Oracle9*i*

JDBC 開発者ガイドおよびリファレンス

リリース 1（9.0.1）

2001 年 10 月

部品番号 : J04187-01

ORACLE®

Oracle9*i* JDBC 開発者ガイドおよびリファレンス , リリース 1（9.0.1）

部品番号 : J04187-01

原本名 ：Oracle9*i* JDBC Developer's Guide and Reference, Release 1 (9.0.1)

原本部品番号 ：A90211-01

原本著者 ：Mike Sanko, Brian Wright, Thomas Pfaeffle

原本協力者 ：Magdi Morsi, Ron Peterson, Ekkehard Rohwedder, Ashok Shivarudraiah, Catherine Wong, Ed Shirk, Sunil Kunisetty, Joyce Yang, Mehul Bastawala, Luxi Chidambaran, Srinath Krishnaswamy, Rajkumar Irudayaraj, Scott Urman, Jerry Schwarz, Steve Ding, Soulaiman Htite, Douglas Surber, Anthony Lai, Paul Lo, Prabha Krishna, Ellen Barnes, Susan Kraft, Sheryl Maring, Angie Long

目次

2 スタート・ガイド

3 基本機能

4 JDBC 2.0 サポートの概要

5 Oracle 拡張機能の概要

6 Oracle データへのアクセスと操作

7 LOB と BFILE の操作

8 Oracle オブジェクト型の操作

9 Oracle オブジェクト参照の操作

10 Oracle コレクションの操作

11 結果セット拡張

12 パフォーマンス拡張要素

13 文キャッシュ

14 接続プーリングとキャッシュ

15 分散トランザクション

16 JDBC OCI 拡張機能

17 Java Transaction API

18 上級トピック

19 コーディングのヒントおよびトラブルシューティング

20 サンプル・アプリケーション

21 リファレンス情報

A 行セット

B JDBC エラー・メッセージ

索引

はじめに

ここでは、このマニュアルの対象読者、構成および表記規則について説明します。Oracle の関連文書の一覧も示します。

対象読者

このマニュアルは、JDBC プログラミングに関心があるすべての読者を対象にしていますが、少なくとも、次の知識は必要です。

- Java
- SQL
- Oracle PL/SQL
- Oracle データベース

構成

このマニュアルは、次に示す章と付録で構成されています。

- 第 1 章「概要」 — Oracle による JDBC の実装方法および Oracle JDBC ドライバ・アーキテクチャの概要を説明します。
- 第 2 章「スタート・ガイド」 — Oracle JDBC ドライバおよびその使用例を紹介します。また、実行したインストールと構成をテストする基本的な方法も説明します。
- 第 3 章「基本機能」 —任意の JDBC アプリケーションを作成する際の基本的な処理を説明します。また、Oracle JDBC ドライバがサポートする Java と JDBC の基本機能についても説明します。
- 第 4 章「JDBC 2.0 サポートの概要」 — JDBC 2.0 の機能の概要を示し、JDK 1.2.x 環境と JDK 1.1.x 環境での、この機能のサポート状況の違いについて説明します。
- 第 5 章「Oracle 拡張機能の概要」 — Oracle が提供する JDBC 拡張要素クラスの概要について説明します。
- 第 6 章「Oracle データへのアクセスと操作」 — Java 形式ではなく、Oracle データ型形式を使用してデータにアクセスする方法を説明します。
- 第 7 章「LOB と BFILE の操作」 — LOB および LOB データへのアクセスおよび操作を行う、JDBC 標準に対する Oracle 拡張機能について説明します。
- 第 8 章「Oracle オブジェクト型の操作」—標準 JDBC または Oracle 拡張機能を使用して、Oracle オブジェクト型を Java クラスにマップする方法を説明します。
- 第 9 章「Oracle オブジェクト参照の操作」 —オブジェクト参照へのアクセスおよび操作を行う、標準 JDBC に対する Oracle 拡張機能について説明します。
- 第 10 章「Oracle コレクションの操作」—配列および配列のデータへのアクセスおよび操作を行う、標準 JDBC に対する Oracle 拡張機能について説明します。
- 第 11 章「結果セット拡張」 —スクロール可能な結果セット、更新可能な結果セットなど、JDBC 2.0 の結果セット拡張機能について説明します。JDK 1.1.x でのサポートについても説明します。

- 第 12 章「パフォーマンス拡張要素」－アプリケーションのパフォーマンスを向上させる、JDBC 標準に対する Oracle 拡張機能について説明します。
- 第 13 章「文キャッシュ」－キャッシュのための Oracle 拡張機能文について説明します。
- 第 14 章「接続プーリングとキャッシュ」－ JDBC 2.0 データ・ソース（および JNDI の使用方法）、接続プーリング機能（接続キャッシュ実装の枠組み）および Oracle が提供する接続キャッシュ実装のサンプルについて説明します。
- 第 15 章「分散トランザクション」－分散トランザクション（グローバル・トランザクション）および標準 XA 機能について説明します。（分散トランザクションとは、協調してコミットされる必要があるトランザクションの集合で、多くの場合、複数のデータベースを対象にします。）
- 第 16 章「JDBC OCI 拡張機能」－ OCI ドライバ固有の機能について説明します。
- 第 17 章「Java Transaction API」－トランザクション内の複数のデータベースに対するすべての変更を処理するための、JTA グローバル・トランザクション内での JDBC 接続の使用方法について説明します。
- 第 18 章「上級トピック」－ NLS、アプレット、サーバー側ドライバ、および埋込み SQL92 構文の使用方法など、JDBC の上級トピックについて説明します。
- 第 19 章「コーディングのヒントおよびトラブルシューティング」－ JDBC アプリケーションのトラブルシューティングを行う際の、コード記述上のヒントおよび一般的な指針を説明します。
- 第 20 章「サンプル・アプリケーション」－ JDBC の高度な機能と Oracle 拡張機能の特徴を示すサンプル・アプリケーションを紹介します。
- 第 21 章「リファレンス情報」－詳細な JDBC 参照情報を提供します。
- 付録 A「行セット」－ JDBC およびキャッシュされた行セットについて説明します。
- 付録 B「JDBC エラー・メッセージ」－ JDBC エラー・メッセージと、対応する ORA エラー番号の一覧を示します。

表記規則

このマニュアルでは、Solaris の構文表記法を使用していますが、Windows NT 環境でのファイル名とディレクトリ名も、特に明示されていない限り同一です。

[ORACLE_HOME] は、Oracle のホーム・ディレクトリのフル・パスを意味します。

たとえば、特に明示されていない限り、各行の最後では改行を入力することが暗黙の規則になっています。つまり、入力行の最後で [Enter] キーを押す必要があります。

次の規則も、このマニュアルで使用します。

表記規則	意味
. . .	例題中で使用される縦の省略記号は、例題に直接関係しない情報が省略されていることを示します。
...	文またはコマンドで使用される横の省略記号は、例題に直接関係しない文またはコマンドの一部が省略されていることを示します。
< >	山カッコは、その中にユーザーが名前を入力することを示します。
[ ]	大カッコは、その中の 1 つを選択することも、何も選択しないことも可能なオプション句であることを示します。

関連文書

この項では、その他の関連文書を示します。

Oracle Java Platform グループが提供する、次のドキュメントを参照してください。

- 『Oracle9*i* Java Developer's Guide』

  Oracle9*i* の Java の基本概念を紹介し、サーバー側の構成と機能に関する一般情報を提供します。特定の製品（JDBC、SQLJ、EJB など）ではなく、Oracle Java プラットフォーム全体に関する情報が含まれています。

- 『Oracle9*i* JPublisher ユーザーズ・ガイド』

  JPublisher ユーティリティを使用して、オブジェクト型および他のユーザー定義型を Java クラスに変換する方法を説明します。オブジェクト型、VARRAY 型、NESTED TABLE 型、またはオブジェクト参照型を使用する SQLJ または JDBC アプリケーションを開発する場合、これらの型にマップするカスタム Java クラスを JPublisher で生成できます。

- 『Oracle9*i* SQLJ 開発者ガイドおよびリファレンス』

  SQLJ を使用して静的 SQL 操作を直接 Java コードに埋め込む方法を説明します。SQLJ 言語の構文および SQLJ トランスレータのオプションと機能についても説明します。標準 SQLJ 機能と Oracle 独自の SQLJ 機能の両方の説明を含みます。

- 『Oracle9*i* Java Stored Procedures Developer's Guide』

  Java ストアド・プロシージャを説明します。Java ストアド・プロシージャは、Oracle9*i* サーバーで直接実行するプログラムです。ストアド・プロシージャ（ファンクション、プロシージャ、データベース・トリガーおよび SQL メソッド）を使用して、ビジネス・ロジックをサーバー・レベルで実装することにより、Java 開発者はアプリケーションのパフォーマンス、拡張性およびセキュリティを改善できます。

- 『Oracle9*i* Enterprise JavaBeans Developer's Guide and Reference』

  Enterprise JavaBeans および CORBA 仕様に対する Oracle 拡張機能について説明します。

- 『Oracle9*i* Java Tools Reference』

  Oracle JVM ツールおよび関連するオプションについて説明します。

Oracle Server Technologies グループが提供する、次のドキュメントも参照してください。

- 『Oracle9*i* Net Services 管理者ガイド』

  Oracle8 Connection Manager および Oracle Net ネットワーク管理に関する一般情報が含まれています。

- 『Oracle9*i* グローバリゼーション・サポート・ガイド』

  NLS 環境変数、キャラクタ・セットおよび地域とロケール設定の情報が含まれています。さらに、このマニュアルには、NLS 共通の問題、標準例、および NLS に関して OCI や SQL プログラマが考慮する必要がある点も含まれます。

- 『Oracle Advanced Security 管理者ガイド』

  Oracle Advanced Security（旧バージョンの ANO または ASO）の機能を説明します。

- 『Oracle9*i* アプリケーション開発者ガイド－基礎編』

  Oracle9*i* データベースを使用するとき、およびデータベースにアクセスするアプリケーションを作成するときの、基本的な設計概念とプログラミング機能を説明します。

- 『Oracle9*i* アプリケーション開発者ガイド－ラージ・オブジェクト』

  Oracle9*i* のデータベース・ラージ・オブジェクト（LOB）に関する一般的な機能を説明します。

- 『Oracle9*i* アプリケーション開発者ガイド－オブジェクト・リレーショナル機能』

  構造化オブジェクトおよびその他の Oracle9*i* のオブジェクト・リレーショナル・データベース機能に関する一般情報が含まれています。

- 『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』

  Oracle9*i* サーバーの一部として使用可能な PL/SQL パッケージについて説明します。これらのパッケージは、JDBC アプリケーションからのコールに役立ちます。

- 『PL/SQL ユーザーズ・ガイドおよびリファレンス』

  SQL に対する Oracle の手続き型言語拡張要素である PL/SQL の概念と機能について説明します。

- 『Oracle9*i* SQL リファレンス』

  Oracle データベースの情報管理に使用する SQL コマンドの内容および構文と、機能について詳しく説明します。

- 『Oracle9*i* データベース・リファレンス』

  Oracle9*i* サーバーに関する一般的な参照情報が含まれています。

- 『Oracle9*i* データベース・エラー・メッセージ』

  Oracle9*i* サーバーから渡される可能性のあるエラー・メッセージに関する情報が含まれています。

次の Oracle グループが提供する資料も参照してください。

- 「Oracle JDeveloper オンライン・ヘルプ」

  Oracle JDeveloper による Oracle JDBC のサポートに関する情報が含まれています。

- 『Oracle Call Interface プログラマーズ・ガイド』

  OCI を使用したアプリケーションの開発方法について説明します。

SQLJ 標準機能および構文の詳細は、ANSI NCITS 331.2-2000 の次のマニュアルを参照してください。

- 『Information Technology - SQLJ - Part 2: SQL Types using the JavaTM Programming Language』

このマニュアルは、次の ANSI Web サイトから入手できます。

`http://www.ansi.org/`

1

概要

この章では、JDBC の Oracle 実装の概要を説明します。次の項目が含まれます。

概要

この項では、Oracle JDBC の概要を説明します。SQLJ との比較も行います。

JDBC とは

JDBC（Java Database Connectivity）は、Java からリレーショナル・データベースに接続するための標準 Java インタフェースです。Sun 社の定義による JDBC 標準は、各プロバイダが独自の JDBC ドライバで標準を実装および拡張することを可能にします。

JDBC は、X/Open SQL コール・レベル・インタフェースに基づき、SQL92 エントリ・レベル標準に準拠しています。

Oracle ドライバは、標準 JDBC API をサポートする他、Oracle 固有のデータ型をサポートし、パフォーマンスを向上させる拡張要素を備えています。

JDBC と SQLJ

次の項目が含まれます。

- 静的 SQL における、SQLJ の JDBC に対する優位点
- JDBC と SQLJ を使用する際の一般的な指針

クライアント側 C コードの Oracle コール・インタフェース（OCI）レイヤーをよく理解している開発者にとって、JDBC は、OCI が C や C++ プログラマに提供しているような強力で柔軟性のある機能を、Java プログラマに提供していると理解できます。OCI の場合と同様、JDBC を使用して、たとえば列の数と種類が実行時まで不明な表の問合せおよび更新を実行できます。この機能は、動的 SQL と呼ばれます。このため、JDBC は Java プログラムで動的 SQL 文を使用する方法の 1 つです。コール・プログラムは、JDBC を使用して実行時に SQL 文を構成できます。JDBC プログラムは、他のすべての Java プログラムと同様にコンパイルおよび実行されます。SQL 文の分析もチェックも行われません。SQL コード内で発生するエラーはすべて、実行時エラーを生成します。JDBC は、動的 SQL 用の API として設計されています。

ただし、多くのアプリケーションでは、使用する SQL 文が固定または静的であるため、SQL 文を動的に組み立てる必要はありません。この場合、SQLJ を使用して静的 SQL を Java プログラムに埋め込むことが可能です。静的 SQL では、すべての SQL 文は完結しています（または、Java プログラムの本文内で明白です）。これは、列名、表の列数、表名などのデータベース・オブジェクトの詳細が実行時以前に明らかであることを意味します。SQLJ は、プリコンパイル時にエラー・チェックを許可するため、これらのアプリケーションでは利点となります。

SQLJ プログラムのプリコンパイル処理では、埋込み SQL の構文チェック、Java と SQL 間で交換されたデータの型の互換性と適切な型変換を保証するためのタイプ・チェック、および SQL 構文とデータベース・スキーマの一致を保証するためのスキーマ・チェックが行われます。プリコンパイルの結果、Java ソース・コードとともに SQL ランタイム・コードが生成されます。そして、SQL ランタイム・コードから JDBC をコールできます。生成された Java コードは、他のすべての Java プログラムと同様にコンパイルおよび実行が可能です。

SQLJ は、プログラムの記述時に明白な静的 SQL 操作を直接サポートしますが、JDBC 経由で動的 SQL と相互運用することも可能です。SQLJ を使用すると、JDBC オブジェクトを、動的 SQL 操作に必要な時点で作成できます。このようにして、SQLJ と JDBC は 1 つのプログラム内で共存可能です。JDBC 接続と SQLJ 接続のコンテキスト間、および JDBC 結果セットと SQLJ イテレータ間の有用な変換がサポートされます。詳細は、『Oracle9*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。

SQLJ と JDBC の構文と方法は実行時の構成に依存しないため、クライアント側やデータベース側または中間層での実装が可能です。

静的 SQL における、SQLJ の JDBC に対する優位点

JDBC が Java からリレーショナル・データベースへの完全な動的 SQL インタフェースを提供するのに対し、SQLJ は静的 SQL で補完的役割を果たします。

静的 SQL 文は、JDBC プログラムでも使用可能ですが、SQLJ ではより便利な使用方法があります。静的 SQL 文で JDBC ではなく SQLJ を使用することにより、次のような利点があります。

- SQLJ の構文は短いため、SQLJ ソース・プログラムは等価な JDBC プログラムよりもサイズが小さくなります。
- SQLJ は、接続（および接続がアクセスする SQL エンティティの集合）、問合せ出力、戻りパラメータに対して、強い型指定を行います。
- SQLJ は、データベース接続を使用して、コンパイル時に静的 SQL コードのタイプ・チェックを実行できます。JDBC は、完全な動的 API であるため、実行時までまったくタイプ・チェックを行いません。
- SQLJ プログラムは、Java バインド式を SQL 文中に直接埋め込むことが可能です。JDBC では、バインド変数ごとにコール文を取出しまたは設定（あるいはその両方）する必要があります。
- SQLJ は、SQL ストアド・プロシージャおよびファンクションをコールする際のルールが簡素化されています。

JDBC と SQLJ を使用する際の一般的な指針

SQLJ は、次の場合に効果的です。

- 実行時ではなくコンパイル時にプログラムのエラーをチェックする場合。
- 別のデータベースにも配置可能なアプリケーションを記述する場合。SQLJ を使用すると、実行時に、対象のデータベースに合せて静的 SQL をカスタマイズできます。
- コンパイル済みの SQL を含むデータベースで作業する場合。JDBC プログラムでは SQL 文をコンパイルできないために、SQLJ を使用する場合があります。

JDBC は、次の場合に効果的です。

- プログラムで動的 SQL を使用する場合。たとえば、問合せをリアルタイムで構築したり、対話式問合せコンポーネントを使用するプログラムでは、JDBC を使用します。
- 配置時や開発時に SQLJ レイヤーを使用しない場合。たとえば、低速接続でのダウンロード時間を最小にするため、SQLJ ランタイム・ライブラリをダウンロードせずに JDBC Thin ドライバのみをダウンロードする場合があります。

注意： 同じソースに SQLJ のコードと JDBC のコードを混在させることができます。詳細は『Oracle9*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。

Oracle JDBC ドライバの概要

この項では、Oracle JDBC ドライバと、その基本アーキテクチャおよび使用例について説明します。ここでは、すべての JDBC ドライバのコア機能を取り上げます。ただし、OCI ドライバに固有の特殊な機能に関する詳細は、第 16 章「JDBC OCI 拡張機能」を参照してください。

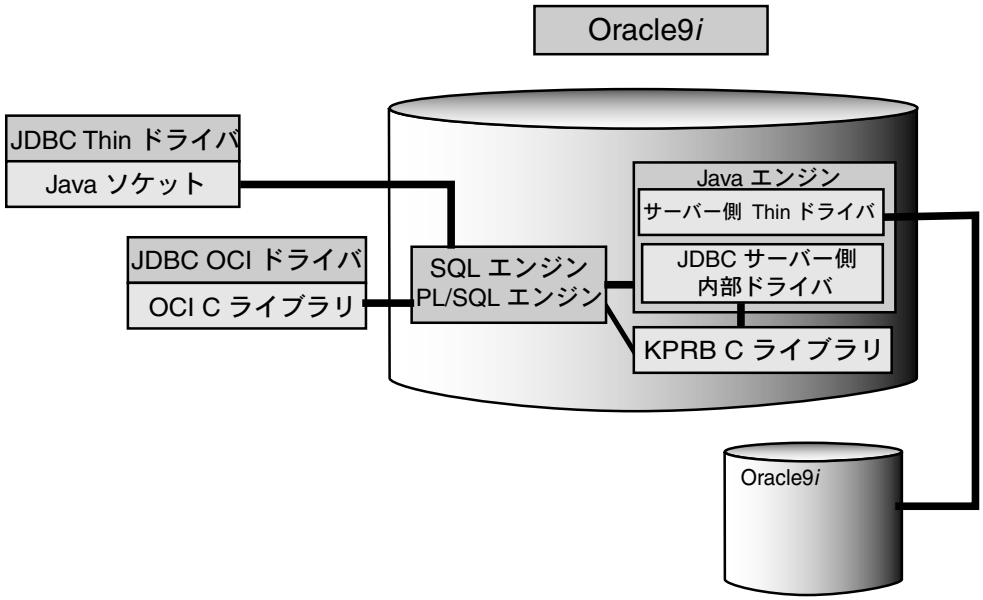
次の JDBC ドライバがあります。

- **Thin ドライバ**: クライアント側で使用する 100% Java ドライバで、Oracle のインストールは必要ありません。特に、アプレットで使用されます。
- **OCI ドライバ**: Oracle クライアントのインストールのクライアント側で使用されます。
- **サーバー側 Thin ドライバ**: 機能的にはクライアント側 Thin ドライバと同じですが、Oracle Server の内部で実行し、リモート・サーバーにアクセスする必要があるコード用です。中間層からデータベースにアクセスする場合も含まれます。
- **サーバー側内部ドライバ**: ターゲット・サーバーの内部（つまり、アクセスする必要がある Oracle Server の内部）で実行するコード用です。

図 1-1 に、JDBC Thin、OCI およびサーバー側内部ドライバ用のドライバとデータベースのアーキテクチャを示します。

この項の後半で、Oracle ドライバの共通機能を説明し、次に、各ドライバを個別に説明します。最後に、アプリケーションに適したドライバを選択するための考慮事項について説明します。

図 1-1 ドライバ / データベース・アーキテクチャ



Oracle JDBC ドライバの共通機能

サーバー側およびクライアント側の Oracle JDBC ドライバは、同じ基本機能を提供します。どのドライバも次の標準規格や機能をサポートします。

- JDK 1.2.x / JDBC 2.0 または JDK 1.1.x / JDBC 1.22 (JDBC 2.0 機能に対する Oracle 拡張機能も含みます)

  これら 2 つの実装は、異なるクラス・ファイルの集合を使用します。

- 共通の構文と API
- 共通の Oracle 拡張機能
- マルチスレッド・アプリケーションの完全なサポート

Oracle JDBC ドライバは、Sun 社の標準の `java.sql` インタフェースを実装します。`oracle.jdbc` パッケージを使用することにより、Sun の機能に加えて Oracle の機能へのアクセスが可能になります。このパッケージは、Oracle9*i* で廃止された `oracle.jdbc.driver` パッケージと同じです。

表 1-1 に、クライアント側ドライバの機能を比較します。

表 1-1 JDBC クライアント側ドライバの機能の比較

ドライバ	型	サイズ	プロトコル	100% Java	使用目的	必要な外部ライブラリ	プラットフォーム依存	パフォーマンス	機能の充実度
Thin	IV	小さい	TTC	あり	アプレットおよびアプリケーション	なし	なし	よい	よい
OCI	II	大きい	TTC	なし	アプリケーション	あり	あり	最高	最高

注意： オブジェクト、配列、LOB など、JDBC 2.0 のほとんどの機能は、Oracle 拡張機能により、JDK 1.1.x 環境で使用可能です。

JDBC Thin ドライバ

Oracle JDBC Thin ドライバは、100% pure Java の Type IV ドライバです。このドライバは Oracle JDBC アプレット用ですが、アプリケーションでも使用できます。このドライバは、すべてが Java で記述されているためにプラットフォームに依存しません。クライアント側に、追加の Oracle ソフトウェアは必要ありません。Thin ドライバは TTC を使用してサーバーと対話します。TTC は、Oracle リレーショナル・データベース管理システム（RDBMS）にアクセスするために Oracle が開発したプロトコルです。

アプレットの場合、実行する Java アプレットとともに、ブラウザにダウンロードできます。HTTP プロトコルはステートレスですが、Thin ドライバはステートレスではありません。アプレットおよび Thin ドライバをダウンロードする、最初の HTTP 要求はステートレスです。Thin ドライバがデータベース接続を確立すると、ブラウザとデータベース間の通信はステートフル、かつ 2 層構造になります。

JDBC Thin ドライバを使用すると、Java ソケットの上で Oracle Net と TTC（OCI が使用する接続プロトコル）をエミュレートする TCP/IP の実装が提供されるので、データベースに直接接続できます。これらのプロトコルは、サーバーで実装されている機能の軽量版です。Oracle Net プロトコルは、TCP/IP のみで動作します。

このドライバでは、TCP/IP プロトコルのみがサポートされます。また、データベース・サーバーの TCP/IP ソケットをリスニングする TNS リスナーが必要です。

注意： JDBC Thin ドライバをアプレットとともに使用する場合、クライアントのブラウザが Java ソケットをサポートしている必要があります。

Oracle Server の内部または中間層で Thin ドライバを使用する方法については、「JDBC サーバー側 Thin ドライバ」で説明します。

JDBC OCI ドライバ

JDBC OCI ドライバは、クライアント / サーバー Java アプリケーション用の Type II ドライバです。このドライバには、Oracle クライアントのインストールが必要です。Oracle プラットフォーム固有なので、アプレットには適しません。

注意： Oracle9*i* では、すべてのデータベース・バージョンで OCI ドライバを使用します。以前のリリースで使用されていた OCI8 および OCI7 ドライバは使用しません。OCI8 および OCI7 ドライバは Oracle9*i* では廃止されましたが、下位互換性を保持するためにサポートされています。

JDBC OCI ドライバは、OCI 接続プール機能を提供します。この機能は JDBC クライアントまたは JDBC ストアド・プロシージャの一部です。OCI ドライバの接続プーリングでは、標準の接続プーリングと比べて必要な物理接続が少なくて済み、共通のインタフェースも提供されます。また、接続プールの属性を動的に構成することができます。OCI ドライバ接続プーリングの詳細は、16-2 ページの「OCI ドライバ接続プーリング」を参照してください。

OCI ドライバは、最高位の互換性で Oracle7、Oracle8/8*i* および Oracle9*i* をサポートします。また、このドライバは、IPC、名前付きパイプ、TCP/IP および IPX/SPX を含む、インストールされたすべての Oracle Net アダプタをサポートします。

OCI ドライバは、Java と C の組合せで作成されています。C のエントリ・ポイントをコールするシステム固有なメソッドを使用して、JDBC の起動を Oracle コール・インタフェース（OCI）に変換します。これらのコールは、Oracle Net 経由で Oracle データベース・サーバーへ送信されます。OCI ドライバは、Oracle が開発した TTC プロトコルを使用してサーバーと対話します。

OCI ドライバは、インストールされたクライアント・マシンにある OCI ライブラリ、C エントリ・ポイント、Oracle Net、CORE ライブラリおよびその他の必要なファイルを使用します。

Oracle Call Interface（OCI）は、Application Program Interface（API）の 1 つです。OCI により、第三世代言語のネイティブ・プロシージャやファンクション・コールを使用して、Oracle データベース・サーバーにアクセスし、SQL 文の実行のすべての段階を制御するアプリケーションを作成できます。OCI ドライバは、多数のユーザーを安定してサポートできるスケーラブルなマルチスレッド・アプリケーションを作成するめに設計されています。

Oracle9*i* JDBC OCI ドライバには、次のような機能があります。

- OCI の使用
- 接続プーリング
- OCI の最適化フェッチ
- プリフェッチ
- 最高速の LOB アクセス
- クライアント側オブジェクト・キャッシュ
- 透過的アプリケーション・フェイルオーバー（TAF）
- 中間層認証

JDBC サーバー側 Thin ドライバ

Oracle JDBC サーバー側 Thin ドライバは、クライアント側 Thin ドライバと同じ機能を提供しますが、Oracle データベースの内部で実行され、リモート・データベースにアクセスします。

このドライバは、特に、次の 2 つの場合に役立ちます。

- 中間層として動作する Oracle Server からリモート Oracle Server にアクセスする場合
- より一般的には、Java ストアド・プロシージャまたは Enterprise JavaBean などの Oracle Server の内部から別の Oracle Server にアクセスする場合

Thin ドライバをクライアント・アプリケーションから使用する場合とサーバーの内部から使用する場合では、コードの違いは生じません。

注意： 文 `cancel()` および `setQueryTimeout()` メソッドは、サーバー側 Thin ドライバではサポートされません。

サーバー側 Thin ドライバの許可 Thin ドライバは、接続用のソケットをオープンします。Oracle Server では Java セキュリティ・モデルを施行しています。これは `SocketPermission` オブジェクトについてチェックが実行されることを意味します。

JDBC サーバー側 Thin ドライバを使用するには、接続するユーザーに適切な許可を付与する必要があります。次は、ユーザー `SCOTT` に許可を付与する方法を示します。

```
create role jdbcthin;
call dbms_java.grant_permission('JDBCTHIN',
'java.net.SocketPermission',
'*', 'connect' );
grant jdbcthin to scott;
```

`grant_permission` コールの `JDBCTHIN` は大文字で指定する必要があるので注意してください。`'*.'` はパターンです。特定のマシンまたはポートのみに接続するように許可を限定することができます。`java.net.SocketPermission` クラスの詳細は、Javadoc を参照してください。また、Oracle Server 内での Java セキュリティの詳細は、『Oracle9*i* Java Developer's Guide』を参照してください。

JDBC サーバー側内部ドライバ

Oracle JDBC サーバー側内部ドライバは、Java ストアド・プロシージャまたは Enterprise JavaBean など、Oracle データベースの内部で実行され、同じデータベースにアクセスする必要がある、あらゆる Java コードをサポートします。このドライバにより、Java Virtual Machine（JVM）は SQL エンジンに直接接続できます。

サーバー側内部ドライバ、JVM、データベース、KPRB（サーバー側）C ライブラリ、および SQL エンジンはすべて、同一アドレス空間で実行されます。そのため、ネットワーク・ラウンドトリップの発行は不要です。プログラムは、ファンクション・コールを使用して SQL エンジンにアクセスします。

サーバー側内部ドライバは、クライアント側ドライバとの一貫性を保ち、同一の機能と拡張要素をサポートします。サーバー側内部ドライバの詳細は、18-27 ページの「サーバー上の JDBC: サーバー側内部ドライバ」を参照してください。

注意： サーバー側内部ドライバがサポートするのは、JDK 1.2.x のみです。

適切なドライバの選択

アプリケーションまたはアプレットに使用する JDBC ドライバを選択するときには、次の点を考慮します。

- アプレットを作成する場合は、JDBC Thin ドライバを使用してください。JDBC OCI ベースのドライバ・クラスは、システム固有な（C 言語の）メソッドをコールするため、Web ブラウザでは動作しません。
- Oracle8*i* 以前のリリースで最大限の移植性とパフォーマンスを実現するには、JDBC Thin ドライバを使用します。JDBC Thin ドライバを使用すると、アプリケーションとアプレットのどちらからでも Oracle Server に接続できます。
- パフォーマンスを最も重視する Oracle クライアント環境のクライアント・アプリケーションを作成する場合、JDBC OCI ドライバを選択します。
- 中間層として動作する Oracle Server でコードを実行する場合、サーバー側 Thin ドライバを使用します。
- ターゲット Oracle Server の内部でコードを実行する場合、JDBC サーバー側内部ドライバを使用し、サーバーにアクセスします。（サーバー側 Thin ドライバを使用して、リモート・サーバーにアクセスすることもできます。）
- アプリケーションにおいてパフォーマンスが重視され、Oracle Server に最大限の拡張性が必要である場合、または TAF などの拡張可用性機能や中間層認証などの拡張プロキシ機能が必要な場合には、OCI ドライバを使用します。

アプリケーションおよびアプレット機能の概要

この項では、JDBC アプリケーションとアプレットの基本機能を比較し、アプリケーションおよびアプレットのプログラマが使用できる Oracle 拡張機能を紹介します。

アプリケーションの基本

クライアント・アプリケーションには、Oracle JDBC Thin ドライバまたは OCI ドライバを使用できます。JDBC OCI ドライバはネイティブ・メソッドを使用するため、アプリケーションでこのドライバを使用すると大幅なパフォーマンスの向上を期待できます。

クライアントで実行できるアプリケーションは、JDBC サーバー側内部ドライバを使用することにより、Oracle Server でも実行できます。

アプリケーションで JDBC OCI ドライバを使用している場合、アプリケーションが動作するために、Oracle をクライアントへインストールする必要があります。たとえば、アプリケーションには Oracle Net とクライアント・ライブラリをインストールする必要があります。

JDBC Thin ドライバと OCI ドライバはどちらも、Oracle Advanced Security オプション（旧バージョンの ANO または ASO）の、データ暗号化のサポートおよび整合性チェックサム機能を提供します。詳細は、18-8 ページの「JDBC クライアント側セキュリティ機能」を参照してください。サーバー側内部ドライバには、このようなセキュリティは必要ありません。

アプレットの基本

この項では、JDBC Thin ドライバを利用するアプレットを記述する際に考慮する必要のある問題について説明します。

アプレットと、アプレットに関連するファイアウォール、ブラウザおよびセキュリティの問題については、18-16 ページの「アプレット内の JDBC」を参照してください。

アプレットとセキュリティ

特別な準備をしなければ、アプレットがオープンできるネットワーク接続は、ダウンロード元のホスト・マシンへの接続のみです。このため、アプレットはその発行元のマシン上のデータベースにのみ接続可能です。別のマシンで実行中のデータベースに接続するには、次の 2 つの方法があります。

- ホスト・マシン上で Oracle8 Connection Manager を使用します。アプレットを Oracle8 Connection Manager に接続し、Connection Manager から別のマシン上のデータベースに接続します。
- 署名付きアプレットを使用すると、他のマシンへのソケット接続権限を要求できます。

これらの項目は、18-17 ページの「アプレットを介したデータベースへの接続」で詳しく説明します。

Thin ドライバは、Oracle Advanced Security オプションの、データ暗号化のサポートおよび整合性チェックサム機能を提供します。詳細は、18-8 ページの「JDBC クライアント側セキュリティ機能」を参照してください。

アプレットとファイアウォール

JDBC Thin ドライバを使用するアプレットは、ファイアウォール経由でデータベースに接続できます。ファイアウォールの構成およびアプレット用接続文字列の作成の詳細は、18-22 ページの「ファイアウォールとアプレットの使用方法」を参照してください。

アプレットのパッケージ化と配置

アプレットをパッケージ化および配置するには、JDBC Thin ドライバ・クラスおよびアプレット・クラスを 1 つの zip ファイル内に配置する必要があります. 詳細は、18-24 ページの「アプレットのパッケージ化」を参照してください。

Oracle 拡張機能

Oracle JDBC アプリケーションおよびアプレットのプログラマが使用可能な Oracle 拡張機能は多数あり、次のカテゴリに分類されます。

- 型拡張要素（ROWID 型や REF CURSOR 型など）
- SQL 型のラッパー・クラス（`oracle.sql` パッケージ）
- ユーザー定義型にマップするカスタム Java クラスのサポート
- 拡張 LOB のサポート
- 接続、文および結果セット機能の拡張
- パフォーマンス強化

型拡張要素と拡張機能の概要は、第 5 章「Oracle 拡張機能の概要」を参照してください。詳細は、その先の章を参照してください。Oracle パフォーマンス強化については、第 12 章「パフォーマンス拡張要素」を参照してください。

パッケージ oracle.jdbc

Oracle9*i* では、JDBC 対応の Oracle 拡張機能はパッケージ oracle.jdbc に格納されます。このパッケージには、Oracle 拡張機能を指定するクラスおよびインタフェースが含まれています。java.sql のクラスおよびインタフェースでパブリック JDBC API が指定される方法と類似しています。

コードには、Oracle の以前のバージョンで使用されていたパッケージ oracle.jdbc.driver ではなく、パッケージ oracle.jdbc を使用してください。パッケージ oracle.jdbc.driver の使用は廃止されましたが、下位互換性を保持するためにサポートされています。

コードを変換するには、ソースの「oracle.jdbc.driver」を「oracle.jdbc」に置き換えて再コンパイルするだけです。この操作はピース単位ではできません。1 つのアプリケーションで参照されるすべてのクラスおよびインタフェースを変換する必要があります。変換は必須ではありませんが、お薦めします。Oracle の今後のリリースでは、パッケージ oracle.jdbc.driver と互換性のない機能が導入されていく予定です。

この変更の目的は、Oracle JDBC ドライバで複数の実装を可能にすることです。Oracle9*i* を含む、これまでのリリースでは、すべての Oracle JDBC ドライバで同じ最上位レベルの実装クラス（パッケージ oracle.jdbc.driver のクラス）が使用されています。oracle.jdbc を使用するようにコードを変換することで、別の実装クラスを使用するという今後の拡張機能が利用できるようになります。Oracle9*i* ではまだこのような拡張機能は導入されていませんが、近い将来には実現される予定です。

また、これらのインタフェースを使用すれば、パッケージ oracle.jdbc.driver では使用するのが難しいコード・パターンの使用も可能になります。たとえば、Oracle JDBC クラス用のラッパー・クラスを容易に作成することができます。すべての SQL 文を記録するために OracleStatement クラスをラップする場合も、OracleStatement をラップするクラスを作成するだけで実現できます。このクラスは、インタフェース oracle.jdbc.OracleStatement を実装し、インスタンス変数として oracle.jdbc.OracleStatement を保持します。パッケージ oracle.jdbc.driver を使用した場合、クラス oracle.jdbc.driver.OracleStatement を拡張できないため、このようなラッピング・パターンを実現するのは難しくなります。

コードには、パッケージ oracle.jdbc.driver のかわりに、新しいパッケージ oracle.jdbc を使用することをお薦めします。下位互換性を保持するために、oracle.jdbc.driver は継続してサポートされるため、変換は必須ではありません。ただし、コードで oracle.jdbc.driver を使用する場合は、今後のリリースでサポートされない機能が導入されることを考慮して、変換を実行することをお薦めします。

サーバー側の基本

Oracle JDBC サーバー側内部ドライバを使用すると、Java ストアド・プロシージャまたは Enterprise JavaBeans など、Oracle データベースの内部で実行されるコードは、実行中のデータベースにアクセスできます。

サーバー側ドライバの詳細は、18-27 ページの「サーバー上の JDBC: サーバー側内部ドライバ」を参照してください。

セッション・コンテキストおよびトランザクション・コンテキスト

サーバー側内部ドライバは、デフォルト・セッションおよびデフォルト・トランザクションのコンテキストで動作します。サーバー側ドライバ用のデフォルト・セッション・コンテキストおよびトランザクション・コンテキストの詳細は、18-31 ページの「サーバー側内部ドライバのセッション・コンテキストとトランザクション・コンテキスト」を参照してください。

データベースへの接続

サーバー側内部ドライバは、データベースへのデフォルト接続を使用します。データベースに接続するには、`DriverManager.getConnection()` メソッドまたは Oracle 固有の `OracleDriver` クラスの `defaultConnection()` メソッドを使用します。サーバー側ドライバを使用したデータベースへの接続の詳細は、18-28 ページの「サーバー側内部ドライバを使用したデータベースへの接続」を参照してください。

環境およびサポート

この項では、Oracle JDBC ドライバのプラットフォーム、環境およびサポート機能を簡単に説明します。次の項目が含まれます。

- サポートする JDK および JDBC のバージョン
- JNI 環境および Java 環境
- JDBC と Oracle Application Server
- JDBC と IDE

サポートする JDK および JDBC のバージョン

Oracle には 2 つのバージョンの Thin ドライバと OCI ドライバがあります。1 つは JDK 1.2.x 互換で、もう 1 つは JDK 1.1.x 互換です。JDK 1.2.x バージョンは、標準 JDBC 2.0 をサポートします。JDK 1.1.x バージョンは、ほとんどの JDBC 2.0 機能をサポートしますが、JDK 1.1.x バージョンでは JDBC 2.0 機能を使用できないので、Oracle 拡張機能によってこの機能を実現する必要があります。

JDK 1.1.x 環境から JDK 1.2.x 環境に移行するために必要な作業はほとんどありません。詳細は、4-5 ページの「JDK 1.1.x から JDK 1.2.x への移行」を参照してください。

注意：

- サーバー側内部ドライバがサポートするのは、JDK 1.2.x のみです。
- JDK 1.0.2 は現行ドライバではサポートされていません。
- ドライバ実装はそれぞれ、独自の JDBC クラス ZIP ファイル（JDK 1.2.x バージョンは `classes12.zip`、JDK 1.1.x バージョンは `classes111.zip`）を使用します。

サポートされるドライバ・バージョン、JDK バージョンおよびデータベース・バージョンの組合せについては、2-2 ページの「Oracle JDBC ドライバの要件および互換性」を参照してください。

JNI 環境および Java 環境

Oracle JDBC OCI ドライバは、標準 JNI（Java ネイティブ・インタフェース）を使用して Oracle OCI C ライブラリをコールします。リリース 8.1.6 より前のバージョンでは、OCI ドライバが JDK 1.0.2 をサポートしている場合は、NMI（Native Method Interface）を使用して C をコールしていました。NMI は、Sun 社による古い仕様で、JDK 1.0.2 がサポートする唯一のシステム固有なコール・インタフェースでした。

現在、Oracle JDBC は JNI をサポートしているので、Sun 社以外の Java Virtual Machine（JVM）、特に、Microsoft や IBM の JVM で OCI ドライバを使用できます。これらの JVM がサポートするシステム固有な C コールは、JNI のみです。

JDBC と Oracle Application Server

Oracle Application Server（OAS）は、分散オブジェクト指向アプリケーション用の、スケーラブル、強力、安全、かつ拡張可能なプラットフォームを提供するミドルウェア・サービスおよびツールのコレクションです。OAS は、Hypertext Transfer Protocol（HTTP）を使用する Web クライアント（ブラウザ）からのアプリケーションへのアクセスと、Common Object Request Broker Architecture（CORBA）と Internet Inter-ORB Protocol（IIOP）を使用する CORBA クライアントからのアプリケーションへのアクセスを、両方ともサポートしています。

中間層の JDBC OCI ドライバは、OAS（旧バージョンの Web Application Server: WAS）バージョン 3.0 以降で使用できます。OAS の配布ファイルには、JDBC がバンドルされています。OAS で JDBC を使用する方法の詳細は、Oracle Application Server のマニュアルを参照してください。

JDBC と IDE

Oracle JDeveloper Suite は、開発者に対し、Oracle Internet プラットフォームでコンポーネント・ベースのデータベース・アプリケーションを作成、デバッグおよび配置するための、単一の統合された製品群を提供します。Oracle JDeveloper 環境には、100% pure JDBC Thin ドライバおよびシステム固有な OCI ドライバなど、JDBC および SQLJ の統合的なサポートが含まれています。Oracle JDeveloper のデータベース・コンポーネントは、JDBC ドライバを使用してクライアントとサーバー上で稼働するアプリケーション間の接続を管理します。詳細は、Oracle JDeveloper のマニュアルを参照してください。

2

スタート・ガイド

この章では、初めに Oracle JDBC ドライバ・バージョン、データベース・バージョンおよび JDK バージョン間の互換性について説明します。次に、インストールと構成のテスト、および簡単なアプリケーションの実行方法の基本を説明します。次の項目が含まれます。

- Oracle JDBC ドライバの要件および互換性
- JDBC クライアント・インストールの検証

Oracle JDBC ドライバの要件および互換性

次の表 2-1 に、Oracle JDBC ドライバと Oracle データベースのバージョン間の互換性を示します。各 JDBC ドライバ・バージョンによってサポートされる JDK バージョンもリストします。

注意： リリース 8.1.6 から、Oracle JDBC ドライバは JDK 1.0.x バージョンをサポートしていません。

表 2-1 JDBC ドライバとデータベースの互換性

ドライバ・バージョン	サポートされるデータベース・バージョン	サポートされる JDK バージョン	使用可能なドライバ	備考
9.0.1	9.0.1、8.1.7、8.1.6、8.1.5、8.0.6、8.0.5、8.0.4、7.3.4	1.2.x、1.1.x	JDBC Thin ドライバ JDBC OCI ドライバ JDBC サーバー側 Thin ドライバ JDBC サーバー側内部ドライバ（8.1.6 以降のデータベースおよび JDK 1.2.x のみをサポートします）	
8.1.7	8.1.7、8.1.6、8.1.5、8.0.6、8.0.5、8.0.4、7.3.4	1.2.x、1.1.x	JDBC Thin ドライバ JDBC OCI ドライバ JDBC サーバー側 Thin ドライバ JDBC サーバー側内部ドライバ（8.1.6 以降のデータベースおよび JDK 1.2.x のみをサポートします）	
8.1.6	8.1.6、8.1.5、8.0.6、8.0.5、8.0.4、7.3.4	1.2.x、1.1.x	JDBC Thin ドライバ JDBC OCI ドライバ JDBC サーバー側 Thin ドライバ JDBC サーバー側内部ドライバ（8.1.6 以降のデータベースおよび JDK 1.2.x のみをサポートします）	Thin ドライバは標準サーバーがインストールされたサーバーでも使用できます。これには、データベース内部からリモート・データベースにアクセスするために、クライアント側 Thin ドライバと同じ使用方法と機能があります。
8.1.5	8.1.5、8.0.6、8.0.5、8.0.4、7.3.4	1.1.x、1.0.x	JDBC Thin ドライバ JDBC OCI ドライバ JDBC サーバー側内部ドライバ（8.1.5 データベースおよび JDK 1.1.x のみをサポートします）	リリース 8.1.5 のデータベースで使用する場合、クライアント側ドライバとサーバー側ドライバのどちらも構造化オブジェクトの完全なサポートを提供します。

表 2-1 JDBC ドライバとデータベースの互換性（続き）

ドライバ・バージョン	サポートされるデータベース・バージョン	サポートされるJDKバージョン	使用可能なドライバ	備考
8.0.6	8.0.6、8.0.5、8.0.4、7.3.4	1.1.x、1.0.x	JDBC Thin ドライバ JDBC OCI ドライバ **注意**：JDBC サーバー側内部ドライバは、バージョン 8.0.x 以前では使用できません。	
8.0.5	8.0.5、8.0.4、7.3.4	1.1.x、1.0.x	JDBC Thin ドライバ JDBC OCI ドライバ **注意**：JDBC サーバー側内部ドライバは、バージョン 8.0.x 以前では使用できません。	
8.0.4	8.0.4、7.3.4	1.1.x、1.0.x	JDBC Thin ドライバ JDBC OCI ドライバ **注意**：JDBC サーバー側内部ドライバは、バージョン 8.0.x 以前では使用できません。	
7.3.4	7.3.4	1.1.x、1.0.x	JDBC Thin ドライバ JDBC OCI ドライバ **注意**：JDBC サーバー側内部ドライバは、バージョン 8.0.x 以前では使用できません。	

注意：

- 各 JDK は、それぞれ別のクラス・ファイル、つまり `classes12.zip`、`classes111.zip` および `classes102.zip` 内のクラスを要求します。
- リリース 8.0.*x* のデータベースで使用する場合、JDBC ドライバは構造化オブジェクトをサポートしません。これは、JDBC がこれらのリリースに存在しない PL/SQL ファンクションに依存しているためです。
- Oracle 7.3.x には構造化オブジェクトまたは LOB のサポートはありません。
- クライアント側ドライバは、7.3.4 より前の 7.x データベースで動作する場合がありますが、これはテストもサポートもされていません。

JDBC クライアント・インストールの検証

次の項目が含まれます。

Oracle JDBC ドライバのインストールは、プラットフォームによって異なります。プラットフォーム固有の情報が記載されたマニュアルの、ドライバのインストール手順に従ってください。

この項では、JDBC ドライバの Oracle クライアントへのインストールを検証する手順を説明します。ここでは、選択したドライバのインストールが完了しているものとして説明します。

JDBC Thin ドライバのインストールを行った場合は、クライアント・マシンにその他のインストールは必要ありません。(JDBC Thin ドライバではデータベース上に TCP/IP リスナーが必要です。)

JDBC OCI ドライバのインストールを行った場合は、Oracle クライアント・ソフトウェアもインストールする必要があります。これには、Oracle Net および OCI ライブラリが含まれます。

インストールされたディレクトリとファイルの確認

この項では、（他の形式の Java もサポートされますが）Sun 社の Java Development Kit（JDK）がすでにシステムにインストールしてあるものとして説明します。Oracle は JDK 1.2.x バージョンまたは JDK 1.1.x バージョンのいずれかと互換性のある JDBC ドライバを提供しています。

Oracle8*i* Java 製品をインストールすると、`[ORACLE_HOME]/jdbc` ディレクトリが作成され、その下に次のようなサブディレクトリとファイルが格納されます。

- `demo/samples`: `samples` サブディレクトリには、SQL92 と Oracle SQL の構文、PL/SQL ブロック、ストリーム、ユーザー定義型、追加 Oracle 型拡張要素、Oracle パフォーマンス拡張要素の使用方法の例などのサンプル・プログラムが含まれます。
- `doc`: `doc` ディレクトリには、JDBC ドライバに関するドキュメントが含まれます。
- `lib`: `lib` ディレクトリには、Java クラスで必要な次の `.zip` ファイルが含まれます。
  - `classes12.zip` には、JDK 1.2.x で使用するためのクラス、つまり NLS サポートのために必要なクラスを除くすべての JDBC ドライバ・クラスが含まれます。
  - `nls_charset12.zip` には、JDK 1.2.x での NLS サポートのために必要なクラスが含まれます。
  - `jta.zip` および `jndi.zip` には、JDK 1.2.x 用の Java Transaction API および Java Naming and Directory Interface のためのクラスが含まれます。これらのファイルは、分散トランザクション管理のための JTA 機能、またはネーミング・サービスのための JNDI 機能を使用している場合にのみ要求されます。（Sun 社の Web サイトからもこれらのファイルを入手できますが、Oracle ドライバでテスト済みの Oracle からのバージョンを使用することをお薦めします。）
  - `classes111.zip` には、JDK 1.1.x で使用するためのクラス、つまり NLS サポートのために必要なクラスを除くすべての JDBC ドライバ・クラスが含まれます。

    `classes111.zip` には、JDK 1.1.x でオブジェクト、配列および LOB のための JDBC 2.0 機能を使用できる Oracle 拡張機能も含まれます。
  - `nls_charset11.zip` には、JDK 1.1.x での NLS サポートのために必要なクラスが含まれます。

  `nls_charset12.zip` ファイルと `nls_charset11.zip` ファイルは、固有 NLS キャラクタ・セットのサポートを提供します。完全な NLS サポートが不要な場合に、キャラクタ・セットを除外するオプションを提供するために、これらのファイルは `classes*.zip` ファイルから分離されています。`nls_charset12.zip` と `nls_charset11.zip` の詳細は、18-5 ページの「グローバリゼーション・サポートとオブジェクト型」を参照してください。
- `readme.txt`: `readme.txt` ファイルには、このマニュアルで説明されていないドライバに関するリリース固有の最新情報が記載されています。

これらのディレクトリがすべて作成され、ファイルが格納されていることを確認してください。

環境変数の確認

この項では、主に Sun Microsystems Solaris および Microsoft Windows NT プラットフォームに関して、JDBC OCI ドライバと JDBC Thin ドライバのために設定する必要がある環境変数について説明します。

インストールした JDBC OCI または Thin ドライバ用の CLASSPATH を指定する必要があります。JDK バージョン 1.2.x または 1.1.x のどちらを使用しているかによって、CLASSPATH に次の中から適切な値を指定してください。

- [Oracle Home]/jdbc/lib/classes12.zip
  (完全な NLS キャラクタ・サポートには、オプションとして
  [Oracle Home]/jdbc/lib/nls_charset12.zip)

または

- [Oracle Home]/jdbc/lib/classes111.zip
  (完全な NLS キャラクタ・サポートには、オプションとして
  [Oracle Home]/jdbc/lib/nls_charset11.zip)

CLASSPATH には、1 つの classes*.zip ファイル・バージョンと 1 つの nls_charset*.zip ファイル・バージョンしかないことを確認してください。

> **注意：** 第 14 章「接続プーリングとキャッシュ」で説明する JTA 機能または JNDI 機能を使用する場合は、CLASSPATH に jta.zip と jndi.zip も格納する必要があります。

JDBC OCI ドライバ： JDBC OCI ドライバをインストールする場合は、ライブラリ・パス環境変数に次の値を設定する必要もあります。

- Solaris では、次のように LD_LIBRARY_PATH を設定します。

  ```
  [Oracle Home]/lib
  ```

  このディレクトリには、libocijdbc9.so 共有オブジェクト・ライブラリが格納されます。

- Windows NT では、次のように PATH を設定します。

  ```
  [Oracle Home]¥lib
  ```

  このディレクトリには、ocijdbc8.dll 動的リンク・ライブラリが格納されます。

JDBC Thin ドライバ： JDBC Thin ドライバをインストールする場合は、他の環境変数を指定する必要はありません。

Java のコンパイルと実行の確認

Java がクライアント・システムで正しくセットアップされたことを確認するには、samples ディレクトリ（Windows NT マシンで JDBC ドライバを使用する場合は C:¥oracle¥ora81¥jdbc¥demo¥samples）で、javac（Java コンパイラ）および java（Java インタプリタ）を実行してエラーが発生しないことを確認してください。次のように入力します。

```
javac
```

その後、次のコマンドを入力します。

```
java
```

各コマンドは、オプションとパラメータのリストを示して終了します。単純なテスト・プログラムをコンパイルし、実行できることを確認してください。

JDBC ドライバのバージョンの確認

インストールした JDBC ドライバのバージョンを確認する必要がある場合は、OracleDatabaseMetaData クラスの getDriverVersion() メソッドをコールします。

ドライバのバージョンを確認する方法を、次にサンプル・コードで示します。

```
import java.sql.*;
import oracle.jdbc.*;

class JDBCVersion
{
   public static void main (String args[])
          throws SQLException
   {
      // Load the Oracle JDBC driver
      DriverManager.registerDriver
              (new oracle.jdbc.driver.OracleDriver());
      Connection conn = DriverManager.getConnection
              ("jdbc:oracle:thin:@host:port:sid","scott","tiger");

      // Create Oracle DatabaseMetaData object
      DatabaseMetaData meta = conn.getMetaData();

      // gets driver info:
      System.out.println("JDBC driver version is " + meta.getDriverVersion());
   }
}
```

JDBC およびデータベース接続のテスト : JdbcCheckup

samples ディレクトリには、特定の Oracle JDBC ドライバ用のサンプル・プログラムが格納されています。その 1 つである JdbcCheckup.java は、JDBC およびデータベース接続のテスト用です。このプログラムには、ユーザー名、パスワードおよび接続するデータベース名の入力が必要です。このプログラムは、データベースに接続し、「Hello World」という文字列の問合せを行い、それを画面に出力します。

samples ディレクトリで JdbcCheckup.java をコンパイルおよび実行します。問合せの結果の画面出力でエラーが発生しなければ、Java および JDBC は正しくインストールされています。

JdbcCheckup.java は簡単なプログラムですが、次の操作を実行する重要な機能を備えています。

- JDBC クラスを含む、必要な Java クラスのインポート
- JDBC ドライバの登録
- データベースへの接続
- 簡単な問合せの実行
- 問合せ結果の画面への出力

詳細は、3-2 ページの「JDBC での最初の処理」を参照してください。JDBC OCI ドライバ用の JdbcCheckup.java のリストを次に示します。

```
/*
 * This sample can be used to check the JDBC installation.
 * Just run it and provide the connect information.  It will select
 * "Hello World" from the database.
 */

// You need to import the java.sql package to use JDBC
import java.sql.*;

// We import java.io to be able to read from the command line
import java.io.*;

class JdbcCheckup
{
   public static void main(String args[])
          throws SQLException, IOException
   {
      // Load the Oracle JDBC driver
      DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
```

```
   // Prompt the user for connect information
   System.out.println("Please enter information to test connection to
                       the database");
   String user;
   String password;
   String database;

   user = readEntry("user: ");
   int slash_index = user.indexOf('/');
   if (slash_index != -1)
   {
      password = user.substring(slash_index + 1);
      user = user.substring(0, slash_index);
   }
   else
      password = readEntry("password: ");
   database = readEntry("database(a TNSNAME entry): ");

   System.out.print("Connecting to the database...");
   System.out.flush();

   System.out.println("Connecting...");
   Connection conn = DriverManager.getConnection
                    ("jdbc:oracle:oci8:@" + database, user, password);

   System.out.println("connected.");

   // Create a statement
   Statement stmt = conn.createStatement();

   // Do the SQL "Hello World" thing
   ResultSet rset = stmt.executeQuery("select 'Hello World'
                                       from dual");

   while (rset.next())
      System.out.println(rset.getString(1));
   // close the result set, the statement and connect
   rset.close();
   stmt.close();
   conn.close();
   System.out.println("Your JDBC installation is correct.");
}
```

```
   // Utility function to read a line from standard input
   static String readEntry(String prompt)
   {
      try
      {
         StringBuffer buffer = new StringBuffer();
         System.out.print(prompt);
         System.out.flush();
         int c = System.in.read();
         while (c != '\n' && c != -1)
         {
            buffer.append((char)c);
            c = System.in.read();
         }
         return buffer.toString().trim();
      }
      catch(IOException e)
      {
         return "";
      }
   }
}
```

3

基本機能

この章では、すべての JDBC アプリケーションに当てはまる、基本的な処理を説明します。また、Oracle JDBC ドライバがサポートする Java と JDBC の基本機能についても説明します。

次の項目が含まれます。

- JDBC での最初の処理
- サンプル : 接続、問合せおよび結果処理
- データ型マッピング
- JDBC 内の Java ストリーム
- JDBC プログラムでのストアド・プロシージャ・コール
- SQL 例外の処理

JDBC での最初の処理

この項では、Oracle JDBC ドライバの起動および実行方法について説明します。Oracle JDBC ドライバを使用する場合は、プログラムに特定のドライバ固有情報を含める必要があります。この項では、ドライバ固有情報の追加場所と追加方法について、チュートリアルの形式で説明します。チュートリアルを実行することにより、クライアントからデータベースへの接続および問合せを行うコードを作成できます。

クライアントからデータベースへの接続および問合せを行うには、次のタスクを実行するコードを提供する必要があります。

1. パッケージのインポート
2. JDBC ドライバの登録
3. データベースへの接続のオープン
4. Statement オブジェクトの作成
5. 問合せの実行と結果セット・オブジェクトの戻り
6. 結果セットの処理
7. 結果セットと Statement オブジェクトのクローズ
8. データベースの変更
9. 変更のコミット
10. 接続のクローズ

最初の 3 つのタスク用の Oracle ドライバ固有情報を用意して、プログラムが JDBC API を使用してデータベースへ接続することを可能にします。その他のタスクについては、任意の Java アプリケーションの場合と同様に、標準 JDBC Java コードを使用できます。

パッケージのインポート

使用する Oracle JDBC ドライバの種類にかかわらず、次の import 文をプログラム（必要な場合にのみ、java.math）の最初に記述する必要があります。

```
import java.sql.*;          標準 JDBC パッケージ
import java.math*;          BigDecimal クラスと BigInteger クラス
```

Oracle ドライバの提供する拡張機能を利用する場合は、次の Oracle パッケージをインポートします。ただし、これらのパッケージは、この項の例では必要ありません。

```
import oracle.jdbc.*;                Oracle の JDBC 対応拡張要素
import oracle.sql.*;
```

JDBC 標準の Oracle 拡張機能の概要は、第 5 章「Oracle 拡張機能の概要」を参照してください。

JDBC ドライバの登録

インストールしたドライバをプログラムに登録するためのコードを記述する必要があります。この作業は、JDBC DriverManager クラスの静的な registerDriver() メソッドを使用して行います。このクラスは、JDBC ドライバ・セットの管理に必要な基本サービスを提供します。

注意： 別の方法として、java.lang.Class クラスの forName() メソッドを使用して、JDBC ドライバを直接ロードすることもできます。たとえば、次のようになります。

```
Class.forName ("oracle.jdbc.OracleDriver");
```

ただし、このメソッドは、JDK に準拠した Java Virtual Machine でのみ有効です。Microsoft Java Virtual Machine 上では、使用できません。

Oracle JDBC ドライバのいずれかを使用するために、特定のドライバ名文字列を registerDriver() に宣言します。ドライバは、Java アプリケーションへ一度のみ登録します。

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

データベースへの接続のオープン

JDBC DriverManager クラスの静的な getConnection() メソッドを使用して、データベースへの接続をオープンします。このメソッドは、JDBC Connection クラスのオブジェクトを戻します。このオブジェクトには、ユーザー名、パスワード、使用する JDBC ドライバを識別する接続文字列および接続対象のデータベース名の入力が必要です。

データベースへの接続では、Oracle JDBC ドライバ固有情報の getConnection() メソッドへの入力が必要です。このメソッドを理解していない場合は、次の「getConnection() の形式について」を参照してください。

getConnection() メソッドをよく理解している場合は、インストールしたドライバに基づき、次の項のいずれかにスキップできます。

- 3-9 ページ「JDBC OCI ドライバ用の接続のオープン」
- 3-10 ページ「JDBC Thin ドライバ用の接続のオープン」

注意：

- JDK 1.2 では、接続方法として JNDI（Java Naming and Directory Interface）の使用をお薦めします。14-2 ページの「JNDI の Oracle データ・ソース・サポートに関する簡単な概要」および 14-8 ページの「データ・ソース・インスタンスの作成、JNDI での登録および接続」を参照してください。
- Thin ドライバを使用している場合は、接続時の OS 認証がサポートされていないことに注意してください。このため、特殊ログインはサポートされていません。
- この項での説明は、暗黙的な接続を使用するサーバー側内部ドライバには適用されません。詳細は、18-28 ページの「サーバー側内部ドライバを使用したデータベースへの接続」を参照してください。

---

getConnection() の形式について

次の項で Driver Manager クラスの getConnection() メソッドのシグネチャと機能について説明します。

- 3-5 ページ「データベース URL、ユーザー名およびパスワードの指定」
- 3-6 ページ「ユーザー名とパスワードを含むデータベース URL の指定」
- 3-6 ページ「データベース URL とプロパティ・オブジェクトの指定」

接続でデータベース名を指定する場合の書式は、次のいずれかにする必要があります。

- Oracle Net キーワード値ペア
- 書式 <*host_name*>:<*port_number*>:<*sid*> の文字列（Thin ドライバのみ）
- TNSNAMES エントリ（OCI ドライバのみ）

キーワード値ペアまたは TNSNAMES エントリを指定する方法については、『Oracle9*i* Net Services 管理者ガイド』を参照してください。

データベース URL、ユーザー名およびパスワードの指定

次のシグネチャは、別個のパラメータとして URL、ユーザー名およびパスワードを取ります。

```
getConnection(String URL, String user, String password);
```

URL の形式は次のとおりです。

```
jdbc:oracle:<drivertype>:@<database>
```

次の例では、Thin ドライバを使用して、パスワード tiger のユーザー scott を、SID orcl のデータベースへ、ホスト myhost のポート 1521 経由で接続します。

```
Connection conn = DriverManager.getConnection
                  ("jdbc:oracle:thin:@myhost:1521:orcl", "scott", "tiger");
```

OCI ドライバでデフォルト接続を使用する場合は、次のどちらかを指定します。

```
Connection conn = DriverManager.getConnection
                  ("jdbc:oracle:oci8:scott/tiger@");
```

または

```
Connection conn = DriverManager.getConnection
                  ("jdbc:oracle:oci8:@", "scott", "tiger");
```

すべての JDBC ドライバで、Oracle Net キーワード値ペアを使用してデータベースを指定することもできます。Oracle Net キーワード値ペアは、TNSNAMES エントリと置き換えて使用できます。次の例では、前述の例と同じパラメータを、キーワード値書式で使用します。

```
Connection conn = DriverManager.getConnection
                  (jdbc:oracle:oci8:@MyHostString","scott","tiger");
```

または

```
Connection conn = DriverManager.getConnection
   ("jdbc:oracle:oci8:@(description=(address=(host= myhost)
   (protocol=tcp)(port=1521))(connect_data=(sid=orcl)))","scott", "tiger");
```

ユーザー名とパスワードを含むデータベース URL の指定

次のシグネチャは、すべて URL パラメータの一部として URL、ユーザー名およびパスワードを取ります。

```
getConnection(String URL);
```

URL の形式は次のとおりです。

```
jdbc:oracle:<drivertype>:<user>/<password>@<database>
```

次の例では、OCI ドライバを使用して、パスワード tiger のユーザー scott をホスト myhost 上のデータベースに接続します。ただし、この場合、URL は、ユーザー ID とパスワードを含む、唯一の入力パラメータです。

```
Connection conn = DriverManager.getConnection
                        ("jdbc:oracle:oci8:scott/tiger@myhost");
```

Thin ドライバで接続する場合は、ポート番号と SID を指定する必要があります。たとえば、ポート 1521 上に TCP/IP リスナーを持つホスト myhost 上のデータベースに接続し、SID（システム識別子）が orcl である場合は、次のようになります。

```
Connection conn = DriverManager.getConnection
                        ("jdbc:oracle:thin:scott/tiger@myhost:1521:orcl");
```

データベース URL とプロパティ・オブジェクトの指定

次のシグネチャは、ユーザー名とパスワード（他の項目もあります）を指定するプロパティ・オブジェクトとともに、URL を取ります。

```
getConnection(String URL, Properties info);
```

URL の形式は、次のとおりです。

```
jdbc:oracle:<drivertype>:@<database>
```

URL に加え、標準 Java Properties クラスのオブジェクトを入力として使用します。たとえば、次のようになります。

```
java.util.Properties info = new java.util.Properties();
info.put ("user", "scott");
info.put ("password","tiger");
info.put ("defaultRowPrefetch","15");
getConnection ("jdbc:oracle:oci8:@",info);
```

表 3-1 に、Oracle JDBC ドライバがサポートする接続プロパティをリストします。

表 3-1 Oracle JDBC ドライバが認識する接続プロパティ

名前	短縮名	型	説明
user	利用不可	文字列型	データベースにログインするためのユーザー名。
password	利用不可	文字列型	データベースにログインするためのパスワード。
database	server	文字列型	データベースの接続文字列。
internal_logon	利用不可	文字列型	`sys` としてログオンできる `sysdba` または `sysoper` などのロール。
defaultRowPrefetch	prefetch	文字列型（整数値を含む）	サーバーからプリフェッチするデフォルト行数（デフォルト値は「10」）。
remarksReporting	remarks	文字列型（ブール値を含む）	`getTables()` および `getColumns()` が `TABLE_REMARKS` をレポートする場合は「TRUE」。`setRemarksReporting()` を使用する場合と等価（デフォルト値は「FALSE」）。
defaultBatchValue	batchvalue	文字列型（整数値を含む）	実行要求が発生するデフォルトのバッチ値（デフォルト値は「10」）。
includeSynonyms	synonyms	文字列型（ブール値を含む）	`DataBaseMetaDatagetColumns()` コールを実行する場合に、定義済みシノニム SQL エンティティから列情報を含めるには「TRUE」。これは接続 `setIncludeSynonyms()` コールと等価（デフォルト値は「FALSE」）。

暗号化ドライバおよび整合性ドライバの詳細は、表 18-4「暗号化と整合性のための OCI ドライバ・クライアント・パラメータ」および表 18-5「暗号化と整合性のための Thin ドライバ・クライアント・パラメータ」を参照してください。

sys ログオンのためのロールの使用方法

sys でログオンするためのロール（モード）を指定するには、internal_logon 接続プロパティを使用します。（この接続プロパティの詳細は、表 3-1「Oracle JDBC ドライバが認識する接続プロパティ」を参照してください。）sys でログオンするには、internal_logon 接続プロパティを、sysdba または sysoper に設定します。

> **注意：** ロールを指定する機能は、sys ユーザー名にのみサポートされます。

例： 次の例は、sys ログオンを指定するための、internal_logon および sysdba 引数の使用方法を示しています。

```
//import packages and register the driver
import java.sql.*;
import java.math.*;
DriverManager.registerDriver (new oracle.jdbc.OracleDriver());

//specify the properties object
java.util.Properties info = new java.util.Properties();
info.put ("user", "sys");
info.put ("password", "change_on_install");
info.put ("internal_logon","sysdba");

//specify the connection object
Connection conn = DriverManager.getConnection
                        ("jdbc:oracle:thin:@database",info);
...
```

Oracle パフォーマンス拡張要素のプロパティ これらのプロパティの一部は、Oracle パフォーマンス拡張要素で使用します。これらのプロパティの設定は、次のように OracleConnection オブジェクトで対応するメソッドを使用することと等価です。

- defaultRowPrefetch プロパティの設定は、setDefaultRowPrefetch() のコールと等価です。

  詳細は、12-19 ページの「Oracle 行プリフェッチ」を参照してください。

- remarksReporting プロパティの設定は、setRemarksReporting() のコールと等価です。

  詳細は、12-25 ページの「DatabaseMetaData TABLE_REMARKS レポート」を参照してください。

- defaultBatchValue プロパティの設定は、setDefaultExecuteBatch() コールと等価です。

  詳細は、12-4 ページの「Oracle バッチ更新」を参照してください。

例： 次の例は、java.util.Properties クラスの put() メソッドの使用方法を示します。この例では、Oracle パフォーマンス拡張要素のパラメータを設定する方法を示します。

```
//import packages and register the driver
import java.sql.*;
import java.math.*;
DriverManager.registerDriver (new oracle.jdbc.OracleDriver());

//specify the properties object
java.util.Properties info = new java.util.Properties();
info.put ("user", "scott");
info.put ("password", "tiger");
info.put ("defaultRowProfetch","20");
info.put ("defaultBatchValue", "5");

//specify the connection object
Connection conn = DriverManager.getConnection
                        ("jdbc:oracle:thin:@database",info);
...
```

JDBC OCI ドライバ用の接続のオープン

JDBC OCI ドライバでは、TNSNAMES エントリでデータベースを指定します。接続元のクライアント・コンピュータ上のファイル tnsnames.ora にリストされた、利用可能な TNSNAMES エントリを検索できます。Windows NT では、このファイルは [ORACLE_HOME]¥NETWORK¥ADMIN ディレクトリに格納されています。UNIX システムでは、/var/opt/oracle ディレクトリに格納されています。

たとえば、ホスト myhost 上のデータベースに、ユーザー scott、パスワード tiger（MyHostString の TNSNAMES エントリを持つ）で接続する場合、次のように入力します。

```
Connection conn = DriverManager.getConnection
                  ("jdbc:oracle:oci8:@MyHostString", "scott", "tiger");
```

「:」と「@」の両方の文字が必要であることに注意してください。

JDBC OCI ドライバと Thin ドライバでは、Oracle Net キーワード値ペアを使用してデータベースを指定することもできます。この方法は、TNSNAMES エントリよりも可読性が低下しますが、TNSNAMES.ORA ファイルの正確さに依存しません。Oracle Net キーワード値ペアは、その他の JDBC ドライバとの動作も可能です。

たとえば、ポート 1521 上に TCP/IP リスナーを持つホスト myhost 上のデータベースに接続し、SID（システム識別子）が orcl である場合、次のような文を使用します。

```
Connection conn = DriverManager.getConnection
   ("jdbc:oracle:oci8:@(description=(address=(host= myhost)
   (protocol=tcp)(port=1521))(connect_data=(sid=orcl)))","scott", "tiger");
```

> **注意：** Oracle JDBC はログイン・タイムアウトをサポートしていません。静的な DriverManager.setLoginTimeout() メソッドをコールしても、効果は生じません。

JDBC Thin ドライバ用の接続のオープン

Oracle クライアントのインストールに依存しないアプレットで JDBC Thin ドライバを使用可能なため、接続対象のデータベースの識別に TNSNAMES エントリは使用できません。次のいずれかを実行する必要があります。

- 明示的にホスト名、接続対象データベースの TCP/IP ポートおよび Oracle SID のリストを作成します。

または

- キーワード値ペアのリストを使用します。

> **注意：** JDBC Thin ドライバは、TCP/IP プロトコルのみをサポートします。

たとえば、ポート 1521 上にデータベース SID（システム識別子）orcl 用の TCP/IP リスナーを持つホスト myhost 上のデータベースへ接続する場合、この文字列を使用します。ユーザー scott、パスワード tiger としてログインできます。

```
Connection conn = DriverManager.getConnection
                  ("jdbc:oracle:thin:@myhost:1521:orcl", "scott", "tiger");
```

Oracle Net キーワード値ペアを使用してデータベースを指定することもできます。これは、最初のバージョンよりも可読性が低下しますが、他の JDBC ドライバとの動作も可能です。

```
Connection conn = DriverManager.getConnection
    ("jdbc:oracle:thin:@(description=(address=(host=myhost)
    (protocol=tcp)(port=1521))(connect_data=(sid=orcl)))", "scott", "tiger");
```

> **注意：** Oracle JDBC はログイン・タイムアウトをサポートしていません。静的な DriverManager.setLoginTimeout() メソッドをコールしても、効果は生じません。

Statement オブジェクトの作成

データベースへ接続すると、処理の進行中に Connection オブジェクトを作成します。その後、Statement オブジェクトを作成します。使用する JDBC Connection オブジェクトの createStatement() メソッドは、JDBC Statement クラスのオブジェクトを戻します。Connection オブジェクト conn を作成した前項の例の続きとなる、Statement オブジェクトの作成方法の例を次に示します。

```
Statement stmt = conn.createStatement();
```

この文は、標準 JDBC 構文に準拠しており、Oracle 固有の部分がないことに注意してください。

問合せの実行と結果セット・オブジェクトの戻り

データベースへの問合せを行う場合、Statement オブジェクトの executeQuery() メソッドを使用します。このメソッドは、入力として SQL 文を受け取り、JDBC ResultSet オブジェクトを戻します。

この例の、Statement オブジェクト stmt 作成後の次の処理は、問合せを実行し、ResultSet オブジェクトに、EMP という名前の従業員の表の ENAME（従業員名）列の内容を移入することです。

```
ResultSet rset = stmt.executeQuery ("SELECT ename FROM emp");
```

この文にも、Oracle 固有の部分はなく、標準 JDBC 構文に準拠しています。

結果セットの処理

問合せの実行後は、ResultSet オブジェクトの next() メソッドを使用して結果を反復します。このメソッドは、結果セットを行ごとに進み、結果セットの最後に達するとそれを検出します。

結果セット内を反復しながらデータを引き出すには、ResultSet オブジェクトの適切な getXXX() メソッドを使用します。この XXX には、Java のデータ型が対応します。

たとえば、次のコードは、ResultSet オブジェクト rset 内を以前の項から反復して、各従業員名の取出しおよび出力を行います。

```
while (rset.next())
   System.out.println (rset.getString(1));
```

この文も、標準 JDBC 構文に準拠しています。next() メソッドは、結果セットの最後に達すると FALSE を戻します。従業員名は、Java 文字列として実装されます。

問合せを実行し、結果を出力する方法を示す完全なサンプル・アプリケーションについては、20-2 ページの「EMP 表からの名前のリストの取出し : Employee.java」を参照してください。

結果セットと Statement オブジェクトのクローズ

ResultSet と Statement オブジェクトの使用後に、明示的にこれらをクローズする必要があります。これは、Oracle JDBC ドライバの使用時に作成した、すべての ResultSet および Statement オブジェクトに適用されます。ドライバには、ファイナライザ・メソッドはありません。このため、クリーン・アップ・ルーチンは、ResultSet および Statement クラスの close() メソッドを使用して実行します。明示的に ResultSet および Statement オブジェクトをクローズしないと、深刻なメモリー・リークが発生する場合があります。また、データベースのカーソルが不足します。結果セットまたは文をクローズすると、データベース内の対応するカーソルが解放されます。

たとえば、ResultSet オブジェクトが rset で、Statement オブジェクトが stmt の場合、次の行を使用して結果セットと文をクローズできます。

```
rset.close();
stmt.close();
```

指定した Connection オブジェクトが作成する Statement オブジェクトをクローズする場合、接続自体はオープンしたままになります。

注意 : 一般に、close() 文は finally 句に書き込みます。

データベースの変更

INSERT 操作または UPDATE 操作など、データベースへの変更を記述するには、一般に PreparedStatement オブジェクトを作成します。これによって、様々な入力パラメータのセットで文を実行できます。JDBC Connection オブジェクトの prepareStatement() メソッドを使用すると、様々なバインド・パラメータを取り、文定義で JDBC PreparedStatement オブジェクトを戻す文を定義できます。

データベースに送信するプリコンパイルされた SQL 文にデータをバインドするには、PreparedStatement オブジェクトで setXXX() メソッドを使用します。様々な setXXX() メソッドについては、6-11 ページの「標準 setObject() および Oracle setOracleObject() メソッド」および 6-12 ページの「他の setXXX() メソッド」を参照してください。

この項で説明した機能は、標準 JDBC 構文に準拠しており、Oracle 独自の部分がないことに注意してください。

次の例では、プリコンパイルされた SQL 文を使用して、EMP 表に 2 行を追加する INSERT 操作を実行する方法を示します。完全なサンプル・アプリケーションについては、20-3 ページの「EMP 表への名前の挿入 : InsertExample.java」を参照してください。

```
// Prepare to insert new names in the EMP table
PreparedStatement pstmt =
  conn.prepareStatement ("insert into EMP (EMPNO, ENAME) values (?, ?)");

// Add LESLIE as employee number 1500
pstmt.setInt (1, 1500);          // The first ? is for EMPNO
pstmt.setString (2, "LESLIE");   // The second ? is for ENAME
// Do the insertion
pstmt.execute ();

// Add MARSHA as employee number 507
pstmt.setInt (1, 507);           // The first ? is for EMPNO
pstmt.setString (2, "MARSHA");   // The second ? is for ENAME
// Do the insertion
pstmt.execute ();

// Close the statement
pstmt.close();
```

変更のコミット

デフォルトでは、DML 操作（INSERT、UPDATE、DELETE）は実行時に自動的にコミットされます。これは自動コミット・モードともいいます。ただし、Connection オブジェクトでの次のメソッドのコールによって、自動コミット・モードを使用禁止にできます。

```
conn.setAutoCommit(false);
```

（自動コミット・モードの詳細と、使用禁止にする方法の例は、19-6 ページの「自動コミット・モードの無効化」を参照してください。）

自動コミット・モードを使用禁止にした場合は、Connection オブジェクトで適切なメソッドをコールして、変更を手動でコミットするか、ロールバックする必要があります。

```
conn.commit();
```

または

```
conn.rollback();
```

COMMIT 操作または ROLLBACK 操作は、直前の COMMIT または ROLLBACK 以降に実行されたすべての DML 文に影響を与えます。

重要：

- 自動コミット・モードを使用禁止にして、直前の変更を明示的にコミットまたはロールバックせずに接続をクローズした場合は、暗黙的な COMMIT 操作が実行されます。
- CREATE または ALTER などのすべての DDL 操作には、常に暗黙的な COMMIT が含まれます。自動コミット・モードを使用禁止にした場合、この暗黙的な COMMIT は DDL 文をコミットするのみではなく、まだ明示的にコミットまたはロールバックされていない保留 DML 文もコミットします。

接続のクローズ

作業が完了した後に、データベースへの接続をクローズする必要があります。これは、次のように Connection オブジェクトの close() メソッドを使用して実行します。

```
conn.close();
```

注意： 一般に、close() 文は finally 句に書き込みます。

サンプル : 接続、問合せおよび結果処理

次の例は、前の項で説明した処理を例証するものです。Oracle JDBC Thin ドライバを登録してデータベースに接続し、Statement オブジェクトを作成して問合せを実行し、結果セットを処理します。

Statement オブジェクトの作成、問合せの実行、ResultSet の復帰と処理、および文と接続のクローズを行うコードはすべて標準 JDBC 構文に準拠していることに注意してください。

```
import java.sql.*;
import java.math.*;
import java.io.*;
import java.awt.*;

class JdbcTest {
   public static void main (String args []) throws SQLException {
      // Load Oracle driver
      DriverManager.registerDriver (new oracle.jdbc.OracleDriver());
      // Connect to the local database
      Connection conn = DriverManager.getConnection
                ("jdbc:oracle:thin:@myhost:1521:ORCL","scott", "tiger");

      // Query the employee names
      Statement stmt = conn.createStatement ();
      ResultSet rset = stmt.executeQuery ("SELECT ename FROM emp");
      // Print the name out
      while (rset.next ())
         System.out.println (rset.getString (1));

      //close the result set, statement, and the connection
      rset.close();
      stmt.close();
      conn.close();
   }
}
```

OCI ドライバ用のコードを利用する場合は、Connection 文を次の文で置き換えます。

```
Connection conn = DriverManager.getConnection
                  ("jdbc:oracle:oci8:@MyHostString", "scott", "tiger");
```

MyHostString には、TNSNAMES.ORA ファイル内のエントリを指定します。

データ型マッピング

Oracle JDBC ドライバは、Oracle 固有の BFILE および ROWID データ型と REF CURSOR カテゴリの型に加えて、標準 JDBC 1.0 および 2.0 型をサポートしています。

この項では、標準および Oracle 固有の SQL-Java デフォルト型マッピングについて説明します。

マッピングの表

参照のために、表 3-2 には SQL データ型、JDBC タイプコード、標準 Java 型および Oracle 拡張型間のデフォルト・マッピングを示します。

SQL データ型列には、データベースに存在する SQL 型をリストします。

JDBC タイプコード列には、JDBC 標準によってサポートされ、java.sql.Types クラス、または oracle.jdbc.OracleTypes クラスで Oracle によって定義されるデータのタイプコードをリストします。標準型のタイプコードは、この 2 つのクラスで同一です。

標準 Java 型列には、Java 言語で定義される標準型をリストします。

Oracle 拡張機能 Java 型列には、データベース内の各 SQL データ型に対応する oracle.sql.* Java 型をリストします。これらは、oracle.sql.* Java 型の SQL データすべての取出しを可能にする Oracle 拡張機能です。SQL データ型を oracle.sql データ型にマップすることにより、データの格納と取出しが情報の損失なしで可能になります。oracle.sql.* パッケージの詳細は、5-6 ページの「パッケージ oracle.sql」を参照してください。

表 3-2 SQL 型と Java 型間のデフォルト・マッピング

SQL データ型	JDBC タイプコード	標準 Java 型	Oracle 拡張機能 Java 型
	標準 JDBC 1.0 型:		
CHAR	java.sql.Types.CHAR	java.lang.String	oracle.sql.CHAR
VARCHAR2	java.sql.Types.VARCHAR	java.lang.String	oracle.sql.CHAR
LONG	java.sql.Types.LONGVARCHAR	java.lang.String	oracle.sql.CHAR
NUMBER	java.sql.Types.NUMERIC	java.math.BigDecimal	oracle.sql.NUMBER
NUMBER	java.sql.Types.DECIMAL	java.math.BigDecimal	oracle.sql.NUMBER
NUMBER	java.sql.Types.BIT	boolean	oracle.sql.NUMBER
NUMBER	java.sql.Types.TINYINT	byte	oracle.sql.NUMBER
NUMBER	java.sql.Types.SMALLINT	short	oracle.sql.NUMBER
NUMBER	java.sql.Types.INTEGER	int	oracle.sql.NUMBER

表 3-2 SQL 型と Java 型間のデフォルト・マッピング （続き）

SQL データ型	JDBC タイプコード	標準 Java 型	Oracle 拡張機能 Java 型
NUMBER	java.sql.Types.BIGINT	long	oracle.sql.NUMBER
NUMBER	java.sql.Types.REAL	float	oracle.sql.NUMBER
NUMBER	java.sql.Types.FLOAT	double	oracle.sql.NUMBER
NUMBER	java.sql.Types.DOUBLE	double	oracle.sql.NUMBER
RAW	java.sql.Types.BINARY	byte[]	oracle.sql.RAW
RAW	java.sql.Types.VARBINARY	byte[]	oracle.sql.RAW
LONGRAW	java.sql.Types.LONGVARBINARY	byte[]	oracle.sql.RAW
DATE	java.sql.Types.DATE	java.sql.Date	oracle.sql.DATE
DATE	java.sql.Types.TIME	java.sql.Time	oracle.sql.DATE
DATE	java.sql.Types.TIMESTAMP	javal.sql.Timestamp	oracle.sql.DATE
	標準 JDBC 2.0 型 :		
BLOB	java.sql.Types.BLOB	java.sql.Blob	oracle.sql.BLOB
CLOB	java.sql.Types.CLOB	java.sql.Clob	oracle.sql.CLOB
ユーザー定義オブジェクト	java.sql.Types.STRUCT	java.sql.Struct	oracle.sql.STRUCT
ユーザー定義参照	java.sql.Types.REF	java.sql.Ref	oracle.sql.REF
ユーザー定義コレクション	java.sql.Types.ARRAY	java.sql.Array	oracle.sql.ARRAY
	Oracle 拡張機能 :		
BFILE	oracle.jdbc.OracleTypes.BFILE	利用不可	oracle.sql.BFILE
ROWID	oracle.jdbc.OracleTypes.ROWID	利用不可	oracle.sql.ROWID
REF CURSOR 型	oracle.jdbc.OracleTypes.CURSOR	java.sql.ResultSet	oracle.jdbc.OracleResultSet

注意： JDK 1.1.x で、JDBC 2.0 型をサポートするには、Oracle パッケージ `oracle.jdbc2` が必要です。（JDK 1.2.x では、JDBC 2.0 型は標準 `javal.sql` パッケージによってサポートされています。）

SQL データ型の有効なマップが可能なすべての Java データ型のリストは、21-2 ページの「有効な SQL-JDBC データ型マッピング」を参照してください。

型マッピングの詳細は、第 5 章「Oracle 拡張機能の概要」を参照してください。第 5 章では、次の詳細も参照できます。

- パッケージ oracle.sql、oracle.jdbc および oracle.jdbc2
- Oracle BFILE および ROWID データ型に対応した型拡張要素、および REF CURSOR カテゴリのユーザー定義型

マッピングに関する注意

この項では、NUMBER およびユーザー定義型のマッピングに関する詳細を示します。

ユーザー定義型について

オブジェクト、オブジェクト参照およびコレクションなどのユーザー定義型は、デフォルトで（java.sql.Struct などの）緩い Java 型にマップされますが、かわりに強い型指定のカスタム Java クラスにもマップできます。カスタム Java クラスは、次のいずれかのインタフェースを実装できます。

- 標準 java.sql.SQLData（ユーザー定義オブジェクトのみ）
- Oracle 固有 oracle.sql.ORAData（主にユーザー定義オブジェクト、オブジェクト参照およびコレクション用です。ただし、任意の種類のカスタマイズされた処理が必要な場合は任意の SQL 型からマップできます）。

カスタム Java クラスと SQLData および ORAData インタフェースの詳細は、8-2 ページの「Oracle オブジェクトのマッピング」および 8-10 ページの「Oracle オブジェクト用のカスタム・オブジェクト・クラスの作成と使用方法」を参照してください。（これらの項では、主にユーザー定義オブジェクトのカスタム Java クラスについて扱いますが、その他の種類のカスタム Java クラスについての一般情報もあります。）

NUMBER 型について

Oracle NUMBER 値が対応できる様々なタイプコードでは、マッピングを正しく動作させるために、データのサイズに適切な getter ルーチンをコールします。たとえば、-128 < x < 128 の項目 x に対して Javatinyint 値を取得するには、getByte() をコールします。

JDBC 内の Java ストリーム

次の項目が含まれます。

- LONG または LONG RAW 列のストリーム
- CHAR、VARCHAR または RAW 列のストリーム
- データ・ストリームと複数列
- ストリームと行のプリフェッチ
- ストリームのクローズ
- LOB および外部ファイルのストリーム

この項では、Oracle JDBC ドライバで様々なデータ型の Java ストリームを処理する方法を説明します。データ・ストリームを使用することにより、2GB までの LONG 型の列データを読み取れます。ストリームに対応付けられたメソッドを使用すると、データを増分的に読み取れます。

Oracle JDBC ドライバは、サーバーとクライアント間の双方向のデータ・ストリーム操作をサポートします。ドライバは、すべてのストリーム変換、つまりバイナリ、ASCII および Unicode をサポートします。次に、ストリームの各タイプを簡単に説明します。

- データの RAW バイトのために使用されるバイナリ・ストリーム。これは、`getBinaryStream()` メソッドに対応します。
- ISO-Latin-1 エンコーディングで ASCII バイトのために使用される ASCII ストリーム。これは、`getAsciiStream()` メソッドに対応します。
- `UCS-2` エンコーディングで Unicode バイトのために使用される Unicode ストリーム。これは、`getUnicodeStream()` メソッドに対応します。

メソッド `getBinaryStream()`、`getAsciiStream()` および `getUnicodeStream()` は、`InputStream` オブジェクト内のバイト・データを戻します。これらのメソッドの詳細は、第 7 章「LOB と BFILE の操作」を参照してください。

ストリーム・データを読み書きする方法を示す完全なサンプル・アプリケーションについては、20-17 ページの「ストリーム : StreamExample.java」を参照してください。

LONG または LONG RAW 列のストリーム

問合せが 1 つ以上の LONG または LONG RAW 列を選択すると、JDBC ドライバはストリーム・モードでこれらの列をクライアントに転送します。executeQuery() または next() へのコール後に、LONG 列のデータは読込み待機状態になります。

LONG 列のデータにアクセスするには、この列を Java InputStream として取得し、InputStream オブジェクトの read() メソッドを使用します。また、データを文字列またはバイト配列として取得することにより、ドライバによるストリーム処理を行う方法もあります。

3 種類のストリームのうち、任意のものを使用して、LONG または LONG RAW データを取得できます。ドライバは、データベースおよびドライバのキャラクタ・セットに応じて、NLS 変換を行います。NLS の詳細は、18-2 ページの「JDBC およびグローバリゼーション・サポート」を参照してください。

LONG RAW データの変換

getBinaryStream() へのコールにより、RAW がその状態のまま戻されます。getAsciiStream() へのコールにより、RAW データの 16 進データへの変換が行われ、ASCII コードで戻されます。getUnicodeStream() へのコールにより、RAW データの 16 進データへの変換が行われ、Unicode バイトが戻されます。

たとえば、LONG RAW 列にバイト 20 21 22 が含まれている場合、次のバイトが戻されます。

LONG RAW	BinaryStream	ASCIIStream	UnicodeStream
20 21 22	20 21 22	49 52 49 53 49 54 次の値も等価 '1' '4' '1' '5' '1' '6'	0049 0052 0049 0053 0049 0054 次の値も等価 '1' '4' '1' '5' '1' '6'

たとえば、LONG RAW の値 20 は、16 進数では、14 または "1" "4" で表されます。ASCII では、1 は "49" で、"4" は "52" で表されます。Unicode では、個別の値を区別するために、0 の埋込みが行われます。このため、16 進数の 14 は、0 "1" 0 "4" になります。この値を Unicode で表すと、0 "49" 0 "52" になります。

LONG データの変換

LONG データを getAsciiStream() で取得すると、ドライバはデータベースの基礎となるデータに US7ASCII または WE8ISO8859P1 キャラクタ・セットが使用されていると仮定します。この仮定が正しいと、ドライバは ASCII 文字に対応するバイトを戻します。データベースで US7ASCII と WE8ISO8859P1 のいずれのキャラクタ・セットも使用されていない場合、getAsciiStream() へのコールに意味のないコードが戻されます。

getUnicodeStream() を使用して LONG データを取得すると、Unicode 文字のストリームが UCS-2 エンコーディングで戻されます。これは、Oracle がサポートする、データベースの基礎となるキャラクタ・セットすべてに当てはまります。

getBinaryStream() を使用して LONG データを取得する場合、次の 2 つのケースが考えられます。

- ドライバが JDBC OCI で、クライアント・キャラクタ・セットが US7ASCII または WE8ISO8859P1 以外に設定されている場合、getBinaryStream() をコールすると、UTF-8 が戻されます。クライアント・キャラクタ・セットが US7ASCII または WE8ISO8859P1 に設定されている場合、US7ASCII バイト・ストリームが戻されます。
- ドライバが JDBC Thin で、データベース・キャラクタ・セットが US7ASCII または WE8ISO8859P1 以外に設定されている場合、getBinaryStream() をコールすると、UTF-8 が戻されます。サーバー側キャラクタ・セットが US7ASCII または WE8ISO8859P1 に設定されている場合、US7ASCII バイト・ストリームが戻されます。

キャラクタ・セットに基づく、ドライバからのデータの戻りの詳細は、18-2 ページの「JDBC およびグローバリゼーション・サポート」を参照してください。

注意： LONG または LONG RAW 列をストリーム（デフォルト）として受信するには、データベースから受信するデータの順序に注意する必要があります。詳細は、3-26 ページの「データ・ストリームと複数列」を参照してください。

表 3-3 に、LONG および LONG RAW データ変換の概要をストリーム・タイプ別に示します。

表 3-3 LONG および LONG RAW データの変換

データ型	BinaryStream	AsciiStream	UnicodeStream
LONG	Unicode UTF-8 の文字を表すバイト。次のような場合、バイトは US7ASCII または WE8ISO8859P1 のキャラクタを表します。 ■ クライアントの NLS_LANG の値が US7ASCII または WE8ISO8859P1 の場合。 または ■ データベース・キャラクタ・セットが US7ASCII または WE8ISO8859P1 の場合。	ISO-Latin-1（WE8ISO8859P1）エンコーディングの文字を表すバイト。	Unicode UCS-2 エンコーディングの文字を表すバイト。
LONG RAW	変換なし。	16 進バイトの ASCII 表現。	16 進バイトの Unicode 表現。

LONG RAW データのストリーム例

getXXXStream() メソッドには、データの増分的取得を可能にする機能があります。一方、getBytes() は、すべてのデータを一度のコールでフェッチします。この項には、バイナリ・データ・ストリームの取得方法を示す 2 つの例が含まれています。最初の例では、getBinaryStream() メソッドを使用して LONG RAW データを取得します。2 番目の例では、getBytes() メソッドを使用します。

getBinaryStream() を使用した LONG RAW データ列の取得 Java を使用したこの例では、LONG RAW 列の内容をローカル・ファイル・システム上のファイルに書き込みます。この場合、ドライバはデータを増分的にフェッチします。

次のコードは、名前 LESLIE に対応付けられた LONG RAW データ列を格納する表を作成します。

```
-- SQL code:
create table streamexample (NAME varchar2 (256), GIFDATA long raw);
insert into streamexample values ('LESLIE', '00010203040506070809');
```

次の Java コードの一部は、LESLIE LONG RAW 列のデータを、leslie.gif というファイルに書き込みます。

```
ResultSet rset = stmt.executeQuery
                 ("select GIFDATA from streamexample where NAME='LESLIE'");

// get first row
if (rset.next())
{
    // Get the GIF data as a stream from Oracle to the client
    InputStream gif_data = rset.getBinaryStream (1);
   try
   {
      FileOutputStream file = null;
      file = new FileOutputStream ("leslie.gif");
      int chunk;
      while ((chunk = gif_data.read()) != -1)
         file.write(chunk);
   }
   catch (Exception e)
   {
      String err = e.toString();
      System.out.println(err);
   }
   finally
   {
      if file != null()
         file.close();
   }
}
```

この例では、GIFDATA 列の内容は、チャンク・サイズのピースで増分的にデータベースとクライアント間で転送されます。getBinaryStream() へのコールにより戻される InputStream オブジェクトは、データベース接続から直接データを読み込みます。

getBytes() を使用した LONG RAW データ列の取得 この例では、getBinaryStream() のかわりに getBytes() を使用して、GIFDATA 列の内容を取得します。この場合、ドライバは一度のコールですべてのデータをフェッチして、バイト配列に格納します。以前のコードは、次のように書き換えられます。

```
ResultSet rset2 = stmt.executeQuery
                  ("select GIFDATA from streamexample where NAME='LESLIE'");

// get first row
if (rset2.next())
{
   // Get the GIF data as a stream from Oracle to the client
   byte[] bytes = rset2.getBytes(1);
   try
   {
      FileOutputStream file = null;
      file = new FileOutputStream ("leslie2.gif");
      file.write(bytes);
   }
   catch (Exception e)
   {
      String err = e.toString();
      System.out.println(err);
   }
   finally
   {
      if file != null()
         file.close();
   }
}
```

LONG RAW 列には、2GB までのデータを格納できるため、getBytes() の使用例は getBinaryStream() の使用例に比べて多量のメモリーを使用します。LONG または LONG RAW 列のデータの最大サイズが不明な場合は、ストリームを使用してください。

LONG または LONG RAW のストリーム回避

JDBC ドライバは、任意の LONG および LONG RAW を自動的にストリーム処理します。ただし、データ・ストリームを避ける状況が生じる場合もあります。たとえば、LONG 列のサイズが非常に小さい場合、データが増分的ではなく、一度のコールで戻される方が望ましい場合があります。

ストリームを回避するには、defineColumnType() メソッドを使用して LONG 列の型を再定義します。たとえば、LONG または LONG RAW 列を VARCHAR 型または VARBINARY 型として再定義すると、ドライバがデータを自動的にストリーム処理することはなくなります。

列の型を defineColumnType() を使用して再定義する場合、問合せの中ですべての列を宣言する必要があります。すべての列を宣言しないと、executeQuery() は失敗します。また、Statement オブジェクトを oracle.jdbc.OracleStatement オブジェクトにキャストする必要があります。

さらに、defineColumnType() を使用することにより、問合せの実行時にドライバがデータベースへ 2 往復せずにすみます。defineColumnType() を使用しない場合、JDBC ドライバは列タイプのデータ型を要求する必要があります。

前の項の例を使用して、Statement オブジェクト stmt は OracleStatement へキャストされ、LONG RAW データを含む列は VARBINARAY 型として再定義されます。このデータはストリーム化されず、バイト配列で戻されます。

```
//cast the statement stmt to an OracleStatement
oracle.jdbc.OracleStatement ostmt =
   (oracle.jdbc.OracleStatement)stmt;

//redefine the LONG column at index position 1 to VARBINARY
ostmt.defineColumnType(1, Types.VARBINARY);

// Do a query to get the images named 'LESLIE'
ResultSet rset = ostmt.executeQuery
         ("select GIFDATA from streamexample where NAME='LESLIE'");

// The data is not streamed here
rset.next();
byte [] bytes = rset.getBytes(1);
```

CHAR、VARCHAR または RAW 列のストリーム

defineColumnType() Oracle 拡張機能を使用して CHAR、VARCHAR または RAW 列を LONGVARCHAR または LONGVARBINARY として再定義する場合は、列をストリームとして取得できます。このプログラムは、列が実際に LONG または LONG RAW 型であるかのように動作します。通常これらの列は短いため、これが問題になることはほとんどありません。

CHAR、VARCHAR または RAW 列を、列タイプを再定義することなくデータ・ストリームとして取得しようとすると、JDBC ドライバは Java InputStream を戻しますが、実際のストリームは発生しません。これらのデータ型の場合、JDBC ドライバは、executeQuery() メソッドまたは next() メソッドへのコール中に、データをメモリー内のバッファに完全にフェッチします。getXXXStream() エントリ・ポイントは、このバッファからデータを読み取るストリームを戻します。

データ・ストリームと複数列

問合せにより選択された複数列の 1 つにデータ・ストリームが含まれる場合、ストリーム列に続く列の内容は、ストリームの読込みが終了するまで使用できません。後続の列が読み込まれると、ストリーム列は使用できなくなります。ストリーム列を超えて列を読み込もうとすると、ストリーム列がクローズされます。詳細は、3-29 ページの「ストリーム・データを使用する際の注意」を参照してください。

複数列によるストリーミングの例

次の問合せについて検討します。

```
ResultSet rset = stmt.executeQuery
        ("select DATECOL, LONGCOL, NUMBERCOL from TABLE");
while rset.next()
{
   //get the date data
   java.sql.Date date = rset.getDate(1);

   // get the streaming data
   InputStream is = rset.getAsciiStream(2);

   // Open a file to store the gif data
   FileOutputStream file = new FileOutputStream ("ascii.dat");

   // Loop, reading from the ascii stream and
   // write to the file
   int chunk;
   while ((chunk = is.read ()) != -1)
      file.write(chunk);
   // Close the file
   file.close();

   //get the number column data
   int n = rset.getInt(3);
}
```

各行の受信データは、次の形式になります。

```
<a date><the characters of the long column><a number>
```

イテレータの各列を処理するときには、数値列を読み込む前にストリーム列の処理を完了する必要があります。

ただし、サーバーとクライアント間を Java ストリームとしても転送されることのある LOB データの場合、この動作は当てはまりません。ドライバによる LOB データの処理方法の詳細は、3-27 ページの「LOB および外部ファイルのストリーム」を参照してください。

ストリーム・データ列のバイパス

ストリーム・データを含む列の読込みを回避することが望ましい場合があります。ストリーム列のデータの読込みを回避するには、ストリーム・オブジェクトの close() メソッドをコールします。このメソッドを使用すると、ストリーム・データが廃棄され、ドライバがストリーム以降の非ストリーム列データを継続して読み込むことが可能になります。意図的にストリームを廃棄する場合でも、SELECT で指定した順序で列をコールすることは、よいプログラミング手法です。

次の例では、LONG 列のストリーム・データを廃棄し、DATE および NUMBER 列のデータのみをリカバリします。

```
ResultSet rset = stmt.executeQuery
        ("select DATECOL, LONGCOL, NUMBERCOL from TABLE");

while rset.next()
{
   //get the date
   java.sql.Date date = rset.getDate(1);

   // access the stream data and discard it with close()
   InputStream is = rset.getAsciiStream(2);
   is.close();

   // get the number column data
   int n = rset.getInt(3);
}
```

