境では、Java のソース・コード、バイナリおよびリソースは、ファイルとしてファイル・システムに格納されます。

- ソース・コード・ファイルは `.java` ファイルと呼ばれます。
- コンパイル済み Java バイナリ・ファイルは `.class` ファイルと呼ばれます。
- リソースは、`.properties` ファイルや `.ser` ファイルなどの任意のデータ・ファイルであり、ファイル・システム階層に格納され、実行時にロードまたは使用されます。

また、Java を実行するときは、CLASSPATH を指定します。CLASSPATH は、ファイルが格納されているファイル・システム・ツリーのルートで構成されています。Java では、これらのファイルを単一のアーカイブ形式の ZIP または JAR ファイルにまとめることもできます。

この 2 つの概念はいずれもデータベース内では異なります。次に、Oracle8*i* JVM による Java クラスの処理方法と依存クラスの検索方法を説明します。

Java のコード、バイナリ、およびリソース	Oracle8*i* JVM 環境では、ソース、クラスおよびリソースは Oracle8*i* データベースに常駐します。これらはデータベースに常駐するため、Java スキーマ・オブジェクトと呼ばれます。ここで言うスキーマとは、データベース・ユーザーに対応します。Java オブジェクトは 3 種類あります。ソース、クラスおよびリソースの 3 種類です。サーバー上には、.java、.class、.sqlj、.properties または .ser ファイルはなく、これらのファイルは Java のソース、クラスおよびリソース・スキーマ・オブジェクトにマップされます。
Java クラスの検索	ソース、クラスおよびリソースの Java スキーマ・オブジェクトを検索するには、CLASSPATH のかわりに、リゾルバを使用してスキーマを 1 つ以上指定します。

この説明の中で使用されるコールとセッションという用語は、Java の用語ではなく、Oracle8*i* JVM プラットフォームに適用されるサーバー用語です。Java プログラムの状態は、セッション全体を通して、つまり次のコールまで、Aurora メモリー・マネージャによって保持されます。Oracle8*i* JVM は Java スキーマ・オブジェクト（つまり、スキーマ内の Java ソース、クラスおよびリソース）を Oracle データベースに保持します。リゾルバでは、Java をサーバーで実行したときの、ソース・コード、クラスおよびリソースの検索方法を指定できます。

Java クラス・メソッドの実行準備

Java メソッドを実行するには、次のような準備が必要になります。

1. ソースをいつコンパイルするかを決定します。
2. データベース内の Java クラスの検索にデフォルトのリゾルバまたは別のリゾルバを使用するかを決定します。
3. クラスをデータベースにロードします。デフォルトのリゾルバを使用しない場合は、ロード・コマンドで別のリゾルバを指定してください。
4. クラスまたはメソッドを公開します。

Java クラスのコンパイル

ソースのコンパイルは次のいずれかの方法で実行できます。

- クライアント・マシンで明示的にソースをコンパイルしてから、javac などの Java コンパイラを通じてデータベースにロードできます。
- データベースに対して、loadjava ツール内で管理されるロード・プロセス中にソースをコンパイルするように求めることができます。
- 実行時に動的にコンパイルが行われるように強制できます。

注意： loadjava を通じてコンパイルを実行する場合は、コンパイラ・オプションを指定できます。詳細は、2-16 ページの「コンパイラ・オプションの指定」を参照してください。

---

javac によるソースのコンパイル

javac などの従来の Java コンパイラを使用しても、Java をコンパイルできます。コンパイル終了後、ソースそのものではなく、コンパイル済みのバイナリをデータベースにロードします。通常はデータベースではなく、自分のシステムで Java コードをデバッグする方が簡単なので、これは便利な方法です。

loadjava によるソースのコンパイル

ソース・ファイルに対する loadjava に -resolve オプションを指定すると、次のような処理が行われます。

1. ソース・ファイルがソース・スキーマ・オブジェクトとしてロードされます。
2. ソース・ファイルがコンパイルされます。
3. コンパイル済みの .java ファイルで定義された各クラスについて、クラス・スキーマ・オブジェクトが作成されます。
4. コンパイル済みのコードがクラス・スキーマ・オブジェクトに格納されます。

Oracle8*i* JVM はすべてのコンパイル・エラーを loadjava のログファイルと USER_ERRORS ビューに記録します。USER_ERRORS ビューの詳細は、『Oracle8*i* リファレンス・マニュアル』を参照してください。

ランタイムでのソースのコンパイル

-resolve オプションを指定せずにデータベースに Java ソースをロードすると、実行時にクラスが必要である場合には、Oracle8*i* JVM によってソースが自動的にコンパイルされます。そのソース・ファイルは、ソース・スキーマ・オブジェクトにロードされます。

Oracle8*i* JVM はすべてのコンパイル・エラーを loadjava のログファイルと USER_ERRORS ビューに記録します。USER_ERRORS ビューの詳細は、『Oracle8*i* リファレンス・マニュアル』を参照してください。

コンパイラ・オプションの指定

コンパイラに対して、次の 2 通りの方法でオプションを指定できます。

- loadjava コマンドラインにコンパイラ・オプションを指定します。encoding オプションは、loadjava コマンドラインで指定できます。
- JAVA$OPTIONS という名前のスキーマごとのデータベース表に永続的コンパイラ・オプションを指定します。コンパイルを実行するたびに、コンパイラはこれらのオプションを使用します。ただし、loadjava コマンドにコンパイラ・オプションが指定されていると、この表に定義されたオプションはオーバーライドされます。

  この方法でコンパイラ・オプションを指定するには、自分でこの表を作成する必要があります。JAVA$OPTIONS 表を作成する方法は、2-16 ページの「データベース表に指定されるコンパイラ・オプション」を参照してください。

以降では、コンパイラ・オプションの説明をします。

- デフォルトのコンパイラ・オプション
- コマンドラインのコンパイラ・オプション
- データベース表に指定されるコンパイラ・オプション

デフォルトのコンパイラ・オプション コンパイル対象のソース・スキーマ・オブジェクトに対して、コマンドラインのオプション値も JAVA$OPTIONS 表のエントリもない場合は、コンパイラは次のデフォルト値をとります。

- encoding = latin1
- online = true このオプションの詳細は、『Oracle8*i* 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。このオプションの適用対象は、SQLJ 構文を含む Java ソースのみです。
- debug = true このオプションは javac -g と同じです。

コマンドラインのコンパイラ・オプション loadjava コンパイラ・オプション、encoding は、.java ファイルのエンコーディングを識別します。このオプションは、JAVA$OPTIONS 表の一致する値をすべてオーバーライドします。値は、javac の -encoding オプションと同じです。このオプションはソース・ファイルのロード時のみ有効です。

データベース表に指定されるコンパイラ・オプション JAVA$OPTIONS 表の各行の内容は、オプション設定を適用するソース・スキーマ・オブジェクトの名前です。複数行を使用して、ソース・スキーマ・オブジェクトごとに異なるオプションを設定できます。

JAVA$OPTIONS 表のエントリを設定するには、次のファンクションとプロシージャを使用します。これらのファンクションとプロシージャはデータベース・パッケージ DBMS_JAVA に定義されています。

- PROCEDURE set_compiler_option(name VARCHAR2, option VARCHAR2, value VARCHAR2) ;
- FUNCTION get_compiler_option(name VARCHAR2, option VARCHAR2) RETURNS VARCHAR2;
- PROCEDURE reset_compiler_option(name VARCHAR2, option VARCHAR2);

これらのメソッドのパラメータは次のとおりです。

パラメータ	説明
name	name パラメータには、Java パッケージ名または完全なクラス名指定です。または空文字列にします。コンパイラは Java ソース・スキーマ・オブジェクトのコンパイルに使用するオプションを JAVA$OPTIONS 表で検索するときに、スキーマ・オブジェクトの完全なクラス名指定に最も近い name の行を使用します。値が空文字列の name は、あらゆるスキーマ・オブジェクト名に一致します。
option	option パラメータで、'online'、'encoding' または 'debug' を指定します。これらのオプションに対して指定できる値は、『Oracle8*i* 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。

スキーマには最初、JAVA$OPTIONS 表がありません。JAVA$OPTIONS 表を作成するには、DBMS_JAVA パッケージの java.set_compiler_option プロシージャを使用して、値を設定します。このプロシージャは、表が存在しない場合には作成します。パラメータは引用符で囲みます。次のように指定してください。

```
SQL> execute dbms_java.set_compiler_option('x.y', 'online', 'false');
```

表 2-1 は、JAVA$OPTIONS データベース表のサンプルです。パターン一致規則は、表エントリに対してできるだけ多くのスキーマ名を一致させるものです。パターン一致の名前解決度が高いスキーマ名がエントリとして適用されます。この表には、encoding オプションのエントリがないので、コンパイラはデフォルトまたはコマンドラインで指定された値を使用します。この表の online オプションは、次のようにスキーマ・オブジェクト名に対応付けられます。

- 名前 a.b.c.d は、a.b.c.d で始まるクラス名およびパッケージ名に一致します。このパッケージとクラスは、online = true でコンパイルされます。
- 名前 a.b は、a.b で始まるクラス名およびパッケージ名に一致します。名前 a.b は、a.b.c.d に一致しないので、パッケージとクラスは online = false でコンパイルされます。
- その他のパッケージおよびクラスは空文字列のエントリに一致し、online = true でコンパイルされます。

表 2-1 JAVA$OPTIONS 表の例

JAVA$OPTIONS エントリ			
名前	**オプション**	**値**	**一致例**
a.b.c.d	online	true	■ a.b.c.d— パターンに完全に一致。 ■ a.b.c.d.e— 前半はパターンに完全に一致。名前全体に一致する規則はない。
a.b	online	false	■ a.b— パターンに完全に一致。 ■ a.b.c.x— 前半はパターンに完全に一致。指定の規則名以降の部分に一致する規則はない。
（空文字列）	online	true	■ a.c— 定義された名前と一致するパターンがない。デフォルト（空文字列）規則が適用。 ■ x.y— 定義された名前と一致するパターンがない。デフォルト（空文字列）規則が適用。

自動再コンパイル

Oracle8*i* JVM には、依存関係管理や自動作成機能があります。ソース・プログラムまたはバイナリ・プログラムを変更すると、これらのプログラムに依存しているソース・プログラムがこの機能によって透過的に再コンパイルされます。次のケースを想定してください。

```
public class A
{
      B b;
      public void assignB () {b = new B()}
}
public class B
{
      C c;
      public void assignC () {c = new C()}
}
public class C
{
      A a;
      public void assignA () {a = new A()}
}
```

システムはクラス・レベルで依存関係を追跡します。この例では、クラス A、B および C が相互に依存しています。A が B のインスタンスを保持し、B が C のインスタンスを保持し、C が A のインスタンスを保持しているためです。クラス A に新しいフィールドを追加して、このクラスの定義を変更すると、Oracle8*i* の依存関係機構によってクラス B と C が無効に

なったことを示すフラグが設定されます。そして、これらのクラスを再度使用する前に、Oracle8*i* によってこれらのクラスは解決され、必要であれば、再コンパイルされます。クラスの再コンパイルは、ソースがサーバーにある場合のみ行われます。

この依存性のあるシステムによって、クラス間の依存関係が Oracle8*i* によって管理され、解決と再コンパイルが自動的に行われます。開発作業中に、早期に問題を特定する必要がある場合以外は、コンパイルと解決を強制する必要はありません。`loadjava` ユーティリティには、依存関係管理システムに処理を任せずに、コンパイルと解決を意図的に行うための機能もあります。

クラス依存関係の解決

Java クラスの多くは他のクラスへの参照を含んでいます。これによりコードの再利用が可能になります。従来の Java 仮想マシンは、CLASSPATH に指定されたディレクトリからクラス、ZIP および JAR ファイルを検索していました。一方、Aurora Java 仮想マシンはデータベース・スキーマでクラス・オブジェクトを検索します。Oracle8*i* ではすべての Java クラスがデータベースにロードされるため、ある Java クラスが依存しているクラスを検索するためには、データベース上で検索場所を指定する必要があります。

データベースにロードされたすべてのクラスはクラス・スキーマ・オブジェクトと呼ばれ、特定のスキーマの中にロードされます。`java.lang.*` などの JVM クラスはすべて `PUBLIC` にロードされます。自分の作成したクラスが、自分が事前に定義した他のクラスに依存するような場合、自分のスキーマにすべてのクラスをロードすることになります。たとえば、自分のスキーマが `SCOTT` である場合、データベース・リゾルバ（`CLASSPATH` に代わるデータベース）は `PUBLIC` を検索する前に `SCOTT` スキーマを検索します。スキーマの検索リストはリゾルバ仕様と呼ばれます。リゾルバ仕様はクラスごとにあります。一方、従来の Java 仮想マシンの CLASSPATH はすべてのクラスを対象にしたグローバルな仕様です。

クラス間の依存関係を検索、解決する場合に、リゾルバは依存するすべてのクラスが見つかったか否かを判断して、各クラスに有効または無効とマークします。ロードしたクラスの参照先のクラスが適切なスキーマに存在しない場合は、このクラスは無効とマークされます。実行時に解決できないと「class not found」例外になります。また、クラスのツリーが大きすぎると、データベース・リソースの不足のために、実行時に解決できないことがあります。

注意： Java コンパイラと同様、`loadjava` はクラスへの参照を解決しますが、リソースへの参照は解決しません。したがって、クラスに必要なリソース・ファイルを正しくロードする必要があります。

クラス内のクラス間参照ごとに、リゾルバはリゾルバ仕様で指定されたスキーマで、参照を満足させる有効なクラス・スキーマ・オブジェクトを検索します。すべての参照を解決すると、リゾルバはこのクラスを有効とマークします。解決を行っていないクラス、または解決できなかったクラスは無効とマークされます。無効化されたスキーマ・オブジェクトに依存するクラスも無効とマークされます。

依存クラスを簡単に検索できるように、Oracle8*i* には定義者のスキーマを検索してから PUBLIC を検索するデフォルトのリゾルバとリゾルバ仕様が用意されています。これでデータベースにロードされたほとんどのクラスを検索できます。ただし、自分または PUBLIC 以外のスキーマにロードされたクラスにアクセスする場合は、自分でリゾルバ仕様を定義する必要があります。

- 定義者のスキーマと PUBLIC を検索する、Oracle のデフォルト・リゾルバを使用してロードします。

  ```
  loadjava -resolve
  ```

- SCOTT スキーマ、OTHER スキーマおよび PUBLIC を含む、自分のリゾルバ仕様の定義を使用してロードします。

  ```
  loadjava -resolve -resolver "((* SCOTT)(* OTHER)(* PUBLIC))"
  ```

-resolver オプションを使用して、定義されたスキーマ内で検索するオブジェクトを指定します。前述の例では、SCOTT、OTHER および PUBLIC で、すべてのクラス・スキーマ・オブジェクトが検索されます。スキーマで特定のクラスまたはクラス・グループのみを検索する場合は、検索範囲を絞り込むこともできます。たとえば、OTHER スキーマの「my/gui/*」クラスのみ検索するには、次のようにリゾルバ仕様を定義します。

```
loadjava -resolve -resolver '((* SCOTT) ("my/gui/*" OTHER) (* PUBLIC))'
```

リゾルバ仕様の最初のパラメータは、クラス・スキーマ・オブジェクトを指定し、次のパラメータはこれらのクラス・スキーマ・オブジェクトを検索するスキーマを定義します。

存在しないクラスへの参照のパーミッション

存在しないクラスへの未解決の参照を可能にする特別なオプションを、リゾルバ仕様に指定できます。製品において内部クラスがまったく使用されないこともあります。たとえば、ISV によっては JAR ファイルからの内部テスト・クラスに対する参照をすべて削除しないまま製品を出荷することもあります。標準的な Java 環境では、そのメソッドがコールされない限り Sun Microsystems の JVM はこれらのメソッドを無視するため、問題は起こりません。ただし、Oracle8*i* のリゾルバは、未使用クラスも含め、JAR ファイル内の参照先のクラスをすべて解決しようとします。参照を有効化できないと、JAR ファイル内のクラスは無効とマークされます。

リゾルバ仕様でワイルドカード "-" を指定することによって、参照を無視できます。次の例は、たとえ参照先のクラスがリゾルバ仕様スキーマ・リストに存在しなくても、「my/gui」内のクラスへのすべての参照を許すように指定するものです。

```
loadjava -resolve -resolver '((* SCOTT) (* PUBLIC) ("my/gui/*" -))'
```

さらに、検索できなかったクラスをすべて無視するように定義することも可能です。ワイルドカードを指定しないと、あるスキーマにおいて依存クラスが見つからなければ、そのクラスは無効としてマークされ実行できなくなります。ただしこの場合、使用するクラスに対し

て依存クラスがあったとしても、依存クラス無しでは実行できないクラスを有効としてマークする恐れがあるので危険です。このような場合には、実行時に例外となります。

SCOTT または PUBLIC に存在しないクラスをすべて無視するには、次のようなリゾルバ仕様を指定します。

```
loadjava -resolve -resolver "((* SCOTT) (* PUBLIC) (* -))"
```

注意： 後にロードするクラスが "-" を含むリゾルバの使用を必要とするような場合は、はじめから "-" を含むリゾルバを使用しないでください。そのかわりに、参照先クラスをすべてスキーマに取り込んでから、解決してください。

ByteCode ベリファイヤ

JVM 仕様によると、`.class` ファイルが検証されて始めて、その定義されたクラスが JVM で使用可能となります。Oracle8*i* JVM では、クラスの解決時に検証が行われます。リゾルバが次のような問題を検出すると、該当する Oracle エラー・コードが表示されます。

ORA-29545　リゾルバがクラスの形式を不正と判断すると、そのクラスは有効とマークされません。リゾルバがクラスを拒否すると、ORA-29545 エラー（クラスの形式が不正です）が発行されます。エラーは `loadjava` ツールによってレポートされます。たとえば、`.class` ファイルの内容が Java コンパイルの結果に一致しない場合や、ファイルが壊れている場合はこのエラーが発生します。

ORA-29552　ケースによっては例外を実行時に発生できるように、リゾルバが有効とマークしたクラスのバイトコードを変更することもあります。このような場合には、リゾルバは ORA-29552（証明の警告）を発行し、`loadjava` によってこのエラーがレポートされます。Java 言語仕様で `IncompatibleClassChangeError` が発生するように規定されていると、`loadjava` ツールはこの警告を発行します。Oracle8*i* JVM は、リゾルバにこのような状況を検出させることによって、JLS で規定されているランタイム動作をサポートします。

さらに、リゾルバは次のように定義された警告を発行します。

- 「-」を含むリゾルバ

  この種のリゾルバは、参照先クラスの有無に関係なく、クラスを有効とマークします。継承とインタフェースによって、あるクラスのインスタンスをスーパークラスまたは特定のインタフェースのインスタンスであるかのように使用する Java メソッドを作成できます。検証対象のメソッドがクラス A への参照をクラス B への参照であるかのように使用しているときは、リゾルバは A が B を拡張または実装しているかどうかを確認します。たとえば、次にあげるメソッドについて考えてみます。そのシグネチャでは暗黙的に B のインスタンスを返すことになっていますが、その本体は A のインスタンスを戻しています。

```
B myMethod(A a) { return a; }
```

A が B を拡張しているか、または A がインタフェース B を実装している場合のみ、このメソッドは有効です。A または B の解決時に "-" が使用された場合は、リゾルバはこのメソッドが安全かどうかを判断できません。`myMethod` がコールされる場合は、`myMethod` のバイトコードが例外を発生するバイトコードに置き換えられます。

- 他のリゾルバの使用

  A と B のクラス定義で具体的に特定したスキーマに A と B がある場合、A と B のクラス定義を検索し適切に解決するには、リゾルバを使用します。ロード対象の既存の JAR ファイルに、別の非システム・クラスを参照する JAR ファイルに含まれないクラスがある場合は、代替リゾルバの使用を検討してください。

クラス解決とデータベースのクラスのロードの詳細は、『Oracle8*i* Java Tools リファレンス』を参照してください。

クラスのロード

ここでは、データベースにクラスをロードするときに理解しておく必要のある重要な点を概説します。ここでは、`loadjava` ツールの様々なオプションを簡単に説明します。また、SQL 内で `loadjava` を実行することもできます。`loadjava` の詳細は、『Oracle8*i* Java Tools リファレンス』を参照してください。

従来の Java 仮想マシンのコンパイルおよびロード対象はファイルでしたが、Aurora Java 仮想マシンのコンパイルおよびロード対象はデータベース・スキーマ・オブジェクトです。

`.java` ソース・ファイルまたは `.sqlj` ソース・ファイル	Java ソース・スキーマ・オブジェクトに対応
`.class` コンパイル済み Java ファイル	Java クラス・スキーマ・オブジェクトに対応
`.properties` Java リソース・ファイル、`.ser` SQLJ プロファイル・ファイル、またはデータ・ファイル	Java リソース・スキーマ・オブジェクトに対応

データベース内の他のクラスで使用するには、すべてのクラスまたはリソースをデータベースにロードする必要があります。さらに、ロード時にはデータベース内のクラスを実行できるユーザーを定義します。

`loadjava` ツールは各タイプのファイルに対して次のことを実行します。

スキーマ・オブジェクト	オブジェクトに対する loadjava の操作
.java ソース・ファイル	**1.** 別のスキーマが指定された場合を除き、定義者のスキーマの中にソース・スキーマ・オブジェクトを作成します。 **2.** スキーマ・オブジェクトにソース・ファイルの内容をロードします。 **3.** ソース・ファイルに定義されたすべてのクラスでクラス・スキーマ・オブジェクトを作成します。 **4.** -resolve が指定された場合は、次のことを実行します。 a. ソース・スキーマ・オブジェクトをコンパイルします。 b. クラスとその依存クラスを解決します。 c. コンパイル済みのクラスをクラス・スキーマ・オブジェクトに格納します。
.sqlj ソース・ファイル	**1.** 別のスキーマが指定された場合を除き、定義者のスキーマの中にソース・スキーマ・オブジェクトを作成します。 **2.** スキーマ・オブジェクトにソース・ファイルの内容をロードします。 **3.** ソース・ファイルに定義されたすべてのクラスおよびリソースにクラス・スキーマ・オブジェクトを作成します。 **4.** -resolve が指定された場合は、次のことを実行します。 a. ソース・スキーマ・オブジェクトを変換してコンパイルします。 b. コンパイル済みのクラスをクラス・スキーマ・オブジェクトに格納します。 c. プロファイルを .ser リソース・スキーマ・オブジェクトに格納してカスタマイズします。
.class コンパイル済み Java ファイル	**1.** 別のスキーマが指定された場合を除き、定義者のスキーマの中にクラス・スキーマ・オブジェクトを作成します。 **2.** クラス・ファイルをスキーマ・オブジェクトにロードします。 **3.** -resolve が指定された場合は、クラスおよびその依存クラスを解決して検証します。

.properties Java リソース・ファイル	1. 別のスキーマが指定された場合を除き、定義者のスキーマの中にリソース・スキーマ・オブジェクトを作成します。 2. スキーマ・オブジェクトにリソース・ファイルをロードします。
.ser SQLJ プロファイル	1. 別のスキーマが指定された場合を除き、定義者のスキーマの中にリソース・スキーマ・オブジェクトを作成します。 2. .ser リソース・ファイルをスキーマ・オブジェクトに格納してカスタマイズします。

dropjava ツールは、loadjava の逆の動作をします。Java ファイルに対応するスキーマ・オブジェクトを削除します。loadjava で作成した Java スキーマ・オブジェクトを削除するには、必ず dropjava を使用してください。SQL DDL コマンドで削除しても、loadjava および dropjava で管理される補助データは更新されません。また、SQL コマンド内で dropjava を実行することもできます。

> **注意：** loadjava には他にもオプションがあります。しかし、ここでは主なオプションのみを取り上げます。loadjava と dropjava の詳細は、『Oracle8*i* Java Tools リファレンス』を参照してください。

データベースにクラスをロードするときには、次の項で説明する特定の規則に従う必要があります。

- 同じクラスの 2 つの定義
- 必要なデータベース権限および JVM のパーミッション
- JAR ファイルまたは ZIP ファイルのロード

クラスにロードした後、データベース・スキーマで USER_OBJECTS ビューにアクセスして、クラスおよびリソースが正しくロードされたことを確認できます。詳細は、2-27 ページの「Java アップロードのチェック」を参照してください。

同じクラスの 2 つの定義

同じクラスに 2 つの異なる定義を指定することはできません。この規則により、次のような 2 つの制限を受けることになります。

- 特定の .class ファイルか、その .java ファイルのどちらか一方をロードできますが、両方はロードできません。

  Oracle8*i* JVM はクラス・ファイルまたはソース・ファイルのどちらがロードされたかを追跡します。クラスを更新するには、先にロードしたのと同じタイプのファイルをロードする必要があります。他のタイプを更新するには、最初にロードしたタイプのファイ

ルを削除してから次のタイプのファイルをロードする必要があります。たとえば、クラス y のソースとして x.java をロードしてある場合に、x.class をロードするには、まず x.java を削除する必要があります。

- 1 つのスキーマ内の異なる 2 つのスキーマ・オブジェクトに、同じクラスを定義することはできません。たとえば、x.java でクラス y を定義し、y の定義を z.java に移動しようとします。x.java がすでにロードされている場合は、loadjava によって z.java のロードが拒否されます。これは、z.java にも y が定義されているためです。かわりに、次のどちらかの方法で操作する必要があります。
  - x.java を削除し、（y が定義されている）z.java をロードします。次に新しい（y が定義されていない）x.java をロードします。
  - 新しい（y が定義されていない）x.java をロードします。次に（y が定義されている）z.java をロードします。

必要なデータベース権限および JVM のパーミッション

クラスをロードするには、次のような SQL データベース権限が必要です。

- スキーマにロードするための CREATE PROCEDURE および CREATE TABLE 権限
- 別のスキーマにロードするための CREATE ANY PROCEDURE および CREATE ANY TABLE 権限
- oracle.aurora.security.Oracle8*i* JVMPermission.loadLibraryInClass.<*classname*>。詳細は、5-25 ページの「クラスをロードするためのパーミッション」を参照してください。

JAR ファイルまたは ZIP ファイルのロード

loadjava ツールは .class、.java、.properties、.sqlj、.ser、.jar、または .zip ファイルをアクセプトします。JAR ファイルまたは ZIP ファイルにはソース、クラスおよびデータ・ファイルが含まれます。loadjava に JAR ファイルまたは ZIP ファイルを渡すと、loadjava はアーカイブをオープンして、そのメンバーを個別にロードします。JAR または ZIP のスキーマ・オブジェクトはありません。JAR ファイルまたは ZIP ファイルの内容が前回のロード時から変更されていない場合は、再ロードはされません。したがって、JAR ファイルまたは ZIP ファイルのロードによるパフォーマンス低下はほとんどありません。実際、loadjava を使用するには、JAR ファイルまたは ZIP ファイルをロードするのが最も簡単な方法です。

注意： 前回のロード時から変更がない場合、Oracle8i JVM はクラスを再ロードしません。ただし、`loadjava -force` オプションを使用すると強制的にクラスを再ロードできます。

実行権限を付与する方法

クラスをロードするときには、`loadjava` のオプションを使用して別のユーザーに実行権限を付与することができます。クラスを実行できるユーザーを定義するには、2 つの方法があります。

- 特定のユーザーまたはスキーマに実行権限を付与する方法

  Java アプリケーションを定義したクラスは、その所有者の SQL スキーマにある、Oracle8*i* の RDBMS に格納されます。デフォルトでは、特定のユーザーのスキーマに格納されたクラスは、セキュリティ保護の観点から、他のユーザーからは実行できません。`loadjava -grant` オプションを使用して、自分のクラスの実行権限を他のユーザー（スキーマ）に与えることができます。

- クラスの実行権限の付与対象として評価するユーザーまたはスキーマを指定する方法

  実行権限の付与対象として評価されるユーザーを実行者または定義者のどちらかに指定できます。実行者権限と定義者権限は、Java クラスに適用できる SQL の概念です。SQL では、定義者権限で実行する場合には、PL/SQL をロードしたスキーマのユーザー認証を使用して実行します。実行者権限の場合には、実際に PL/SQL を起動するスキーマのユーザー認証を使用して SQL を実行します。これは Java クラスにも適用されますが、JVM セキュリティのパーミッションにより、Java クラス内における実行者と評価されるスキーマは、そのクラスが実行者権限でロードされたのか、定義者権限でロードされたのかに依存します。

図 2-2 実行者権限と定義者権限

メソッドの起動: クラス A がクラス B を起動し、クラス B がクラス C を起動します。
クラスの実行権限:
* クラス A にはクラス B に対する実行権限があります。
* クラス A にはクラス C に対する実行権限はありません。
* クラス B にはクラス C に対する実行権限があります。

図 2-2 の例の場合、クラス C のメソッドを実行するときにクラス A または B のどちらがチェックされるのでしょうか？これは実行者権限または定義者権限によって決まります。

* 実行者権限 － デフォルトでは、クラス C のメソッドを最初に起動したクライアントのスキーマの有効な認証と権限でクラスが実行されます。この場合は、クラス A の権限がチェックされます。クラス A はクラス C に対する実行権限を持っていないので、要求は拒否されます。これがデフォルト設定です。
* 定義者権限 － クラスは、そのクラスをロードしたユーザーの有効な認証で実行します。クラス B をロードした際に定義者権限を指定した場合、クラス C のメソッドが起動されるとクラス B がチェックされます。クラス B はクラス C のメソッドに対する実行権限を持っているので、メソッドは正常に完了します。クラス B は、`loadjava-definer` を実行して定義者権限でロードされています。

JVM セキュリティのパーミッションの詳細は、第 6 章「Oracle8i Java アプリケーションのパフォーマンス」を参照してください。

Java アップロードのチェック

`USER_OBJECTS` データベース・ビューに問合せをして、Java ソース、クラスおよびリソースを含め、自分が所有するリソースなどのスキーマ・オブジェクトに関する情報を得ることができます。たとえば、ロードしたソース、クラスまたはリソースが正しくスキーマ・オブジェクトに格納されているかどうかを確認できます。

`USER_OBJECTS` の列には、次の表 2-2 に示す項目が含まれています。

表 2-2 主な USER_OBJECT 列

名前	説明
OBJECT_NAME	オブジェクトの名前
OBJECT_TYPE	オブジェクトのタイプ（`JAVA SOURCE`、`JAVA CLASS` または `JAVA RESOURCE` など）
STATUS	オブジェクトの状態（`VALID` または `INVALID`）（`JAVA RESOURCE` については常に `VALID`）

オブジェクト名とタイプ

`USER_OBJECTS` の `OBJECT_NAME` は、短縮名です。31 文字を超える名前は短縮名で格納されます。完全名と短縮名の詳細は、2-30 ページの「クラスの短縮名」を参照してください。

サーバーでスキーマ・オブジェクトの短縮名を使用する場合、短縮名形式や変換規則を知らなくても、サーバーの `DBMS_JAVA` パッケージの `LONGNAME()` ルーチンを使用して、問合せをすることにより名前を完全名形式で受け取ることができます。

```
SQL*Plus> SELECT dbms_java.longname(object_name) FROM user_objects
          WHERE object_type='JAVA SOURCE';
```

このルーチンは、完全名形式で Java ソース・スキーマ・オブジェクトを表示します。短縮名が使用されていない場合には、短縮名と完全名が同じなので変換は行われません。

データベースで短縮名に変換されたかどうかを知らなくても、DBMS_JAVA パッケージの SHORTNAME() ルーチンを使用すれば、問合せ基準として完全名を使用できます。

```
SQL*Plus> SELECT object_type FROM user_objects
          WHERE object_name=dbms_java.shortname('known_fullname');
```

このルーチンは、指定された完全名のスキーマ・オブジェクトの OBJECT_TYPE を表示します。完全名がデータベース・キャラクタ・セットで表現できることが前提です。

```
SVRMGR> select * from javasnm;
SHORT                          LONGNAME
-----------------------------------------------------------------------
/78e6d350_BinaryExceptionHandl sun/tools/java/BinaryExceptionHandler
/b6c774bb_ClassDeclaration     sun/tools/java/ClassDeclaration
/af5a8ef3_JarVerifierStream1   sun/tools/jar/JarVerifierStream$1
```