LOB および外部ファイルのストリーム

ラージ・オブジェクト（LOB）という用語は、データベース表に直接格納するには大きすぎるデータ項目を指します。一方、ロケータはデータベース表に格納されて、実際のデータの場所を指します。外部ファイル（バイナリ・ファイルまたは BFILE）も同様に管理されます。JDBC ドライバは、ストリームの使用によってこれらの型をサポートできます。

- BLOB（非構造化バイナリ・データ）
- CLOB（文字データ）
- BFILE（外部ファイル）

LOB と BFILE は、この章で説明した他のストリーム・データとは動作が異なります。ドライバは、データを Java ストリームとして、サーバーとクライアント間で転送します。ただし、多くの Java ストリームとは異なり、データを表すロケータは表に格納されます。このため、接続が有効な間はデータにいつでもアクセスできます。

BLOB と CLOB のストリーム

問合せにより、1 つ以上の CLOB または BLOB 列を選択すると、JDBC ドライバはロケータが指すデータをクライアントに転送します。ドライバは、Java ストリームとして転送を行います。JDBC より取得した CLOB または BLOB データを操作する場合、Oracle の拡張機能クラス oracle.sql.BLOB および oracle.sql.CLOB 内のメソッドを使用します。これらのクラスは、CLOB または BLOB から入力ストリーム内への読込み、出力ストリームから CLOB または BLOB 内への書込み、CLOB または BLOB の長さの決定および CLOB または BLOB のクローズなどの機能を提供します。

ストリーム CLOB および BLOB データの使用方法の詳細は、7-6 ページの「BLOB および CLOB データの読込みと書込み」を参照してください。

重要： JDBC 2.0 の仕様では、PreparedStatement のメソッド setBinaryStream() および setObject() を使用して、ストリーム値を BLOB として入力でき、また、PreparedStatement のメソッド setAsciiStream()、setUnicodeStream()、setCharacterStream() および setObject() を使用して、ストリーム値を CLOB として入力できると規定されています。これにより、LOB ロケータを通さず、直接 LOB データ自体にアクセスできます。

Oracle JDBC ドライバの実装では、この機能はリリース 8.1.6 以降のデータベースおよび 8.1.6 以降の JDBC OCI ドライバを使用した構成でのみサポートされます。**データの破損が生じる可能性があるため、他の構成でこの機能を使用しないでください。**

ストリーム BFILE

外部ファイルである BFILE は、データベースの外部で、データ・サーバーのファイル・システム上のいずれかの場所にあるファイルに格納する場合にロケータを使用します。ロケータは、ファイルが実際に格納された場所を指します。

問合せにより、1 つ以上の BFILE 列を選択すると、JDBC ドライバはロケータが指すファイルをクライアントに転送します。転送は、Java ストリームの形式で行われます。JDBC より取得した BFILE データを操作する場合、Oracle の拡張機能クラス oracle.sql.BFILE 内のメソッドを使用します。このクラスは、BFILE から入力ストリーム内への読込み、出力ストリームから BFILE 内への書込み、BFILE の長さの決定および BFILE のクローズなどの機能を提供します。

ストリーム BFILE データの使用方法の詳細は、7-21 ページの「BFILE データの読込み」を参照してください。

ストリームのクローズ

ストリームから取得したデータは、ストリームの close() メソッドをコールすることにより廃棄できます。また、結果セットまたは接続オブジェクトを閉じることによってストリームのクローズおよび廃棄を実行できます。データ・ストリームに実行する close() メソッドの詳細は、3-27 ページの「ストリーム・データ列のバイパス」を参照してください。データ・ストリームを誤ってクローズして廃棄することを回避する方法については、3-29 ページの「ストリーム・データを使用する際の注意」を参照してください。

ストリームに関する注意

この項では、ストリームの使用に関して、次に示すいくつかの重要な注意事項を説明します。

- ストリーム・データを使用する際の注意
- setBytes() と setString() への制限を回避するためのストリームの使用方法
- ストリームと行のプリフェッチ

ストリーム・データを使用する際の注意

この項では、誤ってストリーム・データを廃棄または喪失することがないよう、事前に注意する必要がある点を説明します。カレント・ストリームの読込み以外の、データベースと通信する任意の JDBC 操作を実行すると、ドライバは自動的にストリーム・データを廃棄します。2 つの共通する注意事項について説明します。

- ストリーム・データはアクセス後に使用します。

  データ・ストリームを含む列からデータをリカバリするには、列の get では十分ではありません。列の内容をただちに処理する必要があります。そうしないと、次の列の取得時にその内容は廃棄されます。

- SELECT のリスト順にストリーム列をコールします。

  問合せを使用して複数列を選択すると、データベースは各行を、列を表すバイト・セットとして、SELECT で指定された順序で送信します。列の 1 つにストリーム・データが含まれる場合は、データベースは、次の列を処理する前にデータ・ストリーム全体を送信します。

  SELECT リストの順序を使用せずにデータへアクセスする場合は、ストリーム・データを失う可能性があります。つまり、ストリーム・データ列をバイパスして次の列のデータにアクセスすると、ストリーム・データは失われます。たとえば、ストリーム・データ列からデータを読み取る前に、NUMBER 列のデータにアクセスしようとすると、JDBC ドライバはストリーム・データを読み取った後、自動的にそのデータを廃棄します。LONG 列に大量のデータが格納されている場合、これは非常に非効率的です。

  LONG 列に後でアクセスしようとしても、データは使用できず、ドライバは「Stream Closed」エラーを戻します。

次の例では、2 番目の点について例証します。

```
ResultSet rset = stmt.executeQuery
        ("select DATECOL, LONGCOL, NUMBERCOL from TABLE");
while rset.next()
{
   int n = rset.getInt(3);  // This discards the streaming data
   InputStream is = rset.getAsciiStream(2);
                            // Raises an error: stream closed.
}
```

ストリームを取得しても、NUMBER 列の取得前に使用しないと、ストリームは自動的にクローズします。

```
ResultSet rset = stmt.executeQuery
                ("select DATECOL, LONGCOL, NUMBERCOL from TABLE");
while rset.next()
{
   InputStream is = rset.getAsciiStream(2); // Get the stream
   int n = rset.getInt(3);
   // Discards streaming data and closes the stream
}
int c = is.read(); // c is -1: no more characters to read-stream closed
```

setBytes() と setString() への制限を回避するためのストリームの使用方法

PreparedStatement クラスの setBytes() メソッドを使用してバインドできる配列の最大サイズと、setString() メソッドを使用してバインドできる文字列のサイズには制限があります。

使用するサーバーのバージョンによって異なるこの制限を超える場合は、かわりに setBinaryStream() または setCharacterStream() を使用してください。

Oracle8 データベースに接続する場合は、setBytes() の制限は 2000 バイトで（Oracle8 での RAW の最大サイズ）、setString() の制限は 4000 バイトです（Oracle8 での VARCHAR2 の最大サイズ）。

Oracle7 データベースに接続する場合は、setBytes() の制限は 255 バイトで（Oracle7 での RAW の最大サイズ）、setString() の制限は 2000 バイトです（Oracle7 での VARCHAR2 の最大サイズ）。

8.1.6 以降の Oracle JDBC ドライバでは、setBytes() または setString() を使用しているときに、制限値を超えてもエラーが発生するとは限りませんが、次のエラーが表示されることがあります。

```
ORA-17070: データ・サイズがこの型の最大サイズを超えています。
```

将来のバージョンの Oracle ドライバでは、データ長が前述の制限を超えた場合にエラーが発生します。

注意： この説明は、PL/SQL ではなく、SQL のバインドに関係します。

ストリームと行のプリフェッチ

JDBC ドライバがデータ・ストリームを含む列に遭遇すると、行のプリフェッチは 1 に再設定されます。

行のプリフェッチは、データベースにアクセスするたびに複数行のデータを取り出すことができる Oracle のパフォーマンス強化です。詳細は、12-19 ページの「Oracle 行プリフェッチ」を参照してください。

JDBC プログラムでのストアド・プロシージャ・コール

この項では、Oracle JDBC ドライバが次の種類のストアド・プロシージャをサポートする方法について説明します。

- PL/SQL ストアド・プロシージャ
- Java ストアド・プロシージャ

PL/SQL ストアド・プロシージャ

Oracle JDBC ドライバは、PL/SQL ストアド・プロシージャおよび無名ブロックの実行をサポートします。このドライバは、SQL92 エスケープ構文と Oracle PL/SQL ブロック構文の両方をサポートします。次の PL/SQL コールは、任意の Oracle JDBC ドライバで使用できます。

```
// SQL92 syntax
CallableStatement cs1 = conn.prepareCall
                      ( "{call proc (?,?)}" ) ; // stored proc
CallableStatement cs2 = conn.prepareCall
                      ( "{? = call func (?,?)}" ) ; // stored func
// Oracle PL/SQL block syntax
CallableStatement cs3 = conn.prepareCall
                      ( "begin proc (?,?); end;" ) ; // stored proc
CallableStatement cs4 = conn.prepareCall
                      ( "begin ? := func(?,?); end;" ) ; // stored func
```

Oracle 構文の使用例として、ここでは、ストアド・ファンクションを作成する PL/SQL コードの一部を使用します。PL/SQL ファンクションは、文字列を取得し、それに接尾辞を連結します。

```
create or replace function foo (val1 char)
return char as
begin
   return val1 || 'suffix';
end;
```

JDBC プログラム内の起動コールは、次のようになります。

```
Connection conn = DriverManager.getConnection
                  ("jdbc:oracle:oci8:@<hoststring>", "scott", "tiger");

CallableStatement cs = conn.prepareCall ("begin ? := foo(?); end;");
cs.registerOutParameter(1,Types.CHAR);
cs.setString(2, "aa");
cs.executeUpdate();
String result = cs.getString(1);
```

PL/SQL ストアド・プロシージャおよびファンクションをコールする、SQL92 構文および Oracle PL/SQL ブロック構文の完全なサンプル・アプリケーションについては、20-5 ページの「PL/SQL ストアド・プロシージャのコール : PLSQLExample.java」および 20-7 ページの「PL/SQL ブロックでのプロシージャの実行 : PLSQL.java」を参照してください。

Java ストアド・プロシージャ

JDBC を使用して、SQL および PL/SQL エンジン経由で Java ストアド・プロシージャを起動できます。Java ストアド・プロシージャをコールするための構文は、正しく公開されている（つまり、Oracle データ・ディクショナリに公開するために、コール仕様が書き込まれている）ことを仮定すれば、PL/SQL ストアド・プロシージャをコールするための構文と同じです。Java ストアド・プロシージャの記述、公開および使用方法の詳細は、『Oracle9*i* Java Stored Procedures Developer's Guide 』を参照してください。

SQL 例外の処理

エラー状況を処理するために、Oracle JDBC ドライバは SQL 例外を生成し、クラス `java.sql.SQLException` またはサブクラスのインスタンスを作成します。エラーは JDBC ドライバまたはデータベース（RDBMS）自体のいずれかで発生する可能性があります。表示されるメッセージには、エラーおよびエラーを発行したメソッドの説明が含まれます。ランタイム情報が追加されることもあります。

基本例外処理には、エラー・メッセージの取出し、エラー・コードの取出し、SQL 状態の取出しおよびスタック・トレースの出力が含まれます。`SQLException` クラスには、使用可能な場合にこのようなすべての情報を取り出す機能があります。

JDBC ドライバで発生したエラーは、付録 B「JDBC エラー・メッセージ」にそれぞれの `ORA` 番号とともにリストされています。

RDBMS で発生するエラーについては、『Oracle9*i* データベース・エラー・メッセージ』リファレンスで説明します。

エラー情報の取出し

次の `SQLException` メソッドで、基本エラー情報を取り出すことができます。

- `getMessage()`

  JDBC ドライバで発生したエラーの場合は、このメソッドによって接頭辞なしでそのエラー・メッセージが戻されます。RDBMS で発生したエラーの場合は、対応する `ORA` 番号の接頭辞を付けたエラー・メッセージが戻されます。

- `getErrorCode()`

  JDBC ドライバまたは RDBMS のいずれかで発生したエラーの場合は、このメソッドによって 5 桁の `ORA` 番号が戻されます。

- `getSQLState()`

  JDBC ドライバで発生したエラーの場合は、有用な情報は戻されません。RDBMS で発生したエラーの場合は、このメソッドによって SQL 状態を示す 5 桁のコードが戻されます。コードでは NULL データを処理できるようにしておいてください。

次の例は getMessage() コールからの情報を出力します。

```
catch(SQLException e)
{
   System.out.println("exception: " + e.getMessage());
}
```

これは JDBC ドライバで発生したエラーに対して、次のような情報を出力します。

```
exception: Invalid column type
```

(getErrorCode() コールで ORA 番号を取得できますが、JDBC ドライバで発生したエラーに ORA 番号メッセージ接頭語はありません。)

注意： Oracle がサポートしている代替言語とキャラクタ・セットでエラー・メッセージ・テキストを使用できます。

スタック・トレースの出力

SQLException クラスには、スタック・トレースを出力するための次のメソッドがあります。

- printStackTrace()

  このメソッドは、発生可能なオブジェクトのスタック・トレースを標準エラー・ストリームに出力します。出力のために、java.io.PrintStream オブジェクトまたは java.io.PrintWriter オブジェクトも指定できます。

次のコード・フラグメントは、SQL の例外を捕捉してスタック・トレースを表示する方法を示しています。

```
try { <some code> }
catch(SQLException e) { e.printStackTrace (); }
```

JDBC ドライバでエラーを処理する方法を説明するために、次のコードでは不適切な列索引を使用していると仮定します。

```
// Iterate through the result and print the employee names
// of the code

try {
  while (rset.next ())
      System.out.println (rset.getString (5));  // incorrect column index
}
catch(SQLException e) { e.printStackTrace (); }
```

列索引が不適切であると仮定すると、このプログラムの実行によって次のエラー・テキストが生成されます。

```
java.sql.SQLException: Invalid column index
at oracle.jdbc.dbaccess.DBError.check_error(DBError.java:235)
at oracle.jdbc.OracleStatement.prepare_for_new_get(OracleStatemen
t.java:1560)
at oracle.jdbc.OracleStatement.getStringValue(OracleStatement.jav
a:1653)
at oracle.jdbc.OracleResultSet.getString(OracleResultSet.java:175
)
at Employee.main(Employee.java:41)
```

4

JDBC 2.0 サポートの概要

Oracle JDBC の主な特長は、JDBC 2.0 機能です。JDBC 2.0 機能には、以前はサポートされていなかった新しい機能と、Oracle 拡張機能によって以前にもサポートされていた標準機能があります。

この章では、特に JDK 1.2.x 環境と JDK 1.1.x 環境間でのサポートの違いを中心に、Oracle JDBC ドライバでの JDBC 2.0 のサポートの概要を示します。次の項目が含まれます。

概要

Oracle JDBC ドライバは、JDBC 2.0 仕様に準拠しています。構造化オブジェクト、オブジェクト参照、配列および LOB などの oracle.jdbc2 パッケージ内の Oracle 拡張機能を通じて以前に実装されていた JDBC 2.0 機能は、現在 JDK 1.2 の標準 java.sql パッケージを通じて実装されています。

JDK 1.1.x 環境を使用している場合は、引き続き oracle.jdbc2 パッケージを使用できます。オブジェクトを Oracle 型にキャストすることによって、JDK 1.1.x でも接続オブジェクト、文オブジェクト、結果セット・オブジェクトおよびデータベース・メタデータ・オブジェクトで、JDBC 2.0 機能を使用できます。

さらに、JDK 1.2.x 環境または JDK 1.1.x 環境では（JDBC 2.0 Standard Extension API とも呼ばれる）JDBC 2.0 Optional Package の機能を使用できます。接続プーリングや分散トランザクションなどを含むこれらの機能は、標準 javax.sql パッケージを通じてサポートされます。このパッケージとそのインタフェースを実装するクラスは、JDK 1.2.x または JDK 1.1.x のどちらの環境でも JDBC クラス ZIP ファイルに含まれます。

JDBC 2.0 サポート : JDK 1.2.x と JDK 1.1.x

標準 JDBC 2.0 機能のサポートは、JDK 1.2.x または JDK 1.1.x のどちらを使用しているかによって異なります。次の 3 点を検討してください。

- オブジェクト、配列および LOB などのデータ型サポート。このサポートは、JDK 1.2.x では標準 java.sql パッケージを通じて、JDK 1.1.x では Oracle 拡張機能 oracle.jdbc2 パッケージを通じて処理されます。
- 結果セット拡張およびバッチ更新などの標準機能サポート。このサポートは、JDK 1.2.x では Connection、ResultSet および PreparedStatement などの標準クラスを通じて処理されますが、JDK 1.1.x では Oracle 固有機能が要求されます。
- データ・ソース、接続プーリングおよび分散トランザクションなどの（JDBC2.0 Standard Extension API とも呼ばれる）JDBC 2.0 Optional Package の拡張機能サポート。JDK 1.2.x または JDK 1.1.x のどちらでも同じサポートと機能が提供されます。

この項では、JDBC 2.0 で使用可能なパフォーマンス拡張、Oracle 拡張機能としても使用可能なバッチ更新およびフェッチ・サイズについても説明し、最後に JDK 1.1.x から JDK 1.2.x への移行に関して簡単に説明します。

データ型のサポート

Oracle JDBC は、標準 java.sql パッケージ内のインタフェースの実装による標準 JDBC 2.0 機能を含む、JDK 1.2.x を完全にサポートしています。これらのインタフェースは、oracle.sql パッケージと oracle.jdbc パッケージ内のクラスによって、適宜実装されます。

classes12.zip を使用する場合は、JDK 1.2.x の JDBC 2.0 機能について特別なインポートは必要ありません。次のインポートのみが必要です。この 2 つのインポートは JDBC 2.0 機能を使用しない場合でも必要です。

```
import java.sql.*;
import oracle.sql.*;
```

JDK 1.1.x では、JDBC 2.0 機能はサポートされていません。ただし、Oracle は classes111.zip を使用して、JDK 1.1.x で JDBC 2.0 データ型の有効なサブセットを使用できる拡張要素を提供しています。これらの拡張要素は、データベース・オブジェクト、オブジェクト参照、配列および LOB をサポートします。

パッケージ oracle.jdbc2 は、classes111.zip に含まれます。このパッケージは、SQL3 および拡張データ型のために、JDK 1.2.x で標準となった JDBC 2.0 関連インタフェースを疑似実行するインタフェースを提供します。oracle.jdbc2 内のインタフェースは、JDK 1.1.x 環境では oracle.sql パッケージ内のクラスによって適宜実装されます。

JDK 1.1.x では、JDBC 2.0 データ型のために次のインポートが必要です。

```
import java.sql.*;
import oracle.jdbc2.*;
import oracle.sql.*;
```

標準機能のサポート

（classes12.zip の JDBC クラスを使用する）JDK 1.2.x 環境では、スクロール可能結果セット、更新可能結果セットおよびバッチ更新などの JDBC 2.0 機能は、標準の JDBC 2.0 インタフェースによって指定されるメソッドを通じてサポートされます。したがって、JDK 1.2.x では、Connection、DatabaseMetaData、ResultSetMetaData、Statement、PreparedStatement、CallableStatement および ResultSet などの標準クラスを使用して、前述の機能を使用できます。

（classes111.zip の JDBC クラスを使用する）JDK 1.1.x 環境では、Oracle JDBC は Oracle 拡張機能として前述の JDBC 2.0 機能をサポートします。この機能を使用するには、オブジェクトを次の Oracle 型にキャストする必要があります。

- OracleConnection
- OracleDatabaseMetaData
- OracleResultSetMetaData

- `OracleStatement`
- `OraclePreparedStatement`
- `OracleCallableStatement`
- `OracleResultSet`

たとえば、JDBC 2.0 結果セット拡張を使用するには、次を実行する必要があります。

- `OracleResultSet` 型としてスクロール可能結果セットまたは更新可能結果セットを明示的に入力するか、キャストします。
- 文オブジェクトを作成するために接続オブジェクトが必要になるたびに、`OracleConnection` 型として接続オブジェクトを明示的に入力するか、キャストします。この文オブジェクトによって、スクロール可能結果セットまたは更新可能結果セットが作成されます。

また、`OracleStatement`、`OraclePreparedStatement` または `OracleCallableStatement` に文オブジェクトをキャストして、`OracleDatabaseMetaData` にデータベース・メタデータ・オブジェクトをキャストする必要が生じることがあります。これは、11-28 ページの「結果セット拡張用新規メソッドのサマリー」で説明する JDBC 2.0 文またはデータベース・メタデータ・メソッドを使用する場合に必要になります。

拡張機能のサポート

データ・ソース、接続プーリングおよび分散トランザクションなどの（JDBC2.0 Standard Extension API とも呼ばれる）JDBC 2.0 Optional Package の機能は、JDK 1.2.x 環境または JDK 1.1.x 環境のどちらでも同等にサポートされています。

標準 `javax.sql` パッケージとそのインタフェースを実装するクラスは、どちらの環境でも JDBC クラス ZIP ファイルに含まれます。

JDBC2.0 Standard Extension API とオラクル社独自のパフォーマンス強化 API

JDBC 2.0 では、以前は Oracle 拡張機能として提供されていた次の 2 つのパフォーマンス拡張を使用できます。

- バッチ更新
- フェッチ・サイズ / 行プリフェッチ

どちらの場合でも、標準モデルまたは Oracle モデルを使用するオプションがあります。ただし、これらの機能のために、単一のアプリケーション内で標準モデルと Oracle モデルを同時に使用しないでください。

詳細は、次の項を参照してください。

- 12-2 ページ「バッチ更新」
- 11-20 ページ「フェッチ・サイズ」
- 12-19 ページ「Oracle 行プリフェッチ」

JDK 1.1.x から JDK 1.2.x への移行

次に、JDK 1.1.x から JDK 1.2.x へのすべての移行要件を示します。

- 4-3 ページの「データ型のサポート」で説明したように、`oracle.jdbc2` パッケージのインポートを削除します。
- `oracle.jdbc2.*` インタフェースへの直接の参照を、標準 `java.sql.*` インタフェースへの参照に置換します。
- JDK 1.1.x では、`java.util.Dictionary` クラスを拡張する必要がある（Java 型への SQL 構造化オブジェクトのマッピングのための）型マップ・オブジェクトは、JDK 1.2.x では `java.util.Map` インタフェースを実装する必要があります。ただし、クラス `java.util.Hashtable` は、どちらの要件も満たしていることに注意してください。JDK 1.1.x で型マップのために `Hashtable` オブジェクトを使用していた場合は、変更は不要です。詳細は、8-12 ページの「型マップ・オブジェクトの作成と SQLData 実装のマッピング定義」を参照してください。

これらの点が作成しているコードに当てはまらない場合は、JDK 1.2.x で実行するためにコードの変更または再コンパイルを行う必要はありません。

JDBC 2.0 機能の概要

表 4-1 には、JDBC 2.0 機能の主な領域と、Oracle によるサポートの詳細を示すこのマニュアルの参照先をリストします。

表 4-1 JDBC 2.0 機能の主な領域

機能	コメントと参照
バッチ更新	以前にも Oracle 拡張機能としても使用可能でした。JDK 1.2.x または JDK 1.1.x のどちらでも標準バッチ更新モデルまたは Oracle モデルを使用できます。 詳細は、12-2 ページの「バッチ更新」を参照してください。
結果セット拡張（スクロール可能結果セットと更新可能結果セット）	JDK 1.1.x 環境の場合は、Oracle 拡張機能として使用可能です。 詳細は、第 11 章「結果セット拡張」を参照してください。
フェッチ・サイズ / 行プリフェッチ	JDBC 2.0 フェッチ・サイズ機能は、Oracle 拡張機能として JDK 1.1.x でも使用可能です。 また、JDK 1.2.x または JDK 1.1.x のどちらでも、Oracle 行プリフェッチを使用できます。この機能は、JDBC 2.0 フェッチ・サイズ機能とほぼ同じですが、JDBC 2.0 より早い時期に提供されました。 詳細は、11-20 ページの「フェッチ・サイズ」および 12-19 ページの「Oracle 行プリフェッチ」を参照してください。
データベース接続を指定し、取得するための JNDI（Java Naming and Directory Interface）の使用	この機能には、`javax.sql` パッケージ内の JDBC 2.0 Optional Package（JDBC 2.0 Standard Extension API）の一部であるデータ・ソースが必要です。これは JDK 1.2.x または JDK 1.1.x のどちらでも使用可能です。 詳細は、14-2 ページの「JNDI の Oracle データ・ソース・サポートに関する簡単な概要」および 14-8 ページの「データ・ソース・インスタンスの作成、JNDI での登録および接続」を参照してください。
接続プール（接続キャッシュのためのフレームワーク）	この機能には、`javax.sql` パッケージ内の JDBC 2.0 Optional Package（JDBC 2.0 Standard Extension API）が必要です。これは JDK 1.2.x または 1.1.x のどちらでも使用可能です。 詳細は、14-12 ページの「接続プーリング」を参照してください。
接続キャッシュ（サンプル Oracle 実装）	この機能には、`javax.sql` パッケージ内の JDBC 2.0 Optional Package（JDBC 2.0 Standard Extension API）が必要です。これは JDK 1.2.x または 1.1.x のどちらでも使用可能です。 詳細は、14-16 ページの「接続キャッシュ」を参照してください。

表 4-1 JDBC 2.0 機能の主な領域 （続き）

機能	コメントと参照
分散トランザクション /XA 機能	この機能には、javax.sql パッケージ内の JDBC 2.0 Optional Package（JDBC 2.0 Standard Extension API）が必要です。これは JDK 1.2.x または 1.1.x のどちらでも使用可能です。 詳細は、第 15 章「分散トランザクション」を参照してください。
様々な getXXX() メソッド	JDK 1.2.x と JDK 1.1.x ではどの getXXX() メソッドが Oracle 拡張機能であるか、および JDBC 2.0 との機能性の相違については、6-7 ページの「他の getXXX() メソッド」を参照してください。
様々な setXXX() メソッド	JDK 1.2.x と JDK 1.1.x ではどの setXXX() メソッドが Oracle 拡張機能であるか、および JDBC 2.0 との機能性の相違については、6-12 ページの「他の setXXX() メソッド」を参照してください。

注意： Oracle JDBC ドライバは、java.sql.Date タイムゾーン情報をサポートできないため、Calendar データ型をサポートしていません。setXXX() または getXXX() メソッド・コールの Date、Time および Timestamp に Calendar 型を入力しても無視されます。Calendar 型は、将来の Oracle リリースでサポートされます。

5

Oracle 拡張機能の概要

JDBC 標準への Oracle 拡張機能には、Oracle データ型にアクセスして操作したり、Oracle パフォーマンス拡張機能を使用できる Java パッケージおよびインタフェースなどがあります。標準 JDBC に比べて、Oracle 拡張機能ではデータを操作する方法の柔軟性が高くなります。この章では、標準 JDBC への Oracle 拡張機能に含まれるパッケージとクラスの概要を示します。また、この拡張機能がサポートする主な機能についても説明します。

次の項目が含まれます。

注意： この章では、主に型拡張機能について説明します。パフォーマンス拡張機能の詳細は、第 12 章「パフォーマンス拡張要素」で説明します。

Oracle 拡張機能の概要

Oracle は、JDBC ドライバの 2 つの実装を提供しています。1 つは Sun 社の JDK 1.2.x をサポートし、Sun JDBC 2.0 標準に準拠しています。もう 1 つは JDK 1.1.x をサポートし、Sun JDBC 1.22 標準に準拠しています。

標準機能に加えて、Oracle JDBC ドライバには Oracle 固有型拡張要素およびパフォーマンス拡張要素があります。

どちらの実装にも、次の Java パッケージがあります。

- `oracle.sql`（すべての Oracle 型拡張要素をサポートするクラス）
- `oracle.jdbc`（Oracle 型形式でのデータベース・アクセスおよび更新をサポートするクラス）

これらのパッケージに加えて、JDK 1.1.x の実装には次の Java パッケージがあります。このパッケージは、標準 `java.sql` パッケージ内の JDBC 2.0 インタフェースを疑似実行するインタフェースを提供することによって、一部の JDBC 2.0 機能をサポートします。

- `oracle.jdbc2`（標準 JDBC 2.0 インタフェースと等価のインタフェース）

（たとえば、`oracle.jdbc2.Struct` は、JDK 1.2 に存在する `java.sql.Struct` を疑似実行します。）

5-6 ページの「Oracle JDBC パッケージとクラス」で前述したパッケージとそのクラスについて詳しく説明します。

Oracle 拡張機能のサポート機能

JDBC への Oracle 拡張機能には、Oracle データベースを操作する能力を高めるいくつかの機能があります。たとえば、Oracle データ型、Oracle オブジェクトおよび固有スキーマ命名のサポートなどがあります。

Oracle データ型のサポート

Oracle JDBC 拡張機能の主要機能は、`oracle.sql` パッケージの型のサポートです。このパッケージには、すべての Oracle SQL データ型にマップするクラスが含まれます。これらのクラスは、実際の SQL データのラッパーとして動作します。この機能には、SQL データを操作する上で次の 2 つの重要な利点があります。

- SQL 形式のデータに直接アクセスすることにより、データを Java 形式に変換してからアクセスするよりも効率性が向上します。
- データへの数学的操作を SQL 形式で直接実行することにより、SQL と Java との間での変換中に生じる精度の低下を避けられます。

操作が完了して、情報を出力する際には、各 oracle.sql.* 型をサポートするクラスは、データを適切な Java 形式に変換するすべてのメソッドを保持します。これらの一般的な問題の詳細は、5-6 ページの「パッケージ oracle.sql」を参照してください。

特定の oracle.sql.* データ型クラスの詳細は、次の項を参照してください。

- SQL CHAR および SQL NCHAR データ型を含む oracle.sql.* 文字データ型については、5-27 ページの「Oracle 文字データ型のサポート」を参照してください。
- ROWID および REF CURSOR 型での oracle.sql.* データ型クラスについては、5-32 ページの「その他の Oracle 型拡張機能」を参照してください。
- BLOB、CLOB および BFILE での oracle.sql.* データ型サポートについては、第 7 章「LOB と BFILE の操作」を参照してください。
- データベース内の複合データ構造（Oracle オブジェクト）での oracle.sql.* データ型サポートについては、第 8 章「Oracle オブジェクト型の操作」を参照してください。
- オブジェクト参照での oracle.sql.* データ型サポートについては、第 9 章「Oracle オブジェクト参照の操作」を参照してください。
- コレクション（VARRAY および NESTED TABLE）での oracle.sql.* データ型サポートについては、第 10 章「Oracle コレクションの操作」を参照してください。

Oracle オブジェクトのサポート

Oracle JDBC は、データベース内での構造化オブジェクトの使用をサポートしています。オブジェクトのデータ型は、ネストした属性を持つユーザー定義型です。たとえば、ユーザーのアプリケーションで Employee オブジェクト型が定義可能な場合、各 Employee オブジェクトは、firstname 属性（文字列）、lastname 属性（別の文字列）および employeenumber 属性（整数）を保持します。

Oracle の JDBC 実装は、Oracle オブジェクト・データ型をサポートします。Java アプリケーションで Oracle オブジェクト・データ型を使用する場合、次の点を考慮する必要があります。

- Oracle オブジェクト・データ型と Java クラス間のマップ方法
- Oracle オブジェクトの属性を対応する Java オブジェクトに格納する方法（標準 Java 型と oracle.sql.* 型のどちらでも格納可能）
- SQL と Java 形式間で属性データを変換する方法
- データへのアクセス方法

Oracle オブジェクトは、弱い型指定の java.sql.Struct または oracle.sql.STRUCT 型や、強い型指定のカスタマイズされたクラスのいずれかにマップできます。これらの強い型指定はカスタム Java クラスと呼ばれ、標準 java.sql.SQLData インタフェースまたは Oracle 拡張機能 oracle.sql.ORAData インタフェースのいずれかを実装する必要があります。（これらのインタフェースの詳細は、第 8 章「Oracle オブジェクト型の操作」で説明

します。）各インタフェースは、SQL と Java との間でデータ変換を行うメソッドを指定します。

注意： CustomDatum インタフェースにかわって ORAData インタフェースが導入されました。前者のインタフェースは Oracle9*i* では廃止されましたが、下位互換性を保持するためにサポートされています。

使用する Oracle オブジェクトに対応するカスタム Java クラスを作成する場合、Oracle9*i* JPublisher ユーティリティを使用してクラスを作成することをお薦めします。クラスを作成するには、データの格納方法に応じた属性を定義する必要があります。JPublisher は、コマンドライン・オプションで、透過的にこの作業を実行し、SQLData または ORAData 実装のいずれかを生成できます。

SQLData 実装では、型マップは、Oracle オブジェクト・データ型と Java クラス間の対応関係を定義します。型マップは、各 Oracle オブジェクト・データ型に対応する Java クラスを指定する、特別な Java クラスです。Oracle JDBC はこれらの型マップを使用して、結果セットから Oracle オブジェクト・データを取り出す際に、どの Java クラスのインスタンス化および移入を行うかを決定します。

注意： 移植性が重要ではない場合は、SQLData インタフェースのかわりに ORAData インタフェースを使用することをお薦めします。ORAData は、Oracle Java プラットフォームが提供するその他の機能とともに、より容易にかつ柔軟に動作します。

JPublisher は、カスタム Java クラスの getXXX() メソッドを自動的に定義します。このメソッドはデータを Java アプリケーションに取り込みます。JPublisher ユーティリティの詳細は、『Oracle9*i* JPublisher ユーザーズ・ガイド』を参照してください。

第 8 章「Oracle オブジェクト型の操作」に、Oracle JDBC による Oracle オブジェクトのサポートが説明されています。

スキーマの命名サポート

Oracle JDBC クラスには、完全修飾スキーマ名の受入れおよび復帰を行う機能があります。完全修飾スキーマ名はすべて、次の構文になります。

{ [*schema_name*] . } [*sql_type_name*]

schema_name にはスキーマ名を、*sql_type_name* にはオブジェクトの SQL 型名を指定します。*schema_name* および *sql_type_name* は、ドット（.）で区切られることに注意してください。

JDBC でオブジェクト型を定義する場合、その完全修飾名（つまり、スキーマ名と SQL 型名）を使用します。型名がカレント・ネームスペース内（つまりカレント・スキーマ内）にある場合、スキーマ名を入力する必要はありません。スキーマは、次の規則に従って命名されます。

- スキーマ名と型名は、どちらも引用符で囲んでも囲まなくてもかまいません。ただし、CORPORATE.EMPLOYEE のように、SQL 型名にドットが含まれる場合、型名を引用符で囲む必要があります。
- JDBC ドライバは、オブジェクト名内の引用符で囲まれていない最初のドットを検索して、ドットの前の文字列をスキーマ名として使用し、ドットの後の文字列を型名として使用します。ドットが見つからない場合、JDBC ドライバはカレント・スキーマをデフォルトとします。つまり、オブジェクト型名がカレント・スキーマに属している場合、（スキーマを指定せずに）型名のみを指定できます。これが、型名にドットが含まれない場合に型名を引用符で囲む理由です。

  たとえば、ユーザー Scott が person.address という型を作成し、自分のセッション内でそれを使用するとします。Scott は、スキーマ名をスキップして、person.address で JDBC ドライバに渡します。この場合、person.address を引用符で囲まないと、ドットが検出されて、JDBC ドライバが person をスキーマ名として、address を型名として誤って解釈してしまいます。
- JDBC は、オブジェクト型名の文字列をデータベースにそのまま渡します。つまり、JDBC ドライバは、文字列が引用符で囲まれていてもその大 / 小文字を変更しません。

  たとえば、ScOtT.PersonType が JDBC ドライバにオブジェクト型名として渡されると、JDBC ドライバはその文字列をそのままデータベースに渡します。別の例として、型名の文字列内に空白文字が含まれている場合、JDBC ドライバは空白文字を削除しません。

OCI 拡張機能

次の OCI ドライバに固有な情報については、第 16 章「JDBC OCI 拡張機能」を参照してください。

- OCI ドライバ接続プーリング
- プロキシ接続を介する中間層認証
- OCI ドライバの透過的アプリケーション・フェイルオーバー
- OCI HeteroRM XA
- PL/SQL 索引付き表へのアクセス

Oracle JDBC パッケージとクラス

この項では、Oracle JDBC 拡張要素をサポートする Java パッケージ、およびこれらのパッケージの主要クラスについて説明します。

- パッケージ oracle.sql
- パッケージ oracle.jdbc
- パッケージ oracle.jdbc2（JDK 1.1.x の場合のみ）

この項で説明するすべてのクラスの詳細は、Oracle JDBC Javadoc を参照してください。

パッケージ oracle.sql

`oracle.sql` パッケージは、SQL 形式でデータへの直接アクセスをサポートしています。このパッケージは、主に SQL データ型への Java マッピングを提供するクラスから構成されます。

実質的に、このクラスは実際の SQL データの Java ラッパーとして機能します。`oracle.sql.*` オブジェクト内のデータは SQL 形式のままであるため、情報が失われることはありません。SQL 基本型の場合、これらのクラスは SQL データを単にラップします。SQL 構造化型（オブジェクトおよび配列）の場合、これらのクラスは変換方法や構造の詳細などの追加情報を提供します。

`oracle.sql.*` データ型の各クラスは、すべてのデータ型に共通する機能をカプセル化したスーパークラスの `oracle.sql.Datum` を拡張します。その中には、JDBC 2.0 準拠のデータ型に対応したクラスもあります。これらのクラスは、表 5-1 に示すように、`oracle.sql.Datum` クラスを拡張すると同時に、`java.sql` パッケージ（JDK 1.1.x の `oracle.jdbc2`）の標準 JDBC 2.0 インタフェースを実装します。

oracle.sql パッケージのクラス

表 5-1 に、oracle.sql データ型クラスおよび対応する Oracle SQL 型のリストを示します。

表 5-1 Oracle データ型クラス

Java クラス	実装される Oracle SQL 型とインタフェース
oracle.sql.STRUCT	STRUCT（オブジェクト）は java.sql.Struct（JDK 1.1.x 環境では oracle.jdbc2.Struct）を実装します。
oracle.sql.REF	REF（オブジェクト参照）は java.sql.Ref（JDK 1.1.x 環境では oracle.jdbc2.Ref）を実装します。
oracle.sql.ARRAY	VARRAY または NESTED TABLE（コレクション）は java.sql.Array（JDK 1.1.x 環境では oracle.jdbc2.Array）を実装します。
oracle.sql.BLOB	BLOB（バイナリ・ラージ・オブジェクト）は java.sql.Blob（JDK 1.1.x 環境では oracle.jdbc2.Blob）を実装します。
oracle.sql.CLOB	CLOB（キャラクタ・ラージ・オブジェクト）およびグローバリゼーション・サポート NCLOB データ型はいずれも、java.sql.Clob（JDK 1.1.x 環境では oracle.jdbc2.Clob）を実装します。
oracle.sql.BFILE	BFILE（外部ファイル）
oracle.sql.CHAR	CHAR、NCHAR、VARCHAR2、NVARCHAR2。
oracle.sql.DATE	DATE
oracle.sql.TIMESTAMP	TIMESTAMP
oracle.sql.TIMESTAMPTZ	TIMESTAMPTZ (タイム・ゾーンを指定したタイムスタンプ)
oracle.sql.TIMESTAMPLTZ	TIMESTAMPLTZ (ローカル・タイム・ゾーンを指定したタイムスタンプ)
oracle.sql.NUMBER	NUMBER
oracle.sql.RAW	RAW
oracle.sql.ROWID	ROWID（行識別子）
oracle.sql.OPAQUE	OPAQUE

この章の後半で、各クラスについてより詳細に説明します。Oracle 拡張型（STRUCT、REF、ARRAY、BLOB、CLOB、BFILE および ROWID）の詳細は、次の章または項目で説明します。

- 5-27 ページ「Oracle 文字データ型のサポート」
- 5-32 ページ「その他の Oracle 型拡張機能」
- 第 7 章「LOB と BFILE の操作」
- 第 8 章「Oracle オブジェクト型の操作」
- 第 9 章「Oracle オブジェクト参照の操作」
- 第 10 章「Oracle コレクションの操作」

注意：

- 結果セットまたはコール可能なオブジェクトから、Java 型に対応する oracle.sql.* 型にデータを取り込む場合の詳細は、第 6 章「Oracle データへのアクセスと操作」を参照してください。
- LONG、LONG RAW SQL 型と REF CURSOR 型カテゴリには、oracle.sql.* クラスは含まれません。これらの型では、標準 JDBC 機能を使用してください。たとえば、LONG または LONG RAW データを、標準 JDBC 結果セットおよびコール可能文メソッドである getBinaryStream() および getCharacterStream() を使用して入力ストリームとして取り出します。REF CURSOR 型の場合は、getCursor() メソッドを使用してください。

データ型クラスに加え、oracle.sql パッケージには、主にオブジェクトとコレクションで使用するための次のサポート・クラスおよびインタフェースが含まれます。

- oracle.sql.ArrayDescriptor クラス：oracle.sql.ARRAY オブジェクトの作成に使用します。配列の SQL 型を記述します。（10-9 ページの「ARRAY オブジェクトと記述子の作成」を参照してください。）
- oracle.sql.StructDescriptor クラス：oracle.sql.STRUCT オブジェクトの作成に使用します。このオブジェクトは、データベース内の Oracle オブジェクトへのデフォルト・マッピングとして使用できます。（8-5 ページの「STRUCT オブジェクトと記述子の作成」を参照してください。）
- oracle.sql.ORAData および oracle.sql.ORADataFactory インタフェース：Oracle オブジェクト・サポートの Oracle ORAData シナリオを実装する Java クラスで使用します。（利用可能な別のシナリオとして、JDBC 標準の SQLData の実装があります。）ORAData の詳細は、8-21 ページの「ORAData インタフェース」を参照してください。
- oracle.sql.OpaqueDescriptor クラス：oracle.sql.OPAQUE クラスのインスタンスにメタデータを取得するのに使用します。

一般的な oracle.sql* データ型のサポート

各 Oracle データ型クラスは、特に次の機能をサポートします。

- 1 つ以上のコンストラクタ。通常、ロー・バイトを入力とするコンストラクタ 1 つと、Java 型を入力とするコンストラクタ 1 つです。
- SQL データ用の Java バイト配列であるデータ記憶域。
- JDBC がデータベースからデータを受け取ったロー形式でバイト配列として SQL データを戻す `getBytes()` メソッド。
- データを、JDBC 仕様の定義に従って、対応する Java クラスのオブジェクトに変換する `toJdbc()` メソッド。

  JDBC ドライバは、`ROWID` などの、JDBC 仕様の一部ではない Oracle 固有のデータ型を変換しません。ドライバは、対応する `oracle.sql.*` 形式でオブジェクトを戻します。たとえば、Oracle ROWID を `oracle.sql.ROWID` として戻します。
- SQL データを Java 型に変換するための適切な `xxxValue()` メソッド。
  例：`stringValue()`、`intValue()`、`booleanValue()`、`dateValue()`、`bigDecimalValue()`
- 補足的な変換。データ型の機能（データをストリームとして取り出す LOB クラスのメソッドやオブジェクト参照を使用してオブジェクト・データの取出しや設定を行う `REF` クラスのメソッドなど）に適した `getXXX()` および `setXXX()` メソッドを利用します。

これらのクラスの詳細は、Oracle JDBC Javadoc を参照してください。`oracle.sql.CHAR` クラスで文字データをサポートする方法の詳細は、5-28 ページの「クラス oracle.sql.CHAR」を参照してください。

クラス oracle.sql.STRUCT の概要

特定の Oracle オブジェクト型では、一般に SQL と Java 間でカスタム・マッピングを定義することをお薦めします。（`SQLData` カスタム Java クラスを使用する場合は、このマッピングは型マップで定義する必要があります。）

ただし、マッピングを定義しない場合は、オブジェクト型からのデータは `oracle.sql.STRUCT` クラスのインスタンス内で Java によってインスタンス化されます。

`STRUCT` クラスは、標準 JDBC 2.0 `java.sql.Struct` インタフェース（JDK 1.1.x の `oracle.jdbc2.Struct`）を実装し、`oracle.sql.Datum` クラスを拡張します。

データベース内では、Oracle はオブジェクト・データのロー・バイトを線形化された形式で格納します。`STRUCT` オブジェクトは、Oracle オブジェクトのロー・バイトのラッパーです。これには Oracle オブジェクトの SQL 型名と SQL 形式で属性値を保持する `oracle.sql.Datum` オブジェクトの値配列が含まれます。

STRUCT の属性は、`getOracleAttributes()` メソッドを使用する場合は `oracle.sql.Datum[]` オブジェクトとして、また `getAttributes()` メソッドを使用する場合は `java.lang.Object[]` オブジェクトとしてインスタンス化できます。

oracle.sql.* オブジェクトとして属性をインスタンス化すると、oracle.sql.* 形式のすべての利点を利用できます。

- oracle.sql.* 形式で oracle.sql.STRUCT データをインスタンス化すると、SQL 形式でデータが完全に保たれます。変換は実行されません。これは、データにアクセスしても表示する必要がない場合に便利です。
- Java アプリケーションでは自由度の高い方法でデータをリストアできます。

注意：

- 値配列の要素は、一般的な Datum 型の要素であっても、特定の属性に適する oracle.sql.* 型と対応付けられたデータを含みます。要素は、適切な oracle.sql.* 型にキャストできます。たとえば、STRUCT 内の CHAR データ属性は、oracle.sql.Datum としてインスタンス化されます。これを CHAR データとして使用するには、oracle.sql.CHAR 型にキャストする必要があります。
- STRUCT オブジェクトの値配列内のネストしたオブジェクトは、STRUCT のインスタンスとして JDBC ドライバによりインスタンス化されます。

手動で STRUCT オブジェクトを作成して、プリコンパイルされた SQL 文やコール可能文へ渡す場合もあります。これを行うには、StructDescriptor オブジェクトも作成する必要があります。

oracle.sql.STRUCT および StructDescriptor クラスを使用して Oracle オブジェクトを操作する方法の詳細は、8-3 ページの「Oracle オブジェクト用のデフォルト STRUCT クラスの使用方法」を参照してください。

クラス oracle.sql.REF の概要

oracle.sql.REF クラスは、Oracle オブジェクト参照をサポートする一般クラスです。このクラスは、すべての oracle.sql.* データ型のクラスと同様に、oracle.sql.Datum のサブクラスです。このクラスは、標準 JDBC 2.0 java.sql.Ref インタフェース（JDK 1.1.x 環境では oracle.jdbc2.Ref）を実装します。

REF クラスには、オブジェクト参照を取り出して渡すためのメソッドがあります。ただし、オブジェクト参照の選択によって、オブジェクトへのポインタのみが取り出されることに注意してください。これによってオブジェクト自体がインスタンス化されることはありません。ただし、REF クラスにはオブジェクト・データを取り出して渡すためのメソッドも含まれています。

JDBC アプリケーションでは REF オブジェクトを作成できません。つまり、データベースから既存の REF オブジェクトのみを取り出すことができます。

oracle.sql.REF クラスを使用して Oracle オブジェクト参照を操作する方法の詳細は、第 9 章「Oracle オブジェクト参照の操作」を参照してください。

クラス oracle.sql.ARRAY の概要

oracle.sql.ARRAY クラスは、Oracle コレクション、つまり VARRAY または NESTED TABLE のいずれかをサポートします。データベースから VARRAY または NESTED TABLE を選択すると、JDBC ドライバは ARRAY クラスのオブジェクトとしてインスタンス化します。データ構造は、どちらを選択した場合も同じです。oracle.sql.ARRAY クラスは、oracle.sql.Datum を拡張して、標準 JDBC 2.0 java.sql.Array インタフェース（JDK 1.1.x の oracle.jdbc2.Array）を実装します。

OraclePreparedStatement または OracleCallableStatement クラスの setARRAY() メソッドを使用して、プリコンパイルされた SQL 文への入力パラメータとして配列を渡すことができます。同様に、ARRAY オブジェクトを手動で作成して、プリコンパイルされた SQL 文またはコール可能文に渡し、データベースへの挿入などの操作を行う場合も考えられます。このような場合は、ArrayDescriptor オブジェクトを使用します。

oracle.sql.ARRAY および ArrayDescriptor クラスを使用して Oracle コレクションを操作する方法の詳細は、10-5 ページの「コレクション（配列）機能の概要」を参照してください。

クラス oracle.sql.BLOB、oracle.sql.CLOB、oracle.sql.BFILE の概要

BLOB および CLOB（まとめて LOB といいます）と BFILE（外部ファイル用）は、大きすぎてデータベース表に直接格納できないデータ項目に使用します。一方、データベース表はデータの実際の場所を指すロケータを格納します。

oracle.sql パッケージは、これらのデータ型を次のいくつかの方法でサポートします。

- BLOB は、ラージ非構造化バイナリ・データ項目を指し、oracle.sql.BLOB クラスによりサポートされます。
- CLOB は、ラージ固定幅文字データ項目（文字ごとにバイト定数を必要とする文字列）を指し、oracle.sql.CLOB クラスによりサポートされます。
- BFILE は、外部ファイル（オペレーティング・システム・ファイル）の内容を指し、oracle.sql.BFILE クラスによりサポートされます。

標準 SELECT 文を使用して、BLOB、CLOB または BFILE ロケータをデータベースから選択できます。ただし、受け取るのは、データ自体ではなく、ロケータのみであることに留意してください。データの取出しには、追加処理が必要です。

LOB および BFILE でロケータとデータにアクセスし、操作する方法は、第 7 章「LOB と BFILE の操作」を参照してください。

クラス oracle.sql.DATE、oracle.sql.NUMBER および oracle.sql.RAW

これらのクラスは、標準 JDBC の一部であるプリミティブ SQL データ型にマップされ、対応する JDBC Java 型間での変換を提供します。詳細は、Javadoc を参照してください。

クラス oracle.sql.TIMESTAMP、oracle.sql.TIMESTAMPTZ および oracle.sql.TIMESTAMPLTZ

Oracle9*i* JDBC ドライバでは、次のような日付 / 時刻データ型をサポートしています。

- タイムスタンプ（TS）
- タイム・ゾーンを指定したタイムスタンプ（TSTZ）
- ローカル・タイム・ゾーンを指定したタイムスタンプ（TSLTZ）

Oracle9*i* JDBC ドライバでは、DATE および日付 / 時刻データ型の間での変換が可能です。たとえば、DATE 値として TIMESTAMPTZ 列にアクセスできます。

Oracle9*i* JDBC ドライバは、業界で最も一般的なタイム・ゾーン名と、Sun 社の JDK で定義されたタイム・ゾーン名をサポートしています。タイム・ゾーンは、`java.util.Calendar` クラスを使用して指定します。

> **注意：** Oracle タイム・ゾーン名すべてが Sun 社の JDK に定義されているわけではないので、タイム・ゾーン・オブジェクトの作成には `TimeZone.getTimeZone` を使用しないでください。

次のコードは、JDK に定義されていないタイム・ゾーン名である `US/Pacific` に `TimeZone` オブジェクトと `Calendar` オブジェクトを作成する方法を示します。

```
TimeZone tz = TimeZone.getDefault();
tz.setID("US/Pacific");
GregorianCalendar gcal = new GregorianCalendar(tz);
```

次のクラスは、日付 / 時刻データ型用です。

- `oracle.sql.TIMESTAMP`
- `oracle.sql.TIMESTAMPTZ`
- `oracle.sql.TIMESTAMPLTZ`

`oracle.jdbc.OraclePreparedStatement` インタフェースから日付 / 時刻項目を設定するには、次のメソッドを使用してください。

- `setTIMESTAMP(int paramIdx,TIMESTAMP x)`
- `setTIMESTAMPTZ(int paramIdx,TIMESTAMPTZ x)`
- `setTIMESTAMPLTZ(int paramIdx,TIMESTAMPLTZ x)`

oracle.jdbc.OracleCallableStatement インタフェースから日付 / 時刻項目を取得するには、次のメソッドを使用してください。

- TIMESTAMP getTIMESTAMP (int paramIdx)
- TIMESTAMPTZ getTIMESTAMPTZ(int paramIdx)
- TIMESTAMPLTZ getTIMESTAMPLTZ(int paramIdx)

oracle.jdbc.OracleResultSet インタフェースから日付 / 時刻項目を取得するには、次のメソッドを使用してください。

- TIMESTAMP getTIMESTAMP(int paramIdx)
- TIMESTAMP getTIMESTAMP(java.lang.String colName)
- TIMESTAMPTZ getTIMESTAMPTZ(int paramIdx)
- TIMESTAMPTZ getTIMESTAMPTZ(java.lang.String colName)
- TIMESTAMPLTZ getTIMESTAMPLTZ(int paramIdx)
- TIMESTAMPLTZ getTIMESTAMPLTZ(java.lang.String colName)
- TIMESTAMPLTZ getTIMESTAMPLTZ(int paramIdx)

TIMESTAMPLTZ データにアクセスする前に、OracleConnection.setSessionTime() メソッドをコールしてセッション・タイム・ゾーンを設定します。このメソッドをコールすると、JDBC ドライバは接続のセッション・タイム・ゾーンを設定し、セッション・タイム・ゾーンを保存します。これにより、このセッション・タイム・ゾーンを使用して、JDBC を通じてアクセスした TIMESTAMPLTZ データを調整することができます。

クラス oracle.sql.ROWID の概要

このクラスは、データベース表の行の一意な識別子である、Oracle ROWID をサポートします。表から任意のデータ列を選択する場合と同じように、ROWID を選択できます。ただし、手動で ROWID を更新できません。適宜、Oracle データベースが自動的に更新します。

oracle.sql.ROWID クラスは、oracle.sql.Datum スーパークラスが提供する機能以外の重要な機能を実装していません。ただし、ROWID は、oracle.sql.Datum クラスの stringValue() メソッドをオーバーライドし、ROWID バイトの 16 進値表現を戻す stringValue() メソッドを提供します。

ROWID データへのアクセスの詳細は、5-32 ページの「Oracle ROWID 型」を参照してください。

クラス oracle.sql.OPAQUE

oracle.sql.OPAQUE クラスは、OPAQUE 型の名前および特性と任意の属性を提供します。OPAQUE 型では、インスタンスの連続バイトにのみアクセスできます。

> **注意：** Oracle9*i* リリース 1（9.0.1）では、OPAQUE 型のサポートは限定されています。

oracle.sql.OPAQUE クラスには、次のメソッドがあります。

- getBytesValue(): OPAQUE オブジェクトの値を表すバイトを、データベースで使用される形式で戻します。
- public boolean isConvertibleTo(Class jClass): Datum オブジェクトを特定のクラスに変換できるかどうか判断します。Class は任意のクラス、jClass は変換先のクラスを示します。jClass への変換が可能な場合は TRUE、jClass への変換ができない場合は FALSE が戻されます。
- getDescriptor(): タイプ情報を含んだ OpaqueDescriptor オブジェクトを戻します。
- getJavaSqlConnection(): 受信者と対応付けられた接続を戻します。oracle.jdbc.driver パッケージを使用するメソッドは廃止されたため、getConnection() メソッドにかわって getJavaSqlConnection() メソッドを使用します。
- getSQLTypeName(): java.sql.Struct インタフェース・ファンクションを実装し、この Struct オブジェクトが表す SQL 構造化型の SQL 型名を取得します。このメソッドは、この STRUCT オブジェクトが表す SQL 構造化型の完全修飾型名を戻します。
- getValue(): 値（ロー・バイト）を表す Java オブジェクトを戻します。
- toJdbc(): Datum オブジェクトの JDBC 表現を戻します。

パッケージ oracle.jdbc

oracle.jdbc パッケージのインタフェースは、Oracle 固有の拡張機能を提供します。これにより、oracle.sql.* オブジェクトを使用して、実際の SQL 形式のデータへのアクセスが可能になります。

注意： 以前のリリースで使用されていた oracle.jdbc.driver パッケージのかわりに、oracle.jdbc パッケージを使用します。（詳細は、1-13 ページの「パッケージ oracle.jdbc」を参照してください。）

表 5-2 に、oracle.jdbc パッケージに含まれる接続のための主なインタフェースとクラス、文および結果セットのリストを示します。

表 5-2 oracle.jdbc パッケージの主なインタフェースおよびクラス

名前	インタフェースまたはクラス	主要機能
OracleDriver	クラス	java.sql.Driver の実装。
OracleConnection	インタフェース	Oracle statement オブジェクトを戻すメソッド。カレント接続で実行される任意の文に対応した Oracle パフォーマンス拡張要素を設定するメソッド（java.sql.Connection を実装します）。
OracleStatement	インタフェース	文ごとに Oracle パフォーマンス拡張要素を設定するメソッド。OraclePreparedStatement および OracleCallableStatement のスーパークラス（java.sql.Statement を実装）。
OraclePreparedStatement	インタフェース	oracle.sql.* 型をプリコンパイルされた SQL 文にバインドする setXXX() メソッド（java.sql.PreparedStatement を実装し、OracleCallableStatement のスーパークラスである OracleStatement を拡張します）。

表 5-2 oracle.jdbc パッケージの主なインタフェースおよびクラス （続き）

名前	インタフェースまたはクラス	主要機能
OracleCallableStatement	インタフェース	データを oracle.sql 形式で取り出す getXXX() メソッド。oracle.sql.* 型をコール可能文にバインドする setXXX() メソッド (java.sql.CallableStatement を実装し、OraclePreparedStatement を拡張します)。
OracleResultSet	インタフェース	oracle.sql 形式でデータを取り出す getXXX() メソッド (java.sql.ResultSet を実装します)。
OracleResultSetMetaData	インタフェース	列名およびデータ型などの Oracle 結果セットのメタ情報を取り出すメソッド (java.sql.ResultSetMetaData を実装します)。
OracleDatabaseMetaData	クラス	データベース製品名 / バージョン、表情報およびデフォルト・トランザクション分離レベルなどのデータベースに関するメタ情報を取り出すメソッド (java.sql.DatabaseMetaData を実装します)。
OracleTypes	クラス	SQL 型を識別する整数定数を定義します。標準型の場合、同じ値を標準 java.sql.Types クラスとして使用します。さらに、Oracle 拡張型に対応した定数も追加します。

以降では、oracle.jdbc パッケージのインタフェースおよびクラスについて説明します。これらのインタフェースおよびクラスを使用して Oracle 型拡張機能にアクセスする方法は、第 6 章「Oracle データへのアクセスと操作」を参照してください。

クラス oracle.jdbc.OracleDriver

このクラスを使用して、Oracle JDBC ドライバをアプリケーションで使用するための登録を行います。このクラスの新規インスタンスを java.sql.DriverManager クラスの静的 registerDriver() メソッドに入力して、アプリケーションから Oracle ドライバへアクセスして使用できるようにします。registerDriver() メソッドは、OracleDriver の場合と同様に、ドライバ・クラス（java.sql.Driver インタフェースを実装するクラス）を入力として取ります。

Oracle JDBC ドライバの登録が完了すると、DriverManager クラスを使用して接続を作成できます。ドライバの登録および接続文字列の書込みの詳細は、3-2 ページの「JDBC での最初の処理」を参照してください。

インタフェース oracle.jdbc.OracleConnection

このインタフェースは、標準 JDBC 接続機能を拡張して、Oracle 文オブジェクトの作成と復帰、Oracle パフォーマンス拡張要素用のフラグやオプションの設定、および Oracle オブジェクト用型マップのサポートを行います。

行のプリフェッチ、バッチ更新およびメタデータ TABLE_REMARKS のレポートに関する情報を含む、パフォーマンス拡張要素の詳細は、12-19 ページの「その他の Oracle パフォーマンス拡張要素」を参照してください。

主要メソッドは、次のとおりです。

- createStatement(): 新規 OracleStatement オブジェクトを割り当てます。
- prepareStatement(): 新規 OraclePreparedStatement オブジェクトを割り当てます。
- prepareCall(): 新規 OracleCallableStatement オブジェクトを割り当てます。
- getTypeMap(): この接続の型マップを取り出します（Oracle オブジェクト型の Java クラスへのマッピングで使用するため）。
- setTypeMap(): この接続用の型マップの初期化または更新を行います（Oracle オブジェクト型の Java クラスへのマッピングで使用するため）。
- getTransactionIsolation(): この接続のカレント分離モードを取り出します。
- setTransactionIsolation():TRANSACTION_* 値の 1 つを使用してトランザクション分離レベルを変更します。

これらの oracle.jdbc.OracleConnection メソッドは、Oracle 定義の拡張機能です。

- getDefaultExecuteBatch(): この接続に対応したデフォルトのバッチ更新値を取り出します。
- setDefaultExecuteBatch(): この接続に対応したデフォルトのバッチ更新値を設定します。

- getDefaultRowPrefetch(): この接続に対応したデフォルトの行プリフェッチ値を取り出します。
- setDefaultRowPrefetch(): この接続に対応したデフォルトの行プリフェッチ値を設定します。
- getRemarksReporting():TABLE_REMARKS のレポート処理が有効な場合、TRUE を戻します。
- setRemarksReporting():TABLE_REMARKS のレポート処理を有効または無効にします。

インタフェース oracle.jdbc.OracleStatement

このインタフェースは、標準 JDBC 文の機能を拡張した、OraclePreparedStatement および OracleCallableStatement クラスのスーパーインタフェースです。拡張機能には、文単位で Oracle パフォーマンス拡張要素用のフラグおよびオプションの設定のサポートが含まれます。この機能は、接続単位でこれらの設定を行う OracleConnection インタフェースと対照的な役割を果たします。

12-19 ページの「その他の Oracle パフォーマンス拡張要素」では、行のプリフェッチと列型の定義を含むパフォーマンス拡張機能について説明します。

主要メソッドは、次のとおりです。

- executeQuery(): データベースへの問合せを実行し、OracleResultSet オブジェクトを戻します。
- getResultSet():OracleResultSet オブジェクトを取り出します。
- close(): カレント文をクローズします。

これらの oracle.jdbc.OracleStatement メソッドは、Oracle 定義の拡張機能です。

- defineColumnType(): 特定のデータベース表の列からデータを取り出す際に使用する型を定義します。
- getRowPrefetch(): この文の行プリフェッチ値を取り出します。
- setRowPrefetch(): この文の行プリフェッチ値を設定します。

インタフェース oracle.jdbc.OraclePreparedStatement

このインタフェースは、OracleStatement インタフェースを拡張し、さらに標準 JDBC のプリコンパイルされた SQL 文の機能を拡張します。また、oracle.jdbc.OraclePreparedStatement インタフェースは OracleCallableStatement インタフェースによって拡張されます。拡張機能には、oracle.sql.* 型およびオブジェクトをプリコンパイルされた SQL 文にバインドする setXXX() メソッド、および文単位で Oracle パフォーマンス拡張機能をサポートするメソッドが含まれます。

12-19 ページの「その他の Oracle パフォーマンス拡張要素」では、データベースのバッチ更新を含むパフォーマンス拡張機能について説明します。

主要メソッドは、次のとおりです。

- getExecuteBatch(): この文のバッチ更新値を取り出します。
- setExecuteBatch(): この文のバッチ更新値を設定します。
- setOracleObject(): これは oracle.sql.* データをプリコンパイルされた SQL 文に oracle.sql.Datum オブジェクトとしてバインドする、汎用の setXXX() メソッドです。
- setXXX(): setBLOB() などのメソッドは、特定の oracle.sql.* 型をプリコンパイルされた SQL 文にバインドします。
- setORAData(): ORAData オブジェクトを（Oracle オブジェクト型を Java へマッピングするために）プリコンパイルされた文にバインドします。
- setNull(): SQL 型名で指定されたオブジェクトの値を NULL に設定します。setNull（*param_index*、*type_code*、*sql_type_name*）では、type_code が REF、ARRAY または STRUCT の場合、*sql_type_name* には SQL 型の完全修飾名（*schema.sql_type_name*）が当てはまります。
- setFormOfUse(): このメソッドで使用する形式を設定します。使用形式を指定する定数には、FORM_CHAR と FORM_NCHAR の 2 つがあります。デフォルトは FORM_CHAR です。使用する形式を FORM_NCHAR に設定すると、JDBC ドライバは提供されたデータをサーバーの各国語キャラクタ・セットで表示します。次のコードは、FORM_NCHAR の使用方法を示します。

  ```
  pstmt.setFormOfUse
    (parameter index,
    oracle.jdbc.OraclePreparedStatement.FORM_NCHAR)
  ```

- close(): カレント文をクローズします。

インタフェース oracle.jdbc.OracleCallableStatement

このインタフェースは、（OracleStatement インタフェースを拡張する）OraclePreparedStatement インタフェースを拡張し、標準 JDBC のコール可能文機能を組み込みます。

主要メソッドは、次のとおりです。

- getOracleObject(): これは、データを oracle.sql.Datum オブジェクト内に取り込む汎用の getXXX() メソッドです。必要に応じて、特定の oracle.sql.* 型にキャストできます。
- getXXX(): getCLOB() などのこれらのメソッドは、データを特定の oracle.sql.* オブジェクト内に取り込みます。
- setOracleObject(): これは、oracle.sql.* データをコール可能文に oracle.sql.Datum オブジェクトとしてバインドする汎用の setXXX() メソッドです。
- setXXX(): setBLOB() などのこれらのメソッドは、特定の oracle.sql.* オブジェクトをコール可能文にバインドするための OraclePreparedStatement から継承されます。
- setNull(): SQL 型名で指定されたオブジェクトの値を NULL に設定します。setNull（*param_index*、*type_code*、*sql_type_name*）では、*type_code* が REF、ARRAY または STRUCT の場合、*sql_type_name* には SQL 型の完全修飾名（*schema.type*）が当てはまります。
- setFormOfUse(): このメソッドで使用する形式を設定します。使用形式を指定する定数には、FORM_CHAR と FORM_NCHAR の 2 つがあります。デフォルトは FORM_CHAR です。使用する形式を FORM_NCHAR に設定すると、JDBC ドライバは提供されたデータをサーバーの各国語キャラクタ・セットで表示します。次のコードは、FORM_NCHAR の使用法を示します。

  ```
  pstmt.setFormOfUse
    (parameter index,
    oracle.jdbc.OraclePreparedStatement.FORM_NCHAR)
  ```

- registerOutParameter(): 文の出力パラメータの SQL タイプコードを登録します。JDBC は、OUT パラメータを取る任意のコール可能文でこれを必要とします。整数値のパラメータ索引（文内の出力変数の他のパラメータに対する相対位置）および整数値の SQL 型（oracle.jdbc.OracleTypes で定義された型定数）を取ります。

  このメソッドでは、オーバーロードが行われます。このメソッドの 1 つのバージョンは、名前付きの型、つまり SQL タイプコードが OracleTypes.REF、STRUCT または ARRAY の場合のみを対象とします。この場合、このメソッドは、パラメータ索引と SQL 型に加え、String SQL 型名（EMPLOYEE などの、データベース内の Oracle ユーザー定義型の名前）を取ります。
- close(): カレント結果セット（存在する場合）およびカレント文をクローズします。

インタフェース oracle.jdbc.OracleResultSet

このインタフェースは、標準 JDBC 結果セットの機能を拡張し、getXXX() メソッドを実装してデータを oracle.sql.* オブジェクト内に取り込みます。

主要メソッドは、次のとおりです。

- getOracleObject(): これは、データを oracle.sql.Datum オブジェクト内に取り込む汎用の getXXX() メソッドです。このメソッドは、必要に応じて特定の oracle.sql.* 型にキャストできます。
- getXXX():getCLOB() などのこれらのメソッドは、データを oracle.sql.* オブジェクト内に取り出します。

インタフェース oracle.jdbc.OracleResultSetMetaData

このインタフェースは、標準 JDBC 結果セットのメタデータ機能を拡張して、Oracle 結果セット・オブジェクトの情報を取得します。OracleResultSetMetadata インタフェースの機能の詳細は、6-18 ページの「結果セット・メタデータ拡張要素の使用方法」を参照してください。

クラス oracle.jdbc.OracleTypes

OracleTypes クラスは、JDBC が SQL 型を識別するために使用する定数を定義します。このクラス内の各変数は、整数の定数値を持ちます。oracle.jdbc.OracleTypes クラスには、標準 Java java.sql.Types クラスのタイプコード定義のコピーと、次の Oracle 拡張機能用の補足的なタイプコードが含まれます。

- OracleTypes.BFILE
- OracleTypes.ROWID
- OracleTypes.CURSOR（REF CURSOR 型の場合）

java.sql.Types の場合と同様、どの変数名もすべて大文字になります。

JDBC は、OracleTypes クラスの要素により識別される SQL 型を、出力パラメータの登録、および PreparedStatement クラスの setNull() メソッドという 2 つの分野で使用します。

OracleTypes と出力パラメータの登録 java.sql.Types または oracle.jdbc.OracleTypes のタイプコードは、java.sql.CallableStatement インタフェースと oracle.jdbc.OracleCallableStatement インタフェースの registerOutParameter() メソッドで出力パラメータの SQL 型を識別します。

次に、registerOutputParameter() が CallableStatement および OracleCallableStatement で利用可能な形式を示します（標準コール可能文オブジェクト cs を想定します）。

```
cs.registerOutParameter(int index, int sqlType);

cs.registerOutParameter(int index, int sqlType, String sql_name);

cs.registerOutParameter(int index, int sqlType, int scale);
```

これらのシグネチャでは、*index* にはパラメータ索引、*sqlType* には SQL データ型のタイプコード、*sql_name* にはデータ型に与えられた名前（*sqlType* が STRUCT、REF または ARRAY タイプコードの場合はユーザー定義型の名前）、*scale* には小数点の右側にある数値の数（*sqlType* が NUMERIC または DECIMAL タイプコードの場合）が当てはまります。

注意： 2 番目のシグネチャは、JDK 1.2.x 環境の JDBC 2.0 では標準ですが、JDK 1.1.x では Oracle 拡張機能です。

次の例では、CallableStatement を使用して charout という名前のプロシージャをコールします。このプロシージャは、CHAR データ型を戻します。registerOutParameter() メソッドでは OracleTypes.CHAR タイプコードを使用することに注意してください（ただし、java.sql.Types.CHAR も使用できます）。

```
CallableStatement cs = conn.prepareCall ("BEGIN charout (?); END;");
cs.registerOutParameter (1, OracleTypes.CHAR);
cs.execute ();
System.out.println ("Out argument is: " + cs.getString (1));
```

次の例では、CallableStatement を使用して structout をコールします。このプロシージャは、STRUCT データ型を戻します。registerOutParameter() の形式では、タイプコード（Types.STRUCT または OracleTypes.STRUCT）と SQL 名（EMPLOYEE）を指定する必要があります。

この例では、EMPLOYEE 型でいかなる型マップも宣言されていないことを前提としているため、STRUCT データ型内に取り込まれます。oracle.sql.STRUCT オブジェクトとして EMPLOYEE の値を取り出すために、文オブジェクト cs は、OracleCallableStatement にキャストされ、Oracle 拡張機能 getSTRUCT() メソッドが実行されます。

```
CallableStatement cs = conn.prepareCall ("BEGIN structout (?); END;");
cs.registerOutParameter (1, OracleTypes.STRUCT, "EMPLOYEE");
cs.execute ();

// get the value into a STRUCT because it
// is assumed that no type map has been defined
STRUCT emp = ((OracleCallableStatement)cs).getSTRUCT (1);
```

OracleTypes および setNull() メソッド Types および OracleTypes のタイプコードは、データ項目の SQL 型を識別します。そのデータ項目は setNull() メソッドにより NULL に設定されます。setNull() メソッドは、java.sql.PreparedStatement インタフェースおよび oracle.jdbc.OraclePreparedStatement インタフェース内に存在します。

setNull() が PreparedStatement および OraclePreparedStatement オブジェクトに対して使用可能な形式を次に示します（標準コンパイル済み SQL 文オブジェクト ps を想定します）。

```
ps.setNull(int index, int sqlType);

ps.setNull(int index, int sqlType, String sql_name);
```

これらのシグネチャでは、*index* にはパラメータ索引、*sqlType* には SQL データ型のタイプコード、*sql_name* にはデータ型に与えられた名前（*sqlType* が STRUCT、REF または ARRAY の場合はユーザー定義型の名前）が当てはまります。無効な *sqlType* を入力すると、「パラメータ・タイプ矛盾」の例外が発生します。

注意： 2 番目のシグネチャは、JDK 1.2.x 環境の JDBC 2.0 では標準ですが、JDK 1.1.x では Oracle 拡張機能です。

次の例では、PreparedStatement を使用して数値 NULL をデータベースに挿入します。NULL に設定する数値オブジェクトの識別に OracleTypes.NUMERIC を使用することに注意してください（ただし、Types.NUMERIC も使用されることがあります）。

```
PreparedStatement pstmt =
    conn.prepareStatement ("INSERT INTO num_table VALUES (?)");

pstmt.setNull (1, OracleTypes.NUMERIC);
pstmt.execute ();
```

この例では、プリコンパイルされた SQL 文を使用して、EMPLOYEE 型の NULL STRUCT オブジェクトをデータベースに挿入します。

```
PreparedStatement pstmt = conn.prepareStatement
                              ("INSERT INTO employee_table VALUES (?)");

pstmt.setNull (1, OracleTypes.STRUCT, "EMPLOYEE");
pstmt.execute ();
```

Oracle 固有の機能のための Oracle インタフェース

以前のリリースで使用されていた oracle.jdbc.driver パッケージの同名のクラスのかわりに、Oracle9*i* で導入された oracle.jdbc インタフェースを使用することをお薦めします。基本的に、これらのインタフェースは oracle.jdbc.driver パッケージに含まれる機能のコピーと同じです。

次の例は、oracle.jdbc パッケージを使用して、pstmt を Oracle 型としてキャストする方法を示します。

```
java.sql.PreparedStatement pstmt
  = conn.prepareStatement(...);

((oracle.jdbc.OraclePreparedStatement) pstmt)
  .setExecuteBatch(10);   // Oracle-specific method
```

メソッド getJavaSqlConnection()

oracle.sql.* クラスの getJavaSqlConnection() メソッドは java.sql.Connection を戻すのに対して、getConnection() メソッドは oracle.jdbc.driver.OracleConnection を戻します。oracle.jdbc.driver パッケージを使用するメソッドは廃止されたため、getConnection() メソッドにかわって getJavaSqlConnection() メソッドを使用します。

次の Oracle データ型クラスに getJavaSqlConnection() メソッドが追加されています。

- oracle.sql.ARRAY
- oracle.sql.BFILE
- oracle.sql.BLOB
- oracle.sql.CLOB
- oracle.sql.OPAQUE
- oracle.sql.REF
- oracle.sql.STRUCT

Array クラスの getJavaSqlConnection() メソッドおよび getConnection() メソッドを次に示します。

```
public class ARRAY
{
  // New API
  //
  java.sql.Connection getJavaSqlConnection()
    throws SQLException;

  // Deprecated API.  It throws a SQLException
  //
  oracle.jdbc.driver.OracleConnection
    getConnection() throws SQLException;

  ...
}
```

パッケージ oracle.jdbc2（JDK 1.1.x の場合のみ）

oracle.jdbc2 パッケージは、JDK 1.1.x で使用するための Oracle 実装で、標準の JDBC 2.0 クラスおよびインタフェースのサブセットを疑似実行するクラスとインタフェースが含まれます（これは標準 java.sql パッケージの JDK 1.2 バージョン内にあります）。

次のインタフェースは、JDK 1.1.x では JDBC 2.0 に準拠した Oracle 型拡張機能対応の oracle.sql.* 型クラスにより実装されています。

- oracle.jdbc2.Array は、oracle.sql.ARRAY により実装されます。
- oracle.jdbc2.Struct は、oracle.sql.STRUCT により実装されます。
- oracle.jdbc2.Ref は、oracle.sql.REF により実装されます。
- oracle.jdbc2.Clob は、oracle.sql.CLOB により実装されます。
- oracle.jdbc2.Blob は、oracle.sql.BLOB により実装されます。

また、Oracle オブジェクトにマップする Java クラスを JDBC 標準 SQLData インタフェースを使用して作成するユーザーのために、oracle.jdbc2 パッケージには次のインタフェースも含まれています。これらのインタフェースでも JDK 1.2 で使用可能な java.sql インタフェースを疑似実行します。

- oracle.jdbc2.SQLData は、Oracle オブジェクトにマップするクラスによって実装されます。ユーザーはこの実装を提供する必要があります。
- oracle.jdbc2.SQLInput は、オブジェクト・データを読み込むクラスによって実装されます。Oracle は JDBC ドライバで使用する SQLInput クラスを提供します。
- oracle.jdbc2.SQLOutput は、オブジェクト・データを書き込むクラスによって実装されます。Oracle は JDBC ドライバで使用する SQLOutput クラスを提供します。

SQLData インタフェースは、Java で Oracle オブジェクトをサポートするために使用可能な 2 つの機能の 1 つです。（他のオプションは、Oracle ORAData インタフェースで、oracle.sql パッケージに含まれています。）SQLData、SQLInput および SQLOutput の詳細は、8-15 ページの「SQLData インタフェース」を参照してください。

Oracle 文字データ型のサポート

Oracle 文字データ型には、SQL CHAR および SQL NCHAR データ型が含まれます。この項では、Oracle JDBC ドライバを使用してこれらのデータ型にアクセスする方法について説明します。

SQL CHAR データ型

SQL CHAR データ型には、CHAR、VARCHAR および CLOB データ型が含まれます。これらのデータ型を使用して、データベース・キャラクタ・セットのコード体系で文字データを格納できます。データベースのキャラクタ・セットは、データベースの作成時に決まります。

SQL NCHAR データ型

SQL NCHAR データ型は、グローバリゼーション・サポート（以前の NLS）用に作成されたものです。SQL NCHAR データ型には、NCHAR、NVARCHAR2 および NCLOB データ型が含まれます。これらのデータ型を使用して、データベース NCHAR キャラクタ・セットのコード体系で Unicode データを格納できます。NCHAR キャラクタ・セットは、データベースの作成時に決まり、変更されることはありません。SQL NCHAR データ型の詳細は、『Oracle9*i* グローバリゼーション・サポート・ガイド』を参照してください。

注意： UnicodeStream クラスにかわって CharacterStream クラスが導入されたため、NCHAR データ型でのアクセスには setUnicodeStream() メソッドおよび getUnicodeStream() メソッドはサポートされません。ストリーム・アクセスを使用する場合は、setCharacterStream() メソッドおよび getCharacterStream() メソッドを使用してください。

SQL NCHAR データ型の使用方法は、SQL CHAR（CHAR、VARCHAR2 および CLOB）データ型の使用方法と同じです。JDBC では、対応する SQL CHAR データ型で使用されるのと同じクラスおよびメソッドを使用して、SQL NCHAR データ型にアクセスします。したがって、SQL NCHAR データ型用の oracle.sql パッケージには対応する別のクラスが定義されていません。同様に、SQL NCHAR データ型用の oracle.jdbc.OracleTypes パッケージにも、対応する別の定数は定義されていません。この 2 つのデータ型で唯一使用方法が異なるのは、データをバインドする方法です。データを SQL NCHAR データ型にバインドするように指定する場合、JDBC プログラムは setFormOfUse() メソッドをコールする必要があります。

注意： Oracle9*i* リリース 1（9.0.1）の場合、予測できない結果を回避するために、registerOutParameter() メソッドをコールする前に setFormOfUse() メソッドをコールする必要があります。

次のコードは、SQL NCHAR データにアクセスする方法を示します。

```
//
// Table TEST has the following columns:
// - NUMBER
// - NVARCHAR2
// - NCHAR
//
oracle.jdbc.OraclePreparedStatement pstmt =
  (oracle.jdbc.OraclePreparedStatement)
conn.prepareStatement("insert into TEST values(?, ?, ?)");

//
// oracle.jdbc.OraclePreparedStatement.FORM_NCHAR should be used for all NCHAR,
// NVARCHAR2 and NCLOB data types.
//
pstmt.setFormOfUse(2, FORM_NCHAR);
pstmt.setFormOfUse(3, FORM_NCHAR);

pstmt.setInt(1, 1);                     // NUMBER column
pstmt.setString(2, myUnicodeString1);  // NVARCHAR2 column
pstmt.setString(3, myUnicodeString2);  // NCHAR column
pstmt.execute();
```

クラス oracle.sql.CHAR

CHAR クラスは、文字データを処理し変換するために Oracle JDBC によって使用されます。データベースから文字データの読込みが行われると、JDBC ドライバは oracle.sql.CHAR オブジェクトを作成して移入します。

注意： oracle.sql.CHAR クラスは、SQL CHAR および SQL NCHAR の両方のデータ型に使用されます。

JDBC ドライバによって作成または戻される CHAR オブジェクトは、データベース・キャラクタ・セット、UTF-8 または ISO-Latin-1（WE8ISO8859P1）になります。Oracle オブジェクト属性である CHAR オブジェクトは、データベース・キャラクタ・セットで戻されます。

JDBC アプリケーション・コードで CHAR オブジェクトを直接作成する必要はほとんどありません。これはデータベースから文字データが取り出されたときに、JDBC ドライバが CHAR オブジェクトを自動的に作成するためです。ただし、CHAR オブジェクトをアプリケーション・コードで直接作成するほうが効率的な場合もあります。たとえば、1 つ以上のプリコンパイルされた SQL 文に同じ文字データを繰り返し渡す場合は、毎回 Java 文字列から変換するオーバーヘッドをなくすことができます。

oracle.sql.CHAR オブジェクトとキャラクタ・セット

CHAR クラスは、文字データを変換するためのグローバリゼーション・サポート機能を提供します。このクラスには、グローバリゼーション・サポート・キャラクタ・セットおよび文字データという 2 つの主要な属性があります。グローバリゼーション・サポート・キャラクタ・セットは、文字データのエンコーディングを定義します。これは、CHAR オブジェクトの作成時に必ず渡されるパラメータです。グローバリゼーション・サポート・キャラクタ・セットが不明な状態では、CHAR オブジェクト内のデータのバイトは無意味です。

oracle.sql.CharacterSet クラスは、キャラクタ・セットを表すためにインスタンス化されます。CHAR オブジェクトを作成する場合は、CharacterSet クラスのインスタンスによって、CHAR オブジェクトにキャラクタ・セット情報を提供する必要があります。このクラスの各インスタンスは、Oracle がサポートしているグローバリゼーション・サポート・キャラクタ・セットの 1 つを表します。CharacterSet インスタンスは、キャラクタ・セットのメソッドおよび属性をカプセル化し、主に他のキャラクタ・セットとの相互変換機能を提供します。Oracle がサポートするキャラクタ・セットの完全なリストは、『Oracle9*i* グローバリゼーション・サポート・ガイド』を参照してください。

CHAR オブジェクトを作成するときには、使用できるキャラクタ・セットに制限があります。Unicode 列（NCHAR、NVARCHAR2 および NCLOB）用に CHAR オブジェクトを作成する場合は、データベース・キャラクタ・セットを使用してください。CHAR、VARCHAR または VARCHAR2 型の列に CHAR オブジェクトを作成する場合は、表 5-3 に示すいずれか 1 つのキャラクタ・セットを使用します。

表 5-3 SQL CHAR データ型のキャラクタ・セット

キャラクタ・セット	使用する場合
US7ASCII	データベース・キャラクタ・セットが US7ASCII の場合
WE8ISO8859P1	データベース・キャラクタ・セットが WE8ISO8859P1 の場合
UTF8	データベース・キャラクタ・セットが US7ASCII または WE8ISO8859P1 の場合

NCHAR データ型用に CHAR オブジェクトを作成するときには、データベース NCHAR キャラクタ・セットを使用します。

oracle.sql.CHAR オブジェクトの作成

CHAR オブジェクトを構築する際、次の一般的な手順に従ってください。

1. 静的 CharacterSet.make() メソッドをコールすることにより、CharacterSet オブジェクトを作成します。

   このメソッドは、キャラクタ・セット・インスタンスのファクトリです。make() メソッドは、Oracle がサポートするキャラクタ・セット ID に対応する整数を入力として取ります。たとえば、次のようになります。

   ```
   int oracleId = CharacterSet.JA16SJIS_CHARSET; // this is character set ID,
                                                 // 832
   ...
   CharacterSet mycharset = CharacterSet.make(oracleId);
   ```

   Oracle がサポートする各キャラクタ・セットは、事前定義済みの一意な Oracle ID を保持します。

   > **注意：** 無効なキャラクタ・セット ID を入力しても、例外は発生しません。そのかわりに、予測できない結果になります。

   キャラクタ・セットおよびキャラクタ・セット ID の詳細は、『Oracle9*i* グローバリゼーション・サポート・ガイド』を参照してください。

2. CHAR オブジェクトを作成します。

   コンストラクタに文字列（または文字列を表すバイト）と、キャラクタ・セットに基づくバイトの解釈方法を指定する CharacterSet オブジェクトを渡します。たとえば、次のようになります。

   ```
   String mystring = "teststring";
   ...
   CHAR mychar = new CHAR(teststring, mycharset);
   ```

   CHAR クラスには、CharacterSet オブジェクトとともに入力として文字列、バイト配列またはオブジェクトを取ることができる複数のコンストラクタがあります。文字列の場合は、CharacterSet オブジェクトが示すキャラクタ・セットに変換されてから、CHAR オブジェクトに格納されます。

   詳細は、oracle.sql.CHAR クラス Javadoc を参照してください。

注意：

- CharacterSet オブジェクトを NULL にすることはできません。
- CharacterSet クラスは抽象クラスであるため、コンストラクタを持ちません。インスタンスを作成するには、make() メソッドを使用するのが唯一の方法です。
- サーバーは、特別な値である CharacterSet.DEFAULT_CHARSET をデータベース・キャラクタ・セットとして認識します。クライアントの場合、この値は無意味です。
- ユーザーが CharacterSet クラスを拡張することを、オラクル社は意図しておらず、お薦めもしません。

oracle.sql.CHAR Conversion メソッド

CHAR クラスは、文字データを文字列に変換するための次のようなメソッドを提供します。

- getString(): CHAR オブジェクトによって表現された文字のシーケンスを文字列に変換し、Java String オブジェクトを戻します。無効な OracleID を入力した場合、キャラクタ・セットは認識されず、getString() メソッドは SQLException を発生します。
- toString(): getString() メソッドと同じです。ただし、無効な OracleID を入力した場合、キャラクタ・セットは認識されず、toString() メソッドは CHAR データの 16 進表現を戻します。この場合、SQLException は発生しません。
- getStringWithReplacement(): getString() と同じですが、この CHAR オブジェクトのキャラクタ・セット内に Unicode 表現を持たない文字が、デフォルトの置換文字で置換される点が異なります。このデフォルトの置換文字は、キャラクタ・セットごとに異なりますが、通常はクエスチョン・マーク（?）です。

サーバー（データベース）とクライアント、またはクライアント上で動作するアプリケーションでは、使用するキャラクタ・セットが異なっていてもかまいません。CHAR クラスのメソッドを使用してデータをサーバーとクライアント間で転送する場合、JDBC ドライバはデータをサーバーのキャラクタ・セットからクライアントのキャラクタ・セットに（またはその逆に）変換する必要があります。データを変換する際、ドライバはグローバリゼーション・サポートを使用します。JDBC ドライバによるキャラクタ・セット間の変換の詳細は、18-2 ページの「JDBC およびグローバリゼーション・サポート」を参照してください。

その他の Oracle 型拡張機能

主要な Oracle 型拡張機能については、このマニュアルの次の章を参照してください。

- 第 7 章「LOB と BFILE の操作」
- 第 8 章「Oracle オブジェクト型の操作」
- 第 9 章「Oracle オブジェクト参照の操作」
- 第 10 章「Oracle コレクションの操作」

この項では、その他の Oracle 型拡張機能を取り上げ、最後に Oracle 拡張機能のサポートに関する現行の Oracle JDBC ドライバと Oracle 8.0.x および 7.3.x ドライバ間の違いについて説明します。

Oracle JDBC ドライバでは、Oracle 固有の BFILE および ROWID のデータ型と、REF CURSOR 型をサポートしています。これらのデータ型は Oracle7 から導入され、標準 JDBC の仕様にはありません。この項では、ROWID および REF CURSOR 型拡張機能について説明します。BFILE については、第 7 章「LOB と BFILE の操作」を参照してください。

ROWID は Java 文字列として、REF CURSOR は JDBC 結果セットとしてサポートされます。

Oracle ROWID 型

ROWID は、Oracle データベース表の各行で一意の識別タグです。ROWID は、各行の ID を含む仮想列と見なすこともできます。

oracle.sql.ROWID クラスは、型 ROWID SQL データのラッパーとして提供されます。

ROWID は、java.sql.ResultSet インタフェースで指定される getCursorName() メソッドと java.sql.Statement インタフェースで指定される setCursorName() メソッドに類似している機能を提供します。

問合せに ROWID 擬似列を含めた場合は、結果セット getString() メソッドを使用して（列索引または列名を渡します）ROWID を取り出すことができます。また、setString() メソッドを使用すると、ROWID に PreparedStatement パラメータをバインドできます。これにより、次の例に示すように、埋込み更新を行うことができます。

注意： oracle.jdbc.driver.ROWID のかわりに、oracle.sql.ROWID クラスを使用します。このクラスは、以前のリリースの Oracle JDBC で使用されていました。

例 : ROWID 次の例では、ROWID データに対するアクセスおよび操作の方法を示します。

```
Statement stmt = conn.createStatement();

// Query the employee names with "FOR UPDATE" to lock the rows.
// Select the ROWID to identify the rows to be updated.

ResultSet rset =
   stmt.executeQuery ("SELECT ename, rowid FROM emp FOR UPDATE");

// Prepare a statement to update the ENAME column at a given ROWID

PreparedStatement pstmt =
   conn.prepareStatement ("UPDATE emp SET ename = ? WHERE rowid = ?");

// Loop through the results of the query
while (rset.next ())
{
    String ename = rset.getString (1);
    oracle.sql.ROWID rowid = rset.getROWID (2);  // Get the ROWID as a String
    pstmt.setString (1, ename.toLowerCase ());
    pstmt.setROWID (2, rowid); // Pass ROWID to the update statement
    pstmt.executeUpdate ();     // Do the update
}
```

Oracle REF CURSOR 型カテゴリ

カーソル変数には、問合せ作業域の内容ではなく、問合せ作業域のメモリー位置（アドレス）が保持されます。カーソル変数を宣言すると、ポインタが作成されます。SQL では、ポインタにはデータ型 REF *x* が設定されています。REF は REFERENCE の短縮名で、*x* は参照されているエンティティを表します。REF CURSOR はカーソル変数への参照を意味します。多くのカーソル変数が存在し、様々な作業領域をポイントしているため、REF CURSOR は、カテゴリまたは多くの異なる種類のカーソル変数を識別するデータ型識別子と見なすこともできます。

カーソル変数を作成するには、まず REF CURSOR カテゴリに属する型を識別します。たとえば、次のようになります。

```
DECLARE TYPE DeptCursorTyp IS REF CURSOR
```

次に、カーソル変数を DeptCursorTyp 型と宣言することにより、カーソル変数を作成します。

```
dept_cv DeptCursorTyp  - - declare cursor variable
...
```

REF CURSOR は、特定のデータ型ではなく、データ型のカテゴリです。

ストアド・プロシージャにより、REF CURSOR カテゴリのカーソル変数が戻されます。この出力は、データベース・カーソルまたは JDBC 結果セットと等価です。REF CURSOR には、基本的に問合せの結果が格納されます。

JDBC では、REF CURSOR は、ResultSet オブジェクトとしてインスタンス化され、次の方法でアクセスできます。

1. ストアド・プロシージャをコールするには JDBC コール可能文を使用します。出力パラメータがあるため、プリコンパイルされた SQL 文ではなく、コール可能文を使用する必要があります。
2. ストアド・プロシージャにより REF CURSOR が戻されます。
3. Java アプリケーションによりコール可能文が Oracle コール可能文にキャストされ、OracleCallableStatement クラスの getCursor() メソッドを使用して REF CURSOR が JDBC ResultSet オブジェクトとしてインスタンス化されます。
4. 結果セットは要求どおりに処理されます。

重要： REF CURSOR と対応付けられたカーソルは、その REF CURSOR を作成した文オブジェクトがクローズされるたびにクローズされます。

以前のリリースとは異なり、REF CURSOR と対応付けられたカーソルは、その REF CURSOR がインスタンス化された結果セット・オブジェクトをクローズしたときにクローズされることはありません。

例 : REF CURSOR データへのアクセス この例では、REF CURSOR データにアクセスする方法を示します。

```
import oracle.jdbc.*;
...
CallableStatement cstmt;
ResultSet cursor;

// Use a PL/SQL block to open the cursor
cstmt = conn.prepareCall
         ("begin open ? for select ename from emp; end;");

cstmt.registerOutParameter(1, OracleTypes.CURSOR);
cstmt.execute();
cursor = ((OracleCallableStatement)cstmt).getCursor(1);

// Use the cursor like a normal ResultSet
while (cursor.next ())
    {System.out.println (cursor.getString(1));}
```

この例について、説明します。

- CallableStatement オブジェクトは、接続クラスの prepareCall() メソッドを使用して作成されます。
- コール可能文によって、REF CURSOR を戻す PL/SQL プロシージャが実装されます。
- コール可能文の出力パラメータは、通常どおり登録してその型を定義する必要があります。REF CURSOR ではタイプコード OracleTypes.CURSOR を使用します。
- コール可能文が実行され、REF CURSOR が戻されます。
- 標準 JDBC API の Oracle 拡張機能である getCursor() メソッドを使用するため、CallableStatement オブジェクトを OracleCallableStatement オブジェクトにキャストします。REF CURSOR が ResultSet オブジェクトに戻されます。

REF CURSOR を使用した完全なサンプル・アプリケーションについては、20-36 ページの「REF CURSOR: RefCursorExample.java」を参照してください。

8.0.x および 7.3.x JDBC ドライバでの Oracle 拡張機能のサポート

現行の Oracle JDBC ドライバによってサポートされている Oracle 型拡張機能には、Oracle 8.0.x および 7.3.x JDBC ドライバではサポートされていなかったり、サポートに違いがある機能があります。次に主要な点を示します。

- 8.0.x および 7.3.x ドライバには、oracle.sql パッケージがありません。つまり実際の SQL データをラップするために使用できる oracle.sql.NUMBER および oracle.sql.CHAR などのラッパー型はありません。
- 8.0.x および 7.3.x ドライバは、Oracle オブジェクトおよびコレクション型をサポートしません。
- 8.0.x および 7.3.x ドライバは、oracle.jdbc パッケージの OracleRowid クラスで、ROWID データ型をサポートします。
- 8.0.x ドライバは、oracle.jdbc パッケージの OracleBlob、OracleClob および OracleBfile クラスで、Oracle BLOB、CLOB および BFILE データ型をサポートしています。これらのクラスには、LOB および BFILE の操作メソッドは含まれていないため、かわりに PL/SQL DBMS_LOB パッケージを使用する必要があります。
- 7.3.x ドライバは、BLOB、CLOB および BFILE をサポートしていません。

表 5-4 では、これらの相違点について要約します。OracleTypes 定義とは、`oracle.jdbc.OracleTypes` クラスで定義される静的タイプコード定数のことです。

表 5-4 8.0.x および 7.3.x JDBC ドライバでの Oracle 型拡張機能のサポート

Oracle データ型	OracleTypes 定義	カレント・ドライバの型拡張機能	8.0.x / 7.3.x ドライバの型拡張機能
NUMBER	OracleTypes.NUMBER	oracle.sql.NUMBER	ラッパー・クラスに対する型拡張機能はありません。
CHAR	OracleTypes.CHAR	oracle.sql.CHAR	ラッパー・クラスに対する型拡張機能はありません。
RAW	OracleTypes.RAW	oracle.sql.RAW	ラッパー・クラスに対する型拡張機能はありません。
DATE	OracleTypes.DATE	oracle.sql.DATE	ラッパー・クラスに対する型拡張機能はありません。
ROWID	OracleTypes.ROWID	oracle.sql.ROWID	oracle.jdbc.driver.OracleRowid。
BLOB	OracleTypes.BLOB	oracle.sql.BLOB	8.0.x では oracle.jdbc.driver.OracleBlob です。 7.3.x ではサポートされません。
CLOB	OracleTypes.CLOB	oracle.sql.CLOB	8.0.x では oracle.jdbc.driver.OracleClob です。 7.3.x ではサポートされません。
BFILE	利用不可	oracle.sql.BFILE	8.0.x では oracle.jdbc.driver.OracleBfile です。 7.3.x ではサポートされません。
構造化オブジェクト	OracleTypes.STRUCT	oracle.sql.STRUCT またはカスタム・クラス	サポートされません。
オブジェクト参照	OracleTypes.REF	oracle.sql.REF またはカスタム・クラス	サポートされません。
コレクション（配列）	OracleTypes.ARRAY	oracle.sql.ARRAY またはカスタム・クラス	サポートされません。
OPAQUE	OracleTypes.OPAQUE	oracle.sql.OPAQUE	サポートされません。

6

Oracle データへのアクセスと操作

この章では、oracle.sql.* 形式でのデータ・アクセスについて、標準 Java 形式と対比させて説明します。前の章で説明したように、oracle.sql.* 形式は Oracle JDBC 拡張要素のキーとなる要因であり、SQL データを操作する際の効率と精度を大きく向上させます。

oracle.sql.* 形式の使用には、結果セットおよび文を適宜 OracleResultSet、OracleStatement、OraclePreparedStatement、OracleCallableStatement オブジェクトにキャストすること、およびこれらのクラスの getOracleObject()、setOracleObject()、getXXX()、setXXX()（XXX は oracle.sql パッケージの型に対応）メソッドを使用することが関係しています。

次の項目が含まれます。

- データ変換での考慮事項
- 結果セットと文拡張要素
- Oracle の get および set メソッドと標準 JDBC の比較
- 結果セット・メタデータ拡張要素の使用方法

データ変換での考慮事項

JDBC プログラムが SQL データを Java に取り込む場合は、標準 Java 型または oracle.sql パッケージの型を使用できます。このパッケージ内のクラスは、単純に実際の SQL データをラップします。

標準型と Oracle 型

処理速度と作業に関して、oracle.sql.* クラスは SQL データを表すための最も効率的な方法です。これらのクラスは、通常の SQL データ表現をバイト配列として格納します。データ形式の再設定やキャラクタ・セットの変換（通常のネットワーク変換を除く）は行われません。データは SQL 形式のままであるため、情報が失われることはありません。SQL プリミティブ型（NUMBER や CHAR など）の場合、oracle.sql.* クラスは SQL データを単にラップするのみです。SQL 構造化型（オブジェクトや配列など）の場合、クラスは変換方法や構造の詳細などの追加情報を提供します。

データベース内でデータを移動する場合は、大抵データを oracle.sql.* 形式のままにしておきます。データを表示していたり、データベースの外部で動作する Java アプリケーションでデータを計算している場合は、大抵データを java.sql.* 型または java.lang.* 型などの標準型のインスタンスとしてインスタンス化します。同様に、使用するパーサーが標準 Java 形式のデータを前提としている場合、oracle.sql.* 形式ではなく、いずれかの標準形式を使用する必要があります。

SQL NULL データの変換

Java は、SQL NULL データを、Java の値 null で表現します。Java データ型は、プリミティブ型（byte、int、float など）とオブジェクト型（クラス・インスタンス）の 2 つのカテゴリに分類されます。プリミティブ型で NULL を表すことはできません。かわりに、NULL を 0（ゼロ）値（JDBC 仕様で定義されているように）として格納します。これは、結果の解釈時にあいまいさが生じる原因ともなります。

一方、Java オブジェクト型は NULL を表現できます。Java 言語は、NULL を表現可能な各プリミティブ型に対応したオブジェクト・ラッパー型（たとえば、int には Integer、float には Float）を定義します。オブジェクト・ラッパー型は、SQL データが SQL NULL をあいまいさを残さずに検出するためのターゲットとして使用する必要があります。

結果セットと文拡張要素

JDBC Statement オブジェクトは、java.sql.ResultSet として型指定された OracleResultSet オブジェクトを戻します。標準 JDBC メソッドのみをオブジェクトに適用する場合は、オブジェクトの ResultSet 型を維持してください。ただし、オブジェクト上で Oracle 拡張機能を使用する場合、Oracle 拡張機能を OracleResultSet 型にキャストする必要があります。Java コンパイラがオブジェクトの変更を識別するために使用する型は変更されますが、オブジェクト自体は変更されません。

たとえば、標準 Statement オブジェクト stmt があり、標準 JDBC ResultSet メソッドのみを使用する場合、次のように記述します。

```
ResultSet rs = stmt.executeQuery("SELECT * FROM emp");
```

Oracle 拡張機能が JDBC に提供する拡張機能で必要とする場合、前述のように結果を標準 ResultSet オブジェクト内で選択してから、そのオブジェクトを OracleResultSet オブジェクトにキャストします。

同様に、コール可能文を使用してストアド・プロシージャを実行する場合、JDBC ドライバは、java.sql.CallableStatement として型指定された OracleCallableStatement オブジェクトを戻します。標準 JDBC メソッドのみをオブジェクトに適用する場合は、オブジェクトの CallableStatement 型を維持してください。ただし、オブジェクトで Oracle 拡張機能を使用する場合は、オブジェクトを OracleCallableStatement 型にキャストする必要があります。Java コンパイラがオブジェクトの変更を識別するために使用する型は変更されますが、オブジェクト自体は変更されません。

PreparedStatement オブジェクトを作成するには、標準 JDBC java.sql.Connection.prepareStatement() メソッドを使用します。標準 JDBC メソッドのみをオブジェクトに適用する場合は、オブジェクトの PreparedStatement 型を維持してください。ただし、オブジェクト上で Oracle 拡張機能を使用する場合、オブジェクトを OraclePreparedStatement 型にキャストする必要があります。Java コンパイラがオブジェクトの変更を識別するために使用する型は変更されますが、オブジェクト自体は変更されません。

結果セットおよび文クラスへの主要拡張要素には、getOracleObject() および setOracleObject() メソッドが含まれます。これらのメソッドを使用すると、標準 Java 形式のかわりに oracle.sql.* 形式でデータへのアクセスおよび操作を実行できます。詳細は、「Oracle の get および set メソッドと標準 JDBC の比較」を参照してください。

Oracle の get および set メソッドと標準 JDBC の比較

この項では、get および set メソッド、特に JDBC 標準 getObject() と setObject() メソッドおよび Oracle 固有の getOracleObject() と setOracleObject() メソッドについて説明します。また、oracle.sql.* 形式のデータへのアクセス方法を Java 形式と比較しながら説明します。

Oracle SQL 型はすべて、対応する特定の getXXX() メソッドを持ちますが（6-7 ページの「他の getXXX() メソッド」を参照）、便宜上や簡素化の目的で、または受け取るデータの型が不明な場合に、一般的な get メソッドも使用できます。

標準 getObject() メソッド

結果セットまたはコール可能文の標準 JDBC getObject() メソッドは、データを java.lang.Object オブジェクト内に戻します。戻されるデータ形式は、次に示すようにオリジナルの型に基づきます。

- Oracle 固有ではない SQL データ型の場合、JDBC 仕様で指定されたマッピングに従い、列の SQL 型に対応するデフォルトの Java 型を戻します。
- Oracle 固有のデータ型（5-32 ページの「Oracle ROWID 型」で説明した ROWID など）の場合、適切な oracle.sql.* クラス（oracle.sql.ROWID など）のオブジェクトを戻します。
- Oracle オブジェクトの場合、使用する型マップで指定された Java クラスのオブジェクトを戻します。（型マップは、Java クラスとデータベース SQL 型の相関関係を指定します。型マップの詳細は、8-11 ページの「SQLData を実装するための型マップ」を参照してください。）getObject(*parameter_index*) メソッドは、接続のデフォルト型マップを使用します。getObject(*parameter_index*, *map*) を使用すると、型マップでの引渡しが可能です。型マップが特定の Oracle オブジェクトのマッピングを提供しない場合、getObject() は oracle.sql.STRUCT オブジェクトを戻します。

getObject() 戻り型の詳細は、6-6 ページの表 6-1「getObject() および getOracleObject() 戻り型のまとめ」を参照してください。

Oracle getOracleObject() メソッド

結果セットまたはコール可能文から oracle.sql.* オブジェクトにデータを取り出す場合、結果セットを OracleResultSet 型にキャストするか、コール可能文を OracleCallableStatement 型にキャストして、getOracleObject() メソッドを使用します。

getOracleObject() を使用する場合、データは適切な oracle.sql.* 型になり、oracle.sql.Datum オブジェクト内に戻されます（oracle.sql 型クラスは Datum を拡張します）。メソッドのシグネチャは次のとおりです。

```
public oracle.sql.Datum getOracleObject(int parameter_index)
```

Datum オブジェクト内にデータを取り出した場合は、標準 Java instanceof 演算子を使用して、実際の oracle.sql.* 型を決定します。

getOracleObject() 戻り型の詳細は、6-6 ページの表 6-1「getObject() および getOracleObject() 戻り型のまとめ」を参照してください。

例 : 結果セットでの getOracleObject() の使用方法 次の例では、文字データ列（この場合は行番号）を含む表および BFILE ロケータを含む列を作成します。SELECT 文では、表の内容を結果セットに取り出します。getOracleObject() は CHAR データを char_datum 変数に、BFILE ロケータを bfile_datum 変数に取り出します。getOracleObject() は Datum オブジェクトを戻すため、結果をそれぞれ CHAR および BFILE にキャストする必要があります。

```
stmt.execute ("CREATE TABLE bfile_table (x varchar2 (30), b bfile)");
stmt.execute
    ("INSERT INTO bfile_table VALUES ('one', bfilename ('TEST_DIR', 'file1'))");

ResultSet rset = stmt.executeQuery ("SELECT * FROM bfile_table");
while (rset.next ())
{
   CHAR char_datum = (CHAR) ((OracleResultSet)rset).getOracleObject (1);
   BFILE bfile_datum = (BFILE) ((OracleResultSet)rset).getOracleObject (2);
   ...
}
```

例 : コール可能文での getOracleObject() の使用方法 次の例では、文字列（この場合は名前）を日付と対応付けるプロシージャ myGetDate() へのコールを作成します。このプログラムは、文字列 SCOTT を準備済みコールに渡し、DATE 型を出力パラメータとして登録します。コールの実行後、getOracleObject() は名前 SCOTT と対応付けられた日付を取り出します。getOracleObject() は Datum オブジェクトを戻すため、結果セットは DATE オブジェクトにキャストされる点に留意してください。

```
OracleCallableStatement cstmt = (OracleCallableStatement)conn.prepareCall
                                  ("begin myGetDate (?, ?); end;");

cstmt.setString (1, "SCOTT");
cstmt.registerOutParameter (2, Types.DATE);
cstmt.execute ();

DATE date = (DATE) ((OracleCallableStatement)cstmt).getOracleObject (2);
...
```

getObject() および getOracleObject() 戻り型のまとめ

表 6-1 は、6-4 ページの「標準 getObject() メソッド」および「Oracle getOracleObject() メソッド」で説明した情報を表にまとめたものです。

表に、各 Oracle SQL 型の各メソッドに対応した、基礎となる戻り型も示します。ただし、コードを記述する際、メソッドのシグネチャを念頭に置く必要があります。

- `getObject()`: データを常に `java.lang.Object` インスタンスに戻します。
- `getOracleObject()`: データを常に `oracle.sql.Datum` インスタンスに戻します。

いずれかの特殊機能を使用するためには、戻されたオブジェクトをキャストする必要があります（6-10 ページの「get メソッドの戻り値のキャスト」を参照してください）。

表 6-1 getObject() および getOracleObject() 戻り型のまとめ

Oracle SQL 型	getObject() 基礎となる戻り型	getOracleObject() 基礎となる戻り型
CHAR	String	oracle.sql.CHAR
VARCHAR2	String	oracle.sql.CHAR
LONG	String	oracle.sql.CHAR
NUMBER	java.math.BigDecimal	oracle.sql.NUMBER
RAW	byte[]	oracle.sql.RAW
LONGRAW	byte[]	oracle.sql.RAW
DATE	java.sql.Timestamp	oracle.sql.DATE
ROWID	oracle.sql.ROWID	oracle.sql.ROWID
REF CURSOR	java.sql.ResultSet	（未サポート）
BLOB	oracle.sql.BLOB	oracle.sql.BLOB
CLOB	oracle.sql.CLOB	oracle.sql.CLOB
BFILE	oracle.sql.BFILE	oracle.sql.BFILE
Oracle オブジェクト	型マップで指定されたクラス または oracle.sql.STRUCT （型マップにエントリがない場合）	oracle.sql.STRUCT
Oracle オブジェクト参照	oracle.sql.REF	oracle.sql.REF
コレクション（VARRAY または NESTED TABLE）	oracle.sql.ARRAY	oracle.sql.ARRAY

すべての SQL および Java 型間の型互換性については、21-2 ページの表 21-1「有効な SQL データ型 ― Java クラス・マッピング」を参照してください。

他の getXXX() メソッド

標準 JDBC では、各標準 Java 型に対応した getXXX()（getByte()、getInt()、getFloat() など）が提供されます。これらの各メソッドは、そのメソッド名が意味するもの（byte、int、float など）を正確に戻します。

さらに、OracleResultSet および OracleCallableStatement クラスは、すべての oracle.sql.* 型に対応した getXXX() メソッドを完全に補完します。各 getXXX() メソッドは、oracle.sql.XXX オブジェクトを戻します。たとえば、getROWID() メソッドは、oracle.sql.ROWID オブジェクトを戻します。

これらの拡張要素には、JDBC 2.0 仕様から取り込まれたものもあります。これらは oracle.sql.* ではなく、java.sql.*（JDK 1.1.x では oracle.jdbc2.*）型のオブジェクトを戻します。たとえば、次のメソッド名と戻り型を比較してください。

```
java.sql.Blob getBlob(int parameter_index)

oracle.sql.BLOB getBLOB(int parameter_index)
```

特定の getXXX() メソッドを使用してもパフォーマンス面でのメリットはありませんが、これらのメソッドは特定のオブジェクト型を戻すため、キャストによるトラブルを防ぐことができます。

getXXX() メソッドの戻り型と入力パラメータ型

表 6-2 では、getXXX() メソッドごとに、基礎となる戻り型と入力パラメータ型の概要を示し、どれが JDK 1.2.x および JDK 1.1.x で Oracle 拡張機能であるかを示します。Oracle 拡張機能であるメソッドを使用するには、OracleResultSet または OracleCallableStatement をキャストします。

表 6-2 getXXX() 戻り型のまとめ

メソッド	基礎となる戻り型	シグネチャ型	JDK 1.2.x? での Oracle 拡張機能	JDK 1.1.x? での Oracle 拡張機能
getArray()	oracle.sql.ARRAY	java.sql.Array （JDK 1.1.x では oracle.jdbc2.Array）	なし	あり
getARRAY()	oracle.sql.ARRAY	oracle.sql.ARRAY	あり	あり
getAsciiStream()	java.io.InputStream	java.io.InputStream	なし	なし
getBfile()	oracle.sql.BFILE	oracle.sql.BFILE	あり	あり

表 6-2 getXXX() 戻り型のまとめ （続き）

メソッド	基礎となる戻り型	シグネチャ型	JDK 1.2.x? での Oracle 拡張機能	JDK 1.1.x? での Oracle 拡張機能
getBFILE()	oracle.sql.BFILE	oracle.sql.BFILE	あり	あり
getBigDecimal()（後述の注意事項を参照）	java.math.BigDecimal	java.math.BigDecimal	なし	なし
getBinaryStream()	java.io.InputStream	java.io.InputStream	なし	なし
getBlob()	oracle.sql.BLOB	java.sql.Blob（JDK 1.1.x では oracle.jdbc2.Blob）	なし	あり
getBLOB	oracle.sql.BLOB	oracle.sql.BLOB	あり	あり
getBoolean()	boolean	boolean	なし	なし
getByte()	byte	byte	なし	なし
getBytes()	byte[]	byte[]	なし	なし
getCHAR()	oracle.sql.CHAR	oracle.sql.CHAR	あり	あり
getCharacterStream()	java.io.Reader	java.io.Reader	なし	あり
getClob()	oracle.sql.CLOB	java.sql.Clob（JDK 1.1.x では oracle.jdbc2.Clob）	なし	あり
getCLOB()	oracle.sql.CLOB	oracle.sql.CLOB	あり	あり
getDate()（後述の注意事項を参照）	java.sql.Date	java.sql.Date	なし	なし
getDATE()	oracle.sql.DATE	oracle.sql.DATE	あり	あり
getDouble()	double	double	なし	なし
getFloat()	float	float	なし	なし
getInt()	int	int	なし	なし
getLong()	long	long	なし	なし
getNUMBER()	oracle.sql.NUMBER	oracle.sql.NUMBER	あり	あり

表 6-2 getXXX() 戻り型のまとめ （続き）

メソッド	基礎となる戻り型	シグネチャ型	JDK 1.2.x? での Oracle 拡張機能	JDK 1.1.x? での Oracle 拡張機能
getOracleObject()	oracle.sql.Datum のサブクラス	oracle.sql.Datum	あり	あり
getRAW()	oracle.sql.RAW	oracle.sql.RAW	あり	あり
getRef()	oracle.sql.REF	java.sql.Ref （JDK 1.1.x では oracle.jdbc2.Ref）	なし	あり
getREF()	oracle.sql.REF	oracle.sql.REF	あり	あり
getROWID()	oracle.sql.ROWID	oracle.sql.ROWID	あり	あり
getShort()	short	short	なし	なし
getString()	String	String	なし	なし
getSTRUCT()	oracle.sql.STRUCT.	oracle.sql.STRUCT	あり	あり
getTime() （後述の注意事項を参照）	java.sql.Time	java.sql.Time	なし	なし
getTimestamp() （後述の注意事項を参照）	java.sql.Timestamp	java.sql.Timestamp	なし	なし
getUnicodeStream()	java.io.InputStream	java.io.InputStream	なし	なし

getXXX() メソッドに関する特別な注意

この項では、いくつかの `getXXX()` メソッドについて追加の詳細情報を示します。

getBigDecimal() に関する注意事項

JDBC 2.0 は、`getBigDecimal()` メソッドのために単純化されたメソッド・シグネチャをサポートします。以前の入力シグネチャは次のいずれかでした。

（`int columnIndex, int scale`）または（`String columnName, int scale`）

新しい入力シグネチャは次のように簡略化されています。

（`int columnIndex`）または（`String columnName`）

小数点の右にある数字の数を指定するために使用する `scale` パラメータは不要になりました。Oracle JDBC ドライバは、完全な精度で数値を取り出します。

getDate()、getTime() および getTimestamp() に関する注意事項

JDBC 2.0 では、getDate()、getTime() および getTimestamp() メソッドには次の入力シグネチャがあります。

```
(int columnIndex, Calendar cal)
```

または

```
(String columnName, Calendar cal)
```

Oracle JDBC ドライバは、Calendar オブジェクト入力を無視します。これは、現在そのデータとともに java.sql.Date タイムゾーン情報をサポートできないためです。列索引または列名のみを取る以前の入力シグネチャを引き続き使用してください。カレンダ入力は、将来の Oracle JDBC リリースでサポートされます。

get メソッドの戻り値のキャスト

6-4 ページの「標準 getObject() メソッド」で説明したように、Oracle による実装では、getObject() は常に java.lang.Object インスタンスを戻し、getOracleObject() は常に oracle.sql.Datum インスタンスを戻します。通常、戻されたオブジェクトを適切なクラスにキャストすることにより、そのクラスの特定のメソッドおよび機能を使用可能にします。

また、汎用の getObject() や getOracleObject() メソッドのかわりに、特定の getXXX() メソッドも使用できます。getXXX() の戻り型は、戻されるオブジェクトの型に対応しているため、getXXX() メソッドを使用することにより、キャストを回避できます。たとえば、getCLOB() は、java.lang.Object インスタンスではなく、oracle.sql.CLOB インスタンスを戻します。

例 : 戻り値のキャスト この例では、CHAR 型のデータを結果セット内（この例では列 1）にフェッチしたものとします。CHAR データの精度を保持したまま操作するため、結果セットを OracleResultSet にキャストし、getOracleObject() を使用して、oracle.sql.* 形式で CHAR データを戻します。結果セットをキャストしない場合、getObject() を使用する必要があります。このメソッドは、文字データを Java String に戻すため、SQL データの精度がいくらか失われます。

getOracleObject() メソッドは、oracle.sql.CHAR オブジェクトを oracle.sql.Datum 戻り変数内に戻します。CHAR 戻り変数を使用し、そのクラスの特殊機能のいずれか（文字表現に使用するキャラクタ・セットを戻す getCharacterSet() メソッドなど）を使用する場合は、getOracleObject() の出力を oracle.sql.CHAR にキャストします。

```
CHAR char = (CHAR)ors.getOracleObject(1);
CharacterSet cs = char.getCharacterSet();
```

あるいは、汎用の oracle.sql.Datum 戻り変数にオブジェクトを戻しておき、後で CHAR getCharacterSet() メソッドを使用する必要が生じたときにこのオブジェクトをキャストする方法もあります。

```
Datum rawdatum = ors.getOracleObject(1);
...
CharacterSet cs = ((CHAR)rawdatum).getCharacterSet();
```

この例では、oracle.sql.CHAR の getCharacterSet() メソッドを使用しています。getCharacterSet() メソッドは oracle.sql.Datum では定義されていないため、キャストしない場合は到達不可能です。

標準 setObject() および Oracle setOracleObject() メソッド

結果セットおよびデータ取出し用のコール可能文に、標準 getObject() および Oracle 固有の getOracleObject() が存在するように、Oracle のプリコンパイルされた SQL 文およびデータ更新用のコール可能文には標準 setObject() および Oracle 固有の setOracleObject() が存在します。setOracleObject() メソッドは、oracle.sql.* を入力パラメータとして取ります。

標準 Java 型をプリコンパイルされた SQL 文またはコール可能文にバインドするには、setObject() メソッドを使用します。このメソッドは java.lang.Object を入力として取ります。setObject() メソッドは、少数の oracle.sql.* 型をサポートします。このメソッドは、oracle.sql.* クラスのインスタンスも入力できるように実装されています。oracle.sql.* クラスは、BLOB、CLOB、BFILE、STRUCT、REF および ARRAY の JDBC の 2.0 準拠の Oracle 拡張機能です。

oracle.sql.* 型をプリコンパイルされた SQL 文またはコール可能文にバインドするには、setOracleObject() メソッドを使用します。このメソッドは oracle.sql.Datum（または任意のサブクラス）を入力として取ります。setOracleObject() を使用するには、プリコンパイルされた SQL 文またはコール可能文を OraclePreparedStatement または OracleCallableStatement オブジェクトにキャストする必要があります。

例 : プリコンパイルされた SQL 文での setObject() と setOracleObject() の使用方法 この例は、文字データを標準結果セット内（この例では列 1）にフェッチしたことを前提にしています。また、Oracle 固有の形式とメソッドを使用するために、結果を OracleResultSet にキャストするものとします。データを oracle.sql.CHAR 形式で使用するため、getOracleObject()（oracle.sql.Datum 型を戻す）の結果を CHAR にキャストします。同様に、列 2 のデータを文字列として操作するため、データを Java String 型にキャストします（getObject() は Object 型のデータを戻すため）。この例では、rs には結果セットが、charVal には列 1 の oracle.sql.CHAR 形式のデータが、strVal には列 2 の Java String 形式のデータが当てはまります。

```
CHAR charVal=(CHAR)((OracleResultSet)rs).getOracleObject(1);
String strVal=(String)rs.getObject(2);
...
```

プリコンパイルされた SQL 文オブジェクト ps では、setOracleObject() メソッドは、charVal 変数で表現された oracle.sql.CHAR データをプリコンパイルされた SQL 文にバインドします。oracle.sql.* データをバインドするには、プリコンパイルされた SQL 文を OraclePreparedStatement にキャストする必要があります。同様に、setObject() メソッドは、変数 strVal で表現された Java String データをバインドします。

```
PreparedStatement ps= conn.prepareStatement("text_of_prepared_statement");
((OraclePreparedStatement)ps).setOracleObject(1,charVal);
ps.setObject(2,strVal);
```

他の setXXX() メソッド

getXXX() メソッドの場合と同様、固有の setXXX() メソッドがいくつか存在します。標準 setXXX() メソッドは、標準 Java 型のバインド用に提供されます。また、Oracle 固有の setXXX() メソッドは、Oracle 固有の型のバインド用に提供されます。

注意： JDK 1.1.x では、JDBC 2.0 標準との互換性を維持するため、OraclePreparedStatement クラスと OracleCallableStatement クラスは setXXX() メソッドを提供します。このメソッドは、BLOB、CLOB、オブジェクト参照および配列用の oracle.jdbc2 入力パラメータを取ります。たとえば、setBlob() メソッドは、JDK 1.2.x では java.sql.Blob 入力パラメータを取るのに対して、oracle.jdbc2.Blob 入力パラメータを取ります。

同様に、setNull() メソッドには次の 2 つの形式があります。

- void setNull(int *parameterIndex,* int *sqlType*)

  これは、標準の java.sql.PreparedStatement インタフェースで指定されるシグネチャです。このシグネチャは、java.sql.Types または oracle.jdbc.OracleTypes クラスによって定義されるパラメータ索引と SQL タイプコードを取得します。REF、ARRAY または STRUCT 以外のオブジェクトを NULL に設定する場合は、このシグネチャを使用します。

- void setNull(int *parameterIndex*, int *sqlType*, String *sql_type_name*)

  JDBC 2.0 の場合、このシグネチャは標準の java.sql.PreparedStatement インタフェースにも指定されます。JDK 1.1.x の場合は、Oracle 拡張機能として使用可能です。このメソッドは、パラメータ索引および SQL タイプコードに加え、SQL 型名を取ります。このメソッドは、SQL タイプコードが java.sql.Types.REF、ARRAY または STRUCT の場合に使用します。（タイプコードが REF、ARRAY または STRUCT 以外の場合は、指定された SQL 型は無視されます。）

同様に、registerOutParameter() メソッドには、REF、ARRAY または STRUCT データとともに使用するためのシグネチャがあります。

```
void registerOutParameter
                (int parameterIndex, int sqlType, String sql_type_name)
```

Oracle 固有型のバインドのために、標準 Java 型のバインドのためのメソッドではなく、適切な固有 setXXX() メソッドを使用すると、多少パフォーマンスが向上することがあります。

setXXX() メソッドの入力パラメータ型

表 6-3 では、すべての setXXX() メソッドの入力型の概要を示し、JDK 1.2.x と JDK 1.1.x でどれが Oracle 拡張機能であるかを示します。Oracle 拡張機能のメソッドを使用するには、OraclePreparedStatement または OracleCallableStatement に文をキャストする必要があります。

表 6-3 setXXX() 入力パラメータ型のまとめ

メソッド	入力パラメータ型	JDK 1.2.x? での Oracle 拡張機能	JDK 1.1.x? での Oracle 拡張機能
setArray()	java.sql.Array （JDK 1.1.x では oracle.jdbc2.Array）	なし	あり
setARRAY()	oracle.sql.ARRAY	あり	あり
setAsciiStream() （後述の「追加の入力を取得する setter メソッド」を参照）	java.io.InputStream	なし	なし
setBfile()	oracle.sql.BFILE	あり	あり
setBFILE()	oracle.sql.BFILE	あり	あり
setBigDecimal()	BigDecimal	なし	なし
setBinaryStream() （後述の「追加の入力を取得する setter メソッド」を参照）	java.io.InputStream	なし	なし
setBlob()	java.sql.Blob （JDK 1.1.x では oracle.jdbc2.Blob）	なし	あり
setBLOB()	oracle.sql.BLOB	あり	あり
setBoolean()	boolean	なし	なし

表 6-3 setXXX() 入力パラメータ型のまとめ （続き）

メソッド	入力パラメータ型	JDK 1.2.x? での Oracle 拡張機能	JDK 1.1.x? での Oracle 拡張機能
setByte()	byte	なし	なし
setBytes()	byte[]	なし	なし
setCHAR() （後述の「WHERE 句に CHAR データをバインドするためのメソッド setFixedCHAR()」を参照）	oracle.sql.CHAR	あり	あり
setCharacterStream() （後述の「追加の入力を取得する setter メソッド」を参照）	java.io.Reader	なし	あり
setClob()	java.sql.Clob （JDK 1.1.x では oracle.jdbc2.Clob）	なし	あり
setCLOB()	oracle.sql.CLOB	あり	あり
setDate() （後述の「追加の入力を取得する setter メソッド」を参照）	java.sql.Date	なし	なし
setDATE()	oracle.sql.DATE	あり	あり
setDouble()	double	なし	なし
setFixedCHAR() （後述の「WHERE 句に CHAR データをバインドするためのメソッド setFixedCHAR()」を参照）	java.lang.String	あり	あり
setFloat()	float	なし	なし
setInt()	int	なし	なし
setLong()	long	なし	なし
setNUMBER()	oracle.sql.NUMBER	あり	あり
setRAW()	oracle.sql.RAW	あり	あり
setRef()	java.sql.Ref （JDK 1.1.x では oracle.jdbc2.Ref）	なし	あり

表 6-3 setXXX() 入力パラメータ型のまとめ （続き）

メソッド	入力パラメータ型	JDK 1.2.x? での Oracle 拡張機能	JDK 1.1.x? での Oracle 拡張機能
setREF()	oracle.sql.REF	あり	あり
setROWID()	oracle.sql.ROWID	あり	あり
setShort()	short	なし	なし
setString()	String	なし	なし
setSTRUCT()	oracle.sql.STRUCT	あり	あり
setTime() （後述の「追加の入力を取得する setter メソッド」を参照）	java.sql.Time	なし	なし
setTimestamp() （後述の「追加の入力を取得する setter メソッド」を参照）	java.sql.Timestamp	なし	なし
setUnicodeStream() （後述の「追加の入力を取得する setter メソッド」を参照）	java.io.InputStream	なし	なし

SQL および Java 型でサポートされているすべての型マッピングについては、21-2 ページの表 21-1「有効な SQL データ型 ― Java クラス・マッピング」を参照してください。

Oracle8 と Oracle7 での setter メソッドのサイズ制限

表 6-4 には、Oracle8 および Oracle7 データベースで SQL をバインドするための `setBytes()` メソッドと `setString()` メソッドのサイズ制限をリストします。（これらの制限は、PL/SQL バインドには適用されません。）ストリーム API を使用してこれらの制限を回避する方法については、3-30 ページの「setBytes() と setString() への制限を回避するためのストリームの使用方法」を参照してください。

表 6-4 setBytes() メソッドと setString() メソッドのサイズ制限

	Oracle8	Oracle7
setBytes() サイズ制限	2000 バイト	255 バイト
setString() サイズ制限	4000 バイト	2000 バイト

追加の入力を取得する setter メソッド

次の setXXX() メソッドは、パラメータ索引とデータ項目自体の他に、追加の入力パラメータを取ります。

- setAsciiStream(int paramIndex, InputStream istream, int length)

  ストリーム長をバイト単位で取得します。

- setBinaryStream(int paramIndex, InputStream istream, int length)

  ストリーム長をバイト単位で取得します。

- setCharacterStream(int paramIndex, Reader reader, int length)

  ストリーム長を文字数で取得します。

- setUnicodeStream(int paramIndex, InputStream istream, int length)

  ストリーム長をバイト単位で取得します。

特に、LONGVARCHAR パラメータへの入力が非常に大きな Unicode 値の場合は、java.io.Reader オブジェクトを通じて送信したほうが実用的であるため、setCharacterStream() メソッドが役立ちます。JDBC はファイル終りマークに達するまで、必要に応じてストリームからデータを読み込みます。JDBC ドライバは、必要に応じて Unicode をデータベース・キャラクタ形式に変換します。

重要： 前述のストリーム・メソッドは、LOB にも使用できます。詳細は、7-6 ページの「BLOB および CLOB データの読込みと書込み」を参照してください。

- setDate(int paramIndex, Date x, Calendar cal)
- setTime(int paramIndex, Time x, Calendar cal)
- setTimestamp(int paramIndex, Timestamp x, Calendar cal)

  setDate()、setTime() および setTimestamp() の JDBC 2.0 シグネチャには、Calendar オブジェクトが含まれています。ただし、このデータとともに java.sql.Date タイムゾーン情報をサポートできないため、Oracle JDBC ドライバはこの入力を無視します。パラメータ索引とデータ項目のみを取る以前のシグネチャを引き続き使用してください。カレンダ入力は、将来のリリースでサポートされます。

WHERE 句に CHAR データをバインドするためのメソッド setFixedCHAR()

データベース内の CHAR データは、列幅まで埋め込まれます。このため、SELECT 文の WHERE 句に文字データをバインドするための setCHAR() メソッドの使用に関して、制限が生じます。つまり、WHERE 句の文字データも、SELECT 文で合致させるために、列幅まで埋め込む必要があります。これは特に列幅がわからない場合に問題になります。

これを修正するために、Oracle は OraclePreparedStatement クラスに setFixedCHAR() メソッドを追加しました。このメソッドは埋込みなしの比較を実行します。

注意：

- setFixedCHAR() メソッドを使用するには、必ずプリコンパイルされた SQL 文オブジェクトを OraclePreparedStatement にキャストしてください。
- INSERT 文で、setFixedCHAR() を使用する必要はありません。データベースは挿入時に、常にそのデータを列幅まで自動的に埋め込みます。

例： 次の例では、setCHAR() メソッドと setFixedCHAR() メソッドの違いを示します。

```
/* Schema is :
 create table my_table (col1 char(10));
 insert into my_table values ('JDBC');
*/
 PreparedStatement pstmt = conn.prepareStatement
                    ("select count(*) from my_table where col1 = ?");

 pstmt.setString (1, "JDBC");  // Set the Bind Value
 runQuery (pstmt);             // This will print " No of rows are 0"

 CHAR ch = new CHAR("JDBC      ", null);
 ((OraclePreparedStatement)pstmt).setCHAR(1, ch); // Pad it to 10 bytes
 runQuery (pstmt);             // This will print "No of rows are 1"

 ((OraclePreparedStatement)pstmt).setFixedCHAR(1, "JDBC");
  runQuery (pstmt);            // This will print "No of rows are 1"

 void runQuery (PreparedStatement ps)
 {
   // Run the Query
   ResultSet rs = pstmt.executeQuery ();

   while (rs.next())
     System.out.println("No of rows are " + rs.getInt(1));
```

```
    rs.close();
    rs = null;
  }
```

Oracle 8.0.x と 7.3.x JDBC ドライバの制限

Oracle 8.0.x JDBC ドライバは、Oracle 7.3.x JDBC ドライバと同じプロトコルを使用します。どちらの場合にも、Oracle データ型は Oracle 7.3.x データベースでの定義に従って、2KB を超えるデータ項目は LONG にする必要があります。

他の LONG データと同様に、アプリケーションとデータベース間でデータの読み書きを実行するにはストリーム API を使用します。実質上、これは 8.0.x および 7.3.x ドライバを使用している場合は、2KB を超えるデータを読み書きするために、通常の getString() メソッドと setString() メソッドを使用できないことを示します。

ストリーム API には、getBinaryStream()、setBinaryStream()、getAsciiStream() および setAsciiStream() などのメソッドが含まれます。これらのメソッドについては、3-19 ページの「JDBC 内の Java ストリーム」で説明します。

結果セット・メタデータ拡張要素の使用方法

oracle.jdbc.driver.OracleResultSetMetaData インタフェースは JDBC 2.0 に準拠していますが、基本となるプロトコルによってサポートされていないため、getSchemaName() メソッドと getTableName() メソッドは実装されません。ただし、Oracle は、Oracle 結果セットに関する情報を取り出すために、多くのメソッドを実装しています。

主要メソッドは次のとおりです。

- int getColumnCount(): Oracle 結果セット内の列数を戻します。
- String getColumnName(int column): Oracle 結果セット内の指定された列名を戻します。
- int getColumnType(int column): Oracle 結果セット内の指定された列の SQL 型を戻します。列が Oracle オブジェクトまたはコレクションを格納している場合、このメソッドは、それぞれ OracleTypes.STRUCT または OracleTypes.ARRAY を戻します。
- String getColumnTypeName(int column): 指定された REF、STRUCT または ARRAY 型列の SQL 型名を戻します。列が配列またはコレクションを格納している場合、このメソッドはその SQL 型名を戻します。列が REF データを格納している場合、このメソッドはオブジェクト参照が指すオブジェクトの SQL 型名を戻します。

次の例は、OracleResultSetMetadata インタフェースのメソッドをいくつか使用して、EMP 表からの列数、および各列の数値型と SQL 型名を取り出します。

```
DatabaseMetaData dbmd = conn.getMetaData();
ResultSet rset = dbmd.getTables("", "SCOTT", "EMP", null);

 while (rset.next())
 {
   OracleResultSetMetaData orsmd = ((OracleResultSet)rset).getMetaData();
   int numColumns = orsmd.getColumnCount();
   System.out.println("Num of columns = " + numColumns);

   for (int i=0; i<numColumns; i++)
   {
     System.out.print ("Column Name=" + orsmd.getColumnName (i+1));
     System.out.print (" Type=" + orsmd.getColumnType (i + 1) );
     System.out.println (" Type Name=" + orsmd.getColumnTypeName (i + 1));
  }
}
```

このプログラムは次の出力を戻します。

```
Num of columns = 5
Column Name=TABLE_CAT Type=12 Type Name=VARCHAR2
Column Name=TABLE_SCHEM Type=12 Type Name=VARCHAR2
Column Name=TABLE_NAME Type=12 Type Name=VARCHAR2
Column Name=TABLE_TYPE Type=12 Type Name=VARCHAR2
Column Name=TABLE_REMARKS Type=12 Type Name=VARCHAR2
```

7

LOB と BFILE の操作

この章では、JDBC および `oracle.sql.*` クラスを使用して、LOB ロケータとデータおよび BFILE ロケータとデータにアクセスし、操作する方法を説明します。次の項目が含まれます。

- LOB および BFILE の Oracle 拡張機能
- BLOB と CLOB の操作
- BFILE の操作

LOB および BFILE の Oracle 拡張機能

LOB（ラージ・オブジェクト）は、領域を最適化し効率的にアクセスできるように格納されます。JDBC ドライバは、BLOB（非構造化バイナリ・データ）と CLOB（文字データ）の 2 種類の LOB をサポートします。BLOB および CLOB データは、ロケータを使用してアクセスおよび参照します。ロケータはデータベース表内に格納され、表の外部にある BLOB または CLOB データを指します。

BFILE は、データベース表領域外のオペレーティング・システム・ファイルに格納されたラージ・バイナリ・データ・オブジェクトです。これらのファイルは、参照セマンティクスを利用します。また、これらのファイルはハード・ディスク、CD-ROM、PhotoCD および DVD などの 3 次記憶装置上にも配置できます。BLOB や CLOB と同様、BFILE は、ロケータによりアクセスまたは参照されます。ロケータは、データベース表内に格納され、BFILE データを指します。

LOB データを使用するには、まず LOB ロケータを取得する必要があります。その後、LOB データの読込みまたは書込み、データ操作が実行できます。次の項では、表内で LOB 列を作成および移入する方法も説明します。

JDBC ドライバは、BLOB、CLOB および BFILE に対応した、次の `oracle.sql.*` クラスをサポートします。

- `oracle.sql.BLOB`
- `oracle.sql.CLOB`
- `oracle.sql.BFILE`

`oracle.sql.BLOB` および `CLOB` クラスは、それぞれ `java.sql.Blob` および `Clob` インタフェースを実装します（JDK 1.1.x では、`oracle.jdbc2.Blob` および `Clob` インタフェース）。これに対し、`BFILE` は Oracle 拡張機能で、対応する `java.sql`（または `oracle.jdbc2`）インタフェースはありません。

これらのクラスのインスタンスは、これらのデータ型のロケータのみを含み、データは含みません。ロケータへのアクセス後に、データにアクセスをするために追加処理を行う必要があります。これらの処理については、7-6 ページの「BLOB および CLOB データの読込みと書込み」および 7-21 ページの「BFILE データの読込み」を参照してください。

注意： JDBC アプリケーションでは、`BLOB`、`CLOB` または `BFILE` オブジェクトを作成できません。データベースから既存の BLOB、CLOB または BFILE を取り出すか、`createTemporary()` メソッドおよび `empty_lob()` メソッドを使用してオブジェクトを作成することができます。

BLOB と CLOB の操作

この項では、LOB ロケータを使用して、Oracle データベースのバイナリ・ラージ・オブジェクト（Binary Large Object: BLOB）とキャラクタ・ラージ・オブジェクト（Character Large Object: CLOB）に対し、データの読込みおよび書込みを行う方法を説明します。

Oracle9*i* LOB とその使用方法については、『Oracle9*i* アプリケーション開発者ガイド－ラージ・オブジェクト』を参照してください。

BLOB および CLOB ロケータの取出しと引渡し

データベースから LOB ロケータを取り出すとき、またはデータベースに LOB ロケータを渡すときに使用可能な getter および setter メソッドには、標準のメソッドと Oracle 固有のメソッドがあります。

BLOB および CLOB ロケータの取出し

BLOB または CLOB ロケータを含む標準 JDBC 結果セット（`java.sql.ResultSet`）またはコール可能文（`java.sql.CallableStatement`）がある場合、次の標準 getter メソッドを使用して、ロケータにアクセスできます。この項で説明する標準および Oracle 固有の getter メソッドはすべて、入力として `int` 列索引または `String` 列名を取ります。

- JDK 1.2.x では、標準の `getBlob()` および `getClob()` メソッドを使用できます。これらのメソッドはそれぞれ、`java.sql.Blob` および `Clob` オブジェクトを戻します。
- JDK 1.1.x には、標準の BLOB または CLOB 機能がありません。しかし、`java.lang.Object` を戻す汎用の `getObject()` メソッドを使用して、出力を必要に応じてキャストできます。

結果セットまたはコール可能文を、`OracleResultSet` または `OracleCallableStatement` オブジェクトに取り出すかキャストすると、次のように Oracle 拡張機能を使用できます。

- JDK 1.2.x または JDK 1.1.x のどちらでも、`getBLOB()` および `getCLOB()` メソッドを使用できます。これらのメソッドはそれぞれ、`oracle.sql.BLOB` および `CLOB` オブジェクトを戻します。
- JDK 1.2.x または JDK 1.1.x のどちらでも、`oracle.sql.Datum` オブジェクトを戻す `getOracleObject()` メソッドを使用して、出力を適切にキャストできます。
- JDK 1.1.x では、Oracle 拡張機能の `getBlob()` および `getClob()` も使用できます。これらのメソッドはそれぞれ、`oracle.jdbc2.Blob` および `Clob` オブジェクトを戻します。（これらの `Blob` および `Clob` インタフェースは、JDK 1.2.x で使用可能な標準インタフェースを擬似実行します。）

注意： `getObject()` または `getOracleObject()` を使用する場合、必要に応じて出力をキャストしてください。詳細は、6-10 ページの「get メソッドの戻り値のキャスト」を参照してください。

例 : 結果セットからの BLOB および CLOB ロケータの取出し データベースに `lob_table` という表があり、その中に BLOB ロケータ用の列 `blob_col`、および CLOB ロケータ用の列 `clob_col` があるとします。この例では、`Statement` オブジェクト `stmt` が作成済みであるものとします。

まず、LOB ロケータを標準結果セット内に Select します。その後、LOB データを適切な Java クラス内に取り込みます。

```
// Select LOB locator into standard result set.
ResultSet rs =
   stmt.executeQuery ("SELECT blob_col, clob_col FROM lob_table");
while (rs.next())
{
   // Get LOB locators into Java wrapper classes.
   java.sql.Blob blob = (java.sql.Blob)rs.getObject(1);
   java.sql.Clob clob = (java.sql.Clob)rs.getObject(2);
   (...process...)
}
```

出力は、`java.sql.Blob` および `Clob` にキャストします。また、出力を `oracle.sql.BLOB` および `CLOB` にキャストして、`oracle.sql.*` クラスの提供する拡張機能を利用する方法もあります。たとえば、前述のコードを次のように書き直して、LOB ロケータを取り出すこともできます。

```
   // Get LOB locators into Java wrapper classes.
   oracle.sql.BLOB blob = (BLOB)rs.getObject(1);
   oracle.sql.CLOB clob = (CLOB)rs.getObject(2);
   (...process...)
```

例 : コール可能文からの CLOB ロケータの取出し LOB 取出し用のコール可能文は、結果セットのメソッドと同一です。

たとえば、CLOB 出力パラメータを取る関数 `func` をコールする `OracleCallableStatement ocs` がある場合、次のようにコール可能文を設定します。

この例では、出力パラメータのタイプコードとして、`OracleTypes.CLOB` が登録されます。

```
OracleCallableStatement ocs =
   (OracleCallableStatement)conn.prepareCall("{? = call func()}");
ocs.registerOutParameter(1, OracleTypes.CLOB);
ocs.execute();
oracle.sql.CLOB clob = ocs.getCLOB(1);
```

BLOB および CLOB ロケータの引渡し

標準 JDBC のプリコンパイルされた SQL 文（`java.sql.PreparedStatement`）またはコール可能文（`java.sql.CallableStatement`）がある場合、次の標準 setter メソッドを使用して、LOB ロケータを渡せます。この項で説明する標準および Oracle 固有の setter メソッドはすべて、入力として `int` パラメータ索引および LOB ロケータを取ります。

- JDK 1.2.x では、標準の `setBlob()` および `setClob()` メソッドを使用できます。これらのメソッドはそれぞれ、入力として `java.sql.Blob` および `Clob` ロケータを取ります。
- JDK 1.1.x には、標準の BLOB または CLOB 機能がありません。しかし、`java.lang.Object` の入力を指定する汎用の `setObject()` メソッドを使用できます。

Oracle 固有の `OraclePreparedStatement` または `OracleCallableStatement` がある場合、次のように Oracle 拡張機能を使用できます。

- JDK 1.2.x または JDK 1.1.x のどちらでも、`setBLOB()` および `setCLOB()` メソッドを使用できます。これらのメソッドはそれぞれ、入力として `oracle.sql.BLOB` および `CLOB` ロケータを取ります。
- JDK 1.2.x または JDK 1.1.x のどちらでも、`oracle.sql.Datum` の入力を指定する `setOracleObject()` メソッドを使用できます。
- JDK 1.1.x でも、Oracle 拡張機能の `setBlob()` および `setClob()` も使用できます。これらのメソッドはそれぞれ、入力として `oracle.jdbc2.Blob` および `Clob` ロケータを取ります。（これらの `Blob` および `Clob` インタフェースは、JDK 1.2.x で使用可能な標準インタフェースを擬似実行します。）

例 : BLOB ロケータのプリコンパイルされた SQL 文への引渡し
OraclePreparedStatement オブジェクト ops と、my_blob という名前の BLOB がある場合、次のように BLOB をデータベースに書き込みます。

```
OraclePreparedStatement ops = (OraclePreparedStatement)conn.prepareStatement
                              ("INSERT INTO blob_table VALUES(?)");
ops.setBLOB(1, my_blob);
ops.execute();
```

例 : CLOB ロケータのコール可能文への引渡し OracleCallableStatement オブジェクト ocs と、my_clob という名前の CLOB がある場合、次のように CLOB をストアド・プロシージャ proc に入力します。

```
OracleCallableStatement ocs =
   (OracleCallableStatement)conn.prepareCall("{call proc(?)}");
ocs.setClob(1, my_clob);
ocs.execute();
```

BLOB および CLOB データの読込みと書込み

LOB ロケータがあれば、JDBC メソッドを使用して、LOB データの読込みおよび書込みができます。LOB データは、Java 配列またはストリームとしてインスタンス化されます。ただし、大抵の Java ストリームとは異なり、LOB データを表すロケータは表に格納されます。このため、接続されている間、LOB データにはいつでもアクセスできます。

LOB データの読込みおよび書込みを行う場合、oracle.sql.BLOB または oracle.sql.CLOB クラスのメソッドを適宜使用します。これらのクラスは、LOB から入力ストリーム内への読込み、出力ストリームから LOB 内への書込み、LOB の長さの決定および LOB のクローズなどの機能を提供します。

注意 :

- LOB データを書き込むには、アプリケーションで LOB オブジェクトの書込みロックを取得する必要があります。SELECT FOR UPDATE を使用することで取得できます。また、自動コミット・モードを無効化します。
- データ・アクセス API の実装は、JDBC OCI およびサーバー側内部ドライバのダイレクト・ネイティブ・コールを使用するため、パフォーマンスが向上します。すべての Oracle JDBC ドライバの LOB クラスで、同じ API を使用できます。
- JDBC Thin ドライバの場合に限り、データ・アクセス API の実装は、PL/SQL DBMS_LOB パッケージを内部で使用します。DBMS_LOB を直接使用する必要はまったくありません。これは、8.0.*x* ドライバと対照的です。DBMS_LOB パッケージの詳細は、『Oracle9*i* PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。

LOB データの読込みおよび書込みを実行する際、次のメソッドを使用できます。

- BLOB から読み込むには、oracle.sql.BLOB オブジェクトの getBinaryStream() メソッドを使用して、BLOB 全体を入力ストリームとして取り出します。これにより、java.io.InputStream オブジェクトが戻されます。

  InputStream オブジェクトの場合と同様に、オーバーロードされた read() メソッドの 1 つを使用して LOB データを読み込み、完了時に close() メソッドを使用します。

- BLOB に書き込むには、oracle.sql.BLOB オブジェクトの getBinaryOutputStream() メソッドを使用して、BLOB を出力ストリームとして取り出します。このメソッドは、BLOB に書き戻される java.io.OutputStream オブジェクトを戻します。

  OutputStream オブジェクトの場合と同様に、オーバーロードされた write() メソッドの 1 つを使用して、LOB データを更新し、完了時に close() メソッドを使用します。

- CLOB から読み込むには、oracle.sql.CLOB オブジェクトの getAsciiStream() または getCharacterStream() メソッドを使用して、CLOB 全体を入力ストリームとして取り出します。getAsciiStream() メソッドは、java.io.InputStream オブジェクトの ASCII 入力ストリームを戻します。getCharacterStream() メソッドは、java.io.Reader オブジェクトの Unicode 入力ストリームを戻します。

  InputStream または Reader オブジェクトの場合と同様、オーバーロードされた read() メソッドの 1 つを使用して、LOB データを読み込み、完了時に close() メソッドを使用します。

  oracle.sql.CLOB オブジェクトの getSubString() メソッドを使用して、CLOB のサブセットを java.lang.String 型の文字列として取り出せます。

- CLOB に書き込むには、oracle.sql.CLOB オブジェクトの getAsciiOutputStream() または getCharacterOutputStream() メソッドを使用して、CLOB を出力ストリームとして取り出してから、CLOB に書き戻します。getAsciiOutputStream() メソッドは、java.io.OutputStream オブジェクトの ASCII 出力ストリームを戻します。getCharacterOutputStream() メソッドは、java.io.Writer オブジェクトの Unicode 出力ストリームを戻します。

  OutputStream または Writer オブジェクトの場合と同様、オーバーロードされた write() メソッドの 1 つを使用して、LOB データを更新し、完了時に flush() および close() メソッドを使用します。

注意 :

- この項で説明しているストリームの書込み用メソッドは、出力ストリームへの書込み時に、直接データベースへの書込みを行います。データの書込みに UPDATE を実行する必要はありません。
- CLOB および BLOB はトランザクションによって制御されます。CLOB または BLOB に対して書込みを実行後、変更を永続化するためにはそのトランザクションをコミットする必要があります。BFILE はトランザクション制御型ではありません。BFILE への書込みを行うと、外部ファイル・システムによって別の操作が行われないかぎり、トランザクションがロールバックされても、変更は永続化されます。
- CLOB に対して書込みまたは読込みを実行する際、JDBC ドライバがすべてのキャラクタ・セット変換を実行します。

重要 : JDBC 2.0 の仕様では、PreparedStatement のメソッド setBinaryStream() および setObject() を使用して、ストリーム値を BLOB として入力でき、また、PreparedStatement のメソッド setAsciiStream()、setUnicodeStream()、setCharacterStream() および setObject() を使用して、ストリーム値を CLOB として入力できると規定されています。これにより、LOB ロケータを通さず、直接 LOB データ自体にアクセスできます。

Oracle JDBC ドライバの実装では、この機能はリリース 8.1.6 以上のデータベースおよび 8.1.6 以上の JDBC OCI ドライバを使用した構成でのみサポートされます。**その他の構成では、この機能を使用しないでください。データが破損する可能性があります。**

例 : BLOB データの読込み oracle.sql.BLOB クラスの getBinaryStream() メソッドを使用して、BLOB データを読み込みます。getBinaryStream() メソッドは、BLOB データをバイナリ・ストリーム内に読み込みます。

次の例は、getBinaryStream() メソッドを使用して BLOB データをバイト・ストリーム内に読み込み、次にバイト・ストリームを読み込んでバイト配列に格納します（読み込んだバイト数も戻します）。

```
// Read BLOB data from BLOB locator.
InputStream byte_stream = my_blob.getBinaryStream();
byte [] byte_array = new byte [10];
int bytes_read = byte_stream.read(byte_array);
...
```

例 : CLOB データの読込み 次の例では、getCharacterStream() メソッドを使用して、CLOB データを Unicode 文字ストリーム内に読み込みます。次に、文字ストリームを読み込んで文字配列に格納します（読み込んだ文字数も戻します）。

```
// Read CLOB data from CLOB locator into Reader char stream.
Reader char_stream = my_clob.getCharacterStream();
char [] char_array = new char [10];
int chars_read = char_stream.read (char_array, 0, 10);
...
```

次の例では、oracle.sql.CLOB クラスの getAsciiStream() メソッドを使用して、CLOB データを ASCII 文字ストリーム内に読み込みます。次に、ASCII ストリームを読み込んでバイト配列に格納します（読み込んだ文字数も戻します）。

```
// Read CLOB data from CLOB locator into Input ASCII character stream
Inputstream asciiChar_stream = my_clob.getAsciiStream();
byte[] asciiChar_array = new byte[10];
int asciiChar_read = asciiChar_stream.read(asciiChar_array,0,10);
```

例 : BLOB データの書込み oracle.sql.BLOB オブジェクトの getBinaryOutputStream() メソッドを使用して、BLOB データを書き込みます。

次の例では、データのベクトルをバイト配列内に読み込み、getBinaryOutputStream() メソッドを使用して、文字データの配列を BLOB に書き込みます。

```
java.io.OutputStream outstream;

// read data into a byte array
byte[] data = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

// write the array of binary data to a BLOB
outstream = ((BLOB)my_blob).getBinaryOutputStream();
outstream.write(data);
...
```

例 : CLOB データの書込み getCharacterOutputStream() メソッドまたは getAsciiOutputStream() メソッドを使用して、データを CLOB に書き込みます。getCharacterOutputStream() メソッドは Unicode 出力ストリームを戻し、getAsciiOutputStream() メソッドは ASCII 出力ストリームを戻します。

次の例では、データのベクトルを文字配列内に読み込み、getCharacterOutputStream() メソッドを使用して文字データの配列を CLOB に書き込みます。getCharacterOutputStream() メソッドは、java.sql.Clob オブジェクトではなく、oracle.sql.CLOB オブジェクトの java.io.Writer インスタンスを戻します。

```
java.io.Writer writer;

// read data into a character array
char[] data = {'0','1','2','3','4','5','6','7','8','9'};

// write the array of character data to a CLOB
writer = ((CLOB)my_clob).getCharacterOutputStream();
writer.write(data);
writer.flush();
writer.close();
...
```

次の例では、データのベクトルをバイト配列内に読み込み、getAsciiOutputStream() メソッドを使用して、ASCII データの配列を CLOB に書き込みます。getAsciiOutputStream() は ASCII 出力ストリームを戻すため、出力を oracle.sql.CLOB データ型にキャストする必要があります。

```
java.io.OutputStream out;

// read data into a byte array
byte[] data = {'0','1','2','3','4','5','6','7','8','9'};

// write the array of ascii data to a CLOB
out = ((CLOB)clob).getAsciiOutputStream();
out.write(data);
out.flush();
out.close();
```

BLOB または CLOB 列の作成と移入

表に BLOB 列または CLOB 列を作成および移入するには、SQL 文を使用します。

注意： Java の new 文を使用して、アプリケーションで新規の BLOB ロケータまたは CLOB ロケータを作成することはできません。SQL 操作でロケータを作成してから、アプリケーションでロケータを選択するか、createTemporary() メソッドまたは empty_lob() メソッドを使用してロケータを選択する必要があります。

SQL の CREATE TABLE 文で表に BLOB 列または CLOB 列を作成し、LOB を移入します。これには、表内での LOB エントリの作成、LOB ロケータの取出し、データ用ファイル・ハンドラの作成（ファイルからデータを読み取る場合）、およびデータの LOB へのコピーが含まれます。

新しい表での BLOB または CLOB 列の作成

新しい表で BLOB または CLOB 列を作成する場合、SQL CREATE TABLE 文を実行します。次の例では、新しい表に BLOB 列を作成します。この例では、Connection オブジェクト conn および Statement オブジェクト stmt を作成してあるものとします。

```
String cmd = "CREATE TABLE my_blob_table (x varchar2 (30), c blob)";
stmt.execute (cmd);
```

この例では、VARCHAR2 列は 1 や 2 などの行番号を表し、BLOB 列には BLOB データのロケータが格納されます。

新しい表での BLOB または CLOB 列の移入

この例では、ストリームからデータを読み取って、BLOB または CLOB 列を移入する方法を示します。Connection オブジェクト conn および Statement オブジェクト stmt は、すでに作成してあるものとします。表 my_blob_table は、前の項で作成した表です。

次の例では、GIF 形式のファイル john.gif を BLOB に書き込みます。

1. まず、SQL 文を使用して BLOB エントリを表の中に作成します。empty_blob 構文を使用して、BLOB ロケータを作成します。

   ```
   stmt.execute ("INSERT INTO my_blob_table VALUES ('row1', empty_blob())");
   ```

2. 表から BLOB ロケータを取り出します。

   ```
   BLOB blob;
   cmd = "SELECT * FROM my_blob_table WHERE X='row1'";
   ResultSet rest = stmt.executeQuery(cmd);
   BLOB blob = ((OracleResultSet)rset).getBLOB(2);
   ```

3. john.gif ファイル用のファイル・ハンドラを宣言し、ファイルの長さを出力します。この値は、ファイル全体が BLOB 内に読み込まれたことを保証するために使用されます。次に、FileInputStream オブジェクトを作成して GIF ファイルの内容を読み込み、OutputStream オブジェクトを作成して BLOB をストリームとして取り出します。

```
File binaryFile = new File("john.gif");
System.out.println("john.gif length = " + binaryFile.length());
FileInputStream instream = new FileInputStream(binaryFile);
OutputStream outstream = blob.getBinaryOutputStream();
```

4. getBufferSize() をコールして、(JDBC ドライバの計算による) BLOB への書込みに使用する理想的なバッファ・サイズを取り出し、buffer バイト配列を作成します。

```
int size = blob.getBufferSize();
byte[] buffer = new byte[size];
int length = -1;
```

5. read() メソッドを使用して GIF ファイルをバイト配列 buffer 内に読み込み、次に write() メソッドを使用してそれを BLOB に書き込みます。完了時に、入力および出力ストリームをクローズします。

```
while ((length = instream.read(buffer)) != -1)
   outstream.write(buffer, 0, length);
instream.close();
outstream.close();
```

データが BLOB または CLOB 内に格納されると、データの操作が可能になります。この詳細は、次の項の「BLOB および CLOB データのアクセスと操作」で説明します。

BLOB および CLOB データのアクセスと操作

BLOB または CLOB ロケータが表内に格納されると、ロケータが指すデータへのアクセスおよび操作が可能になります。データへのアクセスおよび操作を行うには、まずそのロケータを結果セットまたはコール可能文から選択する必要があります。この方法の詳細は、7-3 ページの「BLOB および CLOB ロケータの取出しと引渡し」を参照してください。

ロケータの選択後に、BLOB または CLOB データを取り出せます。通常、結果セットを OracleResultSet データ型にキャストして、データを oracle.sql.* 形式で取り出せるようにします。BLOB または CLOB データの取出し後、任意の方法でデータを操作できます。

この例は、前の項の例からの継続です。SQL SELECT 文を使用して、表 my_blob_table で BLOB ロケータを選択し、結果セットに格納します。データ操作の結果として、BLOB の長さがバイト単位で出力されます。

```
// Select the blob - what we are really doing here
// is getting the blob locator into a result set
BLOB blob;
cmd = "SELECT *  FROM my_blob_table";
ResultSet rset = stmt.executeQuery (cmd);

// Get the blob data - cast to OracleResult set to
// retrieve the data in oracle.sql format
String index = ((OracleResultSet)rset).getString(1);
blob = ((OracleResultSet)rset).getBLOB(2);

// get the length of the blob
int length = blob.length();

// print the length of the blob
System.out.println("blob length" + length);

// read the blob into a byte array
// then print the blob from the array
byte bytes[] = blob.getBytes(1, length);
blob.printBytes(bytes, length);
```

その他の BLOB および CLOB 機能

この章で説明した他に、oracle.sql.BLOB および CLOB クラスには、より高度な機能を実現する多数のメソッドがあります。

注意： oracle.sql.CLOB クラスは、Oracle データ・サーバーが CLOB 型に対してサポートする、すべてのキャラクタ・セットをサポートします。

その他の BLOB メソッド

oracle.sql.BLOB クラスには、次のメソッドが含まれます。

- close(): ロケータと対応付けられた BLOB をクローズします。（詳細は、7-18 ページの「LOB とオープンおよびクローズの使用」を参照してください。）
- freeTemporary(): テンポラリ BLOB によって使用されている記憶域を解放します。（詳細は、7-17 ページの「テンポラリ LOB の使用」を参照してください。）
- getBinaryOutputStream(): BLOB にデータをストリームとして書き込むために java.io.OutputStream を戻します。
- getBinaryOutputStream(long): BLOB にデータをストリームとして書き込むために java.io.OutputStream を戻します。データは、引数で指定された BLOB 内の位置から書き込まれます。
- getBinaryStream(): この Blob インスタンスの BLOB データをバイト・ストリームとして戻します。
- getBinaryStream(long): この Blob インスタンスの BLOB データを、引数で指定された BLOB 内の位置から、バイト・ストリームとして戻します。
- getBufferSize(): JDBC ドライバの計算による、BLOB データの読込みおよび書込みに使用する理想的なバッファ・サイズを戻します。この値は、チャンク・サイズ（後述の getChunkSize() を参照）の倍数で、32K に近い値です。
- getBytes(): 指定された位置から BLOB データを読み込み、提供されたバッファに格納します。
- getChunkSize(): Oracle チャンク・サイズを戻します。これは、データベース管理者が LOB 列を最初に作成するときに指定できます。Oracle ブロックのこの値によって、BLOB 値へのアクセスまたは変更の際、LOB データ・レイヤーで読込みまたは書込みが行われるデータのチャンク・サイズが判断されます。各チャンクの一部にはシステム関連情報が格納され、残りに LOB データが格納されます。読込みおよび書込みのとき、チャンク・サイズの倍数を使用するように要求すると、パフォーマンスが向上します。
- isOpen(): open() メソッドをコールして BLOB がオープンされた場合は TRUE を戻し、それ以外の場合は FALSE を戻します。（詳細は、7-18 ページの「LOB とオープンおよびクローズの使用」を参照してください。）
- isTemporary(): BLOB がテンポラリ BLOB である場合は、TRUE を戻します。（詳細は、7-17 ページの「テンポラリ LOB の使用」を参照してください。）
- length(): BLOB の長さをバイト単位で戻します。
- open(): ロケータと対応付けられた BLOB をオープンします。（詳細は、7-18 ページの「LOB とオープンおよびクローズの使用」を参照してください。）
- open(int): ロケータと対応付けられた BLOB を引数で指定されたモードでオープンします。（詳細は、7-18 ページの「LOB とオープンおよびクローズの使用」を参照してください。）

- `position()`: 指定されたパターンが始まる BLOB 内のバイト位置を判断します。
- `putBytes()`: 提供されたバッファから、指定された位置を始点として BLOB データの書込みを行います。
- `trim(long)`: BLOB の値を引数で指定された長さに切り捨てます。

その他の CLOB メソッド

`oracle.sql.CLOB` クラスには、次のメソッドが含まれます。

- `close()`: ロケータと対応付けられた CLOB をクローズします。（詳細は、7-18 ページの「LOB とオープンおよびクローズの使用」を参照してください。）
- `freeTemporary()`: テンポラリ CLOB によって使用されている記憶域を解放します。（詳細は、7-17 ページの「テンポラリ LOB の使用」を参照してください。）
- `getAsciiOutputStream()`: CLOB にデータをストリームとして書き込むために `java.io.OutputStream` を戻します。
- `getAsciiOutputStream(long)`: CLOB にデータをストリームとして書き込むために `java.io.OutputStream` オブジェクトを戻します。データは、引数で指定された CLOB 内の位置から書き込まれます。
- `getAsciiStream()`: `Clob` オブジェクトにより指定された CLOB 値を ASCII バイト・ストリームとして戻します。
- `getAsciiStream(long)`: CLOB オブジェクトによって指定された CLOB 値を、引数で指定された CLOB 内の位置から、ASCII バイト・ストリームとして戻します。
- `getBufferSize()`: JDBC ドライバの計算による、CLOB データの読込みおよび書込みに使用する理想的なバッファ・サイズを戻します。この値は、チャンク・サイズ（後述の `getChunkSize()` を参照）の倍数で、32K に近い値です。
- `getCharacterOutputStream()`: CLOB にデータをストリームとして書き込むために `java.io.Writer` を戻します。
- `getCharacterOutputStream(long)`: CLOB にデータをストリームとして書き込むために `java.io.Writer` オブジェクトを戻します。データは、引数で指定された CLOB 内の位置から書き込まれます。
- `getCharacterStream()`: CLOB データを `Unicode` 文字のストリームとして戻します。
- `getCharacterStream(long)`: CLOB データを引数で指定された CLOB 内の位置から、Unicode 文字のストリームとして戻します。
- `getChars()`: CLOB データの指定された位置から文字を取出し、文字配列に格納します。

- `getChunkSize()`: Oracle チャンク・サイズを戻します。これは、データベース管理者が LOB 列を最初に作成するときに指定できます。Oracle ブロックのこの値によって、CLOB 値へのアクセスまたは変更の際、LOB データ・レイヤーで読込みまたは書込みが行われるデータのチャンク・サイズが判断されます。各チャンクの一部にはシステム関連情報が格納され、残りに LOB データが格納されます。読込みおよび書込みのとき、チャンク・サイズの倍数を使用するように要求すると、パフォーマンスが向上します。
- `isOpen()`: open() メソッドをコールして CLOB がオープンされた場合は `TRUE` を戻し、それ以外の場合は `FALSE` を戻します。（詳細は、7-18 ページの「LOB とオープンおよびクローズの使用」を参照してください。）
- `isTemporary()`: CLOB がテンポラリ CLOB である場合は、TRUE を戻します。（詳細は、7-17 ページの「テンポラリ LOB の使用」を参照してください。）
- `length()`: CLOB の長さを文字数で戻します。
- `open()`: ロケータと対応付けられた CLOB をオープンします。（詳細は、7-18 ページの「LOB とオープンおよびクローズの使用」を参照してください。）
- `open(int)`: ロケータと対応付けられた CLOB を引数で指定されたモードでオープンします。（詳細は、7-18 ページの「LOB とオープンおよびクローズの使用」を参照してください。）
- `position()`: CLOB 内で、指定された部分文字列の始まる文字位置を判断します。
- `putChars()`: 文字配列から、CLOB データ内の指定された位置へ文字を書き込みます。
- `getSubString()`: CLOB データ内の指定された位置から部分文字列を取り出します。
- `putString()`: 文字列を CLOB データ内の指定された位置へ書き込みます。
- `trim(long)`: CLOB の値を引数で指定された長さに切り捨てます。

空の LOB の作成

データを内部 LOB に書き込む前に、LOB 列および属性が `NULL` でないことを必ず確認してください。LOB 列には、ロケータを含む必要があります。`oracle.sql.BLOB` および `oracle.sql.CLOB` クラスで定義された次の `empty_lob()` メソッドを使用して、`INSERT` 文または `UPDATE` 文で内部 LOB を空の LOB として初期化することによって実現できます。

- `public static BLOB empty_lob() throws SQLException`
- `public static CLOB empty_lob() throws SQLException`

JDBC ドライバは、データベース・ラウンドトリップを行わずに空の LOB インスタンスを作成します。空の LOB は、次のメソッド、オブジェクトおよび要素で使用できます。

- OraclePreparedStatement クラスの setXXX() メソッド
- 更新可能な結果セットの updateXXX() メソッド
- STRUCT オブジェクトの属性
- ARRAY オブジェクトの要素

> **注意：** empty_lob() メソッドではロケータを含まない特別なマーカーが作成されるため、JDBC アプリケーションではこのマーカーを読み書きできません。データベースに格納する前に JDBC アプリケーションで空の LOB を読取りまたは書込みしようとすると、JDBC ドライバでは例外 ORA-17098「空の LOB 操作は無効です。」が発生します。

テンポラリ LOB の使用

テンポラリ LOB は、一時データの格納に使用できます。データは、通常の表領域ではなく、一時表領域に格納されます。不要になったテンポラリ LOB は解放する必要があります。解放しないと、LOB によって使用されている一時表領域は再生されません。

テンポラリ LOB は、oracle.sql.BLOB および oracle.sql.CLOB クラスに定義された静的メソッド、createTemporary(Connection, boolean, int) を使用して作成します。テンポラリ LOB を解放するには、freeTemporary() メソッドを使用します。

```
public static BLOB createTemporary(Connection conn, boolean isCached, int duration);
public static CLOB createTemporary(Connection conn, boolean isCached, int duration);
```

引数の duration には、oracle.sql.BLOB クラスまたは oracle.sql.CLOB クラスに定義された DURATION_SESSION または DURATION_CALL を使用する必要があります。クライアント・アプリケーションには DURATION_SESSION が適切です。Java ストアド・プロシージャでは、DURATION_SESSION または DURATION_CALL のいずれか適切なほうを使用できます。

isTemporary() メソッドをコールすることで、LOB がテンポラリ LOB かどうかをテストできます。LOB が createTemporary() メソッドをコールして作成された場合、isTemporary() メソッドは TRUE を戻し、それ以外の場合は FALSE を戻します。

テンポラリ LOB を解放するには、freeTemporary() メソッドをコールします。セッションまたはコールを終了する前に、テンポラリ LOB を解放してください。そうしないと、テンポラリ LOB によって使用されている記憶域は再生されません。

注意： テンポラリ LOB を解放しないと、LOB によって使用されている記憶域が使用不可になります。頻繁にテンポラリ LOB を解放しないと、一時表領域が使用不可な LOB 記憶域でいっぱいになってしまいます。

LOB とオープンおよびクローズの使用

LOB をオープンおよびクローズする必要はありません。パフォーマンスを向上させるために、LOB をオープンおよびクローズすることができます。

オープンおよびクローズのコール操作の中に LOB 操作をラップしない場合、LOB を変更するたびに、LOB は暗黙的にオープンおよびクローズされ、ドメイン索引でトリガーが起動します。この場合、LOB を変更するとただちに、LOB のドメイン索引がすべて更新されることに注意してください。したがって、ドメイン LOB 索引は常に有効となり、いつでも使用できます。

オープンおよびクローズ操作に LOB 操作をラップした場合、LOB を変更するたびにトリガーが起動することはありません。そのかわりに、クローズがコールされた時点でドメイン索引でトリガーが起動します。たとえば、`close()` メソッドをコールするまでドメイン索引が更新されないように、アプリケーションを設計することができます。ただし、オープン操作からクローズ操作までの間は、LOB のドメイン索引が有効でなくなります。

LOB をオープンするには、`open()` メソッドまたは `open(int)` メソッドをコールします。その後、その LOB に対応付けられたトリガーを起動することなく、LOB の読込みまたは書込み操作を実行できます。LOB へのアクセスが終了後、`close()` メソッドをコールして LOB をクローズします。LOB をクローズすると、その LOB に対応付けられたトリガーが起動します。`isOpen()` メソッドをコールすると、LOB がオープンまたはクローズしているかどうかを確認できます。`open(int)` メソッドをコールして LOB をオープンした場合は、引数の値が `oracle.sql.BLOB` クラスおよび `oracle.sql.CLOB` クラスに定義された `MODE_READONLY` または `MODE_READWRITE` と等しい必要があります。`MODE_READONLY` を使用して LOB をオープンした場合、その LOB に書込みを試みると SQL 例外が発生します。

注意： トランザクションでオープンしたすべての LOB をクローズする前に、そのトランザクションをコミットしようとすると、エラーが発生します。オープン状態の LOB は廃棄されますが、トランザクションは正常にコミットされます。したがって、LOB に対して行われたすべての変更と、トランザクション内の非 LOB データはコミットされますが、ドメイン索引に対するトリガーは一定ではありません。

BFILE の操作

この項では、ファイル・ロケータを使用して、外部バイナリ・ファイル（BFILE）に対してデータの読込みおよび書込みを行う方法を説明します。

BFILE ロケータの取出しと引渡し

データベースから BFILE ロケータを取り出すとき、またはデータベースに BFILE ロケータを渡すときには、getter および setter メソッドを使用します。

BFILE ロケータの取出し

標準 JDBC 結果セットまたは BFILE ロケータを含むコール可能文オブジェクトがある場合、標準結果セットの `getObject()` メソッドを使用して、ロケータにアクセスできます。このメソッドは、`oracle.sql.BFILE` オブジェクトを返します。

また、結果セットを `OracleResultSet` にキャストするか、コール可能文を `OracleCallableStatement` にキャストし、`getOracleObject()` または `getBFILE()` メソッドを使用することにより、ロケータにアクセスすることもできます。

注意：

- `OracleResultSet` および `OracleCallableStatement` クラスでは、`getBFILE()` と `getBfile()` の両方が `oracle.sql.BFILE` を戻します。BFILE 用の `java.sql` インタフェース（または `oracle.jdbc2` インタフェース）はありません。
- `getObject()` または `getOracleObject()` を使用する場合、必要に応じた出力をキャストすることに留意してください。詳細は、6-10 ページの「get メソッドの戻り値のキャスト」を参照してください。

例：結果セットからの BFILE ロケータの取出し データベースに `bfile_table` という表があり、この表には BFILE ロケータ用の列 `bfile_col` が 1 つ含まれているとします。この例では、`Statement` オブジェクト `stmt` をすでに作成してあるものとします。

BFILE ロケータを選択し、標準結果セット内に select します。結果セットを `OracleResultSet` にキャストすると、`getBFILE()` を使用して BFILE ロケータを取り出せます。

```
// Select the BFILE locator into a result set
ResultSet rs = stmt.executeQuery("SELECT bfile_col FROM bfile_table");
while (rs.next())
{
   oracle.sql.BFILE my_bfile = ((OracleResultSet)rs).getBFILE(1);
}
```

別の方法として、getObject() を使用して BFILE ロケータを戻すこともできます。この場合、getObject() は java.lang.Object を戻すため、結果を BFILE にキャストしてください。たとえば、次のようになります。

```
oracle.sql.BFILE my_bfile = (BFILE)rs.getObject(1);
```

例 : コール可能文からの BFILE ロケータの取出し BFILE 出力パラメータを持つ関数 func をコールする、OracleCallableStatement オブジェクト ocs があるとします。次のコード例は、コール可能文を設定し、出力パラメータを OracleTypes.BFILE として登録し、文を実行し、BFILE ロケータを取り出します。

```
OracleCallableStatement ocs =
   (OracleCallableStatement)conn.prepareCall("{? = call func()}");
ocs.registerOutParameter(1, OracleTypes.BFILE);
ocs.execute();
oracle.sql.BFILE bfile = ocs.getBFILE(1);
```

BFILE ロケータの引渡し

BFILE ロケータをプリコンパイルされた SQL 文またはコール可能文（BFILE ロケータを更新するときなど）に渡すには、次のどちらかの方法を使用します。

- 標準 setObject() メソッドを使用します。

または

- 文を OraclePreparedStatement または OracleCallableStatement にキャストし、setOracleObject() または setBFILE() メソッドを使用します。

これらのメソッドは、パラメータ索引および oracle.sql.BFILE オブジェクトを入力として取ります。

例 : BFILE ロケータのプリコンパイルされた SQL 文への引渡し BFILE ロケータを表に挿入するとき、表にデータを挿入する OraclePreparedStatement オブジェクト ops があるとします。最初の列は文字列（行番号を示す）で、2 番目の列は BFILE です。また、有効な oracle.sql.BFILE オブジェクト（bfile）があります。この場合、次のように、BFILE をデータベースに書き込みます。

```
OraclePreparedStatement ops = (OraclePreparedStatement)conn.prepareStatement
                              ("INSERT INTO my_bfile_table VALUES (?,?)");
ops.setString(1,"one");
ops.setBFILE(2, bfile);
ops.execute();
```

例 : BFILE ロケータのコール可能文への引渡し BFILE ロケータをコール可能文に渡す方法は、プリコンパイルされた SQL 文に渡す方法と同じです。この例では、BFILE ロケータは myGetFileLength() プロシージャに渡され、このプロシージャが BFILE の長さを数値で戻します。

```
OracleCallableStatement cstmt = (OracleCallableStatement)conn.prepareCall
                                ("begin ? := myGetFileLength (?); end;");
try
{
   cstmt.registerOutParameter (1, Types.NUMERIC);
   cstmt.setBFILE (2, bfile);
   cstmt.execute ();
   return cstmt.getLong (1);
}
```

BFILE データの読込み

BFILE データを読み込むには、まず BFILE ロケータを取り出す必要があります。ロケータは、コール可能文からも結果セットからも取出し可能です。詳細は、7-19 ページの「BFILE ロケータの取出しと引渡し」を参照してください。

ロケータを取り出すと、BFILE をオープンせずに、BFILE に対して多数のメソッドを起動できます。たとえば、oracle.sql.BFILE メソッド fileExists() や isFileOpen() を使用して、BFILE があるかどうか、オープンしているかどうかを判断できます。ただし、データの読込みおよび操作を行うには、次のように BFILE をオープンし、クローズする必要があります。

- oracle.sql.BFILE クラスの openFile() メソッドを使用して、BFILE をオープンします。
- 完了後、BFILE クラスの closeFile() メソッドを使用します。

BFILE データは、Java ストリームとしてインスタンス化されます。BFILE から読み込むには、oracle.sql.BFILE オブジェクトの getBinaryStream() メソッドを使用して、ファイル全体を入力ストリームとして取り出します。これにより、java.io.InputStream オブジェクトが戻されます。

InputStream オブジェクトの場合と同様に、オーバーロードされた read() メソッドの 1 つを使用してファイル・データを読み込み、完了時に close() メソッドを使用します。

注意 :

- BFILE は読取り専用です。BFILE へのデータの挿入および書込みはできません。
- BFILE の新規作成に JDBC は使用できません。BFILE は、常に外部的に作成されます。

例 : BFILE データの読込み 次の例では、oracle.sql.BFILE オブジェクトの getBinaryStream() メソッドを使用して、BFILE データをバイト・ストリームに読み込み、その後バイト・ストリームをバイト配列に読み込みます。この例では、BFILE がすでにオープンされているものとします。

```
// Read BFILE data from a BFILE locator
Inputstream in = bfile.getBinaryStream();
byte[] byte_array = new byte{10};
int byte_read = in.read(byte_array);
```

BFILE 列の作成と移入

この項では、SQL 操作を使用して表に BFILE 列を作成し、BFILE の格納場所を指定する方法を説明します。次の例では、Connection オブジェクト conn および Statement オブジェクト stmt がすでに作成してあるものとします。

新しい表での BFILE 列の作成

BFILE データを操作するには、表内に BFILE 列を作成してから、BFILE の格納場所を指定します。BFILE の格納場所を指定するには、SQL CREATE DIRECTORY...AS 文を使用して BFILE の存在するディレクトリの別名を指定します。その後、文を実行します。この例では、ディレクトリの別名は test_dir で、BFILE の格納場所は /home/work です。

```
String cmd;
cmd = "CREATE DIRECTORY test_dir AS '/home/work'";
stmt.execute (cmd);
```

SQL CREATE TABLE 文を使用して BFILE 列を含む表を作成してから、文を実行します。この例では、表の名前は my_bfile_table です。

```
// Create a table containing a BFILE field
cmd = "CREATE TABLE my_bfile_table (x varchar2 (30), b bfile)";
stmt.execute (cmd);
```

この例では、VARCHAR2 列は行番号を示し、BFILE 列には BFILE データのロケータが格納されます。

BFILE 列の移入

SQL INSERT INTO...VALUES 文を使用して VARCHAR2 および BFILE フィールドを移入してから、文を実行します。BFILE 列には、BFILE データを指すロケータが移入されます。BFILE 列を移入するには、bfilename 関数を使用してディレクトリの別名および BFILE のファイル名を指定します。

```
cmd ="INSERT INTO my_bfile_table VALUES ('one', bfilename(test_dir,
                                         'file1.data'))";
stmt.execute (cmd);
cmd ="INSERT INTO my_bfile_table VALUES ('two', bfilename(test_dir,
                                         'jdbcTest.data'))";
stmt.execute (cmd);
```

この例では、ディレクトリの別名は test_dir です。BFILE file1.data のロケータは、行 one の BFILE 列にロードされます。BFILE jdbcTest.data のロケータは、行 two の bfile 列にロードされます。

別の方法として、この時点で行番号用の行および BFILE ロケータを作成し、ロケータの挿入は後で行うこともできます。この場合、行番号を表に挿入し、BFILE ロケータのプレースホルダとして null を挿入します。

```
cmd ="INSERT INTO my_bfile_table VALUES ('three', null)";
stmt.execute(cmd);
```

この例では、three が行番号列に挿入され、プレースホルダとして null が挿入されます。プログラムの後半で、プリコンパイルされた SQL 文を使用して BFILE ロケータを表に挿入します。

まず、有効な BFILE ロケータを取出して、bfile オブジェクトに挿入します。

```
rs = stmt.executeQuery("SELECT b FROM my_bfile_table WHERE x='two'");
rs.next();
oracle.sql.BFILE bfile = ((OracleResultSet)rs).getBFILE(1);
```

その後、プリコンパイルされた SQL 文を作成します。この例では setBFILE() メソッドを使用して BFILE を識別しているため、次のようにプリコンパイルされた SQL 文を OraclePreparedStatement にキャストする必要があります。

```
OraclePreparedStatement ops = (OraclePreparedStatement)conn.prepareStatement
                      (UPDATE my_bfile_table SET b=? WHERE x = 'three');
ops.setBFILE(1, bfile);
ops.execute();
```

これで、行 two と行 three には、同じ BFILE が格納されます。

表の中で使用可能な BFILE ロケータの準備が完了すると、BFILE データへのアクセスおよび操作が可能になります。この詳細は、次の項の「BFILE データへのアクセスと操作」で説明します。

BFILE データへのアクセスと操作

表の中に BFILE ロケータを配置すると、ロケータの指すデータへのアクセスと操作が可能になります。データへのアクセスおよび操作を行うには、まず結果セットまたはコール可能文からロケータを選択する必要があります。

次のコードは、7-23 ページの「BFILE 列の移入」の例から続いています。表の行 two から BFILE のロケータを取り出し、結果セットに挿入します。結果セットを OracleResultSet にキャストして、結果セットに対して oracle.sql.* メソッドを使用可能にします。BFILE に適用されるメソッドには、getDirAlias() や getName() のように、BFILE をオープンする必要がないものもあります。読込み、長さの取出し、表示など、BFILE データを操作するメソッドでは、BFILE をオープンする必要があります。

BFILE データの操作完了時には、BFILE をクローズする必要があります。完全な BFILE の例については、20-37 ページの「BFILE: FileExample.java」を参照してください。

```
// select the bfile locator
cmd = "SELECT *  FROM my_bfile_table WHERE x = 'two'";
rset = stmt.executeQuery (cmd);

if (rset.next ())
    BFILE bfile = ((OracleResultSet)rset).getBFILE (2);

// for these methods, you do not have to open the bfile
println("getDirAlias() = " + bfile.getDirAlias());
println("getName() = " + bfile.getName());
println("fileExists() = " + bfile.fileExists());
println("isFileOpen() = " + bfile.isFileOpen());

// now open the bfile to get the data
bfile.openFile();

// get the BFILE data as a binary stream
InputStream in = bfile.getBinaryStream();
int length ;

// read the bfile data in 6-byte chunks
byte[] buf = new byte[6];

while ((length = in.read(buf)) != -1)
{
    // append and display the bfile data in 6-byte chunks
   StringBuffer sb = new StringBuffer(length);
   for (int i=0; i<length; i++)
      sb.append( (char)buf[i] );
   System.out.println(sb.toString());
}
```

```
// we are done working with the input stream. Close it.
in.close();

// we are done working with the BFILE. Close it.
bfile.closeFile();
```

その他の BFILE 機能

この章で説明した機能の他に、oracle.sql.BFILE クラスには、次のように、より高度な機能を実現する多数のメソッドがあります。

- openFile(): 外部ファイルを読取り専用でアクセスするためにオープンします。
- closeFile(): 外部ファイルをクローズします。
- getBinaryStream(): 外部ファイルの内容をバイト・ストリームとして戻します。
- getBinaryStream(long): 外部ファイルの内容を、引数で指定された外部ファイル内の位置から、バイト・ストリームとして戻します。
- getBytes(): 外部ファイルの指定された位置から読込みを開始し、提供されたバッファに格納します。
- getName(): 外部ファイルの名前を取り出します。
- getDirAlias(): 外部ファイルのディレクトリの別名を取り出します。
- length(): BFILE の長さをバイトで戻します。
- position(): 指定されたバイト・パターンの始まるバイト位置を判断します。
- isFileOpen(): BFILE が（読取り専用でアクセスするために）オープンされているかどうかを判断します。

8

Oracle オブジェクト型の操作

この章では、JDBC のユーザー定義オブジェクト型サポートについて説明します。汎用的な弱い型指定の `oracle.sql.STRUCT` クラスの機能を説明します。また、JDBC 標準 `SQLData` インタフェースまたは Oracle `ORAData` インタフェースを実装するカスタム Java クラスにマップする方法も説明します。JDBC ドライバで SQL 表現の SQLJ オブジェクト型にアクセスする方法についても説明します。

次の項目が含まれます。

- Oracle オブジェクトのマッピング
- Oracle オブジェクト用のデフォルト STRUCT クラスの使用方法
- Oracle オブジェクト用のカスタム・オブジェクト・クラスの作成と使用方法
- オブジェクト型の継承
- JPublisher を使用したカスタム・オブジェクト・クラスの作成
- オブジェクト型の記述
- SQLJ オブジェクト型

注意： Oracle オブジェクト機能については、『Oracle9*i* アプリケーション開発者ガイド ― オブジェクト・リレーショナル機能』を参照してください。

Oracle オブジェクトのマッピング

Oracle オブジェクト型ではデータベースの複合データ構造がサポートされます。たとえば、名前（CHAR 型）、電話番号（CHAR 型）および従業員番号（NUMBER 型）などの属性を持つ Person 型を定義できます。

Oracle では、Oracle オブジェクト機能と JDBC 機能が密接に統合されています。標準の汎用 JDBC 型を使用して Oracle オブジェクトにマップすることも、カスタム Java 型定義クラスを作成してマッピングをカスタマイズすることもできます。このマニュアルでは、Oracle オブジェクトにマップするように作成された Java クラスをカスタム Java クラスと呼びます。さらに限定して、カスタム・オブジェクト・クラスとも呼びます。この呼び方により、オブジェクト参照にマップするカスタム参照クラスや、Oracle コレクションにマップするカスタム・コレクション・クラスと区別します。カスタム・オブジェクト・クラスは、データの読込みおよび書込みを行う標準 JDBC インタフェースまたは Oracle 拡張機能インタフェースを実装できます。

JDBC では、Oracle オブジェクトを特定の Java クラスのインスタンスとしてインスタンス化します。JDBC を使用して Oracle オブジェクトにアクセスするには、主に Oracle オブジェクト用に Java クラスを作成し、そのクラスを移入するという 2 つの処理を行います。次の方法からどちらかを選んで行います。

- JDBC によりオブジェクトを STRUCT としてインスタンス化します。詳細は、8-3 ページの「Oracle オブジェクト用のデフォルト STRUCT クラスの使用方法」を参照してください。

または

- Oracle オブジェクトと Java クラス間のマッピングを明示的に指定します。これにはオブジェクト・データ用に Java クラスをカスタマイズすることも含まれます。そうすることにより、ドライバが指定されたカスタム・オブジェクト・クラスのインスタンスを移入できるようになります。この場合、Java クラスにいくらかの制約が生じます。これらの制約を満たすには、JDBC 標準 java.sql.SQLData インタフェースまたは Oracle 拡張機能 oracle.sql.ORAData インタフェースを実装するようにクラスを定義します。詳細は、8-10 ページの「Oracle オブジェクト用のカスタム・オブジェクト・クラスの作成と使用方法」を参照してください。

  Oracle JPublisher ユーティリティを使用して、カスタム Java クラスを生成できます。

注意： SQLData インタフェースを使用する場合、弱い型指定の java.sql.Struct オブジェクトで十分な場合を除き、Java 型マップを使用して SQL と Java のマッピングを指定する必要があります。詳細は、8-11 ページの「SQLData を実装するための型マップ」を参照してください。

Oracle オブジェクト用のデフォルト STRUCT クラスの使用方法

Oracle オブジェクトの SQL と Java のマッピングを行うカスタム Java クラスを提供しない場合、Oracle JDBC はオブジェクトを `oracle.sql.STRUCT` クラスのインスタンスとしてインスタンス化します。

通常は、SQL データを操作する場合に、カスタム Java オブジェクトのかわりに `STRUCT` オブジェクトを使用します。たとえば、Java アプリケーションを、エンド・ユーザー・アプリケーションではなく、データベースの任意のオブジェクト・データを操作するツールとして使用する場合があります。データベースから選択したデータを `STRUCT` オブジェクトに挿入したり、データベースにデータを挿入するために `STRUCT` オブジェクトを作成できます。`STRUCT` オブジェクトは、データを SQL 形式で維持するため、データを完全に保存します。便利な形式で情報を保存する必要がない場合、`STRUCT` オブジェクトを使用すると、データをより効率的に、より正確に保存できます。

`STRUCT` クラスを使用した SQL オブジェクト・データへのアクセスおよび操作の完全なサンプル・アプリケーションについては、20-28 ページの「弱い型指定のオブジェクト: PersonObject.java」を参照してください。

STRUCT クラス機能

この項では、標準メソッドの機能と `oracle.sql.STRUCT` の Oracle 固有の機能を対比させて説明します。また、`STRUCT` 記述子を説明し、`STRUCT` クラスのメソッド一覧を示すことにより、機能の概要を示します。

標準 java.sql.Struct メソッド

コードを標準 JDBC 2.0 に準拠させる必要がある場合、`java.sql.Struct` インスタンス（JDK 1.1.x では `oracle.jdbc2.Struct`）を使用し、次の標準メソッドを使用します。

- `getAttributes(map)`: 属性値を取り出します。このとき、指定された型マップのエントリから、構造化オブジェクト型である属性をインスタンス化するために使用する Java クラスが判断されます。その他の属性値の Java 型は、基礎となる SQL 型のデータで `getObject()` をコールしたときと同じになります（デフォルト JDBC 型）。
- `getAttributes()`: 前述の `getAttributes(map)` メソッドと同じですが、接続のデフォルト型マップが使用されます。
- `getSQLTypeName()`: この `Struct` が表す Oracle オブジェクト型（`SCOTT.EMPLOYEE` など）の完全修飾名（*`schema.sql_type_name`*）を表す Java `String` を戻します。

Oracle oracle.sql.STRUCT クラス・メソッド

Oracle 定義のメソッドで提供される拡張機能を利用するには、oracle.sql.STRUCT インスタンスを使用します。

oracle.sql.STRUCT クラスは java.sql.Struct インタフェース（JDK 1.1.x では oracle.jdbc2.Struct）を実装し、JDBC 2.0 標準を超える拡張機能を提供します。

STRUCT クラスには、標準 Struct 機能の他に、次のメソッドが含まれます。

- getOracleAttributes(): 属性値を oracle.sql.* オブジェクトの配列として取り出します。
- getDescriptor(): この STRUCT オブジェクトに対応する SQL 型の StructDescriptor オブジェクトを戻します。
- getJavaSQLConnection(): 現行の接続インスタンス（java.sql.Connection）を戻します。
- toJdbc(): 接続のデフォルト型マップを参考にしてマップするクラスを判断し、次に toClass() を使用します。
- toJdbc(map): 指定の型マップを参考にしてマップするクラスを判断し、次に toClass() を使用します。

STRUCT 記述子

STRUCT オブジェクトを作成および使用するには、STRUCT オブジェクトに対応する SQL 型（EMPLOYEE など）の記述子が必要です。記述子とは、oracle.sql.StructDescriptor クラスのインスタンスです。同じ SQL 型に対応する STRUCT オブジェクトがいくつあっても、必要な StructDescriptor オブジェクトは 1 つのみです。

STRUCT 記述子の詳細は、8-5 ページの「STRUCT オブジェクトと記述子の作成」を参照してください。

STRUCT オブジェクトと記述子の作成

この項では、STRUCT オブジェクトと記述子の作成方法を説明し、StructDescriptor クラスの便利なメソッドを一覧で示します。

StructDescriptor および STRUCT オブジェクト作成の手順

この項では、指定された Oracle オブジェクト型の oracle.sql.STRUCT オブジェクトを作成する方法を説明します。STRUCT オブジェクトを作成するには、次を実行します。

1. 指定された Oracle オブジェクト型の StructDescriptor オブジェクトを作成します（まだ作成されていない場合）。
2. StructDescriptor を使用して、STRUCT オブジェクトを作成します。

StructDescriptor は oracle.sql.StructDescriptor クラスのインスタンスで、Oracle オブジェクト（SQL 構造化オブジェクト）の型を記述します。各 Oracle オブジェクト型に必要な StructDescriptor は、1 つのみです。ドライバは、StructDescriptor オブジェクトをキャッシュして、すでに遭遇した型を再作成しなくても済むようにします。

STRUCT オブジェクトを作成するには、先に、指定された Oracle オブジェクト型の StructDescriptor が存在している必要があります。StructDescriptor オブジェクトが存在しない場合、静的 StructDescriptor.createDescriptor() メソッドをコールして作成できます。このメソッドには、Oracle オブジェクト型の SQL 型名と接続オブジェクトを渡す必要があります。

```
StructDescriptor structdesc = StructDescriptor.createDescriptor
                                          (sql_type_name, connection);
```

sql_type_name は、Oracle オブジェクト型の名前（EMPLOYEE など）を含む Java 文字列です。connection は、使用する接続オブジェクトです。

Oracle オブジェクト型の StructDescriptor オブジェクトを作成した後、STRUCT オブジェクトを作成できます。そのためには、StructDescriptor、使用する接続オブジェクト、および STRUCT に含める属性を含む Java オブジェクトの配列を渡します。

```
STRUCT struct = new STRUCT(structdesc, connection, attributes);
```

structdesc には以前に作成した StructDescriptor を、connection には使用する接続オブジェクトを、attributes には java.lang.Object[] 型の配列を指定します。

StructDescriptor のメソッドの使用方法

StructDescriptor は、型オブジェクトとみなすことができます。これは、タイプコード、型名、指定した型と相互変換を行う方法など、オブジェクト型の情報が含まれていることを意味します。任意の Oracle オブジェクト型 1 つに対応する StructDescriptor オブジェクトは 1 つのみであることに注意してください。その後、この記述子を使用して、その型に必要な数の STRUCT オブジェクトを作成します。

StructDescriptor クラスには、次のメソッドが含まれます。

- getName(): Oracle オブジェクトの完全修飾 SQL 型名（*schema.sql_type_name* 形式、CORPORATE.EMPLOYEE など）を戻します。
- getLength(): オブジェクト型のフィールド数を戻します。
- getMetaData(): この型に関するメタデータを戻します（結果セット・オブジェクトの getMetaData() と同様）。戻された ResultSetMetaData オブジェクトには、属性名、属性タイプコードおよび属性の型の詳細情報が含まれています。ResultSetMetaData オブジェクトの列索引は、STRUCT の属性の位置にマップされます。このとき、最初の属性は、索引 1 から始まります。

  getMetaData() の詳細は、8-48 ページの「オブジェクト・メタデータの取出し機能」を参照してください。

シリアル化可能な STRUCT 記述子

8-5 ページの「StructDescriptor および STRUCT オブジェクト作成の手順」で説明したように、STRUCT オブジェクトを作成する場合は、最初に StructDescriptor オブジェクトを作成する必要があります。このオブジェクトを作成するには、StructDescriptor.createDescriptor() メソッドをコールします。oracle.sql.StructDescriptor クラスはシリアル化可能です。つまり、StructDescriptor オブジェクトの状態を出力ストリームへ書き込み、後で使用できます。StructDescriptor オブジェクトを再作成するには、入力ストリームからそのシリアル化可能な状態を読み込みます。この操作をデシリアライズと呼びます。シリアル化された StructDescriptor オブジェクトでは、StructDescriptor.createDescriptor() メソッドをコールする必要はありません。単純に StructDescriptor オブジェクトをデシリアライズするのみです。

オブジェクト型が複雑で、頻繁に変更しない場合は、StructDescriptor オブジェクトをシリアル化することをお薦めします。

デシリアライズによって StructDescriptor オブジェクトを作成する場合は、setConnection() メソッドを使用して、StructDescriptor オブジェクトに適切なデータベース接続インスタンスを指定する必要があります。

次のコードで、StructDescriptor オブジェクトの接続インスタンスを指定します。

```
public void setConnection (Connection conn) throws SQLException
```

注意： JDBC ドライバでは、setConnection() メソッドからの接続オブジェクトが、型記述子の導出元と同じデータベースへ接続しているかどうかが検証されません。

STRUCT オブジェクトと属性の取出し

この項では、Oracle 固有の機能または JDBC 2.0 標準機能を使用して、Oracle オブジェクトとその属性を取り出し、操作する方法を説明します。

注意： JDBC ドライバは、埋込みオブジェクト（STRUCT オブジェクトの属性である STRUCT オブジェクト）を、通常のオブジェクトの場合と同様に、シームレスに処理します。JDBC ドライバがオブジェクトである属性を取り出す場合、同じ変換規則に従って、型マップ（使用可能な場合）またはデフォルトのマッピング（型マップが使用不能な場合）を使用します。

oracle.sql.STRUCT オブジェクトとしての Oracle オブジェクトの取出し

Oracle オブジェクトを直接 oracle.sql.STRUCT インスタンスに取り出すことができます。次の例では、getObject() を使用して表 struct_table の列 1（col1）から NUMBER オブジェクトを取得します。getObject() により Object 型が戻されるため、戻り値は oracle.sql.STRUCT にキャストします。次の例では、Statement オブジェクト stmt はすでに生成されているものとします。

```
String cmd;
cmd = "CREATE TYPE type_struct AS object (field1 NUMBER,field2 DATE)";
stmt.execute(cmd);

cmd = "CREATE TABLE struct_table (col1 type_struct)";
stmt.execute(cmd);

cmd = "INSERT INTO struct_table VALUES (type_struct(10,'01-apr-01'))";
stmt.execute(cmd);

cmd = "INSERT INTO struct_table VALUES (type_struct(20,'02-may-02'))";
stmt.execute(cmd);

ResultSet rs= stmt.executeQuery("SELECT * FROM struct_table");
oracle.sql.STRUCT oracleSTRUCT=(oracle.sql.STRUCT)rs.getObject(1);
```

オブジェクトを STRUCT オブジェクトとして戻す方法が、もう 1 つあります。結果セットを OracleResultSet オブジェクトにキャストし、Oracle 拡張機能 getSTRUCT() メソッドを使用します。

```
oracle.sql.STRUCT oracleSTRUCT=((OracleResultSet)rs).getSTRUCT(1);
```

java.sql.Struct オブジェクトとしての Oracle オブジェクトの取出し

前の例で、getObject() などの標準 JDBC 機能を使用して、データベースから Oracle オブジェクトを java.sql.Struct（JDK 1.1.x では oracle.jdbc2.Struct）のインスタンスとして取り出すこともできます。getObject() により java.lang.Object が戻されるため、メソッドの出力を Struct にキャストする必要があります。たとえば、次のようになります。

```
ResultSet rs= stmt.executeQuery("SELECT * FROM struct_table");
java.sql.Struct jdbcStruct = (java.sql.Struct)rs.getObject(1);
```

oracle.sql 型としての属性の取出し

STRUCT または Struct インスタンスから oracle.sql 型として Oracle オブジェクト属性を取り出すには、次の oracle.sql.STRUCT クラスの getOracleAttributes() メソッドを使用します（Struct インスタンスの場合、STRUCT インスタンスにキャストする必要があります）。

前の例では、次のようになります。

```
oracle.sql.Datum[] attrs = oracleSTRUCT.getOracleAttributes();
```

または

```
oracle.sql.Datum[] attrs =
        ((oracle.sql.STRUCT)jdbcStruct).getOracleAttributes();
```

標準 Java 型としての属性の取出し

STRUCT または Struct インスタンスから標準 Java 型として Oracle オブジェクト属性を取り出すには、次のように標準 getAttributes() メソッドを使用します。

```
Object[] attrs = jdbcStruct.getAttributes();
```

STRUCT オブジェクトの文へのバインド

oracle.sql.STRUCT オブジェクトをプリコンパイルされた SQL 文またはコール可能文にバインドするには、標準の setObject() メソッド（タイプコードを指定）を使用するか、文オブジェクトを Oracle 文オブジェクトにキャストしてから Oracle 拡張機能の setOracleObject() メソッドを使用します。たとえば、次のようになります。

```
PreparedStatement ps= conn.prepareStatement("text_of_prepared_statement");
STRUCT mySTRUCT = new STRUCT (...);
ps.setObject(1, mySTRUCT, Types.STRUCT); //OracleTypes.STRUCT under JDK 1.1.x
```

または

```
PreparedStatement ps= conn.prepareStatement("text_of_prepared_statement");
STRUCT mySTRUCT = new STRUCT (...);
((OraclePreparedStatement)ps).setOracleObject(1, mySTRUCT);
```

STRUCT 自動属性バッファリング

Oracle JDBC ドライバには、STRUCT 属性のバッファリングを有効または無効にするためのパブリック・メソッドが用意されています。（ARRAY 要素のバッファ方法については、10-8 ページの「ARRAY 自動要素バッファリング」を参照してください。）

oracle.sql.STRUCT クラスには、次のメソッドが含まれます。

- public void setAutoBuffering(boolean enable)
- public boolean getAutoBuffering()

setAutoBuffering(boolean) メソッドは自動バッファリングを有効または無効にします。getAutoBuffering() メソッドは、現行の自動バッファリング・モードを戻します。デフォルトでは、自動バッファリングは無効です。

STRUCT 属性に getAttributes() および getArray() メソッドで複数回アクセスする場合（ARRAY データをオーバーフローなく JVM メモリーに格納できると仮定する場合）は、JDBC アプリケーションで自動バッファリングを有効にすることをお薦めします。

重要： 変換した属性をバッファリングすると、JDBC アプリケーションでは、大量のメモリーが消費されます。

自動バッファリングを有効にすると、oracle.sql.STRUCT オブジェクトでは、変換したすべての属性のローカルのコピーが保持されます。このデータが保持されるため、この情報に 2 回目にアクセスするときにはデータ・フォーマット変換処理を実行しなくて済みます。

Oracle オブジェクト用のカスタム・オブジェクト・クラスの作成と使用方法

Oracle オブジェクト用にカスタム・オブジェクト・クラスを作成する場合、型マップのエントリを定義する必要があります。ドライバはこの型マップに従って Oracle オブジェクトに対応するカスタム・オブジェクト・クラスをインスタンス化します。

Oracle オブジェクトとその属性データからカスタム・オブジェクト・クラスのインスタンスを作成し、移入する方法も提供する必要があります。ドライバによって、カスタム・オブジェクト・クラスの読込みおよび移入が実行できる必要があります。また、カスタム・オブジェクト・クラスは、提供する必要があるかどうかにかかわらず、Oracle オブジェクトの属性に対応する `getXXX()` および `setXXX()` メソッドも提供できます。カスタム・クラスの作成と移入、およびドライバの読込み / 書込み機能の設定を行うには、次のインタフェースのいずれかを選択します。

- JDBC 標準 `SQLData` インタフェース
- Oracle が提供する `ORAData` および `ORADataFactory` インタフェース