Status

STATUS は、Java スキーマ・オブジェクトの妥当性を示す文字列です。ソース・スキーマ・オブジェクトは、正常にコンパイルされると VALID と表示され、クラス・スキーマ・オブジェクトは、正しく解決されると VALID と表示されます。リソースは解決されないため、リソース・スキーマ・オブジェクトは常に VALID です。

例 :USER_OBJECTS へのアクセス 次の SQL*Plus スクリプトは、USER_OBJECTS ビューにアクセスして、アップロードされた Java ソース、クラスおよびリソースの情報を表示します。

```
COL object_name format a30
COL object_type format a15
SELECT object_name, object_type, status
   FROM user_objects
   WHERE object_type IN ('JAVA SOURCE', 'JAVA CLASS', 'JAVA RESOURCE')
   ORDER BY object_type, object_name;
```

次の例のように、USER_OBJECTS の問合せにワイルドカードを使用することもできます。

```
SELECT object_name, object_type, status
   FROM user_objects
   WHERE object_name LIKE '%Alerter';
```

このルーチンは、Alerter で終わる OBJECT_NAME エントリを検索します。

USER_OBJECTS の詳細は、『Oracle8*i* Java ストアド・プロシージャ開発者ガイド』を参照してください。

公開

Oracle8*i* では、データベースにロードされた Java メソッドは、いったん公開されると、クライアントおよび SQL で起動できるようになります。次のように、Java アプリケーションのタイプに応じて、オブジェクトそのもの、または個々のメソッドを公開できます。

Java API	メソッドの公開	参照マニュアル
Java ストアド・プロシージャ	SQL DML または PL/SQL で、直接あるいは間接的にトリガーで起動する Java ストアド・プロシージャを作成した後、クラスの個々のメソッドを公開する必要があります。コール仕様を通じてメソッドへのアクセス方法を指定します。Java プログラムは多数のクラスに入っている多数のメソッドで構成されますが、コール仕様で公開される静的メソッドはわずかです。	『Oracle8*i* Java ストアド・プロシージャ開発者ガイド』
サーブレットと JavaServer Pages	JNDI ネームスペース内のサーブレットと JavaServer Pages の URL を公開します。	『Oracle8*i* Oracle Servlet Engine ユーザーズ・ガイド』 『Oracle8*i* JavaServer Pages 開発者ガイドおよびリファレンス』
CORBA および EJB の開発	CORBA オブジェクトや EJB オブジェクトにはコール仕様は使用しません。この場合には、クライアントが取り出すオブジェクト参照を公開します。クライアントがいったんオブジェクトを取り出せば、そのオブジェクトの中の特定のメソッドを起動することができます。 Oracle8*i* の CORBA と EJB の実装は、標準の CORBA および Java の方式である名前によるオブジェクトの公開をサポートしており、これらのオブジェクトに対する CORBA と Java のそれぞれの仕様を組み合せたインタフェースを提供しています。 ■ CORBA IOR は `publish` ツールによって公開します。 ■ EJB のホーム・インタフェースおよびリモート・インタフェースは `deployejb` ツールによって公開します。	『Oracle8*i* Enterprise JavaBeans 開発者ガイドおよびリファレンス』または『Oracle8*i* CORBA 開発者ガイド』

サーバー上のユーザー・インタフェース

Oracle8*i* では、すべての Java コア・クラス・ライブラリがサーバー上にあります。ユーザー・インタフェースに使用するライブラリ（`java.awt` や `java.applet` など）もサーバー上にあります。しかし、ユーザー・インタフェース処理をサーバー上で実行することは適切ではありません。世界中でインターネット・アプリケーションを実行している何千人ものユーザーが、サーバーのハードウェアに表示されるダイアログをクリックする必要がある場合を想像してみるとわかります。ユーザー・インタフェースを作成しない場合に限り、`java.awt` クラスを参照および使用する Java プログラムを作成できます。

アプレットを作成する場合は、`java.awt` とピア実装を使用してアプレットをテストします。この実装は、特定のウィンドウ・システムをサポートするプラットフォーム固有のクラスの集合で構成されます。ユーザーがアプレットをダウンロードすると、該当するクライアントのピア・ライブラリが動的にロードされ、クライアント側で使用中のオペレーティング・システムまたはウィンドウ・システムに応じて表示されます。Oracle8*i* にも、この方式が採用されています。Oracle8*i* Server で Java コードを実行し、ユーザー・インタフェースを実際に作成しようとすると、Oracle 固有のピア実装は例外 `oracle.aurora.awt.UnsupportedOperation` を発生します。

Oracle8*i* がサーバー上でのユーザー・インタフェースの作成をサポートしていないので、Java 2 Compatibility Kit では、`java.awt`、`java.awt.manual` および `java.applet` のテストを行っていません。Oracle RDBMS は、あらゆるユーザー・インタフェースをクライアント・アプリケーションに対してのみサポートします。ただし、Windows NT などでは、サーバーと同じハードウェア上でこれらのインタフェースが表示されます。サーバー側でユーザー・インタフェースをサポートしないため、オラクル社の Java Compatibility Kit の完全テストの中からは、これらのテストを除いてあります。

Java 埋込みデバイスのベンダーおよびハンドヘルド・デバイス（パーソナル Java）に関しても同様です。Java と Java Compatibility Kit の将来のリリースでは、Java サーバー・プラットフォームのベンダーがこの問題に適切に対応できるように、ユーザー・インタフェースのサポートの切分けを改良する予定です。

クラスの短縮名

Java ソース、クラスおよびリソースは、サーバーのそれぞれのスキーマ・オブジェクトに格納されます。スキーマ・オブジェクトの名前は、関連するパスやパッケージ情報を含む完全な名前指定から導き出されます。ドットはスラッシュに置き換えられます。この章では、ロードされたソース、クラスおよびリソース、生成済みのクラスおよびリソースに使用されるこれらの（スラッシュを使用した）完全な名前指定をスキーマ・オブジェクトの完全名と呼びます。

ただし、スキーマ・オブジェクト名は最大 31 文字までで、すべての文字が有効で、データベース・キャラクタ・セットの文字に変換可能であることが必要です。完全名が 31 文字以上の場合、あるいは無効な文字または変換できない文字が含まれる場合、Oracle8*i* Server は完全名を短縮名に変換して、その名前をスキーマ・オブジェクト名として採用し、完全名と短縮名とその変換方法を追跡します。完全名が 31 文字以下で、無効な文字や変換できない文字が含まれない場合は、完全名がスキーマ・オブジェクト名として使用されます。

Java クラスとメソッドの名前は SQL 識別子の最大長より長くなることがあるため、Oracle8*i* では SQL アクセスのために内部で短縮名を使用します。Oracle8*i* の `DBMS_JAVA` パッケージには、どの短縮名についても元の Java クラス名を取り出すことができるメソッドが含まれています。

```
FUNCTION longname (shortname VARCHAR2) RETURN VARCHAR2
```

このファンクションは、Java スキーマ・オブジェクトから完全名を返します。例では、なんらかの理由で無効になったクラスの完全なクラス名指定を出力します。

```
select dbms_java.longname (object_name) from user_objects
   where object_type = 'JAVA CLASS' and status = 'INVALID';
```

また、DBMS_JAVA パッケージの shortname() ルーチンを使用してデータベースに完全名を指定できます。このルーチンは完全名を入力として取り、対応する短縮名を戻します。USER_OBJECTS ビューの問合せを通じてロードしたクラスの妥当性の検証に便利です。

```
FUNCTION shortname (longname VARCHAR2) RETURN VARCHAR2
```

この機能の使用例とサーバー上の Java スキーマ・オブジェクトの特定方法は、『Oracle8*i* Java ストアド・プロシージャ開発者ガイド』を参照してください。

Oracle8*i* JVM の Class.forName()

Java 言語仕様には、Class.forName() について次のように記述されています。

> クラスの完全な名前指定を行うと、このメソッドはクラスを検索、ロードおよびリンクしようとします。成功すると、このクラスの Class オブジェクトへの参照を戻します。失敗すると、ClassNotFoundException が発生します。

クラスの参照は常に参照側クラスに代わって ClassLoader を通じて実行されます。JDK 実装と Oracle8*i* JVM 実装の違いは、クラスの検索方法です。

- JDK は、1 つの ClassLoader を使用して、環境変数 CLASSPATH で指定された一連のディレクトリ・ツリー・ルートを検索します。
- Oracle8*i* JVM は複数のリゾルバを定義します。クラスの検索方法は、このリゾルバにより定義されます。各クラスにはリゾルバが対応付けられており、それぞれのクラスは異なるリゾルバを持つ可能性があります。Class.forName() をコールするメソッドを実行すると、現在実行中のクラス（this）のリゾルバを使用してクラスが検索されます。リゾルバの詳細は、2-19 ページの「クラス依存関係の解決」を参照してください。

予期しないリゾルバを使用してクラスを検索しようとすると、予期しない結果になります。たとえば、スキーマ X のクラス X がスキーマ Y のクラス Y にクラス Z を参照するように要求する場合、クラス X のリゾルバを使用しようとするとエラーになります。参照を実行しているのはクラス Y のため、クラス Z の検索にはクラス Y に対応付けられたリゾルバが使用されます。つまり、クラスが別のスキーマに入っていて、異なるクラスの異なるリゾルバを指定すると（クラスが異なるスキーマに入っている場合には、デフォルトで起こります）、クラスを検索できない場合があります。

このようなリゾルバの問題は次のようにして解決できます。

- Class オブジェクトそのものを渡してクラス名の参照を回避します。
- Class.forName メソッドに ClassLoader を渡します。
- Oracle8*i* JVM の classForNameAndSchema メソッドにクラスとそのクラスが常駐するスキーマを渡します。

- スキーマ名とクラス名を ClassForName.lookupClass に渡します。
- スキーマ名とクラス名でオブジェクトをシリアル化します。

注意： システム・クラスが Class.forName() をコールすると、やはり予期しない動作が発生することがあります。目的のクラスが SYS または PUBLIC に常駐していないとクラスは検索されません。クラスが SYS または PUBLIC のいずれにも常駐していない場合は、クラスに PUBLIC シノニムを宣言することができます。

Class.forName への ClassLoader の提供

Oracle8*i* JVM はスキーマ内でのクラスの検索にリゾルバを使用します。クラスにはそれぞれ指定のリゾルバが対応付けられています。クラスに対応付けられたリゾルバはそれぞれ異なる場合があります。したがって、クラスの検索は対応付けられたリゾルバの定義によって決まります。ClassLoader は、指定されたクラスに基づいて検索に使用するリゾルバを判断します。Class.forName() に ClassLoader を渡すと、そのクラスのリゾルバに定義されたスキーマでクラスが検索されます。この可変の Class.forName の構文は次のとおりです。

```
Class forName (String name, boolean initialize, ClassLoader loader);
```

次の例は、現在のクラス・インスタンスのクラス・ローダーを渡す方法、またはコールする側のクラス・インスタンスのクラス・ローダーを渡す方法を示します。

例 2-3 現在のクラスからのリゾルバの取出し

Class.getClassLoader メソッドを使用して任意のインスタンスのクラス・ローダーを取り出すことができます。次の例では、インスタンス x で表されたクラスのクラス・ローダーが取り出されます。

```
Class c1 = Class.forName (x.whatClass(), true, x.getClass().getClassLoader());
```

例 2-4 コールする側のクラスからのリゾルバの取出し

oracle.aurora.vm.OracleRuntime.getCallerClass メソッドを通じて、実行中のメソッドをコールしたインスタンスのクラスを取り出すことができます。クラスを取り出した後、戻されたクラスの Class.getClassLoader メソッドをコールします。次の例では、workForCaller メソッドをコールしたインスタンスのクラスを取り出します。次に、そのクラス・ローダーが取り出され、Class.forName メソッドに対して渡されます。こうして、クラスの検索に使用されたリゾルバがコール側のクラスのリゾルバとなります。

```
void workForCaller() {
    ClassLoader c1 =
             oracle.aurora.vm.OracleRuntime.getCallerClass().getClassLoader();
    ...
    Class c = Class.forName (name, true, c1);
```

classForNameAndSchema へのクラス名およびスキーマ名の提供

検索するスキーマを認識できるリゾルバを渡すか、クラスがロードされているスキーマを渡せば、どこでクラスを検索するかという問題を解決できます。クラスのロード先スキーマが分かっている場合は、classForNameAndSchema メソッドが使用できます。Oracle8*i* JVM は DbmsJava クラスに、クラス名とそのクラスが常駐するスキーマ名の両方を取るメソッドを提供します。このメソッドは指定されたスキーマの中でクラスを検索します。

例 2-5 スキーマ名とクラス名の提供

次の例は、save メソッドにスキーマ名とクラス名を保存する方法を示します。クラス名とスキーマ名が取り出され、DbmsJava.classForNameAndSchema メソッドを使用してそのクラスが検索されます。

```
import oracle.aurora.rdbms.ClassHandle;
import oracle.aurora.rdbms.Schema;
import oracle.aurora.rdbms.DbmsJava;

void save (Class c1) {
    ClassHandle handle = ClassHandle.lookup(c1);
    Schema schema = handle.schema();
    writeNmae (schema.getName());
    writeName (c1.getName());
}

Class restore() {
    String schemaName = readName();
    String className = readName();
    return DbmsJava.classForNameAndSchema (schemaName, className);
}
```

lookupClass へのクラス名およびスキーマ名の提供

スキーマ名とクラス名が入った 1 つの文字列を oracle.aurora.util.ClassForName.lookupClass メソッドに渡すことができます。このメソッドをコールすると、指定されたスキーマでそのクラスが検索されます。文字列は次の形式で指定する必要があります。

```
"<schema>:<class>"
```

たとえば、スキーマ SCOTT の中で com.package.myclass を検索するには、次のコードを実行します。

```
oracle.aurora.util.ClassForName.lookupClass("SCOTT:com.package.myclass");
```

注意： スキーマ名は大文字で指定します。この場合、スキーマ名は大 / 小文字を区別します。

シリアル化を行う際のクラス名およびスキーマ名の提供

クラスのシリアル化を解除すると、処理の中で、名前に基づいたクラスの検索が行われます。検索が正常に行われるためには、シリアル化されたオブジェクトにクラス名とスキーマ名の両方が含まれていることが必要です。

Oracle8*i* JVM では、オブジェクトのシリアル化 / シリアル化解除に次のクラスを使用できます。

- `oracle.aurora.rdbms.DbmsObjectOutputStream`

このクラスは `java.io.ObjectOutputStream` を拡張し、適切な場所にスキーマ名を追加します。

- `oracle.aurora.rdbms.DbmsObjectInputStream`

このクラスは `java.io.ObjectInputStream` を拡張し、`DbmsObjectOutputStream` によって書き込まれたストリームを読み込みます。このクラスはどの環境でも使用できます。Oracle8*i* JVM で使用した場合、スキーマ名が読み出されて、クラスの検索時に使用されます。クライアントで使用した場合は、スキーマ名は無視されます。

Class.forName の例

次に、クラスを参照するいくつかのメソッドの例を挙げます。

- このインスタンスのクラスのリゾルバを使用するには、`lookupWithClassLoader` をコールします。このメソッドは、`from` 変数の `Class.forName` メソッドにクラス・ローダーを渡します。`from` 変数に指定されたクラス・ローダーはこのクラスにデフォルト設定されます。
- 特定のクラスのリゾルバを使用するには、明示したクラス名の後に `lookupWithClassLoader` を指定して `ForName` をコールします。`ForName` メソッドは `from` 変数を指定されたクラスに設定します。`lookupWithClassLoader` メソッドは指定されたクラスのクラス・ローダーを使用します。
- コール側クラスのリゾルバを使用するには、パラメータを指定せずに `ForName` メソッドをコールします。このメソッドは `from` 変数をコール側クラスに設定します。次に、`lookupWithClassLoader` をコールして、コール側クラスのリゾルバを使用してクラスを検索します。
- スキーマを指定してクラスを検索するには、`lookupWithSchema` メソッドをコールします。このメソッドは、`classForNameAndSchema` メソッドにクラス名とスキーマ名を渡します。

```
import oracle.aurora.vm.OracleRuntime;
import oracle.aurora.rdbms.Schema;
import oracle.aurora.rdbms.DbmsJava;

public class ForName {

  private Class from;
  /* Supply an explicit class to the constructor */
  public ForName(Class from) {
    this.from = from;
  }
  /* Use the class of the code containing the "new ForName()" */
  public ForName() {
    from = OracleRuntime.getCallerClass();
  }

  /* lookup relative to Class supplied to constructor */
  public Class lookupWithClassLoader(String name) throws ClassNotFoundException
{
    /* A ClassLoader uses the resolver associated with the class*/
    return Class.forName(name, true, from.getClassLoader());
  }

  /* In case the schema containing the class is known */
  static Class lookupWithSchema(String name, String schema) {
    Schema s = Schema.lookup(schema);
    return DbmsJava.classForNameAndSchema(name, s);
  }
}
```

オペレーティング・システム・リソースの管理

オペレーティング・システムのリソースは、どのコンピュータでも制限されています。Java は、単なるプログラミング言語ではなく、コンピューティング・プラットフォームの提供を目的としています。したがって、プラットフォームに依存しないクラスと、プラットフォーム固有のリソースにアクセスするためのフレームワークが用意されています。Java のクラス・メソッドは JVM を通じてオペレーティング・システム・リソースにアクセスします。Java のこのモデルには潜在的な問題があります。プログラマがガベージ・コレクタに全リソースの管理を任せる一方、ガベージ・コレクタは Java オブジェクトが使用するオペレーティング・システム・リソースではなく、Java オブジェクトのみを管理するためです。

さらに、Aurora JVM はデータベースに組み込まれているため、Java オブジェクトの中で保持されていオペレーティング・システム・リソースが、1 つのセッションのコールにわたって保持されると無効になる可能性があります。

以降でこの問題を取り上げます。

- オペレーティング・システム・リソースの概要
- ガベージ・コレクションとオペレーティング・システム・リソース
- コール間で影響を受けるオペレーティング・システム・リソース

オペレーティング・システム・リソースの概要

一般に、オペレーティング・システム・リソースには次のものが含まれています。

メモリー　Aurora はメモリーを内部的に管理します。オブジェクトを新規作成すると、メモリーを割り当て、オブジェクトが不要になると、メモリーを解放します。Java 言語とクラス・ライブラリでは、メモリーの割当および解放を直接行うための手段はサポートされていません。1-12 ページの「自動記憶域管理」のガベージ・コレクションの説明を参照してください。

ファイル　Java にはファイル・リソースを表すクラスが含まれています。これらのクラスのインスタンスは、ファイル・ハンドルなどのオペレーティング・システムのファイル・リソースを保持するため、1 つのセッションのコール間で無効になることがあります。

ソケット　Java にはソケット・リソースを表すクラスが含まれています。これらのクラスのインスタンスはソケット・リソースを保持します。ソケット・リソースは 1 つのセッションのコール間で無効になることがあります。コール間のソケットの保持に関する詳細は、2-40 ページの「ソケット」を参照してください。

スレッド　拡張性の問題があるため、Aurora JVM ではスレッド使用は禁止されます。ただし、データベースではマルチスレッド・アプリケーションを実行できます。Aurora の JVM スレッド・モデルの詳細は、2-41 ページの「Oracle8i JVM におけるスレッド」を参照してください。

オペレーティング・システム・リソースへのアクセス

デフォルトでは、Java ユーザーはオペレーティング・システムの大半のリソースに直接アクセスできません。システム管理者は JVM セキュリティ制限を変更して、ユーザーにオペレーティング・システムのリソースへのアクセスのパーミッションを与えることができます。JVM セキュリティは、Java 2 セキュリティに準拠するシステム・リソースで実現されます。詳細は、5-3 ページの「Java 2 セキュリティ」を参照してください。

オペレーティング・システム・リソースの存続期間

オペレーティング・システム・リソースにアクセスするには、標準のコア Java クラスとメソッドを使用します。アクセスしたリソースのアクティブな（使用可能な）期間は、リソースの種類によって異なります。

リソース	存続期間
ファイル	データベースのコールが終了すると、システムはオープンしているすべてのファイルを終了します。
メモリー	メモリーのガベージ・コレクションの詳細は、1-12 ページの「自動記憶域管理」を参照してください。
スレッド	コールが終了するとすべてのスレッドが終了します。
オペレーティング・システム・リソースに依存するオブジェクト	（スレッド・オブジェクトの定義済み継続時間など）オブジェクトの使用可能期間に関係なく、Java オブジェクトの有効期間はセッションが存続している期間になります。たとえば、Java オブジェクトを静的クラスまたはこのクラスを直接または間接的に参照するクラス変数に格納すると、セッション終了まで有効になります。使用可能期間が過ぎた Java オブジェクトを使用しようとすると、Aurora が例外を発生します。これは次のような例に当てはまります。 ■ 前のコールの終了時にクローズした `java.io.FileInputStream` から読み出そうとすると、`java.io.IOException` が発生します。 ■ 前のコールで実行したスレッド・オブジェクトが以降のコールでもアクセス可能な場合は、`java.lang.Thread.isAlive()` が FALSE になります。
ソケット	■ ソケットはコール間で存続できます。 ■ MTS サーバーの ServerSocket は、コールが終了すると終了します。 ■ 専用サーバー上の ServerSockets は、コール間で存続できます。 詳細は、2-40 ページの「ソケット」を参照してください。

ガベージ・コレクションとオペレーティング・システム・リソース

メモリーが Java オブジェクト・メモリーとオペレーティング・システムの 2 つの領域に分割されているとします。Java オブジェクト・メモリー領域にはすべてのオブジェクトと変数が入っています。オペレーティング・システム・リソースには、オペレーティング・システムが要求に応じてオブジェクトに割り当てるリソースが入っています。これらのリソースにはファイルやソケットなどが含まれます。

基本的なプログラミング規則では、プログラムから Java オブジェクトとオペレーティング・システム・リソースの両方のメモリーをクローズすることになっています。Java プログラマはガベージ・コレクタによってすべてのメモリーが解放されると誤解しています。ガベージ・コレクタは、未使用の Java オブジェクト・メモリーを収集するために作成されたものであり、オペレーティング・システム・リソースをクローズするわけではありません。オペレーティング・システム・リソースは Java オブジェクトを収集する前にプログラムでクローズする必要があります。

たとえば、オブジェクトがファイルをオープンするたびに、オペレーティング・システムはそのファイルを作成してオブジェクトにファイル・ハンドルを与えます。ファイルがクローズしないと、オペレーティング・システムはコールが終了するか、JVM が終了するまでファイル・ハンドル・リソースを保持します。これにより、ファイル・ハンドル・リソースが不

足することになります。各オペレーティング・システムのハンドルは限られています。ハンドルが不足するのを回避するには、メソッドを終了する前にリソースをクローズしてください。ソケットに連結されたストリームに関しても同様です。ソケットをクローズする前に、ソケットに連結されたストリームをクローズする必要があります。

それではガベージ・コレクタを拡張してすべてのオペレーティング・システム・リソースをクローズさせるというのはどうでしょうか？パフォーマンス上の理由から、ガベージ・コレクタは各オブジェクトにハンドルが含まれるかどうかを調べることができません。つまり、ガベージ・コレクタは Java オブジェクトおよび変数を収集しても、ハンドルを解放する適切なオペレーティング・システム・メソッドを発行することはありません。

例 2-6 は、オペレーティング・システム・リソースをクローズする方法を示します。

例 2-6 オペレーティング・システム・リソースのクローズ

```
public static void addFile(String[] newFile) {
 File inFile = new File(newFile);
 FileReader in = new FileReader(inFile);
 int i;

 while ((i = in.read()) != -1)
    out.write(i);
 /*closing the file, which frees up the operating system file handle*/
 in.close();
 }
```

in ファイルをクローズしないと、File オブジェクトはガベージ・コレクタによって回収されます。ただし、File オブジェクトがガベージ・コレクタによって回収されても、ファイルがまだクローズしていないため、オペレーティング・システムはそのファイルを使用中と判断します。

注意： Java ファイナライザを使用してリソースをクローズすることもできます。ただし、ファイナライザはタイムリな実行が保証されません。ガベージ・コレクタに時間的な余裕があると、ファイナライザは実行するためにキューに入れられます。ファイナライザ内のリソースをクローズすると、JVM が終了するまでリソースが解放されない可能性があります。メソッドの中でリソースをクローズするのが最も確実な方法です。

コール間で影響を受けるオペレーティング・システム・リソース

1 つのコール終了後、そのコールに対してローカルなリソースをクローズします。ただし、オペレーティング・システム・リソースを保持する静的オブジェクトに関しては、これらのリソースがコール終了後に受ける影響を考える必要があります。

コールが終了すると、JVM はオープンしているオペレーティング・システム・リソース（例 2-7 の場合はファイル・ハンドル）を自動的にクローズします。これは、Java オブジェクト

内のオペレーティング・システム・リソースすべてに影響します。たとえば、静的変数の中でファイルをオープンした場合、コールが終わるとそのファイル・ハンドルが自動的にクローズします。したがって、コール間にまたがって File オブジェクトを保持すると、次にファイル・ハンドルを使用しようとすると例外が発生します。

例 2-7 では、クラス Concat により複数のファイルを 1 つのファイル、outFile に書き込むことを可能にしています。最初のコールで outFile が作成されます。最初の入力ファイルがオープンされ、読み込まれて outFile に入力され、コールが終了します。outFile は静的に定義されているため、次のコールが起動される間にセッション・スペースに移動します。しかし、ファイル・ハンドル、つまり FileDescriptor は、コールが終了するとクローズします。addFile を次にコールすると、例外が発生します。

例 2-7 オペレーティング・システム・リソースの調整

```
public class Concat {
  static File outFile = new File("outme.txt");
  FileWriter out = new FileWriter(outFile);

public static void addFile(String[] newFile) {
 File inFile = new File(newFile);
 FileReader in = new FileReader(inFile);
 int i;

 while ((i = in.read()) != -1)
    out.write(i);
 in.close();
 }
}
```

これには、対処法が 1 つあります。ハンドルを有効なまま維持するには、コールが終了するたびにファイルやバッファなどをクローズし、次のコールの開始時にリソースを再オープンすることです。もう 1 つの方法は、オペレーティング・システム・リソースではなくデータベースを使用することです。たとえば、ファイルではなくデータベース表を使用します。あるいは、複数のコールにわたって存続する静的オブジェクトにオペレーティング・システム・リソースを格納せずに、そのコールに対してローカルなオブジェクトの中でのみ、オペレーティング・システム・リソースを使用します。

例 2-8 は、例 2-7 の場合のようにオペレーティング・システム・リソースを調整することなく、連結を実行する方法を示します。addFile メソッドは各コール内の outme.txt ファイルをオープンし、そのファイルに書き込まれたものがすべて最後に追加されていることを確認します。各コールの終了時にファイルはクローズします。この場合、次の 2 つのことが起こります。

1. File オブジェクトはコール外では存続しなくなります。
2. オペレーティング・システム・リソースである outme.txt ファイルはコールごとに再オープンされます。File オブジェクトを静的変数に指定していた場合、各コール内で

outme.txt をクローズすることによって、オペレーティング・システム・リソースを妥協する必要はなくなります。

例 2-8 オペレーティング・システム・リソースの正しい管理

```
public class Concat {

public static void addFile(String[] newFile) {
 /*open the output file each call; make sure the input*/
 /*file is written out to the end by making it "append=true"*/
 FileWriter out = new FileWriter("outme.txt", TRUE);
 File inFile = new File(newFile);
 FileReader in = new FileReader(inFile);
 int i;

 while ((i = in.read()) != -1)
    out.write(i);
 in.close();
 /*close the output file between calls*/
 out.close();
 }
}
```

ソケット

ソケットは、クライアントとサーバー間の接続をセットアップするために使用されます。データベース接続ごとに、接続の両側でソケットが使用されます。接続をセットアップするのはアプリケーションではなく、基礎となるネットワーク・プロトコルです。Net8 の TTC または IIOP 接続の構成方法は、4-5 ページの「Oracle8i JVM の構成」を参照してください。

また、データベースに格納された 1 つのクラスから指定の URL に接続するなど、別の接続をセットアップすることもできます。これには、接続のクライアント側とサーバー側に対応するようにソケットをインスタシエートします。

- `java.net.Socket()` コンストラクタで、クライアント・ソケットを作成します。
- `java.net.ServerSocket()` コンストラクタで、サーバー・ソケットを作成します。

ソケットは接続の両側にあります。着信コールをリスニングするサーバー側の接続に対応するのは `ServerSocket` です。要求を送るクライアント側の接続に対応するのは `Socket` です。JVM に定義されたソケットを使用するには、次のような制限があります。MTS サーバー内の `ServerSocket` インスタンスはコール間にまたがって存続できません。

`Socket` 接続のクライアント側は送信側になるため、`Socket` インスタンスは MTS サーバーまたは専用サーバー内のコール間にわたって対応が可能です。

`ServerSocket` 接続のサーバー側はリスナーです。

- 専用サーバー － `ServerSocket` は専用サーバー内でのみコール間にわたってリスニングできます。専用サーバーは 1 つのクライアントにのみ対応します。
- MTS サーバー － `ServerSocket` は MTS サーバー内の 1 つのコールが終了するとクローズします。MTS は共有サーバーを使用し、共有サーバーはコールが終わるたびに別のクライアントに対応します。作成元以外のコールで `ServerSocket` を使用しようとすると、「ソケットはクローズしました」というメッセージとともに I/O 例外が発生します。

Oracle8*i* JVM におけるスレッド

Aurora JVM は非プリエンプティブ・スレッド・モデルを実装します。このモデルでは、JVM は 1 つのオペレーティング・システム・スレッドですべての Java スレッドを実行します。JVM はラウンドロビン形式でスレッドをスケジューリングし、ブロックされた場合にのみスレッドを切り替えます。たとえば、`Thread.yield()` メソッドをコールしたり、`mySocket.read()` をコールしてネットワーク・ソケットを待機すると、ブロックが発生します。

Oracle8*i* JVM のスレッド・モデルの利点	不利な点
■ プログラムが容易 ■ スレッド切替えにシステム・コールが不要なので、Java 仮想マシンでの実装が効率的 ■ JVM はプリエンプティブ JVM をハングさせ、ランタイム例外を発生させるデッドロックを検出できるので、より安全	■ 並行性がない ■ 移植性がない ■ スレッドがブロックしている場合のロックにシステム・コールが必要なため、パフォーマンスが低下する恐れがある ■ 効率的なマルチスレッド・メモリーの割当てには大量のメモリー・プールが必要となるため、メモリーの拡張性が低下する

Oracle がこのモデルを採用したのは、Java アプリケーションは単一プロセッサ・システムで作成されても、マルチプロセッサ・システムで作成されてもまったく同じように動作するからです。また、Aurora はデータベースに組み込まれているため、従来の JVM よりも高い並行性を提供するので、Java スレッドの間で並行性がないことも問題になりません。

Oracle Server はプリエンプティブに JVM セッションをスケジューリングするので、アプリケーション・ロジックの中でスレッドを使用する必要もありません。トランザクションを同時に数百個または数千個サポートする必要がある場合は、それぞれの JVM でトランザクションを開始します。これは、Oracle8*i* JVM 上でセッションを作成するときもまったく同じです。Java 仮想マシン間の調整とデータ転送は、Oracle データベース・サーバーの通常のトランザクション機能が行います。一般的な Java 仮想マシンのフットプリントは 6MB から 8MB であるのに対して、Oracle Server はそれぞれで使用するメモリーが 40KB 以下の Java 仮想マシンを何千も作成できるので、拡張性の問題もありません。

1 つのスレッドが完了するか、スレッドのブロックが発生するまで、スレッドは Aurora 内で管理されます。ネットワーク・ソケットを生成または待機することでスレッドのブロックが発生すると、JVM は別のスレッドに対応します。ただし、ブロックが発生しなければ、JVM はスレッドが完了するまで対応します。

Aurora JVM には、パフォーマンスとスレッド管理を向上させる次のような機能が追加されています。

- システム・コールを最小限に抑制。Aurora では通常のシステム・コールのかわりに非システム・ソリューションが採用されています。たとえば、モニター同期ブロックやメソッドに入るのにシステム・コールは必要ありません。
- デッドロックの検出
  - Aurora はスレッド間のデッドロックを監視します。デッドロックが発生すると、Aurora はいずれかのスレッドを終了させて、`oracle.aurora.vm.DeadlockError` 例外を発行します。
  - シングル・スレッド・アプリケーションは中断できません。スレッドが 1 つしかないアプリケーションを中断しようとすると、`oracle.aurora.vm.LimboError` 例外が発生します。

スレッドのライフ・サイクル

実行されているスレッドが 1 つのみのときは、次のイベントが発生すると、コールが終了します。

1. スレッドがそのコール元に戻る。
2. Java コードでキャッチされない例外が発生する。
3. `System.exit()`、`oracle.aurora.vm.OracleRuntime.exitCall()` または `oracle.aurora.mts.session.Session.THIS_SESSION().endSession()` メソッドがコールされる。

初期スレッドが他の Java スレッドを作成して起動した場合は、コール終了規則が多少複雑になります。この場合は、次のどちらかの方法でコールを終了します。

1. メイン・スレッドがそのコール元に戻るか、またはこのスレッドでキャッチされない例外を発生して、その他のすべての非デーモン・スレッドの実行を完了します。非デーモ

ン・スレッドは、その初期メソッドから戻るか、スレッドでキャッチされない例外が発生すると終了します。

2. いずれかのスレッドで `System.exit()`、`oracle.aurora.vm.OracleRuntime.exitCall()` または `oracle.aurora.mts.session.Session.THIS_SESSION().endSession()` メソッドをコールします。

メソッドから戻るか、例外がキャッチされなかったためにコールが終了すると、Aurora はすべてのデーモン・スレッドで `ThreadDeathException` を発生します。`ThreadDeathException` はスレッドの実行を停止します。

`System.exit()`、`oracle.aurora.vm.OracleRuntime.exitCall()` または `oracle.aurora.vm.oracleRuntime.exitSession()` のコールによってコールが終了すると、Aurora はコールをただちに終了し、すべてのスレッドを終了します。しかし、`ThreadDeathException` は発生しません。

単一コールの実行中、Java プログラムが再帰的に Java コードを実行することがあります。たとえば、プログラムで JDBC または SQLJ を使用して SQL 問合せを発行すると、Java で書かれたトリガーが起動することがあります。コールの存続期間に関するこれまでの説明は、Java コードへの最上位コールに関する内容です。再帰的コールに関する説明ではありません。たとえば、再帰的コールで `System.exit()` をコールすると、再帰コールのみではなく、Java への最上位コール全体が終了します。

3

データベースでの Java のコール

第 2 章「Oracle8i での Java アプリケーションの作成」では、Oracle8*i* で Java アプリケーションを作成および配置する基本を説明しました。この章では、データベースにおいて Java を起動する方法を例をあげながら概説していきます。

- 概要
- Java メソッドのコール
- SQLJ および JDBC を使用したデータベースの問合せ
- サーバー・アプリケーションのデバッグ
- サーバー上での実行を見分ける方法
- サーバー上の出力のリダイレクト

概要

Oracle8*i* では、次のいずれかの方法で Java を使用します。

- Java メソッドのコール － データベースにロードされたクラスで Java メソッドを起動します。これには、Java ストアド・プロシージャ、サーブレット、JavaServer Pages、CORBA および EJB が含まれます。

> **注意：** JavaServer Pages とサーブレットの詳細は、『Oracle8*i* JavaServer Pages 開発者ガイドおよびリファレンス』と『Oracle8*i* Oracle Servlet Engine ユーザーズ・ガイド』を参照してください。

- SQLJ および JDBC を使用したデータベースの問合せ － JDBC または SQLJ を使用して、Java クライアントからデータベースの問合せを実行できます。

Oracle8*i* における Java の開発手法は、段階的に習得していくことをお薦めします。データベースで Java を起動する最も簡単な方法は、Java ストアド・プロシージャを使用する方法です。すでにこの方法を習得している場合は、CORBA および EJB アプリケーションの説明に進んでください。

1. 2-14 ページの「Java クラス・メソッドの実行準備」および『Oracle8*i* Java ストアド・プロシージャ開発者ガイド』で説明している簡単な Java ストアド・プロシージャを作成する手順を習得してください。これには、Java クラスの作成、リゾルバの決定、データベースへのクラスのロード、およびクラスの公開が含まれます。
2. Java から SQL データにアクセスして操作する方法を理解する必要があります。大半の Java サーバー・プログラム、および Oracle8*i* で実行する Java プログラムはデータベース上のデータを操作します。そのための標準 API として、JDBC と SQLJ の 2 つが用意されています。SQLJ は JDBC を基盤としているので、コードで SQLJ のみを使用する場合でも、この 2 つの関係を理解する必要があります。
3. また、Java ロジックをクライアントとサーバー間で、または N 層アーキテクチャで配布する場合は、Oracle8*i* における CORBA と EJB の動作方法を理解する必要があります。CORBA と EJB は、ロジックの分散化に対するインターネット標準にのっとった、最も簡単なソリューションです。このソリューションでは、トランザクション・アプリケーションにコンポーネント・ベースの開発手法を適用できます。また、EJB と CORBA では、Oracle8*i* の Java ストアド・プロシージャおよび JDBC の機能を利用します。

Java は、ストアド・プロシージャを作成するための単純かつ汎用的な言語です。JDBC と SQLJ を使用すれば、Java で SQL データにアクセスできます。JDBC と SQLJ は、SQL 操作と概念、Java タイプと SQL タイプ間の変数のバインド、および Java クラスを SQL タイプにマップするクラスをサポートしています。したがって、クライアント側でもサーバー側でも、そのまま実行できる移植可能な Java コードを書くことができます。通常、JDBC と SQLJ を使用すると、クライアントとサーバー間の切分けが明確になります。つまり、SQL

操作はサーバー側で行われ、アプリケーション・プログラムのロジックはクライアント側に存在します。

より複雑な Java プログラムを作成するときは、プログラム・ロジックが実行される場所を制御することによって、パフォーマンスと拡張性を向上できます。ネットワーク上の通信量を減らし、SQL データの参照をローカライズできます。JDBC および SQLJ には、このような目標を実現する方法が用意されています。ただし、Java アプリケーションでオブジェクト・モデルを利用しようとすると、SQL データへのアクセスおよび操作時間に比べて、Java コードの実行時間が膨大になります。したがって、インターネット・アプリケーションでは、Java オブジェクトがどこに格納されどこで実行されるかを、理解し指定する必要があります。これで、CORBA と Enterprise JavaBeans の世界に入る準備ができました。

Java メソッドのコール

クライアントが Java メソッドをコールする方法は、Java アプリケーションのタイプにより異なります。以降では、データベースにロードしてクライアントからアクセスできる Java クラスを作成する際に、使用する各種 Java API を説明します。

- Java ストアド・プロシージャの利用
- CORBA と EJB を使用した分散オブジェクトの使用
- Remote Method Invocation（RMI）の使用
- Java ネイティブ・インタフェース（JNI）サポートの使用
- SQLJ および JDBC を使用したデータベースの問合せ

Java ストアド・プロシージャの利用

Java ストアド・プロシージャの実行方法は PL/SQL の場合とほぼ同じです。通常、Java ストアド・プロシージャは、トリガーまたは DML コールの結果としてコールされるため、データベース操作に付随して実行されます。

Java ストアド・プロシージャをコールするには、コール仕様を通じて Java ストアド・プロシージャを公開する必要があります。次の例は、「Hello world」をエコーする簡単な Java ストアド・プロシージャを作成、解決、ロードおよび公開する手順です。

1. Java クラスを書きます。

   クラス `Hello` と、このクラスの唯一のメソッド `Hello.world()` を定義します。このメソッドは、文字列「`Hello world`」を戻します。

```
public class Hello
{
   public static String world ()
   {
      return "Hello world";
   }
}
```

2. このクラスをクライアント・システムでコンパイルします。Sun Microsystems の JDK などを使用して、次のように Java コンパイラ `javac` を起動します。

   ```
   javac Hello.java
   ```

   通常、`CLASSPATH` は `javac` コマンドラインで指定することをお薦めします。特に、シェル・スクリプトまたは Make ファイルの作成時は、コマンドラインでの指定をお薦めします。Java コンパイラによって Java バイナリ・ファイル（この例の場合は `Hello.class`）が作成されます。

   この Java コードが実行される場所に注目してください。`Hello.class` をクライアント・システムで実行すると、必要なすべてのコア・クラスが `CLASSPATH` で検索されます。この結果、次のいずれかから依存クラスが検索されます。

   - ディレクトリ内の個々のファイル。このディレクトリは CLASSPATH で指定されます。
   - `.jar` ファイルまたは `.zip` ファイルの中。このディレクトリは CLASSPATH で指定されます。

3. クラスに使用するリゾルバを決めます。

   ここでは、`Hello.class` がサーバーにロードされ、Java スキーマ・オブジェクトとしてデータベースに格納されています。サーバー上で `Hello.class` の `world()` メソッドを実行すると、リゾルバを使用して、必要なサポート・クラス（`String` など）が検索されます。この例では、デフォルトのリゾルバが使用されます。デフォルトのリゾルバは、最初にカレント・スキーマ、次に PUBLIC スキーマで、クラスを検索します。すべてのコア・クラス・ライブラリは PUBLIC スキーマにあります。`java.lang` パッケージもこのスキーマにあります。実行時以前に問題の存在を発見するためには、他のリゾルバを指定し、また `loadjava` を使用して解決を強制的に行う必要があります。リゾルバと `loadjava` の詳細は、2-19 ページの「クラス依存関係の解決」または『Oracle8*i* Java Tools リファレンス』を参照してください。

4. `loadjava` を使用して Oracle8*i* にクラスをロードします。ユーザー名とパスワードを指定してください。

   ```
   loadjava -user scott/tiger Hello.class
   ```

5. コール仕様を通じてストアド・プロシージャを公開します。

   SQL コールで Java の静的メソッドをコールするには、コール仕様によってメソッドを公開する必要があります。コール仕様は SQL に対する定義であり、メソッドがとる引数とメソッドが戻す SQL 型を指定します。

   SQL*Plus では、次のようにデータベースに接続し、`Hello.world()` に対する最上位のコール仕様を定義します。

```
SQL> connect scott/tiger
connected
SQL> create or replace function HELLOWORLD return VARCHAR2 as
  2  language java name 'Hello.world () return java.lang.String';
  3  /
Function created.
```

6. ストアド・プロシージャをコールします。

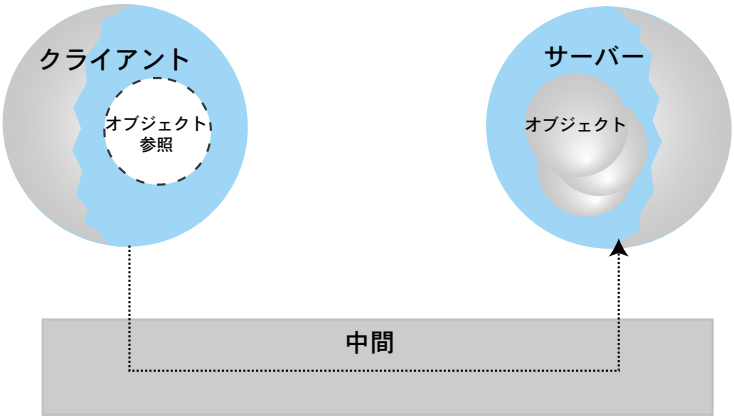
```
SQL> variable myString varchar2[20];
SQL> call HELLOWORLD() into :myString;
Call completed.
SQL> print myString;

MYSTRING
---------------------------------------
Hello world

SQL>
```

   `call HELLOWORLD() into :myString` 文は Oracle8*i* の最上位コールを実行します。Oracle 固有の `select HELLOWORLD from DUAL` も使用できます。SQL と PL/SQL では、ストアド・プロシージャをどの言語（Java、PL/SQL など）で作成しても同じです。コール仕様によって、言語間のコールを一貫した方法で結び付ける手段が使用できます。コール仕様を必要とするのは、トリガーで起動するエントリ・ポイント、または SQL および PL/SQL コールのみです。さらに、JDeveloper で自動的にコール仕様を作成することもできます。

   Java ストアド・プロシージャ、トリガー内での Java の使用方法、コール仕様、権限モデルおよび言語間コールの詳細は、『Oracle8*i* Java ストアド・プロシージャ開発者ガイド』を参照してください。

CORBA と EJB を使用した分散オブジェクトの使用

通常、1 つのプログラムのロジックを分散する場合は、クライアント、中間層およびデータベース・サーバーからなる 3 層アーキテクチャが最適です。



クライアント層	通常は、中間層から提供される情報の表示専用です。
中間層	クライアントとサーバー間の通信を容易にします。通常は、サーバー・オブジェクトを管理します。パラメータをマーシャリングおよびアンマーシャリングして値を返します。
サーバー層	ビジネス・ロジックまたはアプリケーション・ロジックを実行します。

3 層モデルのサーバー・オブジェクトはビジネス・ロジックを実行します。これにはデータベースにアクセスして SQL 問合せを実行することが含まれる場合とそうでない場合があります。Oracle8*i* では、サーバー・オブジェクトがデータベースにアクセスするための、分散アプリケーションにおける物理的な中間層が不要になります。Oracle8*i* は 3 層の論理アーキテクチャを維持していますが、中間層とデータベース・サーバー層を組み合せることによって、物理的には 2 層アーキテクチャを実現しています。このアーキテクチャの柔軟性は、クライアントがネットワークを介してサーバーと対話し、情報を Web ブラウザに表示するインターネット・アプリケーションに最適です。これらのサーバーは、そのかわりとして、独自のクライアント / サーバー間の対話動作に共用、連携されることによって、イントラネットまたはインターネット・アプリケーションの Web ベースのクライアントに情報を提供します。

アプリケーションに 2 階層分散オブジェクト・アプローチを使用することで、CORBA または EJB API を使用できます。

- CORBA では、言語に依存しない方法で Interface Definition Language（IDL）を使用して、コンポーネントと呼ばれる一連のグループにアクセスする方法およびこれらのグループを使用する方法を指定します。Oracle8*i* は、サーバー上でそれぞれ Java 仮想マシンを実行しているかのように各クライアントと対話します。Oracle8*i* JVM では単一の

ORB が複数のクライアント要求に対応することはありません。そのかわり、Oracle8*i* JVM では、Oracle8*i* のマルチスレッド・サーバー（MTS）アーキテクチャを利用して、セッションごとに ORB を用意します。

- Enterprise JavaBeans は次のものに依存します。
  - Java クラス定義はコンポーネントへのインタフェースを指定します。
  - RMI 形式のディプロイメント・ディスクリプタはトランザクション・アプリケーションや保護アプリケーションでコンポーネントをどのように処理するかを定義します。

  EJB プログラマはビジネス・ロジックとコンポーネントへのインタフェースを作成するだけでよく、配置ツール `deployejb` にコンポーネントのロードと公開を任せることができます。IDL に関する知識は不要です。この Java ベースの移植可能なサーバー・フレームワークは、Java ベースの 3 層アプリケーションに対する高速でスケーラブルな、簡単なソリューションです。

CORBA と EJB は相補関係にあります。Oracle8*i* JVM 実装の Enterprise JavaBeans 1,1 仕様は、CORBA のサポートとサービスを基盤にしています。

IIOP 移送

クライアントが Net8 経由で通信するセッションと異なり、CORBA および EJB セッションへのアクセスは IIOP を介します。CORBA および EJB セッションは複数のクライアント接続に対応できます。通常スケーラブルなアプリケーションはクライアント / サーバーの対話ごとに 1 つのセッションを提供できますが、複数のクライアントに対応できればセッションの柔軟性が高まります。IIOP により、分散通信におけるコールアウト、コールバックおよびループバックが可能になります。

ネーミング

コンポーネントへのアクセスには、ネーム・サービスを使用します。ネーム・サービスは、ファイル・システムに似たツリーであり、オブジェクトを名前で格納できます。CORBA または EJB オブジェクトをネームスペースに入れると、オブジェクトが公開されます。

Oracle8*i* では 2 つのネーミング・プロトコルがサポートされます。

- Java Naming and Directory Interface（JNDI）－このパッケージを使用すると、複数のネーム・サービスへのインタフェースが統一化されます。JNDI には、プラットフォームに依存するファイル・システムへのアクセスをプラットフォームに依存せずに抽象化する手段が用意されています。
- CORBA の CosNaming － CORBA プログラマは、CosNaming を使用してアプリケーションをブートストラップする方法に精通しています。クライアント Java コードには、サーバー上に常駐するオブジェクトへのハンドルが含まれています。これらのオブジェクトはネーム・サービスを通じて参照できます。ORB はネーム・サービスを提供しますが、これは、クライアント・アプリケーションのブートストラップ時にサーバー・オブジェクトを検索する場合、ORB が実行されていることを前提としています。Oracle8*i*

JVM は、CORBA の CosNaming に基づく起動サービスを提供します。セッション内で CORBA オブジェクトを参照および起動するには CosNaming へのインタフェースとして JNDI 内の URL ベースの名前を使用します。このネームスペースによって、セッションが URL に直接取り込まれるので、クライアントは複数のセッションを簡単に操作できます。Oracle8*i* JVM とのセッションを確立し、標準 JNDI と CORBA CosNaming によって参照可能となった Oracle8*i* データベース内の常時使用可能なオブジェクトを使用することによって、ブートストラップが行われます。ほとんどの CORBA アプリケーションのように、Inter-ORB References（IOR）を使用することはありません。

Enterprise JavaBeans の作成と配置

CORBA および EJB アプリケーション開発の詳細は、『Oracle8*i* Enterprise JavaBeans 開発者ガイドおよびリファレンス』と『Oracle8*i* CORBA 開発者ガイド』を参照してください。ここでは、EJB コンポーネントを作成する方法の一例を紹介します。この例では、Oracle RDBMS に格納された従業員レコードを参照する EmployeeBean を作成します。

1. ホーム・インタフェースを作成します。ホーム・インタフェースはサーバーに常駐し、サーバー上に EJB のインスタンスを作成するときに使用します。ホーム・インタフェースは、EJBHome を拡張した Java インタフェースです。ホーム・インタフェースは、クライアントから見えるネームスペースで公開される、唯一のオブジェクトです。クライアントは JNDI を通じてコンポーネントにアクセスします。

2. リモート・インタフェースを作成します。リモート・インタフェースでは、EJB に実装するメソッドを指定します。指定されるメソッドには、クライアントからコールするインスタンス・メソッドなどがあります。リモート・インタフェースは、EJBObject を拡張した Java インタフェースです。このインタフェースを使用すると、Bean に実装されているメソッドを指定できます。

3. リモート・インタフェースで定義した Bean のクラスとメソッドを実装します。EmployeeBean の例では、`getEmployee()` が唯一のメソッドです。SessionBean インタフェースを実装するクラスを作成することによって、Bean を作成します。

4. ディプロイメント・ディスクリプタを作成します。ディプロイメント・ディスクリプタは Bean の属性として、トランザクションのプロパティやセキュリティ対策などを指定します。指定した属性は、`deployejb` ツールによって確実にサーバーで適用されます。

5. EJB を配置します。EJB を配置するときには `deployejb` ツールで次のことを実行します。

   - ホーム・インタフェースと EJB をサーバーに配置します。
   - ホーム・インタフェースをネームスペースで公開します。
   - サーバー側で Java コードを生成して、ディプロイメント・ディスクリプタに指定されたトランザクションおよびセキュリティを管理します。
   - クライアントから Bean のリモート機能にアクセスするためのスタブ・インタフェースが生成され、返されます。

EJB の使用方法

EJB を作成して配置した後、クライアント・プログラムから使用できるようにする必要があります。N 層アプリケーションでは、EJB をサーバー間でも使用できます。この場合、あるサーバーではクライアントであったものが、他のクライアントのサーバーになることがあります。クライアント・コードで、次の操作を行う必要があります。

1. サーバーに常駐しているホーム・インタフェース・オブジェクトを検索します。オブジェクトの検索には、Java の標準 JNDI ルックアップ機能を使用します
2. サーバーに対してクライアントを認証します。EJB と CORBA クライアントによるデータベース・セッションの使用方法は、他の Oracle クライアントの場合と同じです。セッションを開始するには、正当なユーザーであることをサーバーに知らせる必要があります。安全な認証方式は複数用意されています。
3. Bean のインスタンスを起動します。JNDI で検索したオブジェクトはホーム・インタフェースなので、その `create()` メソッドの 1 つを使用して、起動した EJB のインスタンスを返します。
4. Bean のメソッドをコールします。Bean のメソッドをコールすると、サーバー上で実際にメソッドが実行され、該当するパラメータと戻りオブジェクトが、IIOP 接続を介して、透過的に（コピーによって）送られます。EJB が戻すオブジェクトはすべてシリアル化可能であることが必要です。つまり、`java.io.Serializable` を実装する必要があります。

これらの手順の詳細は、『Oracle8*i* Enterprise JavaBeans 開発者ガイドおよび リファレンス』と『Oracle8*i* CORBA 開発者ガイド』のマニュアルを参照してください。Java IDE では、Oracle の JDeveloper と同様、配置およびディスクリプタの処理を自動化し、簡略化できます。

セッション・シェル

セッション・シェルは、Java ストアド・プロシージャおよび CORBA を使用して完全に Java で書かれたツールの一例です。セッション・シェルは、UNIX シェル・コマンドを使用することでセッション内の CORBA を通じて参照できる、サーバー常駐型のオブジェクトと対話します。詳細は、『Oracle8*i* Enterprise JavaBeans 開発者ガイドおよびリファレンス』と『Oracle8*i* CORBA 開発者ガイド』を参照してください。このツールで、CORBA を使用して開発者とエンド・ユーザーの負担を軽減するツールを作成する方法が分かります。

セッション・シェルは、シェルに似たサーバー・インタフェースを提供します。このシェルを介して、ユーザーは `mkdir`、`ls`、`rm` などの使い慣れた UNIX コマンドでセッションのネームスペースを操作できます。また、セッション・シェルでは、`java` コマンドを使用して、サーバー上で Java プログラムを実行できる便利な方法を提供しています。セッション・シェルの `java` コマンドには、クラス名とユーザーが入力する任意の引数を指定できます。セッション・シェルはクラスから静的な `main(String[])` メソッドをコールし、サーバーで Java プログラムを実行します。`System.out` と `System.err` は捕捉され、透過的にユーザーのコンソールにリダイレクトされます。

Remote Method Invocation（RMI）の使用

Oracle8*i* JVM は Java の Remote Method Invocation（RMI）を完全にサポートしています。すべての RMI クラスと `java.net` のサポートが用意されています。通常、Oracle8*i* JVM アプリケーションでは、RMI はそれほど有用ではありませんし、スケーラブルでもありません。リモート・オブジェクトのメソッドをコールするには、CORBA と EJB が好ましい API です。Sun Microsystems の RMI サーバーは、Oracle8*i* JVM プラットフォームで機能します。ただし、Sun Microsystems の RMI サーバーはオペレーティング・システムのソケットを使用しており、プレゼンテーション経由ではアクセスできません。したがって、RMI サーバーは Oracle8*i* JVM への 1 つのコールとして実行されます。この RMI サーバーは、Java 言語レベルのスレッドに強く依存しています。一方、Oracle8*i* の ORB と EJB はデータベース・サーバー・プロセスを使用して拡張性を実現しています。RMI サーバーをプレゼンテーションを使用して実装することは可能ですが、現行では CORBA と EJB がこの役割を果たしています。

注意： プレゼンテーションとは、Net8 または IIOP の着信接続をデータベースへに受け入れるオブジェクトです。詳細は、4-5 ページの「Oracle8i JVM の構成」を参照してください。

Java ネイティブ・インタフェース（JNI）サポートの使用

Java ネイティブ・インタフェース（JNI）は、Java のネイティブ・メソッドを書き、Java 仮想マシンをネイティブ・アプリケーションに埋め込むための標準プログラミング・インタフェースです。JNI の主な目的は、プラットフォーム固有のネイティブ・ライブラリを使用する Java アプリケーションのバイナリ互換を提供することです。

Oracle は、Oracle8*i* JVM アプリケーションでの JNI の使用をサポートしていません。JNI を使用すると、アプリケーションが 100% pure Java でなくなります。その結果、他のプラットフォームで使用するには移植作業が必要になります。また、ネイティブ・メソッドはサーバーのクラッシュ、セキュリティ違反およびデータの破壊を引き起こす可能性があります。

SQLJ および JDBC を使用したデータベースの問合せ

2 つのプロトコルのどちらかを使用して、Java クライアントからデータベースへの問合せを実行できます。どちらのプロトコルも指定されたユーザー名 / パスワードでデータベースとのセッションを確立し、データベースに対して SQL 問合せを実行します。

JDBC　より複雑または動的な SQL 問合せにはこのプロトコルを使用します。JDBC では、セッションを確立し、問合せを作成するなどの作業が必要になります。

SQLJ　静的で簡単な SQL 問合せにはこのプロトコルを使用します。一般に、SQLJ は 1 行で、列名が分かっている既知の表に対して実行します。

JDBC

JDBC は、Sun Microsystems 社が開発した業界標準の API であり、SQL 文を Java メソッドの引数として埋め込むことができます。JDBC は、X/Open SQL Call Level Interface を基盤とし、SQL92 Entry Level の標準に準拠しています。オラクル社などの各ベンダーは、Sun Microsystems 社の `java.sql` パッケージをインタフェースとして実装することによって、独自の JDBC 実装を作成しています。Oracle はこれらの標準インタフェースを実装した 3 つの JDBC ドライバを用意しています。

1. JDBC Thin ドライバは、100% pure Java ソリューションで、クライアント側アプリケーションまたはアプレットに使用でき、Oracle クライアントをインストールする必要がありません。
2. JDBC OCI ドライバ（OCI 8 または OCI 7）はクライアント側アプリケーションに使用でき、Oracle クライアントをインストールする必要があります。
3. Oracle8*i* Server に組み込まれたサーバー側 JDBC ドライバ。

JDBC による開発では、操作手順を段階的に記述する必要があります。つまり、目的の SQL 操作に合った statement オブジェクトを作成し、SQL 操作にバインドするローカル変数を代入し、操作を実行するというように、順次展開する必要があります。多くのアプリケーションでは、この方法で十分ですが、複雑な文の場合は手間がかかります。動的な SQL 操作の場合は、実行時まで操作内容がわからないため、JDBC を使用する必要があります。ただし、代表的なアプリケーションでは、この SQL 操作をあまり採用していません。

SQLJ

SQLJ では、1 回の操作で、静的な SQL 操作を業界標準に規定された方式で直接 Java ソース・コードに埋め込むことができます。JDBC のように段階的に展開する必要はありません。Oracle SQLJ は ANSI 規格 X3H2-98-320 に準拠しています。

SQLJ の構成要素は、トランスレータ（標準 SQLJ プログラム構文をサポートするプリコンパイラ）とランタイム・コンポーネントです。SQLJ ソース・コードを `.sqlj` ファイルに作成してから、このコードをトランスレータで処理します。トランスレータは SQLJ ソース・コードを標準 Java ソース・コードに変換します。SQL 操作は、SQLJ ランタイムへのコールに変換されます。Oracle SQLJ 実装では、トランスレータが Java コンパイラを起動して Java ソース・コードをコンパイルします。Oracle SQLJ アプリケーションを実行すると、SQLJ ランタイムが JDBC をコールして、データベースとやり取りします。

SQLJ では、実行前に SQL 文のエラーをキャッチできます。一方、JDBC コードは純粋な Java なので、そのままコンパイルされます。このコンパイラは SQL を理解しないため、SQL エラーを検出しません。一方、SQLJ コードを変換すると、トランスレータが埋め込まれている SQL 文のセマンティクスと構文を分析し、SQL エラーをキャッチします。開発時にエラーをキャッチできるため、エンド・ユーザーでのアプリケーション実行時にエラーが発生することはありません。

JDBC と SQLJ の比較例

次に、単純な操作例を最初に JDBC コードで示し、次に SQLJ コードで示します。

JDBC:

```
// (Presume you already have a JDBC Connection object conn)
// Define Java variables
String name;
int id=37115;
float salary=20000;

// Set up JDBC prepared statement.
PreparedStatement pstmt = conn.prepareStatement
   ("select ename from emp where empno=? and sal>?");
pstmt.setInt(1, id);
pstmt.setFloat(2, salary);

// Execute query; retrieve name and assign it to Java variable.
ResultSet rs = pstmt.executeQuery();
while (rs.next()) {
   name=rs.getString(1);
   System.out.println("Name is: " + name);
}

// Close result set and statement objects.
rs.close()
pstmt.close();
```

1. Java の変数 `name`、`id` および `salary` を定義します。
2. プリコンパイルされた SQL 文を定義しています（ここでは、データベースへの接続がすでに確立されていると仮定しているため、接続オブジェクトの `prepareStatement()` メソッドを使用できます）。

   SQL 文の値を動的に設定する必要がある場合は、いつでもプリコンパイルされた SQL 文を使用できます（変数の値を変えることで同じプリコンパイルされた SQL 文を繰返し使用できます）。プリコンパイルされた SQL 文中の疑問符は Java 変数のプレースホルダです。コードの `pstmt.setInt()` 行と `pstmt.setFloat()` 行で値を指定します。最初の「?」は `int` 変数 `id`（値 37115）に設定されます。次の「?」は `float` 変数 `salary`（値 20000）に設定されます。
3. 問合せが実行され、JDBC の結果セット・オブジェクトにデータが返されます。（問合せ結果のデータは、結果セットを使用して収集できます。）
4. 目的のデータ（名前）が結果セットから取得され、出力されます。結果セットは、通常複数のデータ行で構成されますが、この例では 1 行のみです。

次に、同じ処理を行う SQLJ コードを示します。すべての SQLJ 文、つまり宣言と実行文は、`#sql` トークンで始められることに注意してください。

SQLJ:

```
String name;
int id=37115;
float salary=20000;
#sql {select ename into :name from emp where empno=:id and sal>:salary};
System.out.println("Name is: " + name);
```

SQLJ では、SQL 文を Java コードに直接埋め込めるばかりでなく、Java のホスト式（つまり、バインド式）をそのまま SQL 文で使用できます。最も簡単なホスト式はこの例のように単純な変数ですが、複雑な式も使用できます。各ホスト式は「:」（コロン）で始められます。この例では、Java のホスト式 `name`、`id` および `salary` を使用しています。SQLJ では、ホスト式を使用できるので、取得するデータが 1 行のときは、結果セットなどを使用する必要がありません。

完全な SQLJ の例

ここでは、簡単な SQLJ プログラムの完全な例を示します。

```
import java.sql.*;
import sqlj.runtime.ref.DefaultContext;
import oracle.sqlj.runtime.Oracle;
#sql iterator MyIter (String ename, int empno, float sal);

public class MyExample
{
   public static void main (String args[]) throws SQLException
   {
      Oracle.connect
         ("jdbc:oracle:thin:@oow11:5521:sol2", "scott", "tiger");

      #sql { insert into emp (ename, empno, sal)
         values ('SALMAN', 32, 20000) };
      MyIter iter;

      #sql iter={ select ename, empno, sal from emp };
      while (iter.next()) {
         System.out.println
            (iter.ename()+" "+iter.empno()+" "+iter.sal());
      }
   }
}
```

1. イテレータを宣言します。SQLJ では、JDBC の結果セットを厳密に型指定した、イテレータと呼ばれるものを使用します。JDBC の結果セットと異なり、イテレータでは、列の数と各列のデータ型を指定できます。イテレータの型は、次のように事前に定義しておく必要があります。

```
#sql iterator MyIter (String ename, int empno, float sal);
```