作成するカスタム・オブジェクト・クラスでは、これらのインタフェースのどちらかを実装する必要があります。`ORAData` インタフェースは、カスタム・オブジェクト・クラスに対応するカスタム参照クラスを実装するときにも使用できます。しかし、`SQLData` インタフェースを使用する場合、使用できるのは、緩い Java の参照型（`java.sql.Ref` または `oracle.sql.REF`）のみです。`SQLData` インタフェースは、SQL オブジェクトのマッピング専用です。

たとえば、データベースに `EMPLOYEE` という Oracle オブジェクト型が含まれており、そのオブジェクト型には `Name`（`CHAR` 型）および `EmpNum`（従業員番号、`NUMBER` 型）という 2 つの属性が設定されているとします。型マップを使用して、`EMPLOYEE` オブジェクトが `JEmployee` というカスタム・オブジェクト・クラスにマップされるように指定します。`JEmployee` クラスでは、`SQLData` または `ORAData` インタフェースのどちらかを実装できます。

カスタム・オブジェクト・クラスは独自に作成できますが、Oracle JPublisher ユーティリティを使用して作成すると便利です。JPublisher は標準 `SQLData` インタフェースと Oracle 固有の `ORAData` インタフェースの両方をサポートしており、どちらかを実装するクラスを生成できます。詳細は、8-44 ページの「JPublisher を使用したカスタム・オブジェクト・クラスの作成」を参照してください。

> **注意：** オブジェクト型を継承するためにカスタム・オブジェクト・クラスを作成する必要がある場合は、8-28 ページの「オブジェクト型の継承」を参照してください。

次の項で、`ORAData` と `SQLData` の機能を比較します。

ORAData と SQLData の利点

2 つのインタフェースのうちどちらを実装するか決定する場合、次の点を考慮してください。

ORAData の利点

- Oracle オブジェクトの型マップ・エントリが必要ありません。
- Oracle 拡張機能に対応しています。
- oracle.sql.STRUCT から ORAData を作成できます。この方法は、ネイティブな Java 型への変換が最小限で済むため、より効率的です。
- toDatum メソッドを使用して、ORAData オブジェクトから、(oracle.sql 形式の) 対応する Datum オブジェクトを取得できます。
- パフォーマンスが向上します。ORAData は、Datum 型を直接使用して動作します。Datum 型は、Oracle オブジェクトを保持するために、ドライバによって使用される内部形式です。

SQLData の利点

- JDBC 標準であるため、コードの移植が容易です。

SQLData インタフェースは、SQL オブジェクトのマッピング専用です。ORAData インタフェースは、より柔軟性が高く、他の SQL 型と同じように SQL オブジェクトをマップし、処理をカスタマイズできます。ORAData オブジェクトは、Oracle データベースの任意のデータ型から作成できます。これは、たとえば、Java の RAW データをシリアル化するときに役立ちます。

SQLData を実装するための型マップ

カスタム・オブジェクト・クラスで SQLData インタフェースを使用する場合、カスタム・オブジェクト・クラスを指定する型マップ・エントリを作成する必要があります。このエントリを使用して、Oracle オブジェクト型 (SQL オブジェクト型) を Java にマップします。接続オブジェクトのデフォルト型マップを使用することも、結果セットからデータを取り出すときに型マップを指定して使用することもできます。ResultSet インタフェースの getObject() メソッドには、型マップを指定するための次のシグネチャがあります。

```
rs.getObject(int columnIndex);
```

または

```
rs.getObject(int columnIndex, Map map);
```

SQLData を使用してカスタム・オブジェクト・クラスを作成する方法については、8-10 ページの「Oracle オブジェクト用のカスタム・オブジェクト・クラスの作成と使用方法」を参照してください。

SQLData 実装を使用する場合、型マップ・エントリを含めないと、オブジェクトはデフォルトで oracle.sql.STRUCT クラスにマップされます。（これに対し、ORAData 実装には独自のマッピング機能があるので、型マップ・エントリが必要ありません。ORAData 実装を使用する場合は、標準 getObject() メソッドではなく、Oracle getORAData() メソッドを使用します。）

型マップを使用して、Java クラスを Oracle オブジェクトの SQL 型名に対応付けます。このマップは 1 対 1 のマッピングで、ハッシュ表にキーワード値ペアとして格納されます。Oracle オブジェクトからデータを読み込むと、JDBC ドライバでは、型マップが参照され、Oracle オブジェクト型（SQL オブジェクト型）のデータのインスタンス化に使用される Java クラスが決定されます。Oracle オブジェクトにデータを書き込むと、JDBC ドライバによって、SQLData インタフェースの getSQLTypeName() メソッドがコールされ、Java クラスの SQL 型名が取り出されます。SQL と Java 間の実際の変換は、ドライバにより実行されます。

Oracle オブジェクトに対応している Java クラスの属性では、ネイティブな Java 型または Oracle ネイティブ型（oracle.sql.* クラスのインスタンス）を使用して属性を格納できます。

型マップ・オブジェクトの作成と SQLData 実装のマッピング定義

SQLData 実装を使用する場合、型マップを提供するのは、JDBC アプリケーション・プログラマの役割です。型マップは、次のクラスのインスタンスにする必要があります。

- JDK 1.2.x では、標準 java.util.Map インタフェースを実装するクラスのインスタンス

または

- JDK 1.1.x では、標準 java.util.Dictionary クラスを拡張するクラスのインスタンス（または、Dictionary クラス自体のインスタンス）

独自のクラスも作成できますが、JDK 1.2.x または JDK 1.1.x のどちらでも、標準クラス java.util.Hashtable が要件を満たしています。

注意： JDK 1.1.x から JDK 1.2.x に移行する場合、コードで使用されているクラスが、Map インタフェースを実装していることを確認する必要があります。1.1.x で java.util.Hashtable クラスを使用していた場合、変更の必要はありません。

型マップに使用する Hashtable などのクラスは、put() メソッドを実装します。このメソッドは、キーワード値ペアを入力として取ります。各キーは完全修飾 SQL 型名で、対応する値は指定された Java クラスのインスタンスです。

型マップは、接続インスタンスに対応付けられます。標準 java.sql.Connection インタフェースと Oracle 固有 oracle.jdbc.OracleConnection クラスには、getTypeMap() メソッドが含まれています。JDK 1.2.x では、どちらも Map オブジェクトを戻します。JDK 1.1.x では、どちらも Dictionary オブジェクトを戻します。

この項の残りの部分では、次のトピックを取り上げます。

- 既存の型マップへのエントリの追加
- 新しい型マップの作成

既存の型マップへのエントリの追加

最初に接続インスタンスが確立されたときには、デフォルト型マップは空です。SQL-Java マッピング機能を使用するには、デフォルト型マップを移入する必要があります。

既存の型マップにエントリを追加するには、次の手順に従ってください。

1. OracleConnection オブジェクトの getTypeMap() メソッドを使用して、接続の型マップ・オブジェクトを戻します。getTypeMap() メソッドは、java.util.Map オブジェクト（JDK 1.1.x では、java.util.Dictionary）を戻します。たとえば、OracleConnection インスタンス oraconn があるとします。

   ```
   java.util.Map myMap = oraconn.getTypeMap();
   ```

   注意： OracleConnection インスタンスの型マップが初期化されていないと、getTypeMap() を最初にコールしたとき、空のマップが戻されます。

2. 型マップの put() メソッドを使用して、マップ・エントリを追加します。put() メソッドでは 2 つの引数、つまり、SQL 型名文字列およびその SQL 型をマップする Java クラスのインスタンスを指定します。

   ```
   myMap.put(sqlTypeName, classObject);
   ```

   sqlTypeName は、データベースの SQL 型名の完全修飾名を表す文字列です。*classObject* は、その SQL 型をマップする Java クラス・オブジェクトです。次のように Class.forName() メソッドを使用して、クラス・オブジェクトを取り出します。

   ```
   myMap.put(sqlTypeName, Class.forName(className));
   ```

   たとえば、CORPORATE データベース・スキーマに PERSON SQL データ型が定義されている場合は、その SQL データ型を、次の文で Person として定義された PERSON Java クラスにマップできます。

   ```
   myMap.put("CORPORATE.PERSON", Class.forName("Person"));
   ```

マップには、CORPORATE データベースの PERSON SQL データ型が Person Java クラスにマップされているエントリがあります。

注意： 型マップの SQL 型名は、Oracle データベースに大文字で格納されているので、すべて大文字にする必要があります。

新しい型マップの作成

新しい型マップを作成するには、次の手順に従ってください。この例では、java.util.Hashtable のインスタンスを使用します。これは、java.util.Dictionary を拡張し、JDK 1.2.x では java.util.Map も実装します。

1. 新しい型マップ・オブジェクトを作成します。

```
Hashtable newMap = new Hashtable();
```

2. 型マップ・オブジェクトの put() メソッドを使用して、マップにエントリを追加します。put() メソッドの詳細は、8-13 ページの「既存の型マップへのエントリの追加」の手順 2 を参照してください。たとえば、CORPORATE データベースに EMPLOYEE という SQL 型が定義されている場合は、次の文を使用して、その SQL 型を Employee.java に定義されている Employee クラス・オブジェクトにマップできます。

```
newMap.put("CORPORATE.EMPLOYEE", class.forName("Employee"));
```

3. エントリをマップに追加した後に、OracleConnection オブジェクトの setTypeMap() メソッドを使用して、接続の既存の型マップを上書きします。たとえば、次のようになります。

```
oraconn.setTypeMap(newMap);
```

この例では、setTypeMap() によって oraconn 接続の元のマップが newMap に上書きされます。

注意： 接続インスタンスのデフォルト型マップは、マッピングが必要なときに、マップ名を指定しないと使用されます。たとえば、入力としてマップを指定せずに、結果セットの getObject() をコールしたときに使用されます。

型ファイルで指定されていないオブジェクト型のインスタンス化

getObject() コールを使用するときに、適切なエントリを持つ型マップを指定しないと、JDBC ドライバは oracle.sql.STRUCT クラスのインスタンスとして Oracle オブジェクトをインスタンス化します。Oracle オブジェクト型に埋込みオブジェクトが格納されていて、それらのオブジェクトが型マップに存在しない場合、ドライバは埋込みオブジェクトも oracle.sql.STRUCT のインスタンスとしてインスタンス化します。埋込みオブジェクトが型マップに存在する場合、getAttributes() メソッドをコールすると、埋込みオブジェクトは型マップで指定された Java クラスのインスタンスとして戻されます。

SQLData インタフェース

Java アプリケーションで使用可能な Oracle オブジェクトとその属性データを作成する方法の 1 つに、SQLData インタフェースを実装するカスタム・オブジェクト・クラスを作成する方法があります。このインタフェースを使用する場合、データベースの Oracle オブジェクト型、およびそのオブジェクトに対して作成するカスタム・オブジェクト・クラスに対応する名前を指定する型マップを提供する必要があります。

SQLData インタフェースでは、Oracle データベース・オブジェクトに対して行う SQL と Java 間の変換の方法を定義します。標準 JDBC の java.sql パッケージ（JDK 1.1.x では oracle.jdbc2）には、SQLData インタフェースと、そのコンパニオン・インタフェースである SQLInput および SQLOutput インタフェースが含まれています。

SQLData を実装するカスタム・オブジェクト・クラスを作成する場合、SQLData インタフェースで指定されているように、readSQL() メソッドおよび writeSQL() メソッドを用意する必要があります。

JDBC ドライバは readSQL() メソッドをコールし、データベースからデータ値のストリームを読み込み、カスタム・オブジェクト・クラスのインスタンスに移入します。通常、ドライバでは、OracleResultSet オブジェクトの getObject() コールの一部としてこのメソッドが使用されます。

同様に、JDBC ドライバは writeSQL() メソッドをコールし、カスタム・オブジェクト・クラスのインスタンスからデータ値のシーケンスを読み込み、データベースに書き込めるストリームに書き込みます。通常、ドライバでは、OraclePreparedStatement オブジェクトの setObject() コールの一部としてこのメソッドが使用されます。

SQLInput インタフェースおよび SQLOutput インタフェース

JDBC ドライバには、SQLInput および SQLOutput インタフェースを実装するクラスが含まれます。SQLOutput または SQLInput オブジェクトを実装する必要はありません。JDBC ドライバで実装されます。

SQLInput の実装は入力ストリーム・クラスで、このインスタンスを readSQL() メソッドに渡す必要があります。SQLInput には、readObject()、readInt()、readLong()、readFloat()、readBlob() など、Oracle オブジェクトの属性と Java 型の変換に使用されるすべての readXXX() メソッドが含まれます。各 readXXX() メソッドにより、SQL データが Java データに変換され、対応する Java 型の出力パラメータに戻されます。たとえば、readInt() では、整数が戻されます。

SQLOutput の実装は出力ストリーム・クラスで、このインスタンスは writeSQL() メソッドに渡す必要があります。SQLOutput には、各 Java 型の writeXXX() メソッドが含まれます。各 writeXXX() メソッドでは、関連する Java 型のパラメータを入力として指定し、Java データを SQL データに変換します。たとえば、writeString() では、Java クラスの文字列属性を入力として指定します。

readSQL() および writeSQL() メソッドの実装

SQLData を実装するカスタム・オブジェクト・クラスを作成するときは、readSQL() および writeSQL() メソッドを実装する必要があります。

readSQL() は、次のように実装する必要があります。

```
public void readSQL(SQLInput stream, String sql_type_name) throws SQLException
```

- readSQL() メソッドは、SQLInput ストリームと、データの SQL 型名を示す文字列（EMPLOYEE などの Oracle オブジェクト型名）を入力として取ります。

  Java アプリケーションから getObject() をコールすると、JDBC ドライバでは SQLInput ストリーム・オブジェクトを作成し、データベースのデータを使用してそのオブジェクトを移入します。また、ドライバでは、データベースからデータを読み込むときに、データの SQL 型名を決定します。ドライバでは、readSQL() をコールするときに、これらのパラメータを渡します。

- readSQL() では、Oracle オブジェクトの属性にマップする Java データ型ごとに渡された SQLInput ストリームの適切な readXXX() メソッドをコールする必要があります。

  たとえば、CHAR 変数に従業員名、NUMBER 変数に従業員番号が定義されている EMPLOYEE オブジェクトを読み込む場合は、readSQL() メソッドで readString() コールおよび readInt() コールを行う必要があります。JDBC では、Oracle オブジェクト型の SQL 定義に属性が表示される順序に従ってメソッドをコールします。

- readSQL() メソッドは、readXXX() メソッドによって読み込まれて変換されたデータを受け取り、カスタム・オブジェクト・クラス・インスタンスの適切なフィールドまたは要素に割り当てます。

writeSQL() は、次のように実装する必要があります。

```
public void writeSQL(SQLOutput stream) throws SQLException
```

- writeSQL() メソッドは、SQLOutput ストリームを入力として受け取ります。

  Java アプリケーションから setObject() をコールすると、JDBC ドライバは SQLOutput ストリーム・オブジェクトを作成し、そのオブジェクトにカスタム・オブジェクト・クラス・インスタンスを移入します。ドライバでは、writeSQL() をコールするときに、このストリーム・パラメータを渡します。

- writeSQL() では、Oracle オブジェクトの属性にマップする Java データ型ごとに、渡された SQLOutput ストリームの適切な writeXXX() メソッドをコールする必要があります。

  たとえば、CHAR 変数に従業員名、NUMBER 変数に従業員番号が定義されている EMPLOYEE オブジェクトに書き込む場合は、writeSQL() メソッドで writeString() コールおよび writeInt() コールを行う必要があります。これらのメソッドは、Oracle オブジェクト型の SQL 定義に属性が表示される順序に従ってコールする必要があります。

- writeSQL() メソッドは、writeXXX() メソッドによって変換されたデータを SQLOutput ストリームに書き込みます。このデータは、プリコンパイルされた SQL 文を実行したときに、データベースに書き込まれます。

指定された SQL オブジェクト定義への SQLData インタフェースの実装例については、20-41 ページの「SQLData 実装 : SQLDataExample.java」を参照してください。

SQLData 実装によるデータの読込みおよび書込み

この項では、Oracle オブジェクトに対応する Java クラスが SQLData を実装している場合に、その Oracle オブジェクトに対してデータの読込みまたは書込みを行う方法について説明します。

結果セットからの SQLData オブジェクトの読込み

この項では、カスタム・オブジェクト・クラスに対して SQLData 実装を選択したときに、Oracle オブジェクトのデータを Java アプリケーションに読み込む手順について説明します。

この手順では、すでに Oracle オブジェクト型を定義し、対応するカスタム・オブジェクト・クラスを作成し、Oracle オブジェクトと Java クラス間のマッピングを定義する型マップを更新し、文オブジェクト stmt を定義しているものとします。

1. データベースに問合せを行い、Oracle オブジェクトを JDBC 結果セットに読み込みます。

```
ResultSet rs = stmt.executeQuery("SELECT emp_col FROM personnel");
```

PERSONNEL 表には、SQL 型 EMP_OBJECT の列 EMP_COL が含まれています。この SQL 型は、Java クラス Employee にマップされるように、型マップに定義されています。

2. 結果セットの getObject() メソッドを使用して、カスタム・オブジェクト・クラスのインスタンスに結果セットのデータを 1 行移入します。型マップには Employee のエントリが含まれるため、getObject() メソッドによりユーザー定義の SQLData オブジェクトが戻されます。

```
if (rs.next())
   Employee emp = (Employee)rs.getObject(1);
```

型マップにオブジェクトのエントリが存在しない場合は、getObject() により oracle.sql.STRUCT オブジェクトが戻されます。getObject() メソッド・シグネチャによって、汎用の java.lang.Object 型が戻されるため、出力を STRUCT にキャストする必要があります。

```
if (rs.next())
   STRUCT empstruct = (STRUCT)rs.getObject(1);
```

すでに説明したように、getObject() をコールすることにより、SQLData インタフェースの readSQL() および readXXX() がコールされます。

注意： 型マップを使用しないようにするには、getSTRUCT() メソッドを使用します。このメソッドでは、型マップにマッピング・エントリが定義されている場合でも、常に STRUCT オブジェクトが戻されます。

---

3. カスタム・オブジェクト・クラスに get メソッドが定義されている場合、このメソッドを使用して、オブジェクト属性のデータを読み込みます。たとえば、EMPLOYEE に、CHAR 型の EmpName（従業員名）および NUMBER 型の EmpNum（従業員番号）が定義されている場合、Java String を戻す getEmpName() メソッドおよび整数値（int）を戻す getEmpNum() メソッドを定義します。次に、Java アプリケーションでこれらのメソッドを次の方法で起動します。

```
String empname = emp.getEmpName();
int empnumber = emp.getEmpNum();
```

注意： または、getObject() メソッドが定義されているコール可能文オブジェクトを使用して、データをフェッチします。

コール可能文 OUT パラメータからの SQLData オブジェクトの取出し

PL/SQL ファンクション GETEMPLOYEE() をコールする OracleCallableStatement ocs があるとします。プログラムでは、従業員番号（empnumber）をファンクションに渡し、そのファンクションから対応する Employee オブジェクトが戻されます。

1. GETEMPLOYEE() ファンクションをコールする OracleCallableStatement を準備します。

```
OracleCallableStatement ocs =
  (OracleCallableStatement)conn.prepareCall("{ ? = call GETEMPLOYEE(?) }");
```

2. empnumber を、GETEMPLOYEE() の入力パラメータとして宣言します。SQLData オブジェクトを、タイプコード OracleTypes.STRUCT を指定して OUT パラメータとして登録します。次に、文を実行します。

```
ocs.setInt(2, empnumber);
ocs.registerOutParameter(1, OracleTypes.STRUCT, "EMP_OBJECT");
ocs.execute();
```

3. getObject() メソッドを使用して、employee オブジェクトを取り出します。次のコードでは、Oracle オブジェクトを Java 型 Employee にマップする型マップ・エントリがあるものとします。

```
Employee emp = (Employee)ocs.getObject(1);
```

   型マップ・エントリが存在しない場合、getObject() は oracle.sql.STRUCT オブジェクトを戻します。getObject() メソッド・シグネチャによって、汎用の java.lang.Object 型が戻されるため、出力を STRUCT にキャストする必要があります。

```
STRUCT emp = (STRUCT)ocs.getObject(1);
```

SQLData オブジェクトの IN パラメータとしてのコール可能文への引渡し

Employee オブジェクトを IN パラメータとして指定され、そのオブジェクトが PERSONNEL 表に追加されている PL/SQL ファンクション addEmployee(?) を仮定します。この例の emp は、有効な Employee オブジェクトです。

1. addEmployee(?) ファンクションをコールするために OracleCallableStatement を準備します。

```
OracleCallableStatement ocs =
  (OracleCallableStatement) conn.prepareCall("{ call addEmployee(?) }");
```

2. setObject() を使用して、emp オブジェクトを IN パラメータとしてコール可能文に渡します。次に、文を実行します。

```
ocs.setObject(1, emp);
ocs.execute();
```

SQLData 実装を使用したデータの Oracle オブジェクトへの書込み

この項では、カスタム・オブジェクト・クラスに対して SQLData 実装を選択したときに、Java アプリケーションのデータを Oracle オブジェクトに書き込む手順について説明します。

ここでは、あらかじめ Oracle オブジェクト型を定義し、対応するカスタム Java クラスを作成し、Oracle オブジェクトと Java クラス間のマップを定義する型マップを更新していることを前提としています。

1. カスタム・オブジェクト・クラスに set メソッドが定義されている場合、このメソッドを使用して、アプリケーションの Java 変数のデータを Java データ型オブジェクトの属性に書き込みます。

```
emp.setEmpName(empname);
emp.setEmpNum(empnumber);
```

この文では、8-17 ページの「結果セットからの SQLData オブジェクトの読込み」で割り当てた emp オブジェクト、empname 変数および empnumber 変数を使用します。

2. Java データ型オブジェクトのデータを使用して、データベース表の行に格納されている Oracle オブジェクトを更新する文を、適切に準備します。

```
PreparedStatement pstmt = conn.prepareStatement
                          ("INSERT INTO PERSONNEL VALUES (?)");
```

conn は接続オブジェクトです。

3. プリコンパイルされた SQL 文の setObject() メソッドを使用して、Java データ型オブジェクトをプリコンパイルされた SQL 文にバインドします。

```
pstmt.setObject(1, emp);
```

4. 文を実行すると、データベースが更新されます。

```
pstmt.executeUpdate();
```

ORAData インタフェース

Java アプリケーションで使用可能な Oracle オブジェクトとその属性データを作成する方法の 1 つに、oracle.sql.ORAData および oracle.sql.ORADataFactory インタフェースを実装するカスタム・オブジェクト・クラスを作成する方法があります（または、個々のクラスで ORADataFactory を実装できます）。ORAData および ORADataFactory インタフェースは Oracle が提供するもので、JDBC 標準の一部ではありません。

注意： JPublisher ユーティリティは、ORAData および ORADataFactory インタフェースを実装するクラスの生成をサポートしています。詳細は、8-44 ページの「JPublisher を使用したカスタム・オブジェクト・クラスの作成」を参照してください。

ORAData の機能

ORAData インタフェースには、次の利点があります。

- JDBC の Oracle 拡張機能を識別します。ORAData では、oracle.sql.Datum 型が直接使用されます。
- 作成中の Java カスタム・クラスの名前を指定するときに、型マップは必要ありません。
- パフォーマンスが向上します。ORAData は、Datum 型を直接使用して動作します。Datum 型は、Oracle オブジェクトを保持するために、ドライバにより使用される内部形式です。

ORAData および ORADataFactory インタフェースでは、次の処理が実行されます。

- ORAData クラスの toDatum() メソッドでは、データを oracle.sql.* 表現に変換します。
- ORADataFactory では、カスタム・オブジェクト・クラスのコンストラクタと等価の create() メソッドが指定されます。このメソッドでは、ORAData インスタンスを作成および戻します。JDBC ドライバは、create() メソッドを使用して、カスタム・オブジェクト・クラスのインスタンスを Java アプリケーションまたはアプレットに戻します。このメソッドは、入力として、oracle.sql.Datum オブジェクトおよび OracleTypes クラスで指定された対応する SQL タイプコードを示す整数を取ります。

ORAData および ORADataFactory は、次のように定義されます。

```
public interface ORAData
{
    Datum toDatum (java.sql.Connection conn) throws SQLException;
}
```

```
public interface ORADataFactory
{
    ORAData create (Datum d, int sql_Type_Code) throws SQLException;
}
```

conn は接続オブジェクト、*d* は oracle.sql.Datum 型のオブジェクト、*sql_Type_Code* は Datum オブジェクトの SQL タイプコード（標準 Types または OracleTypes クラス）を表します。

オブジェクト・データの取出しと挿入

オブジェクト・データを ORAData のインスタンスとして取得および挿入を行う場合、JDBC ドライバでは次のメソッドを使用します。

オブジェクト・データを取り出すには、次のメソッドを使用します。

- Oracle 固有 OracleResultSet クラスの getORAData() メソッドを使用します（OracleResultSet オブジェクトを ors とします）。

  ```
  ors.getORAData (int col_index, ORADataFactory factory);
  ```

  このメソッドでは、結果セットのデータの列索引および ORADataFactory のインスタンスを入力として指定します。たとえば、カスタム・オブジェクト・クラスに getORAFactory() メソッドを実装して、getORAData() に入力する ORADataFactory インスタンスを作成できます。ORAData を実装する Java クラスを使用するときは、型マップは必要ありません。

または

- ResultSet インタフェースで指定される標準 getObject(*index*、*map*) メソッドを使用して、ORAData のインスタンスとしてデータを取り出します。この場合、指定されたオブジェクト型で使用されるファクトリ・クラスおよび対応する SQL 型名を識別するには、型マップにエントリを定義する必要があります。

オブジェクト・データを挿入するには、次のメソッドを使用します。

- Oracle 固有 OraclePreparedStatement クラスの setORAData() メソッドを使用します（OraclePreparedStatement オブジェクトを ops とします）。

  ```
  ops.setORAData (int bind_index, ORAData custom_obj);
  ```

  このメソッドでは、バインド変数のパラメータ索引、および変数を含むオブジェクト名を入力として指定します。

または

- PreparedStatement インタフェースで指定される標準 setObject() メソッドを使用します。このメソッドを別のフォームで使用して、型マップなしで ORAData インスタンスを挿入することもできます。

次の項以降では、getORAData() および setORAData() メソッドについて説明します。

Oracle オブジェクト EMPLOYEE の例を引き続き使用するには、Java アプリケーションに次のコードを記述する必要があります。

```
ORAData datum = ors.getORAData(1, Employee.getORAFactory());
```

この例では、ors は Oracle 結果セット、getORAData() は ORAData オブジェクトの取出しに使用される OracleResultSet クラス、および EMPLOYEE は結果セットの列 1 です。静的 Employee.getORAFactory() メソッドによって、ORADataFactory が JDBC ドライバに戻されます。JDBC ドライバでは、このオブジェクトから create() をコールし、結果セットのデータが移入された Employee クラスのインスタンスを Java アプリケーションに戻します。

注意：

- ORAData および ORADataFactory は、独立したインタフェースとして定義されるため、必要に応じて異なる Java クラス（Employee クラス、EmployeeFactory クラスなど）から実装できます。
- ORAData インタフェースを使用するには、カスタム・オブジェクト・クラスで oracle.sql.*（少なくとも、ORAData、ORADataFactory および Datum）をインポートする必要があります。

指定された SQL オブジェクト定義への ORAData インタフェースを実装する例については、20-45 ページの「ORAData 実装 : ORADataExample.java」を参照してください。

ORAData 実装によるデータの読込みおよび書込み

この項では、対応する Java クラスが ORAData を実装する場合に、Oracle オブジェクトに対してデータの読込みまたは書込みを行う方法について説明します。

ORAData 実装を使用した Oracle オブジェクトからのデータの読込み

この項では、データを Oracle オブジェクトから Java アプリケーションに読み込む手順について説明します。この手順は、ORADatad を手動で実装する場合と、JPublisher を使用してカスタム・オブジェクト・クラスを作成する場合の両方に適用されます。

この手順では、すでに Oracle オブジェクト型を定義し、対応するカスタム・オブジェクト・クラスを手動、または JPublisher を使用して作成し、文オブジェクト stmt を定義しているものとします。

1. データベースの問合せを行って Oracle オブジェクトを結果セットに読み込み、Oracle 結果セットにキャストします。

```
OracleResultSet ors = (OracleResultSet)stmt.executeQuery
                      ("SELECT Emp_col FROM PERSONNEL");
```

PERSONNEL は、1 列で構成される表です。列名は、Employee_object 型の Emp_col です。

2. Oracle 結果セットの getORAData() メソッドを使用して、カスタム・オブジェクト・クラスのインスタンスに結果セットのデータを 1 行移入します。getORAData() メソッドにより oracle.sql.ORAData オブジェクトが戻されます。このオブジェクトは特定のカスタム・オブジェクト・クラスにキャストできます。

```
if (ors.next())
   Employee emp = (Employee)ors.getORAData(1, Employee.getORAFactory());
```

または

```
if (ors.next())
   ORAData datum = ors.getORAData(1, Employee.getORAFactory());
```

この例では、Employee はカスタム・オブジェクト・クラス名、ors は OracleResultSet オブジェクト名です。

getORAData() を使用しない場合、標準 JDBC ResultSet の getObject() メソッドを使用して ORAData データを取り出せます。ただし、型マップにはエントリが必要です。型マップでは、エントリは指定されたオブジェクト型で使用されるファクトリ・クラス、および対応する SQL 型名を識別します。

たとえば、オブジェクトの SQL 型名が EMPLOYEE の場合は、対応する Java クラスは Employee で、ORAData が実装されます。対応するファクトリ・クラスは EmployeeFactory で、ORADataFactory が実装されます。

次の文を使用して、型マップに EmployeeFactory エントリを宣言します。

```
map.put ("EMPLOYEE", Class.forName ("EmployeeFactory"));
```

次に、getObject() の形式を使用し、マップ・オブジェクトを指定します。

```
Employee emp = (Employee) rs.getObject (1, map);
```

接続のデフォルトの型マップに、指定されたオブジェクト型と対応する SQL 型名に対して使用されるファクトリ・クラスを識別するエントリがすでに存在する場合は、次の形式の getObject() を使用できます。

```
Employee emp = (Employee) rs.getObject (1);
```

3. カスタム・オブジェクト・クラスに get メソッドが定義されている場合、このメソッドを使用して、オブジェクト属性のデータをアプリケーションの Java 変数に読み込みます。たとえば、EMPLOYEE に、CHAR 型の EmpName および NUMBER 型の EmpNum（従業員番号）が定義されている場合、Java 文字列を戻す getEmpName() メソッドおよび整数値を戻す getEmpNum() メソッドを定義します。次に、Java アプリケーションでこれらのメソッドを次の方法で起動します。

```
String empname = emp.getEmpName();
int empnumber = emp.getEmpNum();
```

注意： または、コール可能文オブジェクトにデータをフェッチできます。OracleCallableStatement クラスには、getORAData() メソッドも定義されています。

ORAData 実装を使用したデータの Oracle オブジェクトへの書込み

この項では、データを Java アプリケーションから Oracle オブジェクトに書き込む手順について説明します。この手順は、ORAData を手動で実装する場合と、JPublisher を使用してカスタム・オブジェクト・クラスを作成する場合の両方に適用されます。

この手順では、すでに Oracle オブジェクト型を定義し、対応するカスタム・オブジェクト・クラスを手動で（または JPublisher を使用して）作成しているものとします。

注意： データベースの INSERT および UPDATE 操作の実行時には、型マップは使用されません。

---

1. カスタム・オブジェクト・クラスに set メソッドが定義されている場合、このメソッドを使用して、アプリケーションの Java 変数のデータを Java データ型オブジェクトの属性に書き込みます。

```
emp.setEmpName(empname);
emp.setEmpNum(empnumber);
```

この文で使用されている emp オブジェクト、empname 変数および empnumber 変数の詳細は、8-23 ページの「ORAData 実装を使用した Oracle オブジェクトからのデータの読込み」を参照してください。

2. Java データ型オブジェクトのデータを使用して、Oracle オブジェクトを更新するプリコンパイルされた SQL 文を、適切にデータベース表の行に書き込みます。

```
OraclePreparedStatement opstmt = conn.prepareStatement
   ("UPDATE PERSONNEL SET Employee = ? WHERE Employee.EmpNum = 28959);
```

この例では、conn は Connection オブジェクトです。

3. プリコンパイルされた Oracle 文の setORAData() メソッドを使用して、Java データ型オブジェクトをプリコンパイルされた SQL 文にバインドします。

```
opstmt.setORAData(1, emp);
```

setORAData() メソッドでは、カスタム・オブジェクト・クラス・インスタンスの toDatum() メソッドをコールすることにより、データベースに書き込むことができる oracle.sql.STRUCT オブジェクトが取り出されます。

この手順では、setObject() メソッドを使用して Java データ型をバインドすることもできます。たとえば、次のようになります。

```
opstmt.setObject(1,emp);
```

注意： Java データ型オブジェクトを、IN または OUT バインド変数として使用できます。

その他の ORAData の使用方法

ORAData インタフェースには、SQLData インタフェースより優れた柔軟性があります。SQLData インタフェースは、Oracle オブジェクト型（SQL オブジェクト型）から目的の Java 型へのマッピングをカスタマイズする目的で設計されています。SQLData インタフェースを実装すると、Java と SQL 型間の変換が正常に終了してから、元の SQL オブジェクト・データとカスタム Java クラス・インスタンスのフィールドとの間の移入が、JDBC ドライバによって実行されます。

ORAData インタフェースの場合は、Oracle オブジェクト型から Java 型へのカスタマイズがサポートされるのみでなく、このインタフェースによって、Java オブジェクト型と oracle.sql パッケージがサポートするすべての SQL 型との間のマッピングが可能になります。

これは、カスタム Java クラスを oracle.sql.* 型にラップするときや、カスタマイズされた変換または機能を実装するときに役立ちます。次の使用例が考えられます。

- データの暗号化および復号化、または妥当性チェックを実行します。
- 読込みまたは書込みを行った値のロギングを実行します。
- キャラクタ列（URL 情報を含む文字フィールド）を小さなコンポーネントに解析します。
- 文字列を数値定数にマップします。
- データを、より適した Java 形式にマップ（DATE フィールドを java.util.Date 形式にマップするなど）します。

- データ表現をカスタマイズ（表の列のデータがフィート単位のとき、選択後にメートルで表現するなど）します。
- Java オブジェクトを、RAW フィールドなどとの間で、シリアル化およびデシリアライズします。

たとえば、ORAData を使用すると、データベースの特定の SQL Oracle オブジェクト型に対応していない Java オブジェクトのインスタンスを、SQL 型 RAW の列に格納できます。ORADataFactory の create() メソッドでは、oracle.sql.RAW 型のオブジェクトから目的の Java オブジェクトへの変換を実装する必要があります。ORAData の toDatum() メソッドでは、Java オブジェクトから oracle.sql.RAW 型のオブジェクトへの変換を実装する必要があります。これには、Java のシリアル化などを使用します。

JDBC ドライバでは、データのロー・バイトを oracle.sql.RAW 形式で透過的に取り出します。次に、ORADataFactory の create() メソッドをコールし、oracle.sql.RAW オブジェクトを目的の Java クラスに変換します。

データベースに Java オブジェクトを挿入するときは、そのオブジェクトを RAW 型の列にバインドするのみでデータベースに格納できます。ドライバは、ORAData の toDatum() メソッドを透過的にコールして、Java オブジェクトを oracle.sql.RAW 型のオブジェクトに変換します。このオブジェクトは、データベースに RAW 型の 1 列として格納されます。

ORAData インタフェースのサポートにより、これらの変換が oracle.sql.* 形式を使用して動作するように設計されています。この形式は JDBC ドライバで使用されている内部形式であるため、効率が大幅に向上します。また、ORAData を実装する Java クラスを使用するときに、SQLData インタフェースで必要な型マップは必要ありません。ORAData を実装するクラスに型マップが不要な理由については、8-21 ページの「ORAData インタフェース」を参照してください。

CustomDatum インタフェースの廃止

Oracle9*i* では、oracle.jdbc.driver クラスにかわって oracle.jdbc インタフェースが導入されたため、以前のリリースでカスタマイズされたオブジェクトへのアクセスに使用されていた oracle.sql.CustomDatum および oracle.sql.CustomDatumFactory インタフェースにかわって、oracle.sql.ORAData および oracle.sql.ORADataFactory という新しいインタフェースが使用されます。

CustomDatum および CustomDatumFactory インタフェースの仕様は、次のとおりです。

```
public interface CustomDatum
{
  oracle.sql.Datum toDatum(
    oracle.jdbc.driver.OracleConnection c
  ) throws SQLException ;

// The following is expected to be present in an
// implementation:
//
// - Definition of public static fields for
//   _SQL_TYPECODE, _SQL_NAME and  _SQL_BASETYPE.
//   (See Oracle Jdbc documentation for details.)
//
// - Definition of
//   public static CustomDatumFactory
//     getFactory();
//
}

public interface CustomDatumFactory
{
  oracle.sql.CustomDatum create(
    oracle.sql.Datum d, int sqlType
    ) throws SQLException;
}
```

オブジェクト型の継承

オブジェクト型の継承は、別のオブジェクト型を拡張して新しいオブジェクト型を作成することができる Oracle9*i* の機能です。（Oracle9*i* では、まだ JDBC 3.0 をサポートしていませんが、オブジェクト型の継承はサポートされます。）新しいオブジェクト型は、拡張元のオブジェクト型のサブタイプになります。サブタイプは、そのスーパータイプに定義されたすべての属性およびメソッドを自動的に継承します。サブタイプは、属性およびメソッドを追加したり、スーパータイプから継承されたメソッドをオーバーロードまたはオーバーライドできます。

オブジェクト型の継承により、代入可能性が導入されます。代入可能性とは、T 型の値を保持するように宣言したスロットで、T 型の任意のサブタイプの値を保持できます。Oracle9*i* JDBC ドライバは、透過的に代入可能性を処理します。

データベース・オブジェクトは、情報が失われることなく、最も固有の型で戻されます。たとえば、PERSON_T スロットに STUDENT_T オブジェクトが格納されている場合、Oracle JDBC ドライバは STUDENT_T オブジェクトを表す Java オブジェクトを戻します。

サブタイプの作成

明示的に Oracle オブジェクト型に対応する Java クラスを作成するには、カスタム・オブジェクト・クラスを作成します。（8-10 ページの「Oracle オブジェクト用のカスタム・オブジェクト・クラスの作成と使用方法」を参照してください。）オブジェクト型の階層がある場合には、Java クラスの対応する階層を作成できます。

JDBC でデータベース・サブタイプを作成する最も一般的な方法は、`java.sql.Statement` インタフェースの `execute()` メソッドに SQL の `CREATE TYPE` コマンドを渡すことです。たとえば、次のように型継承階層を作成する場合があります。

```
PERSON_T
|
STUDENT_T
|
PARTTIMESTUDENT_T
```

この場合、JDBC コードは次のようになります。

```
Statement s = conn.createStatement();
s.execute ("CREATE TYPE Person_T (SSN NUMBER, name VARCHAR2(30),
  address VARCHAR2(255))");
s.execute ("CREATE TYPE Student_T UNDER Person_t (deptid NUMBER,
  major VARCHAR2(100))");
s.execute ("CREATE TYPE PartTimeStudent_t UNDER Student_t (numHours NUMBER)");
```

次のコードでは、`ST` 型で `"foo"` メンバー・プロシージャがオーバーロードされ、`"print"` メンバー・プロシージャによって `T` 型から継承されたコピーが上書きされます。

```
CREATE TYPE T AS OBJECT (...,
  MEMBER PROCEDURE foo(x NUMBER),
  MEMBER PROCEDURE Print(),
  ...
  NOT FINAL;

CREATE TYPE ST UNDER T (...,
  MEMBER PROCEDURE foo(x DATE),         <-- overload "foo"
  OVERRIDING MEMBER PROCEDURE Print(),   <-- override "print"
  STATIC FUNCTION bar(...) ...
  ...
  );
```

サブタイプを作成すると、実表の列またはオブジェクト型の属性として使用できます。サブタイプを作成するための構文の詳細は、『Oracle9*i* アプリケーション開発者ガイド－オブジェクト・リレーショナル機能』を参照してください。

サブタイプに対してカスタマイズされたクラスの実装

一般に、カスタマイズされた Java クラスはデータベース・オブジェクト・タイプを表します。サブタイプに対してカスタマイズされた Java クラスを作成するときには、Java クラスでデータベース・オブジェクト・タイプの階層をミラー化するかどうかを選択できます。

クラスの作成には、ORAData または SQLData ソリューションのいずれかを使用して、オブジェクト型の階層をマップできます。

ORAData を使用した型継承階層の作成

oracle.sql.ORAData インタフェースを実装する Java クラスを用いた、カスタマイズされたマッピングを使用することをお薦めします。（8-11 ページの「ORAData と SQLData の利点」を参照してください。）ORAData のマッピングでは、JDBC アプリケーションが ORAData および ORADataFactory インタフェースを実装する必要があります。ORADataFactory インタフェースを実装するクラスは、オブジェクトを作成するファクトリを格納します。各オブジェクトはそれぞれ 1 つのデータベース・オブジェクトを表します。

ORAData インタフェースを実装するクラスの階層は、データベース・オブジェクト・タイプ階層をミラー化できます。たとえば、PERSON_T および STUDENT_T に対してマッピングされる Java クラスは、次のようになります。

ORAData を使用した Person_java 次のコードは、ORAData および ORADataFactory インタフェースを実装する Person_java クラスを示します。

```
class Person implements ORAData, ORADataFactory
{
  static final Person _personFactory = new Person();

  public NUMBER ssn;
  public CHAR name;
  public CHAR address;

  public static ORADataFactory getORADataFactory()
  {
    return _personFactory;
  }

  public Person () {}

  public Person(NUMBER ssn, CHAR name, CHAR address)
  {
    this.ssn = ssn;
    this.name = name;
    this.address = address;
  }
```

```
  public Datum toDatum(OracleConnection c) throws SQLException
  {
    StructDescriptor sd =
      StructDescriptor.createDescriptor("SCOTT.PERSON_T", c);
    Object [] attributes = { ssn, name, address };
    return new STRUCT(sd, c, attributes);
  }

  public ORAData create(Datum d, int sqlType) throws SQLException
  {
    if (d == null) return null;
    Object [] attributes = ((STRUCT) d).getOracleAttributes();
    return new Person((NUMBER) attributes[0],
                      (CHAR) attributes[1],
                      (CHAR) attributes[2]);
  }
}
```

Person.java を拡張する Student.java 次のコードは、Person.java クラスを拡張する Student.java クラスを示します。

```
class Student extends Person
{
  static final Student _studentFactory = new Student ();

  public NUMBER deptid;
  public CHAR major;

  public static ORADataFactory getORADataFactory()
  {
    return _studentFactory;
  }

  public Student () {}

  public Student (NUMBER ssn, CHAR name, CHAR address,
                  NUMBER deptid, CHAR major)
  {
    super (ssn, name, address);
    this.deptid = deptid;
    this.major = major;
  }
```

```
  public Datum toDatum(OracleConnection c) throws SQLException
  {
    StructDescriptor sd =
      StructDescriptor.createDescriptor("SCOTT.STUDENT_T", c);
    Object [] attributes = { ssn, name, address, deptid, major };
    return new STRUCT(sd, c, attributes);
  }

  public CustomDatum create(Datum d, int sqlType) throws SQLException
  {
    if (d == null) return null;
    Object [] attributes = ((STRUCT) d).getOracleAttributes();
    return new Student((NUMBER) attributes[0],
                       (CHAR) attributes[1],
                       (CHAR) attributes[2],
                       (NUMBER) attributes[3],
                       (CHAR) attributes[4]);
  }
}
```

ORAData インタフェースを実装するカスタマイズされたクラスでは、必ずしもオブジェクト型階層をミラー化する必要はありません。たとえば、スーパークラスを指定せずに前述のクラス、Student を宣言できます。この場合、Student クラスには、PERSON_T から継承された属性および STUDENT_T によって宣言された属性を保持するフィールドが含まれます。

ORADataFactory の実装 次の例に示すように、JDBC アプリケーションでは、データベースの問合せにファクトリ・クラスを使用して、Person またはそのサブクラスのインスタンスを戻します。

```
ResultSet rset = stmt.executeQuery ("select person from tab1");
while (rset.next())
{
  Object s = rset.getORAData (1, PersonFactory.getORADataFactory());
  ...
}
```

ORADataFactory インタフェースを実装したクラスでは、対応付けられたカスタム・オブジェクト型のインスタンスおよび任意のサブタイプのインスタンス、または少なくともサポートされるすべての型のインスタンスの作成が可能になります。

次の例では、PersonFactory.getORADataFactory() メソッドが（person、student または parttimestudent Java インスタンスを戻すことで）、PERSON_T、STUDENT_T および PARTTIMESTUDENT_T オブジェクトを処理できるファクトリを戻します。

```
class PersonFactory implements ORADataFactory
{
  static final PersonFactory _factory = new PersonFactory ();

  public static ORADataFactory getORADataFactory()
  {
    return _factory;
  }

  public ORAData create(Datum d, int sqlType) throws SQLException
  {
    STRUCT s = (STRUCT) d;
    if (s.getSQLTypeName ().equals ("SCOTT.PERSON_T"))
      return Person.getORADataFactory ().create (d, sqlType);
    else if (s.getSQLTypeName ().equals ("SCOTT.STUDENT_T"))
      return Student.getORADataFactory ().create(d, sqlType);
    else if (s.getSQLTypeName ().equals ("SCOTT.PARTTIMESTUDENT_T"))
      return ParttimeStudent.getORADataFactory ().create(d, sqlType);
    else
      return null;
  }
}
```

次の例では、次のような表 tabl1 があることを想定しています。

```
CREATE TABLE tabl1 (idx NUMBER, person PERSON_T);
INSERT INTO tabl1 VALUES (1, PERSON_T (1000, 'Scott', '100 Oracle Parkway'));
INSERT INTO tabl1 VALUES (2, STUDENT_T (1001, 'Peter', '200 Oracle Parkway', 101,
'CS'));
INSERT INTO tabl1 VALUES (3, PARTTIMESTUDENT_T (1002, 'David', '300 Oracle Parkway',
102, 'EE'));
```

SQLData を使用した型継承階層の作成

java.sql.SQLData インタフェースを実装するカスタマイズされたクラスを使用すると、データベース・オブジェクト・タイプ階層をミラー化できます。サブクラスの readSQL() および writeSQL() メソッドは、スーパークラス内の対応するメソッドの各コールをカスケードして、サブクラス属性の読取りまたは書込みを行う前にスーパークラスの読取りまたは書込みを実行します。たとえば、PERSON_T および STUDENT_T に対してマッピングされる Java クラスは、次のようになります。

SQLData を使用した Person.java 次のコードは、SQLData インタフェースを実装する Person.java クラスを示します。

```
import java.sql.*;

public class Person implements SQLData
{
  private String sql_type;
  public int ssn;
  public String name;
  public String address;

  public Person () {}

  public String getSQLTypeName() throws SQLException { return sql_type; }

  public void readSQL(SQLInput stream, String typeName) throws SQLException
  {
    sql_type = typeName;
    ssn = stream.readInt();
    name = stream.readString();
    address = stream.readString();
  }

  public void writeSQL(SQLOutput stream) throws SQLException
  {
    stream.writeInt (ssn);
    stream.writeString (name);
    stream.writeString (address);
  }
}
```

Student.java を拡張する Student.java 次のコードは、Person.java クラスを拡張する Student.java クラスを示します。

```
import java.sql.*;

public class Student extends Person
{
  private String sql_type;
  public int deptid;
  public String major;

  public Student () { super(); }

  public String getSQLTypeName() throws SQLException { return sql_type; }

  public void readSQL(SQLInput stream, String typeName) throws SQLException
  {
    super.readSQL (stream, typeName);    // read supertype attributes
    sql_type = typeName;
    deptid = stream.readInt();
    major = stream.readString();
  }

  public void writeSQL(SQLOutput stream) throws SQLException
  {
    super.writeSQL (stream);             // write supertype
                                         // attributes
    stream.writeInt (deptid);
    stream.writeString (major);
  }
}
```

SQLData インタフェースを実装するカスタマイズされたクラスは、必ずしもオブジェクト型階層をミラー化する必要はありません。たとえば、スーパークラスを指定せずに前述のクラス、Student を宣言できます。この場合、Student クラスには、PERSON_T から継承された属性および STUDENT_T によって宣言された属性を保持するフィールドが含まれます。

SQLData を使用した Student.java 次のコードは、Person.java クラスを拡張せずに、SQLData インタフェースを直接実装する Student.java クラスを示します。

```
import java.sql.*;

public class Student implements SQLData
{
  private String sql_type;

  public int ssn;
  public String name;
  public String address;
  public int deptid;
  public String major;

  public Student () {}

  public String getSQLTypeName() throws SQLException { return sql_type; }

  public void readSQL(SQLInput stream, String typeName) throws SQLException
  {
    sql_type = typeName;
    ssn = stream.readInt();
    name = stream.readString();
    address = stream.readString();
    deptid = stream.readInt();
    major = stream.readString();
  }

  public void writeSQL(SQLOutput stream) throws SQLException
  {
    stream.writeInt (ssn);
    stream.writeString (name);
    stream.writeString (address);
    stream.writeInt (deptid);
    stream.writeString (major);
  }
}
```

JPublisher ユーティリティ

SQLData、ORAData および ORADataFactory インタフェースを実装するカスタマイズされたクラスは手動で作成することも可能ですが、Oracle9*i* JPublisher ユーティリティを使用してクラスを自動作成することをお薦めします。JPublisher によって作成され、SQLData、ORAData および ORADataFactory インタフェースを実装するカスタマイズされたクラスでは、継承階層をミラー化できます。

JPublisher の詳細は、8-44 ページの「JPublisher を使用したカスタム・オブジェクト・クラスの作成」および『Oracle9*i* JPublisher ユーザーズ・ガイド』を参照してください。

サブタイプ・オブジェクトの取得

一般的な JDBC アプリケーションでは、サブタイプ・オブジェクトは次のいずれかとして戻されます。

- 問合せ結果
- PL/SQL OUT パラメータ
- 型属性

デフォルト（oracle.sql.STRUCT）、ORAData または SQLData マッピングを使用して、サブタイプを取得できます。

デフォルト・マッピングの使用

デフォルトで、データベース・オブジェクトは、oracle.sql.STRUCT クラスのインスタンスとして戻されます。このインスタンスは、宣言された型または宣言された型のサブタイプのオブジェクトを表します。STRUCT クラスがデータベースのサブタイプ・オブジェクトを表す場合、このクラスにはそのスーパータイプの属性と、サブタイプに定義された属性が含まれます。

Oracle JDBC ドライバは、最も固有の型でデータベース・オブジェクトを戻します。JDBC アプリケーションは、STRUCT クラスの getSQLTypeName() メソッドを使用して、STRUCT オブジェクトの SQL 型を判断します。次のコードを参照してください。

```
// tab1.person column can store PERSON_T, STUDENT_T and PARTIMESTUDENT_T objects
ResultSet rset = stmt.executeQuery ("select person from tab1");
while (rset.next())
{
  oracle.sql.STRUCT s = (oracle.sql.STRUCT) rset.getObject(1);
  if (s != null)
    System.out.println (s.getSQLTypeName());    // print out the type name which
                                                // may be SCOTT.PERSON_T,
                                                // SCOTT.STUDENT_T or
                                                // SCOTT.PARTTIMESTUDENT_T
}
```

SQLData マッピングの使用

SQLData マッピングの場合、JDBC ドライバは SQLData インタフェースを実装するクラスのインスタンスとして、データベース・オブジェクトを戻します。

データベース・オブジェクトの取得に SQLData マッピングを使用するには、次の操作を実行します。

1. 目的のオブジェクト型の SQLData インタフェースを実装するラッパー・クラスを実装します。
2. 各 Oracle オブジェクト型（SQL オブジェクト型）に対応するカスタム Java 型を指定するエントリを含む接続型マップを移入します。
3. SQL オブジェクト値へのアクセスには、getObject() メソッドを使用します。

   JDBC ドライバにより、型マップと一致するエントリがチェックされます。一致するエントリが見つかると、ドライバは SQLData インタフェースを実装するクラスのインスタンスとしてデータベース・オブジェクトを戻します。

次のコードは、カスタマイズされた SQLData マッピングの全プロセスを示します。

```
// The JDBC application developer implements Person.java for PERSON_T,
// Student.java for STUDENT_T
// and ParttimeStudent.java for PARTTIMESTUDEN_T.

Connection conn = ...;  // make a JDBC connection

// obtains the connection typemap
java.util.Map map = conn.getTypeMap ();

// populate the type map
map.put ("SCOTT.PERSON_T", Class.forName ("Person"));
map.put ("SCOTT.STUDENT_T", Class.forName ("Student"));
map.put ("SCOTT.PARTTIMESTUDENT_T", Class.forName ("ParttimeStudent"));

// tab1.person column can store PERSON_T, STUDENT_T and PARTTIMESTUDENT_T objects
ResultSet rset = stmt.executeQuery ("select person from tab1");
while (rset.next())
{
  // "s" is instance of Person, Student or ParttimeStudent
  Object s = rset.getObject(1);
```

```
  if (s != null)
  {
    if (s instanceof Person)
      System.out.println ("This is a Person");
    else if (s instanceof Student)
      System.out.println ("This is a Student");
    else if (s instanceof ParttimeStudent)
      System.out.println ("This is a PartimeStudent");
    else
      System.out.println ("Unknown type");
  }
}
```

JDBC ドライバにより、各コールの接続型マップと次のものが照合されます。

- `java.sql.ResultSet` および `java.sql.CallableStatement` インタフェースの `getObject()` メソッド
- `java.sql.Struct` インタフェースの `getAttribute()` メソッド
- `java.sql.Array` インタフェースの `getArray()` メソッド
- `oracle.sql.REF` インタフェースの `getValue()` メソッド

ORAData マッピングの使用

`ORAData` マッピングの場合、JDBC ドライバは `ORAData` インタフェースを実装するクラスのインスタンスとして、データベース・オブジェクトを戻します。

Oracle JDBC ドライバは、Oracle オブジェクト型に対する Java クラスのマッピングを認識する必要があります。Oracle JDBC ドライバにこの情報を通知するには、次のような 2 つの方法があります。

- JDBC アプリケーションは `getORAData(int idx, ORADataFactory f)` メソッドを使用して、データベース・オブジェクトにアクセスします。`getORAData()` メソッドの 2 番目のパラメータは、カスタマイズされたクラスを作成するファクトリ・クラスのインスタンスを指定します。`getORAData()` メソッドは、`OracleResultSet` および `OracleCallableStatement` クラスで使用可能です。
- JDBC アプリケーションは、各 Oracle オブジェクト型に対応するカスタム Java 型を指定するエントリを含む接続型マップを移入します。Oracle オブジェクト値へのアクセスには、`getObject()` メソッドが使用されます。

最初の方法では型マップの検索が回避されるため、より効率的です。ただし、2 番目の方法では、標準 getObject() メソッドが使用されます。次のサンプル・コードは、最初の方法を示します。

```
// tab1.person column can store both PERSON_T and STUDENT_T objects
ResultSet rset = stmt.executeQuery ("select person from tab1");
while (rset.next())
{
  Object s = rset.getORAData (1, PersonFactory.getORADataFactory());
  if (s != null)
  {
    if (s instanceof Person)
      System.out.println ("This is a Person");
    else if (s instanceof Student)
      System.out.println ("This is a Student");
    else if (s instanceof ParttimeStudent)
      System.out.pritnln ("This is a PartimeStudent");
    else
      System.out.println ("Unknown type");
  }
}
```

サブタイプ・オブジェクトの作成

JDBC アプリケーションが、JDBC ドライバを持つデータベース・サブタイプ・オブジェクトを作成する場合があります。これらのオブジェクトは、バインド変数としてデータベースに送られるか、JDBC アプリケーション内での情報交換に使用されます。

カスタマイズされたマッピングの場合、JDBC アプリケーションは（選択された方法に応じて）データベース・サブタイプ・オブジェクトを表す SQLData または ORAData ベースのオブジェクトを作成します。デフォルト・マッピングの場合、データベース・サブタイプ・オブジェクトを表す STRUCT オブジェクトを作成します。スーパータイプから継承されたすべてのデータ・フィールド、およびサブタイプに定義されたすべてのフィールドに値が必要になります。次のコードを参照してください。

```
Connection conn = ...    // make a JDBC connection
StructDescriptor desc = StructDescriptor.createDescriptor ("SCOTT.PARTTIMESTUDENT",
conn);
Object[] attrs = {
  new Integer(1234), "Scott", "500 Oracle Parkway", // data fields defined in
                                                    // PERSON_T
  new Integer(102), "CS",                           // data fields defined in
                                                    // STUDENT_T
  new Integer(4)                                    // data fields defined in
                                                    // PARTTIMESTUDENT_T
```

```
};
STRUCT s = new STRUCT (desc, conn, attrs);
```

s は、PERSON_T および STUDENT_T から継承されたデータ・フィールドと、PARTTIMESTUDENT_T に定義されたデータ・フィールドによって初期化されます。

サブタイプ・オブジェクトの送信

一般的な JDBC アプリケーションでは、データベース・オブジェクトを表す Java オブジェクトは次のいずれかとしてデータベースに送られます。

- データ操作言語（DML）バインド変数
- PL/SQL IN パラメータ
- オブジェクト型属性値

Java オブジェクトは、STRUCT クラスのインスタンス、または SQLData または ORAData インタフェースを実装するクラスのインスタンスです。Oracle JDBC ドライバは、データベースの SQL エンジンが認識できる線形化された形式に Java オブジェクトを変換します。サブタイプ・オブジェクトは、通常のオブジェクトの場合と同じ方法でバインドされます。

サブタイプ・データ・フィールドへのアクセス

サブタイプ・データ・フィールドにアクセスするためのロジックはカスタマイズされたクラスの一部ですが、デフォルト・マッピングではこのロジックは JDBC アプリケーションそのものに定義されます。データベース・オブジェクトは、oracle.sql.STRUCT クラスのインスタンスとして戻されます。JDBC アプリケーションは、STRUCT クラスの次のいずれかのアクセス・メソッドをコールして、データ・フィールドにアクセスする必要があります。

- `Object[] getAttribute()`
- `oracle.sql.Datum[] getOracleAttribute()`

getAttribute() メソッドのサブタイプ・データ・フィールド

JDBC 2.0 では、java.sql.Struct インタフェースの getAttribute() メソッドを使用してオブジェクト・データ・フィールドにアクセスします。このメソッドは、java.lang.Object 配列を戻します。配列の要素はそれぞれ 1 つのオブジェクト属性を表します。個々の要素型を確認するには、表 5-1「Oracle データ型クラス」の JDBC 変換マトリックスから、対応する属性型を参照します。たとえば、SQL NUMBER 属性は java.math.BigDecimal オブジェクトに変換されます。getAttribute() メソッドは、そのオブジェクト型のスーパータイプに定義されたすべてのデータ・フィールドと、そのサブタイプに定義されたデータ・フィールドを戻します。最初にスーパータイプのデータ・フィールドがリストされ、次にサブタイプのデータ・フィールドがリストされます。

getOracleAttribute() メソッドのサブタイプ・データ・フィールド

getOracleAttributes() メソッドは Oracle の拡張機能メソッドで、getAttribute() メソッドよりも効率的です。getOracleAttributes() メソッドは、データ・フィールドを保持する oracle.sql.Datum 配列を戻します。oracle.sql.Datum 配列の各要素は属性を表します。個々の要素型を確認するには、表 5-1「Oracle データ型クラス」の Oracle 変換マトリックスから、対応する属性型を参照します。たとえば、SQL NUMBER 属性は oracle.sql.NUMBER オブジェクトに変換されます。getOracleAttribute() メソッドは、そのオブジェクト型のスーパータイプに定義されたすべての属性と、そのサブタイプに定義された属性を戻します。最初にスーパータイプのデータ・フィールドがリストされ、次にサブタイプのデータ・フィールドがリストされます。

次のコードは、getAttribute() メソッドの使用方法を示します。

```
// tab1.person column can store PERSON_T, STUDENT_T and PARTIMESTUDENT_T objects
ResultSet rset = stmt.executeQuery ("select person from tab1");
while (rset.next())
{
  oracle.sql.STRUCT s = (oracle.sql.STRUCT) rset.getObject(1);
  if (s != null)
  {
    String sqlname = s.getSQLTypeName();

    Object[] attrs = s.getAttribute();

    if (sqlname.equals ("SCOTT.PERSON")
    {
      System.out.println ("ssn="+((BigDecimal)attrs[0]).intValue());
      System.out.println ("name="+((String)attrs[1]));
      System.out.println ("address="+((String)attrs[2]));
    }
    else if (sqlname.equals ("SCOTT.STUDENT"))
    {
      System.out.println ("ssn="+((BigDecimal)attrs[0]).intValue());
      System.out.println ("name="+((String)attrs[1]));
      System.out.println ("address="+((String)attrs[2]));
      System.out.println ("deptid="+((BigDecimal)attrs[3]).intValue());
      System.out.println ("major="+((String)attrs[4]));
    }
    else if (sqlname.equals ("SCOTT.PARTTIMESTUDENT"))
    {
      System.out.println ("ssn="+((BigDecimal)attrs[0]).intValue());
      System.out.println ("name="+((String)attrs[1]));
      System.out.println ("address="+((String)attrs[2]));
      System.out.println ("deptid="+((BigDecimal)attrs[3]).intValue());
      System.out.println ("major="+((String)attrs[4]));
      System.out.println ("numHours="+((BigDecimal)attrs[5]).intValue());
    }
```

```
    else
      throw new Exception ("Invalid type name: "+sqlname);
  }
}
rset.close ();
stmt.close ();
conn.close ();
```

継承メタデータ・メソッド

Oracle9*i* JDBC ドライバでは、継承プロパティにアクセスするための一連のメタデータ・メソッドを提供します。継承メタデータ・メソッドは、oracle.sql.StructDescriptor クラスおよび oracle.jdbc.StructMetaData クラスに定義されます。

oracle.sql.StructDescriptor クラスには、次の継承メタデータ・メソッドが含まれます。

- String[] getSubtypeNames() : ダイレクト・サブタイプの SQL 型名を戻します。
- boolean isFinalType() : オブジェクト型が最終（FINAL）型かどうかを示します。サブタイプが作成できない場合は、そのオブジェクト型は FINAL 型です。デフォルトは FINAL 型で、サブタイプの作成を可能にする場合は、型宣言部に NOT FINAL キーワードを指定する必要があります。
- boolean isSubTyp() : オブジェクト型がサブタイプかどうかを示します。
- boolean isInstantiable() : オブジェクト型がインスタンス化できるかどうかを示します。この型のインスタンスを作成できない場合、そのオブジェクト型は NOT INSTANTIABLE（インスタンス化不可）になります。
- String getSupertypeName() : ダイレクト・スーパータイプの SQL 型名を戻します。
- int getLocalAttributeCount() : サブタイプに定義された属性数を戻します。

StructMetaData クラスは、サブタイプ属性の継承メタデータ・メソッドを提供します。StructDescriptor クラスの getMetaData() メソッドは、その型の StructMetaData のインスタンスを戻します。StructMetaData クラスには、次の継承メタデータ・メソッドが含まれます。

- int getLocalColumnCount() : サブタイプに定義された属性数を戻します。これは、StructDescriptor クラスの getLocalAttributeCount() メソッドと同じです。
- boolean isInherited(int column) : 属性が継承されたかどうかを示します。列は 1 から始まります。

JPublisher を使用したカスタム・オブジェクト・クラスの作成

カスタム・オブジェクト・クラスを作成するには、その他のカスタム Java クラスを作成するときと同様に、Oracle JPublisher ユーティリティを使用すると便利です。このユーティリティにより、カスタム Java クラスの完全なクラス定義が生成されます。このカスタム Java クラスをインスタンス化することにより、Oracle オブジェクトのデータを保持できます。JPublisher が生成するクラスには、SQL と Java の間でデータを変換するメソッド、およびオブジェクト属性の getter メソッドと setter メソッドが含まれます。

この項では、概要を簡単に説明します。詳細は、『Oracle9*i* JPublisher ユーザーズ・ガイド』を参照してください。

JPublisher の機能

JPublisher を使用して、JPublisher 型マッピングの設定に従った `SQLData` インタフェースまたは `ORAData` インタフェースを実装するカスタム・オブジェクト・クラスを作成できます。

`ORAData` インタフェースを使用する場合、JPublisher によって、Oracle オブジェクト型のオブジェクト参照にマップされるカスタム参照クラスも作成されます。`SQLData` インタフェースを使用する場合、カスタム参照クラスは作成されません。かわりに、標準 `java.sql.Ref` インスタンスを使用します。

機能を追加する場合は、必要に応じてカスタム・オブジェクト・クラスをサブクラス化し、機能を追加します。JPublisher を実行するとき、生成するクラスの名前と実装するサブクラスの名前の両方を指定するコマンドライン・オプションがあります。SQL と Java のマッピングを正しく機能させるために、JPublisher はサブクラスの名前を必要とします。サブクラス名は、生成されるクラスの機能の一部に組み込まれます。

注意： JPublisher で生成されたクラスをサブクラス化せずに手動で編集する方法はお薦めできません。クラスを手動で編集した後になんらかの理由によって JPublisher を再実行する場合は、変更を実装し直す必要があります。

JPublisher 型マッピング

JPublisher では、ユーザー定義型とその属性型を SQL と Java との間でマップする、様々な方法が用意されています。この項の残りの部分で、SQL 型のカテゴリおよび各カテゴリで使用可能なマッピング・オプションの一覧を示します。

SQL と Java の型マッピングについては、3-16 ページの「データ型マッピング」を参照してください。

JPublisher の機能またはオプションの詳細は、『Oracle9*i* JPublisher ユーザーズ・ガイド』を参照してください。

SQL 型のカテゴリ

JPublisher では、SQL 型は次のグループに分類されます。それぞれ、次の JPublisher オプションを指定します。

- ユーザー定義型（UDT）: Oracle オブジェクト、参照、コレクション。

  JPublisher `-usertypes` オプションを使用して、UDT の型マッピング実装が標準 `SQLData` 実装か、Oracle 固有の `ORAData` 実装かを指定します。

- 数値型 : SQL 型 `NUMBER` としてデータベースに格納されるもの。

  JPublisher `-numbertypes` オプションを使用して、数値型の型マッピングを指定します。

- LOB 型 : SQL 型 `BLOB` および `CLOB`。

  JPublisher `-lobtypes` オプションを使用して、LOB 型の型マッピングを指定します。

- 組込み型 : 前述のカテゴリに当てはまらない SQL 型としてデータベースに格納されるもので、`CHAR`、`VARCHAR2`、`LONG`、`RAW` など。

  JPublisher `-builtintypes` オプションを使用して、組込み型の型マッピングを指定します。

型マッピング・モード

JPublisher では、次の型マッピング・モードが定義されています。このうち 2 つは、数値型にのみ適用されます。

- JDBC マッピング（`jdbc` を設定）: SQL 型とネイティブな Java 型の標準デフォルト・マッピングを使用します。カスタム・オブジェクト・クラスの場合、`SQLData` 実装を使用します。
- Oracle マッピング（`oracle` を設定）: 対応する `oracle.sql` 型を使用して、SQL 型にマップします。カスタム・オブジェクト・クラス、参照クラスまたはコレクション・クラスの場合、`ORAData` 実装を使用します。

- オブジェクト JDBC マッピング（数値型のみ）（objectjdbc を設定）：これは、JDBC マッピングの拡張要素です。オブジェクト JDBC マッピングでは、適切であれば、Java プリミティブ型（int、float、double など）のかわりに、標準 java.lang パッケージの数値オブジェクト型（java.lang.Integer、Float、Double など）が使用されます。java.lang 型は NULL 化可能ですが、プリミティブ型は NULL 化できません。
- BigDecimal マッピング（数値型のみ）（bigdecimal を設定）：java.math.BigDecimal を使用して、すべての数値属性をマップします。oracle.sql.NUMBER クラスにマップせずに、大きな数値を扱う場合に適しています。

注意： BigDecimal マッピングを使用すると、パフォーマンスが大幅に低下することがあります。

Oracle オブジェクト型から Java へのマッピング

JPublisher -usertypes オプションを使用すると、Oracle オブジェクト型に対応するカスタム Java クラスを実装する方法が判断されます。

- -usertypes=oracle と設定すると（デフォルト設定）、JPublisher によってカスタム・オブジェクト・クラスの ORAData 実装が作成されます。

  また、対応するカスタム参照クラスの ORAData 実装が作成されます。

- -usertypes=jdbc と設定すると、JPublisher によってカスタム・オブジェクト・クラスの SQLData 実装が作成されます。カスタム参照クラスは作成されません。参照型には、java.sql.Ref または oracle.sql.REF を使用する必要があります。

次の項では、オブジェクト属性に使用できる型マッピング・オプションについて説明します。

注意： SQL コレクション型をマップするときも、JPublisher を使用して -usertypes=oracle と設定し、ORAData 実装を作成します。

SQL コレクション型をマップする場合、-usertypes=jdbc の設定は使用できません。（SQLData インタフェースは、Oracle オブジェクト型のマッピング専用です。）

属性型から Java へのマッピング

Oracle オブジェクト型の属性型のマッピングを指定しない場合、JPublisher は次のデフォルトを使用します。

- 数値属性型の場合、デフォルト・マッピングはオブジェクト JDBC です。
- LOB 属性型の場合、デフォルト・マッピングは Oracle です。
- 組込み型属性の場合、デフォルト・マッピングは JDBC です。

代替マッピングが必要な場合は、必要に応じて、属性型と必要なマッピングに対応する `-numbertypes`、`-lobtypes` および `-builtintypes` オプションを使用します。

属性型自体が Oracle オブジェクト型の場合、`-usertypes` の設定に従ってマップされます。

重要： カスタム・オブジェクト・クラスに対して `SQLData` 実装を指定した場合、コードを移植可能にするには、属性値に対して移植可能なマッピングを使用する必要があることに十分注意してください。数値型と組込み型のデフォルトは移植可能ですが、LOB 型には `-lobtypes=jdbc` を指定する必要があります。

SQL 型カテゴリとマッピング設定のサマリー

表 8-1 に、SQL 型の JPublisher カテゴリ、各カテゴリで使用できるマッピング設定およびデフォルト設定を示します。

表 8-1 JPublisher の SQL 型カテゴリ、サポートされる設定およびデフォルト

SQL 型カテゴリ	JPublisher マップ・オプション	マッピング設定	デフォルト
UDT 型	-usertypes	oracle、jdbc	oracle
数値型	-numbertypes	oracle、jdbc、objectjdbc、bigdecimal	objectjdbc
LOB 型	-lobtypes	oracle、jdbc	oracle
組込み型	-builtintypes	oracle、jdbc	jdbc

注意： 前のリリースで使用されていた JPublisher `-mapping` オプションは将来廃止されますが、現在はサポートされています。JPublisher の `-mapping` オプション設定を新しいマッピング・オプションの設定に変換する方法については、『Oracle9*i* JPublisher ユーザーズ・ガイド』を参照してください。

オブジェクト型の記述

Oracle JDBC には、構造化オブジェクト型の属性名と型に関する情報を取り出す機能が含まれています。この機能は、概念的には結果セットから列名と型に関する情報を取り出す機能に似ています。実際、ほぼ同一のメソッドを使用します。

オブジェクト・メタデータの取出し機能

8-4 ページの「STRUCT 記述子」および 8-5 ページの「StructDescriptor および STRUCT オブジェクト作成の手順」で説明した `oracle.sql.StructDescriptor` クラスには、構造化オブジェクト型に関するメタデータを取り出す機能が含まれています。

`StructDescriptor` クラスには、結果セット・オブジェクトで使用可能な標準 `getMetaData()` メソッドと同じ機能を持つ `getMetaData()` メソッドがあります。このメソッドは、属性名や型など、属性情報の集合を戻します。このメソッドを `StructDescriptor` オブジェクトでコールして、その `StructDescriptor` オブジェクトによって記述されている Oracle オブジェクト型に関するメタデータを取り出します。(構造化オブジェクト型には、それぞれ対応付けられた `StructDescriptor` オブジェクトが必要です。)

`StructDescriptor` クラスの `getMetaData()` メソッドのシグネチャは、標準 `ResultSet` インタフェースの `getMetaData()` に指定されているシグネチャと同じです。

- `ResultSetMetaData getMetaData() throws SQLException`

ただし、このメソッドは、実際には `oracle.jdbc.StructMetaData` のインスタンスを戻します。このクラスは、標準 `java.sql.ResultSetMetaData` インタフェースが結果セット・メタデータのサポートを指定するのと同じ方法で、構造化オブジェクト・メタデータをサポートします。

`StructMetaData` クラスには、次の標準メソッドが含まれます。これらのメソッドは、`ResultSetMetaData` でも指定されています。

- `String getColumnName(int column) throws SQLException`

  「salary」など、指定された属性の名前を指定する `String` を戻します。

- `int getColumnType(int column) throws SQLException`

  `java.sql.Types` および `oracle.jdbc.OracleTypes` クラスに従って、指定された属性のタイプコードを指定する `int` を戻します。

- `String getColumnTypeName(int column) throws SQLException`

  「BigDecimal」など、指定された属性の型を指定する文字列を戻します。

- `int getColumnCount() throws SQLException`

  オブジェクト型の属性数を戻します。

次のメソッドは、StructMetaData でのみサポートされます。

- String getOracleColumnClassName(int column)
  throws SQLException

  OracleResultSet クラスの getOracleObject() メソッドがコールされ、指定された属性の値が取り出された場合にインスタンスが作成される oracle.sql.Datum サブクラスの完全修飾名を戻します。たとえば、「oracle.sql.NUMBER」が戻されます。

  getOracleColumnClassName() メソッドを使用するには、ResultSetMetaData オブジェクト（getMetaData() メソッドから戻されたもの）を StructMetaData オブジェクトにキャストする必要があります。

注意： 前述のメソッド・シグネチャの「column」は、誤解を招く表現です。「column」に 4 を指定すると、オブジェクトの 4 番目の属性を指すことになります。

オブジェクト・メタデータを取り出す手順

次の手順を使用して、構造化オブジェクト型のメタデータを取り出します。

1. 該当する構造化オブジェクト型を記述する StructDescriptor インスタンスを作成または取り出します。
2. StructDescriptor インスタンスの getMetaData() メソッドをコールします。
3. 適切なメタデータ getter メソッド（getColumnName()、getColumnType() および getColumnTypeName()）をコールします。

注意： 構造化オブジェクトの属性自身が構造化オブジェクトの場合、手順 1 ～ 3 を繰り返します。

例： 次のメソッドは、構造化オブジェクト型の属性情報を取り出す方法を示します。この例には、StructDescriptor インスタンスを作成する最初の手順が含まれています。

```
//
// Print out the ADT's attribute names and types
//
void getAttributeInfo (Connection conn, String type_name) throws SQLException
{
  // get the type descriptor
  StructDescriptor desc = StructDescriptor.createDescriptor (type_name, conn);

  // get type meta data
  ResultSetMetaData md = desc.getMetaData ();
```

```
  // get # of attrs of this type
  int numAttrs = desc.length ();

  // temporary buffers
  String attr_name;
  int attr_type;
  String attr_typeName;

  System.out.println ("Attributes of "+type_name+" :");
  for (int i=0; i<numAttrs; i++)
  {
    attr_name = md.getColumnName (i+1);
    attr_type = md.getColumnType (i+1);
    System.out.println (" index"+(i+1)+" name="+attr_name+" type="+attr_type);

    // drill down nested object
    if (attrType == OracleTypes.STRUCT)
    {
      attr_typeName = md.getColumnTypeName (i+1);

      // recursive calls to print out nested object meta data
      getAttributeInfo (conn, attr_typeName);
    }
  }
}
```

SQLJ オブジェクト型

この項では、『Information Technology - SQLJ - Part 2 : SQL Types using the Java TM Programming Language』（ANSI NCITS 331.2-2000）に準拠して、ユーザー定義オブジェクト型用の SQL 型である SQLJ オブジェクト型に Oracle9*i* JDBC ドライバを使用してアクセスする方法を説明します。

注意： SQLJ オブジェクト型は、シリアル化または SQL 表記で表すことができます。Oracle では、SQLJ オブジェクト型のシリアル化表記をサポートしていないため、本書では SQLJ オブジェクト型の SQL 表記についてのみ説明します。

『Information Technology - SQLJ - Part 2』によれば、SQLJ オブジェクト型は Java 用に設計されたデータベース・オブジェクト・タイプです。SQLJ オブジェクト型は Java クラスにマップします。拡張 SQL CREATE TYPE コマンド（DDL 文）を使用してマッピングを登録すると、Oracle9*i* JDBC ドライバを使用して、Java アプリケーションとデータベースの間で直接、Java オブジェクトを挿入または選択することが可能になります。データベース SQL エンジンは、データベースに SQL 属性として格納されている Java オブジェクトのデータ・フィールドにアクセスしたり、Java オブジェクトに定義されているメソッドをコールできます。

拡張 SQL CREATE TYPE コマンドの詳細は、8-52 ページの「SQLJ オブジェクト型の Java クラス定義の作成」を参照してください。

SQLJ オブジェクト型機能には、次のような特長があります。

- 拡張 SQL CREATE TYPE コマンドを使用して既存の Java クラスを SQL に対して発行し、SQL 型と Java 型の間のマッピングを作成します。型マップは不要です。
- データベースの Java オブジェクトにアクセスする標準の方法を提供します。
- Java オブジェクトを永続的に格納するための標準の方法を提供します。
- SQL の静的ファンクションを使用して Java クラスの静的フィールドにアクセスし、他の作用を持つ SQL メンバー・ファンクションを定義します。これは、UPDATE 文で役立ちます。

注意： SQLJ オブジェクト型機能は、カスタム Java クラスを使用して Oracle オブジェクト型（SQL オブジェクト型）をマップするのとほぼ同じです。SQLJ オブジェクト型機能とカスタム Java クラス機能の違いは、SQLJ オブジェクト型ではカスタム Java クラス機能とは逆に、Java クラスをもとに、対応する SQL 型を作成することです。詳細は、8-10 ページの「Oracle オブジェクト用のカスタム・オブジェクト・クラスの作成と使用方法」および 8-61 ページの「SQLJ オブジェクト型とカスタム・オブジェクト型」を参照してください。

SQLJ オブジェクト型に関する追加情報は、ANSI Web サイトから入手できます。

http://www.ansi.org/

SQL 表現での SQLJ オブジェクト型の作成

データベースに SQLJ オブジェクト型を作成するには、次の 3 つの手順を実行する必要があります。

1. データベースでアクセスするインスタンスの元となる Java クラスを作成します。

   次の「SQLJ オブジェクト型の Java クラス定義の作成」を参照してください。

2. データベースにクラス定義をロードします。

   詳細は、8-54 ページの「データベースへの Java クラスのロード」を参照してください。

3. Oracle9*i* の拡張 SQL `CREATE TYPE` コマンドを使用して、Java 型を表す SQLJ オブジェクト型を作成します。

   詳細は、8-54 ページの「データベースでの SQLJ オブジェクト型の作成」を参照してください。

SQLJ オブジェクト型の Java クラス定義の作成

SQLJ オブジェクト型機能を使用するには、Java クラスに次のいずれかの Java インタフェースを実装する必要があります。

- `java.sql.SQLData`
- `oracle.sql.ORAData`（および `oracle.sql.ORADataFactory`）

> **注意：** `CustomDatum` インタフェースにかわって `ORAData` インタフェースが導入されました。前者のインタフェースは Oracle9*i* では廃止されましたが、下位互換性を保持するためにサポートされています。詳細は、8-27 ページの「CustomDatum インタフェースの廃止」を参照してください。

従来の Oracle JDBC 実装のユーザー定義の Oracle オブジェクト型に対応するカスタム Java クラスと同様に、SQLJ オブジェクト型に対応する Java クラスは、`SQLData` インタフェースまたは `ORAData` および `ORADataFactory` インタフェースを実装します。Java クラスは、SQL と Java の間でデータを移動するためのメソッドを提供します。つまり、`SQLData` インタフェースを実装するクラスの場合は、`readSQL()` または `writeSQL()` メソッドを使用し、`ORAData` インタフェースを実装するクラスには `toDatum()` メソッドを使用します。

次のコードは、Person クラスの SQLJ オブジェクト型 PERSON_T が SQLData インタフェースを実装する方法を示します。

```
import java.sql.*;
import java.io.*;

public class Person implements SQLData
{
  private String sql_type = "SCOTT.PERSON_T";
  private int ssn;
  private String name;
  private Address address;
  public String getName() {return name;}
  public void setName(String nam) {name = nam;};
  public Address getAddress() {return address;}
  public void setAddress(Address addr) {address = addr;}

  public Person () {}

  public String getSQLTypeName() throws SQLException { return sql_type; }

  public void readSQL(SQLInput stream, String typeName) throws SQLException
  {
    sql_type = typeName;
    ssn = stream.readInt();
    name = stream.readString();
    address = stream.readObject();
  }

    public void writeSQL(SQLOutput stream) throws SQLException
  {
    stream.writeInt (ssn);
    stream.writeString (name);
    stream.writeObject (address);
  }

  // other methods
  public int length () { ... }
}
```

データベースへの Java クラスのロード

Java クラスを作成した後に、その Java クラスをデータベースで使用可能にする必要があります。これには、Oracle loadjava ツールを使用して、Java クラスをデータベースにロードします。loadjava ツールの詳細は、『Oracle9*i* Java Tools Reference』マニュアルを参照してください。

> **注意：** また、SQL*Plus から、入力文字列として loadjava コマンドラインを指定して、dbms_java.loadjava ('...') プロシージャをコールしても、loadjava ツールを起動できます。

次のコマンドは、loadjava ツールで Person クラスをデータベースにロードします。

```
% loadjava -u SCOTT/TIGER -r -f -v Person.class
```

データベースでの SQLJ オブジェクト型の作成

SQLJ オブジェクト型を作成する最後の手順では、EXTERNAL NAME 句に対応する Java クラスを指定して、拡張 SQL CREATE TYPE コマンドを使用して型を作成します。

次のコードでは、PERSON_T を SQLJ オブジェクト型に、Person を対応する Java クラスに指定します。

```
CREATE TYPE person_t AS OBJECT EXTERNAL NAME 'Person' LANGUAGE JAVA
USING SQLData
  (ss_no number (9) external name 'ssn',
  name VARCHAR2(200) external name 'name',
  address Address_t external name 'address',
  member function length return number external name 'length () return int');
/
```

拡張 SQL CREATE TYPE コマンドは、次の機能を実行します。

- SQLJ オブジェクト型に対応する Java クラスが存在するかどうか、またこのクラスがパブリックで、USING 句に指定された必要なインタフェースを実装しているかどうかをチェックします。
- 属性、関数および Java クラスの外部名を指定してデータベース・カタログを移入します。
- 外部属性名を使用した場合、拡張 SQL CREATE TYPE コマンドにより、（EXTERNAL NAME 句で指定された）Java フィールドが存在するかどうか、またこれらのフィールドが対応する SQL 属性と互換性があるかどうかがチェックされます。

- 外部属性名を使用した場合は、拡張 SQL CREATE TYPE コマンドにより、SQL 外部ファンクションと Java クラス・メソッドが照合されます。
- コンストラクタ、外部静的変数名、および結果として self を戻す外部関数をサポートする内部クラスが生成されます。クラスは、SQL オブジェクト型と同じスキーマに格納されます。

外部 SQL CREATE TYPE コマンドの詳細は、『Oracle9*i* SQL リファレンス』を参照してください。

SQLJ オブジェクト型を作成すると、データベース表の列型や他のオブジェクト型の属性に使用できます。データベース SQL エンジンは、SQL オブジェクト型の属性およびコール・メソッドにアクセスできます。たとえば、SQL*Plus では次のような操作を実行できます。

```
SQL> select col2.ss_no from tab2;
...
SQL> select col2.length() from tab2;
...
```

外部属性名 拡張 SQL CREATE TYPE コマンドは、外部属性名（external name）と対応する Java フィールドを照合して、SQLJ オブジェクト型属性と対応する Java フィールドの互換性を検証します。外部属性名は、Java クラスのフィールドを指定します。たとえば、次のコードでは、ssn 外部名で Person Java クラスの ss_no フィールドを指定します。

```
CREATE TYPE person_t AS OBJECT EXTERNAL NAME 'Person' LANGUAGE JAVA
USING SQLData
  (ss_no number (9) external name 'ssn',
  name VARCHAR2(200) external name 'name',
  address Address_t external name 'address',
  member function length return number external name 'length () return int');
/
```

外部属性名は、SQLJ オブジェクト型の属性と対応する Java クラスのフィールドの間に 1 対 1 の対応がある場合に使用することをお薦めします（ただし、使用はオプションです）。この機能を使用した場合に、Java クラスに宣言された外部属性名が存在しなかったり、SQL 属性と外部属性型に互換性がない場合、外部 SQL CREATE TYPE コマンドの実行時に SQL エラーが発生します。または、指定した SQLData または ORAData インタフェース実装で、SQL 属性と対応する Java フィールドの間での互換性のあるマッピングがサポートされない場合には、例外が発生します。

注意： 外部属性名を指定して宣言された SQL 属性は、プライベート Java フィールドを参照できます。

外部 SQL ファンクション 拡張 SQL CREATE TYPE コマンドは、外部ファンクション（MEMBER FUNCTION または STATIC FUNCTION）と対応する Java メソッドを照合して、SQLJ オブジェクト型関数と対応する Java メソッドの互換性を検証します。外部 SQL ファンクションは、Java クラスのメソッドを指定します。

注意： 外部属性名とは異なり、外部 SQL ファンクションは必須です。

データベースで SQLJ オブジェクト型を作成するときには、1 つ以上の外部 SQL ファンクションおよび属性を宣言できます。SQLJ オブジェクト型の作成に使用できるファンクションについては、表 8-2 で説明しています。

表 8-2 SQLJ オブジェクト型に使用可能な SQL 関数の種類

ファンクション	種類	構文
静的ファンクション	Oracle 固有	`STATIC FUNCTION foo (...) RETURN NUMBER EXTERNAL NAME 'bar (...) return double'`
メンバー・ファンクション	SQLJ Part2 標準	`MEMBER FUNCTION foo (...) RETURN NUMBER EXTERNAL NAME 'todo (...) return double'`
静的 Java フィールド（パブリックのみ）の値を戻す静的ファンクション	Oracle 固有	`STATIC FUNCTION foo RETURN NUMBER EXTERNAL VARIABLE NAME 'max_length'`
Java でコンストラクタをコールする静的ファンクション	Oracle 固有	`STATIC FUNCTION foo (...) RETURN person_t EXTERNAL NAME 'Person (...) return Person'`
（オブジェクトの状態を変更する）他の作用を持つメンバー・ファンクション	SQLJ Part2 標準	`MEMBER FUNCTION foo (...) RETURN SELF AS RESULT EXTERNAL NAME 'dump (...) return Person'`

コード例 次のコードは、SQLJ オブジェクト型に宣言される代表的な外部 SQL ファンクションの例です。

```
CREATE TYPE person_t AS OBJECT EXTERNAL NAME 'Person' LANGUAGE JAVA
USING SQLData
(
  num number external name 'foo',

  STATIC function construct (num number) return person_t
  external name   'Person.Person (int) return Person',
  STATIC function maxvalue return number external variable name 'max_length',
  MEMBER function selfish (num number) return self as result
    external name 'Person.dump (java.lang.Integer) return Person'
)
```

次のコードは、Java クラス Person を表す SQLJ オブジェクト型の PERSON_T を作成する方法を示します。

```
CREATE TYPE person_t AS OBJECT EXTERNAL NAME 'Person' LANGUAGE JAVA
USING SQLData
(
  ss_no NUMBER(9) EXTERNAL NAME 'ssn',
  name VARCHAR2(100) EXTERNAL NAME 'name',
  address address_t EXTERNAL NAME 'address',
  MEMBER FUNCTION length RETURN integer EXTERNAL NAME 'length() return int'
);
```

JDBC を使用した SQLJ オブジェクト型の作成 SQL*Plus などのツールを使用して SQL で直接 SQLJ オブジェクト型を作成する以外に、JDBC コードを使用して SQLJ オブジェクト型を作成することもできます。次のコードを参照してください。

```
Connection conn = ....
Statement stmt = conn.createStatement();
String sql =
  "CREATE TYPE person_t as object external name 'Person' language java
  "  using SQLData "+
  "( "+
  "  ss_no number(9), "+
  "  name varchar2(100), "+
  "  address address_t "+
  ")";
stmt.execute(sql);
stmt.close();     // release the resource
conn.close();     // close the database connection
```

SQLJ オブジェクト型のインスタンスの挿入

SQLJ オブジェクト型インスタンスを作成するには、JDBC アプリケーションで対応する Java インスタンスを作成してから、INSERT 文を使用してそのインスタンスをデータベースに挿入します。Java インスタンスは、次のいずれかの方法で挿入できます。

- バインド変数
- PL/SQL IN パラメータ
- オブジェクト属性値

Oracle JDBC ドライバは、Java オブジェクトをデータベースの SQL エンジンが認識できる形式に変換してから、データベースに送ります。

前項で説明した person_t という SQLJ オブジェクト型を作成する場合、JDBC アプリケーションではまず Person オブジェクトを作成し、次にそのオブジェクトをデータベースに挿入します。次のコードは、person_t SQLJ オブジェクト型インスタンスを SQL の挿入文にバインドします。

```
Person person = new Person();
person.ssn = 1000;
person.name = "SCOTT";
person.address = new Address ("some street", "some city", "CA", 12345);

// insert a SQLJ Object "person_t"
PreparedStatement pstmt = conn.prepareStatement ("insert into tab1 (1, ?)");
pstmt.setObject (1, person);
pstmt.execute ();
```

SQLJ オブジェクト型の Java インスタンスをバインドすることは、通常の Oracle オブジェクト型の Java インスタンスをバインドするのと同じことです。

SQLJ オブジェクト型のインスタンスの取得

通常の JDBC アプリケーションでは、次のいずれかによって SQLJ オブジェクト型の Java インスタンスが戻されます。

- 問合せ結果
- PL/SQL OUT パラメータ
- オブジェクト型属性
- コレクション要素

いずれの場合も、Oracle ODBC ドライバは、データベース SQLJ オブジェクト型インスタンスを、対応する Java クラスのインスタンスとしてインスタンス化します。

SQLJ オブジェクト型 person_t およびデータベース表の作成方法は、8-57 ページの「コード例」を参照してください。

データベース問合せを使用した SQLJ オブジェクト型インスタンスの取得

JDBC アプリケーションで表の SQLJ オブジェクト型の列を問合せする場合、列値は SQL オブジェクト型に対応する Java クラスのインスタンスとして戻されます。

SQLJ オブジェクト型 PERSON_T の col1 という列を含む表 tab1 があるとします。PERSON_T が Java クラス Person にマップするために作成されている場合、Oracle JDBC ドライバを使用して col1 を問い合せると、Person クラスのインスタンスとしてデータが戻されます。次のコードを参照してください。

```
ResultSet rset = stmt.executeQuery ("select col1 from tab1");
while (rset.next())
  Person value = (Person) rset.getObject(1);
```

注意：

- SQLJ オブジェクト型が戻されたときにクライアントに Java クラスが存在しない場合、ランタイム例外が発生します。
- クライアントに Java クラスが存在しても、その Java クラスが変更されている場合、SQLData インタフェースの readSQL() および writeSQL() メソッド、または ORAData インタフェースの create() および toDatum() メソッドが元の SQL 属性セットとの互換性を保持していないと、SQLJ オブジェクト型を正しく読取りまたは書込みすることができません。

出力パラメータとしての SQLJ オブジェクト型インスタンスの取得

SQLJ オブジェクト型を PL/SQL の OUT パラメータとして登録するには、OracleTypes.JAVA_STRUCT タイプコードを registerOutParameter() メソッドに対する入力として使用します。次のコードを参照してください。

```
CallableStatement cstmt = conn.prepareCall (...);
cstmt.registerOutParameter (1, OracleTypes.JAVA_STRUCT, "SCOTT.PERSON_T");
...
cstmt.execute();
Person value = (Person) cstmt.getObject (1);
```

SQLJ オブジェクト型のメタデータ・メソッド

データ型のプロパティを問い合せるには、メタデータ・メソッドを使用します。SQLJ オブジェクト型のメタデータ・メソッドは、oracle.sql.StructDescriptor クラスおよび oracle.jdbc.StructMetaData インタフェースに定義されます。

型記述子を取得するには、oracle.sql.StructDescriptor クラスの静的 createDescriptor() ファクトリ・メソッドを使用します。

```
public static StructDescriptor createDescriptor(String name, Connection conn)
    throws SQLException
```

name は SQLJ オブジェクト型を、conn はデータベースへの接続を示します。

oracle.sql.StructDescriptor クラスは、次のメタデータ（インスタンス）メソッドを定義します。

- boolean isJavaObject() : 型記述子が SQLJ オブジェクト型を参照するかどうかを示します。
- String getJavaClassName() : SQLJ オブジェクト型に対応する Java クラスの名前を戻します。
- String getLanguage() : SQLJ オブジェクト型の場合は文字列 JAVA を戻し、Oracle オブジェクト型（SQL オブジェクト型）の場合は NULL を戻します。
- ResultSetMetaData getMetaData() : 結果セット・メタデータ型として、SQLJ オブジェクト型のメタデータを戻します（8-48 ページの「オブジェクト・メタデータの取出し機能」を参照してください）。
- getLocalAttributeCount() : 使用されるローカル属性の数を戻します。これには、継承を通じて使用される属性の数は含まれません。

oracle.jdbc.StructMetaData インタフェースには、次のメソッドが含まれます。

- String getAttributeJavaName(int idx): SQL 属性の相対位置が指定された場合、フィールド名を戻します。相対位置は 0（ゼロ）で始まり、継承された属性が含まれます。

SQLJ オブジェクト型とカスタム・オブジェクト型

この項では、SQLJ オブジェクト型と Oracle オブジェクト型（カスタム・オブジェクト型）の違いを説明します。

表 8-3 SQLJ オブジェクト型とカスタム・オブジェクト型の機能の比較

機能	SQLJ オブジェクト型の動作	カスタム・オブジェクト型の動作
タイプコード	SQLJ オブジェクト型を SQL の OUT パラメータとして登録するには、`OracleTypes.JAVA_STRUCT` タイプコードを使用します。`OracleTypes.JAVA_STRUCT` タイプコードは、ORAData または SQLData インタフェースを実装するクラスの `_SQL_TYPECODE` フィールドにも使用できます。このタイプコードは、`ResultSetMetaData` インスタンスおよびメタデータまたはストアド・プロシージャにレポートされます。	カスタム・オブジェクト型を SQL の OUT パラメータとして登録するには、`OracleTypes.STRUCT` タイプコードを使用します。`OracleTypes.STRUCT` タイプコードは、ORAData または SQLData インタフェースを実装するクラスの `_SQL_TYPECODE` フィールドにも使用できます。（コード例は、20-45 ページの「ORAData 実装 : ORADataExample.java」を参照してください。）`OracleTypes.STRUCT` タイプコードは、`ResultSetMetaData` インスタンスおよびメタデータまたはストアド・プロシージャにレポートされます。
作成	最初に、SQLData または ORAData および ORADataFactory インタフェースを実装する Java クラスを作成し、その Java クラスをデータベースにロードします。次に、拡張 SQL `CREATE TYPE` コマンドを発行して、SQLJ オブジェクト型を作成します。	カスタム・オブジェクト型に対して SQL `CREATE TYPE` コマンドを発行し、JPublisher を使用するか、または手動で SQLData または ORAData Java ラッパー・クラスを作成します。詳細は、8-44 ページの「JPublisher を使用したカスタム・オブジェクト・クラスの作成」を参照してください。
メソッドのサポート	外部名、コンストラクタ・コールおよび他の作用を持つメンバー関数をサポートします。詳細は、8-56 ページの表 8-2「SQLJ オブジェクト型に使用可能な SQL 関数の種類」を参照してください。	Java メソッドとして型メソッドを実装するためのデフォルト・クラスはありません。一部のメソッドは SQL で実装することも可能です。

表 8-3 SQLJ オブジェクト型とカスタム・オブジェクト型の機能の比較（続き）

機能	SQLJ オブジェクト型の動作	カスタム・オブジェクト型の動作
型マッピング	型マッピングは、拡張 SQL CREATE TYPE コマンドによって自動的に実行されます。ただし、SQLJ オブジェクト型にはクライアント上に定義 Java クラスが必要です。	型マップに SQL と Java の間の対応を登録します。それ以外の場合、型は oracle.sql.STRUCT としてインスタンス化されます。
対応する Java クラスがない場合	SQLJ オブジェクト型がクライアントに戻されたときに対応する Java クラスが存在しない場合、例外が発生します。	カスタム・オブジェクト型がクライアントに戻されたときに対応する Java クラスが存在しない場合は、oracle.sql.STRUCT が使用されます。
継承	SQL 階層を Java クラス階層にマッピングするための規則があります。この規則の詳細は、『Oracle9*i* SQL リファレンス』を参照してください。	マッピング規則はありません。

9

Oracle オブジェクト参照の操作

オブジェクト参照へのアクセスおよび操作を行う、標準 JDBC に対する Oracle 拡張機能について説明します。次の項目が含まれます。

- オブジェクト参照用 Oracle 拡張機能
- オブジェクト参照機能の概要
- オブジェクト参照の取出しと引渡し
- オブジェクト値に対する、オブジェクト参照を介したアクセスと更新
- JPublisher で生成するカスタム参照クラス

オブジェクト参照用 Oracle 拡張機能

Oracle では、Oracle データベース・オブジェクトへの参照（ポインタ）が使用できます。Oracle JDBC では、次のオブジェクト参照がサポートされます。

- SELECT リストの列
- IN または OUT バインド変数
- Oracle オブジェクトの属性
- コレクション（配列）型オブジェクトの要素

SQL では、オブジェクト参照（REF）は厳密に型指定されています。たとえば、EMPLOYEE オブジェクトへの参照は、REF のみではなく、EMPLOYEE REF として定義されます。

Oracle JDBC でオブジェクト参照を選択するときには、オブジェクトそのものではなく、オブジェクトへのポインタのみが取り出されることに注意してください。参照は、弱い型指定の oracle.sql.REF インスタンス（または、移植性の高い java.sql.Ref インスタンス）としてインスタンス化することも、強い型指定の作成済みカスタム Java クラスのインスタンスとしてインスタンス化することもできます。オブジェクト参照に使用するカスタム Java クラスを、このマニュアルではカスタム参照クラスと呼びます。このクラスは、oracle.sql.ORAData インタフェースを実装する必要があります。

oracle.sql.REF クラスは、標準 java.sql.Ref インタフェース（JDK 1.1.x では oracle.jdbc2.Ref）を実装します。

REF インスタンスは、結果セットまたはコール可能文オブジェクトから取り出せます。更新された REF インスタンスは、プリコンパイルされた SQL 文またはコール可能文オブジェクトで、データベースに戻せます。REF クラスには、基礎となるオブジェクト属性値を取り出して、設定する機能や、基礎となるオブジェクトの SQL ベース型名（EMPLOYEE など）を取り出す機能が含まれています。

カスタム参照クラスには、これと同じ機能が含まれている他に、強い型指定が適用されるという利点があります。この強い型指定により、実行時まで検出できないコーディング・エラーを、コンパイル時に発見できます。

カスタム参照クラスの詳細は、9-8 ページの「JPublisher で生成するカスタム参照クラス」を参照してください。

REF クラスを使用して SQL オブジェクト・データにアクセスする完全なサンプル・アプリケーションについては、20-31 ページの「弱い型指定のオブジェクト参照 : StudentRef.java」を参照してください。

注意：

- カスタム・オブジェクト・クラスに対して oracle.sql.ORAData インタフェースを使用する場合、対応するカスタム参照クラスにも ORAData を使用します。ただし、カスタム・オブジェクト・クラスに対して標準 java.sql.SQLData インタフェースを使用する場合、参照に使用できるのは、緩い Java 型（java.sql.Ref または oracle.sql.REF）のみです。SQLData インタフェースは、SQL オブジェクト型のマッピング専用です。
- JDBC アプリケーションでは REF オブジェクトを作成できません。データベースから既存の REF オブジェクトは取り出せます。
- 配列はオブジェクトの同様に構造化型ですが、参照できません。