SQLJ では、この宣言によってイテレータ・クラス MyIter が作成されます。型 MyIter のイテレータに格納できる結果は、最初の列が Java の String にマッピングされ、2 番目の列が Java の int にマッピングされ、3 番目の列が Java の float にマッピングされている結果のみです。この定義では、3 つの列の名前も指定しています。データベース内の表の列名に一致する名前 ename、empno および sal がそれぞれ指定されています。MyIter は名前指定イテレータです。列名の指定が不要な位置指定イテレータに関する詳細は、『Oracle8*i* SQLJ 開発者ガイドおよびリファレンス』の第 3 章を参照してください。

2. データベースに接続します。

```
Oracle.connect("jdbc:oracle:thin:@oow11:5521:sol2","scott", "tiger");
```

Oracle SQLJ には、Oracle クラスがあり、その connect() メソッドによって次の 3 つの重要な処理が行われます。

a. SQLJ がデータベースへのアクセスに使用する Oracle JDBC ドライバを登録します。

b. 指定された URL（ホスト oow11、ポート 5521、SID so12、thin JDBC ドライバ）で、指定されたスキーマ（ユーザー scott、パスワード tiger を使用して）のデータベース接続を開きます。

c. この接続を SQLJ 文のデフォルト接続として確立します。JDBC 文では、接続オブジェクトをそれぞれ明示的に指定する必要がありますが、SQLJ 文ではデフォルトの接続を暗黙的に使用することも、別の接続を指定することもできます。

3. SQL 文を実行します。

a. emp 表に行を挿入します。

```
#sql {insert into emp (ename, empno, sal) values ('SALMAN', 32, 20000)};
```

b. イテレータをインスタンス化して移入します。

```
MyIter iter;
#sql iter={select ename, empno, sal from emp};
```

4. イテレータに移入されたデータにアクセスします。

```
while (iter.next()){
   System.out.println(iter.ename()+" "+iter.empno()+" "+iter.sal());
}
```

next() メソッドは、すべてのイテレータに共通なメソッドです。このメソッドは、JDBC の結果セットの next() メソッドと同じように、true を戻します。行が残っている場合は、次の行に移動します。各行のデータにアクセスするには、列と同じ名前のイテレータ・アクセッサ・メソッドをコールします（名前指定イテレータの場合）。こ

の例では、メソッド ename()、empno() および sal() を使用してデータにアクセスします。

SQLJ の厳密な型指定パラダイム

SQLJ は、結果セットのかわりにイテレータなどの厳密な型指定を使用します。これにより、変換時にデータベースと SQL 命令をチェックすることができます。たとえば、SQLJ では、データベースに接続し、イテレータとその問合せ対象のデータベース表をチェックできます。トランスレータによってイテレータとデータベース表が検証されるので、SQL エラーを変換段階でキャッチできます。つまり、アプリケーションの実行時にユーザーがエラーに遭遇しません。また、後でスキーマを変更した場合、トランスレータを再度実行するだけで、この変更内容がアプリケーションに影響するかどうかを調べることができます。

SQLJ プログラムの変換

Oracle の JDeveloper などの統合開発環境（IDE）は、Java プログラミング用の Windows ベースのビジュアル開発環境であり、SQLJ プログラムの作成時に、変換、コンパイルおよびカスタマイズを行います。IDE を使用しない場合は、SQLJ のフロントエンド・ユーティリティ sqlj を使用できます。次のように実行できます。

```
%sqlj MyExample.sqlj
```

SQLJ トランスレータは、SQL 操作の構文とセマンティクスをチェックします。オンライン・チェックによって、データベースに対する操作をチェックすることもできます。オンライン・チェックを行う場合は、トランスレータ・オプションの設定時にサンプル・データベース・スキーマを指定する必要があります。スキーマに格納するデータは、プログラムが実際に実行するデータでなくてもかまいませんが、表の列名とデータ型は合せる必要があります。user オプションでオンライン・チェックを有効にし、次の例のように、スキーマのユーザー名、パスワードおよび URL を指定します。

```
%sqlj -user=scott/tiger@jdbc:oracle:thin:@oow11:5521:sol2 MyExample.sqlj
```

サーバーでの SQLJ プログラムの実行

多くの SQLJ アプリケーションはクライアントで実行しますが、SQLJ は、SQL を集中的に使用するストアド・プロシージャをプログラミングし、サーバーで実行するときに有利です。

クライアント側 SQLJ プログラムとサーバー側 SQLJ プログラムのコーディング方法はほとんど同じです。SQLJ ランタイム・パッケージはサーバーで自動的に使用可能になります。また、考慮事項も次のようにほとんどありません。

- サーバーで実行するコードに明示的なデータベース接続はなく、1 つの暗黙的な接続があるだけです。通常の接続コードは必要ありません。既存のクライアント側アプリケーションを移植する場合でも、接続コードを削除する必要はありません。接続コードは無視されます。

- サーバー側の内部 JDBC ドライバは、自動コミット機能をサポートしません。トランザクションのコミットとロールバックを手動で行うためには、SQLJ 構文を使用します。
- サーバーでは、デフォルトの出力デバイスはトレース・ファイルです。ユーザー画面ではありません。通常、この点は開発時のみの問題です。サーバー・アプリケーションの配置時は、`System.out` には書き込まないためです。

クライアント側で開発した SQLJ プログラムをサーバーで実行するためには、次の 2 つの選択肢があります。

- SQLJ ソース・コードをクライアント側で変換し、各コンポーネント（Java のクラスとリソース）をサーバーにロードします。この場合は、コンポーネントを最初に `.jar` ファイルにバンドルしておくと簡単です。
- SQLJ ソース・コードをサーバーにロードし、サーバー側のトランスレータで変換します。

どちらの場合も、Oracle の `loadjava` ユーティリティを使用してファイルをサーバーにロードできます。詳細は、『Oracle8*i* 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。

クライアント・アプリケーションの変換とサーバーでの実行

既存の SQLJ クライアント側アプリケーションを変換してサーバーで実行する手順は、次のとおりです。このケースでは、このアプリケーションがクライアント上ですでに変換されていると仮定しています。

1. アプリケーション・コンポーネントの `.jar` ファイルを作成します。
2. `loadjava` ユーティリティで `.jar` ファイルをサーバーにロードします。
3. サーバー上にアプリケーションの SQL ラッパーを作成します。たとえば、前述の `MyExample` アプリケーションをサーバーで実行する場合は、次のようにラッパーを作成します。

```
create or replace procedure SQLJ_MYEXAMPLE as language java
  name 'MyExample.main(java.lang.String[])';
```

これによって、`SQLJ_MYEXAMPLE` を他のストアド・プロシージャと同じように、実行できます。

PL/SQL との対話

すべての Oracle JDBC ドライバは Oracle SQL および PL/SQL とシームレスにやり取りしますが、SQLJ は PL/SQL とインターオペレートすることに注意してください。SQLJ を使用するときに、PL/SQL のストアド・プロシージャを書き直す必要はありません。Oracle SQLJ には PL/SQL ストアド・プロシージャをコールする構文があるので、SQL 操作と同じように、PL/SQL ブロックを SQLJ 実行文に埋め込むことができます。

サーバー・アプリケーションのデバッグ

Oracle8*i* JVM には、JDK の jdb デバッガを使用する開発者にとって便利なデバッグ機能があります。Oracle の JDeveloper では、この Oracle8*i* JVM デバッグ機能が、ユーザーが使いやすいように統合されています。JDeveloper を使用して Java アプリケーションをデバッグする方法の詳細は、JDeveloper のマニュアルを参照してください。他の独立系 IDE ベンダーのデバッガを Oracle8*i* JVM と統合することができます。

Sun Microsystems jdb デバッガは実行プロセスに連結されており、実行プロセスのデバッグに役立ちます。デバッグするアプリケーションは、デバッグ・オプション（-g）でコンパイルされている必要があります。

Oracle8*i* JVM では、Java プログラムをサーバー上でリモート実行します。サーバーは同じ物理マシン上に常駐できますが、通常は別個のマシンに常駐します。Oracle8*i* には、jdb により Oracle8i JVM にロードされた Java アプリケーションをデバッグするための方法が用意されています。

この方法には、Oracle8*i* サーバー上で実行され、実行中の Java アプリケーションと通信を行うデバッグ・エージェントや、クライアント上に常駐して Oracle8*i* サーバーと通信を行うデバッグ・プロキシ、jdb がデバッグ・プロキシと連結するための方法が含まれています。図 3-1 は、デバッグ・エージェント、デバッグ・プロキシ、jdb デバッガの関係を示しています。

図 3-1 デバッグ・プロキシおよびデバッグ・クラスによる jdb デバッガの利用

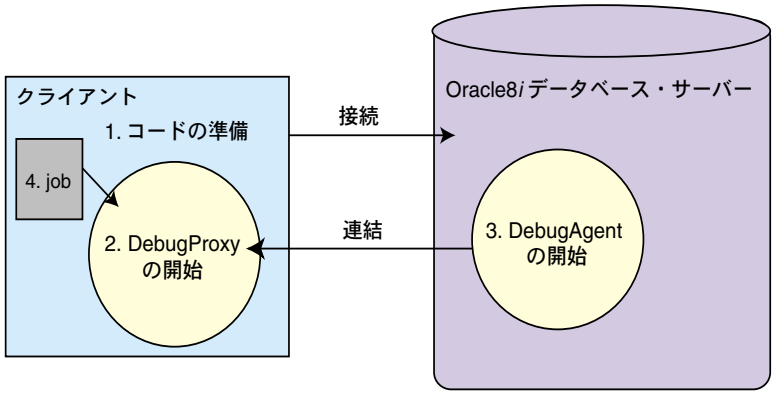


図 3-1 に示すように、Java アプリケーションをリモートでデバッグする手順は次のとおりです。

1. コードをデバッグする準備をします。
2. DebugProxy を起動します。DebugProxy は、DebugAgent がサーバーからデバッグ・プロキシに連結されるまで待機します。

3. デバッグ・プロキシのアドレスを指定して、DebugAgent を起動します。すると、デバッグ・エージェントとデバッグ・プロキシ間の通信が開始されます。
4. jdb デバッガをデバッグ・プロキシと連結します。連結された後は、標準の jdb コマンドを使用できます。

1. コードのデバッグ準備

コードは、-g オプションでコンパイルされている必要があり、ソースはデバッグ・エージェントが検索して使用できるようにしておく必要があります。

アプリケーションをデバッグ・オプション（-g）でコンパイルするには、次の 2 つのうちいずれかの方法で行います。

- 次のように、set_compiler_option プロシージャを使用して、デバッグ・オプションでクラスをコンパイルするように、サーバーに指示します。

```
SQL> call dbms_java.set_compiler_option('myPackage.myCode','debug','true');
```

> **注意：** set_compiler_option プロシージャで、特定のクラス、パッケージ、またはすべてのクラスに多数の異なるコンパイラ・オプションを指定します。この例では、1 つのクラスにオプションを設定しています。このプロシージャの詳細は、『Oracle8*i* Java Tools リファレンス』の「loadjava」の項を参照してください。

その後、次のように、loadjava を使用してソース・コードをロードする必要があります。

```
% loadjava -u SCOTT/TIGER -v -f -r myCode.java
```

サーバーでは、デバッグ・オプションでこのクラスがコンパイルされます。また、サーバーには、デバッグ・エージェントがブレーク・ポイントを示すために必要とするソースとコンパイル済みのバイナリの両方に対するアクセス権があります。

- 次のように、クライアント上でコードを -g オプションを使用してコンパイルし、コンパイル済みのクラスをサーバーにロードして、Java ソース・ファイルを Oracle8*i* JVM に常駐するファイル・システムにコピーします。

```
% javac -g MyCode.java
% loadjava -u SCOTT/TIGER -v -f -r myCode.class
% ftp dbhost
> cd /private/sourcecode
> put myCode.java
```

jdb の起動後、jdb の use コマンドでソース・コードの位置を設定します。これにより、デバッグ・エージェントはソース・コードを検索できます。

```
> use /private/sourcecode
```

注意： 任意のファイルをデータベース・ファイル・システムにコピーするには、正しい FilePermission が必要です。詳細は、「セキュリティ」の章を参照してください。

---

2. デバッグ・プロキシの起動

DebugProxy クラスを使用することにより、リモート Java アプリケーションがローカルにあるかのように見せることができます。デバッグ・プロキシにより、すべての jdb リクエストがサーバー上のデバッグ・エージェントに転送され、連結された jdb デバッガに結果が戻されます。

起動すると、デバッグ・プロキシはデバッグ・エージェントに連結されるまで待機します。aurora_client.jar ファイルが CLASSPATH にある場合は、次のようにデバッグ・プロキシを起動します。

```
debugproxy
```

待機するポートも指定できます。

```
debugproxy -port 2286
```

プロキシが名前、アドレス、および待機しているポートを出力します。

```
Proxy Name: yourmachinename
Proxy Address: aaa.bbb.ccc.ddd
Proxy Port: 2286
```

ただし、DebugProxy を起動する最も簡単な方法は、debugproxy コマンドの末尾に、jdb デバッガを起動するコマンドを追加します。debugproxy コマンドでは、オプションのポートを超えて、開始後に実行するコマンドとして指定された任意のオプションが取り入れられます。この方法を選択する場合、手順 4 を実行する必要はありません。

UNIX の場合は、startjdb と呼ばれる実行可能なシェル・スクリプト内に次のように指定します。

```
#!/bin/sh
xterm -e jdb -password &1 &
```

その後は、次のように debugproxy コマンド内で自動的に jdb デバッガを起動できます。

```
debugproxy -port 1638 startjdb
```

Windows NT 環境ではすべて、startjdb.bat と呼ばれるバッチ・ファイル内で次のように指定します。

```
start jdb -password %1
```

その後は、次のように debugproxy コマンド内で自動的に jdb デバッガを起動できます。

```
debugproxy -port 1638 startjdb.bat
```

3. デバッグ・エージェントの起動、停止および再起動

サーバーに接続（セッションを開始）して、デバッグ・プロキシを起動した後は、プロキシに接続するサーバー上のデバッグ・エージェントを起動できます。DebugAgent が起動すると、手順 4 でデバッガに連結するときに使用するパスワードが DebugProxy によって表示されます。

注意： デバッグ・エージェントを実行するには、ユーザーにデバッグのパーミッション、JAVADEBUGPRIV を付与する必要があります。詳細は、このマニュアルの「デバッグのパーミッション」を参照してください。

プロキシを実行した後、デバッグ・エージェントを起動して、SQL*Plus からプロキシに接続できます。デバッグ・プロキシを実行しているマシンの IP アドレスまたは URL、プロキシが待機しているポート、およびタイムアウト値（秒数）を指定する必要があります。デバッグ・エージェントを起動および停止するには、DBMS_JAVA パッケージに指定されたメソッドを使用します。

```
SQL> call dbms_java.start_debugging('yourmachinename', 2286, 66);
```

サーバー常駐コードを実行して中断する手段、つまり実行後に無限に一時停止モードにする手段はありません。したがって、DebugAgent を起動するときに、DebugAgent の待機タイムアウト期間を指定して、終了を延期する必要があります。開始コールは、タイムアウトになるか、メイン・スレッドが中断、再開後、終了するまで待機します。デバッガが起動する余裕を考慮してタイムアウトを計算してください。ただし、あまりタイムアウトを長く設定すると、デバッガに接続できない場合にセッションが遅延します。

注意： エージェントがすでに動作している場合は、Aurora はエージェントを停止し、新しいエージェントを起動します。

---

stop_debugging メソッドを使用してデバッグ・エージェントを明示的に停止できます。

```
SQL> call dbms_java.stop_debugging();
```

いったん起動したデバッグ・エージェントは、停止処理を行うか、デバッガを切断するか、またはセッションが終了するまで実行し続けます。

restart_debugging メソッドを使用すると、停止したエージェントの全ブレーク・ポイントを設定したまま、再起動できます。コールは、タイムアウトになるまで待機してから、終了します。動作中のエージェントも、数秒間一時停止させて、ブレーク・ポイントを設定することによって再起動できます。

```
SQL> call dbms_java.restart_debugging(66);
```

OracleAgent クラス

DBMS_JAVA デバッグ・エージェントとプロキシ・コールは、oracle.aurora.debug.OracleAgent クラスに常駐する静的メソッドへのエントリ・ポイントで公開されます。次のメソッドで、oracle.aurora.debug.OracleAgent クラスを使用して Java コードのデバッグ・エージェントを直接開始、停止および再開できます。

```
public static void start(String host, int port, long timeout_seconds);
public static void stop();
public static void restart(long timeout_seconds);
```

4. デバッガの接続

jdb を起動し、DebugAgent が DebugProxy への接続時に提供されるパスワードを使用して、起動した jdb をデバッグ・プロキシに連結します。タイムアウトを保持するために、jdb を通じてすべてのスレッドを一時停止し、ブレーク・ポイントを設定してから再開します。

デバッグ・エージェントがデバッグ・プロキシに接続されるたびに、デバッグ・プロキシによりスレッドが起動されて、デバッガからの接続を待機します。スレッドでは、接続元エージェントの番号、名前およびアドレス、待機しているポート、およびポートから生成したパスワードが出力されます。次にポートとパスワードの例を示します。

```
Agent Number: 1
Agent Name: servername
Agent Address: eee.fff.jjj.kkk
Agent Port: 2286
Agent Password: 3i65bn
```

このパスワードを jdb 対応デバッガ（JDK 1.1.6 以降）に渡すには、次のように指定します。

```
jdb -password 3i65bn
```

デバッガで最初に行う操作は、すべてのスレッドの一時停止です。一時停止しないと、ブレーク・ポイントを設定する前に、start_debugging コールがタイムアウトによって終了します。

使用するコードが System.out または System.err に書き込みを行う場合は、jdb に対して dbgtrace フラグを使用することによって、これらのストリームをデバッグ用コンソールにリダイレクトすることもできます。

```
jdb -dbgtrace -password 3i65bn
```

例 3-1 サーバーでの DebugAgent の起動

次の例は、サーバーに常駐するオブジェクトをデバッグする方法を示しています。まず、debugproxy コマンドライン・ツールを使用してプロキシを起動する必要があります。この例では、サーバー tstHost 上でプロキシを起動し、デバッグ・エージェントによって接続されたときに debugproxy に jdb デバッガを起動するように指示しています。

もう 1 つのウィンドウでは、デバッグ・エージェント・ユーザーが正しい権限を持っているかどうかを確認してから、デバッグ・エージェントを起動します。エージェントが起動されると、debugproxy により jdb デバッガが起動され、ブレーク・ポイントを設定できます。エージェントのタイムアウトまでの指定した時間内に、まずするべきことは、すべてのスレッドを一時停止することです。次に、再開する前にブレーク・ポイントをすべて設定します。これにより、実行する準備ができるまではタイムアウトが一時停止されます。

tstHost のウィンドウ 1

```
SQL> call dbms_java.set_compiler_option('', 'debug', 'true');
SQL> exit
% loadjava -u SCOTT/TIGER -v -f -r myCode.java
% 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:myCode
> stop in myCode:updateAccount
> resume
> ...
```

tstHost のウィンドウ 2

```
SQL> grant JavaDebugPriv to SCOTT
SQL> call dbms_java.start_debugging('tstHost',2286,30);
```

サーバー上での実行を見分ける方法

サーバーでの実行方法とクライアントでの実行方法が異なる Java コードを作成する場合があります。通常はこのような Java コードの作成をお薦めしません。実際、JDBC と SQLJ では、サーバー側とクライアント側で使用するドライバが異なる場合でも、この問題を回避する移植可能なコードの作成は手間がかかります。

このコードがサーバー側で実行しているかどうかを判断するには、次のように System.getProperty メソッドを使用します。

```
System.getProperty ("oracle.jserver.version")
```

getProperty メソッドは次のものを戻します。

- サーバーで実行している場合、Oracle8*i* データベースのバージョンを表す文字列（「8.1.5」または「8.1.6」）。
- クライアント上で実行している場合は NULL。

サーバー上の出力のリダイレクト

`System.out` と `System.err` の出力先は、カレント・トレース・ファイルです。出力を SQL*Plus テキスト・バッファにリダイレクトするには、次のように指定します。

```
SQL> SET SERVEROUTPUT ON
SQL> CALL dbms_java.set_output(2000);
```

最小（デフォルト）バッファ・サイズは 2,000 バイトです。最大サイズは 1,000,000 バイトです。次の例では、バッファ・サイズを 5,000 バイトに増やしています。

```
SQL> SET SERVEROUTPUT ON SIZE 5000
SQL> CALL dbms_java.set_output(5000);
```

出力は、コール終了時に行われます。

SQL*Plus の詳細は、『Oracle8*i* SQL*Plus ユーザーズ・ガイドおよびリファレンス』を参照してください。

4

Java のインストールと構成

この章では、Oracle8*i* データベースに Oracle8*i* JVM をインストールおよび構成するために必要な事柄を詳しく説明します。Java メモリーを構成する方法は、第 6 章「Oracle8i Java アプリケーションの パフォーマンス」の「Java のメモリー使用量」の項を参照してください。

- Java 対応データベースの初期化
- Oracle8i JVM の構成
- Java クライアントの使用可能化

Java 対応データベースの初期化

Oracle8*i* と一緒に Oracle8*i* JVM をインストールした場合、そのデータベースは Java 対応になります。つまり、Java ストアド・プロシージャ、JDBC、SQLJ および CORBA/EJB オブジェクトをすぐに実行できます。独自のスクリプトで Oracle インスタンスを作成する場合は、明示的に Oracle8*i* JVM を初期化する必要があります。

Oracle8*i* JVM をインストールするには次の 3 つのうちいずれかの方法を使用します。

- Oracle8*i* 標準または最小セットアップ － 標準または最小 Oracle8*i* インストールを選択すると、Oracle8*i* JVM が自動的にインストールされます。
- Oracle8*i* カスタム・インストール －「カスタム」Oracle8*i* インストールから Oracle8*i* JVM オプションを選択します。
- 手動インストール － `initjvm.sql` スクリプトを起動して Oracle8*i* JVM をインストールします。

手動インストール

Oracle8*i* インストール・オプションで Oracle8*i* JVM をインストールしていない場合には、`ORACLE_HOME/javavm/install` にある `initjvm.sql` スクリプトを使用し、既存のデータベースに Oracle8*i* JVM を追加できます。

`initjvm.sql` スクリプトは、Java のサポートに必要な Java クラスの初期セットをロードし、Java および CORBA ネームスペースをサポートする表を初期化し、コール仕様によって最上位のエントリ・ポイントを公開します。`initjvm.sql` スクリプトは、サポート Java クラスをデータベースにロードします。この Java クラスには次のものが含まれます。

- 標準の Java ランタイム
- バイトコード・ベリファイアおよびオプティマイザ
- Java コンパイラと SQLJ コンパイラ
- JDBC ランタイム
- CORBA ORB および EJB ランタイム
- 4-3 ページの「パッケージ DBMS_JAVA」で説明する `DBMS_JAVA` など、その他のサポート・クラス。

`initjvm.sql` スクリプトでは次のようなアクションが実行されます。

1. SYS スキーマにクラスをロードします。
2. ロードされたクラスに全ユーザーがアクセスするためのパブリック・シノニムを作成します。

3. 一部のクラスを変更し、定義者権限で実行して CORBA のコールアウトをサポートします。
4. データベースの起動トリガーおよびシャットダウン・トリガーを定義します。

注意： `initjvm.sql` スクリプトの実行には 1 時間程度かかります。

要件

Java 対応データベースの初期化には、50MB の `SHARED_POOL_SIZE`、約 20MB の `JAVA_POOL_SIZE`、システム表領域として追加に 30MB、および十分なロールバック・セグメントが必要になります。リソース不足などの原因により、スクリプトが失敗した場合は、必要に応じてリソースを調整し、`initjvm.sql` を再実行できます。データベース初期化ファイルの構成パラメータ、および要件に関する最新情報は、6-7 ページの「Java のメモリー使用量」、および `/javavm/README.txt` ファイルを参照してください。

EJB および CORBA の通信を可能にするための要件もあります。初めのうちは、Oracle8*i* JVM の標準インストールによる初期設定で十分です。詳細は、『Oracle8*i* Enterprise JavaBeans 開発者ガイドおよびリファレンス』、『Oracle8*i* CORBA 開発者ガイド』および『Oracle8*i* Net8 管理者ガイド』を参照してください。

パッケージ DBMS_JAVA

Oracle8*i* JVM を初期化する際に、`initjvm.sql` スクリプトによって PL/SQL パッケージ `DBMS_JAVA` が作成されます。`DBMS_JAVA` のエントリ・ポイントの一部は、開発者が使用するものです。残りのエントリ・ポイントは内部的に使用されます。対応する Java クラス `DbmsJava` は、Java から RDBMS にアクセスするための手段を提供します。

`DBMS_JAVA` パッケージにより次のエントリ・ポイントが提供されます。

FUNCTION longname (shortname VARCHAR2) RETURN VARCHAR2

Java スキーマ・オブジェクトから完全名を戻します。Java のクラスとメソッドの名前は SQL の識別子の最大長を超えることがあるので、Aurora では内部的に SQL アクセス用の短縮名を使用します。この関数は、切り捨てられた名前の元の Java 名を戻すのみです。次の例は、この関数が無効なクラスの完全なクラス名指定を出力するものです。

```
select dbms_java.longname (object_name) from user_objects
   where object_type = 'JAVA CLASS' and status = 'INVALID';
```

FUNCTION shortname (longname VARCHAR2) RETURN VARCHAR2

`DBMS_JAVA` パッケージの `shortname()` ルーチンを使用してデータベースに完全名を指定できます。このルーチンは完全名を入力として取り、対応する短縮名を戻します。`USER_OBJECTS` ビューの問合せによりロードされたクラスの妥当性の検証に便利です。

これらのファンクションの例は、2-30 ページ の「クラスの短縮名」と『Oracle8*i* Java ストアド・プロシージャ開発者ガイド』を参照してください。

```
FUNCTION get_compiler_option(what VARCHAR2, optionName VARCHAR2)
PROCEDURE set_compiler_option(what VARCHAR2, optionName VARCHAR2,
                              value VARCHAR2)
PROCEDURE reset_compiler_option(what VARCHAR2, optionName VARCHAR2)
```

この 3 つのエントリ・ポイントは、Oracle8*i* に同梱されている Oracle8*i* JVM の Java と SQLJ コンパイラのオプションを制御します。これらのオプションの例は、2-14 ページの「Java クラスのコンパイル」を参照してください。また、『Oracle8*i* Java ストアド・プロシージャ開発者ガイド』および『Oracle8*i* SQLJ 開発者ガイドおよびリファレンス』にもオプションとこれらのエントリ・ポイントの説明があります。

```
PROCEDURE set_output (buffersize NUMBER)
```

このプロシージャは Java ストアド・プロシージャとトリガーの出力を DBMS_OUTPUT パッケージにリダイレクトします。例については、3-23 ページの「サーバー上の出力のリダイレクト」を参照してください。

```
PROCEDURE loadjava(options varchar2)
PROCEDURE loadjava(options varchar2, resolver varchar2)
PROCEDURE dropjava(options varchar2)
```

これらのプロシージャを使用すると、コマンドライン・ツールの loadjava または dropjava ではなく、コールを使用して、データベースにクラスをロードしたり、削除できます。Java アプリケーションで実行するには、次のようにします。

```
call dbms_java.loadjava('... options...');
call dbms_java.dropjava('... options...');
```

このオプションは、loadjava および dropjava コマンドライン・ツールに指定するオプションと同じです。各オプションは空白で区切ります。オプションはカンマで区切らないでください。唯一の例外は、空白を含む loadjava -resolver オプションです。-resolver の場合、他のオプションを先に指定してから、これらのオプションをカンマで区切り、次に -resolver をその定義とともに指定してください。-thin、-oci8、-user、-password オプションは、loadjava コマンドライン・ツールのデータベース接続に関連するため、指定しないでください。出力は stderr に送信されます。

loadjava と使用可能なオプションの詳細は、『Oracle8*i* Java Tools リファレンス』を参照してください。

```
PROCEDURE grant_permission( grantee varchar2,
                 permission_type varchar2,
                 permission_name varchar2,
                 permission_action varchar2 )

PROCEDURE restrict_permission( grantee varchar2,
                 permission_type varchar2,
```

```
                  permission_name varchar2,
                  permission_action varchar2)

PROCEDURE grant_policy_permission( grantee varchar2,
                  permission_schema varchar2,
                  permission_type varchar2,
                  permission_name varchar2)

PROCEDURE revoke_permission(permission_schema varchar2,
                  permission_type varchar2,
                  permission_name varchar2,
                  permission_action varchar2)

PROCEDURE disable_permission(key number)

PROCEDURE enable_permission(key number)

PROCEDURE delete_permission(key number)
```

これらのエントリ・ポイントは JVM のパーミッションを制御します。これらのオプションの説明と例は、5-6 ページの「パーミッションの設定」を参照してください。

```
PROCEDURE start_debugging(host varchar2, port number,
                              timeout number)

PROCEDURE stop_debugging

PROCEDURE restart_debugging(timeout number)
```

これらのエントリ・ポイントは、デバッグの際にデバッグ・エージェントを起動および停止します。これらのオプションの説明と例は、3-17 ページの「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。

Oracle8*i* JVM の構成

Oracle8*i* の標準インストールの一部として Oracle8*i* JVM をインストールすると、Oracle8*i* Database Configuration Assistant および Net8 Assistant において Oracle8*i* JVM の構成をする必要が出てきます。ただし、`initjvm.sql` を使用してインストールする場合は、特定の構成ツールを起動するか、初期化ファイルを手動で編集して構成を行う必要があります。

Oracle8*i* での Java クラスの主な構成には、Java メモリー要件、データベース・プロセスのタイプ、およびサーバーとの基礎となる接続プロトコルの構成が含まれます。

- Java メモリー要件 － 20MB の `JAVA_POOL_SIZE`、50MB の `SHARED_POOL_SIZE` が必要です。Oracle インストーラを通じて Oracle8*i* JVM をインストールすれば、これらのパラメータが適切に構成されます。SQL スクリプトの `initjvm.sql` を使用してインストー

ルする場合は、6-7 ページの「Java のメモリー使用量」でデータベース初期化ファイルでこれらのパラメータを構成する方法を参照してください。

- データベース・プロセス – データベース・サーバーに専用サーバー・プロセスまたは MTS プロセスのどちらを使用するかを決める必要があります。
- 接続プロトコル – データベース内のプレゼンテーション層は、データベースへのアクセスの際にクライアントが使用する接続のタイプを定義します。ほとんどのネットワーク・プロトコルでは、プレゼンテーション層により、データがアプリケーション層およびセッション層に対応する形式で表現されていることを確認します。データベースでは、プレゼンテーションは、着信ネットワーク要求をアクセプトし、データベース・カーネル層または Aurora JVM のルーチンを起動して、要求を処理するサービス・プロトコルです。現時点の 2 つの基本プレゼンテーション・タイプは Net8 と GIOP です。GIOP プレゼンテーションは、EJB および CORBA アプリケーションの IIOP 接続に使用します。ほとんどのデータベース接続では、Net8 接続タイプがデフォルトです。

プレゼンテーション・プロトコル	説明
GIOP	IIOP、または、CORBA または EJB アプリケーションの SSL 要求を介した IIOP をアクセプトします。構成情報の詳細は、『Oracle8*i* Enterprise JavaBeans 開発者ガイドおよびリファレンレス』または『Oracle8*i* CORBA 開発者ガイド』を参照してください。
Net8	Oracle Tools（SQL*Plus など）および Forms、Pro*C または OCI を使用したユーザー作成のアプリケーションからの、データベース SQL サービスに対する受信 Net8 要求を処理します。構成情報は、『Oracle8*i* Net8 管理者ガイド』の第 9 章「マルチスレッド・サーバーの構成」を参照してください。