オブジェクト参照機能の概要

オブジェクト参照からオブジェクト・データにアクセスし、更新するには、結果セットまたはコール可能文から参照インスタンスを取出し、プリコンパイルされた SQL 文またはコール可能文のバインド変数に渡す必要があります。オブジェクト属性へのアクセスおよび更新を行う機能は、参照インスタンスに含まれています。

この項では、次の項目の概要を説明します。

- REF インスタンスをデータベースとの間で受け渡す、文および結果セットの getter および setter メソッド。
- オブジェクト属性を取出しおよび設定する、REF クラスの機能。

カスタム参照クラスは、ARRAY クラスのかわりに使用できます。9-8 ページの「JPublisher で生成するカスタム参照クラス」を参照してください。

オブジェクト参照 getter および setter メソッド

次の結果セット、コール可能文およびプリコンパイルされた SQL 文のメソッドを使用して、オブジェクト参照の取出しおよび引渡しができます。コード例は、この章の後半で示します。

結果セットおよびコール可能文の getter メソッド. `OracleResultSet` および `OracleCallableStatement` クラスは、`REF` オブジェクトを出力パラメータとして取り出す `getREF()` および `getRef()` メソッドをサポートします。取り出される REF オブジェクトは、`oracle.sql.REF` インスタンスまたは `java.sql.Ref` インスタンス（JDK 1.1.x では `oracle.jdbc2.Ref`）です。`getObject()` メソッドも使用できます。これらのメソッドは、入力として `String` 列名または `int` 列索引を取ります。

プリコンパイルされた SQL 文およびコール可能文の Setter メソッド
`OraclePreparedStatement` および `OracleCallableStatement` クラスは、`REF` オブジェクトをバインド変数として取り、データベースに渡す、`setREF()` および `setRef()` メソッドをサポートします。`setObject()` メソッドも使用できます。これらのメソッドは、入力として `String` パラメータ名または `int` パラメータ索引を取ります。同時に、それぞれ `oracle.sql.REF` インスタンスまたは `java.sql.Ref` インスタンス（JDK 1.1.x では `oracle.jdbc2.Ref`）も取ります。

主要な REF クラス・メソッド

次の `oracle.sql.REF` クラス・メソッドを使用して、SQL オブジェクト型名を取り出したり、基礎となるオブジェクト・データの取出しおよび引渡しを行います。

- `getBaseTypeName()`: 参照オブジェクトの完全修飾 SQL 構造化型名（`EMPLOYEE` など）を取り出します。

  これは、`java.sql.Ref` インタフェースで指定される標準メソッドです。

- `getValue()`: 参照オブジェクトをデータベースから取り出し、属性値にアクセスできるようにします。型マップ・オブジェクトを取ることも、データベース接続オブジェクトのデフォルト型マップを使用することもできます。

  このメソッドは Oracle 拡張機能です。

- `setValue()`: 参照オブジェクトをデータベースに設定し、属性値を更新できるようにします。入力として、オブジェクト型のインスタンスを取ります（`STRUCT` インスタンスまたはカスタム・オブジェクト・クラスのインスタンス）。

  このメソッドは Oracle 拡張機能です。

オブジェクト参照の取出しと引渡し

この項では、オブジェクト参照の取出しと引渡しを行う JDBC 機能を説明します。

結果セットからのオブジェクト参照の取出し

オブジェクト参照を取り出す方法を示すため、次の例では、まず Oracle オブジェクト型 ADDRESS を定義し、次に PEOPLE 表で ADDRESS を参照します。

```
create type ADDRESS as object
   (street_name     VARCHAR2(30),
    house_no        NUMBER);

create table PEOPLE
    (col1 VARCHAR2(30),
     col2 NUMBER,
     col3 REF ADDRESS);
```

ADDRESS オブジェクト型には、street name と house number という 2 つの属性があります。PEOPLE 表には、3 つの列、つまり文字データ用の列、数値データ用の列、および ADDRESS オブジェクトへの参照を含む列が設定されています。

オブジェクト参照を取り出すには、次の手順に従ってください。

1. 標準 SQL SELECT 文を使用して、データベース表の REF 列から参照を取り出します。
2. getREF() を使用して、結果セットから Address 参照を取り出し、REF オブジェクトに格納します。
3. Address を SQL オブジェクト型の ADDRESS に対応する Java カスタム・クラスに変換します。
4. Java クラス Address と SQL 型 ADDRESS 間の対応を、型マップに追加します。
5. getValue() メソッドを使用して、Address 参照の内容を取り出します。出力を Java Address オブジェクトにキャストします。

これらの手順（型マップへの Address の追加は除く）を実行するコードは、次のようになります。stmt は、あらかじめ定義されている文オブジェクトです。また PEOPLE データベース表の定義については、この項の始めの説明を参照してください。

```
ResultSet rs = stmt.executeQuery("SELECT col3 FROM PEOPLE");
while (rs.next())
{
   REF ref = ((OracleResultSet)rs).getREF(1);
   Address a = (Address)ref.getValue();
}
```

他の SQL 型と同様に、結果セットの getObject() メソッドを使用して参照を取り出すことができます。この場合、出力をキャストする必要があります。たとえば、次のようになります。

```
REF ref = (REF)rs.getObject(1);
```

getREF() のかわりに getObject() を使用しても、パフォーマンスは向上しません。しかし、getREF() を使用すれば、出力をキャストする必要がありません。

オブジェクト参照のコール可能文からの取出し

オブジェクト参照を PL/SQL ブロックの OUT パラメータとして取り出すには、OUT パラメータのバインド型を登録する必要があります。

1. コール可能文を OracleCallableStatement にキャストします。

   ```
   OracleCallableStatement ocs =
      (OracleCallableStatement)conn.prepareCall("{? = call func()}");
   ```

2. OUT パラメータを、次の形式の registerOutParameter() メソッドを使用して登録します。

   ```
   ocs.registerOutParameter
               (int param_index, int sql_type, String sql_type_name);
   ```

   param_index はパラメータ索引、*sql_type* は SQL タイプコード（この例では OracleTypes.REF）です。*sql_type_name* は、この参照が使用される構造化オブジェクト型名です。たとえば、OUT パラメータが ADDRESS オブジェクトに対する参照の場合（9-5 ページの「オブジェクト参照の取出しと引渡し」を参照）、渡される *sql_type_name* は ADDRESS になります。

3. コールを実行します。

   ```
   ocs.execute();
   ```

オブジェクト参照のプリコンパイルされた SQL 文への引渡し

他の SQL 型を渡すときと同様に、オブジェクト参照をプリコンパイルされた SQL 文に渡します。プリコンパイルされた SQL 文のオブジェクトの、setObject() メソッドまたは setREF() メソッドを使用します。

9-5 ページの「オブジェクト参照の取出しと引渡し」の例に続けて次のプリコンパイルされた SQL 文を使用し、ROWID に基づいてアドレス参照を更新します。

```
PreparedStatement pstmt =
   conn.prepareStatement ("update PEOPLE set ADDR_REF = ? where ROWID = ?");
((OraclePreparedStatement)pstmt).setREF (1, addr_ref);
((OraclePreparedStatement)pstmt).setROWID (2, rowid);
```

オブジェクト値に対する、オブジェクト参照を介したアクセスと更新

REF オブジェクトの setValue() メソッドを使用して、データベースにあるオブジェクトの値をオブジェクト参照から更新できます。そのためには、まず、データベース・オブジェクトに対する参照を取り出し、（まだ存在していなければ）データベース・オブジェクトに対応する Java オブジェクトを作成する必要があります。

たとえば、9-5 ページの「オブジェクト参照の取出しと引渡し」のコードを使用して、データベースの ADDRESS オブジェクトへの参照を取り出せます。

```
ResultSet rs = stmt.executeQuery("SELECT col3 FROM PEOPLE");
if (rs.next())
{
   REF ref = rs.getREF(1);
   Address a = (Address)ref.getValue();
}
```

次に、データベースの ADDRESS オブジェクトに対応する Java の Address オブジェクトを作成できます（この例では、Address クラスのコンストラクタの内容は省略しています）。REF クラスの setValue() メソッドを使用して、データベース・オブジェクトの値を設定します。

```
Address addr = new Address(...);
ref.setValue(addr);
```

この例では、setValue() メソッドにより、データベースの ADDRESS オブジェクトが即時更新されます。

JPublisher で生成するカスタム参照クラス

この章では、oracle.sql.REF クラスの機能を主に説明していますが、カスタム Java クラス、より厳密にはカスタム参照クラスからも、Oracle オブジェクト参照にアクセスできます。

カスタム参照クラスは、この章で説明した機能をすべて提供します。それに加え、強い型指定が適用されるという利点があります。カスタム参照クラスは、次の 3 つの要件を満たす必要があります。

- 8-10 ページの「Oracle オブジェクト用のカスタム・オブジェクト・クラスの作成と使用方法」で説明した oracle.sql.ORAData インタフェースを実装する必要があります。カスタム・オブジェクト・クラス用の代替である標準 JDBC SQLData インタフェースは、カスタム参照クラスには使用できないので注意してください。
- コンパニオン・クラスの場合は、カスタム参照クラスのインスタンスを作成するために、oracle.sql.ORADataFactory インタフェースを実装する必要があります。
- オブジェクト・データを参照する方法を用意する必要があります。JPublisher は、oracle.sql.REF 属性を使用することにより、これを実現します。

カスタム参照クラスは独自に作成できますが、Oracle JPublisher ユーティリティで作成すると便利です。JPublisher を使用して Oracle オブジェクトにマップするカスタム・オブジェクト・クラスを生成し、ORAData 実装を使用するように指定すると、JPublisher は ORAData と ORADataFactory を実装し、oracle.sql.REF 属性を含むカスタム参照クラスも生成します。（ORAData 実装は、JPublisher の -usertypes マッピング・オプションに oracle が設定されている場合、使用されます。これがデフォルトです。）

カスタム参照クラスは、強い型指定です。たとえば、Oracle オブジェクト EMPLOYEE を定義すると、JPublisher により Employee カスタム・オブジェクト・クラスおよび EmployeeRef カスタム参照クラスが生成されます。汎用 oracle.sql.REF インスタンスのかわりに EmployeeRef インスタンスを使用すると、実行時ではなくコンパイル時にエラーを簡単に捕捉できます。たとえば、EmployeeRef 変数に別の種類のオブジェクト参照を間違えて割り当てた場合に、簡単にエラーが検出されます。

標準 SQLData インタフェースは SQL オブジェクト・マッピングのみをサポートすることに注意してください。このため、カスタム・オブジェクト・クラスを作成するとき、JPublisher に標準 SQLData インタフェースを実装するように指示すると、カスタム参照クラスは生成されません。この場合、標準 java.sql.Ref インスタンス（または oracle.sql.REF インスタンス）を使用して、オブジェクト参照にマップすることのみが可能です。（SQLData 実装を指定するには、JPublisher の UDT 属性マッピング・オプションに jdbc を設定します。）

JPublisher の詳細は、8-44 ページの「JPublisher を使用したカスタム・オブジェクト・クラスの作成」または『Oracle9*i* JPublisher ユーザーズ・ガイド』を参照してください。

10

Oracle コレクションの操作

この章では、Java 配列とそれぞれのデータにマップされる Oracle コレクションにアクセスしたり、操作するための標準 JDBC への Oracle 拡張機能について説明します。次の項目が含まれます。

- コレクション（配列）のための Oracle 拡張機能
- コレクション（配列）機能の概要
- 配列の作成と使用方法
- 型マップを使用した配列要素のマップ
- JPublisher で生成するカスタム・コレクション・クラス

コレクション（配列）のための Oracle 拡張機能

データベース内の変数配列（VARRAY）または NESTED TABLE のいずれかである Oracle コレクションは、Java では配列にマップされます。JDBC 2.0 配列は、Java で Oracle コレクションをインスタンス化するために使用されます。コレクションおよび配列という用語はどちらも同じ意味で使用されることがありますが、データベース側ではコレクション、JDBC アプリケーション側では配列のほうが適切です。

Oracle は、名前付きコレクションのみをサポートします。このため、SQL 型名を指定してコレクション型を記述する必要があります。

JDBC では、次の配列を使用できます。

- SELECT リストの列
- `IN` または `OUT` バインド変数
- Oracle オブジェクトの属性
- 他の配列の要素（Oracle9*i* のみ）

この項の残りの部分では、コレクションの作成とインスタンス化について説明します。

この章の残りの部分では、Java 配列を介したコレクション・データへのアクセスおよび更新方法について説明します。コレクション列形式の表を作成し、内容の操作および出力を行うコード例については、20-33 ページの「弱い型指定の配列 : ArrayExample.java」を参照してください。

コレクションのインスタンス化に関する選択

アプリケーションでは、弱い型指定の `oracle.sql.ARRAY` クラスのインスタンスとして、または事前に作成した強い型指定のカスタム Java クラスのインスタンスとしてコレクションをインスタンス化するオプションがあります。コレクションのために使用するカスタム Java クラスは、このマニュアルではカスタム・コレクション・クラスといいます。カスタム・コレクション・クラスは、Oracle の `oracle.sql.ORAData` インタフェースを実装する必要があります。さらに、カスタム・クラスまたはコンパニオン・クラスは、`oracle.sql.ORADataFactory` を実装する必要があります。（標準 `java.sql.SQLData` インタフェースは、SQL オブジェクト型のマッピング専用です。）

`oracle.sql.ARRAY` クラスは、標準 `java.sql.Array` インタフェース（JDK 1.1.x では `oracle.jdbc2.Array`）を実装します。

`ARRAY` クラスには、配列全体を取り出し、配列要素のサブセットを取り出し、配列要素の SQL ベース型名を取り出すための機能があります。ただし、setter メソッドがないため、配列に書き込めません。

`ARRAY` クラスと同様に、カスタム・コレクション・クラスを使用すると、配列全体または一部を取り出して、SQL ベース型名を取得できます。また、強い型指定が適用されるという利点があります。これにより、実行時まで検出できないコーディング・エラーを、コンパイル時に発見できます。

また、JPublisher によって作成されるカスタム・コレクション・クラスは、個々にアクセス可能な要素によって、書込み可能な機能を提供します。（カスタム・コレクション・クラスで、自分で実装することもできます。）

注意： VARRAY へのアクセスと NESTED TABLE へのアクセス間で、コード上の違いはありません。ARRAY クラス・メソッドは、VARRAY または NESTED TABLE に適用されているかどうかを判断し、適切なアクションを決定して応答します。

カスタム・コレクション・クラスの詳細は、10-23 ページの「JPublisher で生成するカスタム・コレクション・クラス」を参照してください。

コレクションの作成

この項では、Oracle コレクションの作成についてバックグラウンド情報を示します。

Oracle は名前付きコレクションのみをサポートしているため、特定の VARRAY 型名または NESTED TABLE 型名を制限する必要があります。VARRAY と NESTED TABLE は、型自体ではなく型のカテゴリです。

次の SQL 構文では、作成時にコレクションに SQL 型名が割り当てられます。

```
CREATE TYPE <sql_type_name> AS <datatype>;
```

VARRAY は可変サイズの配列です。これには順序づけられたデータ要素の集合があり、すべての要素は同じデータ型です。各要素には索引があります。この索引は、VARRAY 内の要素の位置に対応する番号です。VARRAY 内の要素数が VARRAY のサイズになります。VARRAY 型の宣言時に、サイズの最大値を指定する必要があります。たとえば、次のようになります。

```
CREATE TYPE myNumType AS VARRAY(10) OF NUMBER;
```

この文では、10 要素以下の NUMBER 値を持つ VARRAY が記述された SQL 型名として、myNumType が定義されています。

NESTED TABLE は、順序付けられていない、同じデータ型データ要素の集合です。データベースは、単一の列を持つ別の表に NESTED TABLE を格納します。その列の型は組込み型またはオブジェクト型です。表がオブジェクト型の場合は、オブジェクト型の属性ごとに列を持つ複数列の表として表示することもできます。次の SQL 構文を使用して、NESTED TABLE を作成します。

```
CREATE TYPE myNumList AS TABLE OF integer;
```

この文は、INTEGER 型の NESTED TABLE に使用される表タイプが定義された SQL 型名として、myNumList を指定しています。

マルチ・レベルのコレクション型の作成

JDBC においてマルチ・レベルのコレクション型を作成する最も一般的な方法は、java.sql.Statement クラスの execute メソッドに SQL の CREATE TYPE 文を渡すことです。次のコードは、execute() メソッドを使用して、1 レベルのネストした表 first_level と 2 レベルのネストした表 second_level を作成します。

```
Connection conn = ....                                          // make a database
                                                                // connection
Statement stmt = conn.createStatement();                        // open a database
                                                                // cursor
stmt.execute("CREATE TYPE first_level AS TABLE OF NUMBER");  // create a nested
                                                                // table of number
stmt.execute("CREATE second_level AS TABLE OF first_level"); // create a two
                                                                // levels nested
                                                                // table
...                                                             // other operations
                                                                // here
stmt.close();                                                   // release the
                                                                // resource
conn.close();                                                   // close the
                                                                // database
                                                                // connection
```

マルチ・レベルのコレクション型を作成すると、実表の列またはオブジェクト型の属性として使用できます。

マルチ・レベルのコレクション型を作成するための SQL 構文と、内部コレクションの記憶表を指定する方法については、『Oracle9*i* アプリケーション開発者ガイド－オブジェクト・リレーショナル機能』を参照してください。

注意： マルチ・レベルのコレクション型は Oracle9*i* でのみ使用可能です。

コレクション（配列）機能の概要

結果セットまたはコール可能文によって配列インスタンス内のコレクション・データを取得し、プリコンパイルされた SQL 文またはコール可能文内のバインド変数として戻すことができます。

標準 `java.sql.Array` インタフェース（JDK 1.1.x では `oracle.jdbc2.Array` インタフェース）を実装する `oracle.sql.ARRAY` クラスは、Oracle コレクション（VARRAY または NESTED TABLE）のデータにアクセスし、更新するために必要な機能を提供します。

この項では、次の項目について説明します。

- データベースとの間で、Java 配列としてコレクションを受け渡すために使用する文および結果セットの getter および setter メソッド
- `ARRAY` 記述子および `ARRAY` クラス・メソッド

`ARRAY` クラスのかわりに、カスタム・コレクション・クラスを使用できます。詳細は、10-23 ページの「JPublisher で生成するカスタム・コレクション・クラス」を参照してください。

配列の getter メソッドと setter メソッド

Java 配列としてコレクションを取り出し、渡すには、次の結果セット、コール可能文およびプリコンパイルされた SQL 文メソッドを使用します。コード例は、この章の後半で示します。

結果セットおよびコール可能文の getter メソッド `OracleResultSet` クラスと `OracleCallableStatement` クラスは、出力パラメータとして、つまり `oracle.sql.ARRAY` インスタンスまたは `java.sql.Array` インスタンス（JDK 1.1.x では `oracle.jdbc2.Array`）として `ARRAY` オブジェクトを取り出すために、`getARRAY()` メソッドと `getArray()` メソッドをサポートします。`getObject()` メソッドも使用できます。これらのメソッドは、入力として `String` 列名または `int` 列索引を取ります。

プリコンパイルされた SQL 文およびコール可能文の Setter メソッド `OraclePreparedStatement` クラスと `OracleCallableStatement` クラスは、更新された `ARRAY` オブジェクトをバインド変数として取得してデータベースに渡すために、`setARRAY()` メソッドと `setArray()` メソッドをサポートします。`setObject()` メソッドも使用できます。これらのメソッドは、入力としてそれぞれ `oracle.sql.ARRAY` インスタンスまたは `java.sql.Array` インスタンス（JDK 1.1.x では `oracle.jdbc2.Array`）に加えて、`String` パラメータ名または `int` パラメータ索引を取ります。

ARRAY 記述子および ARRAY クラス機能

この項では、ARRAY 記述子について説明し、その機能の概要を示すために、ARRAY クラスのメソッドをリストします。

ARRAY 記述子

ARRAY オブジェクトを作成し、使用するには、その配列でインスタンス化されるコレクションの SQL 型に対して、記述子（oracle.sql.ArrayDescriptor クラスのインスタンス）が存在していることが必要です。同じ SQL 型に対応する任意の数の ARRAY オブジェクトに対して、1 つの ArrayDescriptor オブジェクトのみ必要です。

ARRAY 記述子については、10-9 ページの「ARRAY オブジェクトと記述子の作成」で詳細に説明します。

ARRAY クラス・メソッド

oracle.sql.ARRAY クラスには、次のメソッドが含まれます。

- getDescriptor(): 配列型を識別する ArrayDescriptor を戻します。
- getArray(): 配列の内容を、デフォルトの JDBC 型で取得します。オブジェクトの配列を取得すると、getArray() はデータベース接続オブジェクトのデフォルト型マップを使用してその型を決定します。
- getOracleArray(): getArray() と同一ですが、要素を oracle.sql.* 形式で取得します。
- getBaseType(): 配列要素に対応した SQL タイプコードを戻します（タイプコードの詳細は、5-21 ページの「クラス oracle.jdbc.OracleTypes」を参照してください）。
- getBaseTypeName(): この配列要素の SQL 型名を戻します。
- getSQLTypeName()（Oracle 拡張機能）: 配列全体の完全修飾された SQL 型名を戻します。
- getResultSet(): 配列要素を結果セットとしてインスタンス化します。
- getJavaSQLConnection(): この配列と対応付けられた接続インスタンス（java.sql.Connection）を戻します。
- length(): 配列内の要素数を戻します。

注意： getBaseTypeName() と getSQLTypeName() には、たとえば、次のような違いがあります。SCOTT スキーマの PERSON オブジェクトの配列に、配列型として ARRAY_OF_PERSON を定義した場合は、getBaseTypeName() は「SCOTT.PERSON」を戻し、getSQLTypeName() は「SCOTT.ARRAY_OF_PERSON」を戻します。

ARRAY パフォーマンス拡張要素メソッド

この項では、次の項目について説明します。

- Java プリミティブ型の配列としての oracle.sql.ARRAY 要素へのアクセス
- ARRAY 自動要素バッファリング
- ARRAY 自動索引作成

Java プリミティブ型の配列としての oracle.sql.ARRAY 要素へのアクセス

oracle.sql.ARRAY クラスには、配列要素を Java プリミティブ型として戻すメソッドが含まれています。これらのメソッドを使用すると、Datum インスタンスとしてコレクション要素にアクセスしてから Datum インスタンスを Java プリミティブ型に変換する方法に比べ、より容易にコレクション要素にアクセスできます。

注意： oracle.sql.ARRAY クラスのこれらの特殊なメソッドは、数値コレクションに制限されます。

次のメソッドがあります。

- ```
  public int[] getIntArray()throws SQLException

  public int[] getIntArray(long index, int count)
     throws SQLException
  ```
- ```
  public long[] getLongArray()throws SQLException

  public long[] getLongArray(long index, int count)
     throws SQLException
  ```
- ```
  public float[] getFloatArray()throws SQLException

  public float[] getFloatArray(long index, int count)
     throws SQLException
  ```
- ```
  public double[] getDoubleArray()throws SQLException

  public double[] getDoubleArray(long index, int count)
     throws SQLException
  ```
- ```
  public short[] getShortArray()throws SQLException

  public short[] getShortArray(long index, int count)
     throws SQLException
  ```
  ```

最初のシグネチャを使用する各メソッドは、コレクション要素を *XXX*[] として戻します。この *XXX* は、Java プリミティブ型を表します。2 つ目のシグネチャを使用する各メソッドは、count で指定された要素数を含むコレクションの一部を戻し、index の位置で開始されます。

ARRAY 自動要素バッファリング

Oracle JDBC ドライバには、ARRAY の内容のバッファリングを有効または無効にするためのパブリック・メソッドが用意されています。(STRUCT 属性のバッファ方法については、8-9 ページの「STRUCT 自動属性バッファリング」を参照してください。)

oracle.sql.ARRAY クラスには、次のメソッドがあります。

- public void setAutoBuffering(boolean enable)
- public boolean getAutoBuffering()

setAutoBuffering() メソッドは、自動バッファリングを有効または無効にします。getAutoBuffering() メソッドは、現行の自動バッファリング・モードを戻します。デフォルトでは、自動バッファリングは無効です。

ARRAY 要素に getAttributes() および getArray() メソッドで複数回アクセスする場合 (ARRAY データがオーバーフローすることなく JVM メモリーに格納されると想定する場合) は、JDBC アプリケーションで自動バッファリングを有効にすることをお薦めします。

重要： 変換した要素をバッファリングすると、JDBC アプリケーションでは、大量のメモリーが消費されます。

自動バッファリングを有効にすると、oracle.sql.ARRAY オブジェクトでは、変換したすべての要素のローカルのコピーが保持されます。このデータが保持されるため、この情報に 2 回目にアクセスするときにはデータ・フォーマット変換処理を実行しなくて済みます。

ARRAY 自動索引作成

配列を自動索引作成モードにすると、配列オブジェクトは配列要素へのアクセスを迅速に行うために索引表をメンテナンスします。

oracle.sql.ARRAY クラスには、自動配列索引作成をサポートする次のメソッドが含まれています。

- ```
  public synchronized void setAutoIndexing
      (boolean enable, int direction)
      throws SQLException
  ```
- ```
  public synchronized void setAutoIndexing
      (boolean enable)
      throws SQLException
  ```

setAutoIndexing() メソッドは、oracle.sql.ARRAY オブジェクトに自動索引作成モードを設定します。direction パラメータは、JDBC ドライバで最適な索引作成スキームを判別できるように配列オブジェクトをサポートします。direction パラメータには、次の値を指定できます。

- ARRAY.ACCESS_FORWARD
- ARRAY.ACCESS_REVERSE
- ARRAY.ACCESS_UNKNOWN

setAutoIndexing(boolean) メソッドのシグネチャは、アクセス方向をデフォルトで ARRAY.ACCESS_UNKNOWN として設定します。

デフォルトでは、自動索引作成は有効にされていません。JDBC アプリケーションで、getArray() および getResultSet() メソッドを使用して配列要素のランダム・アクセスを実行する場合は、ARRAY オブジェクトの自動索引作成を有効にします。

配列の作成と使用方法

この項では、配列オブジェクトを作成する方法と、配列オブジェクトとしてコレクションを取り出し、渡す方法について説明します。この項には次の項目があります。

ARRAY オブジェクトと記述子の作成

この項では、ARRAY オブジェクトおよび記述子を作成する方法について説明し、ArrayDescriptor クラスの便利なメソッドをリストします。

ArrayDescriptor および ARRAY オブジェクトの作成に関する手順

この項では、oracle.sql.ARRAY オブジェクトを作成する方法について説明します。このためには、次の操作が必要です。

1. その配列に（存在していない場合は）ArrayDescriptor オブジェクトを作成します。
2. ArrayDescriptor オブジェクトを使用して、渡す配列に対して oracle.sql.ARRAY オブジェクトを作成します。

ArrayDescriptor は oracle.sql.ArrayDescriptor クラスのオブジェクトで、配列の SQL 型を記述します。1 つの SQL 型には、1 つの配列記述子のみ必要です。ドライバは、ArrayDescriptor オブジェクトをキャッシュして、すでに遭遇した SQL 型を再作成しなくても済むようにします。同じ記述子オブジェクトを再利用して、同じ配列型に対応した oracle.sql.ARRAY オブジェクトのインスタンスを複数作成できます。

コレクションは、強い型指定です。Oracle は、名前付きコレクション、つまり SQL 型名の指定されたコレクションのみをサポートします。たとえば、CREATE TYPE 文を使用して、次のようにコレクションを作成します。

```
CREATE TYPE num_varray AS varray(22) OF NUMBER(5,2);
```

NUM_VARRAY は、コレクション型の SQL 型名です。

注意： コレクション型の名前は、要素の型名と同じではありません。たとえば、次のようになります。

```
CREATE TYPE person AS object
            (c1 NUMBER(5), c2 VARCHAR2(30));
CREATE TYPE array_of_persons AS varray(10)
              OF person;
```

この文では、コレクション型の SQL 名は、ARRAY_OF_PERSON です。コレクションの要素の SQL 名は、PERSON です。

Array オブジェクトを構築する前に、その配列の特定の SQL 型に ArrayDescriptor が存在している必要があります。ArrayDescriptor が存在しない場合は、コンストラクタにコレクション型の SQL 型名と Connection オブジェクト（JDBC はこれを使用してデータベースからメタデータを収集します）を渡して作成する必要があります。

```
ArrayDescriptor arraydesc = ArrayDescriptor.createDescriptor
                                        (sql_type_name, connection);
```

sql_type_name には配列の型名が、*connection* には使用する Connection オブジェクトが当てはまります。

配列の SQL 型に対して ArrayDescriptor オブジェクトを作成すると、ARRAY オブジェクトを作成できます。これを行うには、配列記述子、接続オブジェクトおよび配列に加える個々の要素を含む Java オブジェクトを渡します。

```
ARRAY array = new ARRAY(arraydesc, connection, elements);
```

arraydesc にはすでに作成した配列記述子が、connection には使用する接続オブジェクトが、elements には Java 配列が当てはまります。elements の内容は、次の 2 つの場合があります。

- Java プリミティブ形の配列。たとえば、int[] です。
- *xxx*[] などの Java オブジェクトの配列。*xxx* は Java クラスの名前です。たとえば、Integer[] です。

注意： OraclePreparedStatement クラスの setARRAY()、setArray() および setObject() メソッドは、オブジェクトの配列ではなく oracle.sql.ARRAY 型のオブジェクトを引数として取ります。

マルチ・レベル・コレクションの作成

シングル・レベル・コレクションと同様に、JDBC アプリケーションではマルチ・レベル・コレクションを表す oracle.sql.ARRAY インスタンスを作成して、そのインスタンスをデータベースに送ることができます。oracle.sql.ARRAY コンストラクタは、次のように定義されます。

```
public ARRAY(ArrayDescriptor type, Connection conn, Object elements)
throws SQLException
```

最初の引数は、マルチ・レベルのコレクション型を記述する oracle.sql.ArrayDescriptor オブジェクトです。2 番目の引数は、現在のデータベース接続です。3 番目の引数は、マルチ・レベル・コレクション要素を保持する java.lang.Object です。これは、シングル・レベル・コレクションの作成に使用されるのと同じコンストラクタですが、Oracle9*i* では、このコンストラクタがマルチ・レベルのコレクション型も作成できるように拡張されています。要素パラメータには、1 ディメンション配列またはネストした Java 配列を使用できるようになりました。

シングル・レベル・コレクションを作成する場合、要素は 1 ディメンション Java 配列です。マルチ・レベル・コレクションを作成する場合は、要素には、oracle.sql.ARRAY[] 要素の配列またはネストした Java 配列、あるいはその組合せを指定できます。

次のコードは、ネストした Java 配列を使用してコレクション型を作成する方法を示します。

```
Connection conn = ...;          // make a JDBC connection

// create the collection types
Statement stmt = conn.createStatement ();
stmt.execute ("CREATE TYPE varray1 AS VARRAY(10) OF NUMBER(12, 2)"); // one
                                                                     // layer
stmt.execute ("CREATE TYPE varray2 AS VARRAY(10) OF varray1"); // two layers
stmt.execute ("CREATE TYPE varray3 AS VARRAY(10) OF varray2"); // three layers
stmt.execute ("CREATE TABLE tab2 (col1 index, col2 value)");
stmt.close ();

// obtain a type descriptor of "SCOTT.VARRAY3"
ArrayDescriptor desc = ArrayDescriptor.createDescriptor("SCOTT.VARRAY3", conn);

// prepare the multi level collection elements as a nested Java array
int[][][] elems = { {{1}, {1, 2}}, {{2}, {2, 3}}, {{3}, {3, 4}} };
```

```
// create the ARRAY by calling the constructor
ARRAY array3 = new ARRAY (desc, conn, elems);

// some operations
...

// close the database connection
conn.close();
```

前述の例でのもう 1 つの実装は、oracle.sql.ARRAY[] 要素の Java 配列として elems を作成し、各 oracle.sql.ARRAY[] 要素で SCOTT.VARRAY3 を表すことです。

ArrayDescriptor メソッドの使用方法

ARRAY 記述子は、型オブジェクトとして参照可能です。この記述子には基礎となるコレクションの SQL 名、配列要素のタイプコード、および構造化オブジェクトの配列の場合は、その要素の SQL 名についての情報が示されます。記述子は、指定された型との相互変換についての情報も含んでいます。任意の型 1 つにつき、必要な記述子は 1 つのみです。その後、記述子を使用してその型の配列を必要な数のみ作成できます。

ArrayDescriptor には、要素のタイプコードおよび型名を取得するための次のメソッドがあります。

- createDescriptor(): これは ArrayDescriptor インスタンスのファクトリで、データベースで名前を参照して、その配列の特性を決定します。
- getBaseType(): この ARRAY 記述子と対応付けられた整数タイプコードを戻します。(OracleTypes クラスで定義された整数定数に従います。このクラスについては、5-15 ページの「パッケージ oracle.jdbc」で説明しています。)
- getBaseName(): この配列要素が STRUCT または REF である場合は、文字列を、配列要素に対応付けられた型名とともに戻します。
- getArrayType(): 配列が VARRAY または NESTED TABLE のどちらであるかを示す整数を戻します。可能な戻り値には、ArrayDescriptor.TYPE_VARRAY と ArrayDescriptor.TYPE_NESTED_TABLE があります。
- getMaxLength(): この配列型の要素の最大数を戻します。
- getJavaSQLConnection(): ARRAY 記述子の作成に使用された接続インスタンス (java.sql.Connection) を戻します (接続インスタンスごとに、新しい記述子を作成する必要があります)。

注意： Oracle9*i* 以前のリリースでは、コレクション内ではコレクションを使用できません。ただし、コレクション属性とともに構造化オブジェクトを使用したり、構造化オブジェクト要素とともにコレクションを使用することはできます。Oracle9*i* では、コレクション内でコレクションを使用できます。

シリアル化可能な ARRAY 記述子

10-9 ページの「ArrayDescriptor および ARRAY オブジェクトの作成に関する手順」で説明したように、ARRAY オブジェクトを作成する場合は、最初に ArrayDescriptor オブジェクトを作成する必要があります。ArrayDescriptor オブジェクトを作成するには、ArrayDescriptor.createDescriptor() メソッドをコールします。oracle.sql.ArrayDescriptor クラスはシリアル化可能です。つまり、ArrayDescriptor オブジェクトの状態を出力ストリームへ書き込み、後で使用できます。ArrayDescriptor オブジェクトを再作成するには、入力ストリームからそのシリアル化可能な状態を読み込みます。この操作をデシリアライズと呼びます。シリアル化された ArrayDescriptor オブジェクトでは、createDescriptor() メソッドをコールする必要はありません。ArrayDescriptor オブジェクトをデシリアライズするのみで済みます。

オブジェクト型が複雑で、頻繁に使用しない場合は、ArrayDescriptor オブジェクトをシリアル化することをお薦めします。

デシリアライズによって ArrayDescriptor オブジェクトを作成する場合は、setConnection() メソッドを使用して、ArrayDescriptor オブジェクトに適切なデータベース接続インスタンスを指定する必要があります。

次のコードで、ArrayDescriptor オブジェクトの接続インスタンスを指定します。

```
public void setConnection (Connection conn) throws SQLException
```

注意： JDBC ドライバでは、setConnection() メソッドからの接続オブジェクトが、型記述子の導出元と同じデータベースへ接続しているかどうかが検証されません。

配列とその要素の取出し

この項では、結果セットから ARRAY インスタンス全体を取り出す方法を最初に説明し、次に ARRAY インスタンスから要素を取り出す方法について説明します。

配列の取出し

oracle.sql.ARRAY オブジェクトを戻す getARRAY() メソッドを使用して、結果セットを OracleResultSet オブジェクトにキャストすることで、SQL 配列を取り出すことができます。結果セットをキャストするのを避けるには、java.sql.ResultSet インタフェースによって指定される標準の getObject() メソッドを使用してデータを取得し、その出力を oracle.sql.ARRAY オブジェクトにキャストします。

データ取出しメソッド

ARRAY オブジェクトに配列を作成すると、次に示す oracle.sql.ARRAY クラスの 3 つのオーバーロード・メソッドのいずれかを使用して、データを取り出すことができます。

- getArray()
- getOracleArray()
- getResultSet()

Oracle は配列のすべての要素、またはサブセットを取り出すことができるメソッドも提供しています。

> **注意：** 構造化オブジェクトの配列を操作している場合は、Oracle は、オブジェクトを Java にマップする方法を選択できるように、型マップを指定できる次の 3 つのメソッドのバージョンを提供します。

getOracleArray() getOracleArray() メソッドは、標準 Array インタフェース（JDK 1.2.x では java.sql.Array、または JDK 1.1.x では oracle.jdbc2.Array）では指定されていない Oracle 固有拡張要素です。getOracleArray() メソッドでは、配列の要素値を取り出して Datum[] 配列に格納します。この要素は、元の配列の SQL 型データに対応する oracle.sql.* データ型です。

構造化オブジェクトの配列では、このメソッドは要素のために oracle.sql.STRUCT インスタンスを使用します。

また、getOracleArray(*index*,*count*) メソッドを使用すると、配列要素のサブセットを取得できます。

getResultSet() getResultSet() メソッドでは、ARRAY オブジェクトで指定されている配列の要素を含む結果セットを戻します。結果セットには、配列要素ごとに 1 行格納されます。各行は 2 列で構成されています。第 1 列にはその要素の配列の索引、第 2 列には要素値が格納されています。VARRAY の場合は、索引は配列内の要素の位置を表します。順序が定義されていない NESTED TABLE の索引は、特定の問合せによって返された要素の順序です。

NESTED TABLE からデータを取り出すときは、getResultSet() を使用することをお薦めします。NESTED TABLE の要素数には、制限はありません。メソッドから戻された ResultSet オブジェクトのポインタの初期値は、データの第 1 行です。next() メソッドおよび適切な getXXX() メソッドを使用すると、NESTED TABLE の内容を取得できます。また、getArray() を使用すると、NESTED TABLE のすべての内容が一度に戻されます。

getResultSet() メソッドでは接続のデフォルト型マップを使用して、Oracle オブジェクトの SQL 型と対応する Java データ型とのマッピングを決定します。接続のデフォルト型マップを使用しない場合は、getResultSet(*map*) を使用して別の型マップを指定できます。

また、getResultSet(*index*,*count*) および getResultSet(*index*,*count*,*map*) メソッドを使用すると、配列のサブセットを取り出すことができます。

getArray() getArray() メソッドは、必要に応じてキャストできる java.lang.Object インスタンスに配列要素を戻す標準 JDBC メソッドです（10-15 ページの「データ取出しメソッドの比較」を参照してください）。要素は、元の配列の SQL 型データに対応する Java 型データに変換されます。

また、getArray(*index*,*count*) メソッドを使用すると、配列要素のサブセットを取り出すことができます。

データ取出しメソッドの比較

配列要素を戻すために getOracleArray() を使用する場合は、そのメソッドが oracle.sql.Datum インスタンスを使用するため、SQL から Java へのデータ変換を行う必要がなくなります。Datum（またはサブクラス）インスタンス内のデータは、未加工の SQL 形式で保持されます。

getResultSet() を使用してプリミティブ・データ型の配列を戻すと、JDBC ドライバにより ResultSet オブジェクトが戻されます。このオブジェクトには、要素ごとに、要素の配列の索引および要素値が含まれています。たとえば、次のようになります。

```
ResultSet rset = intArray.getResultSet();
```

この例では、結果セットに配列要素ごとに 1 行格納されます。各行は 2 列で構成されています。第 1 列には配列の索引、第 2 列には要素値が格納されます。

getArray() を使用してプリミティブ・データ型の配列を取り出すと、要素の値を含む java.lang.Object が戻されます。この配列の要素は Java 型で、SQL 型の要素に対応します。たとえば、次のようになります。

```
BigDecimal[] values = (BigDecimal[]) intArray.getArray();
```

intArray は oracle.sql.ARRAY を表し、NUMBER 型の VARRAY に対応しています。値配列には、java.math.BigDecimal 型の要素の配列が含まれます。Oracle JDBC ドライバでは、SQL NUMBER データ型がデフォルトで Java BigDecimal にマップされるためです。

注意： BigDecimal の使用は、Java ではリソース集中型の処理です。Oracle JDBC はデフォルトで数値 SQL データを BigDecimal にマップするため、getArray() を使用するとパフォーマンスが低下することがあります。このため、この方法は数値コレクションにはお薦めできません。

型マップに従った構造化オブジェクト配列の要素の取出し

デフォルトでは、構造化オブジェクトの要素を持つ配列を操作している場合に、getArray() または getResultSet() を使用すると、デフォルト・マッピングに従って配列の Oracle オブジェクトが、対応する Java データ型にマップされます。これらのメソッドでは、接続のデフォルト型マップを使用してマッピングが決定されるためです。

ただし、デフォルトの処理を変更する場合は、getArray(*map*) または getResultSet(*map*) メソッドを使用して、別のマッピングを含む型マップを指定できます。配列の Oracle オブジェクトに対応するエントリが型マップに存在する場合は、配列の各オブジェクトは、その型マップで指定されている、対応する Java 型にマップされます。たとえば、次のようになります。

```
Object[] object = (Object[])objArray.getArray(map);
```

この例の objArray は oracle.sql.ARRAY オブジェクトを表し、map は java.util.Map オブジェクトを表します。

型マップに特定の Oracle オブジェクトに対応するエントリが含まれない場合は、要素は oracle.sql.STRUCT オブジェクトとして戻されます。

getResultSet(*map*) メソッドは、getArray(*map*) メソッドに動作が似ています。

配列で型マップを使用する方法の詳細は、10-21 ページの「型マップを使用した配列要素のマップ」を参照してください。

配列要素のサブセットの取出し

配列の内容全体を取り出さない場合は、サブセットを取り出すことができる getArray()、getResultSet() および getOracleArray() のシグネチャを使用できます。配列のサブセットを取り出すには、索引と件数を渡して、取出しを開始する配列の位置および取り出す要素数を指定します。前の例と同様に、接続に対して型マップを指定するかデフォルトの型マップを使用して、Java 型に変換します。たとえば、次のようになります。

```
Object object = arr.getArray(index, count, map);
Object object = arr.getArray(index, count);
```

getResultSet() を使用した例です。

```
ResultSet rset = arr.getResultSet(index, count, map);
ResultSet rset = arr.getResultSet(index, count);
```

getOracleArray() を使用した例です。

```
Datum arr = arr.getOracleArray(index, count);
```

arr は oracle.sql.ARRAY オブジェクト、*index* は long 型、*count* は int 型、*map* は java.util.Map オブジェクトを表します。

注意： 配列全体ではなく、配列のサブセットを取り出すことにパフォーマンス上の利点はありません。

oracle.sql.Datum 配列への配列要素の取出し

oracle.sql.Datum[] 配列を戻すには、getOracleArray() を使用します。戻される配列の要素は、元の配列要素の SQL データ型に対応する oracle.sql.* 型です。たとえば、次のようになります。

```
Datum arraydata[] = arr.getOracleArray();
```

この例の arr は oracle.sql.ARRAY オブジェクトを表します。配列とその内容を取り出す方法の例は、20-33 ページの「弱い型指定の配列 : ArrayExample.java」を参照してください。

次の例は、接続オブジェクト conn および statement オブジェクト stmt がすでに作成済みであることを前提としています。この例では、SQL 型名 NUM_ARRAY の配列が NUMBER データの VARRAY を格納するために作成されます。この結果、NUM_ARRAY は、表 VARRAY_TABLE に格納されます。

問合せにより、VARRAY_TABLE の内容が選択されます。結果セットは、OracleResultSet オブジェクトにキャストされます。getARRAY() がそのオブジェクトに適用され、oracle.sql.ARRAY オブジェクトの my_array に配列データが取り出されます。

my_array は oracle.sql.ARRAY 型であるため、getSQLTypeName() メソッドおよび getBaseType() メソッドを my_array に適用して、配列の各要素およびその整数コードの SQL 型名を取り出すことができます。

次に、プログラムにより配列の内容が出力されます。my_array の内容は SQL データ型の NUMBER であるため、まず BigDecimal データ型にキャストする必要があります。for ループでは、配列の各値は BigDecimal にキャストされ、標準出力に出力されます。

```
stmt.execute ("CREATE TYPE num_varray AS VARRAY(10) OF NUMBER(12, 2)");
stmt.execute ("CREATE TABLE varray_table (col1 num_varray)");
stmt.execute ("INSERT INTO varray_table VALUES (num_varray(100, 200))");

ResultSet rs = stmt.executeQuery("SELECT * FROM varray_table");
ARRAY my_array = ((OracleResultSet)rs).getARRAY(1);

// return the SQL type names, integer codes,
// and lengths of the columns
System.out.println ("Array is of type " + array.getSQLTypeName());
System.out.println ("Array element is of typecode " + array.getBaseType());
System.out.println ("Array is of length " + array.length());
```

```
// get Array elements
BigDecimal[] values = (BigDecimal[]) my_array.getArray();

for (int i=0; i<values.length; i++)
{
   BigDecimal out_value = (BigDecimal) values[i];
   System.out.println(">> index " + i + " = " + out_value.intValue());
}
```

getResultSet() を使用して配列を取得する場合は、最初に結果セット・オブジェクトを取得し、次に next() メソッドを使用して操作を反復します。getInt() メソッドでは、パラメータの索引を使用して、要素の索引および要素値を取り出します。

```
ResultSet rset = my_array.getResultSet();
while (rset.next())
{
    // The first column contains the element index and the
    // second column contains the element value
    System.out.println(">> index " + rset.getInt(1)+" = " + rset.getInt(2));
}
```

マルチ・レベルのコレクション型要素へのアクセス

oracle.sql.ARRAY クラスは、コレクション要素にアクセスするための（オーバーロード可能な）3 つのメソッドを提供します。Oracle9*i* JDBC ドライバでは、これらのメソッドがマルチ・レベル・コレクションをサポートするように拡張されています。3 つのメソッドは次のとおりです。

- getArray() メソッド : JDBC 標準
- getOracleArray() メソッド : Oracle 拡張機能
- getResultSet() メソッド : JDBC 標準

getArray() メソッドは、コレクション要素を保持する Java 配列を戻します。配列要素の型は、コレクション要素の型と JDBC のデフォルトの変換マトリックスによって決まります。

たとえば、getArray() メソッドは、SQL NUMBER 型のコレクションの java.math.BigDecimal 配列を戻します。getOracleArray() メソッドは、Datum 形式でコレクション要素を保持する Datum 配列を戻します。マルチ・レベル・コレクションの場合、getArray() メソッドおよび getOracleArray() メソッドはいずれも、oracle.sql.ARRAY 要素の Java 配列を戻します。

getResultSet() メソッドは、マルチ・レベル・コレクション要素をラップする ResultSet オブジェクトを戻します。マルチ・レベル・コレクションの場合、JDBC アプリケーションは ResultSet クラスの getObject()、getARRAY() または getArray() メソッドを使用して、oracle.sql.ARRAY のインスタンスとしてコレクション要素にアクセスします。

次のコードは、getOracleArray()、getArray() および getResultSet() メソッドの使用方法を示します。

```
Connection conn = ...;          // make a JDBC connection
Statement stmt = conn.createStatement ();
ResultSet rset = stmt.executeQuery ("select col2 from tab2 where idx=1");

while (rset.next())
{
  ARRAY varray3 = (ARRAY) rset.getObject (1);
  Object varrayElems = varray3.getArray (1);  // access array elements of
    "varray3"
  Datum[] varray3Elems = (Datum[]) varrayElems;

  for (int i=0; i<varray3Elems.length; i++)
  {
    ARRAY varray2 = (ARRAY) varray3Elems[i];
    Datum[] varray2Elems = varray2.getOracleArray(); // access array elements of
      "varray2"

    for (int j=0; j<varray2Elems.length; j++)
    {
      ARRAY varray1 = (ARRAY) varray2Elems[j];
      ResultSet varray1Elems = varray1.getResultSet(); // access array elements
        of "varray1"

      while (varray1Elems.next())
        System.out.println ("idx="+varray1Elems.getInt(1)+"
          value="+varray1Elems.getInt(2));
    }
  }
}
rset.close ();
stmt.close ();
conn.close ();
```

配列の文オブジェクトへの引渡し

この項では、プリコンパイルされた SQL 文オブジェクトまたはコール可能文オブジェクトに配列を渡す方法について説明します。

配列をプリコンパイルされた SQL 文に渡す

次の手順に従って、配列をプリコンパイルされた SQL 文に渡します。（配列をコール可能文に渡すときと同様の手順を使用します。）配列は、IN または OUT バインド変数を使用できる点に注意してください。

1. 配列を格納する SQL 型の ArrayDescriptor オブジェクトを作成します（この SQL 型のオブジェクトが作成されていない場合）。ArrayDescriptor オブジェクトの作成方法の詳細は、10-9 ページの「ArrayDescriptor および ARRAY オブジェクトの作成に関する手順」を参照してください。

   ```
   ArrayDescriptor descriptor = ArrayDescriptor.createDescriptor
                                        (sql_type_name, connection);
   ```

   sql_type_name は配列のユーザー定義 SQL 型名を指定する Java 文字列、*connection* は Connection オブジェクトを表します。SQL 型名の詳細は、10-2 ページの「コレクション（配列）のための Oracle 拡張機能」を参照してください。

2. プリコンパイルされた SQL 文に oracle.sql.ARRAY オブジェクトとして渡す配列を定義します。

   ```
   ARRAY array = new ARRAY(descriptor, connection, elements);
   ```

   descriptor は手順 1 で作成された ArrayDescriptor オブジェクト、*elements* は要素の Java 配列を含む java.lang.Object を表します。

3. 実行する SQL 文を含む java.sql.PreparedStatement オブジェクトを作成します。

4. プリコンパイルされた SQL 文を OraclePreparedStatement にキャストし、OraclePreparedStatement オブジェクトの setARRAY() メソッドを使用して配列をプリコンパイルされた SQL 文に渡します。

   ```
   (OraclePreparedStatement)stmt.setARRAY(parameterIndex, array);
   ```

   parameterIndex はパラメータ索引、*array* は手順 2 で作成した oracle.sql.ARRAY オブジェクトを表します。

5. プリコンパイルされた SQL 文を実行します。

配列のコール可能文への引渡し

コレクションを PL/SQL ブロックの OUT パラメータとして取り出すには、次の手順を実行して OUT パラメータのバインド型を登録します。

1. コール可能文を OracleCallableStatement にキャストします。

```
OracleCallableStatement ocs =
   (OracleCallableStatement)conn.prepareCall("{? = call func()}");
```

2. OUT パラメータを、次の形式の registerOutParameter() メソッドに登録します。

```
ocs.registerOutParameter
        (int param_index, int sql_type, string sql_type_name);
```

param_index はパラメータ索引、*sql_type* は SQL タイプコードおよび *sql_type_name* は配列型名を表します。*sql_type* は OracleTypes.ARRAY です。

3. コールを実行します。

```
ocs.execute();
```

4. 値を取得します。

```
oracle.sql.ARRAY array = ocs.getARRAY(1);
```

型マップを使用した配列要素のマップ

配列に Oracle オブジェクトが含まれる場合は、型マップを使用して、配列のオブジェクトを、対応する Java クラスと対応付けることができます。型マップを指定しない場合または型マップに特定の Oracle オブジェクトのエントリが含まれない場合は、oracle.sql.STRUCT オブジェクトとして各要素が戻されます。

型マップを使用して、配列の Oracle オブジェクトと対応する Java クラスとのマッピングを決定するときは、マップに適切なエントリを追加する必要があります。既存の型マップへのエントリの追加方法または新しい型マップの作成方法については、8-11 ページの「SQLData を実装するための型マップ」を参照してください。

次の例では、型マップを使用して配列要素をカスタム Java オブジェクト・クラスにマップする方法を説明します。この例の配列は、NESTED TABLE です。この例では、まず、名前属性と従業員番号属性が設定されている EMPLOYEE オブジェクトを定義します。EMPLOYEE_LIST は EMPLOYEE オブジェクトの NESTED TABLE 型です。次に、会社内の部署名および各部署の従業員名を格納する EMPLOYEE_TABLE を作成します。EMPLOYEE_TABLE では、従業員名が EMPLOYEE_LIST 表の形式で格納されます。

```
stmt.execute("CREATE TYPE EMPLOYEE AS OBJECT
            (EmpName VARCHAR2(50),EmpNo INTEGER))");

stmt.execute("CREATE TYPE EMPLOYEE_LIST AS TABLE OF EMPLOYEE");
```

```
stmt.execute("CREATE TABLE EMPLOYEE_TABLE (DeptName VARCHAR2(20),
     Employees EMPLOYEE_LIST) NESTED TABLE Employees STORE AS ntable1");

stmt.execute("INSERT INTO EMPLOYEE_TABLE VALUES ("SALES", EMPLOYEE_LIST
            (EMPLOYEE('Susan Smith', 123), EMPLOYEE('Scott Tiger', 124)))");
```

SALES 部門に属するすべての従業員をカスタム・オブジェクト・クラス EmployeeObj のインスタンスの配列に取り込む場合は、EMPLOYEE SQL 型と EmployeeObj カスタム・オブジェクト・クラス間のマッピングを指定するために、型マップにエントリを追加する必要があります。

エントリを追加するには、まず、文および結果セット・オブジェクトを作成し、次に、SALES 部門に対応付けられた EMPLOYEE_LIST を選択し、結果セットに格納します。getARRAY() メソッドを使用して EMPLOYEE_LIST を ARRAY オブジェクト（次の例では employeeArray）に取り込むために、結果セットを OracleResultSet にキャストします。

この例の EmployeeObj カスタム・オブジェクト・クラスは、SQLData インタフェースを実装しています。EmployeeObj 型を作成するコード例については、20-42 ページの「カスタム・オブジェクト・クラス : SQLData 実装」を参照してください。

```
Statement s = conn.createStatement();
OracleResultSet rs = (OracleResultSet)s.executeQuery
       ("SELECT Employees FROM employee_table WHERE DeptName = 'SALES'");

// get the array object
ARRAY employeeArray = ((OracleResultSet)rs).getARRAY(1);
```

EMPLOYEE_LIST オブジェクトを取り出したので、既存の型マップを取得し、SQL 型 EMPLOYEE を Java 型 EmployeeObj にマップするエントリを追加します。

```
// add type map entry to map SQL type
// "EMPLOYEE" to Java type "EmployeeObj"
Map map = conn.getTypeMap();
map.put("EMPLOYEE", Class.forName("EmployeeObj"));
```

次に、EMPLOYEE_LIST から SQL EMPLOYEE オブジェクトを取り出します。これを行うには、employeeArray 配列オブジェクトの getArray() メソッドを実行します。このメソッドにより、オブジェクト配列が戻されます。getArray() メソッドでは、employees オブジェクト配列に EMPLOYEE オブジェクトが戻されます。

```
// Retrieve array elements
Object[] employees = (Object[]) employeeArray.getArray();
```

最後にループを作成して、各 EMPLOYEE SQL オブジェクトに、EmployeeObj Java オブジェクトの emp を割り当てます。

```
// Each array element is mapped to EmployeeObj object.
for (int i=0; i<employees.length; i++)
{
   EmployeeObj emp = (EmployeeObj) employees[i];
   ...
}
```

JPublisher で生成するカスタム・コレクション・クラス

この章では、主に oracle.sql.ARRAY クラスの機能について説明しています。ただし、カスタム Java クラスより厳密にはカスタム・コレクション・クラスからも、Oracle コレクションにアクセスできます。

カスタム・コレクション・クラスは自分で作成できますが、Oracle JPublisher ユーティリティを使用するのが最も便利な方法です。JPublisher によって生成されるカスタム・コレクション・クラスには、この章で前述したすべての機能に加えて、次の利点があります（このような機能を自分で実装することもできます）。

- カスタム・コレクション・クラスは、強い型指定です。この強い型指定により、実行時まで検出できないコーディング・エラーを、コンパイル時に発見できます。
- カスタム・コレクション・クラスは変更できます（可変）。JPublisher によって作成されるカスタム・コレクション・クラスでは、ARRAY クラスとは異なり、getElement() メソッドと setElement() メソッドを使用して、個々の要素を取得および設定できます。（カスタム・コレクション・クラスで、自分で実装することもできます。）

カスタム・コレクション・クラスは、次の 3 つの要件を満たす必要があります。

- 8-10 ページの「Oracle オブジェクト用のカスタム・オブジェクト・クラスの作成と使用方法」で説明した oracle.sql.ORAData インタフェースを実装する必要があります。カスタム・オブジェクト・クラスにかわる標準 JDBC SQLData インタフェースは、カスタム・コレクション・クラス用ではないことに注意してください。
- このクラスまたはコンパニオン・クラスで、カスタム・コレクション・クラスのインスタンスを作成するには、oracle.sql.ORADataFactory インタフェースを実装する必要があります。
- コレクション・データを格納する手段が必要です。一般に、この目的のために oracle.sql.ARRAY 属性が直接または間接に挿入されます（これは JPublisher によって作成されるカスタム・コレクション・クラスに当てはまります）。

JPublisher によって作成されるカスタム・コレクション・クラスは、ORAData と CustomDatumFactory を実装し、oracle.sql.ARRAY 属性を間接的に挿入します。生成されたカスタム・コレクション・クラスは、oracle.jpub.runtime.MutableArray 属性を持ちます。MutableArray クラスは、oracle.sql.ARRAY 属性を持ちます。

注意： JPublisher を使用してカスタム・コレクション・クラスを作成する場合は、ORAData 実装を使用する必要があります。これは JPublisher の -usertypes マッピング・オプションが、デフォルトの oracle に設定されている場合に当てはまります。

カスタム・コレクション・クラスでは SQLData 実装を使用できません（この実装は、カスタム・オブジェクト・クラス専用です）。-usertypes マッピング・オプションを jdbc に設定することは無効です。

強い型指定のカスタム・コレクション・クラスの例として、Oracle コレクション MYVARRAY を定義する場合は、JPublisher は MyVarray カスタム・コレクション・クラスを作成できます。汎用 oracle.sql.ARRAY インスタンスのかわりに MyVarray インスタンスを使用すると、実行時ではなくコンパイル中に、間違って MyVarray 変数に他の種類の配列を割り当ててしまった場合などのエラーを簡単に捕捉できます。

カスタム・コレクション・クラスを使用しない場合は、コレクションをマップするために標準 java.sql.Array インスタンス（または oracle.sql.ARRAY インスタンス）を使用します。

JPublisher の詳細は、8-44 ページの「JPublisher を使用したカスタム・オブジェクト・クラスの作成」または『Oracle9*i* JPublisher ユーザーズ・ガイド』を参照してください。

11

結果セット拡張

JDK 1.2.x の標準 JDBC 2.0 機能には、結果セット機能の拡張が含まれています。拡張には、前方または後方への処理、相対的または絶対的な位置指定、内部または外部で生じたデータベースに対する変更の参照、および結果セット・データの更新と更新による変更内容のデータベースへのコピーがあります。

この章では、これらの機能を説明します。次の項目が含まれます。

- 概要
- スクロール可能な結果セットまたは更新可能な結果セットの作成
- スクロール可能な結果セットの位置指定および処理
- 結果セットの更新
- フェッチ・サイズ
- 行の再フェッチ
- 内部的および外部的に加えられたデータベース変更の参照
- 結果セット拡張用新規メソッドのサマリー

Oracle JDBC ドライバには、この機能を JDK 1.1.x 環境でサポートするための拡張要素も含まれています。

JDBC 2.0 結果セット拡張に関する一般的な概念については、Sun 社の JDBC 2.0 API 仕様を参照してください。

概要

JDBC 2.0 結果セットの機能とカテゴリの概要を説明します。Oracle JDBC ドライバで必要な実装についても説明します。

JDBC 2.0 でサポートされる結果セット機能および結果セット・カテゴリ

JDBC 2.0 の結果セット機能には、スクロール可能性と位置指定、他からの変更に対する同期性、および更新可能性の拡張が含まれています。

- スクロール可能性、位置指定および同期性は、結果セット型で判断されます。
- 更新可能性は、並行性型で判断されます。

結果セットを作成する文オブジェクトを作成するときに、必要な結果セット型および並行性型を指定します。

様々な結果セット型と並行性型を組み合せて、6 種類の異なるカテゴリの結果セットを作成できます。

これらの拡張、型およびカテゴリの概要を説明します。

スクロール可能性、位置指定および同期性

スクロール可能性とは、結果セットを前方のみでなく後方にも移動できる機能を指します。スクロール可能性には、相対的な位置指定または絶対的な位置指定によって、結果セットの特定の位置に移動できる機能が関連します。

相対的な位置指定により、現在行から前方または後方へ、指定された行数分、移動できます。絶対的な位置指定により、結果セットの先頭または末尾から数えて、指定された行番号に移動できます。

JDBC 1.0（JDK 1.1.x）では、3-12 ページの「結果セットの処理」で説明したように、`next()` メソッドを使用して前方にのみスクロールできます。位置指定機能はありません。開始できるのは先頭のみで、1 行ずつ末尾まで反復します。

JDBC 2.0（JDK 1.2.x）では、スクロール可能で位置指定可能な結果セットも使用可能です。

スクロール可能で位置指定可能な結果セットを作成する場合、同期性も指定する必要があります。同期性とは、結果セットの外部から基礎となるデータベースに対して加えられた変更を検出し、明らかにする、結果セットの機能です。

同期性のある結果セットは、結果セットがオープンしている間に、基礎となるデータの動的なビューによって、データベースに加えられた変更を見ることができます。結果セットの行の基礎となる列値に加えられた変更を参照できます。

同期性のない結果セットは、基礎となるデータを静的に参照するので、結果セットがオープンしている間にデータベースに加えられた変更に同期しません。データベースに加えられた変更を参照するには、新しい結果セットを取り出す必要があります。

同期性は、JDBC 1.0/ 非スクロール結果セットのオプションではありません。

スクロール可能性および同期性の結果セット型

JDBC 2.0 機能で結果セットを作成する場合、特定の結果セット型を選択し、結果セットがスクロール可能で位置指定可能かどうか、基礎となるデータベースの変更と同期するかどうかを指定する必要があります。

JDBC 1.0 機能のみで十分な場合、JDBC 2.0 では、forward-only 結果セット型でサポートされます。forward-only 結果セットには、同期性は設定できません。

スクロール可能な結果セットが必要な場合、同期性も指定する必要があります。スクロール可能で、基礎となる変更に同期する結果セットには、scroll-sensitive 型を指定します。スクロール可能で、基礎となる変更に同期しない結果セットには、scroll-insensitive 型を指定します。

要約すると、JDBC 2.0 では次の 3 つの結果セット型が使用可能です。

- forward-only（JDBC 1.0 機能。スクロール不可、位置指定不可、同期性なし。）
- scroll-sensitive（スクロール可能、位置指定可能、基礎となるデータベースの変更に同期。）
- scroll-insensitive（スクロール可能、位置指定可能、基礎となるデータベースの変更に対しては同期しない。）

注意： scroll-sensitive 結果セットの同期性（外部変更を参照するために更新する頻度）は、フェッチ・サイズに影響されます。詳細は、11-20 ページの「フェッチ・サイズ」および 11-27 ページの「Scroll-Sensitive 結果セットの Oracle 実装」を参照してください。

更新可能性

更新可能性とは、結果セットのデータを更新し、その変更をデータベースにコピーできる機能を指します。この更新には、結果セットへの新しい行の挿入または既存の行の削除が含まれます。

更新可能性は、基礎となるデータベースへのアクセスを調整するために、データベースの書込みロックを必要とします。同時に複数の書込みロックは使用できないので、結果セットの更新可能性は、データベース・アクセスの並行性に関係します。

結果セットは、JDBC 2.0 ではオプションで更新可能にできますが、JDBC 1.0 では更新可能にできません。

注意： 更新可能性は、スクロール可能性および同期性から独立しています。ただし、一般に、更新または削除する特定の行に位置指定できるように、更新可能な結果セットはスクロール可能にします。

更新可能性のための並行性型

結果セットの並行性型によって、更新可能かどうかが判断されます。JDBC 2.0 では、次の並行性型が使用可能です。

- 更新可能（更新、挿入および削除が結果セットで実行可能で、データベースにコピー可能）
- 読取り専用（結果セットは変更不可）

結果セット・カテゴリの概要

スクロール可能性および同期性は更新可能性から独立しているので、3 つの結果セット型と 2 つの並行性型の組合せにより、次に示す合計 6 つの結果セット・カテゴリができます。

- forward-only/ 読取り専用
- forward-only/ 更新可能
- scroll-sensitive/ 読取り専用
- scroll-sensitive/ 更新可能
- scroll-insensitive/ 読取り専用
- scroll-insensitive/ 更新可能

注意： forward-only 更新可能結果セットには、位置指定機能がありません。`next()` メソッドで反復しながら、行を更新する必要があります。

結果セット拡張の Oracle JDBC 実装概要

この項では、スクロール可能性（クライアント側キャッシュを使用）および更新可能性（ROWID を使用）を実現する結果セット拡張の Oracle JDBC 実装に関する主要な視点について説明します。

ユーザーは、独自のクライアント側キャッシュ・メカニズムを実装できます。そのためのインタフェースが提供されています。

結果セットのスクロール可能性の Oracle JDBC 実装

基礎となるサーバーがスクロール可能なカーソルをサポートしていないので、スクロール可能性は別のレイヤーで Oracle JDBC によって実装する必要があります。

この機能は、スクロール可能な結果セットの行をクライアント側のメモリー・キャッシュに格納することにより、実現されていることに注意してください。

重要： スクロール可能な結果セットの行はすべて、クライアント側のキャッシュに格納されます。そのため、結果セットに多くの行、多くの列または非常に大きな列が含まれていると、クライアント側の Java 仮想マシンに障害が発生する可能性があります。大きな結果セットにはスクロール可能性を指定しないでください。

Oracle Server のスクロール可能なカーソル、すなわちサーバー側のキャッシュは、将来の Oracle リリースでサポートされます。

結果セットの更新可能性の Oracle JDBC 実装

更新可能性をサポートするために、Oracle JDBC は ROWID を使用して、結果セットに出現するデータベース行を一意に識別します。更新可能な結果セットに問い合せるたびに、選択された列とともに、Oracle JDBC ドライバによって自動的に ROWID が取り出されます。

注意： 更新可能性そのものは、クライアント側のキャッシュを必要としません。特に、forward-only 更新可能結果セットは、クライアント側のキャッシュを必要としません。

スクロール可能性を実現するカスタム・クライアント側キャッシュの実装

JDBC 2.0 のスクロール可能な結果セットをサポートするクライアント側キャッシュを実装する方法には、柔軟性があります。

Oracle JDBC で、完全な実装が提供されますが、独自に実装できるインタフェース `OracleResultSetCache` も提供されます。

```
public interface OracleResultSetCache
{
  /**
   * Save the data in the i-th row and j-th column.
   */
  public void put (int i, int j, Object value) throws IOException;

  /**
   * Return the data stored in the i-th row and j-th column.
   */
  public Object get (int i, int j) throws IOException;

  /**
   * Remove the i-th row.
   */
  public void remove (int i) throws IOException;
```

```
  /**
   * Remove the data stored in i-th row and j-th column
   */
  public void remove (int i, int j) throws IOException;

  /**
   * Remove all data from the cache.
   */
  public void clear () throws IOException;

  /**
   * Close the cache.
   */
  public void close () throws IOException;
}
```

このインタフェースを独自のクラスで実装する場合、アプリケーション・コードでクラスをインスタンス化した後、OracleStatement、OraclePreparedStatement または OracleCallableStatement オブジェクトの setResultSetCache() メソッドを使用して、独自の実装を使用するキャッシュ・メカニズムを設定する必要があります。次に、メソッド・シグネチャを示します。

- `void setResultSetCache(OracleResultSetCache cache)`
  `throws SQLException`

このメソッドは、問合せを実行する前にコールします。これにより、問合せによって作成される結果セットが、指定のキャッシュ・メカニズムを使用します。

スクロール可能な結果セットまたは更新可能な結果セットの作成

JDBC 1.0 では、結果セットを作成および使用するときに、特別な注意は必要ありません。結果セットは自動的に作成され、問合せの結果が格納されます。使用可能な結果セットは forward-only/ 読取り専用の 1 種類のみなので、結果セットの型またはカテゴリを指定する必要はありません。次に例を示します（接続オブジェクト conn があるとします）。

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT empno, sal FROM emp");
```

しかし、JDBC 2.0 結果セット拡張を使用する場合は、問合せを実行する一般的な文を作成するとき、またはプリコンパイルされた SQL 文やコール可能文を準備するときに、結果セット型（スクロール可能性と同期性）および並行性型（更新可能性）を指定できます。

（ただし、コール可能文は、ストアド・プロシージャおよびファンクションを実行するために用意されており、結果セットを戻すことはほとんどありません。しかし、コール可能文クラスはプリコンパイルされた SQL 文クラスのサブクラスなので、この機能を継承します。）

この項では、JDBC 2.0 拡張を使用する結果セットの作成について説明します。

結果セットのスクロール可能性および更新可能性の指定

JDBC 2.0 では、`Connection` クラスに、結果セット型および並行性型を入力として取る `createStatement()`、`prepareStatement()` および `prepareCall()` のメソッド・シグネチャが追加されています。

- `Statement createStatement (int resultSetType, int resultSetConcurrency)`
- `PreparedStatement prepareStatement (String sql, int resultSetType, int resultSetConcurrency)`
- `CallableStatement prepareCall (String sql, int resultSetType, int resultSetConcurrency)`

作成される文オブジェクトは、適切な種類の結果セットを作成するインテリジェント機能を持ちます。

結果セット型には、次の静的定数値のうち 1 つを指定できます。

- `ResultSet.TYPE_FORWARD_ONLY`
- `ResultSet.TYPE_SCROLL_INSENSITIVE`
- `ResultSet.TYPE_SCROLL_SENSITIVE`

> **注意：** パフォーマンスに与える可能性のある影響については、11-27 ページの「Scroll-Sensitive 結果セットの Oracle 実装」を参照してください。

並行性型には、次の静的定数値のうち 1 つを指定できます。

- `ResultSet.CONCUR_READ_ONLY`
- `ResultSet.CONCUR_UPDATABLE`

注意： JDK 1.1.x 環境で Oracle JDBC ドライバを使用する場合、ここで説明する静的定数は Oracle 拡張機能の一部で、OracleResultSet クラスにのみ属します。そのため、OracleResultSet クラスを指定する必要があります。たとえば、次のようになります。

```
OracleResultSet.TYPE_SCROLL_SENSITIVE
```

次の構文は使用しません。

```
ResultSet.TYPE_SCROLL_SENSITIVE
```

Statement、PreparedStatement または CallableStatement オブジェクトを作成した後、文オブジェクトで次のメソッドをコールすると、文の結果セット型および並行性型を検証できます。

- `int getResultSetType() throws SQLException`
- `int getResultSetConcurrency() throws SQLException`

例： 次のプリコンパイルされた SQL 文オブジェクトの例では、この文で実行される問合せに対して scroll-sensitive で更新可能な結果セットが指定されます（conn は接続オブジェクト）。

```
...
PreparedStatement pstmt = conn.prepareStatement
         ("SELECT empno, sal FROM emp WHERE empno = ?",
         ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);

pstmt.setString(1, "28959");
ResultSet rs = pstmt.executeQuery();
...
```

結果セットの制限事項およびダウングレード・ルール

一部の結果セット型は、ある種類の問合せには不適切です。実行する問合せに対して不適切な結果セット型または並行性型が指定されている場合、JDBC ドライバは次のルール・セットに従い、かわりに使用する最適な型を判断します。

実際の結果セット型および並行性型は、文が実行されるときに判断されます。指定された結果セット型または並行性型が他が不適切であれば、文オブジェクトで SQLWarning が発生します。SQLWarning オブジェクトには、要求された型が適切でない理由が含まれます。受け取った結果セットの型が要求した型かどうかを検証するには、警告をチェックするか、11-10 ページの「結果セット型および並行性型の検証」で説明するメソッドをコールします。

更新可能な結果セットでの FOR UPDATE 句の制限事項

更新可能な結果セットを使用する場合、問合せでは SELECT 文に FOR UPDATE 句を使用できません。FOR UPDATE 句を使用して結果セットを更新しようとすると、SQLException が発生します。

結果セットの制限事項

拡張された結果セットの問合せには、次の制限事項が適用されます。次の指針に当てはまらない場合、JDBC ドライバによって、代替の結果セット型または並行性型が選択されます。

更新可能な結果セットを作成するには、次の制限事項を考慮します。

- 問合せによって選択できるのは、単一の表のみです。結合操作を含めることはできません。

  さらに、挿入するためには、NULL 化可能でない列およびデフォルト値が設定されていない列をすべて、問合せによって選択する必要があります。
- 問合せでは「SELECT *」を使用できません。(回避方法を後で示します。)
- 問合せで選択できるのは、表列のみです。導出列や、列の集合の SUM または MAX など、集合体を選択できません。
- 問合せでは ORDER BY を使用できません。

scroll-sensitive 結果セットを作成するには、次の制限事項を考慮します。

- 問合せでは「SELECT *」を使用できません。(回避方法を後で示します。)
- 問合せによって選択できるのは、単一の表のみです。
- 問合せでは ORDER BY を使用できません。

実際には、行を再フェッチする場合、どのような結果セットでも ORDER BY は使用できません。このルールは、scroll-insensitive/ 更新可能な結果セットにも scroll-sensitive 結果セットにも適用されます。(再フェッチの概要は、11-28 ページの「結果セット拡張用新規メソッドのサマリー」を参照してください。)

回避方法「SELECT *」の制限事項を回避するには、次の例のように、表の別名を使用します。

```
SELECT t.* FROM TABLE t ...
```

ヒント: ある問合せから、scroll-sensitive または更新可能な結果セットを作成できるかどうかを判断する簡単な方法があります。ROWID 列を問合せリストに追加できる場合は、その問合せから scroll-sensitive または更新可能な結果セットを作成できます。(SQL*Plus などを使用して試すことができます。)

結果セットのダウングレード・ルール

指定された結果セット型または並行性型が不適切な場合、Oracle JDBC ドライバは次のルールを使用して、代替型を選択します。

- 指定された結果セット型が TYPE_SCROLL_SENSITIVE で、JDBC ドライバがこの要求を満たせない場合、TYPE_SCROLL_INSENSITIVE へのダウングレードが試行されます。
- 指定された（またはダウングレードされた）結果セット型が TYPE_SCROLL_INSENSITIVE で、JDBC ドライバがこの要求を満たせない場合、TYPE_FORWARD_ONLY へのダウングレードが試行されます。

また、

- 指定された並行性型が CONCUR_UPDATABLE で、JDBC ドライバがこの要求を満たせない場合、CONCUR_READ_ONLY へのダウングレードが試行されます。

注意：

- JDBC ドライバが結果セット型指定を満たせない基準については、11-9 ページの「結果セットの制限事項」のリストを参照してください。
- JDBC ドライバによる結果セット型および並行性型の操作は、それぞれ独立しています。

結果セット型および並行性型の検証

問合せを実行した後、結果セット・オブジェクトのメソッドをコールすることにより、JDBC が実際に使用した結果セット型と並行性型を検証できます。

- `int getType() throws SQLException`

  このメソッドは、問合せに対して使用された結果セット型の int 値を戻します。ResultSet.TYPE_FORWARD_ONLY、ResultSet.TYPE_SCROLL_SENSITIVE または ResultSet.TYPE_SCROLL_INSENSITIVE のどれかになります。

- `int getConcurrency() throws SQLException`

  このメソッドは、問合せに対して使用された並行性型の int 値を戻します。ResultSet.CONCUR_READ_ONLY または ResultSet.CONCUR_UPDATABLE のどちらかになります。

スクロール可能な結果セットの位置指定および処理

スクロール可能な結果セット（結果セット型 TYPE_SCROLL_SENSITIVE または TYPE_SCROLL_INSENSITIVE）を使用すると、結果セットを前方または後方に反復できます。また、目的の行に位置指定できます。

ここでは、スクロール可能な結果セットの位置指定、およびスクロール可能な結果セットを前方ではなく後方に処理する方法を説明します。

この機能を使用した完全なサンプル・アプリケーションについては、20-50 ページの「結果セットでの位置の指定 : ResultSet2.java」を参照してください。

スクロール可能な結果セットの位置指定

スクロール可能な結果セットでは、目的の位置への移動およびカレント位置のチェックを行う結果セット・メソッドを使用できます。

新しい位置へ移動するためのメソッド

スクロール可能な結果セットで新しい位置に移動するには、次の結果セット・メソッドを使用します。

- `void beforeFirst() throws SQLException`
- `void afterLast() throws SQLException`
- `boolean first() throws SQLException`
- `boolean last() throws SQLException`
- `boolean absolute(int row) throws SQLException`
- `boolean relative(int row) throws SQLException`

注意： forward-only 結果セットでは、位置を指定できません。位置の指定やカレント位置の判断を試みると、SQLException が発生します。

---

beforeFirst() メソッド 結果セットの最初の行の前に移動します。結果セットに行が含まれていない場合、何もしません。

これは、結果セットを反復して前方へ処理する場合の一般的な開始位置で、あらゆる種類の結果セットのデフォルトの開始位置です。

`beforeFirst()` をコールすると、結果セット境界の外側に移動します。有効な現在行がなくなるので、この点から相対的に位置を指定できません。

afterLast() メソッド 結果セットの最後の行の後ろに移動します。結果セットに行が含まれていない場合、何もしません。

これは、結果セットを反復して後方へ処理する場合の一般的な開始位置です。

`afterLast()` をコールすると、結果セット境界の外側に移動します。有効な現在行がなくなるので、この点から相対的に位置を指定できません。

first() メソッド 結果セットの最初の行に移動します。結果セットに行が含まれていない場合、`FALSE` を戻します。

last() メソッド 結果セットの最後の行に移動します。結果セットに行が含まれていない場合、`FALSE` を戻します。

absolute() メソッド 結果セットの先頭または末尾から、絶対行に移動します。正数を入力すると、位置は先頭からになります。負数を入力すると、位置は末尾からになります。結果セットに行が含まれていない場合、`FALSE` を戻します。

結果セットが 10 行のときに `absolute(11)` をコールするなど、最後の行を超えて前方へ移動しようとすると、最後の行の後ろに移動します。これは、`afterLast()` をコールした場合と同じ効果です。

結果セットが 10 行のときに `absolute(-11)` をコールするなど、最初の行を超えて後方へ移動しようとすると、最初の行の前に移動します。これは、`beforeFirst()` をコールした場合と同じ効果です。

注意： `absolute(1)` のコールは、`first()` のコールと等価です。`absolute(-1)` のコールは、`last()` のコールと等価です。

---

relative() メソッド 現在行から相対的な位置に移動します。正数が入力された場合は前方へ、負数が入力された場合は後方へ移動します。結果セットに行が含まれていない場合、`FALSE` を戻します。

`relative()` メソッドを使用するには、あらかじめ、結果セットの有効な現在行に位置指定しておく必要があります。

最後の行を超えて前方へ移動しようとすると、最後の行の後ろに移動します。これは、`afterLast()` をコールした場合と同じ効果です。

最初の行を超えて前方へ移動しようとすると、最初の行の前に移動します。これは、`beforeFirst()` をコールした場合と同じ効果です。

`relative(0)` は有効なコールですが、何もしません。

重要： 最初の行の前（デフォルトの開始位置）または最後の行の後ろから相対的な位置を指定できません。これらの位置から相対的な位置指定をしようとすると、SQLException が発生します。

カレント位置をチェックするメソッド

スクロール可能な結果セットでカレント位置をチェックするには、次の結果セット・メソッドを使用します。

- `boolean isBeforeFirst() throws SQLException`

  位置が最初の行の前であれば、TRUE を戻します。

- `boolean isAfterLast() throws SQLException`

  位置が最後の行の後ろであれば、TRUE を戻します。

- `boolean isFirst() throws SQLException`

  位置が最初の行であれば、TRUE を戻します。

- `boolean isLast() throws SQLException`

  位置が最後の行であれば、TRUE を戻します。

- `int getRow() throws SQLException`

  現在行の行番号を戻します。有効な現在行がない場合、0（ゼロ）を戻します。

注意： ブール・メソッド（isFirst()、isLast()、isAfterFirst() および isAfterLast()）はすべて、結果セットに行が含まれていない場合、FALSE を戻します（例外は発生しません）。

スクロール可能な結果セットの処理

スクロール可能な結果セットでは、結果セットを処理するときに、前方ではなく後方へ反復できます。次のメソッドが使用可能です。

- `boolean next() throws SQLException`
- `boolean previous() throws SQLException`

previous() メソッドの動作は、next() メソッドと似ています。新しい現在行が有効であれば TRUE を戻し、行が終了すると（最初の行を超えると）FALSE を戻します。

後方への処理と前方への処理

JDBC 1.0 の場合と同様に、next() メソッドを使用して結果セット全体を前方に向かって処理できます。詳細は、3-12 ページの「結果セットの処理」を参照してください。結果セットのデフォルトの開始位置は、最初の行の前です。結果セットが作成された後、他の位置に移動した場合、beforeFirst() メソッドをコールして戻れます。

結果セット全体を後方に向かって処理するには、afterLast() をコールし、previous() メソッドを使用します。次に例を示します（conn は接続オブジェクトです）。

```
...
/* NOTE: The specified concurrency type, CONCUR_UPDATABLE, is not relevant to this
example. */

Statement stmt = conn.createStatement
         (ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);

ResultSet rs = stmt.executeQuery("SELECT empno, sal FROM emp");

rs.afterLast();
while (rs.previous())
{
   System.out.println(rs.getString("empno") + " " + rs.getFloat("sal"));
}
...
```

相対的な位置指定とは異なり、next() は最初の行の前から、previous() は最後の行の後ろから使用できます（一般的に、そのように使用します）。これらのメソッドを使用するとき、有効な現在行に位置指定しておく必要はありません。

> **注意：** スクロール可能でない結果セットで実行できるのは、next() メソッドによる処理のみです。previous() メソッドを使用しようとすると、SQLException が発生します。

フェッチ方向のプリセット

JDBC 2.0 標準では、結果セットの処理に使用する方向をあらかじめ指定できます。この方向を、フェッチ方向と呼びます。これにより、JDBC ドライバは処理を最適化できます。次の結果セット・メソッドが指定されています。

- `void setFetchDirection(int direction) throws SQLException`
- `int getFetchDirection() throws SQLException`

Oracle JDBC ドライバは前方へのプリセット値のみをサポートします。これは、ResultSet.FETCH_FORWARD 静的定数値を入力することにより、指定できます。

値 ResultSet.FETCH_REVERSE および ResultSet.FETCH_UNKNOWN はサポートされていません。指定しようとすると、SQL 警告が発生し、設定は無視されます。

結果セットの更新

並行性型 CONCUR_UPDATABLE を使用すると、結果セットの行の更新、削除または挿入ができます。

UPDATE または INSERT 操作を結果セットで実行した後、別処理で変更をデータベースに伝播させます。変更を取り消すには、この処理をスキップします。

しかし、結果セットでの DELETE 操作は、データベースにも即時に実行されます（ただし、必ずしもコミットはされません）。

この機能を使用したサンプル・アプリケーションについては、20-53 ページの「結果セットでの行の挿入および削除 : ResultSet3.java」および 20-56 ページの「結果セットでの行の更新 : ResultSet4.java」を参照してください。

注意： 更新可能な結果セットを使用する場合、一般的にはスクロール可能にもします。これにより、変更する行に移動できます。forward-only 更新可能な結果セットの場合、next() メソッドで反復しながら、行を更新する必要があります。

結果セットでの DELETE 操作実行

結果セットの deleteRow() メソッドは、現在行を削除します。次に、メソッド・シグネチャを示します。

- `void deleteRow() throws SQLException`

重要： 結果セットの UPDATE および INSERT 操作では、変更をデータベースに伝播させる別処理が必要ですが、結果セットの DELETE 操作は、データベースの対応する行にも即時に実行されます。

deleteRow() をコールすると、変更は、次のトランザクション COMMIT 操作で確定します。デフォルトでは、自動コミット・フラグは TRUE に設定されていることにも注意してください。つまり、このデフォルトをオーバーライドしなければ、deleteRow() 操作は即時に実行され、コミットされます。

結果セットはスクロール可能でもあるとします。使用可能な位置指定メソッド（有効な現在行に移動しない beforeFirst() と afterLast() は除く）を使用して行の位置を指定し、その行を削除できます。次に、例を示します（結果セットは rs であるとします）。

```
...
rs.absolute(5);
rs.deleteRow();
...
```

位置指定メソッドの情報は、11-11 ページの「スクロール可能な結果セットの位置指定」を参照してください。

重要： 削除された行はデータベースから削除された後も、結果セット・オブジェクトに残ります。

これに対し、スクロール可能な結果セットでは、ローカルな結果セット・オブジェクトにおける DELETE 操作は理解が容易です。DELETE の後、行は結果セットに存在しません。削除された行の前の行が現在行になり、後続の行の行番号が変更されます。

詳細は、11-23 ページの「内部変更の参照」を参照してください。

結果セットでの UPDATE 操作実行

結果セットの UPDATE 操作を実行するには、2 つの別々の処理が必要です。まず、結果セットのデータを更新し、次に、データベースに変更をコピーします。

結果セットはスクロール可能でもあるとします。使用可能な位置指定メソッド（有効な現在行に移動しない beforeFirst() と afterLast() は除く）を使用して行の位置を指定し、その行を必要に応じて更新できます。

位置指定メソッドの情報は、11-11 ページの「スクロール可能な結果セットの位置指定」を参照してください。

結果セットとデータベースの行を更新する手順を示します。

1. 適切な updateXXX() メソッドをコールし、変更する列のデータを更新します。

   JDBC 2.0 では、前に使用可能だったデータベースを直接更新する setXXX() メソッドと同じように、結果セット・オブジェクトに各データ型の updateXXX() メソッドがあります。

   これらのメソッドは、列番号の int または列名の文字列と、適切なデータ型の項目を取り、新しい値を設定します。結果セット rs の例をいくつか示します。

   ```
   rs.updateString(1, "mystring");
   rs.updateFloat(2, 10000.0f);
   ```

2. updateRow() メソッドをコールして、変更をデータベースにコピーします（または、cancelRowUpdates() メソッドをコールして、変更を取り消します）。

updateRow() をコールすると、変更は実行され、次のトランザクション COMMIT 操作で確定します。デフォルトでは、自動コミット・フラグは TRUE に設定されているので、実行された操作は即時コミットされることに注意してください。

データベースにコピーする前に変更を取り消すには、かわりに cancelRowUpdates() メソッドをコールします。これにより、ローカルな結果セット・オブジェクトの対応する行も、元の値に回復します。updateRow() メソッドをコールすると、変更はトランザクションに書き込まれます。取り消すには、トランザクションをロールバックする必要があることに注意してください（ROLLBACK 操作を行うには、自動コミットを無効にしておく必要があります）。

updateRow() をコールする前に別の行に移動しても、変更は取り消され、結果セットは元の値に回復します。

updateRow() をコールする前に、通常の getXXX() メソッドをコールし、値が正しく更新されたかどうかを検証できます。これらのメソッドは、入力として int 列索引または文字列の列名を取ります。たとえば、次のようになります。

```
float myfloat = rs.getFloat(2);
...process myfloat to see if it's appropriate...
```

注意： 結果セットの UPDATE 操作は、すべての結果セット型（forward-only、scroll-sensitive および scroll-insensitive）のローカルな結果セット・オブジェクトで参照できます。

詳細は、11-23 ページの「内部変更の参照」を参照してください。

例： データベースにもコピーされる、結果セットの UPDATE 操作の例を示します。10 番目の行が更新されます。（列番号を使用して列 1 を指定し、列名 sal を使用して列 2 を指定します。）

```
...
Statement stmt = conn.createStatement
         (ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);

ResultSet rs = stmt.executeQuery("SELECT empno, sal FROM emp");

if (rs.absolute(10))        // (returns false if row does not exist)
{
   rs.updateString(1, "28959");
   rs.updateFloat("sal", 100000.0f);
   rs.updateRow();
}
// Changes will be made permanent with the next COMMIT operation.
...
```

結果セットでの INSERT 操作実行

結果セットの INSERT 操作では、結果セットの挿入行と呼ばれるものが使用されます。これは、データベースにコピーされるまで、挿入する行のデータを保持するステージング領域です。この行に明示的に移動し、挿入するデータを書き込む必要があります。

UPDATE 操作と同様に、結果セットの INSERT 操作には、まずデータを挿入行に書き込み、次にデータベースにコピーする、別々の手順が必要です。

結果セットの INSERT 操作を実行する手順を示します。

1. 結果セットの moveToInsertRow() メソッドをコールして、挿入行に移動します。

   注意： moveToInsertRow() をコールする前のカレント位置は、結果セットによって記憶されます。後で、moveToCurrentRow() をコールすると、カレント位置に戻れます。

2. UPDATE 操作と同様に、適切な updateXXX() メソッドを使用し、データを列に書き込みます。たとえば、次のようになります。

   ```
   rs.updateString(1, "mystring");
   rs.updateFloat(2, 10000.0f);
   ```

   (列番号の整数を指定するかわりに、列名の文字列を指定できます。)

   重要： 挿入行の各列の値は、その列に対して updateXXX() メソッドをコールするまで、未定義です。このメソッドをコールして、NULL 化可能でない列すべてに非 NULL 値を指定する必要があります。指定せずに行をデータベースにコピーしようとすると、SQL 例外が発生します。

   ただし、NULL 化可能な列に対しては、updateXXX() をコールしなくてもかまいません。この場合、値は NULL になります。

3. 結果セットの insertRow() メソッドをコールして、変更をデータベースにコピーします。

   insertRow() をコールすると、挿入は実行され、次のトランザクション COMMIT 操作で確定します。

   insertRow() をコールする前に別の行に移動すると、挿入は取り消され、挿入行は消去されます。

   insertRow() をコールする前に、通常の getXXX() メソッドをコールし、値が正しく挿入行に設定されたかどうかを検証できます。これらのメソッドは、入力として int 列索引または文字列の列名を取ります。たとえば、次のようになります。

```
float myfloat = rs.getFloat(2);
...process myfloat to see if it's appropriate...
```

> **注意：** どの結果セット型でも（scroll-sensitive、scroll-insensitive または forward-only）、結果セットの INSERT 操作で挿入された行を参照できません。
>
> 詳細は、11-23 ページの「内部変更の参照」を参照してください。

例： 次の例では、結果セットの INSERT 操作を実行します。挿入行に移動し、データを書き込み、データをデータベースにコピーし、挿入行に移動する前に現在行だった位置に戻ります。（列番号を使用して列 1 を指定し、列名 sal を使用して列 2 を指定します。）

```
...
Statement stmt = conn.createStatement
         (ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);

ResultSet rs = stmt.executeQuery("SELECT empno, sal FROM emp");

rs.moveToInsertRow();
rs.updateString(1, "28959");
rs.updateFloat("sal", 100000.0f);
rs.insertRow();
// Changes will be made permanent with the next COMMIT operation.
rs.moveToCurrentRow();  // Go back to where we came from...
...
```

更新の競合

JDBC ドライバで使用される更新可能な結果セットに関しては、次の事項に注意してください。

- ドライバは、更新可能な結果セットに対する書込みロックを施行しません。
- ドライバは、結果セットの DELETE または UPDATE 操作の競合をチェックしません。

DELETE または UPDATE 操作を、コミットされた別のトランザクションによって更新された行に対して実行すると、競合が発生します。

Oracle JDBC ドライバは、ROWID を使用し、データベース表の行を一意に識別します。ドライバが UPDATE または DELETE 操作をデータベースに送信しようとしたとき、ROWID が有効であれば、操作は実行されます。

コミットされた別のトランザクションによる変更は、レポートされません。競合は警告なしに無視され、新しい変更によって前の変更は上書きされます。

競合を回避するには、結果セットを作成する問合せを実行するときに、Oracle の FOR UPDATE 機能を使用します。これにより、競合を回避できますが、データへの同時アクセスも禁止されます。データ項目に同時に設定できる書込みロックは 1 つのみです。

フェッチ・サイズ

デフォルトでは、Oracle JDBC は問合せを実行するとき、データベース・カーソルから一度に 10 行の結果セットを受け取ります。これが、デフォルトの Oracle 行プリフェッチ値です。データベース・カーソルへの 1 回のラウンドトリップで取り出される行の数を変更するには、行プリフェッチ値を変更します（詳細は、12-19 ページの「Oracle 行プリフェッチ」を参照してください）。

JDBC 2.0 では、1 回のデータベース・ラウンドトリップでフェッチされる行の数を、問合せごとに指定することもできます。この数をフェッチ・サイズと呼びます。Oracle JDBC では、文オブジェクトのデフォルト・フェッチ・サイズとして行プリフェッチ値が使用されます。フェッチ・サイズを設定すると、行プリフェッチ設定が上書きされ、その文オブジェクトで実行される後続の問合せに影響を与えます。

フェッチ・サイズは、結果セットでも使用されます。文オブジェクトが問合せを実行するとき、文オブジェクトのフェッチ・サイズが、その問合せで作成される結果セット・オブジェクトに渡されます。ただし、結果セット・オブジェクトでフェッチ・サイズを設定して、渡された文フェッチ・サイズをオーバーライドすることもできます。（結果セットが作成された後で文オブジェクトのフェッチ・サイズを変更しても、結果セットには影響を与えないことにも注意してください。）

明示的に設定された場合でも、渡された文フェッチ・サイズと等しいデフォルトでも、結果セット・フェッチ・サイズによって、その結果セットでの後続のデータベースへのラウンドトリップで取り出される行の数が判断されます。このラウンドトリップには、元の問合せを完了するために必要なラウンドトリップと、結果セットへのデータの再フェッチに必要なラウンドトリップが含まれます。（scroll-sensitive または scroll-insensitive/ 更新可能な結果セットを更新するために、データは明示的または暗黙的に再フェッチされる可能性があります。11-22 ページの「行の再フェッチ」を参照してください。）

フェッチ・サイズの設定

Statement、PreparedStatement、CallableStatement および ResultSet オブジェクトのすべてで、フェッチ・サイズを設定および取り出す次のメソッドが使用可能です。

- `void setFetchSize(int rows) throws SQLException`
- `int getFetchSize() throws SQLException`

問合せのフェッチ・サイズを設定するには、問合せを実行する前に文オブジェクトの `setFetchSize()` をコールします。フェッチ・サイズを N に設定すると、1 回のデータベースへのラウンドトリップで、N 行がフェッチされます。

問合せを実行した後、結果セット・オブジェクトの `setFetchSize()` をコールすると、文オブジェクト・フェッチ・サイズによって、渡された文オブジェクト・フェッチ・サイズがオーバーライドされます。これは、データベースへの後続のラウンドトリップに影響を与え、元の問合せに対してより多くの行が取り出されます。後で実行される他の行の再フェッチにも影響を与えます。（11-22 ページの「行の再フェッチ」を参照してください。）

標準フェッチ・サイズの使用と Oracle 行プリフェッチ設定

JDBC 2.0 フェッチ・サイズの使用は、基本的に、Oracle 行プリフェッチ値の使用に似ています。ただし、行プリフェッチ値には、文オブジェクトと結果セット・オブジェクトで異なる値を使用する柔軟性がありません。常に、行プリフェッチ値が使用されます。

さらに、JDBC 2.0 フェッチ・サイズの使用方法は移植可能で、他の JDBC ドライバでも使用できます。Oracle 行プリフェッチの使用方法は、ベンダー固有です。

この Oracle 機能に関する一般的な説明は、12-19 ページの「Oracle 行プリフェッチ」を参照してください。

注意： JDBC 2.0 フェッチ・サイズ API と Oracle 行プリフェッチ API をアプリケーションで混在させることはできません。どちらも使用できますが、両方は使用できません。

行の再フェッチ

結果セットのいくつかの型で、データを再フェッチする結果セットの `refreshRow()` メソッドがサポートされています。このメソッドは、データベースに戻って、結果セットの現在行から始まる N 行に対応するデータベースの行を再度取り出します。N はフェッチ・サイズです（11-20 ページの「フェッチ・サイズ」を参照してください）。これにより、外側のトランザクションの分離レベルに依存しますが、操作中の結果セットの外で実行されたデータベースに対する最新の更新を参照できます。

再フェッチによって再度取り出されるのは、すでに結果セットにある行に対応する行のみです。元の問合せの後、データベースで挿入または削除された行に対しては、何もしません。挿入された行は無視されます。行がデータベースから削除された後も、対応する行は結果セットに残ります。データベースで削除された行を再フェッチしようとした場合、結果セットの対応する行には、元の値が保存されます。

`refreshRow()` メソッドのシグネチャを示します。

- `void refreshRow() throws SQLException`

このメソッドをコールするときは、行境界の外側や挿入行ではなく、有効な現在行に位置指定しておく必要があります。

`refreshRow()` メソッドは、次の結果セット・カテゴリでサポートされます。

- scroll-sensitive/ 読取り専用
- scroll-sensitive/ 更新可能
- scroll-insensitive/ 更新可能

将来のリリースでは、Oracle JDBC によって、その他の結果セット・カテゴリがサポートされる可能性があります。

`refreshRow()` メソッドを使用して明示的にデータを再フェッチするコード例については、20-62 ページの「結果セットでの行の再フェッチ : ResultSet6.java」を参照してください。

注意： Scroll-Sensitive 結果セット機能は、`refreshRow()` の暗黙的なコールによって実装されます。詳細は、11-27 ページの「Scroll-Sensitive 結果セットの Oracle 実装」を参照してください。

内部的および外部的に加えられたデータベース変更の参照

この項では、次の事項を参照する結果セットの機能について説明します。

- 内部変更と呼ばれる、結果セット自体の変更（結果セット内の DELETE、UPDATE または INSERT 操作）
- 外部変更と呼ばれる、結果セット以外から加えられた変更（ユーザー自身による結果セット外のトランザクションによる変更、またはその他のコミットされたトランザクションによる変更）

この項の後半で、集計表を示します。

注意： Sun 社の JDBC 2.0 の仕様では、外部変更は「他からの変更」と呼ばれています。

内部変更の参照

更新可能な結果セットの、結果セット自体の変更を参照する機能は、結果セット型と変更の種類（UPDATE、DELETE または INSERT）の両方に依存します。これについては、11-15 ページから始まる「結果セットの更新」で、様々な点から説明します。要約すると次のようになります。

- 内部的な DELETE 操作は、スクロール可能な結果セット（scroll-sensitive または scroll-insensitive）では参照できますが、forward-only 結果セットでは参照できません。

  スクロール可能な結果セットの行を削除すると、前の行が新しい現在行になり、後続の行の行番号が更新されます。

- 内部的な UPDATE 操作は、結果セット型（forward-only、scroll-sensitive または scroll-insensitive）にかかわらず、常に参照できます。
- 内部的な INSERT 操作は、結果セット型（forward-only、scroll-sensitive または scroll-insensitive）にかかわらず、常に参照できません。

内部変更が「可視」であるとは、本質的には、同じデータ項目の、前の updateXXX() コールによって変更されたデータを、後続の getXXX() コールが参照できるという意味です。

JDBC 2.0 の DatabaseMetaData オブジェクトには、これを検証する次のメソッドが含まれています。それぞれ、入力として結果セット型を取ります（ResultSet.TYPE_FORWARD_ONLY、ResultSet.TYPE_SCROLL_SENSITIVE または ResultSet.TYPE_SCROLL_INSENSITIVE）。

- `boolean ownDeletesAreVisible(int) throws SQLException`
- `boolean ownUpdatesAreVisible(int) throws SQLException`
- `boolean ownInsertsAreVisible(int) throws SQLException`

注意： 実行のトリガーの原因になる内部変更を行った場合、トリガーによる変更は外部変更の効果を持ちます。ただし、トリガーが更新中の行のデータに影響を与える場合、スクロール可能または更新可能な結果セットでは、これらの変更は参照されます。更新の後、暗黙的な行の再フェッチが発生するためです。

外部変更の参照

基礎となるデータベースに対する外部変更は、scroll-sensitive 結果セットでのみ参照でき、参照の対象は、外部的な UPDATE 操作による変更のみです。外部的な DELETE または INSERT 操作による変更は、参照できません。

注意： 外部からの変更の参照に関する説明では、外側のトランザクションには、変更を参照できるように、トランザクション自体の分離レベルが設定されているものとします。

外部更新の参照方法および参照できるまでの時間など、scroll-sensitive 結果セットの実装の詳細は、11-27 ページの「Scroll-Sensitive 結果セットの Oracle 実装」を参照してください。