次のように、Java アプリケーションのタイプに応じて、データベースのタイプおよび接続構成には異なる構成が必要になります。

Java API	データベースのタイプ	接続構成
Java ストアド・プロシージャ	Java ストアド・プロシージャは、専用サーバー・モードでもマルチスレッド・サーバー（MTS）モードでも実行できます。主に Java ストアド・プロシージャを作成する場合は、専用サーバー構成で実行できます。	Java ストアド・プロシージャのトリガーとなる Java クライアントまたは PL/SQL クライアントは Net8 接続を通じて接続します。Net8 接続を構成する方法は、『Oracle8*i* Net8 管理者ガイド』を参照してください。
Enterprise JavaBeans（EJB）または CORBA	EJB および CORBA アプリケーションは MTS 構成でのみ実行できます。MTS の構成方法は、『Oracle8*i* Net8 管理者ガイド』を参照してください。	EJB および CORBA クライアントは、CORBA Internet Inter-Orb Protocol（IIOP）を使用します。構成情報は、『Oracle8*i* Enterprise JavaBeans 開発者ガイドおよびリファレンス』または『Oracle8*i* CORBA 開発者ガイド』を参照してください。
Java ストアド・プロシージャおよび EJB または CORBA アプリケーション	EJB および CORBA アプリケーションと Java ストアド・プロシージャを 1 つのアプリケーションに結合する場合、両方のタイプのアプリケーションを次のように構成できます。 ■ MTS 構成で EJB および CORBA アプリケーションをサポートするようにデータベースを構成します。 ■ 専用サーバー構成でストアド・プロシージャをサポートするようにデータベースを構成します。	Net8 接続と IIOP 接続の両方を構成する必要があります。

Java ストアド・プロシージャの構成

データベースを構成して Java ストアド・プロシージャを実行するには、専用サーバー・モードまたは MTS モードのどちらでデータベースを実行するかを決める必要があります。

- 専用サーバー・モード －『Oracle8*i* Java ストアド・プロシージャ開発者ガイド』の説明に従ってデータベースおよびクライアントを構成する必要があります。
- MTS モード －『Oracle8*i* Net8 管理者ガイド』の第 9 章の説明に従って、MTS_DISPATCHERS パラメータを使用してサーバーを MTS モードに構成します。

サーバーで Java ストアド・プロシージャを実行する Java、SQL または PL/SQL クライアントは、Net8 接続を通じてデータベースに接続します。Net8 接続の構成方法の詳細は、『Oracle8*i* Net8 管理者ガイド』を参照してください。

Enterprise JavaBeans と CORBA の構成

クライアントはデータベースの EJB および CORBA アプリケーションに Internet Inter-Orb Protocol（IIOP）接続を通じてアクセスします。IIOP は、TCP/IP 上の GIOP の実装です。IIOP 接続をサポートするには、General Inter-Orb Protocol（GIOP）プレゼンテーションを使用してデータベースを MTS モードに構成する必要があります。

Oracle8*i* は、SSL に必要な証明や秘密鍵などの認証データの使用と、GIOP プロトコル（通常の GIOP およびセッション・ベースの GIOP）の併用もサポートしています。

MTS の構成方法の詳細は、『Oracle8*i* Net8 管理者ガイド』を参照してください。GIOP プレゼンテーションの構成方法の詳細は、『Oralce8*i* Enterprise JavaBeans 開発者ガイドおよびリファレンス』または『Oracle8*i* CORBA 開発者ガイド』を参照してください。

Java クライアントの使用可能化

クライアントとサーバーの間で Java を実行するには、クライアント・システムで次の手順を実行する必要があります。

1. クライアントへの JDK のインストール
2. CLASSPATH の設定
3. ポート /SID の検証
4. サンプルを使用したインストールのテスト

1. クライアントへの JDK のインストール

クライアント・システムは、`loadjava` や `deployejb` などの Oracle8*i* JVM ツールを実行するシステムとして定義されます。クライアントとサーバーに同じシステムを使用できます。

クライアント・システムには JDK 1.1.6 以上が必要です。Solaris 2.6 に含まれる JDK 1.1.3 は、提供されたサンプルでは動作しません。PATH に JDK 1.1.6 以上が含まれ、JDK 1.1.3 が含まれていないことを確認してください。JDK のバージョンを調べるには、次の手順を実行してください。

```
$ which java
/usr/local/packages/jdk1.1.6/bin/java
$ which javac
/usr/local/packages/jdk1.1.6/bin/javac
$ java -version
java version "1.1.6"
```

JDK 1.1.6 がこれらのコマンドに表示されない場合は、PATH の先頭に JDK 1.1.6 のインストール先を入力するか、1.1.3 のインストレーションを削除してください。さらに、CLASSPATH を調べて適切な JDK バージョンが参照されていることを確認してください。

注意： Oracle8*i* Java ベースのクライアント・ツールはすべて Java 2 環境で動作します。

2. CLASSPATH の設定

クライアントが分散アプリケーション（CORBA、EJB または RMI）に関連する Java クライアントの場合、クライアント・コードをコンパイルする前に次の手順を実行する必要があります。

- JAR ファイルまたは ZIP ファイルのサポートを含めるように CLASSPATH を設定します。
- コンパイル行のオプションに JAR ファイルまたは ZIP ファイルのサポートを含めます。

分散アプリケーションの複数のノードに渡って Java クライアントが動作するためには、適切なサーバー・スタブでコンパイルする必要があります。クライアントの CLASSPATH に含める必要のある JAR ファイルまたは ZIP ファイルは、次のとおりです。

注意： NT ユーザーの場合、環境変数は %ORACLE_HOME% と %JAVA_HOME% になります。

基本として組み込む JAR ファイル

```
$ORACLE_HOME/lib/aurora_client.jar
$ORACLE_HOME/lib/mts.jar
$ORACLE_HOME/lib/vbjorb.jar
$ORACLE_HOME/lib/vbjapp.jar
```

実行するには、CLASSPATH に `$ORACLE_HOME/lib/aurora.zip` を組み込みます。

JDK 1.1 クライアントに必要な JAR ファイル

JDK 1.1 クライアントで Java 2 サーバーと通信するには、次の JVM JAR ファイルを組み込む必要があります。

```
$JAVA_HOME/lib/classes.zip
```

JDBC と対話するには、次の ZIP ファイルを組み込みます。

```
$ORACLE_HOME/jdbc/lib/classes111.zip
```

SSL を使用するクライアントの場合は、次の JAR ファイルを組み込みます。

```
$ORACLE_HOME/jlib/jssl-1_1.jar
$ORACLE_HOME/jlib/javax-ssl-1_1.jar
```

Java 2 クライアントに必要な JAR ファイル

Java 2 クライアントで Java 2 サーバーと通信するには、次のいずれかの JVM JAR ファイルが CLASSPATH に含まれていることを確認する必要があります。

- JDK 1.2 の場合は、`dt.jar` を組み込みます。
- JRE 1.2 の場合は、`rt.jar` を組み込みます。

JDBC と対話するには、次の ZIP ファイルを組み込みます。

```
$ORACLE_HOME/jdbc/lib/classes12.zip
```

SSL を使用するクライアントの場合は、次の JAR ファイルを組み込みます。

```
$ORACLE_HOME/jlib/jssl-1_2.jar
$ORACLE_HOME/jlib/javax-ssl-1_2.jar
```

SQLJ を使用するクライアントに組み込む JAR ファイル

```
$ORACLE_HOME/sqlj/lib/translator.zip
```

このファイルの他に、次のように適切な `runtime`*X*`.zip` ファイルを追加します。

- JDBC 8.1.7 を使用する Java 2 クライアント－
  `$ORACLE_HOME/sqlj/lib/runtime12.zip`
- JDBC 8.1.7 を使用する Java 2 Enterprise Edition －
  `$ORACLE_HOME/sqlj/lib/runtime12ee.zip`
- JDBC 8.1.7 を使用する JDK 1.1 クライアント －
  `$$ORACLE_HOME/sqlj/lib/runtime11.zip`
- JDBC 8.1.6 またはそれ以前のバージョンを使用する JDK クライアント－
  `$ORACLE_HOME/sqlj/lib/runtime.zip`

JSP を使用するクライアントに組み込む JAR ファイル

```
$ORACLE_HOME/jsp/lib/ojsp.jar
$ORACLE_HOME/lib/xmlparserv2.jar
$ORACLE_HOME/lib/servlet.jar
```

JSP ページで OracleJML タグまたはデータベース・ユーティリティ JavaBeans を使用している場合は、次のファイルを追加します。

```
$ORACLE_HOME/jsp/lib/ojsputil.jar
$ORACLE_HOME/jsp/lib/oraclexmlsql.jar
```

詳細は、該当する機能のマニュアルを参照してください。機能デモ・サンプル内の README または Make ファイルの両方で使用されるこれらの JAR ファイルと ZIP ファイルの例を参照することもできます。

3. ポート /SID の検証

インストール時に、デフォルトのリスナー・ポート番号またはデータベース SID（ポート 1521、2481、SID `orcl` など）を構成しないと、サンプルは正常に動作しません。すべてのサンプルは、Oracle8*i* の標準インストールで提供されるデフォルトのポート番号およびデータベース SID を前提としています。値が異なる場合は、次のように新しいポート番号を指定してください。

ホスト・タイプ	指示
UNIX	$ make SERVICE=sess_iiop://localhost:myportnum:mysid
Windows NT	コントロール・パネルで、システム環境変数 ORACLE_SERVICE を sess_iiop://localhost:myportnum:mysid に設定します。

4. サンプルを使用したインストールのテスト

`$ORACLE_HOME/javavm/demo/demo.tar`（または Windows NT 用は `demo.zip`）に一連のサンプルが入っています。これらのサンプルは Oracle8*i* の標準インストール・オプションを使用してインストールされたデータベース用にコンパイルおよび実行します。インストールをテストするには、このサンプルを実行してください。

```
$ORACLE_HOME/javavm/demo/examples/jsp/helloworld
$ORACLE_HOME/javavm/demo/examples/corba/basic/helloworld
$ORACLE_HOME/javavm/demo/examples/ejb/basic/helloworld
```

環境に不備があると、サンプルがコンパイルまたは実行されません。同様に、サンプルがコンパイルおよび実行されても、コードがコンパイルまたは実行されない場合は、作成環境またはコードに問題があります。

注意： インストールを検証するときには、付属の Make ファイル（または NT の場合はバッチ・ファイル）を使用してこれらのサンプルを実行することが重要です。

Visual Cafe、JDeveloper、VisualAge などのより複雑な作成環境を使用する前に、サンプルが動作することを確認してください。

5

Oracle8*i* Java アプリケーションのセキュリティ

セキュリティとは、接続のためのネットワーク・セキュリティ、オペレーティング・システム・リソースまたは JVM やユーザー定義クラスのアクセス制御および実行制御、外部ソースからインポートされた JAR ファイルのバイトコード妥当性検査などを含む膨大な分野です。Java アプリケーションを作成する際には、どのようなタイプのセキュリティが適切かを考える必要があります。Oracle8*i* の Java アプリケーションに使用できる様々なセキュリティ・サポートを説明します。

- ネットワーク接続のセキュリティ
- データベースの内容と JVM セキュリティ

ネットワーク接続のセキュリティ

ネットワーク・セキュリティには、認証とデータの機密保護という 2 つの主要な局面があります。認証とデータの機密保護のタイプは、データベースへの接続方法が、Net8、JDBC または分散オブジェクト（EJB または CORBA）のいずれであるかによって決まります。

接続のセキュリティ	説明
Net8	ユーザーがデータベースに接続するには、認証と認可の両方が必要になる場合があります。Net8 データベース接続セキュリティでは、次の 1 つもしくは複数の条件を満たす必要があります。 ■ クライアント認証にユーザー名とパスワードを使用します。データベースへの着信接続要求ではそれぞれ、Net8 に構成された適切なユーザー名 / パスワードを入力する必要があります。詳細は、『Oracle8*i* Net8 管理者ガイド』を参照してください。 ■ 暗号化、Kerberos または secureId に Advanced Networking Option を使用する。『Oracle8*i* Advanced Security 管理者ガイド』を参照してください。 ■ 証明認証に SSL を使用する。『Oracle8*i* Advanced Security 管理者ガイド』を参照してください。
JDBC	JDBC 接続セキュリティでも、Net8 データベース接続に求められるのと同じ制約が必要です。Net8 データベース接続の項に記載された参考文献と併せて、『Oracle8*i* JDBC 開発者ガイドおよびリファレンス』も参照してください。
分散オブジェクト	EJB や CORBA などの分散アプリケーションには、暗号化と認証が必要になります。詳細は、『Oracle8*i* Enterprise JavaBeans 開発者ガイドおよびリファレンス』または『Oracle8*i* CORBA 開発者ガイド』を参照してください。

データベースの内容と JVM セキュリティ

データベースに接続しても、データベースに格納されたリソースにアクセスするには、適切な Java 2 のパーミッションとデータベース権限が必要になります。ここでのリソースとは次のとおりです。

- 表や PL/SQL パッケージなどのデータベース・リソース
- ファイルやソケットなどのオペレーティング・システム・リソース
- Aurora JVM クラス
- ユーザーによってロードされたクラス

これらのリソースは次の 2 つの方式で保護されます。

リソース・セキュリティ	説明
データベース・リソース・セキュリティ	データベース・リソース・セキュリティの認証には、リソースにデータベース権限（Java 2 セキュリティのパーミッションとは異なります）を付与する必要があります。たとえば、データベース・リソースには表、クラスまたは PL/SQL パッケージが含まれます。詳細は、『Oracle8*i* アプリケーション開発者ガイド 基礎編』を参照してください。 すべてのユーザー定義クラスは、他のスキーマのユーザーから保護されます。`loadjava` コマンドのオプションを通じて、他のユーザー / スキーマに対して実行のパーミッションを付与できます。クラスをロードするときの実行権限の設定の詳細は、2-22 ページの「クラスのロード」で説明されている「`-grant` オプション」、loadjava の詳細は、『Oracle8*i* Java Tools リファレンス』を参照してください。
JVM セキュリティ	Oracle8*i* JVM では、オペレーティング・システム・リソースを保護するためにパーミッション・オブジェクトを使用する Java 2 セキュリティが使用されます。Java 2 セキュリティは、起動時に自動的にインストールされ、JAVA_ADMIN 以外のすべてのユーザーから、すべてのオペレーティング・システム・リソースおよび JVM クラスが保護されます。JAVA_ADMIN は他のユーザーにこれらのクラスへのアクセスのパーミッションを付与できます。 Java 2 のパーミッションおよびポリシーを管理および変更する方法は、5-3 ページの「Java 2 セキュリティ」を参照してください。

Java 2 セキュリティ

オペレーティング・システム・リソースにアクセスするには、各ユーザーまたはスキーマに適切なパーミッションを割り当てる必要があります。たとえば、このリソースには、ソケット、ファイルおよびシステム・プロパティが含まれます。

Java 2 セキュリティは、Java アプリケーションに柔軟で構成可能なセキュリティを提供するために作成されたものです。Java 2 セキュリティでは、ロードされた各オブジェクトのスキーマまたはロールに対して個別にパーミッションを定義できます。8.1.5 では、次のような 2 つの保護ロールが選択できました。

- `JAVAUSERPRIV` - プロパティの検査を含むごくわずかのパーミッション
- `JAVASYSPRIV` - JVM の保護パッケージの更新を含む主なパーミッション

注意： 8.1.6 では下位互換性を持たせるために前述の 2 つのロールが残されていますが、Oracle ではこの 2 つのロールを使用するのではなく、それぞれのパーミッションを明示的に指定することをお薦めします。

Oracle8*i* JVM セキュリティは Java 2 セキュリティに基づいているため、クラスのパーミッションはクラス単位で割り当てます。パーミッションには、次の 2 つの文字列属性があります。

- ターゲット（名前）属性
- アクション属性

これらのパーミッションは、データベース管理ツールを使用して割り当てます。各パーミッションは、パーミッション・オブジェクトにカプセル化され、パーミッション表に格納されます。すべてのパーミッションを管理する方法は、この章のほとんどの部分で主題となっています。

Java セキュリティはデータベースを対象に作成されたものではありません。Java 2 セキュリティ・モデルをデータベースの中で適用すると、おのずからその違いが表面化してきます。たとえば、Java 2 セキュリティではアプレットは信頼性が低いものとして定義され、CLASSPATH 内のクラスはすべて信頼性が高いと定義されます。Oracle8*i* では、すべてのクラスはデータベースにロードされるため、Java2 セキュリティにおける信頼性が高いクラスとしては扱われません。

次の表は、Sun Microsystems の Java 2 セキュリティと Oracle8*i* の実装の違いを要約したものです。この表は、読者が Sun Microsystems の Java 2 セキュリティ・モデルをすでに理解していることを前提としています。詳細は、次の文献を参照してください。

- Li Gong 著『Inside Java 2 Platform Security』
- Scott Oaks 著『Java Security』

Java 2 セキュリティ標準	**Oracle8*i* の実装**
CLASSPATH にロードされた Java クラスは信頼性が高い。	Java クラスはすべてデータベース内にロードされます。付与されたパーミッションに基づいてクラスごとに信頼性が決められます。
`java` コマンドラインの `-usepolicy` フラグを通じてポリシーを指定できます。	ポリシーは `PolicyTable` に指定する必要があります。

Java 2 セキュリティ標準	Oracle8*i* の実装
自分で SecurityManager を作成するか、Launcher を使用できます。	自分で SecurityManager を作成する場合、Aurora SecurityManager のみを使用するか、Aurora の SecurityManager を拡張することをお薦めします。動作を変更するには、SecurityManager を定義するのではなく、oracle.aurora.rdbms.SecurityManagerImpl を拡張して特定のメソッドをオーバーライドしてください。
SecurityManager は自動的に初期化されません。SecurityManager は自分で初期化する必要があります。	Aurora は常に起動時に SecurityManager を初期化します。
パーミッションは、アプリケーションまたはアプレットのロード元（URL）またはキーコード（符号付きコード）をもとに決定されます。	パーミッションは、クラスのロード先スキーマによって決まります。Oracle8*i* JVM は符号付きコードをサポートしていません。
セキュリティ・ポリシーはファイルに定義されます。	PolicyTable 定義はデータベース表に保持されます。
（適切なパーミッションがあれば）テキスト・エディタまたはツールを通じてセキュリティ・ポリシー・ファイルを更新できます。	PolicyTable は DBMS_JAVA プロシージャを通じて更新します。初期化後に、PolicyTable を更新するパーミッションを持っているのは JAVA_ADMIN のみです。他のユーザーにパーミッションを付与するには、JAVA_ADMIN から PolicyTable を更新する権限を付与される必要があります。
パーミッションは保護ドメインに対して割り当てられます。クラスは保護ドメインに属することができます。	同じスキーマ内のクラスは同じ保護ドメインの中にあります。
コードを識別するには、`CodeSource` クラスを使用します。 ■ URL と証明が等しければ、`equals` メソッドは TRUE を戻します。 ■ 最初の CodeSource が特定の `CodeSource` オブジェクトを含む汎用表記であれば、`implies` メソッドは TRUE を戻します。	スキーマを識別するには、`CodeSource` クラスを使用します。 ■ スキーマが同じであれば、`equals` メソッドは TRUE を戻します。 ■ スキーマが同じであれば、`implies` メソッドは TRUE を戻します。
正（権限付与）のパーミッションのみをサポートします。	正（権限付与）および制限（制限）の両方のパーミッションをサポートします。

パーミッションの設定

Java 2 セキュリティと同様に、Oracle8*i* はセキュリティ・クラスをサポートしています。通常は、ツールを使用するか、セキュリティ・ポリシー・ファイルを編集して、コード・ベースにパーミッションを設定します。Oracle8*i* では、DBMS_JAVA プロシージャを通じてパーミッションを動的に設定します。これらのプロシージャは、ポリシー表を変更します。ポリシー表は、Java 2 セキュリティのパーミッションを排他的に管理するデータベース内の新しい表です。

ポリシー表を表示するための、USER_JAVA_POLICY と DBA_JAVA_POLICY の 2 つのビューが用意されています。いずれのビューにも付与済みのパーミッションと制限パーミッションに関する情報が含まれています。DBA_JAVA_POLICY ビューには、ポリシー表にあるすべての行が表示されますが、USER_JAVA_POLICY 表には現在のユーザーに関連するパーミッションのみが表示されます。各ビューに表示される行は次のとおりです。

表の列	説明
Kind（種類）	GRANT または RESTRICT。このパーミッションが正（GRANT）または制限（RESTRICT）のどちらであるかが表示されます。
Grantee（権限受領者）	パーミッション・オブジェクトが付与されたユーザー、スキーマまたはロールの名前。
Permission_schema（パーミッションのスキーマ）	パーミッション・オブジェクトのロード先スキーマ。
Permission_type（パーミッションのタイプ）	`java.io.FilePermission` など、完全なクラス名を含む文字列によって指定されたパーミッション・クラス・タイプ。
Permission_name（パーミッション名）	パーミッション・オブジェクトのターゲット属性（名前）。パーミッションを定義するときにはこの名前を使用します。PolicyTable パーミッション・タイプのパーミッションのターゲットを定義する場合は、名前は複雑になります。詳細は、5-12 ページの「ポリシー表を更新するための管理のパーミッションの取得」を参照してください。
Permission_action（パーミッションのアクション）	このパーミッションのアクション属性。多くのパーミッションでは、そのパーミッションに適切なアクションがない場合、NULL 値と想定されます。
Status（状態）	ACTIVE または INACTIVE。作成したパーミッションの行は、使用禁止にしたり再使用可能にすることができます。この列には、パーミッションが使用可能（ACTIVE）または使用禁止（INACTIVE）のいずれの状態にあるのかが表示されます。
Key（キー）	この行を識別するための順序番号。パーミッションを使用禁止、使用可能または削除するときにはこの番号を指定します。

パーミッションを設定する方法は 2 つあります。

- 各パーミッションのきめ細かい定義 － 特定のユーザーまたはロールに対して個別にパーミッションを付与します。アクセスするためのパーミッションを付与しないと、そのスキーマでのアクセスは拒否されます。
- ロールに割り当てられる一般的なパーミッション定義 － ユーザーごとにパーミッションを付与するのではなく、ロールを付与すれば、ユーザーに複数のパーミッションをまとめて設定できます。Oracle8*i* では JAVAUSERPRIV または JAVASYSPRIV のロールを付与できます。

注意： 完全なセキュリティを確保するには、きめ細かい定義を設定してください。一般的な定義の方が簡単ですが、必要なレベルのセキュリティを得られないことがあります。

各パーミッションのきめ細かい定義

ポリシー表に個別にパーミッションを設定するには、次の情報を指定する必要があります。

パラメータ	説明
Grantee（権限受領者）	権限を適用するユーザー、スキーマまたはロールの名前。PUBLIC にすると、その行がすべてのユーザーに適用されます。
Permission type（パーミッションのタイプ）	パーミッションの付与対象となるパーミッション・クラス。たとえば、あるファイルへのアクセスを定義する場合には、パーミッション・タイプは `FilePermission` になります。このパラメータには、`java.lang.security.Permission` を拡張するクラスの完全な名前指定を行います。クラスが SYS に入っていない場合は、クラス名の頭に *<schema>* を追加します。たとえば、`mySchema:myPackage.MyPermission` は、ユーザー生成パーミッションの有効な名前です。
Permission name（パーミッション名）	ターゲット属性の意味は、パーミッション・クラスによって定義されます。該当するパーミッション・クラスから関連する名前を調べてください。
Permission action（パーミッションのアクション）	指定できるアクションのタイプは、パーミッション・タイプによって異なります。たとえば、`FilePermission` には読込みまたは書込みアクションを指定できます。
Key（キー）	パーミッションを付与または制限すると戻される番号で、使用可能、使用禁止または削除メソッドに使用します。

Java 2 のパーミッションを付与するか、自分でパーミッションを作成できます。Java 2 のパーミッションの一覧は表 5-1 を参照してください。自分でパーミッションを作成する方法は、5-14 ページの「パーミッションの作成」を参照してください。

表 5-1 パーミッションのタイプ

- `java.util.PropertyPermission`
- `java.io.SerializablePermission`
- `java.io.FilePermission`
- `java.net.NetPermission`
- `java.net.SocketPermission`
- `java.lang.RuntimePermission`
- `java.lang.reflect.ReflectPermission`
- `java.security.SecurityPermission`
- `oracle.aurora.rdbms.security.PolicyTablePermission`
- `oracle.aurora.security.JServerPermission`

次のように、SQL または Java を使用してパーミッションを付与できます。ただし、Java と SQL のそれぞれによって、パーミッション表内の行を識別する行キー識別子が戻されます。DBMS_JAVA の Java バージョンでは、各メソッドが戻りパラメータまたはパラメータ・リストの OUT 変数として行キー識別子を戻します。PL/SQL の DBMS_JAVA パッケージでは、行キーは key OUT パラメータを定義するプロシージャでのみ戻されます。特定のパーミッションを使用可能または使用禁止にするには、このキーを使用します。詳細は、5-17 ページの「パーミッションの使用可能または使用禁止」を参照してください。

権限付与を実行した後、該当のパーミッション用に行がすでに存在している場合は、更新は発生せず、その行のキーが戻されます。行が使用禁止にされていた場合、権限付与によって既存の行を使用可能にできます。

注意： FilePermission を付与する場合、/private/oracle のように、ディレクトリまたはファイルの物理的な名前を指定する必要があります。$ORACLE_HOME などの環境変数、またはシンボリック・リンクを指定することはできません。また、ディレクトリ内のすべてのファイルを示すには、「/private/oracle/*」のように「*」記号を指定します。また、ディレクトリ内のすべてのディレクトリとファイルを示すには、「/private/oracle/-」のように「-」記号を指定します。

DBMS_JAVA パッケージを使用してパーミッションを付与するには、次のようにします。

```
procedure grant_permission( grantee varchar2, permission_type varchar2,
                            permission_name varchar2,
                            permission_action varchar2 )
```

```
procedure grant_permission( grantee varchar2, permission_type varchar2,
                            permission_name varchar2,
                            permission_action varchar2, key OUT number)
```

Java を使用してパーミッションを付与するには、次のようにします。

```
long oracle.aurora.rdbms.security.PolicyTableManager.grant(
                            java.lang.String grantee,
                            java.lang.String permission_type,
                            java.lang.String permission_name,
                            java.lang.String permission_action);

void oracle.aurora.rdbms.security.PolicyTableManager.grant(
                            java.lang.String grantee,
                            java.lang.String permission_type,
                            java.lang.String permission_name,
                            java.lang.String permission_action,
                            long[] key);
```

DBMS_JAVA パッケージを使用してパーミッションを制限するには、次のようにします。

```
procedure restrict_permission( grantee varchar2, permission_type varchar2,
                            permission_name varchar2,
                            permission_action varchar2)

procedure restrict_permission( grantee varchar2, permission_type varchar2,
                            permission_name varchar2,
                            permission_action varchar2, key OUT number)
```

Java を使用してパーミッションを制限するには、次のようにします。

```
long oracle.aurora.rdbms.security.PolicyTableManager.restrict(
                            java.lang.String grantee,
                            java.lang.String permission_type,
                            java.lang.String permission_name,
                            java.lang.String permission_action);

void oracle.aurora.rdbms.security.PolicyTableManager.restrict(
                            java.lang.String grantee,
                            java.lang.String permission_type,
                            java.lang.String permission_name,
                            java.lang.String permission_action,
                            long[] key);
```

例 5-1 パーミッションの付与

ポリシー表を変更するための適切なパーミッションを持っている場合に、指定ファイルへのユーザー・アクセスを許可するためには、DBMS_JAVA パッケージの grant_permission メソッドを使用して PolicyTable を変更します。この例では、ユーザー Larry が PolicyTable 変更のパーミッションを持っています。SQL パッケージでは、Larry がユーザー Dave に対してファイルの読取り / 書込みのパーミッションを付与します。

```
connect larry/larry

REM Grant DAVE permission to read and write the Test1 file.
call dbms_java.grant_permission('DAVE',
                        'java.io.FilePermission', '/test/Test1',
                        'read,write');

REM commit the changes to the PolicyTable
commit;
```

例 5-2 パーミッションの制限

一般的な規則に対する制限または例外を指定するには、restrict メソッドを使用します。一般的な規則では、ほとんどの場合、パーミッションが真となります。ただし、この規則には例外がある場合があります。これらの例外については、制限パーミッションを指定します。

つまり、ディレクトリ全体に対して読込みまたは書込みを禁止する一般規則を定義してある場合は、restrict メソッドを使用してこの規則の一部を制限するように定義できます。たとえば、/tmp ディレクトリに入っている自分の password ファイルを除くすべてのファイルに対するアクセスを許可するには、/tmp ディレクトリ内の全ファイルに対して読取り / 書込みのパーミッションを付与した後、password ファイルに対する読取り / 書込みアクセスを制限します。

制限に対する例外を指定する場合、明示的な権限付与パーミッションを作成して制限パーミッションをオーバーライドできます。その場合、ファイルの所有者が password ファイルを変更するためには、1 人のユーザーにさらに明示的にパーミッションを付与して、制限をオーバーライドできます。Oracle8*i* JVM セキュリティではすべての規則を考慮して、password ファイルにアクセスできるユーザーが誰であるかを判断します。次の図にこれを示します。

/tmp/* への *PUBLIC* パーミッションの付与

```
/tmp % ls -al
.
..
password
test
myCode.java
myCode.class
updSQL.sqlj
Makefile
```

password	PUBLIC への制限パーミッション
test myCode.java myCode.class	所有者にパーミッションを付与して、 前述の制限をオーバーライドする

明示的な規則は、次のとおりです。

> 制限パーミッションに暗黙的な要求が含まれる場合、付与を有効にするには、その制限パーミッションに暗黙的な付与も含まれている必要があります。

次に、この例を実装するコードを紹介します。

1. すべてのユーザー（PUBLIC）に /tmp 内のすべてのファイルの読込みおよび書込みのパーミッションを付与します。
2. すべてのユーザー（PUBLIC）による /tmp/password 内にある 1 つのファイルの読込みおよび書込みのパーミッションを制限します。
3. Larry（所有者）に password ファイルに対する読取りおよび書込みのパーミッションを明示的に付与します。

```
connect larry/larry

REM Grant permission to all users (PUBLIC) to be able to read and write
REM all files in /tmp.
call dbms_java.grant_permission('PUBLIC',
                        'java.io.FilePermission',
                        '/tmp/*',
                        'read,write');

REM Limit permission to all users (PUBLIC) from reading or writing the
REM password file in /tmp.
call dbms_java.restrict_permission('PUBLIC',
                        'java.io.FilePermission',
                        '/tmp/password',
                        'read,write');

REM By providing a more specific rule that overrides the limitation,
REM Larry can read and write /tmp/password.
```

```
call dbms_java.grant_permission('LARRY',
                        'java.io.FilePermission',
                        '/tmp/password',
                        'read,write');
commit;
```

ポリシー表を更新するための管理のパーミッションの取得

すべてのパーミッションはポリシー表内の行です。ポリシー表はデータベース内の表でありリソースなので、変更にはパーミッションが必要です。特に、PolicyTablePermission オブジェクトは、表の変更に必要です。Oracle8*i* JVM を初めて初期化した時点では、ポリシー表を変更できるのは、PolicyTablePermission を付与された JAVA_ADMIN ロールのみです。データベース管理者にはただちに JAVA_ADMIN ロールが割り当てられます。したがって、DBA グループに設定されていれば、JAVA_ADMIN の全パーミッションを自動的に取得できます。

この表にパーミッションを行として追加するには、JAVA_ADMIN により PolicyTablePermission のスキーマの更新権限が付与されている必要があります。このパーミッションでは、スキーマが表に行を追加できるように定義します。各 PolicyTablePermission は、特定のパーミッション・タイプに対応しています。たとえば、あるファイルへのアクセスを制御するパーミッションを追加するためには、FilePermission に対するパーミッションを付与または制限できる PolicyTablePermission が必要です。一度これを行うと、FilePermission に対する管理のパーミッションが得られます。

管理者は、他のパーミッションと同様の方法で PolicyTablePermissions を付与または制限できますが、構文は複雑になります。使用しやすくするため、DBMS_JAVA パッケージの次のいずれかの方法を使用して管理のパーミッションを付与します。

DBMS_JAVA を使用してポリシー表の管理のパーミッションを付与するには、次のようにします。

```
procedure grant_policy_permission( grantee varchar2, permission_schema varchar2,
                            permission_type varchar2,
                            permission_name varchar2)

procedure grant_policy_permission( grantee varchar2, permission_schema varchar2,
                            permission_type varchar2,
                            permission_name varchar2,
                            key OUT number)
```

Java を使用してポリシー表の管理のパーミッションを付与するには、次のようにします。

```
long oracle.aurora.rdbms.security.PolicyTableManager.grantPolicyPermission(
                            java.lang.String grantee,
                            java.lang.String permission_type,
                            java.lang.String permission_name);
```

```
void oracle.aurora.rdbms.security.PolicyTableManager.grantPolicyPermission(
                                java.lang.String grantee,
                                java.lang.String permission_type,
                                java.lang.String permission_name,
                                long[] key);
```

パラメータ	説明
Grantee（権限受領者）	権限を適用するユーザー、スキーマまたはロールの名前。PUBLIC にすると、その行がすべてのユーザーに適用されます。
Permission_schema（パーミッションのスキーマ）	パーミッション・クラスのロード先の *<schema>*。
Permission_type（パーミッションのタイプ）	パーミッションの付与対象となるパーミッション・クラス。たとえば、あるファイルへのアクセスを定義する場合には、パーミッション・タイプは `FilePermission` になります。このパラメータには、`java.lang.security.Permission` を拡張するクラスの完全な名前指定を行います。クラスが SYS に入っていない場合は、クラス名の頭に *<schema>* を追加します。たとえば、`mySchema:myPackage.MyPermission` は、ユーザー生成パーミッションの有効な名前です。
Permission_name（パーミッション名）	ターゲット属性の意味は、パーミッション・クラスによって定義されます。該当するパーミッション・クラスから関連する名前を調べてください。
Row_ number（行番号）	パーミッションを付与または制限すると戻される番号で、使用可能、使用禁止または削除メソッドに使用します。

注意： ポリシー表を見ると、PolicyTablePermission 行の名前には、# で区切られパーミッション・タイプとパーミッション名の両方が含まれています。たとえば、ファイルの読込みユーザー管理権限を付与する場合、この行の名前には `java.io.FilePermission#read` を含めます。パーミッション・クラスとパーミッション名は # で区切ります。

例 5-3 PolicyTable のパーミッションの付与

次の例は、JAVA_ADMIN が（SYS として）Larry に `FilePermission` の PolicyTable を更新するパーミッションを付与するものです。このパーミッションが付与されると、Larry は他のユーザーにファイルの読込み、書込みおよび削除パーミッションを付与できます。

```
REM Connect as SYS, which is assigned JAVA_ADMIN role, to give Larry permission
REM to modify the PolicyTable
connect SYS/SYS as SYSDBA
```

```
REM SYS grants Larry the right to administer permissions for
REM FilePermission
call dbms_java.grant_policy_permission('LARRY', 'SYS',
                                        'java.io.FilePermission', '*');
```

パーミッションの作成

自分でパーミッション・タイプを作成する方法は次のとおりです。

1. ユーザーのパーミッションを作成してロードする。
2. 指定したユーザーに管理のパーミッションおよびアクションのパーミッションを付与する。
3. パーミッションを使用してセキュリティ・チェックを実装する。

1. ユーザーのパーミッションを作成してロードする Java 2 のパーミッション・クラスを拡張して独自のパーミッションを作成できます。ユーザーが作成するパーミッションはすべて、`Permission` を拡張したものである必要があります。次の例は、`MyPermission` を作成するものです。`MyPermission` は `BasicPermission` を拡張し、`BasicPermission` はさらに `Permission` を拡張します。

```
package test.larry;
import java.security.Permission;
import java.security.BasicPermission;

public class MyPermission extends BasicPermission {

  public MyPermission(String name) {
    super(name);
  }

  public boolean implies(Permission p) {
    boolean result = super.implies(p);
    return result;
  }
}
```

2. 指定したユーザーに管理のパーミッションおよびアクションのパーミッションを付与する パーミッションを作成すると、そのパーミッションの所有者になります。所有者は、暗黙的に管理のパーミッションを付与されます。つまり、所有者はこのパーミッションの管理者になることができるので、`grant_policy_permission` を実行できます。管理のパーミッションを付与されたユーザーは、ユーザー定義のパーミッション用のポリシー表を更新できます。

たとえば、LARRY が `MyPermission` というパーミッションを作成したとすると、自分または他のユーザーのために **`grant_policy_permission`** を起動できるのは LARRY のみにな

ります。このメソッドは、MyPermission に対する権限を付与できるユーザーを定義する PolicyTable を更新します。次のコードはその方法を示すものです。

```
REM Since Larry is the user that owns MyPermission, Larry connects to
REW the database to assign permissions for MyPermission.
connect larry/larry

REM As the owner of MyPermission, Larry grants himself the right to
REM administer permissions for test.larry.MyPermission within the JVM
REM security PolicyTable. Only the owner of the user-defined permission
REM can grant administrative rights.
call dbms_java.grant_policy_permission('LARRY', 'LARRY',
                      'test.larry.MyPermission', '*');

REM commit the changes to the PolicyTable
commit;
```

管理権限を付与された後、ユーザー作成のパーミッションに対してアクションのパーミッションを付与できます。たとえば、次の SQL は MyPermission の中のすべてに対して LARRY に実行のパーミッションを付与します。DAVE には「act」で始まるアクションに対してのみ実行のパーミッションが与えられます。

```
REM Since Larry is the user that creates MyPermission, Larry connects to
REW the database to assign permissions for MyPermission.
connect larry/larry

REM Once able to modify the PolicyTable for MyPermission, Larry grants himself
REM full permission for MyPermission. Notice that the Permission is prepended
REM with its owner schema.
call dbms_java.grant_permission( 'LARRY',
                        'LARRY:test.larry.MyPermission', '*', null);

REM Larry grants Dave permission to do any actions that start with 'act.*'.
call dbms_java.grant_permission
        ('DAVE', 'LARRY:test.larry.MyPermission', 'act.*', null);

REM commit the changes to the PolicyTable
commit;
```

3. パーミッションを使用してセキュリティ・チェックを実装する MyPermission に対してパーミッションを作成、ロードおよび割り当てた後、パーミッションのチェックを実行するために SecurityManager へのコールを実装する必要があります。次の例には、sensitive、act、print、hello の 4 つのメソッドがあります。手順 2 の SQL の例でパーミッションが付与されているので、この例のクラスでは次のユーザーがメソッドを実行できます。

- LARRY はすべてのメソッドを実行できます。
- DAVE には act メソッドに対する実行のパーミッションのみが与えられます。

- print メソッドと hello メソッドはどのユーザーでも実行できます。print メソッドはパーミッションをチェックしないため、どのユーザーでも print メソッドを実行できます。hello メソッドは AccessController.doPrivileged を実行します。つまり、このメソッドは LARRY のパーミッションを使用して実行します。これは定義者権限と呼ばれます。

```
package test.larry;
import java.security.AccessController;
import java.security.Permission;
import java.security.PrivilegedAction;

import java.sql.Connection;
import java.sql.SQLException;

/**
 * MyActions is a class with a variety of public methods that
 * have some security risks associated with them. We will rely
 * on the Java security mechanisms to ensure that they are
 * performed only by code that is authorized to do so.
 */

public class Larry {

  private static String secret = "Larry's secret";

  MyPermission sensitivePermission = new MyPermission("sensitive");

  /**
   * This is a security sensitive operation. That is it can
   * compromise our security if it is executed by a "bad guy".
   * Only larry has permission to execute sensitive.
   */
  public void sensitive() {
    checkPermission(sensitivePermission);
    print();
  }

  /**
   * Will print a message from Larry.  We need to be
   * careful about who is allowed to do this
   * because messages from Larry may have extra impact.
   * Both larry and dave have permission to execute act.
   */
public void act(String message) {
    MyPermission p = new MyPermission("act." + message);
    checkPermission(p);
    System.out.println("Larry says: " + message);
```

```
  }

  /**
   * Print our secret key
   * No permission check is made; anyone can execute print.
   */
  private void print() {
    System.out.println(secret);
  }

  /**
   * Print "Hello"
   * This method invokes doPrivileged, which makes the method run
   * under definer's rights. So, this method runs under Larry's
   * rights, so anyone can execute hello.
   * Only Larry can execute hello
   */
  public void hello() {
    AccessController.doPrivileged(new PrivilegedAction() {
      public Object run() { act("hello"); return null; }
});
  }

  /**
   * If a security manager is installed ask it to check permission
   * otherwise use the AccessController directly
   */
  void checkPermission(Permission permission) {
    SecurityManager sm = System.getSecurityManager();
    sm.checkPermission(permission);
  }
}
```

パーミッションの使用可能または使用禁止

パーミッションを定義する行を作成した場合、それを使用禁止にすることによって無効にできます。ただし、後でその行にアクションが必要となった場合に、再び有効にすることも可能です。不要になった行は表から削除できます。行を削除するには、まずその行を使用禁止にする必要があります。行を使用禁止にしないと、削除はできません。

行を使用禁止にするには、`disable_permission` メソッドまたは `revoke` メソッドを使用します。

- `revoke_permission` メソッドには `grant` および `restrict` メソッドと同じパラメータを指定します。このメソッドは、指定されたパラメータに該当するすべての行をポリシー表で検索します。

- disable_permission メソッドはポリシー表の 1 行のみを使用禁止にします。このために、disable_permission メソッドにはポリシー表キーを指定します。このキーはパーミッションを使用可能にしたり、削除する場合にも必要になります。パーミッション・キー番号を取り出すには、次のいずれかを実行します。
  - grant または limit コールで戻されたキーを保存します。そのパーミッションを使用可能または使用禁止にする必要がないと判断した場合は、パーミッション番号を戻さない grant コールおよび limit コールを使用できます。
  - DBA_JAVA_POLICY または USER_JAVA_POLICY を表示して適切なパーミッション・キー番号を調べます。

DBMS_JAVA を使用してパーミッションを使用禁止にするには、次のようにします。

```
procedure revoke_permission(permission_schema varchar2,
                            permission_type varchar2,
                            permission_name varchar2,
                            permission_action varchar2)

procedure disable_permission(key number)
```

Java を使用してパーミッションを使用禁止にするには、次のようにします。

```
void revoke(String schema, String type, String name, String action);

void oracle.aurora.rdbms.security.PolicyTableManager.disable(long number);
```

DBMS_JAVA を使用してパーミッションを使用可能にするには、次のようにします。

```
procedure enable_permission(key number)
```

Java を使用してパーミッションを使用可能にするには、次のようにします。

```
void oracle.aurora.rdbms.security.PolicyTableManager.enable(long number);
```

DBMS_JAVA を使用してパーミッションを削除するには、次のようにします。

```
procedure delete_permission(key number)
```

Java を使用してパーミッションを削除するには、次のようにします。

```
void oracle.aurora.rdbms.security.PolicyTableManager.delete(long number);
```

パーミッションのタイプ

表 5-2 は、インストールされているパーミッションのタイプの一覧です。パーミッションを付与または制限するには、DBMS_JAVA メソッドにパーミッション・タイプを指定する必要があります。アクセス制御に使用できるパーミッション・タイプは次のとおりです。

- 表 5-2 にリストされている Oracle 提供のパーミッション・タイプ。
- `java.security.Permission` を拡張するユーザー作成のパーミッション・タイプ。

表 5-2 パーミッションのタイプ

- `java.util.PropertyPermission`
- `java.io.SerializablePermission`
- `java.io.FilePermission`
- `java.net.NetPermission`
- `java.net.SocketPermission`
- `java.lang.RuntimePermission`
- `java.lang.reflect.ReflectPermission`
- `java.security.SecurityPermission`
- `oracle.aurora.rdbms.security.PolicyTablePermission`
- `oracle.aurora.security.JServerPermission`

Java のすべてのパーミッション・タイプは、Sun Microsystems の Java 2 のマニュアルを参照してください。

注意： SYS には、Oracle に付属のライブラリをロードするパーミッションが付与されています。ただし、データベースに C をロードすることは危険なため、Oracle8*i* JVM では他のユーザーによるライブラリのロードは禁止されます。したがって、`RuntimePermission` の `loadLibrary.*` に対してパーミッションを付与することは許されません。

次に、Oracle 固有のパーミッションである `PolicyTablePermission` と `JServerPermission` を説明します。

oracle.aurora.rdbms.security.PolicyTablePermission このパーミッションでは、ポリシー表を更新できるユーザーを管理します。特定のパーミッション・タイプのポリシー表を更新する権限を付与されたユーザーは、他のユーザーのリソースへのアクセスを制御できます。

Oracle8*i* JVM を初期化した時点では、`PolicyTablePermission` を通じてポリシー表の管理権限を付与できるのは JAVA_ADMIN ロールのみです。JAVA_ADMIN からこの権限を付

与されたユーザーは、自分の付与パーミッションおよび制限パーミッションを使用してポリシー表を更新できます。

ポリシー表の更新権限を付与する場合に使用する DBMS_JAVA メソッドは、5-12 ページの「ポリシー表を更新するための管理のパーミッションの取得」に説明されている `grant_policy_permission` です。表を更新したときに、DBA_JAVA_POLICY ビューまたは USER_JAVA_POLICY ビューを表示してパーミッションを付与されたユーザーを調べることができます。

oracle.aurora.security.Oracle8*i* JVMPermission Aurora JVM リソースに対するアクセスを付与および制限するには、このパーミッションを使用します。`JServerPermission` は `BasicPermission` から拡張します。次の表は、`JServerPermission` がアクセスのパーミッションを付与する名前の一覧です。

パーミッション名	説明
LoadClassInPackage.<*package_name*>	指定されたパッケージにクラスをロードするパーミッションを付与します。
Verifier	バイトコード・ベリファイアをオンまたはオフに切り替えるパーミッションを付与します。
Debug	デバッガをセッションに接続するパーミッションを付与します。
JRIExtensions	MEMSTAT の使用のパーミッションを付与します。
Memory.Call	コール設定時に `oracle.aurora.vm.OracleRuntime` の特定のメソッドをコールする権限を付与します。
Memory.Stack	スタック設定時に `oracle.aurora.vm.OracleRuntime` の特定のメソッドをコールする権限を付与します。
Memory.SGAIntern	SGA 設定時に `oracle.aurora.vm.OracleRuntime` の特定のメソッドをコールする権限を付与します。
Memory.GC	ガベージ・コレクタ設定時に `oracle.aurora.vm.OracleRuntime` の特定のメソッドをコールする権限を付与します。

初期のパーミッションの付与

Oracle8*i* JVM を初めて初期化すると、複数のロールに特定のパーミッションが移入されます。次の表はこれらのロールとその初期のパーミッションを示します。

1. JAVA_ADMIN ロールにはすべてのパーミッションのポリシー表を変更する権限が与えられます。SYS を含むすべてのデータベース管理者には JAVA_ADMIN が付与されます。次のようなパーミッションに対して、ポリシー表を更新する完全な管理権限が付与されます。

- `java.util.PropertyPermission`
- `java.io.SerializablePermission`
- `java.io.FilePermission`
- `java.net.NetPermission`
- `java.net.SocketPermission`
- `java.lang.RuntimePermission`
- `java.lang.reflect.ReflectPermission`
- `java.security.SecurityPermission`
- `oracle.aurora.rdbms.security.PolicyTablePermission`
- `oracle.aurora.security.JServerPermission`

2. JAVA_ADMIN のパーミッションに加え、SYS には次のパーミッションも付与されます。

注意： `RuntimePermission` 付与では、`loadlibrary.<package>` へ特定のパーミッションを不必要に付与しています。これは、`loadLibrary.*` の PUBLIC に指定された制限をオーバーライドするためです。

表 5-3 SYS の初期のパーミッション

パーミッション・タイプ	パーミッション名	付与されたアクション
`oracle.aurora.rdbms.security.PolicyTablePermission`	*	ポリシー表を変更するための管理権限
`oracle.aurora.security.JServerPermission`	*	NULL
`java.net.NetPermission`	*	NULL
`java.security.SecurityPermission`	*	NULL
`java.util.PropertyPermission`	*	書込み
`java.lang.reflect.ReflectPermission`	*	NULL
`java.lang.RuntimePermission`	*	NULL
	`loadLibrary.xaNative`	NULL
	`loadLibrary.corejava`	NULL
	`loadLibrary.corejava_d`	NULL

3. 全ユーザーには次のようなパーミッションが最初に付与されます。
JServerPermission については、表に指定されたリストを除くすべてのクラスを全ユーザーがロードできます。この例外は制限パーミッションです。制限パーミッションの詳細は、例 5-2 を参照してください。

表 5-4 PUBLIC のデフォルトのパーミッション

パーミッション・タイプ	パーミッション名	付与されたアクション
oracle.aurora.rdbms.security.PolicyTablePermission	java.lang.RuntimePermission.loadLibrary.*	NULL
java.util.PropertyPermission	*	読込み
	user.language	書込み
java.lang.RuntimePermission		NULL
	exitVM	NULL
	createSecurityManager	NULL
	modifyThread	NULL
	modifyThreadGroup	NULL
oracle.aurora.security.JServerPermission	loadClassInPackage.java.*、loadClassInPackage.oracle.aurora.*、および loadClassInPackage.jdbc.* を除く loadClassInPackage.*	NULL

表 5-5 JAVAUSERPRIV のパーミッション

パーミッション・タイプ	パーミッション名	アクション
java.net.SocketPermission	*	接続、解決
java.io.FilePermission	<<ALL FILES>>	読込み
java.lang.RuntimePermission	modifyThreadGroup、stopThread、getProtectionDomain、readFileDescriptor、accessClassInPackage.*、および defineClassInPackage.*	NULL

表 5-6 JAVASYSPRIV のパーミッション

パーミッション・タイプ	パーミッション名	アクション
java.io.SerializablePermission	*	該当アクションなし
java.io.FilePermission	<<ALL FILES>>	読込み、書込み、実行、削除
java.net.SocketPermission	*	アクセプト、接続、リスニング、解決
java.lang.RuntimePermission	createClassLoader	NULL
	getClassLoader	NULL
	setContextClassLoader	NULL
	setFactory	NULL
	setIO	NULL
	setFileDescriptor	NULL
	readFileDescriptor	NULL
	writeFileDescriptor	NULL

表 5-7 JAVADEBUGPRIV のパーミッション

パーミッション・タイプ	パーミッション名	アクション
oracle.aurora.security.JServerPermission	Debug	NULL
java.net.SocketPermission	*	接続、解決

ロールに割り当てられる一般的なパーミッション定義

8.1.5 では、JAVASYSPRIV、JAVAUSERPRIV または JAVADEBUGPRIV のロールをスキーマに付与することで JVM セキュリティは制御されていました。現行バージョンでは、これらのロールがパーミッション・グループとして残っています。各ロールの明示的パーミッションは、前項 5-20 ページの「初期のパーミッションの付与」を参照してください。ユーザー独自のパーミッションの組合せを設定および定義できます。パーミッションを定義した後、任意のユーザーに任意のパーミッションの組合せを付与できます。パーミッションを付与されたユーザーには、そのロールの中に存在する他のパーミッションも付与されます。さらに、追加のパーミッションが必要な場合は、指定したユーザーまたはロールに個別にパーミッションを追加できます。ポリシー表に定義されたパーミッションの効果は累積されていきます。ユーザーまたはロールにパーミッションを付与する方法は、5-7 ページの「各パーミッションのきめ細かい定義」を参照してください。

注意： 以前に PropertyPermission の書込みアクションを通じて付与されていたプロパティへの書込み権限は、現行バージョンでは全ユーザーには付与されません。そのかわりに、JAVA_ADMIN からこのパーミッションを付与されるか、JAVASYSPRIV ロールの付与を通じてこのパーミッションを取得する必要があります。

次の例では、Larry と Dave に次のパーミッションを付与します。

- Larry には JAVASYSPRIV パーミッションが付与されます。
- Dave には JAVADEBUGPRIV パーミッションと、システム上の全ファイルへの読取りおよび書込みのパーミッションが与えられます。

```
REM Granting Larry the same permissions as exist within JAVASYSPRIV
grant javasyspriv to larry;

REM Granting Dave the ability to debug
grant javadebugpriv to dave;

commit;

REM I also want Dave to be able to read and write all files on the system
call dbms_java.grant_permission('DAVE', 'SYS:java.io.FilePermission',
                                '<<ALL FILES>>', 'read,write', null);
```

デバッグのパーミッション

デバッグ・ロール JAVADEBUGPRIV は、デバッガを実行するパーミッションを付与するために作成されたものです。このロールに付与されるパーミッションの一覧は表 5-7 を参照してください。デバッグ・エージェントを起動するパーミッションを取得するには、次のように、コール側に JAVADEBUGPRIV またはデバッグ JServerPermission が付与されていることが必要です。

```
REM Granting Dave the ability to debug
grant javadebugpriv to dave;

REM Larry grants himself permission to start the debug agent.
call dbms_java.grant_permission
        ('LARRY', 'oracle.aurora.security.JServerPermission', 'Debug', null);
```

デバッガは、サーバー上のコードとデータに広範囲にアクセスできますが、ここでは、開発環境での使用方法に絞って説明します。このリリースのデバッグ機能の使用方法に関する詳細は、3-17 ページの「サーバー・アプリケーションのデバッグ」の説明を参照してください。

クラスをロードするためのパーミッション

クラスをロードするには、次のようなパーミッションが必要になります。

```
JServerPermission("LoadClassInPackage." + <class_name>)
```

クラス名には、ロードするクラスの完全なクラス名指定を行います。

このパーミッションの中には、システム・パッケージへのロードやシステム・クラスの置換は含まれません。たとえシステム・クラスのロードのパーミッションが付与されていても、Oracle8*i* JVM ではロードが禁止されます。システム・クラスは、Oracle8*i* で CREATE JAVA SYSTEM によりインストールされるクラスです。システム・クラスを置換しようとすると、次のエラーが発生します。

```
ORA-01031「権限が不足しています。」
```

次に、データベースのインストール後に各ユーザーが持つパーミッションの内容と Oracle8*i* JVM の制限を示します。

- SYS はシステム・クラス以外の全クラスをロードできます。
- どのユーザーも、次のパターンで始まるスキーマ以外の、すべてのスキーマにクラスをロードできます。`java.*`、`oracle.aurora.*`、`oracle.jdbc.*`。そのようなクラスを別のスキーマにロードする場合は、`JServerPermission(LoadClassInPackage.<class>)` のパーミッションが必要になります。

次の例は、`oracle.aurora.*` パッケージにクラスをロードするパーミッションを SCOTT に付与する方法を示します。

```
dbms_java.grant_permission('SCOTT', 'SYS:oracle.aurora.tools.*', null);
```

6

Oracle8*i* Java アプリケーションのパフォーマンス

次のいずれかの方法で、Java アプリケーションのパフォーマンスを向上できます。

- ネイティブ・コンパイル・コード
- Java のメモリー使用量
- コール終了時の移行
- メモリー・プロファイリング・ユーティリティ

ネイティブ・コンパイル・コード

Java 言語は、プラットフォームに依存しない安全な開発モデルを想定して設計されました。これらの目標を達成するために、実行上のパフォーマンスがいくらか低下しました。Java のバイトコードをマシン命令に変換するためにパフォーマンスが低下します。特定のクラスをネイティブにコンパイルすると、低下したパフォーマンスをいくらか向上させることができます。たとえば、CPU インテンシブ・クラスでコードをネイティブにコンパイルします。

ネイティブ・コンパイルを使用しないと、サーバーにロードした Java コードは実行時に解析され、そのコードの基礎となるコア・クラス（`java.lang.*`）がネイティブにコンパイルされます。

ネイティブ・コンパイルを使用すると、バイトコードでの解析のスピードと比較して、実行が 2 ～ 10 倍高速化されます。ただし、厳密な速度値は、次のような複数の要因により異なります。

- 数値の使用
- 多相メッセージを送信する割合
- フィールドに直接アクセスするか、アクセッサ・メソッドを使用するか
- 配列へのアクセス回数
- キャスト

Java バイトコードはコンパクトに設計されているため、ネイティブにコンパイルされるコードは、元のバイトコードよりもかなり大きくなることがあります。ただし、ネイティブ・コードは共有ライブラリに格納されているので、データベースの全ユーザー間で共有されます。

ほとんどの JVM では、メソッドがコールされたときに Java バイトコードをネイティブのマシン命令に変換する Just-In-Time コンパイラが使用されています。Oracle8*i* JVM Accelerator では、Java クラスの再コンパイルに Ahead-Of-Time アプローチが使用されます。

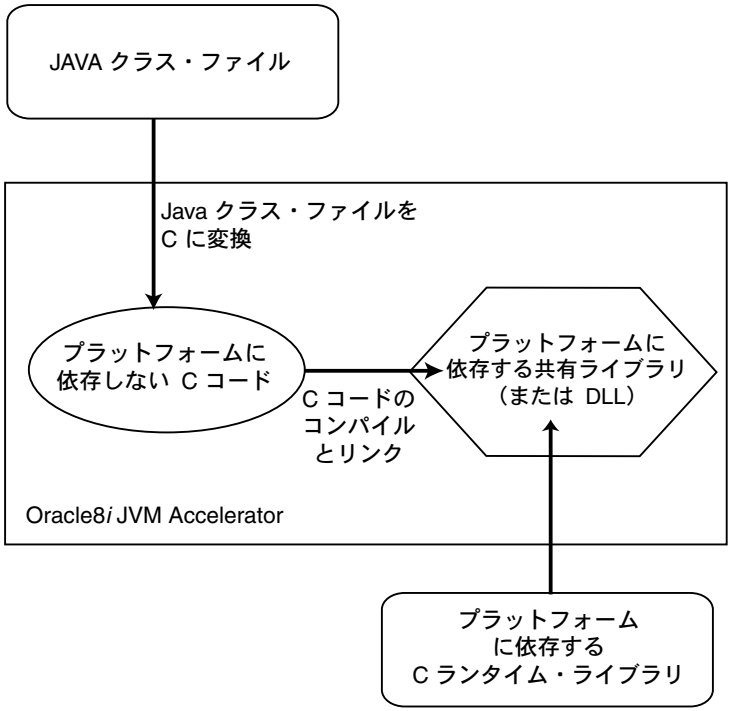
ネイティブ・コンパイラ	説明
Just-In-Time	JDK から要求される直前に Java 命令を変換する JVM の機能です。ネイティブ・コンパイラがどの程度正確にコード・ブランチと次の命令を予測するかによって、効果は違ってきます。不正確であれば、パフォーマンス上の効果はありません。
Ahead-Of-Time	Oracle8*i* JVM Accelerator では、実行前に、JAR ファイル内のすべての Java コードが Java パッケージによって編成されるネイティブの共有ライブラリにコンパイルされます。実行時に、Oracle8*i* JVM Accelerator によって、Java パッケージがネイティブにコンパイルされたかどうかがチェックされ、該当する場合には、配布された Java コードを解釈せずにマシン・コード・ライブラリが使用されます。

この静的なコンパイル・アプローチでは、ユーザー数やサーバー上を横断するコード・パスに関係なく、一定の大きなパフォーマンス上の効果が得られます。コンパイル後、ツールによって、静的にコンパイルされたライブラリが Oracle8*i* JVM にロードされ、ユーザー、プロセス、セッション間で共有されます。

Oracle8*i* JVM Accelerator の概要

ほとんどの Ahead-Of-Time ネイティブ・コンパイラでは、プラットフォーム固有の言語に直接コンパイルされます。これは、移植性の要件としては望ましくありません。図 6-1 に示すとおり、Oracle8*i* JVM Accelerator では、Java クラスがプラットフォームに依存しない C のバージョンに変換されます。この C コードはコンパイルおよびリンクされて、最終的にプラットフォーム固有で、ネイティブ・コンパイルされた、共有ライブラリまたは DLL として提供されます。

図 6-1 Oracle8*i* JVM Accelerator を使用したネイティブ・コンパイル



JAR ファイルを指定すると、Oracle8*i* JVM Accelerator では次のことが実行されます。

1. データベースにロードされたクラスが確認されます。

2. これらのクラスの Java バイトコードがデータベースから検索され、Oracle8*i* JVM Accelerator が起動されたプロジェクト・ディレクトリに格納されます。
3. Java バイトコードが C に変換されます。
4. C コードがプラットフォームの C コンパイラでコンパイルされてリンクされます。

   Oracle8*i* JVM Accelerator では、クライアント上の検索されたクラスが変換され、コンパイルされて、リンクされます。このため、このアプリケーションの配置先のプラットフォーム環境でネイティブにコンパイルする必要があります。その結果、プロジェクトのすべてのファイルに 1 つの配置 JAR ファイルが生成されます。
5. 生成された共有ライブラリは、`$ORACLE_HOME/javavm/admin` ディレクトリにロードされます。

注意： Oracle8*i* JVM Accelerator でネイティブにコンパイルされたライブラリは、Oracle8*i* JVM でのみ使用できます。これらのライブラリは、生成に使用された Oracle8*i* JVM の同じバージョンでのみ使用できます。アプリケーションをそれ以降のリリースでネイティブにコンパイルするには、これらのクラスを再コンパイルする必要があります。つまり、既存のライブラリのネイティブな再コンパイルは、どのアップグレード・プロセスでも自動的には実行されません。

Oracle8*i* JVM の Java コア・クラス・ライブラリ

すべての Java コア・クラス・ライブラリおよび Oracle 提供の Oracle8*i* JVM 内の Java コードは、実行速度を高めるようにネイティブ・コンパイルされています。Java クラスは、`$ORACLE_HOME/javavm/admin` 中に共有ライブラリとして格納され、各共有ライブラリは Java パッケージに対応しています。たとえば、Solaris の `orajox8java_lang.so` と Windows NT の `orajox8java_lang.dll` には、`java.lang` クラスが保持されています。パッケージ化とネーミングの詳細は、プラットフォームにより異なることがあります。Aurora JVM はネイティブ・コンパイル済みの Java ファイルを内部で使用して、実行時に必要に応じてオープンします。

Java アプリケーションのクラス・ライブラリのネイティブ・コンパイル

Oracle8*i* JVM Accelerator は、Oracle8*i* JVM 上で配置され、パフォーマンスの向上が必要な Java アプリケーション製品で使用できます。Oracle8*i* JVM Accelerator コマンドライン・ツール、`ncomp` では、コードがネイティブにコンパイルされて、Oracle8*i* JVM にロードされます。ただし、`ncomp` を使用するには、最初に初期設定を行う必要があります。

インストールの要件

Oracle8*i* JVM Accelerator を起動する前に、次のものをインストールする必要があります。

1. `ncomp` を実行するマシンのプラットフォームに C コンパイラをインストールします。

2. $ORACLE_HOME/javavm/jahome ディレクトリにある System*.properties ファイルで、正しいコンパイラ・コマンドとリンカ・コマンドが参照されることを確認します。コンパイラおよびリンカの情報はプラットフォームに固有のものなので、これらの項目の構成は各プラットフォームの README を参照してください。

3. 次の環境変数に、適切な JDK JAR ファイル、ライブラリ、バイナリ情報を追加します。

環境変数	必要な追加事項
JAVA_HOME	JDK のインストール先を設定します。
CLASSPATH	次のように、CLASSPATH に適切な JDK JAR ファイルを組み込みます。 ■ JDK 1.1 の場合は、$JAVA_HOME/lib/classes.zip を組み込みます。 ■ JDK 1.2 の場合は、$JAVA_HOME/lib/tools.jar と $JAVA_HOME/lib/dt.jar ファイルを組み込みます。
PATH	JDK バイナリ・パス $JAVA_HOME/bin を追加します。
LD_LIBRARY_PATH	JDK バイナリ・パス $JAVA_HOME/lib を追加します。

4. ncomp を実行するユーザーに次のロール・パーミッションとセキュリティ・パーミッションを付与します。

注意： DBA ロールには、JAVA_DEPLOY ロールと $ORACLE_HOME のすべてのファイルに対する FilePermission が含まれています。