JDBC 2.0 の DatabaseMetaData オブジェクトには、これを検証する次のメソッドが含まれています。それぞれ、入力として結果セット型を取ります（ResultSet.TYPE_FORWARD_ONLY、ResultSet.TYPE_SCROLL_SENSITIVE または ResultSet.TYPE_SCROLL_INSENSITIVE）。

- `boolean othersDeletesAreVisible(int) throws SQLException`
- `boolean othersUpdatesAreVisible(int) throws SQLException`
- `boolean othersInsertsAreVisible(int) throws SQLException`

注意： 11-22 ページの「行の再フェッチ」で説明した refreshRow() メソッドの明示的な使用は、この可視性の説明とは別です。たとえば、外部更新が scroll-insensitive 結果セットにおいても「不可視」であったとしても、scroll-insensitive/ 更新可能な結果セットで行を明示的に再フェッチすることで、実行された外部変更を取り出すことができます。ここでいう「可視性」とは、scroll-insensitive/ 更新可能な結果セットでは、このような変更を自動的に、また、暗黙的には参照しないという文脈においてのみ使用しています。

外部変更の可視性と検出

外部ソースによって基礎となるデータベースに加えられる変更に関して、2 つの概念があります。これらは似ていますが、ローカルな結果セットからの変更の可視性という観点から見ると異なります。

- 変更の可視性
- 変更の検出

変更が「可視」であるとは、結果セットの行を見たとき、外部ソースによってデータベースの対応する行に加えられた変更から得られる新しいデータ値を参照できるという意味です。

一方、変更が検出されるとは、結果セットが最初に移入された後の新しい値であることを、結果セットが認識するという意味です。

Oracle 結果セットが新しいデータを参照したときでも（scroll-sensitive 結果セットでの外部 UPDATE のように）、このデータが結果セットが移入された後で変更されたことは認識されません。このような変更は、検出されません。

JDBC 2.0 の DatabaseMetaData オブジェクトには、これを検証する次のメソッドが含まれています。それぞれ、入力として結果セット型を取ります（ResultSet.TYPE_FORWARD_ONLY、ResultSet.TYPE_SCROLL_SENSITIVE または ResultSet.TYPE_SCROLL_INSENSITIVE）。

- `boolean deletesAreDetected(int) throws SQLException`
- `boolean updatesAreDetected(int) throws SQLException`
- `boolean insertsAreDetected(int) throws SQLException`

したがって、JDBC 2.0 で指定されている、変更を検出する結果セット・メソッド（rowDeleted()、rowUpdated() および rowInserted()）は、Oracle JDBC ドライバでは、常に FALSE を戻します。これらをコールする意味はありません。

内部変更および外部変更の可視性のサマリー

表 11-1 で、Oracle JDBC 実装の結果セット・オブジェクトが、結果セット自体が内部的に実行した変更、および基礎となるデータベースに対してユーザー自身のトランザクションまたはその他のコミットされたトランザクションから外部的に実行された変更を参照できるかどうかに関する、前の項の説明を要約します。

表 11-1 Oracle JDBC での内部変更および外部変更の可視性

結果セット型	内部的な DELETE の参照が可能か	内部的な UPDATE の参照が可能か	内部的な INSERT の参照が可能か	外部的な DELETE の参照が可能か	外部的な UPDATE の参照が可能か	外部的な INSERT の参照が可能か
forward-only	不可	可	不可	不可	不可	不可
scroll-sensitive	可	可	不可	不可	可	不可
scroll-insensitive	可	可	不可	不可	不可	不可

外部更新の参照方法および参照できるまでの時間など、scroll-sensitive 結果セットの実装の詳細は、11-27 ページの「Scroll-Sensitive 結果セットの Oracle 実装」を参照してください。

注意：

- 11-22 ページの「行の再フェッチ」で説明した `refreshRow()` メソッドの明示的な使用は、外部変更の可視性の概念とは別です。詳細は、11-24 ページの「外部変更の参照」を参照してください。
- scroll-sensitive 結果セットを基礎とする `UPDATE` 操作のように、外部変更が「可視」であるときでも、これらは検出されません。結果セットの `rowDeleted()`、`rowUpdated()` および `rowInserted()` メソッドは、常に `FALSE` を戻します。詳細は、11-25 ページの「外部変更の可視性と検出」を参照してください。

Scroll-Sensitive 結果セットの Oracle 実装

scroll-sensitive 結果セットの Oracle 実装には、ウィンドウの概念が関係します。フェッチ・サイズの基準は、ウィンドウ・サイズです。ウィンドウ・サイズが、結果セットの行を更新する頻度に影響を与えます。

指定された行に移動して現在行を確立すると（11-11 ページの「スクロール可能な結果セットの位置指定」を参照）、ウィンドウには、結果セットの現在行から始まる N 行が含まれます。N は、結果セットで使用されているフェッチ・サイズです（11-20 ページの「フェッチ・サイズ」を参照してください）。結果セットが最初に作成されたときには、現在行がないので、ウィンドウもありません。デフォルトの位置は最初の行の前で、有効な現在行ではありません。

行を移動するとき、現在行がウィンドウ内にある間は、ウィンドウは変更されません。しかし、ウィンドウの外にある新しい現在行に移動すると、新しい現在行から始まる N 行のウィンドウが再定義されます。

ウィンドウが再定義されると、`refreshRow()` メソッドが暗黙的にコールされ、新しいウィンドウの行に対応するデータベースの N 行が自動的に再フェッチされます（11-22 ページの「行の再フェッチ」を参照してください）。これにより、新しいウィンドウのデータが更新されます。

そのため、外部更新は、再フェッチ結果セットですぐには参照できません。前述のように、自動再フェッチの後で参照できるようになります。

scroll-sensitive 結果セットの機能を使用したサンプル・アプリケーションについては、20-59 ページの「Scroll-Sensitive 結果セット : ResultSet5.java」を参照してください。

注意： この種類の再フェッチは効率が高くなく、最適化された方法でもないので、重大なパフォーマンスの問題になります。現在実装されているような scroll-sensitive 結果セットを使用する前に、十分検討してください。同期性とパフォーマンスの間にも、重大なトレードオフがあります。ほとんどの同期性のある結果セットは、フェッチ・サイズが 1 です。この場合、行を移動するたびに、新しい現在行が再フェッチされます。これは、アプリケーションのパフォーマンスに、重大な影響を与えます。

結果セット拡張用新規メソッドのサマリー

この項では、JDBC 2.0 結果セット拡張用に追加された、接続、結果セット、文およびデータベース・メタデータの新しいメソッドすべてのサマリーを示します。これらのメソッドは、この章で詳しく説明されています。

変更された接続メソッド

文オブジェクトを作成するときに結果セットと並行性の型を指定できるように変更された接続メソッドの、アルファベット順サマリーを次に示します。

- `Statement createStatement (int resultSetType, int resultSetConcurrency)`

  このメソッドにより、一般的な `Statement` オブジェクトを作成するときに、結果セット型および並行性型を指定できます。

- `CallableStatement prepareCall (String sql, int resultSetType, int resultSetConcurrency)`

  このメソッドにより、`PreparedStatement` オブジェクトを作成するときに、結果セット型および並行性型を指定できます。

- `PreparedStatement prepareStatement (String sql, int resultSetType, int resultSetConcurrency)`

  このメソッドにより、`CallableStatement` オブジェクトを作成するときに、結果セット型および並行性型を指定できます。

新しい結果セット・メソッド

JDBC 2.0 結果セット拡張用の新しい結果セット・メソッドのアルファベット順サマリーを次に示します。

- `boolean absolute(int row) throws SQLException`

  結果セットの絶対行位置に移動します。

- `void afterLast() throws SQLException`

  結果セットの最後の行の後ろに移動します（このコールの後、有効な現在行はありません）。

- `void beforeFirst() throws SQLException`

  結果セットの最初の行の前に移動します（このコールの後、有効な現在行はありません）。

- `void cancelRowUpdates() throws SQLException`

  現在行の UPDATE 操作を取り消します（updateXXX() をコールした後、updateRow() をコールする前にコールします）。

- `void deleteRow() throws SQLException`

  現在行を削除します。

- `boolean first() throws SQLException`

  結果セットの最初の行に移動します。

- `int getConcurrency() throws SQLException`

  問合せに対して使用された並行性型の int 値を戻します（ResultSet.CONCUR_READ_ONLY または ResultSet.CONCUR_UPDATABLE）。

- `int getFetchSize() throws SQLException`

  フェッチ・サイズをチェックして、1 回のデータベース・ラウンドトリップでフェッチする行の数を判断します（文オブジェクトでも使用可能）。

- `int getRow() throws SQLException`

  現在行の行番号を戻します。有効な現在行がない場合、0（ゼロ）を戻します。

- `int getType() throws SQLException`

  問合せに対して使用された結果セット型の int 値を戻します（ResultSet.TYPE_FORWARD_ONLY、ResultSet.TYPE_SCROLL_SENSITIVE または ResultSet.TYPE_SCROLL_INSENSITIVE）。

- `void insertRow() throws SQLException`

  結果セットの INSERT 操作をデータベースに書き込みます。updateXXX() メソッドをコールしてデータ値を設定した後、このメソッドをコールします。

- `boolean isAfterLast() throws SQLException`

  位置が最後の行の後ろであれば、TRUE を戻します。

- `boolean isBeforeFirst() throws SQLException`

  位置が最初の行の前であれば、TRUE を戻します。

- `boolean isFirst() throws SQLException`

  位置が最初の行であれば、TRUE を戻します。

- `boolean isLast() throws SQLException`

  位置が最後の行であれば、TRUE を戻します。

- `boolean last() throws SQLException`

  結果セットの最後の行に移動します。

- `void moveToCurrentRow() throws SQLException`

  挿入行ステージング領域から、`moveToInsertRow()` をコールする前に現在行だった位置に戻ります。

- `void moveToInsertRow() throws SQLException`

  挿入行ステージング領域に移動して、挿入する行をセットアップします。

- `boolean next() throws SQLException`

  結果セットを前方に反復します。

- `boolean previous() throws SQLException`

  結果セットを後方に反復します。

- `void refreshRow() throws SQLException`

  結果セットのカレント・ウィンドウに対応するデータベース行を再フェッチして、データを更新します。このメソッドは、scroll-sensitive 結果セットでは暗黙的にコールされます。

- `boolean relative(int row) throws SQLException`

  現在行から前方または後方に、相対行位置を移動します。

- `void setFetchSize(int rows) throws SQLException`

  再フェッチするとき、1 回のデータベース・ラウンドトリップでフェッチする行の数を判断するための、フェッチ・サイズを設定します（文オブジェクトでも使用可能）。

- `void updateRow() throws SQLException`

  `updateXXX()` メソッドを使用してデータ値を更新した後、`UPDATE` 操作をデータベースに書き込みます。

- `void updateXXX() throws SQLException`

  更新または挿入する行のデータ値を設定または更新します。データ型ごとに、`updateXXX()` メソッドがあります。更新または挿入する列に対する適切な `updateXXX()` メソッドをすべてコールした後、`UPDATE` 操作の場合 `updateRow()` を、`INSERT` 操作の場合 `insertRow()` をコールします。

新しい文メソッド

JDBC 2.0 結果セット拡張用の新しい文メソッドのアルファベット順サマリーを次に示します。これらのメソッドは、一般的な文オブジェクト、プリコンパイルされた SQL 文オブジェクトおよびコール可能文オブジェクトで使用可能です。

- `int getFetchSize() throws SQLException`

  フェッチ・サイズをチェックして、問合せを実行するとき 1 回のデータベース・ラウンドトリップでフェッチする行の数を判断します（結果セット・オブジェクトでも使用可能）。

- `void setFetchSize(int rows) throws SQLException`

  問合せを実行するとき、1 回のデータベース・ラウンドトリップでフェッチする行の数を判断するための、フェッチ・サイズを設定します（結果セット・オブジェクトでも使用可能）。

- `void setResultSetCache(OracleResultSetCache cache) throws SQLException`

  スクロール可能な結果セットに、独自のクライアント側キャッシュ実装を使用します。`OracleResultSetCache` インタフェースを実装する独自クラスを作成し、`setResultSetCache()` メソッドを使用して、このクラスのインスタンスを結果セットを作成する文オブジェクトに入力します。

- `int getResultSetType() throws SQLException`

  この文オブジェクトで作成される結果セットの結果セット型をチェックします（文オブジェクトが作成されたときに指定された型）。

- `int getResultSetConcurrency() throws SQLException`

  この文オブジェクトで作成される結果セットの並行性型をチェックします（文オブジェクトが作成されたときに指定された型）。

新しいデータベース・メタデータ・メソッド

JDBC 2.0 結果セット拡張用の新しいデータベース・メタデータ・メソッドのアルファベット順サマリーを次に示します。

- `boolean ownDeletesAreVisible(int) throws SQLException`

  この JDBC 実装で、指定された結果セット型が、結果セット自体の内部的な DELETE 操作の効果を参照できる場合、TRUE を戻します。

- `boolean ownUpdatesAreVisible(int) throws SQLException`

  この JDBC 実装で、指定された結果セット型が、結果セット自体の内部的な UPDATE 操作の効果を参照できる場合、TRUE を戻します。

- `boolean ownInsertsAreVisible(int) throws SQLException`

  この JDBC 実装で、指定された結果セット型が、結果セット自体の内部的な INSERT 操作の効果を参照できる場合、TRUE を戻します。

- `boolean othersDeletesAreVisible(int) throws SQLException`

  この JDBC 実装で、指定された結果セット型が、データベースの外部的な DELETE 操作の効果を参照できる場合、TRUE を戻します。

- `boolean othersUpdatesAreVisible(int) throws SQLException`

  この JDBC 実装で、指定された結果セット型が、データベースの外部的な UPDATE 操作の効果を参照できる場合、TRUE を戻します。

- `boolean othersInsertsAreVisible(int) throws SQLException`

  この JDBC 実装で、指定された結果セット型が、データベースの外部的な INSERT 操作の効果を参照できる場合、TRUE を戻します。

- `boolean deletesAreDetected(int) throws SQLException`

  この JDBC 実装で、指定された結果セット型が、データベースで発生した外部的な DELETE 操作を検出できる場合、TRUE を戻します。このメソッドでは常に FALSE が戻されます。

- `boolean updatesAreDetected(int) throws SQLException`

  この JDBC 実装で、指定された結果セット型が、データベースで発生した外部的な UPDATE 操作を検出できる場合、TRUE を戻します。このメソッドでは常に FALSE が戻されます。

- `boolean insertsAreDetected(int) throws SQLException`

  この JDBC 実装で、指定された結果セット型が、データベースで発生した外部的な INSERT 操作を検出できる場合、TRUE を戻します。このメソッドでは常に FALSE が戻されます。

12

パフォーマンス拡張要素

この章では、JDBC 標準に対する Oracle パフォーマンス拡張要素について説明します。

バッチ更新の説明では、JDBC 2.0 で提供される標準バッチ更新モデルについても説明します。

次の項目が含まれます。

- バッチ更新
- その他の Oracle パフォーマンス拡張要素

注意： Oracle 拡張機能の一般的な概要と、Oracle パッケージおよび型拡張要素の詳細は、第 5 章「Oracle 拡張機能の概要」を参照してください。

バッチ更新

複数の UPDATE、DELETE または INSERT 文を単一のバッチにグループ化し、バッチ全体をデータベースに送信して 1 回のラウンドトリップで処理することにより、データベースへのラウンドトリップの回数を削減できます。その結果、アプリケーションのパフォーマンスが向上します。このマニュアルではこれをバッチ更新（update batching）と呼び、Sun 社の JDBC 2.0 仕様ではバッチ更新（batch updates）と呼びます。

これは、特にプリコンパイルされた SQL 文で、同じ文をバインド変数を変えて繰り返し使用する場合に効果的です。

Oracle JDBC では、2 つの異なるバッチ更新のモデルがサポートされます。

- 標準モデル。Sun 社の JDBC 2.0 仕様を実装します。これは、標準バッチ更新と呼ばれます。
- Oracle 固有モデル。Sun 社の JDBC 2.0 仕様から独立しています。これは、Oracle バッチ更新と呼ばれます。

注意： これらのモデルは混在させることができません。単一アプリケーションで、どちらの構文を使用してもかまいませんが、両方は使用できません。これらの構文を混在させると、Oracle JDBC ドライバで例外が発生します。

バッチ更新モデルの概要

この項では、一般的なモデルとサポートされる文の種類を、標準バッチ更新と Oracle バッチ更新とで比較します。

Oracle モデルと標準モデル

Oracle バッチ更新では、バッチ値を使用することにより、通常、暗黙的なバッチの処理が発生します。バッチ値とは、データベースへのラウンドトリップを発生させるまでにバッチする（蓄積する）操作の数です。この値まで操作がバッチに追加されると、バッチが実行されます。次のことに注意してください。

- 接続オブジェクトにデフォルト・バッチを設定できます。このバッチは、その接続で実行されるプリコンパイルされた SQL 文すべてに適用されます。
- プリコンパイルされた SQL 文オブジェクトには、個別に文バッチ値を設定できます。この値は、接続バッチ値より優先します。
- 接続バッチ値と文バッチ値の両方を無視して、いつでも明示的にバッチを実行できます。

標準のバッチ更新は、手動の明示的なモデルです。バッチ値は設定されていません。手動で操作をバッチに追加し、明示的にバッチの実行を指定します。

Oracle バッチ更新のほうが効率的なモデルです。Oracle バッチ更新では、ドライバが、バッチされる操作の数をあらかじめ知ることができます。この意味で、Oracle モデルのほうが静的で予測可能です。標準モデルでは、ドライバがバッチされる操作の数をあらかじめ知ることはできません。この意味で、標準モデルのほうが本質的に動的です。

バッチ更新を使用する場合、次の方法で 2 つのモデルから 1 つを選択します。

- 移植性が重要でない場合は、Oracle バッチ更新を使用します。これにより、最高のパフォーマンス向上が得られます。
- パフォーマンスよりも移植性の優先順位が高い場合は、標準バッチ更新を使用します。

サポートされている文の型

Oracle による実装のとおり、バッチ更新は、同じ文をバインド変数を変えて繰り返し使用する場合に、プリコンパイルされた SQL 文で使用するように用意されています。次のことに注意してください。

- Oracle バッチ更新でサポートされるのは、プリコンパイルされた SQL 文オブジェクトのみです。Oracle のコール可能文では、接続デフォルト・バッチ値と文バッチ値の両方が値 1 に上書きされます。Oracle の一般的な文では、文バッチ値はなく、接続デフォルト・バッチ値は値 1 にオーバーライドされます。

  Oracle バッチ更新はベンダー固有なので、実際には一般的な `PreparedStatement` オブジェクトではなく、`OraclePreparedStatement` オブジェクトを使用（または、このオブジェクトにキャスト）する必要があることに注意してください。

- JDBC 2.0 標準に準拠するために、標準バッチ更新の Oracle の実装では、プリコンパイルされた SQL 文と同じように、コール可能文および一般的な文もサポートされています。標準バッチ更新構文は、Oracle JDBC アプリケーションに簡単に移行できます。

- バッチできるのは、`UPDATE`、`INSERT` または `DELETE` 操作のみです。結果セットを戻そうとする操作を含むバッチを実行すると、例外が発生します。

  注意： 標準バッチ更新の Oracle 実装では、一般的な文およびコール可能文の実際のバッチ処理は実装されていません。Oracle JDBC は `Statement` および `CallableStatement` オブジェクトに対する標準バッチ処理構文の使用をサポートしますが、パフォーマンスが向上するのは `PreparedStatement` オブジェクトの場合のみです。

  標準バッチ更新は、標準の `PreparedStatement`、`CallableStatement` および `Statement` オブジェクトと、Oracle 固有の `OraclePreparedStatement`、`OracleCallableStatement` および `OracleStatement` オブジェクトのどれでも使用できます。

Oracle バッチ更新

Oracle バッチ更新によって、バッチ値（制限）がプリコンパイルされた SQL 文オブジェクトに対応付けられます。Oracle バッチ更新を使用すると、JDBC ドライバは `executeUpdate()` メソッドがコールされるたびにプリコンパイルされた SQL 文を実行するのではなく、蓄積された実行要求のバッチに文を追加します。バッチ値に達すると、ドライバはすべての操作をデータベースに渡して一度に実行します。たとえば、バッチ値が 10 の場合、10 の操作がバッチに蓄積されるたびにデータベースに送信され、1 回のラウンドトリップで処理されます。

`OracleConnection` クラスのメソッドにより、Oracle 接続全体のデフォルト・バッチ値を設定できます。このバッチ値は、その接続での Oracle PreparedStatement すべてに関係します。`OraclePreparedStatement` クラスのメソッドにより、特定の Oracle PreparedStatement に文バッチ値を設定できます。この値は、接続バッチ値より優先します。手動で保留バッチを実行することにより、両方のバッチ値をオーバーライドすることもできます。

注意：

- 標準更新バッチ構文と Oracle 更新バッチ構文を同じアプリケーションに混在させることはできません。これらの構文を混在させると、JDBC ドライバで例外が発生します。
- どちらかのバッチ更新モデルを使用する場合、自動コミット・モードは無効化します。バッチの実行中にエラーが発生した場合、エラーの前に正常に実行された操作をコミットするかロールバックするかを選択できます。

Oracle バッチ更新の特徴と制限事項

Oracle バッチ更新に関する、次の制限事項と実装の詳細に注意してください。

- デフォルトでは、文バッチ値はなく、接続（デフォルト）バッチ値は 1 です。
- バッチ値は、5 ～ 30 にすると最も効率的です。非常に高い値を設定すると、逆効果になります。特定のアプリケーションでの有効性を検証するために、様々な値を試すことをお薦めします。
- バッチ値が有効かどうかにかかわらず、Oracle PreparedStatement のバインド変数がストリーム型である（またはストリーム型に変換される）場合、Oracle JDBC ドライバによりバッチ値は 1 に設定され、実行待ちの要求はすべてデータベースに送信され、実行されます。
- 次の条件に当てはまると、Oracle JDBC ドライバにより、Oracle PreparedStatement の `sendBatch()` メソッドが自動的に実行されます。1) 接続が、`commit()` メソッドの起動または自動コミット・モードの結果として、`COMMIT` 要求を受信した場合。2) 文が、`close()` 要求を受信した場合。3) 接続が、`close()` 要求を受信した場合。

注意： 標準バッチ更新を使用する場合、接続 COMMIT 要求、文クローズまたは接続クローズは、保留バッチに影響を与えません。Oracle バッチ更新を使用する場合にのみ、影響を与えます。

接続バッチ値の設定

Oracle 接続では、任意の Oracle PreparedStatement にデフォルト・バッチ値を指定できます。この値を指定するには、OracleConnection オブジェクトの setDefaultExecuteBatch() メソッドを使用します。たとえば、次のコードにより、conn 接続オブジェクトに対応付けられているすべてのプリコンパイルされた SQL 文に対して、デフォルト・バッチ値が 20 に設定されます。

```
((OracleConnection)conn).setDefaultExecuteBatch(20);
```

このメソッドにより、その接続のプリコンパイルされた SQL 文すべてに対してデフォルト・バッチ値が設定されますが、Oracle PreparedStatement で個別に setDefaultBatch() をコールすると、その設定値をオーバーライドできます。

接続バッチ値は、このバッチ値がセットされた後で作成された文オブジェクトに適用されます。

Java Properties オブジェクトを使用して接続を確立した場合、setDefaultExecuteBatch() をコールするかわりに、defaultBatchValue Java プロパティを設定することもできます。詳細は、3-6 ページの「データベース URL とプロパティ・オブジェクトの指定」を参照してください。

文バッチ値の設定

特定の Oracle PreparedStatement に文バッチ値を設定するには、次の手順を使用します。これにより、文を実行する接続の OracleConnection インスタンスの setDefaultExecuteBatch() メソッドを使用して設定した接続バッチ値はすべて、オーバーライドされます。

1. プリコンパイルされた SQL 文を記述し、第 1 行の入力値を指定します。

```
PreparedStatement ps = conn.prepareStatement
                              ("INSERT INTO dept VALUES (?,?,?)");
ps.setInt (1,12);
ps.setString (2,"Oracle");
ps.setString (3,"USA");
```

2. プリコンパイルされた SQL 文を OraclePreparedStatement オブジェクトにキャストし、setExecuteBatch() メソッドを適用します。この例では、文のバッチ・サイズは 2 に設定されます。

```
((OraclePreparedStatement)ps).setExecuteBatch(2);
```

必要に応じて、プログラムの任意の場所に getExecuteBatch() メソッドを挿入すると、文のデフォルト・バッチ値を確認できます。

```
System.out.println (" Statement Execute Batch Value " +
                    ((OraclePreparedStatement)ps).getExecuteBatch());
```

3. この時点で更新を実行するコールをデータベースに送信しても、データはデータベースに送信されず、0（ゼロ）が戻されます。

```
// No data is sent to the database by this call to executeUpdate
System.out.println ("Number of rows updated so far: "
                                  + ps.executeUpdate ());
```

4. 第 2 行の入力値の集合および更新の実行を入力すると、executeUpdate() のバッチ・コール回数がバッチ値 2 と等しくなります。データはデータベースに送信され、1 回のラウンドトリップで 2 つの行が挿入されます。

```
ps.setInt (1, 11);
ps.setString (2, "Applications");
ps.setString (3, "Indonesia");

int rows = ps.executeUpdate ();
System.out.println ("Number of rows updated now: " + rows);

ps.close ();
```

バッチ値のチェック

Oracle 接続インスタンスの全体的な接続バッチ値をチェックするには、OracleConnection クラスの getDefaultExecuteBatch() メソッドを使用します。

```
Integer batch_val = ((OracleConnection)conn).getDefaultExecuteBatch();
```

Oracle PreparedStatement の特定の文バッチ値をチェックするには、OraclePreparedStatement クラスの getExecuteBatch() メソッドを使用します。

```
Integer batch_val = ((OraclePreparedStatement)ps).getExecuteBatch();
```

注意： 文バッチ値が設定されていない場合、getExecuteBatch() は接続バッチ値を戻します。

バッチ値のオーバーライド

有効なバッチ値に達する前に、蓄積された操作を実行するには、OraclePreparedStatement オブジェクトの sendBatch() メソッドを使用します。

この例では、接続バッチ値を 20 に設定します。（これにより、接続に対応付けられているすべてのプリコンパイルされた SQL 文オブジェクトのデフォルト・バッチ値は、20 に設定されます。）次のように、接続を OracleConnection オブジェクトにキャストし、接続に setDefaultExecuteBatch() メソッドを適用します。

```
((OracleConnection)conn).setDefaultExecuteBatch (20);
```

次のように、バッチ値をオーバーライドします。

1. プリコンパイルされた SQL 文を記述し、通常の処理と同様に、第 1 行に入力値を指定し、文を実行します。

   ```
   PreparedStatement ps =
      conn.prepareStatement ("insert into dept values (?, ?, ?)");

   ps.setInt (1, 32);
   ps.setString (2, "Oracle");
   ps.setString (3, "USA");

   System.out.println (ps.executeUpdate ());
   ```

   この時点では、バッチは実行されません。ps.executeUpdate() メソッドにより "0" が戻されます。

2. 2 番目の操作の入力値の集合を入力し、もう一度 executeUpdate() をコールしても、データはデータベースに送信されません。文に対して有効なバッチ値は、接続のバッチ値である 20 です。

   ```
   ps.setInt (1, 33);
   ps.setString (2, "Applications");
   ps.setString (3, "Indonesia");

   // this batch is still not executed at this point
   int rows = ps.executeUpdate ();

   System.out.println ("Number of rows updated before calling sendBatch: "
                           + rows);
   ```

   println 文の rows の値が "0" である点に注意してください。

3. この時点で sendBatch() を適用すると、これまでにバッチされていた 2 つの操作が、1 回のラウンドトリップでデータベースに送信されます。sendBatch() メソッドからは、更新された行の合計数も戻されます。sendBatch() のプロパティを println で使用すると、更新された行数が出力されます。

```
// Execution of both previously batched executes will happen
// at this point. The number of rows updated will be
// returned by sendBatch.
rows = ((OraclePreparedStatement)ps).sendBatch ();

System.out.println ("Number of rows updated by calling sendBatch: "
                          + rows);
ps.close ();
```

Oracle バッチ処理による変更のコミット

バッチを実行した後、変更をコミットする必要があります。自動コミットは使用禁止（推奨）にしてあるものとします。

Oracle バッチ処理で接続オブジェクトの commit() をコールすると、実行されたバッチの操作がコミットされるのみでなく、暗黙的な sendBatch() コールが発行され、保留バッチがすべて実行されます。このように、commit() によって、バッチに追加されたすべての操作による変更が効率的にコミットされます。

Oracle バッチ処理による更新件数

バッチを使用しない場合、OraclePreparedStatement オブジェクトの executeUpdate() メソッドからは、操作の影響を受けたデータベース行の数が戻されます。

Oracle バッチ処理を使用する場合、このメソッドからは、次のように、メソッドが起動されたときに影響を受けた行の数が返されます。

- executeUpdate() のコールによって操作がバッチに追加される場合、値 0（ゼロ）が戻されます。その時点では、データベースには何も書き込まれません。
- executeUpdate() のコールによってバッチ値に達し、バッチが実行される場合、バッチに含まれるすべての操作によって影響を受ける行の合計数が戻されます。

同様に、OraclePreparedStatement オブジェクトの sendBatch() メソッドからは、バッチに含まれるすべての操作によって影響を受ける行の合計数が戻されます。

例 : Oracle バッチ更新

次の例では、Oracle バッチ更新機能の使用方法を示します。この例では、oracle.driver.* インタフェースがインポート済みであることを前提にしています。

```
...
Connection conn =
  DriverManager.getConnection("jdbc:oracle:oci8:","scott","tiger");

conn.setAutoCommit(false);

PreparedStatement ps =
  conn.prepareStatement("insert into dept values (?, ?, ?)");

//Change batch size for this statement to 3
((OraclePreparedStatement)ps).setExecuteBatch (3);

ps.setInt(1, 23);
ps.setString(2, "Sales");
ps.setString(3, "USA");
ps.executeUpdate(); //JDBC queues this for later execution

ps.setInt(1, 24);
ps.setString(2, "Blue Sky");
ps.setString(3, "Montana");
ps.executeUpdate(); //JDBC queues this for later execution

ps.setInt(1, 25);
ps.setString(2, "Applications");
ps.setString(3, "India");
ps.executeUpdate(); //The queue size equals the batch value of 3
                    //JDBC sends the requests to the database

ps.setInt(1, 26);
ps.setString(2, "HR");
ps.setString(3, "Mongolia");
ps.executeUpdate(); //JDBC queues this for later execution

((OraclePreparedStatement)ps).sendBatch(); // JDBC sends the queued request
conn.commit();

ps.close();
...
```

暗黙的および明示的なバッチ実行方法を含む Oracle バッチ更新の完全なサンプル・アプリケーションについては、20-68 ページの「暗黙的に実行される Oracle バッチ更新 : SetExecuteBatch.java」および 20-70 ページの「明示的に実行される Oracle バッチ更新 : SendBatch.java」を参照してください。

注意： バッチによって遅延した更新は、他の問合せの結果に影響を与えることがあります。次の例では、バッチにより最初の問合せが遅延した場合は、次の問合せから予期しない結果が戻されます。

```
UPDATE emp SET name = "Sue" WHERE name = "Bob";
SELECT name FROM emp WHERE name = "Sue";
```

標準バッチ更新

Sun 社の JDBC 2.0 仕様に従った標準バッチ更新モデルが実装されています。これは JDBC 2.0 の機能なので、JDK 1.2.x 環境で使用するように用意されています。標準バッチ更新を JDK 1.1.x 環境で使用するには、Oracle 文オブジェクトにキャストする必要があります。

このモデルは、Oracle バッチ更新モデルと異なり、`addBatch()` メソッドを使用して明示的に文をバッチに追加し、`executeBatch()` メソッドを使用して明示的にバッチを処理します。（Oracle モデルでは、バッチ処理を行わない場合と同様に `executeUpdate()` を起動しますが、操作をバッチに追加するか、バッチ全体を実行するかは、通常、あらかじめ定義されているバッチ値に達したかどうかによって、暗黙的に判断されます。）

注意：

- 標準更新バッチ構文と Oracle 更新バッチ構文を同じアプリケーションに混在させることはできません。これらの構文を混在させると、Oracle JDBC ドライバで例外が発生します。
- どちらかのバッチ更新モデルを使用する場合、自動コミット・モードは無効化します。バッチの実行中にエラーが発生した場合、エラーの前に正常に実行された操作をコミットするかロールバックするかを選択できます。

標準バッチ処理の Oracle 実装の制限事項

標準バッチ更新の Oracle の実装に関する、次の制限事項と実装の詳細に注意してください。

- Oracle JDBC アプリケーションでバッチ更新を使用すると、バインド変数の設定を変えてプリコンパイルされた SQL 文を繰り返し使用できます。

  標準バッチ更新の Oracle 実装では、一般的な文およびコール可能文の実際のバッチ処理は実装されていません。Oracle JDBC は `Statement` および `CallableStatement` オブジェクトに対する標準バッチ処理構文の使用をサポートしますが、パフォーマンスは向上しません。

- 標準バッチ更新の Oracle の実装では、バインド値としてストリーム型はサポートされません。（Oracle バッチ更新でもサポートされません。）ストリーム型を使用しようとすると、例外が発生します。

バッチへの操作の追加

文オブジェクトが最初に作成されるとき、文バッチは空です。標準 addBatch() メソッドを使用して、操作を文バッチに追加します。このメソッドは、標準 java.sql.Statement、PreparedStatement および CallableStatement インタフェースで指定されています。これらのインタフェースはそれぞれ、インタフェース oracle.jdbc.OracleStatement、OraclePreparedStatement および OracleCallableStatement によって実装されています。

Statement オブジェクト（または OracleStatement）の場合、addBatch() メソッドは、入力として SQL 操作の Java 文字列を取ります。次に、例を示します（Connection インスタンス conn があるとします）。

```
...
Statement stmt = conn.createStatement();

stmt.addBatch("INSERT INTO emp VALUES(1000, 'Joe Jones')");
stmt.addBatch("INSERT INTO dept VALUES(260, 'Sales')");
stmt.addBatch("INSERT INTO emp_dept VALUES(1000, 260)");
...
```

この時点で、バッチには 3 つの操作が入っています。

（ただし、標準バッチ更新の Oracle 実装では、一般的な文をバッチ処理してもパフォーマンスは向上しません。）

プリコンパイルされた SQL 文の場合、バッチ更新を使用して、バインド・パラメータの設定が異なる同じ文の複数の実行をバッチ処理します。PreparedStatement または OraclePreparedStatement オブジェクトの場合、addBatch() メソッドは入力を取りません。適切な setXXX() メソッドで最後に設定されたバインド・パラメータを使用して、操作をバッチに追加するのみです。（CallableStatement または OracleCallableStatement オブジェクトでも同じです。ただし、標準バッチ更新の Oracle 実装では、コール可能文をバッチ処理してもパフォーマンスは向上しません。）

次に、例を示します（この例でも、Connection インスタンス conn があるとします）。

```
...
PreparedStatement pstmt =
          conn.prepareStatement("INSERT INTO employees VALUES(?, ?)");

pstmt.setInt(1, 2000);
pstmt.setString(2, "Milo Mumford");
pstmt.addBatch();

pstmt.setInt(1, 3000);
pstmt.setString(2, "Sulu Simpson");
pstmt.addBatch();
...
```

この時点で、バッチには 2 つの操作が入っています。

1 つのバッチは単一の PreparedStatement オブジェクトに対応付けられるので、バッチ処理できるのは、この例のような単一のプリコンパイルされた SQL 文の繰返し実行のみです。

バッチの実行

操作のカレント・バッチを実行するには、文オブジェクトの executeBatch() メソッドを使用します。このメソッドは、標準 Statement インタフェースで指定されています。このインタフェースは、標準 PreparedStatement および CallableStatement インタフェースによって拡張されます。

次の例では、前の例で示したプリコンパイルされた SQL 文の addBatch() コールを繰り返した後、バッチを実行します。

```
...
PreparedStatement pstmt =
         conn.prepareStatement("INSERT INTO employees VALUES(?, ?)");

pstmt.setInt(1, 2000);
pstmt.setString(2, "Milo Mumford");
pstmt.addBatch();

pstmt.setInt(1, 3000);
pstmt.setString(2, "Sulu Simpson");
pstmt.addBatch();

int[] updateCounts = pstmt.executeBatch();
...
```

executeBatch() メソッドからは、int 配列が戻されます。通常、1 つの要素がバッチ処理された 1 つの操作に対応し、バッチの実行が成功したか失敗したかを示します。影響を受けた行数に関する情報が含まれていることもあります。詳細は、12-14 ページの「標準バッチ処理の Oracle 実装の更新件数」を参照してください。

注意：

- addBatch() をコールした後、executeUpdate() をコールする前に、executeBatch() または clearBatch() をコールする必要があります。コールしないと、SQL 例外が発生します。
- バッチが実行されるとき、操作は、バッチに入れられた順に実行されます。
- 文のバッチは、executeBatch() が戻ると、空にリセットされます。
- 文オブジェクトのカレント結果セットがある場合、この結果セットは executeBatch() コールによってクローズされます。

標準バッチ処理の Oracle 実装による変更のコミット

バッチを実行した後、変更をコミットする必要があります。推奨されているように、自動コミットは使用禁止にしてあるものとします。

commit() をコールすると、バッチ処理されていない操作がコミットされ、実行済みの文バッチのバッチ処理された操作がコミットされます。しかし、標準バッチ処理の Oracle 実装では、実行されていない保留文バッチは影響を受けません。

バッチの消去

操作のカレント・バッチを実行せずに消去するには、文オブジェクトの clearBatch() メソッドを使用します。このメソッドは、標準 Statement インタフェースで指定されています。このインタフェースは、標準 PreparedStatement および CallableStatement インタフェースによって拡張されます。

次の例では、前の例で示したプリコンパイルされた SQL 文の addBatch() コールを繰り返した後、特定の条件でバッチを消去します。

```
...
PreparedStatement pstmt =
          conn.prepareStatement("INSERT INTO employees VALUES(?, ?)");

pstmt.setInt(1, 2000);
pstmt.setString(2, "Milo Mumford");
pstmt.addBatch();

pstmt.setInt(1, 3000);
pstmt.setString(2, "Sulu Simpson");
pstmt.addBatch();

if (...condition...)
{
   int[] updateCounts = pstmt.executeBatch();
```

```
   ...
}
else
{
   pstmt.clearBatch();
   ...
}
```

注意：

- `addBatch()` をコールした後、`executeUpdate()` をコールする前に、`executeBatch()` または `clearBatch()` をコールする必要があります。コールしないと、SQL 例外が発生します。
- `clearBatch()` をコールすると、文バッチは空にリセットされます。
- `clearBatch()` メソッドの戻り値はありません。

標準バッチ処理の Oracle 実装の更新件数

文バッチが正常に実行された場合（バッチ例外が発生しない場合）、文の `executeBatch()` コールから戻される整数配列、つまり更新件数配列には、常にバッチ操作 1 つに対して 1 つの要素が含まれます。標準バッチ更新の Oracle 実装では、配列要素の値は次のようになります。

- プリコンパイルされた SQL 文のバッチの場合、バッチに含まれている個々の文によって影響を受けたデータベースの行数はわかりません。そのため、配列要素の値はすべて -2 になります。JDBC 2.0 仕様によれば、値 -2 は、操作は正常終了したが影響を受けた行数は不明であることを示します。
- 一般的な文のバッチまたはコール可能な文のバッチの場合、配列には、各操作で影響を受けた行数を示す実際の更新件数を格納します。標準バッチ更新の Oracle 実装では、Oracle JDBC は一般の文およびコール可能文の本当の意味でのバッチ処理を使用できないので、実際の更新件数がわかります。

コードの側では、バッチの正常な実行に対して、配列要素に -2 または実際の更新件数のどちらが設定されても処理できるように準備しておく必要があります。正常なバッチ実行では、配列にはすべて -2 が含まれるか、すべて正の整数が含まれます。

注意： バッチ実行が失敗したときに更新件数配列に格納される値については、12-16 ページの「標準バッチ処理の Oracle 実装のエラー処理」を参照してください。

例 : 標準バッチ更新

この例は、前の項のサンプル・コードを組み合せたもので、次の処理を行います。

- 自動コミット・モードの無効化（どちらかのバッチ更新モデルを使用する場合、無効にする必要があります）。
- PreparedStatement オブジェクトの作成。
- PreparedStatement オブジェクトに対応付けられたバッチへの操作の追加。
- バッチの実行。
- バッチの操作のコミット。

Connection インスタンス conn があるとします。

```
conn.setAutoCommit(false);

PreparedStatement pstmt =
         conn.prepareStatement("INSERT INTO employees VALUES(?, ?)");

pstmt.setInt(1, 2000);
pstmt.setString(2, "Milo Mumford");
pstmt.addBatch();

pstmt.setInt(1, 3000);
pstmt.setString(2, "Sulu Simpson");
pstmt.addBatch();

int[] updateCounts = pstmt.executeBatch();

conn.commit();

pstmt.close();
...
```

更新件数配列を処理して、バッチが正常に実行されたかどうかを判断できます。この詳細は、次の項の「標準バッチ処理の Oracle 実装のエラー処理」で説明します。

完全なサンプル・アプリケーションについては、20-66 ページの「標準バッチ更新 : BatchUpdates.java」を参照してください。

標準バッチ処理の Oracle 実装のエラー処理

executeBatch() がコールされたとき、バッチ処理された操作のうち 1 つでも失敗すると（または結果セットを戻そうとすると）、実行は停止し、java.sql.BatchUpdateException（java.sql.SQLException のサブクラス）が生成されます。

バッチ例外の後、BatchUpdateException オブジェクトの getUpdateCounts() メソッドを使用して、更新件数配列を取り出せます。このメソッドからは、executeBatch() メソッドと同じように、更新件数の int 配列が戻されます。標準バッチ更新の Oracle 実装では、バッチ例外の後の更新件数配列の内容は次のようになります。

- プリコンパイルされた SQL 文のバッチの場合、どの操作が失敗したかはわかりません。配列には、バッチの操作 1 つに対して 1 つの要素が含まれ、各要素には値 -3 が設定されます。JDBC 2.0 仕様によれば、値 -3 は、操作が正常終了しなかったことを示します。この場合、実際に失敗した操作は 1 つのみと考えられます。しかし、JDBC ドライバは、どの操作が失敗したかわからないので、バッチ処理された操作をすべて、失敗したものとしてラベル付けします。

  この場合、必ず、ROLLBACK 操作を実行する必要があります。

- 一般的な文のバッチまたはコール可能文のバッチの場合、更新件数配列は、エラーの時点までの実際の更新件数を格納する、部分的な配列になります。標準バッチ更新の Oracle 実装では、Oracle JDBC は一般の文およびコール可能文の本当の意味でのバッチ処理を使用できないので、実際の更新件数がわかります。

  たとえば、バッチに 20 の操作が含まれているとき、最初の 13 は正常終了し、14 番目で例外が生成された場合、更新件数配列には 13 の要素が含まれ、正常終了した操作の実際の更新件数が設定されます。

  この場合、正常終了した操作をコミットすることも、ロールバックすることもできます。

コードの側では、例外が発生した場合、バッチの失敗した実行に対して、配列要素に -3 または実際の更新件数のどちらが設定されても処理できるように準備しておく必要があります。失敗したバッチ実行では、すべてに -3 が含まれる完全な配列か、正の整数が含まれる部分配列が作成されます。

バッチ処理される文とバッチ処理されない文の混在

文オブジェクトに操作の保留バッチがある場合（つまり、その文オブジェクトに対応付けられているバッチが空でない場合）、通常のバッチ処理されない操作の実行を行うために executeUpdate() をコールできません。

ただし、文バッチに操作を追加する前か、バッチを実行した後で、バッチ処理されない操作を実行する場合は、バッチ処理される操作とバッチ処理されない操作を単一文オブジェクトに混在させることができます。つまり、文オブジェクトの executeUpdate() は、更新バッチが空のときにのみコールできます。バッチが空でない場合、例外が生成されます。

たとえば、次の順序は有効です。

```
...
PreparedStatement pstmt =
          conn.prepareStatement("INSERT INTO employees VALUES(?, ?)");

pstmt.setInt(1, 2000);
pstmt.setString(2, "Milo Mumford");

int scount = pstmt.executeUpdate();   // OK; no operations in pstmt batch

pstmt.setInt(1, 3000);
pstmt.setString(2, "Sulu Simpson");
pstmt.addBatch();                     // Now start a batch

pstmt.setInt(1, 4000);
pstmt.setString(2, "Stan Leland");
pstmt.addBatch();

int[] bcounts = pstmt.executeBatch();

pstmt.setInt(1, 5000);
pstmt.setString(2, "Amy Feiner");

int scount = pstmt.executeUpdate();   // OK; pstmt batch was executed
...
```

ある文オブジェクトのバッチ処理されない操作と、別の文オブジェクトのバッチ処理される操作を、コード上に混在させることはできます。異なる文オブジェクトは、バッチ更新操作に関して、それぞれ独立しています。COMMIT 要求は、バッチ処理されない操作すべてと、実行済みバッチの正常な操作すべてに影響を与えますが、保留バッチには影響を与えません。

早期バッチ・フラッシュ

早期バッチ・フラッシュは、キャッシュされたメタデータが変更されると発生します。キャッシュされたメタデータは、次のような様々な理由から変更されることがあります。

- 最初のバインドが NULL で後続のバインドが非 NULL である場合。
- 最初は文字列としてスカラー型がバインドされ、後でスカラー型としてバインドされた場合。あるいはその逆の場合。

早期バッチ・フラッシュ・カウントは、次の executeUpdate() メソッドまたは sendBatch() メソッドの戻り値に追加されます。

以前の機能では、ここで取得できるすべてのバッチ・フラッシュ値が失われました。以前の機能に切り替えるには、次に示すように AccumulateBatchResult プロパティを FALSE に設定します。

```
HashTable info = new HashTable ();
info.put ("user", "SCOTT");
info.put ("passwd", "TIGER");
// other properties
...

// property: batch flush type
info.put ("AccumulateBatchResult", "false");

Connection con = DriverManager.getConnection ("jdbc:oracle:oci8:@", info);
```

注意： Oracle9*i* では、AccumulateBatchResult はデフォルトで TRUE に設定されます。

例：

```
((OraclePreparedStatement)pstmt).setExecuteBatch (2);

pstmt.setNull (1, OracleTypes.NUMBER);
pstmt.setString (2, "test11");
int count = pstmt.executeUpdate (); // returns 0

/*
* Premature batch flush happens here.
*/
pstmt.setInt (1, 22);
pstmt.setString (2, "test22");
int count = pstmt.executeUpdate (); // returns 0

pstmt.setInt (1, 33);
pstmt.setString (2, "test33");
/*
*  returns 3 with the new batching scheme where as,
*  returns 2 with the old batching scheme.
*/
int count = pstmt.executeUpdate ();
```

その他の Oracle パフォーマンス拡張要素

前述の項までで説明したバッチ更新の他、Oracle JDBC ドライバでは、次の拡張要素がサポートされています。これにより、データベースへのラウンドトリップが減少し、パフォーマンスが向上します。

- 行のプリフェッチ

  行をプリフェッチすることにより、データがフェッチされるたびに複数行がフェッチされるので、データベースへのラウンドトリップが減少します。余分にとってきたデータは、後に使用するためにクライアント側バッファに格納されます。プリフェッチの行数は、目的に応じて設定できます。

- 列型の指定

  問合せの実行および問合せ結果の取出しを行うときに、通常の JDBC プロトコルで生じる非効率性を回避できます。

- データベース・メタデータ `TABLE_REMARKS` 列の抑止

  コストの高い外部結合操作を回避できます。

Oracle では、これらのパフォーマンス拡張要素をサポートするために、接続プロパティ・オブジェクトにいくつかの拡張要素を提供します。これらの拡張要素により、`remarksReporting` フラグと、行プリフェッチとバッチ更新のデフォルト値を設定できます。詳細は、3-6 ページの「データベース URL とプロパティ・オブジェクトの指定」を参照してください。

Oracle 行プリフェッチ

Oracle JDBC ドライバには、問合せの間結果セットが移入される際に、クライアントにプリフェッチしてくる行数を設定できるようにする拡張要素が含まれています。この機能を使用すると、サーバーへのラウンドトリップ回数を減らすことができます。

注意： フェッチ・サイズをプリセットする機能は、JDBC 2.0 では標準機能になりました。この機能の標準実装については、11-20 ページの「フェッチ・サイズ」を参照してください。

Oracle プリフェッチ値の設定

標準 JDBC では、結果セットを一度に 1 行ずつしか受け取れないため、そのたびにデータベースまでのラウンドトリップが必要になります。行のプリフェッチ機能は、整数の行プリフェッチ設定を指定された文オブジェクトに対応付けます。JDBC は、問合せ時にここで設定した複数の行を一度にフェッチします。つまり、プリフェッチ設定が N のときは、JDBC は問合せ条件と一致する N 行をフェッチし、すべての行が一度でクライアントに戻されます。次に、`next()` コールが、この N 行を処理しおわると、問合せ条件に一致する次の N 行がフェッチされます。

特定の Oracle 文（文の型は任意です）に対して、プリフェッチする行数を設定できます。ある接続におけるすべての文に対してプリフェッチされるデフォルトの行数を、リセットすることもできます。クライアントにプリフェッチされるデフォルト行数は、10 です。

次の方法で、特定の文でプリフェッチする行数を設定します。

1. Statement オブジェクトが `OracleStatement`、`OraclePreparedStatement` または `OracleCallableStatement` オブジェクト以外の場合は、これらのいずれかにキャストします。
2. Statement オブジェクトの `setRowPrefetch()` メソッドを使用して、プリフェッチする行数を指定し、その値を整数として渡します。現在設定されているプリフェッチ数を確認するには、Statement オブジェクトの `getRowPrefetch()` メソッドを使用します。このメソッドでは整数が戻されます。

次の方法で、ある接続におけるすべての文に対してプリフェッチするデフォルトの行数を設定します。

1. `Connection` オブジェクトを `OracleConnection` オブジェクトにキャストします。
2. プリフェッチするデフォルト行数を設定する `OracleConnection` オブジェクトの `setDefaultRowPrefetch()` メソッドを使用し、デフォルトに設定する整数を渡します。現在設定されているデフォルト値を確認する場合は、`OracleConnection` オブジェクトの `getDefaultRowPrefetch()` メソッドを使用します。このメソッドでは整数が戻されます。

   Java `Properties` オブジェクトを使用して接続を確立した場合、`setDefaultRowPrefetch()` をコールするかわりに、`defaultRowPrefetch` Java プロパティを設定することもできます。詳細は、3-6 ページの「データベース URL とプロパティ・オブジェクトの指定」を参照してください。

注意：

- JDBC 2.0 フェッチ・サイズ API と Oracle 行プリフェッチ API をアプリケーションで混在させることはできません。どちらも使用できますが、両方は使用できません。
- Oracle 行プリフェッチ値を設定すると、問合せ以外に、次の項目にも影響を与えることがあります。
  1) JDBC 2.0 で使用可能な結果セットの `refreshRow()` メソッドによる結果セットの行の明示的な再フェッチ（scroll-sensitive/ 読取り専用、scroll-sensitive/ 更新可能および scroll-insensitive/ 更新可能結果セットに関係します）行数。
  2) scroll-sensitive 結果セットのウィンドウ・サイズ。自動再フェッチの実行頻度に影響します。ただし、Oracle 行プリフェッチ値は、フェッチ・サイズの設定によってオーバーライドされます。詳細は、11-20 ページの「フェッチ・サイズ」を参照してください。

例 : 行のプリフェッチ 行のプリフェッチ機能の使用例を示します。この例では、oracle.jdbc.* インタフェースがインポート済みであることを前提にしています。

```
Connection conn =
       DriverManager.getConnection("jdbc:oracle:oci8:","scott","tiger");

//Set the default row-prefetch setting for this connection
((OracleConnection)conn).setDefaultRowPrefetch(7);

/* The following statement gets the default row-prefetch value for
   the connection, that is, 7.
 */
Statement stmt = conn.createStatement();

/* Subsequent statements look the same, regardless of the row
   prefetch value. Only execution time changes.
 */
ResultSet rset = stmt.executeQuery("SELECT ename FROM emp");
System.out.println( rset.next () );

while( rset.next () )
    System.out.println( rset.getString (1) );

//Override the default row-prefetch setting for this statement
( (OracleStatement)stmt ).setRowPrefetch (2);

ResultSet rset = stmt.executeQuery("SELECT ename FROM emp");
System.out.println( rset.next () );

while( rset.next() )
   System.out.println( rset.getString (1) );

stmt.close();
```

接続デフォルト行プリフェッチ値と文行プリフェッチ値の設定方法を含む完全なサンプル・アプリケーションについては、20-71 ページの「接続で指定する Oracle 行プリフェッチ : RowPrefetch_connection.java」および 20-73 ページの「文で指定する Oracle 行プリフェッチ : RowPrefetch_statement.java」を参照してください。

Oracle 行プリフェッチの制限事項

プリフェッチの設定に最大値はありませんが、経験値では 10 行が効果的です。多くの場合、10 行を超える値はお薦めしません。接続のプリフェッチ行数のデフォルト値を設定しない場合は、10 がデフォルトになります。

Statement オブジェクトでは、作成時に対応付けられた接続から、行のプリフェッチ設定のデフォルト値を受け取ります。接続の行プリフェッチ設定のデフォルト値を後で変更しても、文のプリフェッチ設定は変更されません。

結果セットの列のデータ型が LONG または LONG RAW （ストリーミング型）の場合、どちらの型の値を実際に読み込まなくても、文の行プリフェッチ設定は JDBC によって 1 に変更されます。

DriverManager クラスの getConnection() メソッドの、Properties オブジェクトを引数として取るフォームを使用する場合、接続のデフォルト行プリフェッチ値はこの方法で設定できます。詳細は、3-6 ページの「データベース URL とプロパティ・オブジェクトの指定」を参照してください。

列型の定義

Oracle JDBC ドライバを使用すると、ドライバに次の問合せの列型を通知することにより、データベースへのラウンドトリップ回数を節約できます。表の定義を取得する必要がなくなるためです。

標準 JDBC から問合せを実行すると、まずデータベースへのラウンドトリップを使用して、結果セットの列に使用される型を決定します。次に、JDBC では、問合せからデータを受け取り、必要に応じて結果セットに移入するときにデータを変換します。

問合せの列型を指定するときは、データベースへの最初のラウンドトリップを行う必要がありません。サーバーによっては、必要な型変換が行われます。

完全なサンプル・アプリケーションについては、20-75 ページの「Oracle 列型定義 : DefineColumnType.java」を参照してください。

問合せの列型を定義するには、次の手順を実行します。

1. Statement オブジェクトが OracleStatement、OraclePreparedStatement または OracleCallableStatement オブジェクト以外の場合は、これらのいずれかにキャストします。
2. 必要に応じて、Statement オブジェクトの clearDefines() メソッドを使用して、この文オブジェクトの以前の列定義を消去します。
3. 必要な結果セットの各列に対して、Statement オブジェクトの defineColumnType() メソッドをコールして次のパラメータを渡します。
   - 列索引（整数）
   - タイプコード（整数）

     java.sql.Types クラスまたは oracle.jdbc.OracleTypes クラスの静的定数を使用します（Types.INTEGER、Types.FLOAT、Types.VARCHAR、OracleTypes.VARCHAR、OracleTypes.ROWID など）。この 2 つのクラスで、標準型のタイプコードは同一です。

- 型名（文字列）（構造化オブジェクト、オブジェクト参照および配列のみ）

  構造化オブジェクト、オブジェクト参照および配列の場合、型名も指定する必要があります（Employee、EmployeeRef、EmployeeArray など）。

- （オプション）最大フィールド・サイズ（整数）

  オプションで、この列の最大データ長も指定できます。

  構造化オブジェクト、オブジェクト参照または配列の列型を定義する場合、最大フィールド・サイズ・パラメータは指定できません。このパラメータを含めても、無視されます。

たとえば、stmt を Oracle 文と仮定した場合は、次の構文を使用します。

```
stmt.defineColumnType(column_index, typeCode);
```

または次のようにします（この方法は列が VARCHAR または等価な型で、長さの制限がわかっている場合にお薦めします）。

```
stmt.defineColumnType(column_index, typeCode, max_size);
```

または次のようにします（列が、構造化オブジェクト、オブジェクト参照および配列の場合）。

```
stmt.defineColumnType(column_index, typeCode, typeName);
```

デフォルトのデータ長をすべて受け取る必要がない場合は、最大フィールド・サイズを設定します。標準 JDBC Statement クラスの setMaxFieldSize() メソッドをコールして、戻されるデータ量の制限を設定します。つまり、戻されるデータのサイズは、次の値の最小値になります。

- defineColumnType() で設定された最大フィールド・サイズ

または

- setMaxFieldSize() で設定された最大フィールド・サイズ

または

- データ型固有の最大サイズ

この処理が終了した後に、文の executeQuery() メソッドを使用して問合せを実行します。

注意： 必要な結果セットのすべての列に対してデータ型を定義する必要があります。型を指定した列の数が結果セットの列の数と一致しない場合、SQL 例外が発生して処理が失敗します。

例 : 列型の定義 次の例は、この機能の使用例です。この例では、oracle.jdbc.* インタフェースがインポート済みであることを前提にしています。

```
Connection conn =
     DriverManager.getConnection("jdbc:oracle:oci8:","scott","tiger");

Statement stmt = conn.createStatement();

/*Ask for the column as a string:
 *Avoid a round trip to get the column type.
 *Convert from number to string on the server.
 */
((OracleStatement)stmt).defineColumnType(1, Types.VARCHAR);

ResultSet rset = stmt.executeQuery("select empno from emp");

while (rset.next() )
    System.out.println(rset.getString(1));

stmt.close();
```

この例に示すとおり、defineColumnType() メソッドのコール時に文（stmt）を OracleStatement 型にキャストする必要があります。接続の createStatement() メソッドにより java.sql.Statement 型のオブジェクトが戻されます。このオブジェクトには defineColumnType() および clearDefines() メソッドは設定されていません。これらのメソッドは、OracleStatement 実装以外では提供されません。

定義の拡張要素では、JDBC 型を使用して目的の型を指定します。使用可能な列型は、列の Oracle 内部型によって異なります。

すべての列は、本来の JDBC 型に定義できます。多くの場合、Types.CHAR または Types.VARCHAR タイプコードに定義できます。

表 12-1 は、defineColumnType() メソッドで使用できる有効な列定義引数のリストです。

表 12-1 有効な列型

列内の Oracle SQL 型	defineColumnType() で 定義できる型
NUMBER, VARNUM	BIGINT, TINYINT, SMALLINT, INTEGER, FLOAT, REAL, DOUBLE, NUMERIC, DECIMAL, CHAR, VARCHAR
CHAR, VARCHAR2	CHAR, VARCHAR
LONG	CHAR, VARCHAR, LONGVARCHAR
LONGRAW	LONGVARBINARY, VARBINARY, BINARY
RAW	VARBINARY, BINARY
DATE	DATE, TIME, TIMESTAMP, CHAR, VARCHAR
ROWID	ROWID

DatabaseMetaData TABLE_REMARKS レポート

データベース・メタデータ・クラスの getColumns()、getProcedureColumns()、getProcedures() および getTables() メソッドを使用して TABLE_REMARKS 列をレポートすると、コストの高い外部結合を必要とするため処理が遅くなります。このため、この JDBC ドライバは、デフォルトでは TABLE_REMARKS 列をレポートしません。

OracleConnection オブジェクトの setRemarksReporting() メソッドに true 引数を渡すと、TABLE_REMARKS レポートが使用可能になります。

Java Properties オブジェクトを使用して接続を確立した場合、setRemarksReporting() をコールするかわりに、remarksReporting Java プロパティを設定することもできます。詳細は、3-6 ページの「データベース URL とプロパティ・オブジェクトの指定」を参照してください。

標準 java.sql.Connection オブジェクトを使用している場合、setRemarksReporting() を使用するには、オブジェクトを OracleConnection にキャストする必要があります。

例 : TABLE_REMARKS レポート

conn は標準 Connection オブジェクト名を表しています。次の文を使用すると TABLE_REMARKS レポートが使用可能になります。

```
( (oracle.jdbc.OracleConnection)conn ).setRemarksReporting(true);
```

getProcedures() および getProcedureColumns() メソッドの考慮事項

JDBC バージョン 1.1 および 1.2 では、getProcedures() および getProcedureColumns() メソッドは、catalog、schemaPattern、columnNamePattern および procedureNamePattern パラメータを同じ方法で処理します。これらのメソッドに関する Oracle の定義では、パラメータの処理方法は次のように異なります。

- catalog: Oracle には複数カタログはありませんが、複数パッケージはあります。このため、catalog パラメータはパッケージ名として処理されます。これは、入力（catalog パラメータ）および出力（戻された ResultSet 内の catalog 列）の両方に適用されます。" "（空白文字列）を入力すると、パッケージなしのプロシージャおよび引数、つまり、スタンドアロン・オブジェクトが取り出されます。NULL 値は、選択条件が削除されたことを意味します。つまり、スタンドアロン・オブジェクトとパッケージ・オブジェクトの情報の両方が戻されます。("%" を渡したときと同じです。）その他の場合は、catalog パラメータはパッケージ名のパターン（必要に応じて、SQL ワイルド・カードが使用できます。）になります。
- schemaPattern: Oracle のすべてのオブジェクトは、スキーマを持ちます。したがって、スキーマを持たないオブジェクトの情報を戻すことは意味がありません。このため、入力時の " "（空白文字列）は、カレント・スキーマ（現在接続しているスキーマ）のオブジェクトと解釈されます。catalog パラメータの動作と一貫性を保つために、NULL は選択条件からスキーマを削除すると解釈されます。("%" を渡した場合と同様です。）また、SQL ワイルド・カードとともにパターンとしても使用できます。
- procedureNamePattern および columnNamePattern: 空白文字列（" "）は、どちらのパラメータに対しても意味を持ちません。すべてのプロシージャおよび引数には名前が必要なためです。このため、" " では例外が発生します。他のパラメータの処理と一貫性を保つために、NULL は、"%" を渡した場合と同じ効果を持ちます。

13

文キャッシュ

この章では、Oracle JDBC 拡張要素である文キャッシュの利点と使用方法について説明します。

次の項目が含まれます。

- 文キャッシュについて
- 文キャッシュの使用方法

文キャッシュについて

文キャッシュにより、繰り返しコールされるループやメソッドなどで何度も使用する実行文がキャッシュされるため、パフォーマンスが向上します。

文キャッシュを使用して、次のことを実行できます。

- カーソル作成の繰返しによるオーバーヘッドを回避します。
- 文の解析と作成の繰返しを回避します。

文キャッシュの基本

文キャッシュは、特定の物理接続に対応付けられている文をキャッシュするために使用します。文キャッシュを有効にすると、暗黙的文キャッシュおよび明示的文キャッシュが自動的に有効にされます この両方のタイプの文キャッシュは、同じキャッシュを共有します。

単純な接続の場合、キャッシュは `OracleConnection` オブジェクトに対応付けられます。プーリングされた接続の場合、キャッシュは `OraclePooledConnection` オブジェクトに対応付けられます。`OracleConnection` および `OraclePooledConnection` オブジェクトには、文キャッシュを有効にするメソッドが含まれています。文キャッシュを有効にすると、close メソッドをコールするときに文オブジェクトがキャッシュされます。

物理接続ごとに独自のキャッシュがあるため、複数の物理接続に対して文キャッシュを有効にすると複数のキャッシュが存在することになります。プーリングされた接続で文キャッシュを有効にすると、すべての論理接続で同じキャッシュが使用されます。プーリングされた接続の論理接続で文キャッシュを有効にしようとすると、例外が発生します。

前述のように、文キャッシュには次の 2 つのフォームがあります。

- 暗黙的
- 明示的

暗黙的文キャッシュ

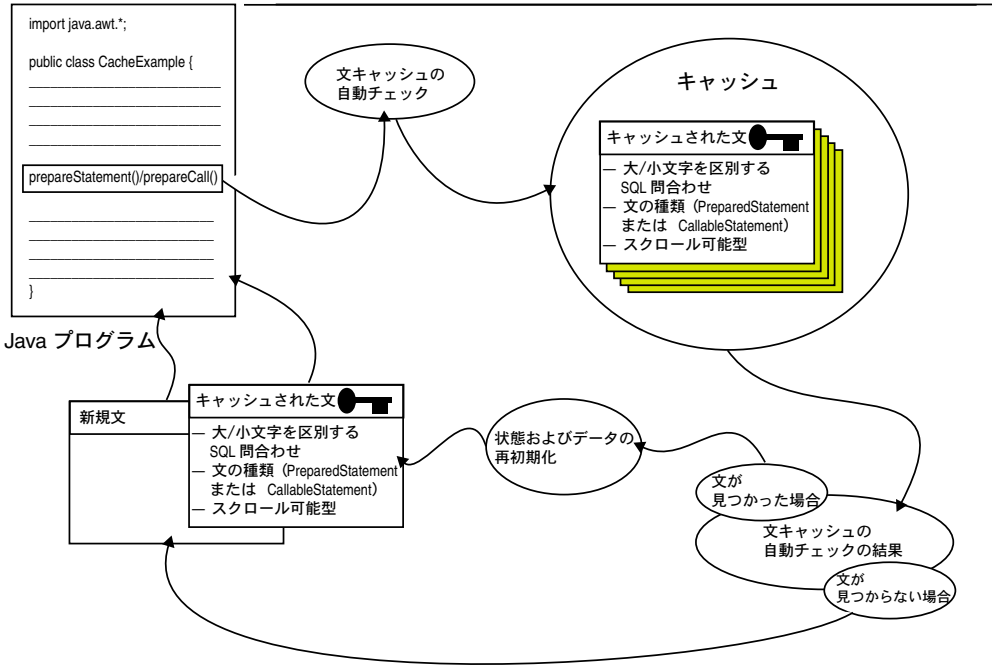
文キャッシュを有効にすると、暗黙的文キャッシュでは、プリコンパイルされた SQL 文またはコール可能文の文オブジェクトの `close()` メソッドをコールしたときに、その文が自動的にキャッシュされます。プリコンパイルされた SQL 文およびコール可能文のキャッシュおよび取出しには、標準の接続オブジェクトおよび文オブジェクト・メソッドを使用します。

暗黙的文キャッシュは SQL 文字列をキーとして使用しますが、プレーン文は SQL 文字列を使用せずに作成されるため、プレーン文は暗黙的にキャッシュされません。このため、暗黙的文キャッシュは、SQL 文字列を使用して作成される `OraclePreparedStatement` および `OracleCallableStatement` オブジェクトにのみ適用されます。これらの文が作成されると、JDBC ドライバはキャッシュを自動的に検索し、一致する文を探します。一致基準は、次のとおりです。

- 文内の SQL 文字列は、キャッシュの SQL 文字列と同一（大 / 小文字区別）である必要があります。
- 文の種類は、同じにする必要があります（プリコンパイルされた SQL 文またはコール可能文）。
- 文によって生成される結果セットのスクロール可能な型は、同じにする必要があります（forward-only または scrollable）。スクロール可能性は、文を作成するときに決定できます。（詳細は、11-7 ページの「結果セットのスクロール可能性および更新可能性の指定」を参照してください。）

図 13-1 に、暗黙的文キャッシュの概念を示します。

図 13-1 暗黙的文キャッシュのプロセス



キャッシュ検索中に一致するものが見つかった場合は、キャッシュされた文が戻されます。一致するものが見つからなかった場合は、新しい文が作成されて戻されます。新しい文は、そのカーソルおよび状態とともに、文オブジェクトの `close()` メソッドをコールするとキャッシュされます。

キャッシュされた `OraclePreparedStatement` または `OracleCallableStatement` オブジェクトが取り出されると、状態およびデータ情報は自動的に再初期化されて、デフォルト値にリセットされますが、メタデータは保存されます。最低使用頻度（LRU）スキームで、文キャッシュ操作は実行されます。

注意： デフォルトでは、JDBC ドライバはメタデータを消去しません。ただし、`clearMetaData` パラメータを `true` に設定して、`setStmtCacheSize()` メソッドをコールすると、メタデータも消去されます。メタデータは、通常消去しません。詳細は、13-6 ページの「文キャッシュの有効化および無効化」を参照してください。

専用のメソッドをコールして、特定の文が暗黙的にキャッシュされないようにできます。このメソッドの詳細は、13-9 ページの「特定の文に対する暗黙的文キャッシュの無効化」を参照してください。

明示的文キャッシュについて

文キャッシュを有効にすると、ユーザー定義のキーに基づく明示的文キャッシュでは、プリコンパイルされた SQL 文、コール可能文およびプレーン文を選択してキャッシュし、取り出すことができます。キーは、ユーザーが指定する任意の Java 文字列です。

文を明示的にキャッシュしたり、明示的にキャッシュされた文を取り出すには、Oracle WithKey メソッドを使用します。このメソッドを使用する場合は、キーを入力パラメータとして指定します。この名前のとおり、明示的文キャッシュはキャッシュする文オブジェクトごとに実行されます。

JDBC ドライバで、指定されたキーと一致する文が検出されると、その文が戻されます。JDBC ドライバで、キャッシュ内に一致する文が見つからない場合は、NULL 値が戻されます。

暗黙的文キャッシュと明示的文キャッシュの比較

明示的文キャッシュでは、メタデータの他に文データおよび状態が保持されるため、メタデータのみを保持する暗黙的文キャッシュよりもパフォーマンスの点では優れています。ただし、明示的文キャッシュでは再使用するための 3 つのタイプの情報がすべて保存されるため、このタイプのキャッシュを使用する場合には注意が必要です。前の文のデータと状態に関して保持されていた内容がわからない場合があるからです。

暗黙的文キャッシュでは、文オブジェクトの割当ておよび取出しに標準のメソッドを使用します。明示的文キャッシュでは、文を割り当てる場合は標準のメソッドを使用し、文オブジェクトをキャッシュおよび取り出す場合は、専用の Oracle WithKey メソッドを使用します。

暗黙的文キャッシュでは、プリコンパイルされた SQL 文またはコール可能文の SQL 文字列がキーとして使用され、ユーザーは特にアクションを実行する必要はありません。明示的文キャッシュでは、キーとして使用する Java 文字列を指定する必要があります。

暗黙的文キャッシュの実行中に、JDBC ドライバでキャッシュ内に一致する文を見つけられない場合は、新しい文が自動的に作成されます。明示的文キャッシュの実行中に JDBC ドライバでキャッシュ内に一致する文を見つけられない場合は、NULL 値が戻されます。

表 13-1 で、暗黙的文キャッシュと明示的文キャッシュで使用されるメソッドの違いを比較します。

表 13-1 文キャッシュで使用されるメソッドの比較

	割当て	キャッシュ	取出し
暗黙的	`prepareStatement()` `prepareCall()`	`close()`	`prepareStatement()` `prepareCall()`

明示的	`createStatement()`
	`prepareStatement()`
	`prepareCall()`
		`closeWithKey()`	`createStatementWithKey()`
			`prepareCallWithKey()`
			`prepareStatementWithKey()`

文キャッシュの使用方法

この項では、次の項目について説明します。

- 文キャッシュの有効化および無効化
- 文作成ステータスのチェック
- キャッシュされた文の物理的なクローズ
- 暗黙的文キャッシュの使用方法
- 明示的文キャッシュの使用方法

文キャッシュの有効化および無効化

文キャッシュは動的に有効化または無効化できます。文キャッシュを有効にするには、接続オブジェクトの `setStmtCacheSize()` メソッドをコールして、キャッシュを 0（ゼロ）よりも大きい値に設定します。キャッシュ・サイズには、キャッシュ内の文の最大数を指定します。

`setStmtCacheSize()` メソッドには、次のシグネチャがあります。

- `setStmtCacheSize (int size)`
- `setStmtCacheSize (int size, boolean clearMetaData)`

`size` パラメータは、キャッシュ内の文の最大数を設定します。オプションの `clearMetaData` パラメータでは、キャッシュされた文を取り出すときにメタデータを消去するかどうかを指定できます。

注意：

- 特定の物理接続に対して文キャッシュを有効化または無効化する場合は、暗黙的文キャッシュと明示的文キャッシュの両方に対して、これを行います。このため、同じセッション中で暗黙的文キャッシュと明示的文キャッシュの両方を実行できるわけです。
- 暗黙的文キャッシュと明示的文キャッシュは同じキャッシュを共有します。このため、文キャッシュのサイズを設定する場合は、この点を考慮してください。

次のコードでは、キャッシュ・サイズを 10 の文に指定します。

```
((OracleConnection)conn).setStmtCacheSize(10);
```

文キャッシュを無効化するには、`setStmtCacheSize()` メソッドをコールし、キャッシュを 0（ゼロ）に設定します。キャッシュ・サイズをチェックするには、`getStmtCacheSize()` メソッドを使用します。

次のコードでは、キャッシュ・サイズを 0（ゼロ）に設定して文キャッシュを無効化します。また、コードの最終行で新しいキャッシュの設定を検証します。

```
((OracleConnection)conn).setStmtCacheSize(0);
System.out.println("Stmt Cache size is " +
   ((OracleConnection)conn).getStmtCacheSize());
```

文作成ステータスのチェック

文オブジェクトの `creationState()` メソッドをコールすると、文が新しく作成されたのか、それとも暗黙的キャッシュの検索または明示的キャッシュの検索のどちらでキャッシュから取り出されたのかを判断できます。`creationState()` メソッドは、プレーン文、プリコンパイルされた SQL 文およびコール可能文に対し次の整数値を戻します。

- `NEW` - 文は新しく作成されました。
- `IMPLICIT` - 文は暗黙的な文の検索で取り出されました。
- `EXPLICIT` - 文は明示的な文の検索で取り出されました。

たとえば、JDBC ドライバは、明示的にキャッシュされた文に対して `OracleStatement.EXPLICIT` を戻します。次のコードは、`stmt` の文作成ステータスをチェックします。

```
int state = ((OracleStatement)stmt).creationState()
   ...(process state)
```

キャッシュされた文の物理的なクローズ

文キャッシュを有効にすると、文の物理的なクローズを手動で実行できなくなります。文オブジェクト・キャッシュの close() メソッドは、文をクローズするのではなく、キャッシュします。文は、次の 3 つの条件で自動で物理的にクローズされます。1）対応付けられている接続がクローズされた場合。2）キャッシュがそのサイズ制限に達し、最低使用頻度の文オブジェクトを、LRU スキームにより、キャッシュが別の人のために転用する場合。3）文キャッシュが無効にされている文で close() メソッドをコールした場合。（詳細は、13-9 ページの「特定の文に対する暗黙的文キャッシュの無効化」を参照してください。）

暗黙的文キャッシュの使用方法

文キャッシュを有効にすると、デフォルトではすべてのプリコンパイルされた SQL 文およびコール可能文が自動的にキャッシュされます。暗黙的文キャッシュでは、次の手順が実行されます。

1. 13-6 ページの「文キャッシュの有効化および無効化」の記述を参照し、キャッシュを有効にします。
2. 標準メソッドの 1 つを使用して文を割り当てます。
3. オプションで、キャッシュさせたくない特定の文に対し暗黙的文キャッシュを無効化します。
4. close() メソッドを使用して文をキャッシュします。
5. 暗黙的にキャッシュされた文を取り出すには、適切な標準の prepare メソッドをコールします。

暗黙的文キャッシュの完全なコード例については、20-76 ページの「暗黙的文キャッシュ：StmtCache1.java」を参照してください。次の項では、暗黙的文キャッシュの処理について詳しく説明します。

暗黙的キャッシュのための文の割当て

暗黙的文キャッシュに文を割り当てるには、通常 prepareStatement() または prepareCall() メソッドを使用します。（これらは接続オブジェクトのメソッドです。）

次のコードでは、pstmt と呼ばれる新しい文オブジェクトを割り当てます。

```
PreparedStatement pstmt = conn.prepareStatement
   ("UPDATE emp SET ename = ? WHERE rowid = ?");
```

特定の文に対する暗黙的文キャッシュの無効化

ある接続の文キャッシュを有効にすると、デフォルトでは、その接続のすべてのコール可能文およびプリコンパイルされた SQL 文が自動的にキャッシュされます。特定のコール可能文またはプリコンパイルされた SQL 文を暗黙的にキャッシュしないようにするには、文オブジェクトの setDisableStmtCaching() メソッドを使用します。キャッシュ領域を管理するためには、使用頻度の低い文で setDisableStmtCaching() メソッドをコールします。

次のコードでは、pstmt に対する暗黙的文キャッシュを無効化します。

```
PreparedStatement pstmt = conn.prepareStatement ("SELECT 1 from DUAL");
((OraclePreparedStatement)pstmt).setDisableStmtCaching (true);
pstmt.close ();
```

文の暗黙的なキャッシュ

割り当てられた文をキャッシュするには、文オブジェクトの close() メソッドをコールします。OraclePreparedStatement または OracleCallableStatement オブジェクトで close() メソッドをコールすると、この文のキャッシュを無効化していない限り、JDBC ドライバではこの文がキャッシュ内に自動的に挿入されます。

次のコードでは、pstmt 文をキャッシュします。

```
((OraclePreparedStatement)pstmt).close ();
```

暗黙的にキャッシュされた文の取出し

暗黙的にキャッシュされた文を再度コールするには、文の種類に応じて、prepareStatement() または prepareCall() メソッドをコールします。

次のコードでは、prepareStatement() メソッドを使用して、キャッシュから pstmt を再度コールします。

```
pstmt = conn.prepareStatement ("UPDATE emp SET ename = ? WHERE rowid = ?");
```

pstmt 文オブジェクトで creationState() メソッドをコールすると、メソッドは IMPLICIT を戻します。pstmt 文オブジェクトがキャッシュ内になかった場合、creationState() メソッドは、JDBC ドライバにより新しい文が作成されたことを示す NEW を戻します。

文を割り当てたり、暗黙的にキャッシュされた文を取り出すメソッドについて、表 13-2 で説明します。

表 13-2 文の割当ておよび暗黙的文キャッシュで使用されるメソッド

メソッド	暗黙的文キャッシュの機能
`prepareStatement()`	目的のキャッシュされた `OraclePreparedStatement` オブジェクトを検索して戻すキャッシュ検索を起動します。一致するものが見つからない場合は新しい `OraclePreparedStatement` オブジェクトを割り当てます。
`prepareCall()`	目的のキャッシュされた `OracleCallableStatement` オブジェクトを検索して戻すキャッシュ検索を起動します。一致するものが見つからない場合は新しい `OracleCallableStatement` オブジェクトを割り当てます。

明示的文キャッシュの使用方法

文キャッシュを有効にすると、プレーン文、プリコンパイルされた SQL 文またはコール可能文を明示的にキャッシュできます。明示的文キャッシュでは、次の処理が行われます。

1. 13-6 ページの「文キャッシュの有効化および無効化」の記述を参照し、キャッシュを有効にします。
2. 標準メソッドの 1 つを使用して文を割り当てます。
3. 文を明示的にキャッシュするには、`closeWithKey()` メソッドを使用して、キーを付けてその文をクローズします。
4. 明示的にキャッシュされた文を取り出すには、適切な WithKey メソッドをコールして、適切なキーを指定します。
5. オープンしている、明示的にキャッシュされた文を再キャッシュするには、`closeWithKey()` メソッドを使用して文を再度クローズします。キャッシュされた文をクローズするたびに、そのキーを使用して文を再キャッシュします。

明示的文キャッシュの完全なコード例については、20-79 ページの「明示的文キャッシュ：StmtCache2.java」を参照してください。次の項では、明示的文キャッシュの処理について詳しく説明します。

明示的なキャッシュのための文の割当て

明示的文キャッシュに文を割り当てるには、通常と同じく createStatement()、prepareStatement() または prepareCall() メソッドを使用します。(これらは接続オブジェクトのメソッドです。)

次のコードでは、pstmt と呼ばれる新しい文オブジェクトを割り当てます。

```
PreparedStatement pstmt =
   conn.prepareStatement ("UPDATE emp SET ename = ? WHERE rowid = ?");
```

文の明示的なキャッシュ

割り当てられた文を明示的にキャッシュするには、文オブジェクトの closeWithKey() メソッドをコールして、キーを指定します。キーは、ユーザーが指定する任意の Java 文字列です。closeWithKey() メソッドは、文をそのままキャッシュします。つまり、データ、状態およびメタデータは消去されずに保持されます。

次のコードでは、キー "mykey" を使用して pstmt 文をキャッシュします。

```
((OraclePreparedStatement)pstmt).closeWithKey ("mykey");
```

明示的にキャッシュされた文の取出し

明示的にキャッシュされた文を再度コールするには、文の種類によって、createStatementWithKey()、prepareStatementWithKey() または prepareCallWithKey() メソッドをコールします。

指定したキーを使用して文を取り出す場合は、指定したキーに基づいて、JDBC ドライバがキャッシュ内でその文を検索します。一致する文が見つかった場合は、一致する文は、状態、データおよびメタデータとともに戻されます。情報は、最後にクローズしたときの状態で戻されます。一致する文が見つからない場合は、JDBC ドライバで null が戻されます。

次のコードでは、prepareStatementWithKey() メソッドで "mykey" キーを使用して、キャッシュから pstmt を再コールします。pstmt 文オブジェクトは "mykey" キーでキャッシュされていました。

```
pstmt = ((OracleConnection)conn).prepareStatementWithKey ("mykey");
```

pstmt 文オブジェクトで creationState() メソッドをコールすると、メソッドは EXPLICIT を戻します。

重要: 明示的にキャッシュされた文を取り出す場合は、キーを指定するときに文の種類に適したメソッドを使用してください。たとえば、prepareStatement() メソッドを使用して文を割り当てた場合は、prepareStatementWithKey() メソッドを使用してキャッシュから文を取り出します。JDBC ドライバでは、戻される文の種類を検証できません。

明示的にキャッシュされた文を取り出す場合に使用するメソッドについて、表 13-3 で説明します。

表 13-3 明示的にキャッシュされた文を取り出す場合に使用するメソッド

メソッド	明示的文キャッシュの機能
`createStatementWithKey()`	キャッシュからプレーン文を取り出すのに必要なキーを指定します。
`prepareStatementWithKey()`	キャッシュからプリコンパイルされた SQL 文を取り出すのに必要なキーを指定します。
`prepareCallWithKey()`	キャッシュからコール可能文を取り出すのに必要なキーを指定します。

14

接続プーリングとキャッシュ

この章では、次の Oracle JDBC 実装について説明します。
(1) データベースなどの使用するリソースを指定するための標準機能であるデータ・ソース。
(2) データベース接続のキャッシュのためのフレームワークである接続プーリング。
(3) Oracle 実装の説明を含む接続キャッシュのサンプル。
標準 Java Naming and Directory Interface（JNDI）の Oracle JDBC サポートについても説明します。

この章で説明している内容は、すべての Oracle JDBC ドライバに該当します。次の項目が含まれます。

- データ・ソース
- 接続プーリング
- 接続キャッシュ

> **注意：** この章では、Sun 社の JDBC 2.0 Standard Extension API の機能について説明します。この API は、Sun 社の `javax` パッケージを通じて使用できます。この Optional Package は、標準 JDK の一部ではありませんが、関連するパッケージが Oracle JDBC `classes111.zip` および `classes12.zip` ファイルとともに含まれています。

これらの項目の詳細は、Sun 社の JDBC 2.0 Standard Extension API の仕様を参照してください。OCI ドライバに固有の接続プーリング機能に関する詳細は、16-2 ページ「OCI ドライバ接続プーリング」を参照してください。

データ・ソース

JDBC2.0 Standard Extension API によって、データ・ソースの概念が導入されました。これは使用するデータベースまたはその他のリソースを指定するための標準汎用オブジェクトです。データ・ソースは、便宜性と移植性のためにオプションとして Java Naming and Directory Interface（JNDI）エントリにバインドでき、論理名でデータベースにアクセスできます。

この機能は、3-4 ページの「データベースへの接続のオープン」で説明する接続機能のさらに標準的で多機能な代替手段です。データ・ソース機能は、以前の JDBC `DriverManager` 機能の完全な代替機能を提供します。

どちらの機能も同じアプリケーションで使用できますが、接続プーリングまたは分散トランザクションのどちらの場合でも、開発者はデータ・ソースを使用して接続を行う方が多くなると考えられます。この結果、Sun 社は、`DriverManager` とその関連のクラスおよび機能の提供をやめる可能性があります。

データ・ソースと JNDI の概要および一般情報については、Sun 社の JDBC 2.0 Optional Package に関する仕様を参照してください。

JNDI の Oracle データ・ソース・サポートに関する簡単な概要

標準 Java Naming and Directory Interface（JNDI）を使用すると、アプリケーションはリモート・サービスおよびリソースを検出して、それにアクセスできます。リモート・サービスはエンタープライズ・サービスのこともありますが、JDBC アプリケーションではデータベース接続およびサービスも含まれます。

JNDI を使用すると、アプリケーションはアプリケーション・コードからベンダー固有の構文を削除し、論理名を使用してこれらのサービスにアクセスできます。JNDI には、目的のサービスの特定のソースと論理名を対応付ける機能があります。

すべての Oracle JDBC データ・ソースでは、JNDI を参照できます。開発者が必ずしもこの機能を使用する必要はありませんが、JNDI 論理名を使用してデータベースにアクセスすると、コードの移植性が高まります。

注意： JNDI の機能を使用するには、ファイル `jndi.jar` が `CLASSPATH` にある必要があります。このファイルは Oracle9*i* CD の Java 製品に入っていますが、`classes12.zip` ファイルと `classes111.zip` ファイルには含まれていません。このファイルは個別に `CLASSPATH` に追加する必要があります。（Sun 社の Web サイトから取得することもできますが、オラクル社が提供するバージョンを使用することをお薦めします。このバージョンは、Oracle ドライバでのテストが済んでいます。）

データ・ソースの機能とプロパティ

JDBC DriverManager クラスを使用して、ドライバ・クラスを登録し、データベース接続をオープンする方法については、3-2 ページの「JDBC での最初の処理」を参照してください。このモデルの問題は、コードにベンダー固有のクラス名、データベース URL、マシン名やポート番号などのその他のプロパティを記述する必要がある点です。

JNDI を使用する JDBC2.0 データ・ソース機能では、ベンダー固有の JDBC ドライバ・クラス名を登録する必要はなく、URL とその他のプロパティの論理名を使用できます。このため、データベース接続をオープンするためのアプリケーション・コードは、その他の環境に対して移植可能になります。

データ・ソース・インタフェースと Oracle による実装

JDBC データ・ソースは、標準 javax.sql.DataSource インタフェースを実装するクラスのインスタンスです。

```
public interface DataSource
{
   Connection getConnection() throws SQLException;
   Connection getConnection(String username, String password)
      throws SQLException;
   ...
}
```

Oracle では、このインタフェースを oracle.jdbc.pool パッケージの OracleDataSource クラスに実装しています。オーバーロードされた getConnection() メソッドは、OracleConnection インスタンスを戻します。オプションで、入力としてユーザー名とパスワードを取ることもあります。

他の値を使用するには、次の項で説明する適切な set メソッドを使用して、プロパティを設定します。代替的なユーザー名とパスワードを設定するために、入力としてこれらの値を取る getConnection() シグネチャも使用できます。これはプロパティ設定よりも優先されます。

注意： OracleDataSource クラスとすべてのサブクラスは、java.io.Serializable インタフェースと javax.naming.Referenceable インタフェースを実装します。

データ・ソース・プロパティ

DataSource インタフェースを実装するすべてのクラスと同様に、OracleDataSource クラスは、接続するデータベースを指定するために使用できるプロパティの集合を提供します。これらのプロパティは JavaBeans 設計パターンに従います。

表 14-1 と表 14-2 には、ドキュメントの OracleDataSource プロパティを示します。表 14-1 のプロパティは、Sun 社仕様に従う標準プロパティです。（ただし、Oracle は標準 roleName プロパティを実装していないことに注意してください。）表 14-2 のプロパティは Oracle 拡張機能です。

表 14-1 標準データ・ソース・プロパティ

名前	型	説明
databaseName	String	Oracle の用語では「SID」ともいうサーバー上の特定のデータベース名。
dataSourceName	String	基礎となるデータ・ソース・クラスの名前（接続プーリングでは基礎となるプーリングされた接続データ・ソース・クラスで、分散トランザクションでは基礎となる XA データ・ソース・クラスです）。
description	String	データ・ソースの説明。
networkProtocol	String	サーバーと通信するためのネットワーク・プロトコル。Oracle では、これは OCI ドライバのみに適用され、デフォルトは tcp です。 （ipc を設定することもできます。詳細は、『Oracle9*i* Net Services 管理者ガイド』を参照してください。）
password	String	ユーザー名のログイン・パスワード。
portNumber	int	サーバーが要求をリスニングするポート番号。
serverName	String	データベース・サーバーの名前。
user	String	ログイン・アカウントの名前。

OracleDataSource クラスは、標準プロパティである次の set メソッドと get メソッドを実装します。

- `public synchronized void setDatabaseName(String dbname)`
- `public synchronized String getDatabaseName()`
- `public synchronized void setDataSourceName(String dsname)`
- `public synchronized String getDataSourceName()`
- `public synchronized void setDescription(String desc)`

- `public synchronized String getDescription()`
- `public synchronized void setNetworkProtocol(String np)`
- `public synchronized String getNetworkProtocol()`
- `public synchronized void setPassword(String pwd)`
- `public synchronized void setPortNumber(int pn)`
- `public synchronized int getPortNumber()`
- `public synchronized void setServerName(String sn)`
- `public synchronized String getServerName()`
- `public synchronized void setUser(String user)`
- `public synchronized String getUser()`

セキュリティ上の理由のために `getPassword()` メソッドはないことに注意してください。

表 14-2 Oracle 拡張データ・ソース・プロパティ

名前	型	説明
`driverType`	`String`	Oracle JDBC ドライバのタイプを `oci`、`thin` または `kprb`（サーバー側内部ドライバ）のいずれかとして表します。
`tnsEntry`	`String`	TNS エントリ名は、OCI ドライバのみに関係します。また、環境変数 `TNS_ADMIN` を適切に設定した Oracle クライアントがインストールされていることを想定しています。 OCI ドライバで HeteroRM XA 機能を使用する場合は、この `OracleXADataSource` プロパティを有効に設定して、Oracle リリース 8.1.6 以前のデータベースにアクセスしてください。HeteroRM XA 機能については、16-17 ページの「OCI HeteroRM XA」で説明しています。HeteroRM XA 機能を使用するときに `Entry` プロパティが設定されていないと、エラー・コード ORA-17207 の SQLException が発生します。
`url`	`String`	データベース接続文字列の URL です。以前のバージョンの Oracle データベースから移行する際の便宜を考えて提供されるものです。Oracle`tnsEntry` および `driverType` プロパティ、標準 `portNumber`、`networkProtocol`、`serverName` および `databaseName` プロパティのかわりに、このプロパティを使用できます。

表 14-2 Oracle 拡張データ・ソース・プロパティ （続き）

名前	型	説明
nativeXA	boolean	OCI ドライバで HeteroRM XA 機能を使用する場合は、この OracleXADataSource プロパティを有効に設定して、Oracle リリース 8.1.6 以前のデータベースにアクセスしてください。HeteroRM XA 機能については、16-17 ページの「OCI HeteroRM XA」で説明しています。nativeXA プロパティを有効にする場合は、必ず tnsEntry プロパティも設定してください。 この DataSource プロパティは、デフォルトで FALSE に設定されます。

注意： nativeXA のほうが JavaXA よりパフォーマンスが高いので、可能なかぎり、nativeXA を使用してください。

OracleDataSource クラスは、Oracle 拡張プロパティの次の setXXX() メソッドと getXXX() メソッドを実装します。

- public synchronized void setDriverType(String dt)
- public synchronized String getDriverType()
- public synchronized void setURL(String url)
- public synchronized String getURL()
- public synchronized void setTNSEntryName(String tns)
- public synchronized String getTNSEntryName()
- public synchronized void setNativeXA(boolean nativeXA)
- public synchronized boolean getNativeXA()

サーバー側内部ドライバを使用している場合は、driverType プロパティは kprb に設定され、その他のプロパティ設定は無視されます。

Thin ドライバまたは OCI ドライバを使用している場合は、次の点に注意してください。

- URL 設定には、次の例のように user と password の設定が含まれることがあります。この場合は、この設定が個々の user および password プロパティ設定よりも優先されます。

  ```
  jdbc:oracle:thin:scott/tiger@localhost:1521:orcl
  ```

- user と password は、直接指定するか、URL を介してまたは getConnection() をコールして設定する必要があります。getConnection() コールによる user と password の設定は、他のプロパティ設定よりも優先されます。
- url プロパティを設定した場合は、tnsEntry、driverType、portNumber、networkProtocol、serverName および databaseName プロパティ設定は無視されます。
- tnsEntry プロパティを設定した場合は（url プロパティは設定されていないものとします）、databaseName、serverName、portNumber および networkProtocol 設定は無視されます。
- OCI ドライバを使用し（driverType プロパティは oci に設定されているものとします）、networkProtocol を ipc に設定した場合は、他のプロパティ設定は無視されます。

データ・ソース・インスタンスの作成と接続（JNDI なし）

この項では、JNDI 機能を使用せずにデータベースに接続する場合の、データ・ソースの最も基本的な使用例を示します。ベンダーによってはベンダー固有のハードコードされたプロパティ設定が必要であることに注意してください。

次の例のように OracleDataSource インスタンスを作成し、必要に応じて接続プロパティを初期化して、接続インスタンスを取得します。

```
...
OracleDataSource ods = new OracleDataSource();

ods.setDriverType("oci8");
ods.setServerName("dlsun999");
ods.setNetworkProtocol("tcp");
ods.setDatabaseName("816");
ods.setPortNumber(1521);
ods.setUser("scott");
ods.setPassword("tiger");

Connection conn = ods.getConnection();
...
```

または、オプションとしてユーザー名とパスワードをオーバーライドします。

```
...
Connection conn = ods.getConnection("bill", "lion");
...
```

完全なサンプル・プログラムについては、20-82 ページの「JNDI を使用しないデータ・ソース : DataSource.java」を参照してください。

データ・ソース・インスタンスの作成、JNDI での登録および接続

この項では、データベースに接続するためのデータ・ソースの使用に関する JNDI 機能を示します。ベンダー固有のハードコードされたプロパティ設定は、データ・ソース・インスタンスを JNDI 論理名にバインドするコードの部分でのみ必要になります。これ以降は、接続インスタンスを取得するデータ・ソースを作成するために、論理名を使用して、移植可能なコードを作成できます。

完全なサンプルについては、20-83 ページの「JNDI を使用するデータ・ソース : DataSourceJNDI.java」を参照してください。

注意 : データ・ソースの作成と登録は、一般に JDBC アプリケーションではなく、JNDI 管理者によって処理されます。

接続プロパティの初期化

次の例に示すように、`OracleDataSource` インスタンスを作成し、必要に応じてその接続プロパティを初期化します。

```
...
OracleDataSource ods = new OracleDataSource();

ods.setDriverType("oci8");
ods.setServerName("dlsun999");
ods.setNetworkProtocol("tcp");
ods.setDatabaseName("816");
ods.setPortNumber(1521);
ods.setUser("scott");
ods.setPassword("tiger");
...
```

データ・ソースの登録

前述の例で示したように、`OracleDataSource` インスタンス `ods` の接続プロパティを初期化すると、次の例に示すように JNDI でこのデータ・ソース・インスタンスを登録できます。

```
...
Context ctx = new InitialContext();
ctx.bind("jdbc/sampledb", ods);
...
```

JNDI `InitialContext()` コンストラクタをコールすると、初期 JNDI ネーミング・コンテキストを参照する Java オブジェクトが作成されます。表示されていないシステム・プロパティによって、どのサービス・プロバイダを使用するか JNDI に指示します。

ctx.bind() コールは、OracleDataSource インスタンスを論理 JNDI 名にバインドします。つまり、ctx.bind() をコールした後、いつでも論理名 jdbc/sampledb を使用して、OracleDataSource インスタンス ods のプロパティによって記述されるデータベースへの接続をオープンできます。論理名 jdbc/sampledb は、このデータベースに論理的にバインドされます。

JNDI ネームスペースの階層構造は、ファイル・システムの階層構造と似ています。この例で、JNDI 名はルート・ネーミング・コンテキストでサブコンテキスト jdbc を指定し、jdbc サブコンテキスト内で論理名 sampledb を指定します。

Context インスタンスと InitialContext クラスは、標準 javax.naming パッケージ内にあります。

注意： JDBC 2.0 仕様では、すべての JDBC データ・ソースを、JNDI ネームスペースの jdbc ネーミング・サブコンテキスト、または jdbc サブコンテキストの子サブコンテキストに登録する必要があります。

Oracle9*i* JNDI でのデータ・ソースの登録

OracleDataSource、OracleConnectionPoolDataSource および OracleXADataSource インスタンスを JNDI にバインドするには、Oracle9*i* sess_sh（セッション・シェル）ツールの bindds コマンドを使用します。sess_sh ツールは、データベース・インスタンスのセッション・ネームスペースに対するインタラクティブ・インタフェースとして Oracle9*i* で提供されています。sess_sh ツールは、UNIX ファイル・システムのルック・アンド・フィールを持ったセッション・ネームスペースを与えるシェル・コマンドを提供します。（sess_sh ツールを起動すると、$ コマンドライン・プロンプトが表示されます。）bindds コマンドの使用方法の詳細は、『Oracle9*i* Java Tools Reference』マニュアルを参照してください。

Oracle9*i* JNDI ネームスペースでデータ・ソースをバインドするには、次の構文を使用します。

```
$ bindds <datasource_name> [options]
```

表 14-3 で、bindds コマンド・オプションを説明します。

表 14-3 bindds コマンドを使用するデータ・ソース・オプション

オプション	説明
-h、-help	bindds オプションに関する情報を出力します。
-version	バージョン情報を出力します。
-describe	コマンドの実行内容の記述を出力します。
-rebind	データ・ソース・オブジェクトがバインドされている場合は、バインドを強制します。

表 14-3 bindds コマンドを使用するデータ・ソース・オプション（続き）

オプション	説明
-g、-G、-grant <schema1,schema2...>	新しいオブジェクトとディレクトリに read または execute スキームを追加します。
-rg、-rG、-recursiveGrant <schema1,schema2...>	新しいデータ・ソース・オブジェクトとディレクトリに、read または execute スキームを追加します。
-dstype <pool/xa/jta>	OracleDataSource、OraclePooled、OracleXA または OracleJTADataSource（デフォルトは OracleDataSource）のいずれかをバインドします。
-url <jdbc_url>	URL（jdbc:oracle:oci:@ など）を指定します。
-host <host_name>	データ・ソースの完全修飾ホスト名を指定します。
-port <port_number>	このデータ・ソースが受信するポートを指定します。
-sid <database_name>	データベース ID を指定します。
-driver <JDBC_driver>	KPRB、OCI または Thin などの JDBC ドライバを指定します。
-dblink <dblink>	完全修飾されたデータベース・リンクを指定します。
-protocol <protocol>	データソースへの接続に使用するプロトコルを指定します。
-tns <tnsEntry>	tnsnames.ora ファイルの Transparent Network Substrate（TNS）エントリを指定します（『Oracle9*i* Net Services 開発者ガイド』を参照）。
-u <user>、-user <user>	データ・ソース接続用のユーザー名を指定します。
-p <pwd>、-password <pwd>	ユーザー名のパスワードを指定します。

次の bindds 例では、jdbcdb（論理名）として Oracle9*i* JNDI ネームスペースにバインドされている Oracle データ・ソース・オブジェクトを作成します。

```
$ bindds jdbcdc -url jdbc:oracle:oci8:@ -u scott -p tiger
```

接続のオープン

lookup を実行して、JNDI 名に論理的にバインドされたデーターベースへの接続をオープンするには、論理 JNDI 名を使用します。これを実行するためには、lookup の結果（Java Object のみも可）を新しい OracleDataSource インスタンスにキャストし、その getConnection() メソッドを使用して接続をオープンします。

次に例を示します。

```
...
OracleDataSource odsconn = (OracleDataSource)ctx.lookup("jdbc/sampledb");
Connection conn = odsconn.getConnection();
...
```

ロギングとトレース

データ・ソース機能を使用すると、JDBC がエラーのロギングおよびトレース情報の出力として使用するキャラクタ・ストリームを登録できます。この機能によって、特定のデータ・ソース・インスタンスに固有なトレースを実行できます。すべてのデータ・ソース・インスタンスで同じキャラクタ・ストリームを使用する場合は、データ・ソース・インスタンスごとにそのストリームを登録する必要があります。

OracleDataSource クラスは、ロギングとトレースのために、次の標準データ・ソース・メソッドを実装します。

- public synchronized void setLogWriter(PrintWriter pw)
- public synchronized PrintWriter getLogWriter()

PrintWriter クラスは、標準 java.io パッケージ内にあります。

注意：

- データ・ソース・インスタンスが作成された場合、デフォルトでロギングは使用禁止になります（ログ・ストリーム名は、最初は NULL です）。
- データ・ソース・インスタンスに対して登録したログ・ストリームに書き込まれるメッセージは、通常 DriverManager によって管理されるログ・ストリームには書き込まれません。
- データ・ソース・インスタンスを初めに JNDI 名にバインドしたときに PrinterWriter を設定した場合でも、JNDI 名前参照から取得される OracleDataSource に PrintWriter セットはありません。

接続プーリング

JDBC 2.0 Standard Extension API の接続プーリングは、データベース接続をキャッシュするためのフレームワークです。接続プーリングによって、物理接続を再利用して、アプリケーションのオーバーヘッドを減らすことができます。接続プーリング機能は、セッションの作成とクローズに関して、リソースを消費する操作を最小限に抑えます。

次に、主な概念を示します。

- 接続プーリング・データ・ソース。概念と機能の点で、前述したデータ・ソースに似ていますが、通常の接続インスタンスのかわりにプーリングされた接続インスタンスを戻すメソッドがあります。
- プーリングされた接続。プーリングされた接続インスタンスは、一連の論理接続インスタンスによって使用中にオープンされたままの、データベースへの単一の物理接続を表します。

  論理接続インスタンスは、プーリングされた接続インスタンスによって戻される単一の接続インスタンスです（標準 `Connection` インスタンスまたは `OracleConnection` インスタンスなど）。各論理接続インスタンスは、プーリングされた接続インスタンスが表す物理接続の一時的なハンドルとしての役割を果たします。

OCI ドライバに固有の接続プーリングに関する詳細は、16-2 ページの「OCI ドライバ接続プーリング」を参照してください。接続プーリングの概要および一般情報については、Sun 社の JDBC 2.0 Optional Package に関する仕様を参照してください。

注意： 接続プーリングの概念は、単純にデフォルト接続を使用するサーバー側内部ドライバには関係せず、単一セッション内のサーバー側 Thin ドライバのみに関係します。

接続プーリングの概念

接続プーリングを使用しない場合は、各接続インスタンス（`java.sql.Connection` インスタンスまたは `oracle.jdbc.OracleConnection` インスタンス）は、独自の物理データベース接続をカプセル化します。接続インスタンスの `close()` メソッドをコールすると、物理接続自体はクローズされます。これは 14-2 ページの「データ・ソース」で説明している JDBC 2.0 データ・ソース機能、または 3-4 ページの「データベースへの接続のオープン」で説明している `DriverManager` 機能のいずれかを使用してその接続インスタンスを取得した場合に当てはまります。

接続プーリングでは、追加ステップを実行すると、物理接続の一時的なハンドルの機能を果たす複数の論理接続インスタンスによって、物理データベース接続を再利用できます。物理データベース接続をカプセル化するプーリングされた接続を戻すには、接続プーリング・データ・ソースを使用します。次に、プーリングされた接続を使用して、それぞれが一時ハンドルとしての役割を果たす JDBC 接続インスタンスを（一度に 1 つずつ）戻します。

プーリングされた接続から取得した接続インスタンスをクローズしても、物理データベース接続はクローズされません。ただし、接続インスタンスのリソースが解放され、状態がクリアされ、接続インスタンスから作成された文オブジェクトがクローズされ、作成される次の接続インスタンスのデフォルトがリストアされます。

実際に物理接続をクローズするには、プーリングされた接続の `close()` メソッドを実行する必要があります。これは、通常中間層で実行します。

接続プーリング・データ・ソース・インタフェースと Oracle 実装

`javax.sql.ConnectionPoolDataSource` インタフェースは、プーリングされた接続のファクトリになる接続プーリング・データ・ソースの標準機能のアウトラインを示します。オーバーロードされた `getPooledConnection()` メソッドは、プーリングされた接続インスタンスを戻し、入力としてオプションのユーザー名とパスワードを取得します。

```
public interface ConnectionPoolDataSource
{
   PooledConnection getPooledConnection() throws SQLException;
   PooledConnection getPooledConnection(String user, String password)
      throws SQLException;
   ...
}
```

Oracle JDBC では、`ConnectionPoolDataSource` インタフェースを `oracle.jdbc.pool.OracleConnectionPoolDataSource` クラスに実装しています。このクラスは `OracleDataSource` クラスを拡張するため、14-4 ページの「データ・ソース・プロパティ」で説明しているすべての接続プロパティと getter および setter メソッドが含まれます。

`OracleConnectionPoolDataSource` クラスの `getPooledConnection()` メソッドは、プーリングされた接続インスタンスの Oracle 実装を戻します。これは（次の項で説明するように）、`OraclePooledConnection` インスタンスです。

注意： 14-8 ページの「データ・ソースの登録」で説明した非プーリング・データ・ソースの場合と同じネーミング規則を使用して、接続プーリング・データ・ソースを JNDI に登録できます。

プーリングされた接続インタフェースと Oracle 実装

プーリングされた接続インスタンスは、データベースへの物理接続をカプセル化します。このデータベースは、プーリングされた接続インスタンスの作成に使用する接続プーリング・データ・ソース・インスタンスの接続プロパティに指定するデータベースになります。

プーリングされた接続インスタンスは、標準 `javax.sql.PooledConnection` インタフェースを実装するクラスのインスタンスです。このインタフェースによって指定される `getConnection()` メソッドは、非プーリング接続インスタンスの場合のように、物理接続をカプセル化するのではなく、物理接続への一時ハンドルとしての役割を果たす論理接続インスタンスを戻します。

```
public interface PooledConnection
{
   Connection getConnection() throws SQLException;
   void close() throws SQLException;
   void addConnectionEventListener(ConnectionEventListener listener) ... ;
   void removeConnectionEventListener(ConnectionEventListener listener);
   void setStmtCacheSize(int size);
   void setStmtCacheSize(int size, boolean clearMetaData);
   int getStmtCacheSize();
}
```

（接続キャッシュで使用するイベント・リスナーについては、14-19 ページの「接続キャッシュの使用に関する一般的な手順」で説明します。）

Oracle JDBC では、`PooledConnection` インタフェースを `oracle.jdbc.pool.OraclePooledConnection` クラスに実装しています。`getConnection()` メソッドは、`OracleConnection` インスタンスを戻します。

プーリングされた接続インスタンスは、通常、その存在中に一連の接続インスタンスの作成を要求されますが、特定の時点では、その中の 1 つの接続インスタンスのみをオープンできます。

プーリングされた接続インスタンスの `getConnection()` メソッドをコールするたびに、デフォルトの動作を示す新しい接続インスタンスが戻され、その時点で存在し、同じプーリングされた接続インスタンスによって戻された以前の接続インスタンスがクローズされます。ただし、前の接続インスタンスは、新しい接続インスタンスをオープンする前に、明示的にクローズする必要があります。

プーリングされた接続インスタンスの `close()` メソッドをコールすると、データベースへの物理接続がクローズされます。これは、中間層レイヤーで通常実行されます。

`OraclePooledConnection` クラスには、プーリングされている接続に対して文キャッシュを有効にするためのメソッドが含まれています。文のキャッシュは、全体としてプーリングされている接続に対してメンテナンスされ、プーリングされている接続から取得されたすべての論理接続がそのキャッシュを共有します。このため、文キャッシュを有効にすると、1 つの論理接続で作成した文は、別の論理接続で再使用できます。同じ理由で、個別の

論理接続では文キャッシュを有効化または無効化できません。この機能は、暗黙的文キャッシュと明示的文キャッシュの両方に適用されます。

次に、文キャッシュに対する OraclePooledConnection メソッド定義を示します。

```
public void setStmtCacheSize (int size)
   throws SQLException

public void setStmtCacheSize (int size, boolean clearMetaData)
   throws SQLException

public int getStmtCacheSize()
```

文キャッシュの詳細は、第 13 章「文キャッシュ」を参照してください。

接続プーリング・データ・ソースの作成と接続

この項では、JNDI 機能を使用せずにデータベースに接続する場合の、接続プーリング・データ・ソースの最も基本的な使用例を示します。14-8 ページの「データ・ソースの登録」で示す汎用データ・ソース・インスタンスの場合と同様に、オプションとして JNDI を使用して、JNDI 論理名に接続プーリング・データ・ソース・インスタンスをバインドできます。

Oracle 接続プーリングのためのインポートの概要

Oracle 接続プーリング機能のために、次のパッケージをインポートする必要があります。

```
import oracle.jdbc.pool.*;
```

このパッケージには、接続キャッシュとイベント処理のためのクラスに加えて、OracleDataSource、OracleConnectionPoolDataSource および OraclePooledConnection クラスが含まれます。これらのクラスについては、14-16 ページの「接続キャッシュ」で説明します。

Oracle 接続プーリング・コードのサンプル

この例では、最初に OracleConnectionPoolDataSource インスタンスを作成し、次にその接続プロパティを初期化し、接続プーリング・データ・ソース・インスタンスからプーリングされた接続インスタンスを取得し、最後にプーリングされた接続インスタンスから接続インスタンスを取得します。(getPooledConnection() メソッドは、OraclePooledConnection インスタンスを自動的に戻しますが、この場合は汎用 PooledConnection 機能のみが要求されます。)

```
...
OracleConnectionPoolDataSource ocpds = new OracleConnectionPoolDataSource();

ocpds.setDriverType("oci8");
ocpds.setServerName("dlsun999");
ocpds.setNetworkProtocol("tcp");
ocpds.setDatabaseName("816");
ocpds.setPortNumber(1521);
ocpds.setUser("scott");
ocpds.setPassword("tiger");

PooledConnection pc = ocpds.getPooledConnection();

Connection conn = pc.getConnection();
...
```

完全なサンプル・プログラムについては、20-86 ページの「プーリングされた接続 : PooledConnection.java」を参照してください。

接続キャッシュ

一般に中間層で実装される接続キャッシュは、物理データベース接続のキャッシュを維持し、使用するための手段です。

接続キャッシュでは、その操作のほとんどで接続プーリング・データ・ソースとプーリングされた接続などの接続プーリング・フレームワークが使用されます。このフレームワークについては、14-12 ページの「接続プーリング」で説明します。

JDBC 2.0 仕様は、接続キャッシュ実装を必須としていませんが、Oracle は単純な実装を提供して、例を示します。

この項には、次の項目があります。

- 接続キャッシュの概要
- 接続キャッシュの使用に関する一般的な手順
- Oracle 接続キャッシュ仕様 : OracleConnectionCache インタフェース
- Oracle 接続キャッシュ実装 : OracleConnectionCacheImpl クラス
- Oracle 接続イベント・リスナー : OracleConnectionEventListener クラス

注意 : 接続キャッシュの概念は、単純にデフォルト接続を使用するサーバー側内部ドライバには関係せず、単一のセッション内のサーバー側 Thin ドライバのみに関係します。

接続キャッシュの概要

各接続キャッシュは、接続キャッシュ・クラスのインスタンスによって表されます。また接続キャッシュには、対応付けられたプーリングされた接続インスタンスのグループがあります。単一接続キャッシュ・インスタンスの場合、対応付けられたプーリングされた接続インスタンスは、すべて同じデータベースとスキーマへの物理接続を表す必要があります。プーリングされた接続インスタンスは、接続が要求された場合や、接続キャッシュに未使用のプーリングされた接続インスタンスがない場合に、必要に応じて作成されます。未使用のプーリングされた接続インスタンスは、現在対応付けられている論理接続インスタンスがない、つまり、その物理接続が使用されていないプーリングされた接続インスタンスです。

接続キャッシュの設定の基本

接続キャッシュの設定では、必要に応じて中間層で接続キャッシュ・クラスのインスタンスを作成し、`serverName`、`databaseName` または `URL` などのデータ・ソース接続プロパティを設定します。接続キャッシュ・クラスは、データ・ソース・クラスを拡張することに注意してください。データ・ソース・プロパティについては、14-4 ページの「データ・ソース・プロパティ」を参照してください。

接続キャッシュ・クラスの例には、`OracleConnectionCacheImpl` があります。このクラスをインスタンス化し、その接続プロパティを設定する方法については、14-23 ページの「OracleConnectionCacheImpl のインスタンス化とプロパティの設定」で説明します。このクラスは、`OracleDataSource` クラスを拡張するため、接続するデータベースを指定するための接続プロパティを設定する set メソッドも含まれます。キャッシュ内のすべてのプーリングされた接続インスタンスは、この同じデータベースおよび実際には同じスキーマへの物理接続を表します。

中間層で接続キャッシュ・インスタンスが作成されると、14-8 ページの「データ・ソースの登録」で説明したデータ・ソース・インスタンスと同様に、オプションとしてこのインスタンスを JNDI にバインドできます。

接続キャッシュのアクセスの基本

JDBC アプリケーションは、キャッシュを使用するために、接続キャッシュ・インスタンスを取り出す必要があります。これは、一般に JNDI lookup を使用して、中間層を通じて行われます。接続キャッシュのシナリオでは、JNDI lookup は汎用データ・ソース・インスタンスのかわりに接続キャッシュ・インスタンスを戻します。接続キャッシュ・クラスはデータ・ソース・クラスを拡張するため、接続キャッシュ・インスタンスにはデータ・ソースの機能が含まれます。

JNDI lookup の実行については、14-11 ページの「接続のオープン」で説明しています。

JNDI を使用しない場合は、中間層には一般にアプリケーション用に接続キャッシュ・インスタンスを取り出す場合に使用するいくつかのベンダー固有 API があります。

接続のオープンの基本

プーリングされた接続クラスと同様、接続キャッシュ・クラスには `getConnection()` メソッドがあります。接続キャッシュ・インスタンスの `getConnection()` メソッドは、そのキャッシュと対応付けられたデータベースとスキーマへの論理接続を戻します。この対応付けは、一般に中間層によって設定される接続キャッシュ・インスタンスの接続プロパティで行います。

接続キャッシュのシナリオで、JDBC アプリケーションがデータベースへの接続を要求すると、常にそのデータベースと対応付けられた接続キャッシュ・インスタンスの `getConnection()` メソッドがコールされます。

この `getConnection()` メソッドは、キャッシュに未使用のプーリングされた接続インスタンスが存在するかどうかをチェックします。存在しない場合には、そのインスタンスを作成します。次に、以前に存在していた、または新規に作成されたプーリングされた接続インスタンスから論理接続インスタンスを取り出し、この論理接続インスタンスがアプリケーションに提供されます。

接続のクローズの基本 : 接続イベントの使用

JDBC は、JavaBeans 形式のイベントを使用して、物理接続（プーリングされた接続インスタンス）をキャッシュに戻すことができる時期、または致命的エラーのためにクローズする時期を追跡します。JDBC アプリケーションが論理接続インスタンスの `close()` メソッドをコールする場合は、イベントがトリガーされ、その論理接続インスタンスを作成したプーリングされた接続インスタンスと対応付けられた 1 つ以上のイベント・リスナーに伝えられます。これによって、接続クローズ・イベントがトリガーされ、物理接続を再利用できることがプーリングされた接続インスタンスに通知されます。これによってプーリングされた接続インスタンスとその物理接続はキャッシュに戻ります。

接続イベント・リスナーが作成され、プーリングされた接続インスタンスで登録される時点は、実装によって異なります。たとえば、プーリングされた接続インスタンスが初めて作成されたとき、または対応付けられた論理接続がクローズされるたびにこの処理が発生することがあります。

また、キャッシュ・クラスによって、接続イベント・リスナー・クラスが実装されることもあります。この場合、接続イベント・リスナーは、接続キャッシュ・インスタンスの一部になります。（これは Oracle サンプル実装の場合には当てはまりません。）ただし、この場合でも接続イベント・リスナーとプーリングされた各接続インスタンス間で明示的な対応付けを行う必要があります。

実装シナリオ

中間層開発者は、独自の接続キャッシュ・クラスと接続イベント・リスナー・クラスを実装できます。

ただし、便宜上、oracle.jdbc.pool パッケージには次のものがすべて用意されています。

- 接続キャッシュ・インタフェース：OracleConnectionCache
- 接続キャッシュ・クラス：OracleConnectionCacheImpl
- 接続イベント・リスナー・クラス：OracleConnectionEventListener

OracleConnectionCacheImpl クラスは、例として Oracle が提供する単純な接続キャッシュ・クラス実装で、十分かつ最低限の機能を提供します。このクラスは OracleConnectionCache インタフェースを実装し、接続イベントのために、OracleConnectionEventListener クラスのインスタンスを使用します。

OracleConnectionCacheImpl が提供するよりも多くの機能が必要で、接続イベントのために OracleConnectionEventListener を使用する場合は、OracleConnectionCache を実装する独自のクラスを作成できます。

または、最初から独自の接続キャッシュ・クラスと接続イベント・リスナーを作成することもできます。

接続キャッシュの使用に関する一般的な手順

この項では、論理接続のオープンとクローズのときに、JDBC アプリケーションと中間層が接続キャッシュを使用する際の、一般的な手順を示します。

接続キャッシュに関する予備手順

次のことがすでに終了していると仮定します。

1. 中間層では、14-17 ページの「接続キャッシュの設定の基本」の説明に従って、接続キャッシュ・インスタンスを作成します。
2. 接続キャッシュ・インスタンスに対して、使用するデータベースとスキーマの接続情報が中間層から提供されます。これは接続キャッシュ・インスタンスを作成したとき実行されます。
3. アプリケーションは、14-17 ページの「接続キャッシュのアクセスの基本」の説明に従って、接続キャッシュ・インスタンスを取り出します。

接続のオープンに関する一般的な手順

JDBC アプリケーションが接続キャッシュ・インスタンスにアクセスすると、アプリケーションと中間層では、アプリケーションが使用する論理接続インスタンスを作成するために、次の手順が実行されます。

1. アプリケーションは、接続キャッシュ・インスタンスの `getConnection()` メソッドで接続を要求します。接続キャッシュ・インスタンスはすでに特定のデータベースおよびスキーマと対応付けられているため、入力は必要ありません。

2. 接続キャッシュ・インスタンスは、次のようにそのキャッシュを検査します。
   a）キャッシュ内にプーリングされた接続インスタンスがあるか確認します。
   b）プーリングされた接続インスタンスがある場合は未使用のものがあるかどうか、つまり、現在論理接続インスタンスが対応付けられていない少なくとも 1 つのプーリングされた接続インスタンスがあるか確認します。

3. 接続キャッシュ・インスタンスは、使用可能なプーリングされた接続インスタンスを選択し、使用可能なインスタンスがない場合は、新しいインスタンスを作成します（これは実装によって異なります）。プーリングされた接続インスタンスの作成時に、接続キャッシュ・インスタンスは、独自の接続プロパティに従って接続プロパティを指定できます。これはプーリングされた接続インスタンスが同じデータベースおよびスキーマと対応付けられるためです。

   注意： 使用可能なプーリングされた接続インスタンスがない場合に何が起きるかは、実装スキームとキャッシュがプーリングされた接続の最大数に制限されているかどうかによります。Oracle サンプル実装の場合については、14-25 ページの「Oracle 実装で、プーリングされた接続を新規に作成するためのスキーム」で説明します。

   ---

4. 状況と実装によって、接続キャッシュ・インスタンスは、接続イベント・リスナー（リスナーと接続キャッシュ・インスタンスを対応付けるプロセス）を作成し、選択したまたは新たに作成したプーリングされた接続インスタンスとリスナーを対応付けます。プーリングされた接続インスタンスとの対応付けは、`PooledConnection` インタフェースによって指定される標準 `addConnectionEventListener()` メソッドをコールすることによって実行されます。このメソッドは、入力として接続イベント・リスナー・インスタンスを取ります。接続キャッシュ・クラスが接続イベント・リスナー・クラスを実装する場合は、`addConnectionEventListener()` メソッドの引数は、`this` オブジェクトになります。

   一部の実装では、接続イベント・リスナーの作成と対応付けは、プーリングされた接続インスタンスが初めて作成されたときにのみ発生する可能性があります。Oracle 実装サンプルでは、これはプーリングされた接続インスタンスが再利用されるたびに発生します。

   接続キャッシュ・インスタンスとプーリングされた接続インスタンスの両方の対応付けに関して、接続イベント・リスナーがブリッジとなることに注意してください。

5. 接続キャッシュ・インスタンスは、プーリングされた接続 `getConnection()` メソッドを使用して、選択したまたは新たに作成したプーリングされた接続インスタンスから、論理接続インスタンスを取得します。

   プーリングされた接続インスタンスはすでに特定のデータベースおよびスキーマと対応付けられているため、`getConnection()` への入力は必要ありません。

6. 接続キャッシュ・インスタンスは、論理接続インスタンスをアプリケーションに渡します。

JDBC アプリケーションは、他の接続インスタンスと同様に、この論理接続インスタンスを使用します。

接続のクローズに関する一般的な手順

JDBC アプリケーションが論理接続インスタンスの使用を終了すると、対応付けられたプーリングされた接続インスタンスは、接続キャッシュに戻されることがあります（または、致命的なエラーが発生した場合は、必要に応じてクローズされます）。アプリケーションと中間層は、このために次の手順を実行します。

1. アプリケーションは（他の接続インスタンスの場合と同様に）論理接続インスタンスで `close()` メソッドをコールします。
2. 論理接続インスタンスを作成したプーリングされた接続インスタンスは、対応付けられた 1 つ以上の接続イベント・リスナーに対してイベントをトリガーします（接続キャッシュ・インスタンスが、プーリングされた接続インスタンスの `addConnectionEventListener()` メソッドに対して以前に実行したコールによって対応付けられます）。
3. 接続イベント・リスナーは、次のいずれかを実行します。

   - プーリングされた接続インスタンスをキャッシュに戻して、使用可能のフラグを付けます（通常の場合）。

   または

   - プーリングされた接続インスタンスをクローズします（物理接続の使用中に致命的エラーが発生した場合）。

   接続イベント・リスナーは、通常、接続キャッシュ・インスタンスのコール・メソッドによってこれらの処理を行います。Oracle サンプル実装では、14-22 ページの「Oracle 接続キャッシュ仕様 : OracleConnectionCache インタフェース」で説明するように、`OracleConnectionCache` インタフェースで指定するメソッドによってこれらの機能が実行されます。

4. 状況と実装によって、接続キャッシュ・インスタンスはプーリングされた接続インスタンスから接続イベント・リスナーの対応付けを解除します。これは、PooledConnection インタフェースによって指定される標準 removeConnectionEventListener() メソッドをコールすることによって実行されます。

   一部の実装では、致命的エラーが発生したり、物理接続とともにアプリケーションが終了したために、プーリングされた接続インスタンスがクローズされた場合に限り、この処理を行えます。ただし、Oracle 実装サンプルでは、プーリングされた接続が使用可能なキャッシュに戻されるたびに、接続イベント・リスナーとプーリングされた接続インスタンスの対応付けが解除されます（Oracle 実装では、接続イベント・リスナーは再利用されるたびにプーリングされた接続インスタンスと対応付けられるためです）。

Oracle 接続キャッシュ仕様 : OracleConnectionCache インタフェース

中間層開発者は、独自の接続キャッシュ・スキームを任意に実装できますが、Oracle は接続キャッシュ・クラスで実装できる OracleConnectionCache インタフェースを提供しています。このインスタンスはリスナー機能のために、OracleConnectionEventListener クラスのインスタンスを使用します。

また、Oracle は OracleConnectionCache インタフェースを実装する OracleConnectionCacheImpl というクラスを提供しています。このクラスは、OracleDataSource クラスを拡張するため、getConnection() メソッドが含まれます。このクラスの詳細は、14-23 ページの「Oracle 接続キャッシュ実装 : OracleConnectionCacheImpl クラス」を参照してください。

これらの Oracle クラスとインタフェースはすべて oracle.jdbc.pool パッケージ内にあります。

OracleConnectionCache インタフェースは、接続キャッシュ・クラスで実装される次のメソッドを（継承されるデータ・ソース・メソッドに加えて）指定します。

- reusePooledConnection(): プーリングされた接続インスタンスを入力として取り、使用可能なプーリングされた接続（実際は使用可能な物理接続）のキャッシュに戻します。

  このメソッドは、（プーリングされた接続 getConnection() メソッドの以前の使用を通じて）プーリングされた接続インスタンスによって提供される論理接続インスタンスを使用して、JDBC アプリケーションが終了した後に、接続イベント・リスナーによって実行されます。

- `closePooledConnection()`: プーリングされた接続インスタンスを入力として取り、クローズします。

  接続イベント・リスナーは、プーリングされた接続インスタンスによって提供される論理接続インスタンスで致命的エラーが発生した後に、このメソッドを実行します。リスナーは、たとえばサーバーのクラッシュに気づいた場合に、`closePooledConnection()` をコールします。

- `close()`: アプリケーションが対応付けられたデータベースで接続キャッシュの使用を終了した後に、接続キャッシュ・インスタンスをクローズします。

`reusePooledConnection()` メソッドと `closePooledConnection()` メソッドの機能は、14-21 ページの「接続のクローズに関する一般的な手順」で概要が説明されている一部の処理の実装です。

Oracle 接続キャッシュ実装 : OracleConnectionCacheImpl クラス

Oracle は `OracleConnectionCacheImpl` クラスによって、接続キャッシュおよび接続イベント・リスナーのサンプル実装を提供しています。このクラスは（一部の他の接続キャッシュ・クラスでオプションとして実装できる）`OracleConnectionCache` インタフェースを実装し、リスナー機能のために `OracleConnectionEventListener` クラスのインスタンスを使用します。

これらの Oracle クラスとインタフェースはすべて `oracle.jdbc.pool` パッケージ内にあります。

接続キャッシュ機能のために `OracleConnectionCacheImpl` クラスを使用する場合は、次に説明するトピックを十分に理解する必要があります。

- OracleConnectionCacheImpl のインスタンス化とプロパティの設定
- プーリングされた接続の最大数の設定
- プーリングされた接続の最小数の設定
- Oracle 実装で、プーリングされた接続を新規に作成するためのスキーム
- その他の OracleConnectionCacheImpl メソッド

OracleConnectionCacheImpl のインスタンス化とプロパティの設定

Oracle による接続キャッシュの実装を使用する中間層では、次の 3 つの方法のいずれかを使用して `OracleConnectionCacheImpl` インスタンスを作成し、その接続プロパティを設定できます。

- 入力として既存の接続プーリング・データ・ソースを取る `OracleConnectionCacheImpl` コンストラクタを使用できます。これは中間層ですでに接続プーリング・データ・ソース・インスタンスを作成し、その接続プロパティが設定されている場合に便利です。たとえば、`cpds` が接続プーリング・データ・ソース・インスタンスの場合は、次のようになります。

  ```
  OracleConnectionCacheImpl ocacheimpl = new OracleConnectionCacheImpl(cpds);
  ```

または

- （入力を取らない）デフォルトの OracleConnectionCacheImpl コンストラクタと、入力として既存の接続プーリング・データ・ソース・インスタンスを取る setConnectionPoolDataSource() メソッドを使用できます。これも中間層ですでに接続プーリング・データ・ソース・インスタンスがあり、その接続プロパティが設定されている場合に便利です。たとえば、cpds が接続プーリング・データ・ソース・インスタンスの場合は、次のようになります。

```
OracleConnectionCacheImpl ocacheimpl = new OracleConnectionCacheImpl();

ocacheimpl.setConnectionPoolDataSource(cpds);
```

注意：

- setConnectionPoolDataSource() メソッドを使用して、以前に設定したプーリングされた接続データ・ソースまたは以前に設定した接続プロパティをオーバーライドすることもできます。
- 使用中の対応付けられた論理接続を伴う接続プーリング・データ・ソースがすでにある場合に setConnectionPoolDataSource() をコールすると、新しい接続プーリング・データ・ソースが以前の接続プーリング・データ・ソースとは異なるデータ・スキーマを指定している場合は、例外が発生します。

または

- デフォルトの OracleConnectionCacheImpl コンストラクタを使用し、setter メソッドによって個々にプロパティを設定できます。たとえば、次のようになります。

```
OracleConnectionCacheImpl ocacheimpl = new OracleConnectionCacheImpl();

ocacheimpl.setDriverType("oci8");
ocacheimpl.setServerName("dlsun999");
ocacheimpl.setNetworkProtocol("tcp");
ocacheimpl.setDatabaseName("816");
ocacheimpl.setPortNumber(1521);
ocacheimpl.setUser("scott");
ocacheimpl.setPassword("tiger");
```

これは 14-8 ページの「接続プロパティの初期化」で説明した、汎用データ・ソースまたは接続プーリング・データ・ソースでのプロパティの設定に相当します。

プーリングされた接続の最大数の設定

すべての接続キャッシュ実装で、中間層開発者は、キャッシュ内のプーリングされた接続の最大数を設定する必要があるかどうかと、使用可能なプーリングされた接続がなく最大数に達している場合の処理方法とを決定する必要があります。

OracleConnectionCacheImpl クラスには、(入力として int を取得する) setMaxLimit() メソッドを使用して設定できる最大キャッシュ・サイズが含まれます。デフォルト値は 1 です。

次に、ocacheimpl が OracleConnectionCacheImpl インスタンスであると仮定した場合の例を示します。

```
ocacheimpl.setMaxLimit(10);
```

この例では、キャッシュのサイズを最大 10 のプーリングされた接続インスタンスに制限します。

プーリングされた接続の最小数の設定

中間層開発者がプーリングされた接続の最大数を設定できるように、キャッシュ内の事前生成済みのプーリングされた接続の最小数を設定する必要があるかどうかを決定できます。ここで設定した最小数は、引数として setMinLimit() メソッドに渡されます。キャッシュに指定された数のプーリングされた接続インスタンスがない場合、キャッシュは指定された最小数を超えない数の新しいスプーリングされた接続インスタンスを作成します。キャッシュでは、アクティブ状態またはアイドル状態に関係なく、常に最低限の数のプーリングされた接続をオープン状態に保ちます。

次に、ocacheimpl は、OracleConnectionCacheImpl インスタンスであると仮定した場合の例を示します。

```
ocacheimpl.setMinLimit(3);
```

この例では、キャッシュには常に最低 3 つのプーリングされた接続インスタンスが存在します。

Oracle 実装で、プーリングされた接続を新規に作成するためのスキーム

OracleConnectionCacheImpl クラスは、3 つの接続キャッシュ・スキームをサポートします。これらのスキームは、次の状況で使用します。

(1) アプリケーションが接続を要求した場合
(2) プーリングされた既存の接続がすべて使用中の場合
(3) キャッシュ内のプーリングされた接続の最大数に達した場合

- 動的

  このデフォルト・スキームでは、プーリングされた新規接続を上限数を超えて作成できますが、提供された論理接続インスタンスが使用されなくなった直後、プーリングされたそれぞれの接続は自動的にクローズされ、解放されます。（これは、プーリングされた接続インスタンスの使用が終了した場合に、使用可能なキャッシュに戻される通常のシナリオとは異なります。）

- 待機なし固定

  このスキームでは、接続数の上限は超えることはできません。最大値にすでに達している場合に接続を要求すると、NULL が戻されます。

- 固定待機

  「待機なし固定」スキームと同じですが、新しい接続の要求は、接続数の上限に達すると待機するという点が異なります。この場合、別のクライアントが接続を解放するまで接続要求は待機し続けます。

OracleConnectionCacheImpl インスタンスの setCacheScheme() メソッドを実行して、前述の 3 つに対応したキャッシュ・スキームを設定します。入力として次のクラス静的定数のいずれかを使用します。

- DYNAMIC_SCHEME
- FIXED_RETURN_NULL_SCHEME
- FIXED_WAIT_SCHEME

たとえば、ocacheimpl は、OracleConnectionCacheImpl インスタンスであると仮定します。

```
ocacheimpl.setCacheScheme(OracleConnectionCacheImpl.FIXED_RETURN_NULL_SCHEME);
```

各スキームの例は、「サンプル・アプリケーション」の章で参照できます。20-98 ページの「Oracle 接続キャッシュ（動的）: CCache1.java」および 20-100 ページの「Oracle 接続キャッシュ（待機なし固定）: CCache2.java」を参照してください。

その他の OracleConnectionCacheImpl メソッド

14-22 ページの「Oracle 接続キャッシュ仕様 : OracleConnectionCache インタフェース」ですでに説明した主なメソッドに加えて、次の OracleConnectionCacheImpl メソッドが役に立つことがあります。

- getActiveSize(): キャッシュ内で現在アクティブなプーリングされた接続（対応付けられた論理接続インスタンスが、JDBC アプリケーションによって使用されているプーリングされた接続インスタンス）の数を戻します。
- getCacheSize(): キャッシュ内のアクティブおよび非アクティブなプーリングされた接続の総数を戻します。

Oracle 接続イベント・リスナー : OracleConnectionEventListener クラス

この項では、コンストラクタとメソッドの概要を示して、OracleConnectionEventListener の機能を説明します。

Oracle 接続イベント・リスナーのインスタンス化

接続キャッシュの Oracle 実装では、OracleConnectionCacheImpl インスタンスは、コンストラクタ引数として接続キャッシュ・インスタンス自体（その this インスタンス）を指定して、Oracle 接続イベント・リスナーを作成します。このインスタンスは、接続イベント・リスナーを接続キャッシュ・インスタンスに関連付けます。

ただし、一般に OracleConnectionEventListener コンストラクタは、入力として任意のデータ・ソース・インスタンスを取れます。たとえば、ds が汎用データ・ソースの場合は、次のようになります。

```
OracleConnectionEventListener ocel = new OracleConnectionEventListener(ds);
```

また、入力を取らず OracleConnectionEventListener クラスの setDataSource() メソッドとともに使用できるデフォルトのコンストラクタもあります。

```
OracleConnectionEventListener ocel = new OracleConnectionEventListener();
...
ocel.setDataSource(ds);
```

OracleConnectionCacheImpl インスタンスを含む、任意の種類のデータ・ソースを入力できます（このクラスが OracleDataSource を拡張するためです）。

Oracle 接続イベント・リスナー・メソッド

OracleConnectionEventListener クラスのメソッドについて概要を示します。

- setDataSource()（前述）: リスナーの作成時に提供されていない場合に、接続イベント・リスナーにデータ・ソースを入力するために使用します。これは入力として任意の種類のデータ・ソースを取ります。
- connectionClosed(): JDBC アプリケーションがその接続の表現上で close() をコールした場合に実行されます。
- connectionErrorOccurred(): SQLException がアプリケーションに対して発行される直前に、致命的接続エラーが発生した場合に実行されます。

15

分散トランザクション

この章では、分散トランザクションの Oracle JDBC 実装について説明します。分散トランザクションとは、協調してコミットされる必要があるマルチ・フェーズ・トランザクションで、多くの場合、複数のデータベースを使用します。関連する XA についても説明します。XA とは、分散トランザクションの汎用標準です（Java に限定されません）。

次の項目が含まれます。

- 概要
- XA コンポーネント
- エラー処理と最適化
- 分散トランザクションの実装

> **注意：** この章では、JDBC 2.0 Optional Package の機能を説明します。これは、以前、JDBC 2.0 Standard Extension API と呼ばれていたもので、Sun 社の `javax` パッケージを通じて使用できます。Optional Package は標準 JDK の一部ではありませんが、関連するパッケージが Oracle JDBC `classes111.zip` および `classes12.zip` ファイルとともに含まれています。

分散トランザクションの詳細および一般情報は、Sun 社の JDBC 2.0 Optional Package および Java Transaction API（JTA）の仕様を参照してください。

OCI 固有の HeteroRM XA 機能の詳細は、16-17 ページの「OCI HeteroRM XA」を参照してください。

概要

分散トランザクションは、グローバル・トランザクションと呼ばれることがあります。これは、関連のある複数のトランザクションの集合で、協調して管理される必要があります。分散トランザクションを構成するトランザクションは、同じデータベースを対象にすることもありますが、一般的には、異なる場所の異なるデータベースを対象にします。分散トランザクションの個々のトランザクションを、トランザクション・ブランチと呼びます。

たとえば、分散トランザクションには、ある銀行のある口座から、別の銀行の口座への送金があります。両方の処理が正常に完了する保証がなければ、トランザクションはコミットできません。

JDBC 2.0 Extension API では、分散トランザクション機能が接続プーリング機能の最上位に構築されます。接続プーリング機能の詳細は、14-12 ページの「接続プーリング」を参照してください。この分散トランザクション機能は、分散トランザクションのオープン XA 標準にも基づいて構築されます。（XA は X/Open 標準の一部で、Java 固有ではありません。）

この概要の残りの部分では、次のトピックを取り上げます。

- 分散トランザクションのコンポーネントおよびシナリオ
- 分散トランザクションの概念
- Oracle XA パッケージ

分散トランザクションと XA の詳細および一般情報は、JDBC 2.0 Optional Package および Java Transaction API の Sun Microsystems 仕様を参照してください。

分散トランザクションのコンポーネントおよびシナリオ

分散トランザクションの項の残りの部分を読むとき、次のポイントを覚えておくと役に立ちます。

- 分散トランザクション・システムは、通常、標準 Java Transaction API 機能を実装するソフトウェア・コンポーネントなど、外部のトランザクション・マネージャを使用して、個々のトランザクションを協調します。

  多くのベンダーが、将来、XA 準拠の JTA モジュールを提供します。このベンダーには、Oracle も含まれます。後で説明する XA の Oracle 実装に基づいた JTA モジュールを開発中です。

- XA 機能は、通常、クライアント・アプリケーションから孤立しています。クライアント・アプリケーションではなく、アプリケーション・サーバーなど、中間層環境に実装されます。

  多くの場合、アプリケーション・サーバーとトランザクション・マネージャはともに中間層に位置します。同時に、アプリケーション・コードの一部も中間層に位置します。

- この項の説明は、主に、中間層の開発者を対象にしています。

- 分散トランザクションの説明では、リソース・マネージャという用語が頻繁に使用されます。リソース・マネージャとは、単に、データまたはその他のリソースを管理するエンティティです。この章で使用された場合、データベースを指します。

注意： JTA 機能を使用するには、CLASSPATH にファイル jta.jar を含める必要があります。（このファイルは、$ORACLE_HOME/jlib に格納されています。）このファイルは、Oracle の JDBC 製品に含まれています。（Sun 社の Web サイトから取得することもできますが、オラクル社が提供するバージョンを使用することをお薦めします。このバージョンは、Oracle ドライバでのテストが済んでいます。）

分散トランザクションの概念

分散トランザクションを使用するソフトウェアは、通常の接続インスタンスの COMMIT、自動コミットまたは ROLLBACK 機能を使用できません。分散トランザクションの COMMIT または ROLLBACK 操作はすべて、協調する必要があります。接続インスタンスの commit() または rollback() メソッドを使用しようとしたり、自動コミットを有効にすると、SQL 例外が発生します。

XA 機能を使用する場合、トランザクション・マネージャが、XA リソース・インスタンスを使用して各トランザクション・ブランチを準備し、協調させ、すべてのトランザクション・ブランチを適切にコミットまたはロールバックします。

XA 機能には、次のキー・コンポーネントが含まれています。

- XA データ・ソース: 接続プーリング・データ・ソースおよびその他のデータ・ソースの拡張要素で、概念および機能の面で似ています。

  分散トランザクションで使用されるリソース・マネージャ（データベース）ごとに 1 つの XA データ・ソース・インスタンスがあります。通常、XA データ・ソース・インスタンスは、（クラス・コンストラクタを使用して）中間層ソフトウェアで作成します。

  XA データ・ソースは、XA 接続を作成します。

- XA 接続: プーリングされた接続の拡張要素で、概念および機能の面で似ています。XA 接続は、物理的なデータベース接続をカプセル化します。個々の接続インスタンスは、これら物理接続の一時的なハンドルです。

  XA 接続インスタンスは、1 つの Oracle セッションに対応します。ただし、1 つのセッションは、プーリングされた接続インスタンスのように、複数の論理接続インスタンスで順に（一度に 1 つずつ）使用できます。

  通常、XA 接続インスタンスは、XA データ・ソース・インスタンスから、（get メソッドを使用して）中間層ソフトウェアで取得します。分散トランザクションが、同じデータベースの複数セッション（複数の物理接続）にかかわる場合、単一 XA データ・ソース・インスタンスから複数の XA 接続インスタンスを取得できます。

  XA 接続は、XA リソース・インスタンスおよび JDBC 接続インスタンスを作成します。

- XA リソース : 分散トランザクションのトランザクション・ブランチを協調させるために、トランザクション・マネージャで使用されます。

  通常、XA リソース・インスタンスは、各 XA 接続インスタンスから 1 つずつ、（get メソッドを使用して）中間層ソフトウェアで取得します。XA リソース・インスタンスと XA 接続インスタンスの間には、1 対 1 の相関関係があります。同様に、XA リソース・インスタンスと Oracle セッション（物理接続）の間には、1 対 1 の相関関係があります。

  一般的なシナリオでは、中間層コンポーネントが XA リソース・インスタンスをトランザクション・マネージャに渡します。トランザクション・マネージャは、これを使用して分散トランザクションを協調させます。

  各 XA リソース・インスタンスは 1 つの Oracle セッションに対応するので、XA リソース・インスタンスに対応付けられたトランザクション・ブランチのうち、同時にアクティブになれるものは 1 つのみです。ただし、他のトランザクション・ブランチを保留しておくことはできます。15-9 ページの「XA リソース・メソッドの機能および入力パラメータ」を参照してください。

  各 XA リソース・インスタンスには、その XA リソース・インスタンスに対応付けられたセッションで実行中のトランザクション・ブランチの操作を、開始、終了、準備、コミットまたはロールバックする機能があります。

  準備処理は、2 フェーズ・コミット操作の最初の処理です。トランザクション・マネージャは、各 XA リソース・インスタンスに `prepare` を発行します。トランザクション・マネージャは、各トランザクション・ブランチの操作が正常に準備されたことを確認すると（本質的に言えば、データベースにエラーなしでアクセスできると）、各 XA リソース・インスタンスに `COMMIT` を発行し、すべての変更をコミットします。

- トランザクション ID: トランザクション・ブランチを識別するために使用されます。各 ID には、トランザクション・ブランチ ID コンポーネントと分散トランザクション ID コンポーネントが含まれます。これにより、ブランチが分散トランザクションに対応付けられます。ある分散トランザクションに対応付けられた XA リソース・インスタンスには、すべて、同じ分散トランザクション ID コンポーネントが含まれているトランザクション ID が設定されます。

Oracle XA パッケージ

Oracle からは、XA 標準に従い分散トランザクション機能を実装するクラスを含む、次の 3 つのパッケージが提供されます。

- `oracle.jdbc.xa`（`OracleXid` および `OracleXAException` クラス）
- `oracle.jdbc.xa.client`
- `oracle.jdbc.xa.server`

XA データ・ソース、XA 接続および XA リソースのクラスは、`client` パッケージと `server` パッケージの両方に含まれています。（それぞれの抽象クラスは、最上位のパッケージに含まれています。）`OracleXid` および `OracleXAException` クラスは、最上位の `oracle.jdbc.xa` パッケージに含まれています。これらの機能は、コードが実行される場所に依存しません。

中間層のシナリオでは、`OracleXid`、`OracleXAException` および `oracle.jdbc.xa.client` パッケージをインポートします。

ただし、XA コードをターゲット Oracle データベースで実行する場合は、`oracle.jdbc.xa.client` パッケージのかわりに、`oracle.jdbc.xa.server` パッケージをインポートします。

ターゲット・データベースの内部で実行するコードでリモート・データベースにもアクセスする必要がある場合、どちらのパッケージもインポートしません。かわりに、`client` パッケージから使用するクラス（リモート・データベースにアクセスするとき）または `server` パッケージから使用するクラス（ローカル・データベースにアクセスするとき）の名前を完全に修飾する必要があります。クラス名は、これらのパッケージで重複しています。

XA コンポーネント

この項では、XA コンポーネントを説明します。JDBC 2.0 Optional Package で指定されている標準 XA インタフェースおよびこのインタフェースを実装する Oracle クラスについて説明します。次の項目があります。

- XA データ・ソース・インタフェースと Oracle 実装
- XA 接続インタフェースと Oracle 実装
- XA リソース・インタフェースと Oracle 実装
- XA リソース・メソッドの機能および入力パラメータ
- XA ID インタフェースと Oracle 実装

XA データ・ソース・インタフェースと Oracle 実装

`javax.sql.XADataSource` インタフェースは、XA 接続のファクトリである XA データ・ソースの標準機能のアウトラインを構成します。オーバーロードされた `getXAConnection()` メソッドは、XA 接続インスタンスを戻します。オプションで、入力としてユーザー名とパスワードを取ることもあります。

```
public interface XADataSource
{
   XAConnection getXAConnection() throws SQLException;
   XAConnection getXAConnection(String user, String password)
      throws SQLException;
   ...
}
```

Oracle JDBC は、`XADataSource` インタフェースを `OracleXADataSource` クラスで実装します。どちらも、`oracle.jdbc.xa.client` パッケージおよび `oracle.jdbc.xa.server` パッケージに含まれています。

`OracleXADataSource` クラスは、`OracleConnectionPoolDataSource` クラス（`OracleDataSource` クラスの拡張）の拡張も行います。そのため、14-4 ページの「データ・ソース・プロパティ」で説明したすべての接続プロパティも継承されます。

`OracleXADataSource` クラスの `getXAConnection()` メソッドは、XA 接続インスタンスの Oracle 実装を戻します。このインスタンスは、`OracleXAConnection` インスタンスです（次の項で説明します）。

注意： 14-8 ページの「データ・ソースの登録」で説明した非プーリング・データ・ソースの場合と同じネーミング規則を使用して、XA データ・ソースを JNDI に登録できます。

XA 接続インタフェースと Oracle 実装

XA 接続インスタンスは、プーリングされた接続インスタンスと同様に、データベースへの物理接続をカプセル化します。このデータベースは、その XA 接続インスタンスを作成した XA データ・ソース・インスタンスの接続プロパティで指定されたデータベースです。

各 XA 接続インスタンスには、対応する XA リソース・インスタンスを作成する機能もあります。XA リソース・インスタンスは、分散トランザクションを協調させるために使用されます。

XA 接続インスタンスは、標準 `javax.sql.XAConnection` インタフェースを実装するクラスのインスタンスです。

```
public interface XAConnection extends PooledConnection
{
   javax.jta.xa.XAResource getXAResource() throws SQLException;
}
```

すでに説明したとおり、`XAConnection` インタフェースは `javax.sql.PooledConnection` インタフェースを拡張します。そのため、14-14 ページの「プーリングされた接続インタフェースと Oracle 実装」で説明する `getConnection()`、`close()`、`addConnectionEventListener()` および `removeConnectionEventListener()` メソッドも継承されます。

Oracle JDBC は、`XAConnection` インタフェースを `OracleXAConnection` クラスで実装します。どちらも、`oracle.jdbc.xa.client` パッケージおよび `oracle.jdbc.xa.server` パッケージに含まれています。

`OracleXAConnection` クラスは、`OraclePooledConnection` クラスも拡張します。

`OracleXAConnection` クラスの `getXAResource()` メソッドは、XA リソース・インスタンスの Oracle 実装を戻します。このインスタンスは、`OracleXAResource` インスタンスです（次の項で説明します）。`getConnection()` メソッドは、`OracleConnection` インスタンスを戻します。

XA 接続インスタンスから戻される JDBC 接続インスタンスは、物理接続のカプセル化ではなく、物理接続への一時的なハンドルとして機能します。物理接続は、XA 接続インスタンスでカプセル化されます。

XA 接続インスタンスの `getConnection()` メソッドがコールされるたびに、デフォルト動作を行う新しい接続インスタンスが戻されます。また、同じ XA 接続インスタンスから戻されて、まだ存在している前の接続インスタンスをすべてクローズします。ただし、前の接続インスタンスは、新しい接続インスタンスをオープンする前に、明示的にクローズすることをお薦めします。

XA 接続インスタンスの `close()` メソッドをコールすると、データベースへの物理接続がクローズされます。これは、通常、中間層で実行します。

XA リソース・インタフェースと Oracle 実装

トランザクション・マネージャは XA リソース・インスタンスを使用して、分散トランザクションを構成するすべてのトランザクション・ブランチを協調させます。

各 XA リソース・インスタンスは、次の主要な機能を提供します。通常、トランザクション・マネージャから起動されます。

- 分散トランザクションと、この XA リソース・インスタンスを作成した XA 接続インスタンスで動作するトランザクション・ブランチの、対応付けおよび対応付けの解除を行います。(基本的には、分散トランザクションを物理接続または XA 接続インスタンスでカプセル化されたセッションに対応付けます。) これは、トランザクション ID を使用して実行されます。
- 分散トランザクションの 2 フェーズ・コミット機能を実行します。すべてのトランザクション・ブランチで正常に処理されることが保証されるまで、その変更がどれか 1 つのトランザクション・ブランチでコミットされることはありません。

詳細は、15-9 ページの「XA リソース・メソッドの機能および入力パラメータ」を参照してください。

注意 :

- XA 接続インスタンスと XA リソース・インスタンスの間には常に 1 対 1 の相関関係があるので、対応付けられた XA 接続インスタンスがクローズされると、XA リソース・インスタンスは暗黙的にクローズされます。
- ある XA リソース・インスタンスがトランザクションをオープンした場合、そのトランザクションは同じ XA リソース・インスタンスによってクローズする必要があります。

XA リソース・インスタンスは、標準 `javax.transaction.xa.XAResource` インタフェースを実装するクラスのインスタンスです。

```
public interface XAResource
{
   void commit(Xid xid, boolean onePhase) throws XAException;
   void end(Xid xid, int flags) throws XAException;
   void forget(Xid xid) throws XAException;
   int prepare(Xid xid) throws XAException;
   Xid[] recover(int flag) throws XAException;
   void rollback(Xid xid) throws XAException;
   void start(Xid xid, int flags) throws XAException;
   boolean isSameRM(XAResource xares) throws XAException;
}
```

Oracle JDBC は、XAResource インタフェースを OracleXAResource クラスで実装します。どちらも、oracle.jdbc.xa.client パッケージおよび oracle.jdbc.xa.server パッケージに含まれています。

Oracle JDBC ドライバは、OracleXAConnection クラスの getXAResource() メソッドがコールされると、OracleXAResource インスタンスを作成して戻します。XA リソース・インスタンスを、接続インスタンスおよびその接続によって実行されるトランザクション・ブランチに対応付けるのも、Oracle JDBC ドライバの役割です。

このメソッドは、特定の接続およびその接続で実行されるトランザクション・ブランチに OracleXAResource インスタンスを関連付けるときのメソッドです。

XA リソース・メソッドの機能および入力パラメータ

OracleXAResource クラスには、トランザクション・ブランチを、対応付けられた分散トランザクションと協調させるためのメソッドがいくつかあります。この機能は、通常、2 フェーズの COMMIT 操作で起動されます。

通常、トランザクション・マネージャが、アプリケーション・サーバーなどの中間層コンポーネントから OracleXAResource インスタンスを受け取り、この機能を起動します。

これらのメソッドは、Xid インスタンスのフォームのトランザクション ID を入力とします。トランザクション ID には、トランザクション・ブランチ ID コンポーネントと分散トランザクション ID コンポーネントが含まれています。各トランザクション・ブランチには一意のトランザクション ID が設定されていますが、同じグローバル・トランザクションに属するトランザクション・ブランチは、トランザクション ID の一部として同じグローバル・トランザクション・コンポーネントを持っています。

OracleXid クラスおよび基本となる標準インタフェースについては、15-13 ページの「XA ID インタフェースと Oracle 実装」を参照してください。

主要な XA リソース機能、使用されるメソッドおよび追加の入力パラメータについて説明します。これらのメソッドでエラーが発生すると、XA 例外が発生します。詳細は、15-14 ページの「XA 例外クラスおよびメソッド」を参照してください。

Start トランザクション・ブランチの処理を開始します。トランザクション・ブランチを分散トランザクションに対応付けます。

```
void start(Xid xid, int flags)
```

flags パラメータには、次の値のうち 1 つを設定できます。

- XAResource.TMNOFLAGS（フラグなし）。この XA リソース・インスタンスに対応付けられたセッションの後続の操作のために、新しいトランザクション・ブランチを起動することを示します。このブランチには、トランザクション ID xid が設定されます。これは、トランザクション・マネージャによって作成された OracleXid インスタンスです。これにより、トランザクション・ブランチは適切な分散トランザクションにマップされます。
- XAResource.TMJOIN。この XA リソース・インスタンスに対応付けられたセッションの後続の操作を、xid で指定した既存のトランザクション・ブランチに結合します。
- XAResource.TMRESUME。xid で指定したトランザクション・ブランチを再開します。（このトランザクション・ブランチは、保留されている必要があります。）

TMNOFLAGS、TMJOIN および TMRESUME は、XAResource インタフェースおよび OracleXAResource クラスの静的メンバーとして定義されています。

注意： TMRESUME を指定して start() メソッドを使用するかわりに、トランザクション・マネージャで OracleXAResource インスタンスにキャストし、Oracle 拡張機能 resume(Xid xid) メソッドを使用することもできます。

トランザクション・ブランチを起動するときに適切なトランザクション ID を作成するには、そのトランザクション・ブランチが属する分散トランザクションを、トランザクション・マネージャに指示する必要があります。このメカニズムは、中間層とトランザクション・マネージャの間で処理されるので、このマニュアルでは扱いません。Sun 社の JDBC 2.0 Optional Package および Java Transaction API の仕様を参照してください。

End xid で指定されたトランザクション・ブランチの処理を終了します。トランザクション・ブランチと分散トランザクションの対応付けを解除します。

```
void end(Xid xid, int flags)
```

flags パラメータには、次の値のうち 1 つを設定できます。

- XAResource.TMSUCCESS。このトランザクション・ブランチが正常であることを示します。
- XAResource.TMFAIL。このトランザクション・ブランチが異常であることを示します。
- XAResource.TMSUSPEND。xid で指定したトランザクション・ブランチを保留することを示します。（トランザクション・ブランチを保留することにより、複数のトランザクション・ブランチを単一セッションで使用できます。ただし、ある時点でアクティブにできるのは、1 つのみです。また、多くの場合、リソースの観点から見て、2 つのセッションを使用するよりも高価です。）

TMSUCCESS、TMFAIL および TMSUSPEND は、XAResource インタフェースおよび OracleXAResource クラスの静的メンバーとして定義されています。

注意：

- TMSUSPEND を指定して end() メソッドを使用するかわりに、トランザクション・マネージャで OracleXAResource インスタンスにキャストし、Oracle 拡張機能 suspend(Xid xid) メソッドを使用することもできます。
- この、トランザクションを保留する XA 機能によって、単一 JDBC 接続で、各種トランザクションを切り替えることができます。分散トランザクション環境になく、XA クラスを必要としない場合でも、この機能を実現するために XA クラスを使用できます。

Prepare xid で指定したトランザクション・ブランチで実行される変更の準備をします。これは、2 フェーズ・コミット操作の最初のフェーズです。データベースがアクセス可能で、変更が正常にコミットされることを確認します。

```
int prepare(Xid xid)
```

このメソッドでは、次の整数値を戻します。

- XAResource.XA_RDONLY: トランザクション・ブランチが、SELECT 文など、読取り専用操作のみを実行する場合に戻されます。
- XAResource.XA_OK: トランザクション・ブランチが更新を実行する場合、すべて準備済みで、エラーがなければ、この値が戻されます。
- n/a（戻り値なし）: トランザクション・ブランチが更新を実行する場合、そのうち 1 つでも準備中にエラーが発生すると、値は戻されません。この場合、XA 例外が発生します。

XA_RDONLY および XA_OK は、XAResource インタフェースおよび OracleXAResource クラスの静的メンバーとして定義されています。

注意：

- prepare() メソッドをコールする前に、必ず、ブランチの end() メソッドをコールする必要があります。
- 分散トランザクションにあるトランザクション・ブランチが 1 つのみの場合、prepare() メソッドをコールする必要はありません。準備なしに、XA リソースの commit() メソッドをコールできます。

Commit xid で指定したトランザクション・ブランチで準備された変更をコミットします。これは、2 フェーズ・コミットの 2 番目のフェーズです。すべてのトランザクション・ブランチが正常に準備された後でのみ実行します。

```
void commit(Xid xid, boolean onePhase)
```

onePhase パラメータは、次のように設定します。

- true: トランザクション・ブランチをコミットするときに、2 フェーズ・プロトコルではなく、1 フェーズ・プロトコルを使用します。分散トランザクションにトランザクション・ブランチが 1 つのみある場合に適しています。prepare 処理はスキップされます。
- false: トランザクション・ブランチをコミットするときに、2 フェーズ・プロトコルを使用します（通常）。

Roll back xid で指定したトランザクション・ブランチで準備された変更をロールバックします。

```
void rollback(Xid xid)
```

Forget リソース・マネージャに、ヒューリスティックに完了したトランザクション・ブランクを破棄するよう通知します。

```
public void forget(Xid xid)
```

Recover トランザクション・マネージャは、リカバリ中にこのメソッドをコールして、現在準備中またはヒューリスティックに完了した状態のトランザクション・ブランチのリストを取得します。

```
public Xid[] recover(int flag)
```

注意： flag パラメータは無視されるため実装されていません。scan オプション（flag パラメータ）は、count パラメータがない場合無効になります。詳細は、Sun 社の Java Transaction API（JTA）仕様を参照してください。

リソース・マネージャは、現在準備中またはヒューリスティックに完了した状態のトランザクション・ブランチに対して、0（ゼロ）以上の Xid を戻します。操作中にエラーが発生した場合は、リソース・マネージャにより適切な XAException が発行されます。

同じ RM のチェック 2 つの XA リソース・インスタンスが同じリソース・マネージャ（データベース）に対応しているかどうかを判断するには、一方の XA リソース・インスタンスから、入力としてもう一方の XA リソース・インスタンスを指定して、`isSameRM()` メソッドをコールします。次の例では、`xares1` および `xares2` は `OracleXAResource` インスタンスとします。

```
boolean sameRM = xares1.isSameRM(xares2);
```

トランザクション・マネージャは特定の Oracle 最適化に関してこのメソッドを使用できます。15-16 ページの「Oracle XA 最適化」を参照してください。

XA ID インタフェースと Oracle 実装

トランザクション・マネージャは、トランザクション ID インスタンスを作成し、これを使用して、分散トランザクションのブランチを協調させます。各トランザクション・ブランチには、一意なトランザクション ID が割り当てられます。トランザクション ID には、次の情報が含まれます。

- フォーマット識別子（4 バイト）

  フォーマット識別子は、Java トランザクション・マネージャを指定します。たとえば、フォーマット識別子 `ORCL` があります。このフィールドは、NULL にはできません。

- グローバル・トランザクション識別子（64 バイト）（前に説明した分散トランザクション ID コンポーネント）
- ブランチ修飾子（64 バイト）（前に説明したトランザクション・ブランチ ID コンポーネント）

64 バイトのグローバル・トランザクション識別子の値は、同じ分散トランザクションに属するトランザクション・ブランチすべてのトランザクション ID で同一です。ただし、トランザクション ID の全体は、トランザクション・ブランチごとに一意です。

XA トランザクション ID インスタンスは、標準 `javax.transaction.xa.Xid` インタフェースを実装するクラスのインスタンスです。このインタフェースは、X/Open トランザクション識別子 XID 構造体の Java マッピングです。

Oracle では、このインタフェースを `oracle.jdbc.xa` パッケージの `OracleXid` クラスに実装しています。`OracleXid` インスタンスは、トランザクション・マネージャでのみ使用されますが、アプリケーション・プログラムまたはアプリケーション・サーバーにとって透過的です。

注意： `OracleXid` は Oracle XA リソースのコールには必要ありません。かわりに、`javax.transaction.xa.Xid` インタフェースを実装するクラスを使用します。

トランザクション・マネージャでは、次のメソッドを使用して、OracleXid インスタンスを作成できます。

```
public OracleXid(int fId, byte gId[], byte bId[]) throws XAException
```

fId は、フォーマット識別子を表す整数値です。gId[] は、グローバル・トランザクション識別子を表すバイト配列です。bId[] は、ブランチ修飾子を表すバイト配列です。

Xid インタフェースでは、次の getter メソッドが指定されています。

- `public int getFormatId()`
- `public byte[] getGlobalTransactionId()`
- `public type[] getBranchQualifier()`

エラー処理と最適化

この項では、次の 2 つの問題を取り上げます。

1) XA 例外の機能とエラー処理。
2) XA 実装での Oracle 最適化。

次の項目があります。

例外とエラー処理の説明には、標準 XA 例外クラスと Oracle 固有の XA 例外クラスが含まれます。また、特定の XA エラー・コードおよびエラー処理の方法も含まれます。

XA 例外クラスおよびメソッド

XA メソッドでは、一般例外や SQL 例外ではなく、XA 例外が発生します。XA 例外は、標準クラス javax.transaction.xa.XAException またはそのサブクラスのインスタンスです。Oracle では、XAException は oracle.jdbc.xa.OracleXAException クラスでサブクラス化されています。

OracleXAException インスタンスは、Oracle エラー部と XA エラー部で構成されています。Oracle JDBC ドライバでは、次のように構築されます。

```
public OracleXAException()
```

または

```
public OracleXAException(int error)
```

エラー値は、Oracle SQL エラー値と XA エラー値を組み合せたエラー・コードです。（Oracle エラー値と XA エラー値を組み合せる正確な方法は、JDBC ドライバによって判断されます。）

OracleXAException クラスには、次のメソッドが含まれます。

- public int getOracleError()

  このメソッドは、例外に含まれる Oracle SQL エラー・コード（標準 ORA エラー番号）を戻します（Oracle SQL エラーがなければ、0 を戻します）。

- public int getXAError()

  このメソッドは、例外に含まれる XA エラー・コードを戻します。XA エラー値は、javax.transaction.xa.XAException クラスで定義されています。詳細は、Sun 社の Web サイトの Javadoc を参照してください。

Oracle エラーと XA エラーのマッピング

表 15-1 で示すように、Oracle エラーは OracleXAException インスタンスの XA エラーに対応しています。

表 15-1 Oracle と XA のエラー・マッピング

Oracle エラー・コード	XA エラー・コード
ORA 3113	XAException.XAER_RMFAIL
ORA 3114	XAException.XAER_RMFAIL
ORA 24756	XAException.XAER_NOTA
ORA 24764	XAException.XA_HEURCOM
ORA 24765	XAException.XA_HEURRB
ORA 24766	XAException.XA_HEURMIX
ORA 24767	XAException.XA_RDONLY
ORA 25351	XAException.XA_RETRY
その他の ORA エラーすべて	XAException.XA_RMERR

XA エラー処理

次の例では、OracleXAException クラスを使用して、XA 例外を処理します。

```
try {
   ...
   ...Perform XA operations...
   ...
} catch(OracleXAException oxae) {
  int oraerr = oxae.getOracleError();
  System.out.println("Error " + oraerr);
}
  catch(XAException xae)
{...Process generic XA exception...}
```

XA 操作によって Oracle 固有の XA 例外が発生しなかった場合、このコードは一般 XA 例外の処理を行いません。

Oracle XA 最適化

Oracle JDBC には、分散トランザクションの 2 つ以上のブランチが同じデータベース・インスタンスを使用する場合、つまり、これらのブランチに対応付けられた XA リソース・インスタンスが同じリソース・マネージャに対応付けられている場合、パフォーマンスを改善する機能があります。

このような場合、これらの XA リソース・インスタンスのうち 1 つの prepare() メソッドのみが XA_OK を戻します（または失敗します）。残りは、更新が行われる場合でも、XA_RDONLY を戻します。これにより、トランザクション・マネージャは、すべてのトランザクション・ブランチを暗黙的に結合し、XA_OK を戻した（または失敗した）XA リソース・インスタンスによって、結合されたトランザクションをコミット（失敗した場合はロールバック）できます。

トランザクション・マネージャは、OracleXAResource クラスの isSameRM() メソッドを使用して、2 つの XA リソース・インスタンスが同じリソース・マネージャを使用しているかどうかを判断できます。このようにして、XA_RDONLY 戻り値の意味を解析できます。

分散トランザクションの実装

この項では、Oracle XA 機能を使用して分散トランザクションを実装する方法の例を示します。

Oracle XA のインポートのサマリー

Oracle XA 機能を使用するには、次のパッケージをインポートする必要があります。

```
import oracle.jdbc.xa.OracleXid;
import oracle.jdbc.xa.OracleXAException;
import oracle.jdbc.pool.*;
import oracle.jdbc.xa.client.*;
import javax.transaction.xa.*;
```

`oracle.jdbc.pool` パッケージには、接続プーリング機能のクラスが含まれています。この一部は、XA 関連クラスによってサブクラス化されています。

また、コードが Oracle データベースの内部で実行され、そのデータベースにアクセスして SQL 操作を行う場合、次のパッケージもインポートする必要があります。

```
import oracle.jdbc.xa.server.*;
```

（コードが実行されるデータベースにのみアクセスする場合、`oracle.jdbc.xa.client` クラスは必要ありません。）

`client` および `server` パッケージには、それぞれのバージョンの `OracleXADataSource`、`OracleXAConnection` および `OracleXAResource` クラスがあります。これら 3 つのクラスの抽象バージョンは、最上位の `oracle.jdbc.xa` パッケージに含まれています。

Oracle の XA コード・サンプル

このサンプルでは、異なるデータベースに対する 2 つのトランザクション・ブランチで、2 フェーズ分散トランザクションを使用します。

簡単にするため、この例では、通常は中間層に置くコードと、通常はトランザクション・マネージャに置くコード（XA リソース・メソッドの起動や、トランザクション ID の作成など）を組み合せているので、注意してください。

短くするため、トランザクション ID 作成の指定（`createID()` メソッドによる）と、SQL 操作の実行（`doSomeWork1()` および `doSomeWork2()` メソッドによる）は、ここでは示しません。完全なサンプルについては、20-107 ページの「2 フェーズ・コミットの操作 XA: XA4.java」を参照してください。

別の完全なサンプル（トランザクションの保留と再開を行う XA リソース機能の使用方法）については、20-102 ページの「保留と再開 XA: XA2.java」を参照してください。

このサンプルは、次の順序で実行します。

1. トランザクション・ブランチ #1 を開始します。
2. トランザクション・ブランチ #2 を開始します。
3. ブランチ #1 の DML 操作を実行します。
4. ブランチ #2 の DML 操作を実行します。
5. トランザクション・ブランチ #1 を終了します。
6. トランザクション・ブランチ #2 を終了します。
7. ブランチ #1 を準備します。
8. ブランチ #2 を準備します。
9. ブランチ #1 をコミットします。
10. ブランチ #2 をコミットします。

```
// You need to import the java.sql package to use JDBC
import java.sql.*;
import javax.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;
import oracle.jdbc.xa.OracleXid;
import oracle.jdbc.xa.OracleXAException;
import oracle.jdbc.xa.client.*;
import javax.transaction.xa.*;

class XA4
{
  public static void main (String args [])
       throws SQLException
  {

    try
    {
        String URL1 = "jdbc:oracle:oci8:@";
        String URL2 ="jdbc:oracle:thin:@(description=(address=(host=dlsun991)
                     (protocol=tcp)(port=5521))(connect_data=(sid=rdbms2)))";

        DriverManager.registerDriver(new OracleDriver());

        // You can put a database name after the @ sign in the connection URL.
        Connection conna =
          DriverManager.getConnection (URL1, "scott", "tiger");
```

```
// Prepare a statement to create the table
Statement stmta = conna.createStatement ();

Connection connb =
  DriverManager.getConnection (URL2, "scott", "tiger");

// Prepare a statement to create the table
Statement stmtb = connb.createStatement ();

try
{
  // Drop the test table
  stmta.execute ("drop table my_table");
}
catch (SQLException e)
{
  // Ignore an error here
}

try
{
  // Create a test table
  stmta.execute ("create table my_table (col1 int)");
}
catch (SQLException e)
{
  // Ignore an error here too
}

try
{
  // Drop the test table
  stmtb.execute ("drop table my_tab");
}
catch (SQLException e)
{
  // Ignore an error here
}
```

```
try
{
  // Create a test table
  stmtb.execute ("create table my_tab (col1 char(30))");
}
catch (SQLException e)
{
  // Ignore an error here too
}

// Create XADataSource instances and set properties.
OracleXADataSource oxds1 = new OracleXADataSource();
oxds1.setURL("jdbc:oracle:oci8:@");
oxds1.setUser("scott");
oxds1.setPassword("tiger");

OracleXADataSource oxds2 = new OracleXADataSource();

oxds2.setURL("jdbc:oracle:thin:@(description=(address=(host=dlsun991)
           (protocol=tcp)(port=5521))(connect_data=(sid=rdbms2)))");
oxds2.setUser("scott");
oxds2.setPassword("tiger");

// Get XA connections to the underlying data sources
XAConnection pc1  = oxds1.getXAConnection();
XAConnection pc2  = oxds2.getXAConnection();

// Get the physical connections
Connection conn1 = pc1.getConnection();
Connection conn2 = pc2.getConnection();

// Get the XA resources
XAResource oxar1 = pc1.getXAResource();
XAResource oxar2 = pc2.getXAResource();

// Create the Xids With the Same Global Ids
Xid xid1 = createXid(1);
Xid xid2 = createXid(2);

// Start the Resources
oxar1.start (xid1, XAResource.TMNOFLAGS);
oxar2.start (xid2, XAResource.TMNOFLAGS);

// Execute SQL operations with conn1 and conn2
doSomeWork1 (conn1);
doSomeWork2 (conn2);
```

```
// END both the branches -- IMPORTANT
oxar1.end(xid1, XAResource.TMSUCCESS);
oxar2.end(xid2, XAResource.TMSUCCESS);

// Prepare the RMs
int prp1 =  oxar1.prepare (xid1);
int prp2 =  oxar2.prepare (xid2);

System.out.println("Return value of prepare 1 is " + prp1);
System.out.println("Return value of prepare 2 is " + prp2);

boolean do_commit = true;

if (!((prp1 == XAResource.XA_OK) || (prp1 == XAResource.XA_RDONLY)))
   do_commit = false;

if (!((prp2 == XAResource.XA_OK) || (prp2 == XAResource.XA_RDONLY)))
   do_commit = false;

System.out.println("do_commit is " + do_commit);
System.out.println("Is oxar1 same as oxar2 ? " + oxar1.isSameRM(oxar2));

if (prp1 == XAResource.XA_OK)
  if (do_commit)
     oxar1.commit (xid1, false);
  else
     oxar1.rollback (xid1);

if (prp2 == XAResource.XA_OK)
  if (do_commit)
     oxar2.commit (xid2, false);
  else
     oxar2.rollback (xid2);

 // Close connections
conn1.close();
conn1 = null;
conn2.close();
conn2 = null;

pc1.close();
pc1 = null;
pc2.close();
pc2 = null;
```

```
      ResultSet rset = stmta.executeQuery ("select col1 from my_table");
      while (rset.next())
        System.out.println("Col1 is " + rset.getInt(1));

      rset.close();
      rset = null;

      rset = stmtb.executeQuery ("select col1 from my_tab");
      while (rset.next())
        System.out.println("Col1 is " + rset.getString(1));

      rset.close();
      rset = null;

      stmta.close();
      stmta = null;
      stmtb.close();
      stmtb = null;

      conna.close();
      conna = null;
      connb.close();
      connb = null;

  } catch (SQLException sqe)
  {
    sqe.printStackTrace();
  } catch (XAException xae)
  {
    if (xae instanceof OracleXAException) {
      System.out.println("XA Error is " +
                   ((OracleXAException)xae).getXAError());
      System.out.println("SQL Error is " +
                   ((OracleXAException)xae).getOracleError());
    }
  }
}

static Xid createXid(int bids)
  throws XAException
{...Create transaction IDs...}

private static void doSomeWork1 (Connection conn)
 throws SQLException
{...Execute SQL operations...}
```

```
  private static void doSomeWork2 (Connection conn)
   throws SQLException
  {...Execute SQL operations...}

}
```

16

JDBC OCI 拡張機能

この章では、OCI ドライバ固有の機能について説明します。次の項目が含まれます。

- OCI ドライバ接続プーリング
- プロキシ接続を介する中間層認証
- OCI ドライバの透過的アプリケーション・フェイルオーバー
- OCI HeteroRM XA
- PL/SQL 索引付き表へのアクセス

OCI ドライバ接続プーリング

OracleOCIConnectionPool によって提供される OCI ドライバ接続プーリング機能は、JDBC クライアントの一部です。接続プーリング機能の拡張により、次のような利点が得られます。

- **拡張性の向上** — より少ない物理接続でより多くの非カレントな論理接続をサポートできるため、OraclePooledConnection クラスのプーリングの細分化よりも優れています。物理接続にはコストがかかるので、この機能は有用です。アプリケーションでの使用が終わると、OraclePooledConnection オブジェクトの物理接続は再利用が可能です。また、OraclePooledConnection オブジェクトが使用可能な接続オブジェクトのプールに戻っても、サーバー側ではユーザー・セッションがまだクローズされていないため、OracleConnectionCacheImpl クラスの getConnection() メソッドを新たにコールするユーザーは同じである必要があります。これにより、専用サーバー・インスタンスの場合は、着信接続の数が減るため、バックエンド Oracle プロセスの数が減少します。パフォーマンスを向上させるために、物理接続は 1 つのコールが存続する間のみロックされます。
- **統一されたインタフェース** — 接続プーリングでは1つの統一されたインタフェースを使用するため、全体的なコードのメンテナンスの手間が軽減されます。
- **柔軟なスキーマ** — 各 OracleOCIConnection オブジェクトは異なるユーザー ID を持つことができるので、異なるスキーマを参照できます。
- **動的構成** — 接続プーリングの動的な構成が可能です。

注意： 1 つの物理接続に対する 1 つの JDBC ユーザー・セッションのマッピングや、OraclePooledConnection クラスを使用した物理接続オブジェクトの再利用といった既存の接続サポートは、まだサポートされています。（詳細は、14-12 ページの「接続プーリング」を参照してください。）しかし、OracleOCIConnectionPool クラスの改善された機能を使用することをお薦めします。

JDBC アプリケーションでは、同時に複数のプールを保持できます。複数のプールを保持すると、複数のアプリケーション・サーバーや異なるデータ・ソースのプールに対応できます。Oracle9*i* の OCI で提供される接続プーリングでは、アプリケーションで少数の物理接続を使用して、多数の論理接続を保持できます。この論理接続でのコールは、その時点で使用可能な物理接続にルーティングされます。コールの継続期間に基づく接続のプーリングは、よりスケーラブルな接続プーリング・ソリューションです。

Oracle JDBC ドライバに適用される Oracle JDBC 接続プーリング機能とキャッシング機能の詳細は、第 14 章の「接続プーリングとキャッシュ」を参照してください。詳細は、20-87 ページの「OCI 接続プール : OCIConnectionPool.java」を参照してください。

OCI ドライバ接続プーリング : 背景

Oracle9*i* JDBC OCI ドライバには、Oracle セッションおよび接続のファイングレイン管理などのいくつかのトランザクション・モニター機能があります。ハイエンド・アプリケーション・サーバーまたはトランザクション・モニターでは、コール・レベルで少数の物理接続を通じて、複数のセッションを多重化できるため、接続とバックエンド Oracle Server プロセスをプーリングすることで、高度な拡張性を実現できます。

`OracleOCIConnectionPool` インタフェースによって提供される接続プーリングでは、物理接続プールの管理を隠すことでセッション / 接続の分割インタフェースが簡素化されます。Oracle セッションは、`OracleOCIConnectionPool` から取得される `OracleOCIConnection` 接続オブジェクトです。通常、接続プールそのものは、さらに少ない物理接続の共有プールで構成されます。これは同じ数の専用サーバー・プロセスを含むバックエンド・サーバー・プールになります。この、少数の共有接続とバックエンド Oracle プロセスのプールにより、もっと多くの Oracle セッションを多重化することができます。

OCI ドライバ接続プーリングと MTS の比較

ある意味では、OCI ドライバ接続プーリングが中間層で提供する機能は、MTS がバックエンドで提供する機能と同じです。OCI ドライバ接続プーリングでは、中間層でセッションの多重化ロジックを管理することで、専用サーバー・インスタンスが MTS インスタンスと同じように動作します。したがって、専用サーバー・プロセスと専用サーバー・プロセスへの着信接続のプーリングは、中間層にある OCI 接続プーリングで制御されます。

OCI 接続プーリングと MTS の主な違いは、共有サーバーを使用する MTS の場合は、通常はデータベース・インスタンス内のディスパッチャに対してクライアントから接続が行われることです。ディスパッチャは、クライアントからの要求を適切な共有サーバーにリダイレクトします。他方、OCI 接続プールからの物理接続は、バックエンド・サーバー・プールにある Oracle 専用サーバー・プロセスに対して中間層から直接確立されます。

OCI 接続プールが有効なのは、中間層がマルチスレッドである場合のみです。各スレッドは、データベースに対して 1 つのセッションを保持できます。データベースに対する実際の接続は、`OracleOCIConnectionPool` によって保持され、（専用データベース・サーバー・プロセスのプールを含む）これらの接続は中間層のすべてのスレッドで共有されます。

ステートレス・セッションとステートフル・セッション

OCI 接続プーリングでは、ステートレスな物理接続とステートフル・セッションを提供します。セッションでステートレスな動作が必要な場合は、`OracleConnectionCacheImpl` インタフェースを使用します。

OCI 接続プールの定義

OCI 接続プールは、アプリケーションの開始時に作成されます。プールから接続を作成する方法は、OracleDataSource クラスを使用して接続を作成する方法とほぼ同じです。

OCI 接続プールの作成には、OracleDataSource クラスを拡張する oracle.jdbc.pool.OracleOCIConnectionPool クラスを使用します。OracleOCIConnectionPool クラス・インスタンスから、論理接続オブジェクトを取得できます。これらの接続オブジェクトは、OracleOCIConnection クラス型です。このクラスは、OracleConnection インタフェースを実装します。OracleOCIConnection クラスから作成した Statement オブジェクトには、OracleConnection インスタンスから作成した OracleStatement オブジェクトと同じフィールドおよびメソッドがあります。

次のコードは、OracleOCIConnectionPool クラスのヘッダー情報を示します。

```
/*
   * @param us  ConnectionPool user-id.
   * @param p   ConnectionPool password
   * @param name  logical name of the pool. This needs to be one in the
   *                  tnsnames.ora configuration file.
     @param config (optional)  Properties of the pool, if the default does not
               suffice. Default connection configuration is min =1, max=1,
               incr=0
                 Please refer setPoolConfig for property names.

                Since this is optional, pass null if the default configuration
                suffices.

   * @return
   *
   * Notes: Choose a userid and password that can act as proxy for the users
   *        in the getProxyConnection() method.

            If config is null, then the following default values will take
            effect
            CONNPOOL_MIN_LIMIT = 1
            CONNPOOL_MAX_LIMIT = 1
            CONNPOOL_INCREMENT = 0

*/

public synchronized OracleOCIConnectionPool
  (String       user,    String   password,  String name, Properties config)
  throws SQLException

/*
 * This will use the user-id, password and connection pool name values set
   LATER using the methods setUser, setPassword, setConnectionPoolName.
```

```
 * @return
 *
 * Notes:

     No OracleOCIConnection objects can be created on
     this class unless the methods setUser, setPassword, setPoolConfig
     are invoked.
     When invoking the setUser, setPassword later, choose a userid and
     password that can act as proxy for the users
 *   in the getProxyConnection() method.
 */
  public synchronized OracleOCIConnectionPool ()
    throws SQLException

/*
 * For getting a connection to the database.
 *
 * Notes: This will take user and password use for the
 * OracleOCIConnectionPool() call unless setUser and setPassword
 * calls were made.

 * @return     connection object
 */
```

oracle.jdbc.pool パッケージおよび oracle.jdbc.oci パッケージのインポート

OCI 接続プールを作成する前に、Oracle OCI 接続プーリング機能に次のパッケージをインポートする必要があります。

```
import oracle.jdbc.pool.*;
import oracle.jdbc.oci.*;
```

`oracle.jdbc.pool.*` パッケージには、接続キャッシュとイベント処理のためのクラスに加えて、`OracleDataSource`、`OracleConnectionPoolDataSource` および `OraclePooledConnectionPool` クラスが含まれます。`oracle.jdbc.oci.*` パッケージには、`OracleOCIConnection` クラスおよび `OracleOCIFailover` インタフェースが含まれます。

OCI 接続プールの作成

次のコードは、cpool という名前の OracleOCIConnectionPool クラスのインスタンスを作成する方法を示します。

```
OracleOCIConnectionPool cpool = new OracleOCIConnectionPool
    ("SCOTT", "TIGER", "jdbc:oracle:oci:@(description=(address=(host=
    myhost)(protocol=tcp)(port=1521))(connect_data=(sid=orcl)))", poolConfig);
```

poolConfig は、接続プールを指定する一連のプロパティです。poolConfig が NULL の場合は、デフォルト値が使用されます。たとえば、次の場合を考えてみましょう。

- poolConfig.put (OracleOCIConnectionPool.CONNPOOL_MIN_LIMIT, "4");
- poolConfig.put (OracleOCIConnectionPool.CONNPOOL_MAX_LIMIT, "10");
- poolConfig.put (OracleOCIConnectionPool.CONNPOOL_INCREMENT, "2");

前述のようなコンストラクタ・コール以外にも、個々のメソッドを使用してユーザー、パスワードおよび接続文字列を指定して、OracleOCIConnectionPool クラスのインスタンスを作成する方法もあります。

```
OracleOCIConnectionPool cpool = new OracleOCIConnectionPool ( );
cpool.setUser("SCOTT");
cpool.setPassword("TIGER");
cpool.setURL("jdbc:oracle:oci:@(description=(address=(host=
    myhost)(protocol=tcp)(port=1521))(connect_data=(sid=orcl)))");
cpool.setPoolConfig(poolConfig);  // In case you want to specify a different
                                  // configuration other than the default
                                  // values.
```

OCI 接続プール・パラメータの設定

接続プールの構成は、次に示す OracleOCIConnectionPool クラス属性によって決まります。

- CONNPOOL_MIN_LIMIT : プールで保持できる物理接続の最小数を指定します。
- CONNPOOL_MAX_LIMIT : プールで保持できる物理接続の最大数を指定します。
- CONNPOOL_INCREMENT : 既存の接続がすべて使用中で、さらに接続が必要になったとき、オープンする物理接続の増分数を指定します。オープンしている物理接続の数が、そのプールについてオープンできる接続の最大数を超えない場合にのみ、接続がオープンされます。
- CONNPOOL_TIMEOUT : 物理接続が切断されるまでのアイドル状態の時間を指定します。これは論理接続には影響しません。

- CONNPOOL_NOWAIT : この属性を有効に設定すると、プール内の最大数の接続が使用されている場合に、コールに物理接続が必要になるとエラーが返されます。この属性を無効に設定すると、接続が使用可能になるまでコールは待機します。いったんこの属性を TRUE に設定すると、FALSE にリセットすることはできません。

これらの属性はすべて動的に構成できます。したがって、アプリケーションではカレント・ロード（オープンしている接続の数と使用中の接続の数）を読み込んで、setPoolConfig() メソッドを使用してこれらの属性を適切に調整できます。

注意： CONNPOOL_MIN_LIMIT、CONNPOOL_MAX_LIMIT および CONNPOOL_INCREMENT パラメータのデフォルト値は、それぞれ 1、1 および 0 です。

OCI 接続プールのプロパティを構成するには、setPoolConfig() メソッドを使用します。次は、OracleOCIConnectionPool クラス属性の代表的な設定例です。

```
...
java.util.Properties p  = new java.util.Properties( );
p.put (OracleOCIConnectionPool.CONNPOOL_MIN_LIMIT, "1");
p.put (OracleOCIConnectionPool.CONNPOOL_MAX_LIMIT, "5");
p.put (OracleOCIConnectionPool.CONNPOOL_INCREMENT, "2");
p.put (OracleOCIConnectionPool.CONNPOOL_TIMEOUT, "10");
p.put (OracleOCIConnectionPool.CONNPOOL_NOWAIT, "true");
cpool.setPoolConfig(p);
...
```

前述の属性を設定するときは、次の規則に従ってください。

- CONNPOOL_MIN_LIMIT、CONNPOOL_MAX_LIMIT および CONNPOOL_INCREMENT パラメータは必須です。
- CONNPOOL_MIN_LIMIT の値は 0（ゼロ）より大きい値である必要があります。
- CONNPOOL_MAX_LIMIT の値は、CONNPOOL_MIN_LIMIT の値と CONNPOOL_INCREMENT の値を足した値以上である必要があります。
- CONNPOOL_INCREMENT の値は 0（ゼロ）以上である必要があります。
- CONNPOOL_TIMEOUT の値は 0（ゼロ）より大きい値である必要があります。
- CONNPOOL_NOWAIT の値は、「TRUE」または「FALSE」である必要があります（大 / 小文字を区別しない）。

OCI 接続プール・ステータスのチェック

接続プールのステータスをチェックするには、OracleOCIConnectionPool クラスの次のメソッドを使用します。

- int getMinLimit() : プールで保持できる物理接続の最小数を取得します。
- int getMaxLimit() : プールで保持できる物理接続の最大数を取得します。
- int getConnectionIncrement() : 既存の接続がすべて使用中に、コールに接続が必要になったときにオープンする物理接続の増分数を取得します。
- int getTimeout() : プールの物理接続が切断されるまでのアイドル状態の時間（秒単位）を取得します。接続の持続時間は最低使用頻度（LRU）スキームに基づいて決定されます。
- String getNoWait() : NOWAIT プロパティが有効かどうかを取得します。「TRUE」または「FALSE」の文字列が戻されます。
- int getPoolSize() : オープンしている物理接続の数を取得します。統計分析用にあくまで目安として使用してください。
- int getActiveSize() : オープンしていて、かつ使用中である物理接続の数を取得します。統計分析用にあくまで目安として使用してください。
- boolean isPoolCreated() : プールが作成されているかどうかを取得します。実際にプールが作成されるのは、OracleOCIConnection (user, password, url, poolConfig) をコールしたとき、または OracleOCIConnection() をコールした後に setUser、setPassword および setURL を実行したときです。

OCI 接続プールへの接続

OracleOCIConnectionPool クラスは、getConnection() メソッドをコールして、OracleOCIConnection クラスのインスタンスを作成します。このインスタンスは 1 つの接続を表します。すべての JDBC ドライバに適用されるデータベース接続の説明は、14-2 ページの「データ・ソース」を参照してください。

OracleOCIConnection クラスは OracleConnection クラスを拡張するため、後者のクラスの機能も備えています。ユーザー・セッションが完了したら、OracleOCIConnection オブジェクトをクローズしてください。クローズしない場合は、このオブジェクトはプール・インスタンスがクローズしたときにクローズされます。

getConnection() をコールするには、次の 2 つの方法があります。

- OracleConnection getConnection(String user, String password) : 指定されたユーザーとパスワードで識別される論理接続を取得します。このユーザーとパスワードは、プールの作成に使用されたのとは異なります。
- OracleConnection getConnection() : ユーザー名とパスワードを指定しないと、接続プールの作成に使用されたデフォルトのユーザー名とパスワードを使用して、接続オブジェクトが作成されます。

OracleConnection の拡張機能として、ユーザーのパスワードを変更するために次のような新しいメソッドが OracleOCIConnection に追加されています。

```
void passwordChange (String user, String oldPassword, String newPassword)
```

次のコードは、再構成中にアプリケーションが接続プールを使用する方法を示します。

```
import oracle.jdbc.oci.*;
import oracle.jdbc.pool.*;

public class cpoolTest
{
  public static void main (String args [])
    throws SQLException
  {
    /* pass the URL and "inst1" as the database link name from tnsnames.ora */
    OracleOCIConnectionPool cpool = new OracleOCIConnectionPool
      ("scott", "tiger", "jdbc:oracle:oci8@inst1", null);

    /* create virtual connection objects from the connection pool "cpool." The
       poolConfig can be null when using default values of min = 1, max = 1, and
       increment = 0, otherwise needs to set the properties mentioned earlier */
    OracleOCIConnection conn1 = (OracleOCIConnection) cpool.getConnection
      ("user1", password1");

    /* create few Statement objects and work on this connection, conn1 */
    Statement stmt = conn1.createStatement();
    ...
    OracleOCIConnection conn90 = (OracleOCIConnection) cpool.getConnection
      ("user90", "password90")   /* work on statement object from virtual
                                    connection "conn90" */
    ...
    /* if the throughput is less, increase the pool size */
    String newmin = String.valueOf (cpool.getMinLimit);
    String newmax = String.valueOf (2*cpool.getMaxLimit());
    String newincr = String.valueOf (1 + cpool.getConnectionIncrement());
    Properties newproperties = newProperties();
    newproperties.put (OracleOCIConnectionPool.CONNPOOL_MIN_LIMIT, newmin);
    newproperties.put (OracleOCIConnectionPool.CONNPOOL_MAX_LIMIT, newmax);
    newproperties.put (OracleOCIConnectionPool.CONNPOOL_INCREMENT, newincr);
    cpool.setPoolConfig (newproperties);
    }  /* end of main */
}  /* end of cpoolTest */
```

文の処理とキャッシュ

文キャッシュは、OracleOCIConnectionPool でサポートされます。キャッシュを使用すると、カーソルをオープン、解析およびクローズする必要がなくなるため、パフォーマンスが向上します。OracleOCIConnection.prepareStatement（SQL 問合せ）を実行すると、文キャッシュから SQL 問合せに一致する文が検索されます。一致する文が見つかった場合は、別の Statement オブジェクトを作成するかわりに、同じ Statement オブジェクトを再利用できます。キャッシュ・サイズは動的に増減できます。デフォルトのキャッシュ・サイズは 0（ゼロ）です。

注意： OracleOCIConnection クラスから作成した OracleStatement オブジェクトは、OracleConnection から作成したオブジェクトと同じように動作します。

OracleOCIConnectionPool での文キャッシュは、OracleConnectionCacheImpl の標準機能とはわずかに異なります。setStmtCacheSize() メソッドは、このプールから取得されるすべての OracleOCIConnection オブジェクトの文キャッシュ・サイズを設定します。ただし、OracleConnectionCacheImpl から取得する論理（OracleConnection）接続オブジェクトとは異なり、論理（OracleOCIConnection）接続オブジェクトの個々のキャッシュ・サイズは変更できます。（デフォルトのキャッシュ・サイズは 0（ゼロ）です。）

次のコードは、getConnection() メソッドのシグネチャを示します。

```
public synchronized OracleConnection getConnection( )
                  throws SQLException

/*
 * For getting a connection to the database.
 *
 * @param us  Connection user-id
 * @param p   Connection password
 * @return     connection object
 */
public synchronized OracleConnection getConnection(String us, String p)
throws SQLException
```

OCI 接続プールに使用される文キャッシュのタイプ

文キャッシュには、暗黙的文キャッシュと明示的文キャッシュの 2 つの形式があります。（暗黙的文キャッシュおよび明示的文キャッシュの詳細は、第 13 章「文キャッシュ」を参照してください。）どちらの形式の文キャッシュでも、`setStmtCacheSize()` メソッドを使用します。明示的文キャッシュでは、`Statement` オブジェクトをオープンおよびクローズするときに、JDBC アプリケーションでキーを指定する必要があります。暗黙的文キャッシュでは、JDBC アプリケーションでキーを指定する必要はなく、キャッシュはアプリケーションに対して透過的に実行されます。 また、明示的文キャッシュでは、結果セットのフェッチ状態が消去されません。したがって、`Statement.close(key="abc")` を実行すると、`Connection.preparedStatement(key="abc")` は `Statement` オブジェクトを戻し、前の `Statement.close(key="abc")` が実行されたときのフェッチ状態のまま、フェッチが続行されます。

暗黙的文キャッシュでは、フェッチ状態は消去され、カーソルも再実行されますが、パフォーマンスを向上させるためにカーソル・メタデータがキャッシュされます。場合によっては、（`clearMetaData` パラメータを使用して）クライアントでメタデータを消去する必要があります。

次のヘッダー情報は、メソッド・シグネチャを示します。

```
synchronized public void setStmtCacheSize (int size)

  /**
   *
   * @param size Size of the Cache
   * @param clearMetaData Whether the state has to be cleared or not
   * @exception     SQLException
   */
  public synchronized void setStmtCacheSize (int size, boolean clearMetaData)

 /**
   * Return the size of Statement Cache.
   * @return int Size of Statement Cache.

                If not set ie if statement caching is not enabled ,
   *             the default 0 is returned.
   */
  public synchronized int getStmtCacheSize()

   /*
   * Check whether Statement
   * Caching is enabled for this pool or Not.
   */
 public synchronized boolean isStmtCacheEnabled ()
```

JNDI および OCI 接続プール

Java Naming and Directory Interface（JNDI）機能により、Java オブジェクトのプロパティが永続化されるため、これらのプロパティを使用して、（オブジェクトのクローニングなど）オブジェクトの新しいインスタンスを作成できます。この利点は、古いオブジェクトを解放して、後でまったく同じプロパティを持つ新しいオブジェクトを作成できることです。`InitialContext.bind()` メソッドは、ファイルまたはデータベースでプロパティを永続化するのに対して、`InitialContext.lookup()` メソッドは永続ストアからプロパティを取得し、これらのプロパティを使用して新しいオブジェクトを作成します。

JNDI 機能を使用して、`OracleOCIConnectionPool` オブジェクトをバインドしたり、検索することができます。`OracleOCIConnectionPool` で新しくインタフェースをコールする必要はありません。

プロキシ接続を介する中間層認証

中間層認証では、1 つの JDBC 接続（セッション）が他の JDBC 接続のプロキシとしての役割を果たします。プロキシ・セッションが必要になるのは、次の場合です。

- 中間層がプロキシ・ユーザーのパスワードを知らない場合。この場合は、まず次のようなコードを使用して認証が行われます。

  ```
  alter user jeff grant connect through scott with roles role1, role2;
  ```

  この後、メソッドにより、すでに認証済みの「`scott`」という資格証明を使用して「`jeff`」として接続できます。中間層がすべてのデータベース・ユーザーのパスワードを知っていると、セキュリティの問題が生じることがあります。作成されたセッションは、（「`jeff`」/「`jeff-password`」を使用して）「`jeff`」が通常のように接続した場合と同様に動作しますが、「`jeff`」は中間層に対してパスワードを明かす必要はありません。このプロキシ・セッションでアクセスできるスキーマは、「`jeff`」のスキーマと、ロール・リストに指定されたスキーマです。したがって、「`scott`」が「`jeff`」を `EMP` 表にアクセスさせるには、次のコードを使用します。

  ```
  create role role1;
  grant select on EMP to role1;
  ```

  この role 句は、ロール・リストに指定された「`scott`」のデータベース・オブジェクトのみに「`jeff`」のアクセスを制限するものとして考えることができます。空のロール・リストを指定することも可能です。

- アカウント管理を目的とした場合。プロキシ・セッションを介して実行されるトランザクションは、（「`scott`」および「`scott2`」が認証済みであることを前提として）「`scott`」や「`scott2`」などの異なるユーザー名でユーザー（「`jeff`」）をプロキシとして実行したほうがアカウント管理が容易になります。これらの異なるプロキシ・セッションの下で「`jeff`」によって実行されたトランザクションは、別に記録されます。

OCI ドライバでプロキシ・セッションを作成するには、3 つの方法があります。ロールは次のどのオプションとも対応付けらることができます。

- USER NAME : ユーザー名またはパスワード（あるいはその両方）を指定します。「パスワード」オプションが存在するのは、ユーザー（「jeff」）によって実行されたデータベース操作のアカウント・ログを取るためです。SQL 句は次のとおりです。

  ```
  alter user jeff grant connect through scott authenticated using password;
  ```

  認証句の指定がないため、デフォルトが使用されます。つまり、パスワードを入力する必要がなく、ユーザー名で認証されます。

- DISTINGUISHED NAME : これは、プロキシとしての役割を果たすユーザーのパスワードにかわるグローバル名です。つまり、作成したユーザー「jeff」は、次のようにグローバルに識別されます。

  ```
  'CN=jeff,OU=americas,O=oracle,L=redwoodshores,ST=ca,C=us';
  ```

  上記の句は、識別名です。したがって、次のような認証が必要になります。

  ```
  alter user jeff grant connect through scott authenticated using distinguished
  name;
  ```

- CERTIFICATE : これは、（プロキシとしての役割を果たす）ユーザーの資格証明をさらに暗号化してデータベースに渡す方法です。証明書には、識別名が含まれています。証明書を生成する 1 つの方法は、（runutl mkwallet を使用して）Wallet を作成してから、その Wallet をデコード化して証明書を取得することです。したがって、次のような認証が必要になります。

  ```
  alter user jeff grant connect through scott authenticated using certificate;
  ```

次のコードは、プロキシ型プロセスに関する情報を含んだ getProxyConnection() メソッドのシグネチャを示します。

```
/*
   * For creating a proxy connection. All macros are defined
   * in OracleOCIConnectionPool.java
   *
   * @param proxyType  Can be one of following types
              PROXYTYPE_USER_NAME
                       - This will be the normal mode of specifying the user
                         name in proxyUser as in Oracle8i

              PROXYTYPE_DISTINGUISHED_NAME
                       - This will specify the distinguished name of the user
                         in proxyUser

             PROXYTYPE_CERTIFICATE
                       - This will specify the proxy certificate
```

```
     The Properties (ie prop) should be set as follows.

     If PROXYTYPE_USER_NAME
        PROXY_USER_NAME and/or PROXY_USER_PASSWORD depending
           on how the connection-pool owner was authenticated
           to act as proxy for this proxy user
           PROXY_USER_NAME (String) = user to be proxied for
           PROXY_PASSWORD (String) = password of the user to be proxied for

    else if PROXYTYPE_DISTINGUISHED_NAME
         PROXY_DISTINGUISHED_NAME (String) = (global) distinguished name of the user
to be proxied for
    else if PROXYTYPE_CERTIFICATE (byte[])
         PROXY_CERTIFICATE = certficate containing the encoded
                                distinguished name



    PROXY_ROLES (String[])  Set of roles which this proxy connection can use. Roles
can be null, and can be associated
with any of the above proxy methods.

   *
   * @return   connection object
   *
   * Notes: The user and password used to create OracleOCIConnectionPool()
   *        must be allowed to act as proxy for user 'us'.
   */
   public synchronized OracleConnection getProxyConnection(String proxyType,
     Properties prop)
     throws SQLException
```

中間層認証の完全なコード例については、20-90 ページの「中間層認証 : NtierAuth.java」を参照してください。

OCI ドライバの透過的アプリケーション・フェイルオーバー

透過的アプリケーション・フェイルオーバー（TAF）または単純にアプリケーション・フェイルオーバーは、OCI ドライバの機能の 1 つです。この機能により、接続先のデータベース・インスタンスがダウンした場合でも、データベースに自動的に再接続できます。この場合、アクティブ・トランザクションはロールバックされます。（トランザクションがロールバックされると、最後にコミットされたトランザクションがリストアされます。）別のノードで作成されても、新しいデータベース接続は元の接続とまったく同じです。接続がどのように切断されても関係ありません。