---

a. JAVA_DEPLOY: ユーザーは、ncomp および deploync ユーティリティが実行されるサーバーに共有ライブラリを配置できるように、JAVA_DEPLOY ロールを付与されている必要があります。たとえば、次のように DAVE にロールを割り当てます。

```
SQL> GRANT JAVA_DEPLOY TO DAVE;
```

b. FilePermission: Oracle8*i* JVM Accelerator では、サーバー上でネイティブ・コンパイルされたコードが共有ライブラリに格納されます。Oracle8*i* JVM Accelerator でこれらのライブラリを格納するには、そのユーザーに、サーバー上の $ORACLE_HOME ディレクトリにあるファイルへの読込みおよび書込みアクセス権 FilePermission が付与されている必要があります。対象となるすべてのディレクトリへの FilePermission を付与する 1 つの方法としては、次のようにユーザーに JAVASYSPRIV ロールを付与します。

```
SQL> GRANT JAVASYSPRIV TO DAVE;
```

JAVASYSPRIV と `FilePermission` の付与の詳細は、このマニュアルの第 5 章を参照してください。

Oracle8*i* JVM Accelerator の実行

次の項で、Oracle8*i* JVM Accelerator を使用して基本的なネイティブ・コンパイルを行う方法を説明します。すべての `ncomp` オプションの詳細は、『Oracle8*i* Java Tools リファレンス』を参照してください。

注意： Java サーバー・コードをネイティブにコンパイルする前に、Oracle8*i* JVM でそのコードをロードしてテストしておく必要があります。テスト前のコードのネイティブ・コンパイルはお薦めしません。

JDeveloper で提供されるデバッガなどのデバッガが使用できるのは、解析済みの Java コードのみであることに注意してください。ネイティブにコンパイルされたライブラリはデバッグできません。

JAR ファイルに含まれたすべての Java クラスは、すでにデータベース内にロードされている必要があります。`ncomp` ツールを実行して、これらすべてのクラスをネイティブにコンパイルするように Oracle8*i* JVM Accelerator に指示します。次の方法により、`pubProject.JAR` ファイル内のすべてのクラスがネイティブにコンパイルされます。

```
ncomp -user scott/tiger pubProject.JAR
```

注意： ネイティブ・コンパイルではすべての Java クラスをコンパイルしてリンクする必要があるため、このプロセスは数時間にわたって実行される場合があります。コードをネイティブにコンパイルするために要する時間は、コンパイルするクラスの数や、マシンのハードウェアのタイプによって異なります。

この JAR ファイル内のいずれかのクラスを変更すると、変更されるクラスが含まれたパッケージの共有ライブラリが Oracle8*i* JVM Accelerator によって再コンパイルされます。この場合、すべての共有ライブラリが再コンパイルされるわけではありません。ただし、以前にネイティブ・コンパイルされたかどうかに関係なく、JAR ファイル内のすべてのクラスを再コンパイルするには、次のように、`ncomp` で `-force` オプションを指定して実行します。

```
ncomp -user scott/tiger -force pubProject.JAR
```

詳細は、『Oracle8*i* Java Tools リファレンス』を参照してください。

Java のメモリー使用量

データベースの標準インストールおよびカスタム・インストールでは、標準的な Java の開発向けにデータベースが設定されます。ただし、実行時の Java の使用するリソースは配置されたアプリケーションのシステム・リソースの使用量によって決まります。開発段階で使用するリソースの量は実行する作業によって大きく左右されます。以降では、必要なパフォーマンスに応じてメモリーを構成する方法と、SGA メモリーの使用量を調べる方法、および Java が使用するメモリーの問題を示すエラーを説明します。

- メモリー初期化パラメータの構成
- Java プール・メモリー
- Java プール・メモリー使用量の表示
- メモリー不足エラーの修正

メモリー初期化パラメータの構成

次のようなデータベース初期化パラメータを変更して、アプリケーションに必要なメモリー使用量がより正確に反映されるようにメモリー使用量をチューニングできます。

- **SHARED_POOL_SIZE** － 共有プール・メモリーは JVM 内のクラス・ローダーによって使用されます。クラス・ローダーはクラスをロードするたびに平均 8KB を使用します。共有プール・メモリーは、データベースにクラスをロードおよび解決するときに使用されます。また、データベース内でソースをコンパイルする場合やデータベースで Java リソース・オブジェクトを使用する場合にも使用されます。

  SHARED_POOL_SIZE で指定したメモリーは、loadjava の使用時に一時的に使用されます。データベースの初期化プロセスでは、インストールされているシード・データベースではなく、クリーンなデータベースに対して initjvm.sql が実行されます。したがって、8,000 クラス以上の Java バイナリをロードし、変換するため、SHARED_POOL_SIZE を 50MB に設定する必要があります。コール仕様を作成するとき、および実行時に動的にロードされた Java クラスをシステムで追跡するときも、SHARED_POOL_SIZE リソースが使用されます。

- **JAVA_POOL_SIZE** － Aurora のメモリー・マネージャは、ランタイム実行時にその他の Java 状態に対して java_pool_size で割り当てたメモリー量を割り当てます。このメモリーには、Java メソッドとクラス定義の共有メモリー表現のみでなく、コール終了時にセッション領域に移行した Java オブジェクトも含まれます。前者の場合は、メモリー・コストをすべての Java ユーザーとの間で分担します。後者の場合は、MTS において、各セッションの静的変数で保持する実際の状態数に基づき、JAVA_POOL_SIZE の割当てを調整する必要があります。「Java プール・メモリー」の詳細は、6-8 ページの「JAVA_POOL_SIZE」を参照してください。

- **JAVA_SOFT_SESSIONSPACE_LIMIT** － このパラメータで、1 つのセッションでの Java メモリー使用量にソフト・リミットを指定して、Java メモリー限界を増やす必要がある

場合に警告を表示することができます。セッションの起動時には、このパラメータで要求したメモリーが割り当てられます。

ユーザーのセッション期間中に、Java 状態がこのサイズを超えると、Aurora が警告を生成し、トレース・ファイルに書き込みます。デフォルトは 1MB です。配置するクラスのメモリー要件、特にセッション領域の使用量を把握する必要があります。

- **JAVA_MAX_SESSIONSPACE_SIZE** − サーバー上で実行されているユーザー起動型の Java プログラムを、メモリー使用量を制限しない方法で実行できる場合は、このパラメータによって使用可能なセッション領域のハード・リミットを設定できます。デフォルトは 4GB です。この上限は、通常到達することのない高い値に設定します。

  ユーザーのセッション期間中の Java の状態がこのサイズを超えそうになると、メモリー不足障害エラーが発生します。

Oracle8*i* JVM 固有のメモリー管理機能および読取り専用構造（バイトコードなど）を共有化したことにより、HelloWorld の実行に必要なセッションあたりのメモリー増分がわずか 35KB に押さえられます。CORBA および EJB アプリケーションで使用する Aurora/ORB など、状態を持つサーバー・アプリケーションの場合は、セッションあたりのメモリー増分は約 200KB です。このようなアプリケーションでは、次のコールまで多数の状態を静的変数に保持する必要があります。コール終了時に行う静的変数の移行制御の詳細は、6-11 ページの「コール終了時の移行」の項を参照してください。

Java プール・メモリー

Java プール・メモリーは、JVM のすべてのセッション固有 Java コード、およびデータのサーバー・メモリーで使用されます。Java プール・メモリーの使用方法は、Oracle8*i* Server の実行モードによって異なります。

専用サーバーで使用される Java プール・メモリー

セッション当たりで使用される各 Java クラスの共有部分。これには、コード・ベクトルやメソッドなどの読取り専用メモリーが含まれます。合計すると、各クラスについて約 4 ～ 8KB になります。

各セッションのセッション当たりの Java の状態は使用されません。専用サーバーの場合、これは SGA ではなく、PGA 内の UGA に格納されます。

Java ストアド・プロシージャのみを使用するアプリケーションではおそらく専用サーバーが使用されますが、必要な Java プール・メモリーは 10MB 程度です。

マルチスレッド・サーバー（MTS）で使用される Java プール・メモリー

セッション当たりで使用される各 Java クラスの共有部分。これには、ベクトルやメソッドなどの読取り専用メモリーが含まれます。合計すると、各クラスについて約 4 ～ 8KB になります。

各セッションの、各セッション状態に使用される UGA の一部は、SGA 内の Java プール・メモリーから割り当てられます。Java プール・メモリーのサイズは一定であるため、アプリ

ケーションに必要なメモリー量を予想し、必要な同時実行セッション数を乗じて、必要な Java プール・メモリーの総量を割り出してください。各 UGA は必要に応じて拡大または縮小できますが、UGA 全体が一定の Java プール領域に収まるようにしてください。

CORBA や EJB を使用するアプリケーションで使用される MTS サーバーでは、この数字は非常に大きくなります。Java を集中的に使用するマルチ・ユーザー・ベンチマークでは 1GB 以上が必要になります。現在の限界は不明ですが、プラットフォームによって異なります。

注意： クライアントでコードをコンパイルしてからサーバーにロードするのではなく、サーバー上でコードをコンパイルする場合は、JAVA_POOL_SIZE をデフォルトの 20MB 以上に設定する必要があります。EJB の配置ではサーバー上の Java コンパイラが使用されるため、JAVA_POOL_SIZE をさらに大きな値に設定する必要があります。

Java プール・メモリー使用量の表示

V$SGASTAT 表を表示すれば、Java プール・メモリーの使用量を調べることができます。この表にはプール、名前およびバイト数が表示されます。最後の 2 行には Java プール・メモリーの使用量と空き容量が表示されます。この 2 行の数字の合計値が、データベース初期化ファイルに設定したメモリー量と等しくなります。

```
SVRMGR> select * from v$sgastat;
POOL        NAME                        BYTES
----------- -------------------------- ----------
            fixed_sga                       69424
            db_block_buffers              2048000
            log_buffer                     524288
shared pool free memory                  22887532
shared pool miscellaneous                  559420
shared pool character set object            64080
shared pool State objects                   98504
shared pool message pool freequeue         231152
shared pool PL/SQL DIANA                  2275264
shared pool db_files                        72496
shared pool session heap                    59492
shared pool joxlod: init P                   7108
shared pool PLS non-lib hp                   2096
shared pool joxlod: in ehe                4367524
shared pool VIRTUAL CIRCUITS               162576
shared pool joxlod: in phe                2726452
shared pool long op statistics array        44000
shared pool table definiti                    160
shared pool KGK heap                         4372
shared pool table columns                  148336
shared pool db_block_hash_buckets           48792
shared pool dictionary cache              1948756
```

```
shared pool fixed allocation callback         320
shared pool SYSTEM PARAMETERS               63392
shared pool joxlod: init s                   7020
shared pool KQLS heap                     1570992
shared pool library cache                 6201988
shared pool trigger inform                  32876
shared pool sql area                      7015432
shared pool sessions                       211200
shared pool KGFF heap                        1320
shared pool joxs heap init                   4248
shared pool PL/SQL MPCODE                  405388
shared pool event statistics per sess      339200
shared pool db_block_buffers               136000
java pool   free memory                 30261248
java pool   memory in use               19742720
37 rows selected.
```

メモリー不足エラーの修正

- コンパイル時のメモリー不足
- ロード時のメモリー不足

コンパイル時のメモリー不足

（loadjava または deployejb で）コンパイル時にメモリー不足になると、エラーが表示されます。

```
A SQL exception occurred while compiling: : ORA-04031: 共有メモリーの %s バイトを割り当てできません ("shared pool","unknown object","joxlod: init h", "JOX: ioc_allocate_pal")
```

この場合には、データベースをシャットダウンして JAVA_POOL_SIZE をさらに大きな値に設定します。エラー・メッセージに「shared pool（共有プール）」と表示されるので、共有グローバル領域のメモリー不足と誤解する可能性がありますが、これは SHARED_POOL_SIZE を増やす必要があるという意味ではありません。この場合は、JAVA_POOL_SIZE を増やしてサーバーを再起動し、再度操作を行ってください。

ロード時のメモリー不足

クラスのロード中にメモリー不足になると、エラーが何も表示されずにロードが失敗し、データベースに無効なクラスが残されます。無効なクラスを起動または解決しようとすると、実行時に ClassNotFoundException 例外または NoClassDefFoundException 例

外が発生します。破損したクラス・ファイルをロードしようとしても同じ例外が発生します。この場合には次のような処置を実行してください。

- サーバーにロードするセットに実際にクラスが含まれていることを検証してください。1 つのクラスのロードを忘れたために、このクラスの追跡に膨大な時間を費やすことが少なくありません。
- `loadjava -force` オプションを使用して、ロードする新しいクラスで、サーバーにすでに常駐しているクラスを強制的に置き換えます。
- `loadjava -resolve` オプションを使用してロード・プロセスでのクラスの解決を試みてください。これにより、実行時ではなくロード時に、不足するクラスをキャッチできます。
- 新しくロードされたクラスの状態を再確認するために、そのクラスが含まれているスキーマのデータベースに対して接続を行い、次のように実行してください。

  ```
  select * from user_objects where object_name = dbms_java.shortname('');
  ```

  STATUS フィールドに「VALID」と表示されていることを確認してください。`loadjava` からメモリーの問題や「connection lost（接続が切断されました）」などのメッセージが表示された場合は、SHARED_POOL_SIZE および JAVA_POOL_SIZE を増やして再試行してください。

コール終了時の移行

Aurora では、Java プログラムの状態を次のコールまで保持するために、コール終了時に、静的変数から参照可能なすべてのオブジェクトをセッション・スペースに移行します。セッション・スペースはクライアントのセッションの中にあり、コール間にまたがって存続する静的変数およびオブジェクトが格納されます。この移行処理は、各コールの終了時に Aurora によって行われます。開発者がする作業は必要ありません。

この移行処理にはメモリーが使用され、パフォーマンスも影響を受けるため、コール間で存続させる静的変数やオブジェクトを最小限に抑えるように注意してください。不必要に静的変数にオブジェクトを格納すると、メモリー・マネージャが移行処理を実行し、セッション当たりのリソースを使用することになり、不要な負担がかかります。静的変数を必要最低限に抑えることで、メモリー・マネージャの負担を軽減し、サーバーのパフォーマンスを向上できます。

Java プログラムを同時に実行できるユーザー数を最大にするには、セッションのフットプリントを最小にする必要があります。特に、拡張性を最大にするには、非アクティブなセッションが占有するメモリー領域を必要最小限にする必要があります。フットプリントを最小化する簡単な方法は、毎回コール終了時に大きなデータ構造を解放することです。データ構造は、別のコールで再度必要になったときに再作成できます。Aurora JVM には、コール終了時など、セッションが非アクティブになる直前に、指定された Java メソッドをコールするためのメカニズムがあります。

このメカニズムは EndOfCallRegistry の通知と呼ばれます。このメカニズムにより、コール終了時に静的変数を消去して、次のコールを受けたときに遅延初期化を使用して変数を再初期化ことができます。これは、コール間での格納にメモリー・マネージャが必要とする記憶域の容量を十分とれない場合以外は実行しないでください。これは、より複雑で状態数の多いサーバー・アプリケーションを Java で実装する場合に、考慮する必要があります。

コール終了時にデータ構造を NULL で埋め、その後新しいコールごとにそれを再作成するかどうかの決定は、時間と領域のトレードオフの問題です。データ構造を再作成することによって、処理時間が多少長くなりますが、次のコールまでデータ構造を保持する必要がないので、メモリー領域をかなり節約できます。また、セッション領域に移行されたオブジェクト（特に大型のオブジェクト）へのアクセスは効率が悪くなります。これは、セッション領域とコール領域のオブジェクト表現の違いによるものです。

このような最適化の対象として、次のデータ構造があります。

- バッファまたはキャッシュ。
- 静的フィールド（配列など）。これは、いったん初期化すると、プログラムの実行が完了するまで変化しません。
- 動的に作成されるデータ構造。最適化によって、コール間のメモリー使用効率の他、コール中の速度効率も向上できます。このデータ構造の最適化を行うことによりコードが複雑になり、管理が困難になることもあります。したがって、手間に見合う程度の領域が節約される場合のみ、この最適化の実施を検討してください。

コール終了時の最適化に対する Oracle 固有のサポート

バッファ、フィールドまたはデータ構造を作成するときに、コール終了時に消去する静的変数を登録できます。Oracle 固有の oracle.aurora.memoryManager.EndOfCallRegistry クラスでは、registerCallback メソッドが、Callback オブジェクトを実装するオブジェクトを取り込みます。registerCallback オブジェクトには、コール終了時までこのオブジェクトが格納されます。コールが終了すると、Aurora は登録されたすべての Callback オブジェクト内の act メソッドをコールします。Callback オブジェクト内の act メソッドは、ユーザー定義のバッファ、フィールドまたはデータ構造を消去するために実装されます。消去が終わると、Callback はレジストリから削除されます。

注意： コール終了時がセッションの終了時でもある場合は、いずれにしてもセッション・スペースが消去されるため、コールバックは起動されません。

EndOfCallRegistry の使用方法は、操作対象のオブジェクトが静的フィールドに保持されているか、インスタンス・フィールドに保持されているかによります。

- 静的フィールド － クラス全体に対応付けられた状態を消去するには EndOfCallRegistry を使用します。この場合、Callback オブジェクトはプライ

ベートな静的フィールドに保持する必要があります。コール間で削除されたキャッシュ内のデータにアクセスする必要があるコードは、キャッシュ・データを遅延して作成または再作成するメソッドをコールする必要があります。次の例はこれを実行する方法です。

1. 静的フィールド thunk に Callback オブジェクトを作成します。
2. この Callback オブジェクトをコール終了時の移行に登録します。
3. Callback.act メソッドを実装して、Callback オブジェクトそのものを含むすべての静的変数を解放します。
4. キャッシュを遅延して再作成するための createCachedField メソッドを渡します。

ユーザーがキャッシュを作成すると、Callback オブジェクトが getCachedField メソッドに自動的に登録されます。コール終了時に、Aurora は登録済みの Callback.act メソッドをコールし、静的メモリーを解放します。

```
import oracle.aurora.memoryManager.Callback;
import oracle.aurora.memoryManager.EndOfCallRegistry;

class Example {
  static Object cachedField = null;
  private static Callback thunk = null;

  static void clearCachedField() {
      // clear out both the cached field, and the thunk so they don't
      // take up session space between calls
       cachedField = null;
       thunk = null;
  }

  private static Object getCachedField() {
   if (cachedField == null) {
      // save thunk in static field so it doesn't get reclaimed
      // by garbage collector
      thunk = new Callback () {
            public void act(Object obj) {
                  Example.clearCachedField();
            }
      };

      // register thunk to clear cachedField at end-of-call.
      EndOfCallRegistry.registerCallback(thunk);
      // finally, set cached field
      cachedField = createCachedField();
   }
   return cachedField;
```

```
  }

  private static Object createCachedField() {
  ....
  }
}
```

- インスタンス・フィールド － インスタンス・フィールドに保持されているデータ構造を消去するには、EndOfCallRegistry を使用します。たとえば、状態があるクラスの各インスタンスに対応付けられている場合、各インスタンスにはそのインスタンスのキャッシュ状態を保持するフィールドがあり、必要に応じてキャッシュ・フィールドが埋められます。状態を確実にキャッシュに入れるメソッドを使用してキャッシュ・フィールドにアクセスできます。

  1. インスタンスを Callback オブジェクトとして実装します。
  2. Callback.act メソッドを実装して、そのインスタンスのフィールドを解放します。
  3. ユーザーがキャッシュを要求すると、Callback オブジェクトがコール終了時の移行に自動的に登録されます。
  4. キャッシュを遅延して再作成するための createCachedField メソッドを渡します。

  ユーザーがキャッシュを作成すると、Callback オブジェクトが getCachedField メソッドに自動的に登録されます。コール終了時に、Aurora は登録済みの Callback.act メソッドをコールし、キャッシュを解放します。

  この方法を使用することによって、Callback オブジェクトとそのインスタンスが同じオブジェクトになるため、Callback オブジェクトの存続期間とそのインスタンスの存続期間が等しくなります。

```
import oracle.aurora.memoryManager.Callback;
import oracle.aurora.memoryManager.EndOfCallRegistry;

class Example2 implements Callback {
   private Object cachedField = null;

   public void act(Object obj) {
    // clear cached field
    cachedField = null;
    obj = null;
   }

  // our accessor method
  private static Object getCachedField() {
   if (cachedField == null) {
      // if cachedField is not filled in then we need to
```

```
        // register self, and fill it in.
        EndOfCallRegistry.registerCallback(self);
        cachedField = createCachedField();
      }
      return cachedField;
    }

    private Object createCachedField() {
    ....
    }
  }
```

弱い参照を使用した表は、エンドオブコール・コールバックのレジストリを保持します。Callback オブジェクトと値のどちらも Java プログラムから参照できない場合（JLS の項 12.6 参照）は、Callback オブジェクトと値が表から削除されます。弱い参照を使用した表を使用してコールバックを保持すると、コールバックを登録しても、ガベージ・コレクタによるオブジェクトの削除を避けられません。したがって、必要なコールバックをプログラムで保持する必要があります。表に保持しておくことはできません。

アプリケーションでは、コール終了時の通知を他の用途にも使用できます。以降では、`EndOfCallRegistry` クラスおよび Callback インタフェースのメソッドを詳しく説明します。

EndOfCallRegistry.registerCallback メソッド

`registerCallback` メソッドはレジストリに `Callback` オブジェクトをインストールします。コールが終了すると、Aurora は登録されたすべての `Callback` オブジェクト内の `act` メソッドをコールします。

`Callback` オブジェクトは自動的に、または `value` オブジェクトを使用して登録できます。オブジェクトに格納されたその他の情報を `act` に渡す必要があれば、このオブジェクトを value パラメータの中に登録できます。

```
public static void registerCallback(Callback thunk, Object value);
public static void registerCallback(Callback thunk);
```

パラメータ	説明
thunk	コール終了時の移行でコールされる Callback オブジェクト。
value	オブジェクトに格納されたその他の情報を `act` に渡す必要があれば、このオブジェクトを `value` パラメータの中に登録できます。場合によっては、callback に必要な状態の保持に `value` パラメータが必要になります。ただし、大半のユーザーは、`value` を指定する必要はありません。

EndOfCallRegistry.runCallbacks メソッド

```
static void runCallbacks()
```

JVM はコール終了時にこのメソッドをコールして、registerCallback で登録された Callback オブジェクトごとに act をコールします。このメソッドは、絶対にコード内でコールしないでください。コール終了時にこのメソッドがコールされてから、オブジェクトが移行され、最終処理が行われます。

Callback インタフェース

```
Interface oracle.aurora.memoryManager.Callback
```

EndOfCallRegistry.registerCallback で登録するオブジェクトは、Callback インタフェースを実装します。このインタフェースは、アプリケーションで、コール終了時に通知が必要な場合に役立ちます。

Callback.act メソッド

```
public void act(Object value)
```

コールの終了時に実行する必要がある任意のアクティビティを実装できます。通常は、このメソッドにはセッション・スペースに保存されるメモリーを消去するプロシージャが含まれます。

メモリー・プロファイリング・ユーティリティ

メモリー・プロファイリング・ユーティリティ（MemStat）の目的は、Oracle8*i* Java プログラムの静的変数を使用してアクセス可能な割当てメモリーのトレース、プロファイル、レポートを行うことです。このレポートの情報を使用すれば、Java クラスの不要な静的データを検索して削除できるので、Java プログラムの静的フットプリントを削減して、データベースへの繰返し Java コールのパフォーマンスを向上させることができます。

Oracle8*i* JVM では、次の 3 種類のメモリーが使用されています。

- データベースへのコール（再帰的な場合がある）の間に使用される、コール・メモリー。
- データベースに接続されたセッションの間に使用される、セッション・メモリー。
- データベース・インスタンスが実行されている間、持続的に使用される、パーマネントまたはグローバル・メモリー。

Java 言語セマンティックでは、静的変数がコール間で持続するように指定されます。各コールの最後で、Oracle8*i* JVM によって、各クラスの静的変数を通じてアクセスできるコール・メモリーがセッション・メモリーにコピーされるので、Java クラスのメソッドへの後続のコールに保存してリストアできます。大量の静的データまたは相互接続するオブジェクトの複雑なグラフがあると、JVM でセッション・メモリーを割り当て、静的データをそれにコピーしている間、コール終了プロセス中に大幅なオーバーヘッドが生じます。

オブジェクト指向のプログラムをチューニングしてパフォーマンスを高速化するために取られる通常の方法は、プログラムから不要なオブジェクトの割当てを排除することです。たとえば、オブジェクトのインスタンスを必要に応じて新規に作成するのではなく、よく使用されるオブジェクトの静的インスタンスを作成しておき必要に応じて再利用できます。ただし、異なるデータベース・メモリー間の対話では、そうした方法は複雑になり、動的オブジェクトを割り当てるためのスピードと、静的オブジェクトのコール終了のコピーに必要な領域を比較して、どちらがより効率がよいかという分析が必要になることがあります。静的オブジェクトが大きい場合、静的オブジェクトが多数ある場合、または多数のコールがある場合は、コール終了プロセス中にオブジェクトのグラフがやり取りされるため、オブジェクトのキャッシュによって得られるスピードの利点は失われる場合があります。

MemStat の機能

起動方法により、MemStat では、1 つのクラスまたは現在のセッションにロードされたすべてのクラスが分析されます。MemStat では、各クラスごとにクラスの静的変数が列挙されます。これらの変数は、ルートとして認識されます。MemStat では、各変数の構造によって、次のように 3 つの異なる分析が実行されます。

- 変数が整数などのプリミティブ・オブジェクトである場合は、そのクラスとサイズが記録されます。
- 変数がプリミティブ・オブジェクトでない場合（たとえば、他のオブジェクトを参照するなど）、参照をたどって、参照先のすべてのオブジェクトが再帰的に列挙されます。
- 変数が配列である場合は、配列のすべての要素が列挙されます。

このプロセスは、すべての静的変数から到達できる全オブジェクトが記録されるまで、再帰的に繰り返されます。大きなオブジェクト・グラフに循環が含まれることもあるので、分析中に検出される循環参照も記録されます。

MemStat の使用

MemStat の目的は、プログラムの静的変数からアクセスできるオブジェクト・グラフの分析とレポートです。プログラムのどこからでも分析を直接起動できますが、コールの最後で分析を実行するように登録することもできます。

データベース・コールの標準出力メカニズムはないので、MemStat では、指定されたディレクトリに HTML ファイルの形式でレポートが生成されます。レポートの終了後、HTML の表示が可能な Web ブラウザでこれらのファイルを表示できます。

MemStat は、クラス `oracle.aurora.memstat.MemStat` に 3 つの静的メソッドで実装されます。

MemStat は、次の 3 つの方法でコールできます。

- 1 つのクラスの静的変数について即時にレポートする。

- 現在のセッションで使用されるすべてのクラスの静的変数について即時にレポートする。
- 現在のセッションで使用されるすべてのクラスの静的変数についてコール終了時にレポートする。

1 つのクラスについてレポートする場合は、次のメソッド・コールを使用します。

```
MemStat.writeDump (Class MyClass, String outputPath, String filePrefix);
```

ロードされた全クラスについてレポートする場合は、次のメソッド・コールを使用します。

```
MemStat.writeDump (String outputPath, String filePrefix);
```

コール終了時に、ロードされた全クラスについてレポートする場合は、次のメソッド・コールを使用します。

```
MemStat.writeDumpAtEOC (String outputPath, String filePrefix);
```

`outputPath` パラメータは、MemStat レポートが生成されるディレクトリを示します。`outputPath` 文字列は、レポートが生成されるプラットフォームに適したファイル名形式で指定する必要があります。たとえば、`/home/base/memstat` は、Solaris プラットフォームの形式です。Windows の形式は、`c:¥¥base¥¥memstat` となります。Java では、文字列内に、スラッシュではなく、`¥` マークを 2 つ続けて入力する必要があります。

`filePrefix` は、`outputPath` ディレクトリに生成される HTML ファイルの基本ファイル名です。MemStat レポートは量が多くなる場合がありますが、ほとんどの Web ブラウザでは表示されるサイズに制限があります。そのため、MemStat では、長いレポートは別個のファイルに分割されます。`filePrefix` は、指定されたレポートのすべてのファイル名の基礎となり、段階的に大きくなる数字の接尾辞がついています。たとえば、`test` レポートでは 3 つのファイルが生成され、主要なレポート・ファイルの名前は `test.htm`、追加のレポート・ファイルの名前は `test1.htm`、`test2.htm` となります。

指定したコールで MemStat を 2 回以上コールする場合は、異なるベース名または異なる出力ディレクトリを使用するよう注意してください。そうでないと、後続のレポートで前のレポートが上書きされます。たとえば、メモリーを消費する操作を実行する前または後で、最初のレポートを `before`、2 番目のレポートを `after` として MemStat をコールすると、名前が競合することなく、同じディレクトリにレポート・ファイルを書き込むことができます。複数のディレクトリを使用する方法は少し複雑です。書込み先の各ディレクトリに、それぞれ別個に FilePermissions（次のサンプルを参照）を付与する必要があります。

サンプルの MemStat コールを示します。

```
MemStat.writeDump (MyClass.class, "c:¥¥base¥¥memstat", "myclass");
MemStat.writeDump ("/home/base/memstat", "test");
MemStat.writeDumpAtEOC ("/home/base/memstat", "eoc");
```

MemStat パーミッション

MemStat では、ユーザーに付与する特定のパーミッション、または MemStat を実行するためのロールが必要です。MemStat は、Oracle サーバー・プロセスで実行されるので、これらのパーミッションでは、MemStat で必要なリソースへのアクセス権が付与されます。

- 図式内のオブジェクトのプライベート変数にアクセスします。このパーミッションが付与されなくても MemStat は実行されますが、プライベート変数によって指定されたオブジェクトはトレースされません。
- 現在のセッションでロードするクラスを決定するために、Java ランタイム・システムにアクセスします。このパーミッションが付与されないと、MemStat では、ロードされたクラスは 0（ゼロ）として検出され、空のレポートが生成されます。
- MemStat HTML レポートが生成されるサーバー・ファイル・システムにアクセスします。このパーミッションが付与されないと、MemStat では、レポート・ファイルの作成時に例外が呼び出されます。

次の SQL 文では、これらのパーミッションがユーザー JIM に付与されます。

```
call dbms_java.grant_permission ('JIM',
     'SYS:java.lang.reflect.ReflectPermission', 'suppressAccessChecks', null);

call dbms_java.grant_permission ('JIM',
     'SYS:oracle.aurora.security.JServerPermission', 'JRIExtensions', null);

call dbms_java.grant_permission ('JIM', 'SYS:java.io.FilePermission',
               '/home/base/memstat', 'read,write');          // Solaris

call dbms_java.grant_permission ('JIM', 'SYS:java.io.FilePermission',
               'c:¥base¥memstat', 'read,write');             // Windows
```

Oracle Server を Windows プラットフォームで実行する場合、MemStat コールで指定された出力ファイル・パスは、SQL `grant_permission` コールのパスとわずかに異なります。Java 文字列では、2 つの ¥ マークを使用する必要があります。SQL では、1 つの ¥ マークのみを使用します。

MemStat レポート形式

この項では、MemStat レポートの形式を説明します。次の例を参照してください。HTML の表示が可能な Web ブラウザを使用すれば、MemStat 出力レポートをブラウズできます。その場合、たとえば、`c:¥base¥memstat¥test.htm` のように、指定したベース・ファイル名でブラウザを指定します。

レポートは、MemStat がコールされた時点のメモリー使用量のサマリーで始まります。このサマリーにより、分析対象の Java コールによって使用されるメモリーの量を全体的に把握することができます。

サマリーの次には、MemStat 分析中にやり取りされる一意のクラスがリストされます。MemStat では、メモリー分析中に検出されるすべてのオブジェクトについて、各コール、セッション、パーマネント・メモリーにそのクラスとサイズが記録されています。サイズが最大のオブジェクトが最初にソートされます。これは、最大のサイズのオブジェクトを排除すると、最も効率的にパフォーマンスを向上できるためです。リストは、実際、*max (call, max (session, permanent))* として計算され、これら 3 つのサイズの最大のオブジェクトでソートされます。また、この表には、各クラスについて、そのクラスのオブジェクトが占めるバイト数、オブジェクト数、オブジェクトの最小サイズ、最大サイズおよび平均サイズが示され、配列の場合は、サイズの標準偏差が示されています。

クラス・サマリーの次には、各ルート・オブジェクトを説明する 1 つ以上の表があります。オブジェクトのタイトルは、オブジェクトのパッケージとクラスを説明しています。表の各行は、次のものを説明しています。

- オブジェクトのフィールド
- そのフィールドのオブジェクトの説明
- 3 つの各メモリー領域のオブジェクトの合計サイズ

ルート・オブジェクトの次には、ルートによって指定されたオブジェクトがあります。オブジェクトは、分割規則によって区切られています。1 つ、2 つまたは 3 つの表で各オブジェクトを説明しています。

- オブジェクト
- オブジェクトが参照する他のオブジェクト
- このオブジェクトを参照するオブジェクト

各オブジェクトのタイトルは、オブジェクトが常駐するメモリーの種類、コール、セッションまたはパーマネントのいずれかを示します。各オブジェクトは、次のもので記述されます。

- 一意の識別子
- オブジェクトに `toString()` メソッドを適用する出力
- オブジェクトの直接サイズ : オブジェクト・ヘッダーとオブジェクトのフィールドのサイズ
- オブジェクトの合計サイズ : オブジェクトが参照するすべてのオブジェクトの合計サイズ

他のオブジェクトを参照するオブジェクトは、参照するオブジェクトを示す表への HTML リンクによってリンクされます。テキスト文書のハイパーリンクをナビゲートするときに、これらのリンクを使用してオブジェクト・グラフをナビゲートできます。

サンプル出力

MemStat の結果

2000-06-01 17:07:05.645

ランタイム値	
セッション・サイズ	143360
NewSpace のサイズ	262144
使用可能な NewSpace	true
インターン表のサイズ	261814

合計メモリー割当て			
	コール	セッション	パーマネント
オブジェクト	726	926	3217
合計サイズ	54861	39348	127418
最小	12	12	12
最大	16396	2060	8076
平均	75.6	42.5	39.6
標準偏差	679.2	93.7	233.7

クラスごとの割当てオブジェクト（コール、セッション、パーマネント）

クラス	バイト			オブジェクト			最小			最大			平均			標準偏差		
char[]	25316	5134	43296	161	104	1177	2	0	0	16384	800	8064	157.2	49.4	36.8	1,283.2	105.0	276.9
java.lang.String	3816	3240	30528	159	135	1272												
java.util.Hash table$Entry	4956	10696	12460	177	382	445												
byte[]	8195	2421	2107	2	34	57	3	0	0	8192	2048	1024	4,097.5	71.2	37.0	4,094.5	344.2	143.3

java.util.PropertyPermission からアクセス可能なオブジェクト				
フィールド	参照マニュアル	合計サイズ		
keyValueSeparators	946: java.lang.String	0	0	50
strictKeyValueSeparators	948: java.lang.String	0	0	40
specialSaveChars	950: java.lang.String	0	0	54
whiteSpaceChars	952: java.lang.String	0	0	46
hexDigit	954: char[16]	0	0	44

用語集

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）。Java の場合は、Java プログラマに特定の機能を提供するために定義されているクラス群とメソッド群で構成されます。JDBC と SQLJ は、SQL データにアクセスするための API です。

CLASSPATH

ディレクトリ・ツリーのルートを指定するときに JDK または JRE で使用する環境変数（またはコマンドラインの引数）。Java のソース、クラスおよびリソースがこのルートの中で検索されます。

CORBA

Common Object Request Broker Architecture。Object Management Group（OMG）によって策定された CORBA は、言語に依存しないアーキテクチャです。Object Request Broker（ORB）によって接続されているネットワークの論理層および物理層に、オブジェクト指向のプログラミング・ロジックを分散化します。

EJB

Enterprise JavaBeans。Oracle8*i* JVM は、Enterprise JavaBeans 1.0 仕様の実装を提供します。Oracle8*i* JVM は Session Bean のみをサポートします。Entity Bean は、EJB 1.0 仕様のオプション部分であり、Oracle8*i* JVM ではサポートされません。

IDE

統合開発環境（Integrated Development Environment）。Java IDE はクライアント・ワークステーションで動作し、Java のクラス・ライブラリと開発ツールにアクセスするためのグラフィカル・ユーザー・インタフェースを提供します。

IDL

Interface Definition Language。CORBA では、プラットフォームに依存しないこの言語によって CORBA コンポーネントへのインタフェースを定義します。IDL を Java コードに変換するには、`idl2java` などのツールを使用します。

Java スキーマ・オブジェクト（Java Schema Object）

Oracle8*i* JVM で、Oracle8*i* データベースに格納されている Java のソース、バイナリまたはリソースを表す用語。JDK では、次の 3 種類のファイルが Java スキーマ・オブジェクトに対応します。.java、.class、および JDK CLASSPATH で使用するその他のファイル（.properties ファイルなど）。

JCK

Java Compatibility Kit。Java 仮想マシンをテストし、Java コンパイラの Java 標準対応を検証する Java クラス群です。JCK のリリースは Sun Microsystems の JDK リリースに対応しています。ただし、Oracle8*i* JVM の場合は、Java クラスのみが Sun Microsystems の JDK に合致しています。仮想マシンは対応していません。

JDBC

Java Database Connectivity。ベンダーに依存しない方法でデータベースにアクセスできる標準 Java クラス群です。

JDBC ドライバ（JDBC Driver）

JDBC のベンダー固有部分で、特定のデータベースへのアクセスを提供します。Oracle が提供する JDBC ドライバは、thin、OCI および server の 3 つです。

JDK

Java Development Kit。Java 仮想マシンと、Sun Microsystems 社提供の Java クラス群およびツール群によって、Java アプリケーションとアプレットの開発がサポートされます。JDK には、Java コンパイラがあります。JRE にはありません。

JRE

Java Runtime Environment。実行時に Java アプリケーションまたはアプレットをサポートする Java クラス群。JRE クラス群は JDK クラス群のサブセットです。

Oracle8*i* JVM

オラクル社のスケーラブルな Java サーバー・プラットフォーム。Oracle8*i* JVM の構成要素は、Oracle8*i* データベース・サーバーで動作する Aurora Java 仮想マシン、Java ランタイム環境および Oracle 固有の拡張機能（Aurora/ORB および Enterprise JavaBeans の実装など）です。

ORB

Object Request Broker。ORB はサーバー上で実行するプログラムです。クライアントからコード化されたメッセージを受け取ってサーバー側オブジェクトで実行し、オブジェクトをクライアントに戻します。通常、ORB はクライアントが使用できる多種のサービス（ネーム・サービスなど）をサポートします。Oracle8*i* JVM の ORB は Aurora/ORB です。

SQLJ

Java に埋め込まれた SQL。SQL データにアクセスするための SQL 文を Java プログラムに埋め込む方法を定義した規格です。トランスレータは SQLJ プログラムを標準 JDBC プログラムに変換します。

オブジェクト・グラフ（Object Graph）

オブジェクトが参照しているオブジェクトはフィールドに保持されています。このオブジェクト群がオブジェクト・グラフを形成します。メモリー・マネージャは、静的変数に保持されているオブジェクト・グラフを移行します。つまり、静的フィールドに保持されているオブジェクトのみではなく、このオブジェクトが参照しているオブジェクトおよび参照で連鎖しているすべてのオブジェクトを移行します。

ガベージ・コレクション（Garbage Collection）

Java 仮想マシンによって提供される自動ストレージ再生機能の通称。

ガベージ・コレクション（Garbage Collection）

メモリー・マネージャは、コール終了後も残存するオブジェクト、つまりセッション内で Java 静的変数からアクセスできるオブジェクトを、このメモリーに保持します。

厳密な型指定（Strong Typing）

Java の要件の 1 つ。Java では、各フィールドと変数のクラス、および各メソッドの返り値の型を明示的に宣言する必要があります。

コール終了（End-of-Call）

セッション内では Java を複数回起動することがあります。そのたびに、Java コードの実行が完了した位置で、コール終了が発生します。コール終了時に、メモリー・マネージャが静的変数をセッション領域に移行します。

コール・メモリー（Call Memory）

新しいオブジェクトを割り当てるときにメモリー・マネージャが使用するメモリー。

コア・クラス・ライブラリ（Core Class Libraries）

通常、Sun Microsystems の JDK で配布される Java パッケージ java.*。一部の sun.* パッケージもコア・クラス・ライブラリと呼びます。

コオペラティブ・マルチタスキング（Cooperative Multitasking）

他のスレッドが実行できるように、コード内の、実行を中断させるのに適した位置に `Thread.yield()` メソッドのコールを挿入します。この方法はエラーの原因になりがちです。プログラムの記述時はプログラムの同時動作へのアクセスが難しいためです。

コンテキスト・スイッチ（Context Switch）

単一プロセッサ・システムでは、カレント・スレッドに優先順位の高いスレッドまたは外部のイベントが割り込むと、システムがスレッドを別のスレッドに切り替えます。通常、優先順位またはスレッドの待機時間に基づき、ディスパッチ先スレッドが選択されます。

システム（System）

通常、ハードウェア、オペレーティング・システムおよび Java 仮想マシンの組合せです。

スレッド（Thread）

レジスタ群、1 つのプログラム・カウンタおよび 1 つのスタックで構成される実行コンテキスト。

対称マルチプロセッシング（Symmetric Multiprocessing: SMP）

ハードウェアが複数のプロセッサで構成されており、オペレーティング・システムは、プロセッサの負荷および可用性に応じて、スレッドを各プロセッサにマッピングします。OS スレッドと Java スレッドのマッピングは、Java 仮想マシンが行います。このメカニズムでは、複数のスレッド間で同時処理が可能ですが、小さなプログラミング・エラーにより、同期しているオブジェクト間でデッドロックによる競合が発生することがあります。

遅延初期化（Lazy Initialization）

従来のプロセッサの機能をエミュレートするプログラム。Java 仮想マシンは Java 仮想マシン仕様の要件に準拠する必要があります。Oracle8*i* JVM の仮想マシンは Aurora 仮想マシンです。

遅延初期化（Lazy Initialization）

データ初期化手法の 1 つ。通常、アクセッサ・メソッドで使用します。フィールドが初期化済み（NULL 以外）かどうかを調べてから、初期化済みオブジェクトをそのフィールドに戻します。このチェックに伴うオーバーヘッドはたいてい少なく、特にアクセスされる可能性のほとんどないにないデータ構造を初期化する場合に比べるとはるかに少なくて済みます。この手法とコール終了処理を併用すると、セッション領域のオーバーヘッドが最小になります。

ディスパッチ（Dispatch）

システムは現在実行中のスレッドの状態を保存し、実行するスレッドの状態をリストアし、新しいスレッドのストアド・プログラム・カウンタに分岐します。これにより、割込みがなかったかのように効率的に新しいスレッドが続行されます。

デッドロック（Deadlock）

同期している 2 つ以上の Java オブジェクト間の競合状態。相手をロックしようとしたときに、相手側からロックされたために相手をロックできない状態を言います。たとえば、オブジェクト A がオブジェクト B をロックしようとしたときに、オブジェクト B がオブジェクト A をロックしようとすると、デッドロックになります。プリエンプティブ方式の Java 仮想マシンはデッドロックの検出も防止も行えないので、この競合状態のデバッグは困難です。デッドロックが検出されないと、デッドロックされたプログラムは単にハングアップします。

ドライバ（Driver）

JDBC の場合と同様、ネットワーク経由の通信および SQL データへのアクセスに使用するライブラリを特定するコード層。Oracle8*i* JVM の JDBC ドライバは、thin、OCI および server の 3 つです。

バイトコード（Bytecodes）

マシンに依存しないシングルバイト命令。Java ソース・コードは Java コンパイラによってバイトコードにコンパイルされます。

プリエンプティブ・マルチタスキング（Preemptive Multitasking）

特定の条件が発生すると、オペレーティング・システムはスレッドに割り込みます、つまり、そのスレッドから別のスレッドに制御を移します。この割込みは、優先順位の高いスレッドの実行準備が整ったとき、または外部から割り込まれたとき、または現在のスレッドがソケットの受付けやファイルの読出しなどの I/O 操作を待機しているときに発生します。一部の Java 仮想マシンでは、特定の仮想マシン命令で現在のスレッドに割り込むことによって、ラウンドロビン方式の割込みを実装しています。つまり、逆方向への分岐、メソッドのコールなどによって、制御フローを変更します。Java スレッドが実際のオペレーティング・システムのスレッドにマッピングされる Java 仮想マシンでは、割込みが、仮想マシンの制御下ではなく、オペレーティング・システムのカーネルで発生します。この結果、適切なパラレル化が可能になりますが、ガベージ・コレクションなどの仮想マシンのアクティビティが複雑になります。

プロセス（Process）

1 つのアドレス領域と 1 つ以上のスレッドで構成されます。

索引

記号

#sql，3-12，3-14

A

act メソッド，6-12
Aurora
　定義，2-2

B

BasicPermission，5-14
Bean，1-19

C

Callback クラス
　act メソッド，6-12
Class class
　getClassLoader メソッド，2-32
classes111.zip，2-8
classes12.zip，2-10
classForNameAndSchema メソッド，2-33
ClassForName クラス
　lookupClass メソッド，2-33
ClassNotFoundException，2-31
CLASSPATH，2-14，2-31，4-9
.class ファイル，2-14，2-22，2-23
CodeSource クラス，5-5
　equals メソッド，5-5
　implies メソッド，5-5
COM，1-20
CORBA
　CosNaming，3-8
　Java 2 サポート，2-10
　Java 2 を使用する純粋な CORBA，2-12
　起動，3-2
　構成，4-7，4-8
　セキュリティ，5-2
　定義済み，viii，1-15，1-19，2-5，3-2，3-6，3-7
　マニュアル，1-20
　例，3-9
CosNaming，3-8

D

DBA_JAVA_POLICY ビュー，5-6，5-18，5-20
DBMS_JAVA package
　get_compiler_option メソッド，4-4
　reset_compiler_option メソッド，4-4
　set_compiler_option メソッド，4-4
DBMS_JAVA パッケージ，3-20，4-3
　delete_permission メソッド，4-5，5-18
　disable_permission メソッド，4-5，5-18
　dropjava メソッド，4-4
　enable_permission メソッド，4-5，5-18
　grant_permission メソッド，4-5，5-8，5-10
　grant_policy_permission メソッド，4-5，5-12，5-20
　loadjava メソッド，4-4
　longname メソッド，2-27，2-31，4-3
　PolicyTable 許可の変更，5-10，5-12
　restart_debugging メソッド，3-21，4-5
　restrict_permission メソッド，4-5，5-9，5-10
　revoke_permission メソッド，4-5，5-18
　set_compiler_option，3-18
　set_output メソッド，3-23，4-4
　shortname メソッド，2-28，2-31，4-3

start_debugging メソッド，3-20，4-5
stop_debugging メソッド，4-5
パーミッションの変更，5-19
セキュリティの操作，2-10
定義済み，5-5
パーミッションの設定，5-6
DBMS_OUTPUT パッケージ，4-4
DbmsJava クラス，「DBMS_JAVA パッケージ」を参照
DbmsObjectInputStream クラス，2-34
DbmsObjectOutputStream クラス，2-34
DeadlockError 例外，2-42
debug
コンパイラ・オプション，2-16
DebugAgent クラス，3-17
DebugProxy クラス，3-17，3-19
delete_permission メソッド，4-5，5-18
delete メソッド，5-18
deployejb ツール，2-29
disable_permission メソッド，4-5，5-18
disable メソッド，5-18
dropjava ツール，2-24
dropjava メソッド，4-4

E

EJB
起動，3-2
構成，4-7，4-8
コンポーネント，1-19
状態，1-19
セキュリティ，5-2
定義済み，viii，1-15，1-19，2-5，3-2，3-6，3-7
マニュアル，1-20
例，3-8
enable_permission メソッド，4-5，5-18
enable メソッド，5-18
encoding
コンパイラ・オプション，2-16
EndOfCallRegistry クラス，6-12
registerCallback メソッド，6-12
endSession メソッド，2-5，2-43
Enterprise JavaBeans，「EJB」を参照
equals メソッド，5-5
exitCall メソッド，2-43

F

FilePermission，5-8，5-19，5-21，5-22，5-23
FilePermission パーミッション，6-5
forName メソッド，2-31

G

General Inter-Orb Protocol，「GIOP」を参照
get_compiler_option メソッド，2-17，4-4
getCallerClass メソッド，2-32
getClassLoader メソッド，2-32
GIOP
構成，4-6，4-7
プレゼンテーション，4-8
grant_permission メソッド，4-5，5-8，5-10
grant_policy_permission メソッド，4-5，5-12，5-20
grantPolicyPermission メソッド，5-13
grant メソッド，5-8
GUI，2-29

I

SHARED_POOL_SIZE パラメータ
エラー，6-10
定義済み，6-7
IIOP
SSL，4-8
構成，4-7
定義済み，3-7
implies メソッド，5-5
initjvm.sql，4-2，4-3，4-6
init メソッド，2-11
IOException，2-37

J

Java
Sun JDK との違い，2-3
アプリケーション，2-1，2-14
ロード，2-22
インタプリタ，2-2
開発環境，2-13
概要，viii，1-1，1-2
起動，2-2，3-3
機能，1-10

クライアント
CLASSPATH，4-9
セットアップ，4-8
クラス，1-2
クラスの解決，2-19
クラスのロード，2-6，3-2
結果のチェック，2-27
公開，2-7
コンパイル，2-14
実行権限，2-26
実行制御，2-6
ストアド・プロシージャ，「Java ストアド・プロシージャ」を参照
属性，1-2
データベース内の，1-1，1-11，2-1，2-3
ネイティブなコンパイル，6-2
パーミッション，4-5
プログラミング・モデル，viii
分散アプリケーション，viii，3-2
ポリモフィズム，1-5
マニュアル，ix，1-1，1-20
メソッド，1-2
リソース，1-2
Java 2
JDK 1.1 からの移行，2-7
セキュリティ，5-2
セキュリティの移行，2-10
Java Compatibility Kit，「JCK」を参照
Java Naming and Directory Interface，「JNDI」を参照
Java Remote メソッド Invocation，「RMI」を参照
Java ネイティブ・インタフェース，「JNI」を参照
JAVA$OPTIONS 表，2-16
JAVA_ADMIN ロール
パーミッションの付与，5-3，5-5，5-12，5-20
例，5-13
割当て済みのパーミッション，5-20
JAVA_DEPLOY ロール，6-5
JAVA_MAX_SESSIONSPACE_SIZE パラメータ，6-8
JAVA_POOL_SIZE パラメータ
エラー，6-10
最小値，4-3
定義済み，6-7，6-8
デフォルト，4-6
JAVA_SOFT_SESSIONSPACE_LIMIT パラメータ，6-8
JAVADEBUGPRIV ロール，5-23，5-24
JavaServer Pages，3-2
java.sql パッケージ，2-8
JAVASYSPRIV ロール，5-3，5-23
JAVAUSERPRIV ロール，5-3，5-22，5-23
java インタプリタ，2-2，2-6
Java 仮想マシン，「JVM」を参照
Java 言語仕様，「JLS」を参照
Java ストアド・プロシージャ，viii，2-5
起動，3-2
公開，2-29
構成，4-7
定義済み，1-15，1-16，3-3
マニュアル，1-20
.java ファイル，2-14，2-22，2-23
JCK，1-10
JDBC
2.0 のサポート，2-8
SQL との連係，3-16
SQL へのアクセス，1-16
Web 情報，ix
セキュリティ，5-2
定義済み，1-15，3-2，3-10
ドライバのタイプ，1-17，3-11
マニュアル，1-20
例，3-12
jdb デバッグ・ツール，3-17，3-21
JDeveloper
開発環境，1-20，3-15，4-11
デバッグ，3-17
JDK
Web アドレス，ix
要件，4-8
JLS
Web 情報，ix
仕様，1-10
JNDI
定義済み，3-7
名前参照，3-8
JNI サポート，3-10
JPublisher
マニュアル，1-20
JServerPermission，5-8，5-19，5-21，5-22，5-23
定義済み，5-20
JVM
Web 情報，ix
ガベージ・コレクション，1-11，1-12
機能，2-4
仕様，1-10
セキュリティ，4-5

定義済み，1-2，1-7
バイトコード，1-7
マルチスレッド，1-11

L

LimboError 例外，2-42
loadjava ツール，2-22 ～ 2-25
ZIP ファイルまたは JAR ファイルのロード，2-25
クラスのロード，2-22
実行権限，2-26，5-3
制限，2-24
ソースのコンパイル，2-15，6-11
メモリーの使用，6-7
例，3-4
loadjava メソッド，4-4
longname メソッド，2-27，2-31，4-3
lookupClass メソッド，2-33

M

main メソッド，2-6
MemStat
警告，6-18
使用，6-18
セキュリティのパーミッション，6-19
分析オプション，6-17
例，6-18，6-21
レポート，6-19
MemStat クラス，6-17
writeDumpAtEOC メソッド，6-18
writeDump メソッド，6-18
MemStat ユーティリティ
概要，6-16

N

ncomp
セキュリティ，6-5
ncomp ツール，6-4
実行，6-6
Net8
構成，4-6
NetPermission，5-8，5-19，5-21

O

ObjectInputStream クラス，2-34
ObjectOutputStream クラス，2-34
online
コンパイラ・オプション，2-16
Oracle8*i* JVM
インストール，4-1，4-2
構成，4-1
定義，ix，2-2
Oracle8*i* JVM Accelerator
インストールの要件，6-4
概要，6-2，6-3
ユーザー・アプリケーションの，6-4
OracleAgent クラス
restart メソッド，3-21
start メソッド，3-21
stop メソッド，3-21
oracle.jdbc2 パッケージ，2-8，2-9
OracleRuntime クラス
exitCall メソッド，2-43
getCallerClass メソッド，2-32
ORBClass プロパティ，2-13
ORBSingletonClass プロパティ，2-13
Orb クラス
init メソッド，2-11

P

PolicyTableManager クラス
delete メソッド，5-18
disable メソッド，5-18
enable メソッド，5-18
revoke メソッド，5-18
PolicyTablePermission，5-8，5-12，5-19，5-21，5-22
PolicyTable クラス
更新，5-5，5-14
ポリシーの指定，5-4
.properties ファイル，2-14，2-22，2-24
PropertyPermission，5-8，5-19，5-21，5-22，5-24
PUBLIC のパーミッション，5-22
publish ツール，2-29

R

ReflectPermission，5-8，5-19，5-21
registerCallback メソッド，6-12
reset_compiler_option メソッド，2-17，4-4
restart_debugging メソッド，3-21，4-5
restart メソッド，3-21
restrict_permission メソッド，4-5，5-9，5-10
restrict メソッド，5-9
revoke_permission メソッド，4-5，5-18
revoke メソッド，5-18
RMI
　サポート，3-10
RuntimePermission，5-8，5-19，5-21，5-22，5-23

S

SecurityManager クラス，5-5
SecurityPermission，5-8，5-19，5-21
SerializablePermission，5-8，5-19，5-21，5-23
ServerSocket クラス，2-40
.ser ファイル，2-14，2-22，2-24
set_compiler_option プロシージャ，3-18
set_compiler_option メソッド，2-17，4-4
set_output メソッド，3-23，4-4
SHARED_POOL_SIZE パラメータ
　最小値，4-3
　デフォルト，4-6
shortname メソッド，2-28，2-31，4-3
SocketPermission，5-8，5-19，5-21，5-22，5-23
SQL
　問合せ，3-2，3-10
SQLJ
　JDBC の使用，1-18
　PL/SQL との連係，3-16
　SQL へのアクセス，1-16
　型付けパラダイム，3-15
　実行，3-15
　定義済み，viii，1-15，1-18，3-2，3-10
　変換，3-15，3-16
　マニュアル，ix，1-20
　例，3-12，3-13
.sqlj ファイル，2-14，2-22，2-23
sqlj ユーティリティ，3-15
SSL，4-8
　構成，4-6
start_debugging メソッド，3-20，4-5
start メソッド，3-21
stop_debugging メソッド，3-20，4-5
stop メソッド，3-21
SYS
　セキュリティのパーミッション，5-19
　割当て済みのパーミッション，5-21
System クラス
　getProperty メソッド，3-22

T

ThreadDeathException，2-43

U

USER_ERRORS ビュー，2-15
USER_JAVA_POLICY ビュー，5-6，5-20
USER_OBJECTS ビュー，4-3，2-24，2-27

V

V$SGASTAT 表，6-9
Visigenic
　サポートされるバージョン，2-10

W

Web サイト，ix

あ

アプリケーション
　開発，2-3
　起動，3-3，3-22
　コンパイル，2-14
　実行権限，2-26
　実行制御，2-6
　スレッド，2-42
　セッション内での実行，2-3
　分散，1-19

い

インストレーション，4-1，4-2
インターネット・ニュースグループ，ix
インタフェース
　定義済み，1-5
　ユーザー，2-29

え

エラー
　コンパイル，2-15

お

オブジェクト
　完全名から短縮名への変換，2-27
　シリアル化，2-34
　スキーマ，2-13
　存続期間，2-37
　短縮名，2-27
オペレーティング・システム
　リソース，2-35，2-36
　　コール間，2-38
　　存続期間，2-36
　　パーミッション，2-36
　　パフォーマンス，6-7

か

ガベージ・コレクション，1-11，1-12，2-5
　間違った使用，2-37
　目的，2-37
　リソースの管理，2-35

き

パーミッションの付与，5-5

く

クライアント
　セットアップ，4-8
クラス
　解決，3-2
　階層構造，1-4
　継承，1-4，1-5
　公開，2-2，2-29，3-2
　参照の解決，2-19
　実行，2-2
　属性，1-2，1-4
　定義，1-2
　動的ロード，1-14
　名前，2-30
　保護，5-25
　メソッド，1-2，1-4
　有効とマークする，2-19
　ロード，2-2，2-6，2-22，3-2
クラス・インタフェース
　forName メソッド，2-31
クラス・スキーマ・オブジェクト，2-13，2-19，2-22，2-23

け

継承，1-4，1-5
権限
　データベース，5-2

こ

公開，2-7，2-14，2-29，3-2
　例，3-5
構成，4-1
　Oracle8i JVM，4-5 ～ 4-8
　パフォーマンス，6-7
コード
　ネイティブ・コンパイル，6-2
コール
　コール間にまたがるリソースの管理，2-38
　静的フィールド，2-5
　定義，2-2
コール終了時の移行，6-11
コール仕様，3-5
コンパイル，2-14
　エラー・メッセージ，2-15
　オプション，2-16
　オプションの設定，3-18
　メモリーの問題，6-10
　ランタイム，2-15
コンポーネント，1-19

さ

サーブレット，3-2

し

実行権限，2-26
実行者権限，2-26
出力
　リダイレクト，3-23
シリアル化，2-34

す

スキーマ・オブジェクト
　使用，2-13
　定義済み，2-22
　名前，2-30
スレッド
　Oracle8i JVM での使用，2-36
　アプリケーション，2-42
　存続期間，2-42
　モデル，1-11，2-41

せ

整合性，5-2
静的変数，2-5
　コール終了時の移行，6-11
セキュリティ，5-1 ～ 5-25
　CORBA，5-2
　EJB，5-2
　Java 2，5-3
　JDBC，5-2
　JVM，4-5
　オペレーティング・システム・リソース，2-36
　参考文献，5-4
　ネットワーク，5-2
セッション
　Java 実行におけるロール，2-3
　JVM による調整，2-4
　存続期間，2-5
　タイムアウト，2-5
　定義，2-2
　フットプリント，1-13
セッション・シェル・ツール，3-9
接続
　構成，4-6
　セキュリティ，5-2

そ

ソース・スキーマ・オブジェクト，2-13，2-22，2-23
属性
　タイプ，1-4
　定義，1-2
ソケット
　コール間，2-36，2-40
　存続期間，2-37，2-40
　定義済み，2-40
ソケット・クラス，2-40

た

タイムアウト，2-5

て

定義者権限，2-26
データの機密保護，5-2
データベース
　権限，5-2
　構成，4-6
デッドロック，2-42
デバッグ，4-5，5-24
　Java アプリケーション，3-17
　OracleAgent クラスの使用，3-21
　エージェント，3-17，3-20
　コンパイラ・オプションの設定，3-18
　デバッガの接続，3-21
　デバッグ・エージェントの起動，3-20
　必要なパーミッション，5-24
　プロキシの起動，3-19

と

トリガー
　Java ストアド・プロシージャの使用，3-3

に

認証，5-2

ね

ネイティブ・コンパイラ，6-2
ネイティブ・コンパイル，1-13，6-2
　静的，6-3
ネーム・サービス，3-7
　CosNaming，3-7
　JNDI，3-7
ネットワーキング
　構成，4-6

は

バージョン
　Visigenic，2-10
　検索，3-22
パーミッション，4-5，5-2 ～ 5-24
　FilePermission，6-5
　JAVA_ADMIN ロール，5-20
　JAVA_DEPLOY ロール，6-5
　JAVADEBUGPRIV ロール，5-23
　JAVASYSPRIV ロール，5-23
　JAVAUSERPRIV ロール，5-22
　PUBLIC，5-22
　SYS パーミッション，5-21
　管理，5-12
　権限付与，5-5，5-8，5-10
　削除，5-18
　作成，5-14
　使用可能にする，5-17
　使用禁止にする，5-17
　制限，5-5，5-9，5-10
　タイプ，5-7，5-19
　ポリシーの指定，5-4
　ポリシーの付与，5-12
　ロールにまとめる，5-23
　割当て，5-4，5-6
パーミッションのクラス，5-7，5-13，5-14，5-19
バイトコード
　検証，2-21
　定義済み，1-7
パッケージ
　DBMS_JAVA，4-3
　oracle.jdbc2，2-8
　保護，5-25
パフォーマンス，1-13，6-1 ～ 6-16

ふ

ファイナライザ，2-38
ファイル，2-36
　コール間，2-38
　存続期間，2-37
フットプリント，1-13，2-4
プレゼンテーション
　RMI との互換性，3-10
プレゼンテーション層
　GIOP，4-8
プロパティ
　ORBClass，2-13
　ORBSingletonClass，2-13
分散オブジェクト，1-19，3-6

へ

変数
　静的，2-5

ほ

表
　管理，5-12
　パーミッションの設定，5-6
　表示，5-6
　変更，5-6
ポリモフィズム，1-5

ま

マルチスレッド，1-11

め

メソッド，1-2，1-4
メモリー
　java プール，6-9
　コール，2-5
　コール間，2-38
　セッション，2-5，6-12
　存続期間，2-36，2-37
　パフォーマンス構成，6-7
　不足，6-10
　マネージャ，2-14

リーク，2-38
レポート割当て，6-16
メモリー・プロファイリング・ユーティリティ，
MemStat を参照

ゆ

ユーザー・インタフェース，2-29

よ

要件
JDK のバージョン，4-8

り

リソース・スキーマ・オブジェクト，2-13，2-22，2-24
リゾルバ，2-19 ～ 2-22
存在しない参照を無視，2-20，2-21
定義済み，2-14，2-20，2-31，3-2
デフォルト，2-20
例，3-4

れ

例外
ClassNotFoundException，2-31
DeadlockError，2-42
IOException，2-37
LimboError，2-42
ThreadDeathException，2-43

ろ

ロード，2-22 ～ 2-28
JAR ファイルまたは ZIP ファイル，2-25
クラス，1-14，2-6，2-14
クラスの再ロード，2-26
結果のチェック，2-24，2-27
コンパイル・オプション，2-15
実行権限の付与，2-26
制限，2-24
必要な権限と許可，2-25
ロギング，2-15