TAF は常にアクティブなので、設定する必要はありません。

OCI および TAF に関する詳細は、『Oracle Call Interface プログラマーズ・ガイド』を参照してください。

フェイルオーバー・タイプ・イベント

`OracleOCIFailover` インタフェースで発生しうるフェイルオーバー・イベントは、次のとおりです。

- `FO_SESSION`：`tnsnames.ora` ファイル `CONNECT_DATA` フラグの `FAILOVER_MODE=SESSION` と等価です。サーバー側ではユーザー・セッションのみが認証され、OCI アプリケーションのオープン・カーソルは再実行する必要があります。
- `FO_SELECT`：`tnsnames.ora` ファイル `CONNECT_DATA` フラグの `FAILOVER_MODE=SELECT` と等価です。サーバー側でユーザー・セッションが認証されるだけでなく、OCI のオープン・カーソルもフェッチを続行できます。これは、クライアント側のロジックで各オープン・カーソルのフェッチ状態が保持されることを意味します。
- `FO_NONE`：`tnsnames.ora` ファイル `CONNECT_DATA` フラグの `FAILOVER_MODE=NONE` と等価です。これはデフォルトで、フェイルオーバー機能は使用されません。また、フェイルオーバーが実行されるのを明示的に禁止することにもなります。さらに、`FO_TYPE_UNKNOWN` は、OCI ドライバから不正なフェイルオーバー型が戻されたことを意味します。
- `FO_BEGIN`：フェイルオーバーで失われた接続が検出され、フェイルオーバーが開始されることを意味します。
- `FO_END`：フェイルオーバーが正常に完了したことを意味します。
- `FO_ABORT`：フェイルオーバーが失敗し、再試行できないことを意味します。
- `FO_REAUTH`：ユーザー・ハンドルが再認証されたことを意味します。
- `FO_ERROR`：フェイルオーバーが一時的に失敗したが、アプリケーションでエラーを処理してフェイルオーバーを再試行できることを意味します。エラー処理の一般的な方法は、`sleep()` メソッドを発行し、値 `FO_RETRY` を戻して再試行することです。

- FO_RETRY : 前述の説明を参照してください。
- FO_EVENT_UNKNOWN : 不正なフェイルオーバー・イベントです。

フェイルオーバー・イベントを使用した完全なコード例については、20-94 ページの「JDBC OCI アプリケーション・フェイルオーバー・コールバック : OCIFailOver.java」を参照してください。

TAF コールバック

TAF コールバックは、1 つのデータベース接続に障害が起きた場合に、別のデータベース接続にフェイルオーバーするために使用します。TAF コールバックは、フェイルオーバーの場合に登録されるコールバックです。コールバックは、フェイルオーバー中にイベントが発生したことを JDBC アプリケーションに通知するためにコールされます。アプリケーション側でもフェイルオーバーをある程度制御できます。

注意 : コールバックの設定はオプションです。

Java TAF コールバック・インタフェース

OracleOCIFailover インタフェースには、次の型およびイベントをサポートする callbackFn() メソッドが含まれます。

```
public interface OracleOCIFailover{

// Possible Failover Types
public static final int FO_SESSION = 1;
public static final int FO_SELECT  = 2;
public static final int FO_NONE  = 3;
public static final int;

// Possible Failover events registered with callback
public static final int FO_BEGIN   = 1;
public static final int FO_END     = 2;
public static final int FO_ABORT   = 3;
public static final int FO_REAUTH  = 4;
public static final int FO_ERROR  = 5;
public static final int FO_RETRY  = 6;
public static final int FO_EVENT_UNKNOWN = 7;

public int callbackFn (Connection conn,
                       Object ctxt, // ANy thing the user wants to save
                       int type, // One of the above possible Failover Types
                       int event ); // One of the above possible Failover Events
```

FO_ERROR イベントの処理

新しい接続にフェイルオーバーするときにエラーが発生した場合、JDBC アプリケーションはそのフェイルオーバーを再試行できます。一般に、アプリケーションはいったんスリープ状態に入ってから、コールバックで FO_RETRY を戻すことで、無限または一定の時間のみ再試行を行います。

FO_ABORT イベントの処理

登録されたコールバックは、FO_ERROR イベントが渡された場合でも、FO_ABORT イベントを戻す必要があります。

OCI HeteroRM XA

Oracle8*i* リリース 8.1.6 以上でのみ動作する通常の JDBC XA 機能とは異なり、JDBC HeteroRM XA 機能を使用すると、Oracle8*i* 8.1.6 以前のリリースでも XA 操作を実行できます。可能なかぎり、HeteroRM XA を使用することをお薦めします。

HeteroRM XA は、OracleXADataSource クラスの tnsEntry および nativeXA プロパティを使用して有効化します。これらのプロパティの詳細は、14-5 ページの表 14-2「Oracle 拡張データ・ソース・プロパティ」を参照してください。

XA の詳細は、第 15 章「分散トランザクション」を参照してください。

構成およびインストール

Solaris 共有ライブラリ（libheteroxa9.so および libheteroxa9_g.so）により、HeteroRM XA 機能で Oracle8*i* 8.1.6 以前のリリースへのアクセスをサポートできます。これらのライブラリの NT 版は、heteroxa9.dll および heteroxa9_g.dll です。HeteroRM XA 機能が正しく動作するには、使用するシステムに応じて、Solaris 検索パスまたは NT DLL パスにこれらのライブラリをインストールする必要があります。

注意： _g という接尾辞が付いたライブラリは、デバッグ・ライブラリです。

例外処理

分散トランザクションで HeteroRM XA 機能を使用する場合は、アプリケーションで `OracleXAException` または `OracleSQLException` をチェックするのではなく、単純に `XAException` または `SQLException` をチェックすることをお薦めします。

HeteroRM XA メッセージのリストについては、B-10 ページの「HeteroRM XA メッセージ」を参照してください。

注意： 標準の XA エラー・コードに対する SQL エラー・コードのマッピングは、HeteroRM XA 機能には適用されません。

HeteroRM XA のコード・サンプル

次のコードは、HeteroRM XA 機能を有効にする方法を示します。HeteroRM XA 機能の完全なコード例については、20-113 ページの「HeteroRM XA: XA6.java」を参照してください。

```
// Create a XADataSource instance
OracleXADataSource oxds = new OracleXADataSource();
oxds.setURL(url);

// Set the nativeXA property to use HeteroRM XA feature
oxds.setNativeXA(true);

// Set the tnsEntry property to an older DB as required
oxds.setTNSEntryName("ora805");
```

PL/SQL 索引付き表へのアクセス

Oracle JDBC OCI ドライバを使用すると、JDBC アプリケーションで、索引付き表パラメータを使用して PL/SQL をコールできます。

PL/SQL 索引付き表パラメータを使用する完全なコード例については、20-10 ページの「JDBC からの PL/SQL 索引付き表へのアクセス : PLSQLIndexTab.java」を参照してください。

重要： PL/SQL レコードの索引付き表はサポートされません。

概要

Oracle JDBC OCI ドライバは、スカラー・データ型の PL/SQL 索引付き表をサポートします。表 16-1 に、サポートされるスカラー型と対応する JDBC タイプコードを示します。

表 16-1 PL/SQL 型と対応する JDBC 型

PL/SQL 型	JDBC 型
BINARY_INTEGER	NUMERIC
NATURAL	NUMERIC
NATURALN	NUMERIC
PLS_INTEGER	NUMERIC
POSITIVE	NUMERIC
POSITIVEN	NUMERIC
SIGNTYPE	NUMERIC
STRING	VARCHAR

注意： Oracle JDBC は、要素型として RAW、DATE および PL/SQL RECORD をサポートしません。

通常の Oracle JDBC 入力バインド、出力登録、およびデータアクセス・メソッドは、PL/SQL 索引付き表をサポートしません。この章では、これらの型をサポートするための追加のメソッドについて説明します。

OraclePreparedStatement および OracleCallableStatement クラスは、追加のメソッドを定義します。これらのメソッドを、次に示します。

- setPlsqlIndexTable()
- registerIndexTableOutParameter()
- getOraclePlsqlIndexTable()
- getPlsqlIndexTable()

これらのメソッドは、PL/SQL 索引付き表を IN、OUT（ファンクション戻り値を含む）または IN OUT パラメータとして処理します。PL/SQL 構文の一般的な情報については、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

次の項では、PL/SQL 索引付き表をバインドおよび登録するために使用するメソッドについて説明します。

IN パラメータのバインド

PL/SQL 索引付き表パラメータを IN パラメータ・モードでバインドするには、OraclePreparedStatement および OracleCallableStatement クラスで定義される setPlsqlIndexTable() メソッドを使用します。

```
synchronized public void setPlsqlIndexTable
   (int paramIndex, Object arrayData, int maxLen, int curLen, int elemSqlType,
   int elemMaxLen) throws SQLException
```

setPlsqlIndexTable() メソッドの引数について表 16-2 で説明します。

表 16-2 setPlsqlIndexTable () メソッドの引数

引数	説明
int paramIndex	この引数は、文内のパラメータの位置を示します。
Object arrayData	この引数は、PL/SQL 索引付き表パラメータにバインドされる値の配列です。この値は、java.lang.Object 型です。この値には、int [] などの Java プリミティブ型配列または BigDecimal [] などの Java オブジェクト配列を指定できます。
int maxLen	この引数は、索引付き表バインド値の表の最大長を指定します。また、この値がバッチ更新の数を示す curLen のとりうる最大値にもなります。スタンドアロンのバインドの場合、maxLen は curLen と同じ値を使用する必要があります。この引数は必須です。
int curLen	この引数は、arrayData の索引付き表バインド値の実際のサイズを指定します。curLen 値が arrayData のサイズより小さい場合は、curLen で指定した数の表要素のみがデータベースに渡されます。curLen 値が arrayData のサイズよりも大きい場合は、arrayData 全体がデータベースへ送信されます。

表 16-2 setPlsqlIndexTable () メソッドの引数 （続き）

引数	説明
int elemSqlType	この引数は、OracleTypes クラスで定義された値に基づく索引付き表の要素型を指定します。
int elemMaxLen	この引数は、要素型が CHAR、VARCHAR または RAW である場合に索引付き表の要素の最大長を指定します。この値は、他の型では無視されます。

次のコード例では、setPlsqlIndexTable() メソッドを使用して、索引付き表を IN パラメータとしてバインドします。

```
// Prepare the statement
OracleCallableStatement procin = (OracleCallableStatement)
   conn.prepareCall ("begin procin (?); end;");

// index-by table bind value
int[] values = { 1, 2, 3 };

// maximum length of the index-by table bind value. This
// value defines the maximum possible "currentLen" for batch
// updates. For standalone binds, "maxLen" should be the
// same as "currentLen".
int maxLen = values.length;

// actual size of the index-by table bind value
int currentLen = values.length;

// index-by table element type
int elemSqlType = OracleTypes.NUMBER;

// index-by table element length in case the element type
// is CHAR, VARCHAR or RAW. This value is ignored for other
// types.
int elemMaxLen = 0;

// set the value
procin.setPlsqlIndexTable (1, values,
                           maxLen, currentLen,
                           elemSqlType, elemMaxLen);

// execute the call
procin.execute ();
```

OUT パラメータの受取り

この項では、PL/SQL 索引付き表を OUT パラメータとして登録する方法について説明します。また、様々なマッピング・スタイルで OUT バインド値にアクセスする方法について説明します。

注意： この項で説明するメソッドは、ファンクション戻り値および IN OUT パラメータ・モードに適用されます。

OUT パラメータの登録

PL/SQL 索引付き表を OUT パラメータとして登録するには、OracleCallableStatement クラスで定義された registerIndexTableOutParameter() メソッドを使用します。

```
synchronized public void registerIndexTableOutParameter
   (int paramIndex, int maxLen, int elemSqlType, int elemMaxLen)
   throws SQLException
```

表 16-3 では、registerIndexTableOutParameter() メソッドの引数について説明します。

表 16-3 registerIndexTableOutParameter () メソッドの引数

引数	説明
int paramIndex	この引数は、文内のパラメータの位置を示します。
int maxLen	この引数は、戻される索引付き表バインド値の表の最大長を指定します。
int elemSqlType	この引数は、OracleTypes クラスで定義された値に基づく索引付き表の要素型を指定します。
int elemMaxLen	この引数は、要素型が CHAR、VARCHAR または RAW である場合に索引付き表の要素の最大長を指定します。この値は、他の型では無視されます。

次のコード例では、registerIndexTableOutParameter() メソッドを使用して、索引付き表を OUT パラメータとして登録します。

```
// maximum length of the index-by table value. This
// value defines the maximum table size to be returned.
int maxLen = 10;

// index-by table element type
int elemSqlType = OracleTypes.NUMBER;

// index-by table element length in case the element type
```

```
// is CHAR, VARCHAR or RAW. This value is ignored for other
// types
int elemMaxLen = 0;

// register the return value
funcnone.registerIndexTableOutParameter
   (1, maxLen, elemSqlType, elemMaxLen);
```

OUT パラメータ値へのアクセス

OUT バインド値へアクセスするために、OracleCallableStatement クラスでは、様々なマッピング・スタイルの索引付き表の値を戻す複数のメソッドを定義します。JDBC ドライバでは、次の 3 つのマッピングを選択できます。

マッピング	使用するメソッド
JDBC デフォルト・マッピング	getPlsqlIndexTable(int)
Oracle マッピング	getOraclePlsqlIndexTable(int)
Java プリミティブ型マッピング	getPlsqlIndexTable(int, Class)

JDBC デフォルト・マッピング (int) シグネチャを持つ getPlsqlIndexTable() メソッドは、JDBC デフォルト・マッピングを使用して索引付き表の要素を戻します。

```
public Object getPlsqlIndexTable (int paramIndex)
   throws SQLException
```

getPlsqlIndexTable() メソッドの引数について、表 16-4 で説明します。

表 16-4 getPlsqlIndexTable () メソッドの引数

引数	説明
int paramIndex	この引数は、文内のパラメータの位置を示します。

戻り値は Java 配列です。この配列の要素は、その SQL 型に対応するデフォルトの Java 型です。たとえば、NUMERIC タイプコードの要素を持つ索引付き表の場合、要素値は、Oracle JDBC ドライバによって BigDecimal にマップされます。また、getPlsqlIndexTable() メソッドは、BigDecimal[] 配列を戻します。JDBC アプリケーションの場合、表要素値にアクセスするには、戻り値を BigDecimal[] 配列へキャストする必要があります。(デフォルトのマッピングのリストについては、3-16 ページの「データ型マッピング」を参照してください。)

次のコード例では、getPlsqlIndexTable() メソッドを使用して、JDBC デフォルト・マッピングで索引付き表の要素を戻します。

```
// access the value using JDBC default mapping
```

```
BigDecimal[] values =
   (BigDecimal[]) procout.getPlsqlIndexTable (1);

// print the elements
for (int i=0; i<values.length; i++)
   System.out.println (values[i].intValue());
```

Oracle マッピング getOraclePlsqlIndexTable() メソッドは、Oracle マッピングを使用して索引付き表要素を戻します。

```
public Datum[] getOraclePlsqlIndexTable (int paramIndex)
      throws SQLException
```

getOraclePlsqlIndexTable() メソッドの引数について、表 16-5 で説明します。

表 16-5 getOraclePlsqlIndexTable () メソッドの引数

引数	説明
int paramIndex	この引数は、文内のパラメータの位置を示します。

戻り値は oracle.sql.Datum 配列です。Datum 配列の要素は、その要素の SQL 型に対応するデフォルトの Datum 型になります。たとえば、数値要素の索引付き表の要素値は、Oracle マッピングの oracle.sql.NUMBER 型にマップされます。また、getOraclePlsqlIndexTable() メソッドは、oracle.sql.NUMBER 要素を格納する oracle.sql.Datum 配列を戻します。

次のコード例では、getOraclePlsqlIndexTable() メソッドを使用して、PL/SQL 索引付き表の OUT パラメータの要素に、Oracle マッピングを使用してアクセスします。(登録用のコードは省略します。)

```
// Prepare the statement
OracleCallableStatement procout = (OracleCallableStatement)
                                  conn.prepareCall ("begin procout (?); end;");

...

// execute the call
procout.execute ();

// access the value using Oracle JDBC mapping
Datum[] outvalues = procout.getOraclePlsqlIndexTable (1);

// print the elements
for (int i=0; i<outvalues.length; i++)
   System.out.println (outvalues[i].intValue());
```

Java プリミティブ型マッピング (int, Class) シグネチャを持つ getPlsqlIndexTable() メソッドは、Java プリミティブ型の索引付き表の要素を戻します。戻り値は Java 配列です。

```
synchronized public Object getPlsqlIndexTable
   (int paramIndex, Class primitiveType) throws SQLException
```

getPlsqlIndexTable() メソッドの引数について、表 16-6 で説明します。

表 16-6 getPlsqlIndexTable () メソッドの引数

引数	説明
int paramIndex	この引数は、文内のパラメータの位置を示します。
Class primitiveType	この引数は、索引付き表の要素の変換先となる Java プリミティブ型を指定します。たとえば、java.lang.Integer.TYPE を指定すると、戻り値は int 配列になります。 このパラメータには、次の値を指定できます。 java.lang.Integer.TYPE java.lang.Long.TYPE java.lang.Float.TYPE java.lang.Double.TYPE java.lang.Short.TYPE

次のコード例では、getPlsqlIndexTable() メソッドを使用して、数値の PL/SQL 索引付き表の要素にアクセスします。この例では、2 つ目のパラメータは java.lang.Integer.TYPE を指定するため、getPlsqlIndexTable() メソッドは int 配列を戻します。

```
OracleCallableStatement funcnone = (OracleCallableStatement)
   conn.prepareCall ("begin ? := funcnone; end;");

// maximum length of the index-by table value. This
// value defines the maximum table size to be returned.
int maxLen = 10;

// index-by table element type
int elemSqlType = OracleTypes.NUMBER;

// index-by table element length in case the element type
// is CHAR, VARCHAR or RAW. This value is ignored for other
// types
int elemMaxLen = 0;
```

```
// register the return value
funcnone.registerIndexTableOutParameter (1, maxLen,
                                         elemSqlType, elemMaxLen);
// execute the call
funcnone.execute ();

// access the value as a Java primitive array.
int[] values = (int[])
   funcnone.getPlsqlIndexTable (1, java.lang.Integer.TYPE);

// print the elements
for (int i=0; i<values.length; i++)
   System.out.println (values[i]);
```

17

Java Transaction API

データベース・リソースに接続するには、JDBC を使用します。ただし、複数のデータベースに対するすべての変更を 1 回のトランザクションで更新するには、JTA グローバル・トランザクションで JDBC 接続を使用する必要があります。この章では、JTA グローバル・トランザクションについて詳しく説明します。トランザクションでデータベース SQL 更新を行うプロセスは、データベース・リソースの取得と呼ばれます。

この章は、JTA の処理知識があるユーザーを対象としています。この章では、主に Sun 社の JTA 仕様と Oracle JTA 実装の違いについて例を示して説明します。Sun 社の JTA 仕様については、`http://www.javasoft.com` を参照してください。

- トランザクションの概要
- JTA の概要
- ネームスペースへのトランザクション・オブジェクトのバインド
- JTA クライアント側での境界設定
- JTA サーバー側でのデータベースの取得
- 2 フェーズ・コミット・エンジンの構成
- DataSource オブジェクトの動的作成
- JDBC の制限事項

トランザクションの概要

トランザクションでは、単一のアプリケーションが行う複数のデータベースに対する変更を、1 つの作業単位として管理します。つまり、1 つ以上のデータベース内でデータを管理するアプリケーションがあり、1 つのトランザクションで管理されている場合は、すべてのデータベースのすべての変更を同時にコミットできます。

トランザクションを、次に示す ACID 属性の用語を使用して説明します。

- アトミック性 : 1 つのトランザクションで実行されたデータベースに対する変更は、どの変更が失敗した場合でもすべてロールバックされます。
- 一貫性 : トランザクションを実行しても、データベースは常に一貫した状態に保たれます。
- 孤立性 : トランザクションの中間処理は、データベースの他のユーザーは参照されません。
- 耐久性 : トランザクションが完了（コミットまたはロールバック）すると、その結果はデータベースに保持されます。

Sun 社により指定されている JTA 実装は、JDBC 2.0 仕様および XA アーキテクチャに大きく依存します。このため、すべてのデータベース間でトランザクションを完全に管理できるようにするためには、アプリケーションに複雑な要件が必要です。Sun 社の Java Transaction API（JTA）1.0.1 および JDBC 2.0 については、`http://www.javasoft.com` で参照できます。

Oracle9*i* 環境で JTA を使用する場合は、次の点に注意する必要があります。

- グローバル・トランザクションとローカル・トランザクション
- トランザクション境界の設定
- リソースの取得
- 2 フェーズ・コミット
- リソースの取得
- JTA 制限事項

グローバル・トランザクションとローカル・トランザクション

これまでは、アプリケーションが JDBC または SQL サーバーを使用してデータベースに接続するたびに、1 個のトランザクションを作成していました。ただし、このトランザクションには単一のデータベースのみが含まれ、データベースに対するすべての更新はそれらの変更の最後にコミットされていました。このトランザクションをローカル・トランザクションと呼びます。

これに対し、グローバル・トランザクションには、トランザクションに含まれるすべてのオブジェクトとデータベースを追跡するオブジェクトの複雑なセットが含まれます。これらのグローバル・トランザクション・オブジェクト（TransactionManager および Transaction）は、グローバル・トランザクションに含まれるすべてのオブジェクトとリソースを追跡します。TransactionManager および Transaction では、トランザクションの最後に、必ずすべてのデータベースの変更が、同時にかつ自動的にコミットされます。

グローバル・トランザクション内では、ローカル・トランザクションを実行できません。実行すると、次のようなエラーが発生します。

```
ORA-2089 下位セッションに COMMIT は使用できません。
```

SQL コマンドの中には、暗黙的にローカル・トランザクションを実行するものがあります。CREATE TABLE などの SQL DDL 文は、暗黙的にローカル・トランザクションを起動してコミットします。DDL 文の実行対象であるデータベースを取得したグローバル・トランザクションが含まれていると、そのグローバル・トランザクションは失敗します。

トランザクション境界の設定

トランザクションには境界が設定されています。つまり、各トランザクションには明確な開始点と終了点があります。たとえば、SQL*Plus などのインタラクティブ・ツールでは、トランザクションは SQL BEGIN 文で開始されます。これにより、各 SQL DML 文がトランザクションの一部となります。トランザクションは、SQL COMMIT 文または ROLLBACK 文の発行時に終了します。

JTA の場合、トランザクションは UserTransaction オブジェクトを使用してプログラム的に境界設定されます。このオブジェクトはネームスペース内でバインドする必要があります。クライアントまたはサーバーでトランザクションを開始するには、UserTransaction.begin メソッドをコールします。トランザクションを終了するには、UserTransaction.commit または UserTransaction.rollback メソッドをコールします。

トランザクションを開始する発信元のクライアントまたはオブジェクトも、commit または rollback でトランザクションを終了する必要があります。クライアントがトランザクションを開始して、サーバー・オブジェクトをコールした場合、クライアントはコールされたメソッドが戻された後にトランザクションを終了する必要があります。コールされたサーバー・オブジェクトは、そのトランザクションを終了できません。

UserTransaction インタフェース

トランザクション境界設定には、次のようなメソッドを使用できます。これらのメソッドは、javax.transaction.UserTransaction インタフェース内に定義されます。

```
public abstract void begin() throws NotSupportedException, SystemException;
```

新しいトランザクションを作成し、そのトランザクションをスレッドに対応付けます。

例外 :

- NotSupportedException: スレッドがすでにトランザクションに対応付けられいる場合に発生します。ネストされたトランザクションはサポートされません。
- SystemException: 予期しないエラー状態が起きると発生します。

```
public abstract void commit() throws RollbackException, HeuristicMixedException,
  HeuristicRollbackException, SecurityException, IllegalStateException,
  SystemException;
```

トランザクションに関与するリソースに対する変更をすべて保存して、既存のトランザクションを完了します。このメソッドが完了すると、スレッドとトランザクションの対応付けが解除されます。

例外 :

- RollbackException: トランザクション内のリソースが正常にコミットできない場合に発生します。リソースに対する変更はすべてロールバックされます。
- HeuristicMixedException: 一部のリソースはコミットされたが、一部のリソースがロールバックされたことを示します。
- HeuristicRollbackException: トランザクションに関与するリソースに対する更新操作の一部がロールバックされたことを示します。
- SecurityException: セキュリティ違反になるために、スレッドがトランザクションをコミットできない場合に発生します。
- IllegalStateException: カレント・スレッドにまだトランザクションが対応付けられていない場合に発生します。これは、まだ開始されていないトランザクションをコミットしようとした場合に発生します。
- SystemException: 予期しないエラー状態が起きると発生します。

```
public abstract void rollback() throws IllegalStateException, SecurityException,
  SystemException;
```

カレント・スレッドに対応付けられたトランザクションをロールバックします。

例外 :

- SecurityException: セキュリティ違反になるために、スレッドがトランザクションをロールバックできない場合に発生します。
- IllegalStateException: カレント・スレッドにまだトランザクションが対応付けられいない場合に発生します。これは、まだ開始されていないトランザクションをロールバックしようとした場合に発生します。
- SystemException: 予期しないエラー状態が起きると発生します。

```
public abstract int getStatus() throws SystemException;
```

カレント・スレッドに対応付けられたトランザクション・ステータスを取り出します。

例外 :

- SystemException: 予期しないエラー状態が起きると発生します。

```
public abstract void setRollbackOnly() throws IllegalStateException,
  SystemException;
```

結果がロールバックされるように、カレント・スレッドに対応付けられたトランザクションを変更します。

例外 :

- IllegalStateException: カレント・スレッドにまだトランザクションが対応付けられていない場合に発生します。まだ開始されていないトランザクションに対してロールバックを設定しようとした場合に発生します。
- SystemException: 予期しないエラー状態が起きると発生します。

```
public abstract setTransactionTimeout(int seconds) throws SystemException;
```

このカレント・スレッドに対応付けられたトランザクションのタイムアウト値を秒単位で設定します。

例外 :

- SystemException: 予期しないエラー状態が起きると発生します。

リソースの取得

Oracle データベースを含む、グローバル・トランザクションで管理する各リソースは、そのトランザクションで取得する必要があります。トランザクション・マネージャはグローバル・トランザクションに関与するすべてのリソースを追跡します。

データベースへの JDBC 接続を取得する方法はいくつかありますが、JTA トランザクションにデータベースを含めることができる方法は 1 つのみです。次の表は、JDBC 接続を取得するための標準的な方法を示します。

表 17-1 JDBC メソッド

取得メソッド	説明
`OracleDriver().defaultConnection()`	JDBC 2.0 以前でローカル接続を取得するために使用されるメソッド。使用はローカル・トランザクション内に限られます。
`DriverManager.getConnection("jdbc:oracle:kprb:")`	JDBC 2.0 以前でローカル接続を取得するために使用されるメソッド。使用はローカル・トランザクション内に限られます。
`DataSource.getConnection("jdbc:oracle:kprb:")`	ローカル・データベースへの接続を取得するための JDBC 2.0 メソッド。JTA トランザクションに使用できます。

前述のメソッドのうち、グローバル・トランザクションにデータベース・リソースを含めることができるのは `DataSource` オブジェクトを使用した場合のみです。JTA トランザクション内で使用する `DataSource` オブジェクトは、`bindds` コマンドの `-jta` オプションを使用してバインドする必要があります。これにより、ネームスペース内に Oracle 固有の JTA `DataSource` がバインドされます。グローバル・トランザクションの場合は、他のタイプの `DataSource` は使用できません。

グローバル・トランザクションに確実に文を含めるためには、次の操作を実行する必要があります。

1. JNDI ネームスペースに、JTA `DataSource` オブジェクト（`OracleJTADataSource`）をバインドします。バインド可能な `DataSource` オブジェクトは数種類あります。このデータベースをグローバル・トランザクションに含めるには、JTA 型のオブジェクトをバインドする必要があります。次に、グローバル・トランザクションのコンテキスト内で接続を取得する必要があります。つまり、`UserTransaction` オブジェクトの `begin` メソッドの後に接続を取得します。
2. グローバル・トランザクションが開始された後に、このメソッドは JNDI ネームスペースから `DataSource` オブジェクトを取得する必要があります。
3. `getConnection` メソッドを使用して、この `DataSource` オブジェクトから接続オブジェクトを取得します。

オブジェクト（クライアントまたはサーバーのオブジェクト）では、グローバル・トランザクションのコンテキスト内の `DataSource` オブジェクトを使用して、リモートのデータベースへの JDBC 接続をオープンします。クライアントでは、任意のクライアント JDBC ドライバを使用してリモートのデータベースへの接続をオープンできます。サーバー・オブジェクトでは、JDBC KPRB ドライバまたは JDBC サーバー側 Thin ドライバのいずれかを使用した場合に限り、データベースへの接続をオープンできます。

トランザクションに複数のデータベースが含まれている場合は、Oracle9*i* を 2 フェーズ・コミット・エンジンとして指定する必要があります。詳細は、17-23 ページの「2 フェーズ・コミット・エンジンの構成」を参照してください。

注意： この場合、Oracle JTA 実装はグローバル・トランザクションへの非 Oracle データベースの挿入をサポートしません。

---

2 フェーズ・コミット

グローバル・トランザクションの主な利点の 1 つに、トランザクション内の複数のデータベース・リソースを単一のユニットとして管理できる点があります。グローバル・トランザクションに複数のデータベース・リソースが含まれている場合は、2 フェーズ・コミット・エンジンを指定する必要があります。この 2 フェーズ・コミット・エンジンは、トランザクション内のすべてのデータベースに対する変更を管理するように設定された Oracle9*i* データベースです。2 フェーズ・コミット・エンジンは、トランザクションの終了時、すべてのデータベースの変更がすべてコミットまたはロールバックされているか確認する役割を果たします。

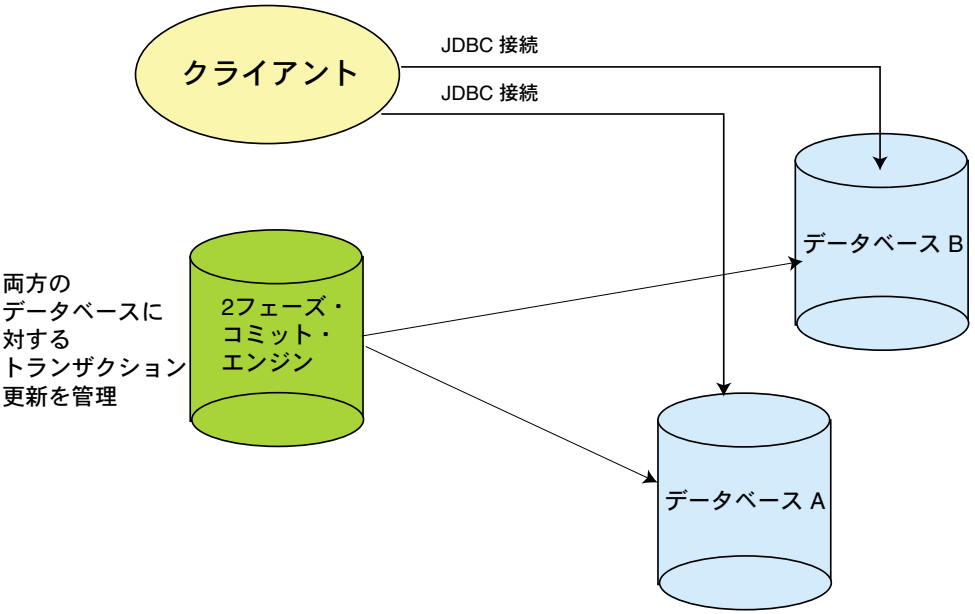
他方、グローバル・トランザクションに複数のサーバー・オブジェクトが含まれているが、データベース・リソースは 1 つしか含まれていない場合は、2 フェーズ・コミット・エンジンを指定する必要はありません。2 フェーズ・コミット・エンジンは、複数のデータベースに対して変更を同期化する場合にのみ必要になります。データベースが 1 つしか含まれない場合は、トランザクション・マネージャを使用して 2 フェーズ・コミットを実行できます。

注意： 2 フェーズ・コミット・エンジンには、どの Oracle9*i* データベースでも使用できます。サーバー・オブジェクトがあるデータベース、またはトランザクションにまったく関与していないデータベースでも使用できます。2 フェーズ・コミット・エンジンのセットアップの詳細は、17-23 ページの「2 フェーズ・コミット・エンジンの構成」を参照してください。

図 17-1 は、トランザクションで取得される 2 つのデータベースと、2 フェーズ・コミット・エンジンとして指定されたもう 1 つのデータベースを示しています。データベース A および B は、グローバル・トランザクション内で各データベースに対し JDBC 接続がオープンすると、取得されます。2 フェーズ・コミット・エンジンは、`UserTransaction` オブジェクト内で識別する必要があります。また、2 フェーズ・コミット・エンジンからトランザクションに含まれる各データベースへのデータベース・リンクを作成する必要があります。このセットアップの実行方法は、17-23 ページの「2 フェーズ・コミット・エンジンの構成」を参照してください。

グローバル・トランザクションが終了すると、2 フェーズ・コミット・エンジンは、データベース A および B で実行したすべての変更が同時にコミットまたはロールバックされているかを確認します。

図 17-1 グローバル・トランザクションの 2 フェーズ・コミット



JTA 制限事項

次に、JTA 仕様の中で、Oracle9*i* でサポートされない部分を示します。

ネストされたトランザクション

このリリースでは、ネストされたトランザクションはサポートされません。既存のトランザクションをコミットまたはロールバックする前に新しいトランザクションを開始しようとすると、トランザクション・サービスは、`SubtransactionsUnavailable` 例外を発行します。

相互運用性

このリリースで提供されるトランザクション・サービスは、他の JTA 実装と相互運用しません。

タイムアウト

Oracle では、`UserTransaction` オブジェクトを通じてのみ、データベース・リソース・アイドル・タイムアウトをサポートします。したがって、タイムアウトの使用はお薦めしません。

JTA の概要

次の項では、トランザクション境界を設定し、トランザクション内のデータベースを取得する方法の詳細をまとめます。以降の章では、これらの詳細について具体的な例を使用して説明していきます。ただし、ここでは参考ポイントを表にまとめてあります。

環境の初期化

バインドされた UserTransaction または DataSource オブジェクトを JNDI ネームスペースから取り出すには、JNDI による検索を実行する前に次の情報を指定する必要があります。

- ユーザー名やパスワードなどの認証情報
- ネームスペース URL

表 17-2 トランザクション・オブジェクトを取得するための環境のセットアップ

ソース	修飾子	環境のセットアップ
		セットアップには、認証情報およびネームスペース URL が含まれます。
クライアント	リモート・オブジェクトまたはリモート・データベース接続を取得します。	リモート JNDI プロバイダから UserTransaction を取得する前に、常に環境セットアップを用意しておく必要があります。真のクライアントから見ると、JNDI プロバイダはすべてリモートになります。
サーバー	■ セッション内アクティブ化を使用して、ローカル・オブジェクトまたはローカル・データベース接続を取得できます。	JNDI プロバイダがオブジェクトと同じデータベース内にある場合は、セッション内参照を使用できます。サーバーは独自のセッションを使用して検索を実行するため、セットアップは不要です。
	■ リモート・オブジェクトまたはリモート・データベース接続を取得します。	JNDI プロバイダはリモートであるため、サーバー・オブジェクトは常に環境セットアップを提供します。

データベース・リソースを取得するためのメソッド

JTA トランザクション内のデータベースを明示的に取得するには、DataSource オブジェクトを使用します。データベースを適切に取得するには、DataSource を正しくバインドする必要があります。また、取得メカニズムには 3 つの方法のいずれかを使用します。これらの方法について、次に説明します。

表 17-3 JDBC 2.0 DataSource の概要

	JDBC 2.0 DataSource
バインド	bindds コマンドを使用して、ネームスペースに JTA DataSource をバインドする必要があります。bindds コマンドには、-dstype jta オプションを含める必要があります。
リモート JNDI プロバイダから DataSource オブジェクトを取得	1. 認証情報とネームスペース URL を含む、環境データ用の Hashtable オブジェクトを提供します。 2. 「jdbc_access://」接頭辞を付けた JNDI 検索を通じて、DataSource オブジェクトを取得します。
ローカル JNDI プロバイダから DataSource オブジェクトを取得	セッション内アクティブ化を使用して DataSource オブジェクトを取得します。環境セットアップと「jdbc_access://」接頭辞は必要ありません。

単一フェーズ・コミットおよび 2 フェーズ・コミットの概要

表 17-4 は、単一フェーズ・コミットのシナリオをまとめたものです。JNDI バインドの要件およびアプリケーション実装ランタイムの要件を説明しています。

表 17-4 単一フェーズ・コミット

側面	説明
バインド	■ UserTransaction にはバインドは必要ありません。UserTransaction オブジェクトは自動的に作成されます。 ■ bindds コマンドを使用して、DataSource オブジェクトをバインドします。
ランタイム	■ JNDI lookup を通じて UserTransaction を取得します。 ■ ランタイムは、トランザクションの開始と終了の役割を担います。 ■ DataSource オブジェクトを使用してトランザクション内の SQL DML 文を管理する場合は、DataSource を取得してください。

表 17-5 は、2 フェーズ・コミットのシナリオをまとめたものです。

表 17-5 2 フェーズ・コミットの要件

側面	要件
UserTransaction のバインド 2 つのシナリオのいずれか:	シナリオ 1 では、UserTransaction から開始されたすべてのグローバル・トランザクションの実行に使用されるユーザー名とパスワードを指定して、UserTransaction をバインドします。 ■ UserTransaction オブジェクトを、2 フェーズ・コミット・エンジンの完全修飾されたアドレスとそのユーザー名およびパスワードにバインドします。 ■ トランザクションに関与する各データベースの DataSource オブジェクトを、2 フェーズ・コミット・エンジンからそれ自身への完全修飾されたパブリック・データベース・リンクにバインドします。
	シナリオ 2 では、ユーザー名とパスワードを指定しないで、UserTransaction をバインドします。つまり、UserTransaction の取得時には、そのトランザクションを完了するユーザー名をユーザー名として使用します。 ■ UserTransaction オブジェクトを、2 フェーズ・コミット・エンジンの完全修飾されたアドレスにバインドします。 ■ トランザクションに関与する各データベースの DataSource オブジェクトを、2 フェーズ・コミット・エンジンからそれ自身への完全修飾されたパブリック・データベース・リンクにバインドします。
DataSource のバインド	トランザクションに関与するデータベースごとに、JTA DataSource をバインドする必要があります。「システム管理」の項の説明に従って、パブリック・データベース・リンクを作成する必要があります。
システム管理	■ (「バインド」の項で説明したように) トランザクションを完了するユーザーには、含まれるすべてのデータベースでそのトランザクションをコミットする権限が必要です。ユーザーがトランザクションを確実に完了するためには、2 つの方法があります。 - UserTransaction オブジェクトにユーザー名がバインドされていない場合、UserTransaction を取得するユーザーがトランザクションの開始と終了に使用されます。つまり、すべてのデータベースに対してセッションをオープンするために、関与するすべてのデータベースにこのユーザーを作成する必要があります。 - UserTransaction オブジェクトを取得するユーザーとは別のユーザー名が UserTransaction オブジェクトにバインドされている場合、そのユーザーが開始していないトランザクションを完了できるように、UserTransaction オブジェクトにバインドされているユーザー名に対して明示的な権限を付与する必要があります。したがって、すべてのデータベースに対するセッションをオープンするために各データベースにユーザーを作成し、各データベースで CONNECT、REMOVE、CREATE SESSION および FORCE ANY TRANSACTION 権限を付与する必要があります。 ■ 2 フェーズ・コミット・エンジンから、関与する各データベースへのパブリック・データベース・リンクを作成します。
ランタイム	ランタイムの要件は、単一フェーズ・コミット表に記載した要件と同じです。

ネームスペースへのトランザクション・オブジェクトのバインド

ほとんどのグローバル・トランザクションについては、ネームスペースに次のようなオブジェクトの少なくとも 1 つをバインドする必要があります。

- UserTransaction オブジェクト
- DataSource オブジェクト

ネームスペースへの UserTransaction オブジェクトのバインド

ネームスペースに UserTransaction オブジェクトをバインドするには、bindut コマンドを使用します。このオブジェクトは、グローバル・トランザクションの境界に使用されます。

単一フェーズ・コミットおよび 2 フェーズ・コミットのいずれのトランザクションについても、sess_sh ツールの bindut コマンドを使用して、UserTransaction オブジェクトをバインドする必要があります。

UserTransaction オブジェクトをバインドするためのオプションは、次に説明するように、トランザクションが単一フェーズ・コミットまたは 2 フェーズ・コミットのどちらを使用するかによって決まります。

単一フェーズ・コミットでの UserTransaction のバインド 単一フェーズ・コミットでは、UserTransaction オブジェクトに JNDI バインド名が必要です。2 フェーズ・コミット・エンジンにアドレスを指定する必要はありません。たとえば、UserTransaction を単一フェーズ・コミット・トランザクションに存在する「/test/myUT」という名前にバインドするには、次を実行します。

```
bindut /test/myUT
```

sess_sh コマンドを使用して、UserTransaction オブジェクトを nsHost にあるネームスペースの名前「/test/myUT」にバインドするには、次を実行します。

```
sess_sh -service jdbc:oracle:thin:@nsHost:5521:ORCL -user SCOTT -password TIGER
& bindut /test/myUT
```

2 フェーズ・コミットでの UserTransaction のバインド 2 フェーズ・コミットのバインドでは、UserTransaction オブジェクトに JNDI バインド名と 2 フェーズ・コミット・エンジンのアドレスを指定する必要があります。2 フェーズ・コミット・エンジンのアドレスは、bindut コマンドで指定し、JDBC URL または sess_iiop URL のいずれかを指定できます。

注意： クライアントで UserTransaction を取得するには、bindut コマンド内で指定したのと同じ情報が必要です。

また、UserTransaction にはユーザー名とパスワードをバインドすることもできます。

- UserTransaction にユーザー名とパスワードをバインドしない場合、UserTransaction を取得するユーザーは、関与するすべてのデータベースで 2 フェーズ・コミットのコミットまたはロールバックを実行するユーザーと同一になります。
- UserTransaction にユーザー名とパスワードをバインドする場合は、関与するすべてのデータベースで 2 フェーズ・コミットをコミットまたはロールバックするときに、バインドしたユーザー名が使用されます。トランザクションは、UserTransaction オブジェクトを取得するユーザーによって開始され、UserTransaction オブジェクトがバインドされているユーザーによって完了します。

2 フェーズ・コミット・トランザクションをコミットまたはロールバックするのに使用されるユーザー名は、2 フェーズ・コミット・エンジンと、トランザクションに関与する各データベースに作成する必要があります。ユーザー名を作成する必要があるのは、2 フェーズ・コミット・エンジンから関与する各データベースに対してデータベース・リンクを使用してセッションをオープンするためです。次に、これらの各データベースに接続できるように、CONNECT、RESOURCE、CREATE SESSION 権限を付与する必要があります。たとえば、トランザクションを完了するのに必要なユーザーが SCOTT の場合、2 フェーズ・コミット・エンジンとトランザクションに関与する各データベースで、次のような操作を実行する必要があります。

```
CONNECT SYSTEM/MANAGER;
CREATE USER SCOTT IDENTIFIED BY SCOTT;
GRANT CONNECT, RESOURCE, CREATE SESSION TO SCOTT;
```

また、UserTransaction オブジェクトにユーザー名とパスワードをバインドした場合は、トランザクションの開始時に使用されたのとは別のユーザー名がトランザクションの終了に使用されます。2 人の別ユーザーがトランザクションを開始および終了できるようにするには、そのトランザクションに関与するすべてのデータベースで FORCE ANY TRANSACTION 権限を付与する必要があります。UserTransaction オブジェクトにバインドされているユーザーが SCOTT の場合、前述の権限に加えて次のような操作を行う必要があります。

```
GRANT FORCE ANY TRANSACTION TO SCOTT;
```

UserTransaction を「/test/myUT」という名前にバインドし、JDBC URL を使用して 2 フェーズ・コミット・エンジンを「2pcHost」に指定するには、次を実行します。

```
bindut /test/myUT -url jdbc:oracle:thin:@2pcHost:5521:ORCL
```

sess_iiop URL を使用して 2 フェーズ・コミット・エンジンを dbsun.mycompany.com に指定し、UserTransaction をネームスペースにバインドするには、次を実行します。

```
bindut /test/myUT -url sess_iiop://dbsun.mycompany.com:2481:ORCL
```

トランザクションがコミットされると、UserTransaction は url オプションで指定された 2 フェーズ・コミット・エンジンに対して、含まれるすべてのデータベースへの変更をすべてコミットするように指示します。この例の場合、UserTransaction オブジェクトにはユーザー名とパスワードがバインドされていないため、JNDI ネームスペースから UserTransaction を取得するユーザーがトランザクションの開始と終了に使用されます。したがって、関与するすべてのデータベースと 2 フェーズ・コミット・エンジンに、このユーザーが存在している必要があります。UserTransaction は、トランザクションに関与するすべてのデータベースを追跡し、2 フェーズ・コミット・エンジンはこれらのデータベースへのデータベース・リンクを使用してトランザクションを完了します。

注意： 2 フェーズ・コミット・エンジンを変更した場合、トランザクションに関与するすべての DataSource オブジェクトにあるすべてのデータベース・リンクを更新して、UserTransaction を再バインドする必要があります。

ネームスペースへの DataSource オブジェクトのバインド

JNDI ネームスペースに DataSource オブジェクトをバインドするには、bindds コマンドを使用します。ローカル・データベースを含め、グローバル・トランザクション内の任意のデータベースを取得するには、JTA DataSource オブジェクトをバインドして、トランザクションに含まれる各データベースを識別する必要があります。シナリオによって、使用する DataSource のタイプは異なります。ただし、JTA トランザクションに使用するには、OracleJTADataSource オブジェクトとも呼ばれる JTA DataSource オブジェクトをバインドして、トランザクションに含まれる各データベースを識別する必要があります。他の DataSource オブジェクトの説明は、『Oracle9*i* Java Tools Reference』の「sess_sh ツール」の「bindds コマンド」を参照してください。

単一フェーズ・コミットのシナリオ 単一フェーズ・コミットの場合、トランザクションに含まれるのは 1 つのデータベースのみです。複数のデータベースに対する更新を調整する必要がないので、コーディネータの指定は不要です。そのかわりに、OracleJTADataSource オブジェクト内にこのデータベースの JNDI バインド名と URL アドレス情報を指定します。トランザクション・コーディネータにデータベース・リンクを指定する必要はありません。

DataSource オブジェクトをネームスペースにバインドするには、sess_sh ツールの bindds コマンドを使用します。このコマンドの詳細は、『Oracle9*i* Java Tools Reference』を参照してください。たとえば、bindds コマンドを使用して、OracleJTADataSource を単一フェーズ・コミット・トランザクションに存在する「/test/empDS」という名前にバインドするには、次を実行します。

```
bindds /test/empDS -url jdbc:oracle:thin:@empHost:5521:ORCL -dstype jta
```

DataSource オブジェクトをネームスペースにバインドすると、サーバーはグローバル・トランザクション内のデータベースを取得できます。

2 フェーズ・コミットのシナリオ グローバル・トランザクションに複数のデータベースが含まれる場合は、トランザクション・コーディネータとして構成された Oracle9*i* データベースである、2 フェーズ・コミット・エンジンが必要になります。基本的に、2 フェーズ・コミット・エンジンには、トランザクションに関与する各データベースへのデータベース・リンクが必要です。トランザクションが終了すると、トランザクション・マネージャは 2 フェーズ・コミット・エンジンに対して、関与するすべてのデータベースの変更をすべてコミットまたはロールバックするかを調整するように指示します。

この調整を可能にするには、次の構成を行う必要があります。

1. システム管理者は、2 フェーズ・コミット・エンジン（Oracle9*i* データベース）からトランザクションに関与する各データベースへの完全修飾されたパブリック・データベース・リンクを作成する必要があります。`OracleJTADataSource` オブジェクトをバインドするときには、これらのデータベース・リンク名を含めてください。
2. トランザクション内の各データベースに、JTA `DataSource`（`OracleJTADataSource`）オブジェクトをバインドします。バインド・コマンドには、次を含める必要があります。
   a. オブジェクトの JNDI バインド名
   b. データベースへの接続を作成するための URL
   c. 2 フェーズ・コミット・エンジンから、このデータベースへの完全修飾されたパブリック・データベース・リンク

注意： 2 フェーズ・コミットの場合、`DataSource` オブジェクトは、2 フェーズ・コミット・エンジンに関連してバインドされます。2 フェーズ・コミット・エンジンを変更した場合、すべてのデータベース・リンクを更新して、関連するすべての `DataSource` オブジェクトと `UserTransaction` オブジェクトをリバインドする必要があります。

---

2 フェーズ・コミット・エンジン上に作成されたデータベース・リンク名として `2pcToEmp` を指定して、`empDS` JTA `DataSource` をネームスペースにバインドするには、次を実行します。

```
% bindds /test/empDS -url jdbc:oracle:thin:@dbsun:5521:ORCL
                    -dstype jta -dblink 2pcToEmp.oracle.com
```

JTA クライアント側での境界設定

JTA の場合、クライアント側で境界が設定されるトランザクションは、UserTransaction インタフェースを使用してプログラム的に境界が設定されます（詳細は、17-4 ページの「UserTransaction インタフェース」を参照してください）。bindds コマンドを使用して、ネームスペースに UserTransaction オブジェクトをバインドする必要があります（詳細は、17-13 ページの「ネームスペースへの UserTransaction オブジェクトのバインド」を参照してください）。クライアント側でのトランザクション境界では、クライアントがトランザクションを制御します。クライアントは、UserTransaction begin メソッドをコールしてグローバル・トランザクションを開始し、commit メソッドまたは rollback メソッドをコールしてトランザクションを終了します。さらに、クライアントは常に、認証情報およびネームスペース・ロケーション URL を Hashtable に含む環境をセットアップする必要があります。

クライアントからトランザクション内にリモート・データベースを含めるには、JTA DataSource としてネームスペースにバインドされた DataSource オブジェクトを使用する必要があります。次に、トランザクションが開始された後に、DataSource オブジェクトの getConnection メソッドをコールして、データベースをグローバル・トランザクションに含めます。詳細は、17-6 ページの「リソースの取得」を参照してください。

トランザクション境界を設定するには、クライアントのランタイムで次のことを実行する必要があります。

1. 環境データ用に用意した Hashtable オブジェクトを、ネームスペース・アドレスおよび認証情報により初期化します。
2. クライアント論理内のネームスペースから UserTransaction オブジェクトを取り出します。クライアントから UserTransaction オブジェクトを取り出す場合は、URL の JNDI 名の前に「jdbc_access://」接頭辞を付ける必要があります。
3. UserTransaction.begin() を使用して、クライアント内でグローバル・トランザクションを開始します。
4. 次に示すように、指定したデータベースへの接続をオープンして、トランザクションに含めるデータベース・リソースを取得します。
   a. クライアント論理内のネームスペースから DataSource オブジェクトを取得します。任意のクライアントから DataSource オブジェクトを取り出す場合は、URL の JNDI 名の前に「jdbc_access://」接頭辞を付ける必要があります。
   b. DataSource.getConnection メソッドを使用して、データベースへの接続をオープンします。
5. オブジェクト参照を取得します。
6. トランザクションに含めるオブジェクト・メソッドをコールします。

7. 取得したデータベースに対して SQL DML 文をコールします。SQL DDL により、グローバル・トランザクションを強制終了するローカル・トランザクションが作成されます。したがって、JTA トランザクション内では SQL DLL を実行できません。

8. `UserTransaction.commit()` または `UserTransaction.rollback()` でトランザクションを終了します。

例 17-1 は、トランザクション内でクライアントがサーバー・オブジェクトをコールし、1 つのデータベースを取得する方法を示しています。

例 17-1 クライアント側で境界が設定されたトランザクションでの Employee クライアント・コード

クライアントを起動する前に、まず JNDI ネームスペースに `UserTransaction` オブジェクトと `DataSource` オブジェクトをバインドする必要があります。これらのオブジェクトをバインドする方法の詳細は、17-13 ページの「ネームスペースへの UserTransaction オブジェクトのバインド」および 17-15 ページの「ネームスペースへの DataSource オブジェクトのバインド」を参照してください。

```
sess_sh -service jdbc:oracle:thin:@nsHost:5521:ORCL -user SCOTT -password TIGER
> bindut /test/myUT
> bindds /test/DataSource/empDB -url jdbc:oracle:thin:@empHost:5521:ORCL
                                -dstype jta
```

クライアント・アプリケーションの開発

```
//Set up the service URL to where the UserTransaction object
//is bound. Since from the client, the connection to the database
//where the namespace is located can be communicated with over either
//a Thin or OCI JDBC driver. This example uses a Thin JDBC driver.
String namespaceURL = "jdbc:oracle:thin:@nsHost:1521:ORCL";

// lookup usertransaction object in the namespace
//1.(a) Authenticate to the database.
// create InitialContext and initialize for authenticating client
Hashtable env = new Hashtable ();
env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put (Context.SECURITY_PRINCIPAL, "SCOTT");
env.put (Context.SECURITY_CREDENTIALS, "TIGER");
env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
//1.(b) Specify the location of the namespace where the transaction objects
//   are bound.
env.put(jdbc_accessURLContextFactory.CONNECTION_URL_PROP, namespaceURL);
Context ic = new InitialContext (env);

//2. Retrieve the UserTransaction object from JNDI namespace
ut = (UserTransaction)ic.lookup ("jdbc_access://test/myUT");
```

```
//3. Start the transaction
ut.begin();

//4.(a) Retrieve the DataSource (that was previously bound with bindds in
// the namespace. After retrieving the DataSource...
// get a connection to a database. You need to provide authentication info
// for a remote database lookup, similar to what you would do from a client.
// In addition, if this was a two-phase commit transaction, you must provide
// the username and password.
DataSource ds = (DataSource)ic.lookup ("jdbc_access://test/empDB");

//4.(b). Get connection to the database through DataSource.getConnection
// in this case, the database requires the same username and password as
// set in the environment.
Connection conn = ds.getConnection ("SCOTT", "TIGER");

//5. Retrieve the server object reference
// lookup employee object in the namespace
Employee employee = (Employee)ic.lookup (sessiiopURL + objectName);

//6. Perform business logic.
...

//7. Close the database connection.
conn.close ();

//8. End the transaction
//Commit the updated value
ut.commit ();
}
```

JTA サーバー側でのデータベースの取得

サーバーが配置されているデータベースにもネームスペースが含まれている場合は、セッション内参照を実行できます。つまり、環境設定を行う必要はありません。ただし、サーバーがネームスペースとは別のマシン上に配置されている場合は、17-17 ページの「JTA クライアント側での境界設定」の説明に従って、環境を正確に初期化する必要があります。この項では、ネームスペースとサーバー・オブジェクトは同じマシン上に存在し、`UserTransaction` および `DataSource` オブジェクトのセッション内参照を示していると仮定します。

サーバーで、トランザクション境界を設定し、トランザクション内のデータベースを取得するには、次を実行します。

1. システム管理者は、必要な JTA オブジェクトをネームスペースにバインドします。
   a. ネームスペース内に `UserTransaction` オブジェクトをバインドします。
   b. ネームスペース内に `DataSource` オブジェクトをバインドします。

   注意： 複数のデータベースを使用している場合は、2 フェーズ・コミット用にセットアップする必要があります。詳細は、17-23 ページの「2 フェーズ・コミット・エンジンの構成」を参照してください。

2. セッション内アクティブ化を使用してネームスペースから `UserTransaction` オブジェクトを取り出します。環境および URL 接頭辞は必要ありません。参照には、バインドされたオブジェクトの JNDI 名のみが必要です。
3. `UserTransaction.begin()` を使用して、クライアント内でグローバル・トランザクションを開始します。
4. 次に示すように、指定したデータベースへの JDBC 接続をオープンして、トランザクションに含めるデータベース・リソースを取得します。
   a. セッション内アクティブ化を使用してネームスペースから `DataSource` オブジェクトを取得します。URL 接頭辞は必要ありません。参照には、バインドされたオブジェクトの JNDI 名のみが必要です。
   b. `DataSource.getConnection()` メソッドを使用して、データベースへの接続をオープンします。このサーバーに対する認証に使用されたユーザー名とパスワードが適切な認証情報である場合は、引数を指定する必要はありません。ただし、認証に別のユーザー名やパスワードが必要な場合は、`getConnection()` メソッドのコールに対する入力パラメータとしてこれらを指定します。
5. 取得したデータベースに対して SQL DML 文をコールします。
6. オープンされているデータベース接続をすべてクローズします。
7. `UserTransaction.commit()` または `UserTransaction.rollback()` でトランザクションを終了します。

例 17-2 単一フェーズ・トランザクションでのデータベースの取得

次の例は、トランザクションでサーバー・オブジェクトがローカル・データベースを取得する方法を示しています。グローバル・トランザクションを初期化し、DataSource.getConnection() メソッドを使用して接続を取得して、ローカル・データベースに対して文を実行します。これらの文は、グローバル・トランザクションがコミットされるとコミットされます。

UserTransaction および DataSource オブジェクトは、このホストにローカルなネームスペースにバインドされているため、セッション内アクティブ化を実行できます。つまり、環境情報は不要で（2 フェーズ・コミットのシナリオを実行しない限り）、参照には「jdbc_access://」接頭辞のない JNDI 名のみ使用します。

```
//retrieve the initial context.
InitialContext ic = new InitialContext ();

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

//start the transaction
ut.begin ();

// get a connection to the local database. If this was a two-phase commit
// transaction, you would provide the username and password for the 2pc engine
DataSource ds = (DataSource)ic.lookup ("/test/empDS");

// get connection to the local database through DataSource.getConnection
Connection conn = ds.getConnection ("SCOTT", "TIGER");

//perform your SQL against the database.
//prepare and execute a sql statement. retrieve the employee's selected benefits
PreparedStatement ps =
    conn.prepareStatement ("update emp set ename = :(employee.name),
sal = :(employee.salary) where empno = :(employee.number)");
    .... //do work
 ps.close();
}

//close the connection
conn.close();

// commit the transaction
ut.commit ();

//return the employee information.
return new EmployeeInfo (name, empno, (float)salary);
```

例 17-3 SQLJ を使用した明示的な取得

例 17-2 に示すように、JNDI プロバイダから JTA DataSource を取得し、接続を取得して、その接続からコンテキストを取得し、そのコンテキストを SQLJ コマンドラインに指定します。

```
//retrieve the initial context.
InitialContext ic = new InitialContext ();

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

//start the transaction
ut.begin ();

// get a connection to the local database. If this was a two-phase commit
// transaction, you would provide the username and password for the 2pc engine
DataSource ds = (DataSource)ic.lookup ("/test/empDS");

// get connection to the local database through DataSource.getConnection
Connection conn = ds.getConnection ("SCOTT", "TIGER");

//setup the context for issuing SQLJ against the database
DefaultContext defCtx = new DefaultContext (conn);

//issue SQL DML statemetns against the database
#sql [defCtx] { update emp set ename = :(remoteEmployee.name),
sal = :(remoteEmployee.salary)
  where empno = :(remoteEmployee.number) };

//close the connection
conn.close();

// commit the transaction
ut.commit ();

//return the employee information.
return new EmployeeInfo (name, empno, (float)salary);
```

2 フェーズ・コミット・エンジンの構成

グローバル・トランザクションに複数のデータベースが含まれる場合、これらのリソースに対する変更すべてを同時にコミットまたはロールバックする必要があります。すなわち、トランザクションが終了すると、トランザクション・マネージャがコーディネータ（2 フェーズ・コミット・エンジンとも呼ばれます）と対話して、含まれるすべてのデータベースに対するすべての変更をコミットまたはロールバックするように指示します。2 フェーズ・コミット・エンジンは、次のように構成された Oracle9*i* データベースです。

- データベース自身から、トランザクションに含まれる各データベースへの完全修飾されたデータベース・リンク。トランザクションが終了すると、2 フェーズ・コミット・エンジンは、完全修飾されたデータベース・リンクを通じて、トランザクションに含まれている各データベースと対話します。
- トランザクションに含まれる各データベースへのセッションを作成するように指定されたユーザー。このユーザーにはコミットまたはロールバックを実行する権限を付与します。トランザクションに含まれるすべてのデータベースに対話を行うユーザーを作成し、そのユーザーに適切な権限を付与する必要があります。

この調整を可能にするには、次の構成を行う必要があります。

1. Oracle9*i* データベースを、2 フェーズ・コミット・エンジンとして指定します。
2. （CREATE DATABASE LINK コマンドを使用して）2 フェーズ・コミット・エンジンから、グローバル・トランザクションに含まれる各データベースへのパブリック・データベース・リンクを構成します。これは、2 フェーズ・コミット・エンジンがトランザクションの最後に各データベースと通信する場合に必要です。JTA DataSource（OracleJTADataSource）オブジェクトをバインドするときには、これらのデータベース・リンク名を含めてください。
3. トランザクション内の各データベースに、JTA DataSource（OracleJTADataSource）オブジェクトをバインドします。bindds コマンドには、次を含める必要があります。
   a. オブジェクトの JNDI バインド名
   b. データベースへの接続を作成するための URL
   c. 2 フェーズ・コミット・エンジンからこのデータベースへの完全修飾されたデータベース・リンク

   データベースの DataSource をネームスペースにバインドするときに、各データベースごとに、bindds の -dblink オプションで完全修飾されたデータベース・リンク名を指定します。

```
bindds /test/empDS -url jdbc:oracle:thin:@empHost:5521:ORCL
-dstype jta -dblink 2pcToEmp.oracle.com
```

注意： 2 フェーズ・コミットの場合、DataSource オブジェクトは、2 フェーズ・コミット・エンジンに関連してバインドされます。2 フェーズ・コミット・エンジンを変更した場合、すべてのデータベース・リンクを更新して、関連するすべての DataSource オブジェクトと UserTransaction オブジェクトをリバインドする必要があります。

---

4. 2 フェーズ・コミット・エンジンに、2 フェーズ・コミットを実行するユーザーを作成します。このユーザーが、トランザクションに関与する各リソースに対してセッションをオープンし、トランザクションを終了します。このためには、各データベースにユーザーを作成し、そのユーザーに対して CONNECT、RESOURCE および CREATE SESSION 権限を付与する必要があります。トランザクションを開始したユーザーとは別のユーザーがトランザクションを終了する場合は、FORCE ANY TRANSACTION 権限も付与する必要があります。これらの権限は、トランザクションに含まれるすべてのデータベースで付与してください。

   FORCE ANY TRANSACTION 権限を付与する必要があるかどうかは、UserTransaction オブジェクトにユーザー名とパスワードをバインドしたかどうかで決まります。

   - UserTransaction にユーザー名とパスワードをバインドしない場合、UserTransaction を取得するユーザーは、関与するすべてのデータベースで 2 フェーズ・コミットのコミットまたはロールバックを実行するユーザーと同一になります。
   - UserTransaction にユーザー名とパスワードをバインドする場合は、関与するすべてのデータベースで 2 フェーズ・コミットをコミットまたはロールバックするときに、バインドしたユーザー名が使用されます。トランザクションは、UserTransaction オブジェクトを取得するユーザーによって開始されます。

   2 フェーズ・コミット・エンジンから、関与する各データベースに対してセッションをオープンできるようにするには、2 つのタイプのユーザーを作成する必要があります。次に、これらの各データベースに接続できるように、CONNECT、RESOURCE、CREATE SESSION 権限を付与する必要があります。たとえば、トランザクションを完了するのに必要なユーザーが SCOTT の場合、2 フェーズ・コミット・エンジンとトランザクションに関与する各データベースで、次のような操作を実行する必要があります。

```
CONNECT SYSTEM/MANAGER;
CREATE USER SCOTT IDENTIFIED BY SCOTT;
GRANT CONNECT, RESOURCE, CREATE SESSION TO SCOTT;
```

   また、UserTransaction オブジェクトにユーザー名とパスワードをバインドした場合は、トランザクションの開始時に使用されたのとは別のユーザー名がトランザクションの終了に使用されます。2 人の別ユーザーがトランザクションを開始および終了できるようにするには、そのトランザクションに関与するすべてのデータベースで FORCE ANY TRANSACTION 権限を付与する必要があります。

ユーザー名を UserTransaction にバインドした場合の利点は、この UserTransaction オブジェクトから開始されたすべてのトランザクションが常にグローバル・ユーザーによってコミットされるものとして扱われることです。したがって、複数の JTA トランザクションがある場合でも、1 人のユーザーを作成し、関与するすべてのデータベースに対する権限をそのユーザーに付与するのみですみます。

たとえば、UserTransaction オブジェクトにバインドされているユーザーが SCOTT の場合、前述の権限に加えて次のような操作を行う必要があります。

```
GRANT FORCE ANY TRANSACTION TO SCOTT;
```

5. ネームスペースに UserTransaction をバインドします。2 フェーズ・コミット・エンジンの完全修飾されたデータベース・アドレスを指定する必要があります。この時点で、(手順 3 の説明に基づいて) ユーザー名とパスワードをバインドするかどうかを決定してください。次の例では、UserTransaction にグローバル・ユーザー名がバインドされているものと想定しています。

```
bindut /test/myUT -url sess_iiop://dbsun.mycompany.com:2481:ORCL
-user SCOTT -password TIGER
```

例 17-4 2 フェーズ・コミットの例

次の例は、セッション内アクティブ化を実行して、ローカルでバインドされた UserTransaction および DataSource オブジェクトの両方を取得するサーバーを示しています。UserTransaction には、2 フェーズ・コミット・エンジンの URL、ユーザー名およびパスワードがバインドされています。DataSource オブジェクトはすべて、適切なデータベース・リンクにバインドされています。

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

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

//remote lookup requires environment setup
Hashtable env = new Hashtable ();
env.put (Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
env.put (Context.SECURITY_PRINCIPAL, user);
env.put (Context.SECURITY_CREDENTIALS, password);
env.put (Context.SECURITY_AUTHENTICATION, ServiceCtx.NON_SSL_LOGIN);
env.put(jdbc_accessURLContextFactory.CONNECTION_URL_PROP, namespaceURL);
InitialContext ic2 = new InitialContext (env);

//retrieve the DataSource for the remote database
DataSource remoteDS = (DataSource)ic2.lookup ("jdbc_access://test/NewYorkDS");
```

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

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

DataSource オブジェクトの動的作成

複数のデータベース・リソースで使用されるネームスペースに単一の DataSource オブジェクトをバインドする場合は、次を実行する必要があります。

1. URL、ホスト、ポート、SID またはドライバの種類を指定せずに DataSource をバインドします。この場合、次のように -dstype jta オプションのみを使用して、bindds コマンドを実行します。

   ```
   sess_sh -service jdbc:oracle:thin:@nsHost:5521:ORCL -user SCOTT -password TIGER
   & bindds /test/empDS -dstype jta
   ```

2. コードの DataSource を取得します。lookup を実行する場合は、戻されたオブジェクトを DataSource ではなく、OracleJTADataSource にキャストする必要があります。Oracle 固有のバージョンの DataSource クラスには、DataSource プロパティを設定するメソッドが含まれています。
3. 次のプロパティを設定します。
   - OracleJTADataSource.setURL メソッドで URL を設定します。
   - 2 フェーズ・コミット・エンジンを使用する場合は、OracleJTADataSource.setDBLink メソッドで完全修飾されたデータベース・リンクを設定します。
4. 他の例で示すように、OracleJTADataSource.getConnection メソッドを使用して接続を取得します。

例 17-5 一般的な DataSource の取得

次の例では、汎用的にバインドされている DataSource を、セッション内参照を使用してネームスペースから取得し、すべての関連フィールドを初期化します。

```
//retrieve an in-session generic DataSource object
OracleJTADataSource ds =
      (OracleJTADataSource)ic.lookup ("java:comp/env/test/empDS");

//set all relevant properties for my database
//URL is for a local database so use the KPRB URL
ds.setURL ("jdbc:oracle:kprb:");
//Used in two-phase commit, so provide the fully qualified database link that
//was created from the two-phase commit engine to this database
ds.setDBLink("localDB.oracle.com");

//Finally, retrieve a connection to the local database using the DataSource
Connection conn = ds.getConnection ();
```

JDBC の制限事項

サーバー・オブジェクトで JDBC コールを使用してデータベースを更新する場合、アクティブなトランザクション・コンテキストがあるときには、JDBC を使用してトランザクション・サービスを実行しない（JDBC 接続でメソッドをコールしない）でください。JDBC トランザクション管理メソッドはコード化しないでください。たとえば、次のようになります。

```
Connection conn = ...
...
conn.commit();  // DO NOT DO THIS!!
```

前述の例を実行すると、SQLException が発生します。このような方法を使用せずに、グローバル・トランザクションを処理するには、取得した UserTransaction オブジェクトを使用してコミットする必要があります。JDBC 接続を使用してコミットする場合は、グローバル・トランザクションではなく、ローカル・トランザクションでコミットを行います。その接続がグローバル・トランザクションに含まれている場合は、グローバル・トランザクションでローカル・トランザクションをコミットしようとするとエラーが発生します。

同様に、JDBC を使用して SQL コミットまたはロールバックを直接実行しないでください。UserTransaction インタフェースを使用して直接トランザクションを処理するようにオブジェクトをコーディングしてください。

グローバル・トランザクション内では、ローカル・トランザクションを実行できません。実行すると、次のようなエラーが発生します。

```
ORA-2089 下位セッションに COMMIT は使用できません。
```

SQL コマンドの中には、暗黙的にローカル・トランザクションを実行するものがあります。CREATE TABLE などの SQL DDL 文は、暗黙的にローカル・トランザクションを起動してコミットします。DDL 文の実行対象であるデータベースを取得したグローバル・トランザクションが含まれていると、そのグローバル・トランザクションは失敗します。

18

上級トピック

この章では、上級 JDBC トピックについて説明します。次の項目が含まれます。

- JDBC およびグローバリゼーション・サポート
- JDBC クライアント側セキュリティ機能
- アプレット内の JDBC
- サーバー上の JDBC: サーバー側内部ドライバ

JDBC およびグローバリゼーション・サポート

簡単な概要の後に、この項では次の項目を説明します。

- JDBC ドライバがグローバリゼーション・サポート変換を行う方法
- グローバリゼーション・サポートとオブジェクト型
- Thin ドライバによる CHAR および VARCHAR2 データ・サイズ制限

Oracle の JDBC ドライバは、グローバリゼーション・サポート（以前の NLS）をサポートしています。グローバリゼーション・サポートにより、Oracle がサポートする任意のキャラクタ・セットで、データの取出しおよびデータベースへのデータの挿入が可能です。クライアントとサーバーで異なるキャラクタ・セットを使用する場合、ドライバでは、データベース・キャラクタ・セットとクライアント・キャラクタ・セット間の変換がサポートされます。

グローバリゼーション・サポート、グローバリゼーション・サポート環境変数および Oracle がサポートするキャラクタ・セットの詳細は、5-27 ページの「Oracle 文字データ型のサポート」と『Oracle9*i* グローバリゼーション・サポート・ガイド』を参照してください。データベース・キャラクタ・セットとその作成方法の詳細は、『Oracle9*i* データベース・リファレンス・マニュアル』を参照してください。

次に、各国語キャラクタ・セットの変換を必要とする一般的な JDBC 用 Java メソッドの例をいくつか示します。

- `java.sql.ResultSet` のメソッド、`getString()` および `getUnicodeStream()` は、それぞれ Java 文字列および Unicode 文字ストリームとしてデータベースから値を戻します。
- `oracle.sql.CLOB` のメソッド、`getCharacterStream()` は、`CLOB` の内容を Unicode ストリームとして戻します。
- `oracle.sql.CHAR` のメソッド、`getString()`、`toString()` および `getStringWithReplacement()` は、次のデータを文字列に変換します。
  - `getString()`: `CHAR` オブジェクトによって表された一連の文字を文字列に変換し、Java `String` オブジェクトを戻します。
  - `toString()`: これは `getString()` と同じですが、キャラクタ・セットが認識されない場合、`toString()` は `CHAR` データの 16 進表現を戻します。
  - `getStringWithReplacement()`: これは `getString()` と同じですが、この `CHAR` オブジェクトのキャラクタ・セットに Unicode 表現がない文字はデフォルトの置換文字に置換される点が異なります。

JDBC ドライバがグローバリゼーション・サポート変換を行う方法

Oracle JDBC ドライバが Java アプリケーションのためにキャラクタ・セットを変換する方法は、データベースで使用されているキャラクタ・セットによって異なります。最も単純なのは、データベースで US7ASCII または WE8ISO8859P1 キャラクタ・セットを使用している場合です。この場合、ドライバではデータベース・キャラクタ・セットと Java アプリケーションで使用される UCS-2 間で直接データ変換が実行されます。

US7ASCII でも WE8ISO8859P1 でもないキャラクタ・セット（日本語や韓国語など）を利用するデータベースを操作する場合は、ドライバはデータをまず UTF-8 に変換してから（この操作はサーバー側内部ドライバには適用されません）、UCS-2 に変換します。たとえば、ドライバは、US7ASCII でも WE8ISO8859P1 でもないキャラクタ・セットの CHAR データおよび VARCHAR2 データを常に変換します。RAW データは変換しません。

注意： JDBC ドライバは、キャラクタ・セットの変換をすべて透過的に行います。変換の実行にユーザーの操作は必要ありません。

JDBC OCI ドライバおよびグローバリゼーション・サポート

JDBC OCI ドライバの場合、クライアント・キャラクタ・セットは、クライアントのインストール時に設定される環境変数 NLS_LANG に格納されています。言語および地域設定は、デフォルトで Java VM ローカル設定に設定されます。

これらのパラメータには、データベースの作成中に決定されるサーバー側設定もあることに注意してください。このため、キャラクタ・セットの変換時に、JDBC OCI ドライバでは次のような点を考慮する必要があります。

- データベース・キャラクタ・セットおよび言語
- クライアントのキャラクタ・セットおよび言語
- Java アプリケーションのキャラクタ・セット : UCS-2

JDBC OCI ドライバは、サーバーのデータをデータベース・キャラクタ・セットでクライアントに転送します。環境変数 NLS_LANG の値により、ドライバはキャラクタ・セットの変換を次のいずれかの方法で処理します。

- NLS_LANG を指定しないか、US7ASCII または WE8ISO8859P1 キャラクタ・セットを指定する場合、JDBC OCI ドライバは、Java を使用して US7ASCII または WE8ISO8859P1 と UCS-2 間で、キャラクタ・セットを直接変換します。

または

- NLS_LANG が US7ASCII でも WE8ISO8859P1 でもないキャラクタ・セットに設定されている場合は、ドライバはクライアント上の NLS_LANG パラメータの値を UTF-8 に変更します。これは自動的に行われ、ユーザーの操作は必要ありません。OCI は NLS_LANG 設定を使用して、データベース・キャラクタ・セットから UTF-8 にデータを変換します。次に、JDBC ドライバによって UTF-8 データが UCS-2 に変換されます。

注意：

- ドライバは NLS_LANG キャラクタ・セットを UTF-8 に設定して、Java で実行する変換回数を最少にします。データベース・キャラクタ・セットから UTF-8 への変換は、C 言語で行われます。
- UTF-8 への変換は、JDBC アプリケーション処理のためのみに行います。
- NLS_LANG パラメータの詳細は、『Oracle9*i* グローバリゼーション・サポート・ガイド』を参照してください。

JDBC Thin ドライバおよびグローバリゼーション・サポート

JDBC Thin ドライバを使用する場合、Oracle クライアントのインストレーションはないことが想定されます。グローバリゼーション・サポート変換は、別の方法で処理する必要があります。

言語と地域 Thin ドライバによって、JVM user.language プロパティの Java ロケールから言語設定と地域設定（NLS_LANGUAGE と NLS_TERRITORY）が取得されます。日付書式（NLS_DATE_FORMAT）は地域設定に従って設定されます。

キャラクタ・セット データベース・キャラクタ・セットが US7ASCII または WE8ISO8859P1 に設定されている場合、データは変換されずにクライアントに転送されます。ドライバでは、このキャラクタ・セットを Java で UCS-2 に変換します。

データベース・キャラクタ・セットが US7ASCII または WE8ISO8859P1 以外の場合は、サーバーはデータを UTF-8 に変換してからクライアントに転送します。クライアント上では、JDBC Thin ドライバが Java でデータを UCS-2 に変換します。

サーバー側内部ドライバおよびグローバリゼーション・サポート

サーバー内で実行している JDBC コードがデータベースにアクセスした場合は、JDBC サーバー側内部ドライバがデータベース・キャラクタ・セットに基づいてキャラクタ・セットを変換します。Java プログラムのターゲット・キャラクタ・セットはすべて UCS-2 です。

グローバリゼーション・サポートとオブジェクト型

Oracle JDBC クラス・ファイルの classes12.zip と classes111.zip によって、Thin ドライバと OCI ドライバでグローバリゼーション・サポートが提供されます。このファイルには、Oracle オブジェクト型またはコレクション型の一部として使用されていない CHAR および NCHAR データ型のすべての Oracle キャラクタ・セットに完全なグローバリゼーション・サポートを提供するために必要なすべてのクラスが入っています。CHAR および NCHAR データ型の詳細は、5-27 ページの「Oracle 文字データ型のサポート」を参照してください。

ただし、Oracle オブジェクトおよびコレクションの CHAR と VARCHAR のデータ部分の場合は、JDBC クラス・ファイルは、一般に使用される次のキャラクタ・セットのみをサポートします。

- US7ASCII
- WE8DEC
- ISO-LATIN-1
- UTF-8

すべての各国語キャラクタ・セットを提供するために、Oracle JDBC ドライバには、2 つの追加ファイル、JDK 1.2.x 用の nls_charset12.zip と JDK 1.1.*x* 用の nls_charset11.zip が含まれています。OCI ドライバと Thin ドライバでは、Oracle オブジェクト型およびコレクション型の一部として使用されている CHAR データと VARCHAR データですべての Oracle キャラクタ・セットをサポートするために、これらのファイルが必要です。このサポートを受けるためには、CLASSPATH に適切な nls_charset*.zip ファイルを追加する必要があります。

nls_charset*.zip ファイルは多数のキャラクタ・セットをサポートする必要があるため、非常に大きいことに注意してください。領域を節約するために、nls_charset*.zip ファイルには必要なクラスのみを残すことができます。これを行うには、次の手順を実行します。

1. 適切な nls_charset*.zip ファイルを解凍します。
2. CLASSPATH に必要なキャラクタ・セットのクラスのみを追加します。
3. システムから不要なキャラクタ・セット・ファイルを削除します。

キャラクタ・セット拡張要素クラス・ファイルには、次の形式で名前が付けられます。

CharacterConverter<*OracleCharacterSetId*>.class

<*OracleCharacterSetId*> は、キャラクタ・セット名に対応する Oracle キャラクタ・セット ID の 16 進表現です。

注意： 前述の説明は、サーバー側内部ドライバを使用する場合には関係ありません。サーバー側内部ドライバは完全なグローバリゼーション・サポートを提供し、各国語キャラクタ・セット・クラスを必要としません。

Thin ドライバによる CHAR および VARCHAR2 データ・サイズ制限

データベース・キャラクタ・セットが、ASCII（US7ASCII）または ISO-LATIN-1（WE8ISO8859P1）以外の場合は、Thin ドライバは CHAR および VARCHAR2 バインド・パラメータに、通常のデータベース・サイズ制限よりも厳しいサイズ制限を課す必要があります。これは変換中にデータの展開を可能にするために必要です。

setXXX() メソッドがコールされたときに、Thin ドライバは CHAR または VARCHAR2 バインド・サイズをチェックします。データ・サイズがサイズ制限を超過している場合は、ドライバは setXXX() コールから SQL 例外（ORA-17070「データ・サイズがこの型の最大サイズを超えています。」）を発生させます。この制限は、グローバリゼーション・サポートの変換が行われ、データ長が増えたときに、データが破損しないようにするために必要です。この制限は、次のすべての場合に強制されます。

- Thin ドライバを使用している場合
- バインド（定義ではない）を使用している場合
- CHAR または VARCHAR2 データ型を使用している場合
- キャラクタ・セットが ASCII（US7ASCII）または ISO-Latin-1（WE8ISO8859P1）以外のデータベースに接続している場合

グローバリゼーション・サポート比の役割

前述したように、データベース・キャラクタ・セットが US7ASCII または WE8ISO8859P1 以外の場合、Thin ドライバは CHAR または VARCHAR2 バインドのために、Java UCS-2 キャラクタを UTF-8 エンコーディング・バイトに変換します。次に UTF-8 エンコーディング・バイトはデータベースに転送され、データベースは UTF-8 エンコーディング・バイトをデータベース・キャラクタ・セット・エンコーディングに変換します。

このキャラクタ・セット・エンコーディングの変換によって、サイズが大きくなることがあります。データベース・キャラクタ・セットのグローバリゼーション・サポート比は、UTF-8 からキャラクタ・セットへの変換について最大許容拡張率を示します。

```
NLS ratio = (maximum possible value of) [(size in database character set) / (size in UTF-8)]
```

サイズ制限計算式

表 18-1 は、CHAR および NCHAR データのデータベース・サイズ制限と、CHAR および NCHAR バインドの Thin ドライバのサイズ制限計算式を示します。データベース制限はバイト単位です。計算式は、バイト単位で UTF-8 エンコーディングの最大サイズを決定します。

表 18-1 Thin ドライバの最大 CHAR および NCHAR バインド・サイズ

Oracle バージョン	データ型	データベースが許容する最大サイズ（バイト）	Thin ドライバ最大バインド・サイズの計算式（UTF-8 バイト）
Oracle9*i*	NCHAR	2000	min(2000, 4000/NLS_ratio)
Oracle9*i*	NVARCHAR2	4000	4000/NLS_ratio
Oracle8 と Oracle8*i* 以上	CHAR	2000	min(2000, 4000/NLS_ratio)
Oracle8 と Oracle8*i* 以上	VARCHAR2	4000	4000/NLS_ratio
Oracle7	CHAR	255	255
Oracle7	VARCHAR2	2000	2000/NLS_ratio

計算式によって、UTF-8 からデータベース・キャラクタ・セットにデータを変換した後に、そのサイズがデータベースの最大サイズを超過することがなくなります。

サポートできる UCS-2 文字の数は、データ内の文字あたりのバイト数によって決定されます。すべての ASCII 文字は、UTF-8 エンコーディングでは、1 バイト長です。他のキャラクタ・タイプは、2 バイト長または 3 バイト長のこともあります。

共通キャラクタ・セットのグローバリゼーション・サポート比と計算されたサイズ制限

表 18-2 は、いくつかの共通サーバー・キャラクタ・セットのグローバリゼーション・サポート比と、適切な計算式でグローバリゼーション・サポート比を使用して決定される CHAR データと VARCHAR2 データの Thin ドライバ最大バインド・サイズをキャラクタ・セットごとに示します。

この表でも、UTF-8 エンコーディングの最大バインド・サイズはバイト単位です。

表 18-2 Oracle8、共通キャラクタ・セットのグローバリゼーション・サポート比およびサイズ制限

サーバー・キャラクタ・セット	グローバリゼーション・サポート比	Thin ドライバ最大 VARCHAR2 バインド・サイズ（UTF-8 バイト）	Thin ドライバ最大 CHAR バインド・サイズ（UTF-8 バイト）
WE8DEC	1	4000	2000
JA16SJIS	2	2000	2000
ISO 8859-1 から ISO 8859-10	3	1333	1333

JDBC クライアント側セキュリティ機能

この項では、Oracle Advanced Security オプションの機能に関して、Oracle JDBC OCI ドライバと Thin ドライバによるログイン認証、データ暗号化およびデータ整合性のサポートについて説明します。

以前は Advanced Networking Option（ANO）または Advanced Security Option（ASO）と呼ばれていた Oracle Advanced Security には、データ暗号化、データ整合性、サード・パーティ認証および認可をサポートする機能があります。Oracle JDBC は、これらの機能のほとんどをサポートしています。ただし、JDBC Thin ドライバは JDBC OCI ドライバとは別に考慮する必要があります。

注意： サーバー側内部ドライバの場合、ドライバを介したすべての通信が完全にサーバー内で行われるため、この説明はサーバー側内部ドライバには関係しません。

JDBC による Oracle Advanced Security のサポート

JDBC OCI ドライバと JDBC Thin ドライバの両方は、少なくとも Oracle Advanced Security の一部の機能をサポートしています。OCI ドライバの 1 つを使用している場合は、Thick クライアント設定の場合と同じ方法で関連するパラメータを設定できます。Thin ドライバは、JDBC クラス ZIP ファイルに含まれる Java クラスの集合によって、Advanced Security 機能をサポートし、Java プロパティ・オブジェクトによってセキュリティ・パラメータ設定をサポートします。

Oracle JDBC `classes111.zip` または `classes12.zip` ファイルには、Oracle Advanced Security の機能が組み込まれたクラスを含む JAR ファイルと、JDBC Thin ドライバとともに使用するために JDBC クラスと Advanced Security クラス間のインタフェースの機能を果たすクラスを含む JAR ファイルが入っています。

OCI ドライバによる Oracle Advanced Security のサポート

JDBC OCI ドライバの 1 つを使用している場合は、Oracle クライアントがインストールされている、Thick クライアント・マシンから実行していることが想定されるため、Oracle Advanced Security と組込みサード・パーティ機能のサポートは、ほとんど Oracle Thick クライアントの場合と同じです。Advanced Security 機能の使用は、クライアント・マシンでの `SQLNET.ORA` ファイルの関連する設定によって決定されます。詳細は、『Oracle Advanced Security 管理者ガイド』を参照してください。

重要： Java に関して、前述した説明の例外となるのは、SSL（Sun 社の標準 Secure Socket Layer プロトコル）です。Oracle JDBC OCI ドライバはアプリケーションでネイティブ・スレッド（システム固有なスレッド）を使用する場合のみ SSL をサポートします。通常はグリーン・スレッドがデフォルトになっているため、これには特別な注意が必要です。

Thin ドライバによる Oracle Advanced Security のサポート

Thin ドライバは、アプレットでダウンロードできるように設計されたため、Thin ドライバを使用する場所に、Oracle クライアントのインストールと、`SQLNET.ORA` ファイルがあるかどうかは明らかではありません。このため、Oracle Advanced Security のサポートのために、新しい 100% Java アプローチの設計が必要になります。

Oracle Advanced Security を実装する Java クラスは、JDBC `classes12.zip` ファイルまたは `classes111.zip` ファイルに含まれています。通常は `SQLNET.ORA` で暗号化と整合性のためのセキュリティ・パラメータを設定しますが、この場合は Java プロパティ・ファイルで設定します。

パラメータ設定の詳細は、18-13 ページの「Thin ドライバによる暗号化と整合性のサポート」を参照してください。

JDBC によるログイン認証のサポート

JDBC を介した基本ログイン認証では、Oracle Server への他のログイン方法と同様に、ユーザー名とパスワードが要求されます。3-4 ページの「データベースへの接続のオープン」で説明するように、Java プロパティ・オブジェクトによって、または直接 `getConnection()` メソッドをコールして、ユーザー名とパスワードを指定します。

これは、使用しているクライアント側 Oracle JDBC ドライバにかかわらず適用されますが、サーバー側内部ドライバを使用している場合には、特殊な直接接続を使用しており、ユーザー名またはパスワードを要求しないため、関係ありません。

Oracle JDBC Thin ドライバは、ユーザーを認証するために、Oracle O3LOGON 要求 / 応答プロトコルを実装します。

注意： RADIUS、Kerberos または SecurID などによって提供される Oracle Advanced Security がサポートするサード・パーティ認証機能は、Oracle JDBC Thin ドライバによってサポートされていません。Oracle JDBC OCI ドライバによるサポートは、すべての Thick クライアントの場合と同様です。『Oracle Advanced Security 管理者ガイド』を参照してください。

JDBC によるデータ暗号化と整合性のサポート

サーバーの関連部分の設定によっては、Java データベース・アプリケーションで、Oracle Advanced Security のデータ暗号化および整合化機能を使用できます。

Thick クライアント設定で OCI ドライバを使用している場合は、Oracle クライアントの場合と同様にパラメータを設定します。Thin ドライバを使用している場合は、Java プロパティ・ファイルでパラメータを設定します。

暗号化は、クライアント側暗号化レベル設定と、サーバー側暗号化レベル設定の組合せに基づいて、使用可能または使用禁止になります。

同様に、整合性はクライアント側整合性レベル設定と、サーバー側整合性レベル設定の組合せに基づいて、使用可能または使用禁止になります。

暗号化と整合性は、REJECTED、ACCEPTED、REQUESTED および REQUIRED の同じ設定レベルをサポートしています。表 18-3 には、この機能を使用可能または使用禁止にするために、クライアント側設定とサーバー側設定を組み合せる方法を示します。

表 18-3 暗号化または整合性のためのクライアント / サーバー間ネゴシエーション表

	クライアント Rejected	クライアント Accepted（デフォルト）	クライアント Requested	クライアント Required
サーバー Rejected	OFF	OFF	OFF	接続に失敗
サーバー Accepted（デフォルト）	OFF	OFF	ON	ON
サーバー Requested	OFF	ON	ON	ON
サーバー Required	接続に失敗	ON	ON	ON

この表は、たとえばクライアントが暗号化を要求し、サーバーが拒否した場合に、その暗号化が使用禁止になることを示しています。整合性の場合も同様です。もう 1 つの例として、クライアントが暗号化をアクセプトし、サーバーが要求した場合は、その暗号化は使用可能になります。整合性の場合も同様です。

一般設定の詳細は、『Oracle Advanced Security 管理者ガイド』で説明します。JDBC アプリケーションでの設定方法については、次の項目で説明します。

注意： 次の項目で示すように、整合性パラメータ名にはまだ CHECKSUM という用語が使用されていますが、この用語はパラメータ以外では使用されません。事実上、「checksum」と「integrity」はシノニムです。

OCI ドライバによるデータ暗号化と整合性のサポート

Oracle JDBC OCI ドライバの 1 つを使用している場合は、Oracle クライアントのインストールを伴う Thick クライアント設定が想定されるため、クライアント・マシンの SQLNET.ORA ファイルの設定によって、すべての Oracle クライアントの場合と同様に、データの暗号化または整合性を使用可能または使用禁止にし、関連するパラメータを設定できます。

表 18-4 に、クライアント・パラメータについてまとめます。

表 18-4 暗号化と整合性のための OCI ドライバ・クライアント・パラメータ

パラメータの説明	パラメータ名	可能な設定
クライアント暗号化レベル	SQLNET.ENCRYPTION_CLIENT	REJECTED ACCEPTED REQUESTED REQUIRED
クライアント暗号化選択リスト	SQLNET.ENCRYPTION_TYPES_CLIENT	RC4_40 RC4_56 DES DES40 (「注意」を参照)
クライアント整合性レベル	SQLNET.CRYPTO_CHECKSUM_CLIENT	REJECTED ACCEPTED REQUESTED REQUIRED
クライアント整合性選択リスト	SQLNET.CRYPTO_CHECKSUM_TYPES_CLIENT	MD5

注意： RC4_128 の設定は、Oracle Advanced Security の国内版のみで使用できます。

前述の設定と、サーバーでの対応する設定の詳細は、『Oracle Advanced Security 管理者ガイド』の付録 A で説明します。

Thin ドライバによる暗号化と整合性のサポート

Thin ドライバによるデータ暗号化および整合性パラメータ設定のサポートは、前の項で説明した Thick クライアント・サポートに対応しています。対応するパラメータは、oracle.net パッケージの下にあり、データベース接続をオープンするときに使用する Java プロパティ・オブジェクトによって設定できます。

表 18-4 のパラメータ名の「SQLNET」を「oracle.net」に置換すると、Thin ドライバがサポートするパラメータ名が取得できます（ただし、Java ではパラメータ名はすべて小文字であることに注意してください）。

表 18-5 には、Thin ドライバのパラメータ情報をリストします。Java でこれらのパラメータを設定する方法の例については、次の項を参照してください。

表 18-5 暗号化と整合性のための Thin ドライバ・クライアント・パラメータ

パラメータ名	パラメータ・タイプ	パラメータ・クラス	可能な設定
oracle.net.encryption_client	string	静的	REJECTED ACCEPTED REQUESTED REQUIRED
oracle.net.encryption_types_client	string	静的	RC4_40 RC4_56 DES40C DES56C
oracle.net.crypto_checksum_client	string	静的	REJECTED ACCEPTED REQUESTED REQUIRED
oracle.net.crypto_checksum_types_client	string	静的	MD5

注意：

- Thin ドライバでの Oracle Advanced Security のサポートは、JDBC クラス ZIP ファイルに直接組み込まれているため、1 つのバージョンのみが存在し、国内版と輸出版に分かれてはいません。輸出版には適するパラメータ設定のみを使用できます。
- DES40C と DES56C の C は、CBC（Cipher Block Chaining）モードを示しています。

Java での暗号化および整合性パラメータの設定

Oracle JDBC Thin ドライバがサポートするデータ暗号化および整合性パラメータを設定するには、Java プロパティ・オブジェクト（java.util.Properties）を使用します。

次の例では、Java プロパティ・オブジェクトをインスタンス化し、それを使用して表 18-5 の各パラメータを設定し、そのプロパティ・オブジェクトを使用してデータベース接続をオープンします。

```
...
Properties prop = new Properties();
prop.put("oracle.net.encryption_client", "REQUIRED");
prop.put("oracle.net.encryption_types_client", "( DES40 )");
prop.put("oracle.net.crypto_checksum_client", "REQUESTED");
prop.put("oracle.net.crypto_checksum_types_client", "( MD5 )");
Connection conn = DriverManager.getConnection
   ("jdbc:oracle:thin:@localhost:1521:main", prop);
...
```

encryption_types_client 設定と crypto_checksum_types_client 設定のパラメータ値を囲むカッコ内では、値のリストを使用できます。現在、Thin ドライバではどちらの場合にも 1 つの値のみを指定できます。ただし、将来複数の値がサポートされた場合に、リストを指定するとサーバーとクライアント間でネゴシエーションが発生して、実際に使用する値が決定されます。

完全な例 次に、データベースに接続して問合せを実行する前に、データ暗号化および整合性パラメータを設定するクラスの完全な例を示します。

この例で、文字列「REQUIRED」は、AnoServices クラスと Service クラスの機能によって動的に取り出されます。この方法で文字列を取り出すことも、前の例のようにハードコードすることもできます。

```
import java.sql.*;
import java.sql.*;
import java.io.*;
import java.util.*;
import oracle.net.ns.*;
import oracle.net.ano.*;

class Employee
{
  public static void main (String args [])
    throws Exception
  {

    // Register the Oracle JDBC driver
    System.out.println("Registring the driver...");
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
```

```
    Properties props = new Properties();

    try {
      FileInputStream defaultStream = new FileInputStream(args[0]);
      props.load(defaultStream);


      int level = AnoServices.REQUIRED;
      props.put("oracle.net.encryption_client", Service.getLevelString(level));
      props.put("oracle.net.encryption_types_client", "( DES40 )");
      props.put("oracle.net.crypto_checksum_client",
                Service.getLevelString(level));
      props.put("oracle.net.crypto_checksum_types_client", "( MD5 )");
    } catch (Exception e) { e.printStackTrace(); }


    // You can put a database name after the @ sign in the connection URL.
    Connection conn = DriverManager.getConnection
               ("jdbc:oracle:thin:@dlsun608.us.oracle.com:1521:main", props);

    // Create a Statement
    Statement stmt = conn.createStatement ();

    // Select the ENAME column from the EMP table
    ResultSet rset = stmt.executeQuery ("select ENAME from EMP");

    // Iterate through the result and print the employee names
    while (rset.next ())
       System.out.println (rset.getString (1));

    conn.close();
  }

}
```

アプレット内の JDBC

この項では、Oracle JDBC アプレットの基本操作をいくつか説明します。クライアントで Oracle インストレーションが要求されないように、JDBC Thin ドライバを使用する必要があります。Thin ドライバは、TCP/IP プロトコルでデータベースに接続します。

Thin ドライバを使用する必要があることと、アプレット接続とセキュリティの問題に注意することを除いて、JDBC アプレットと JDBC アプリケーションのコーディング間に、ほとんど違いはありません。4-5 ページの「JDK 1.1.x から JDK 1.2.x への移行」で説明する一般的な JDK 1.1.x から 1.2.x への移行の問題を除いて、JDK 1.2.x ブラウザまたは JDK 1.1.x ブラウザのコーディングにも違いはありません。

この項では、アプレットをデータベースに接続させるために必要な操作について説明します。これには Web サーバーとは別のホストで動作しているデータベースに接続する場合に、Oracle8 Connection Manager または署名付きアプレットを使用する方法が含まれます。また、ファイアウォールを通過してアプレットをデータベースに接続する方法についても説明します。最後に、アプレットをパッケージ化および配置する方法について説明します。

次の項目があります。

- アプレットを介したデータベースへの接続
- Web サーバーとは異なるホスト上のデータベースへの接続
- ファイアウォールとアプレットの使用方法
- アプレットのパッケージ化
- HTML ページでのアプレットの指定

データベース接続の一般情報は、3-4 ページの「データベースへの接続のオープン」を参照してください。

サンプル・アプレットについては、20-116 ページの「サンプル・アプレット」を参照してください。

注意： Oracle JDBC では、JDK 1.0.x バージョンをサポートしていません。これは JDK1.0.x バージョンを組み込んでいるブラウザで実行されるアプレットにも当てはまります。JDK1.1.x 以降の環境を持つブラウザにアップグレードする必要があります。

アプレットを介したデータベースへの接続

JDBC ドライバを使用したアプレットの最も一般的な作業は、データベースへの接続と問合せです。アプレットのセキュリティ上の制限のため、特定の処理を行わない限り、アプレットはダウンロード元のホスト（Web サーバーが実行されているホスト）への TCP/IP ソケットしかオープンできません。つまり、特定の処理を行わない場合、アプレットが接続できるのは、Web サーバーと同じホスト上で動作するデータベースのみです。

データベースと Web サーバーが同じホストで動作している場合は、問題はなく処理は必要ありません。アプリケーションから接続する場合と同様に、データベースに接続できます。

アプリケーションから接続する場合と同様に、接続情報をドライバに指定する方法は 2 つあります。`host:port:sid` の形式または TNS キーワード値構文の形式で指定できます。

たとえば、接続するデータベースがポート `1521`、ホスト `prodHost`、SID `ORCL` で、ユーザー名を `scott`、パスワードを `tiger` として接続する場合は、次のいずれかの接続文字列を使用します。

`host:port:sid` 構文を使用する場合は、次のようになります。

```
String connString="jdbc:oracle:thin:@prodHost:1521:ORCL";
conn = DriverManager.getConnection(connString, "scott", "tiger");
```

TNS キーワード値構文を使用する場合は、次のようになります。

```
String connString = "jdbc:oracle:thin:@(description=(address_list=
   (address=(protocol=tcp)(port=1521)(host=prodHost)))
   (connect_data=(sid=ORCL)))";
conn = DriverManager.getConnection(connString, "scott", "tiger");
```

TNS キーワード値ペアを使用して JDBC Thin ドライバに接続情報を指定する場合は、プロトコルを TCP として宣言する必要があります。

ただし、Web サーバーと Oracle データベース・サーバーは、どちらも多くのリソースを必要とし、両方のサーバーが同じマシン上で実行されていることはほとんどありません。通常、アプレットは Web サーバーが実行されているホストとは別のホスト上のデータベースに接続します。セキュリティ上の制限に対処する方法は 2 つあります。

- Oracle8 Connection Manager を使用してデータベースに接続できます。

または

- 署名付きアプレットを使用して、データベースに直接接続できます。

次の項の「Web サーバーとは異なるホスト上のデータベースへの接続」で、これらのオプションについて説明します。

Web サーバーとは異なるホスト上のデータベースへの接続

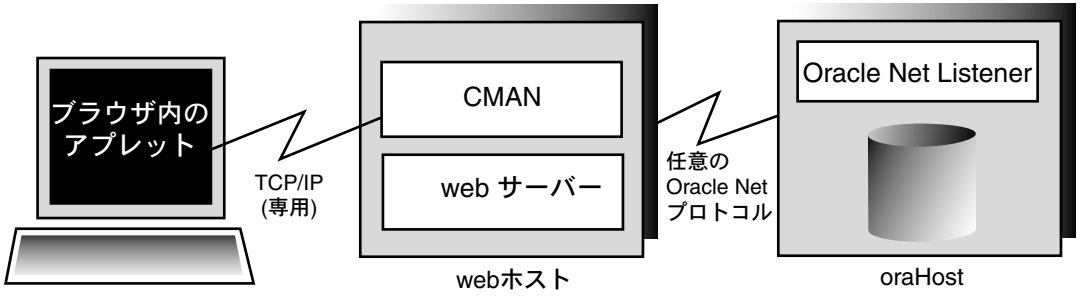
Web サーバーが動作しているホスト以外のホスト上のデータベースに接続する場合は、アプレットのセキュリティの制限に対処する必要があります。これは、Oracle8 Connection Manager または署名付きアプレットを使用することで可能です。

Oracle8 Connection Manager の使用

Oracle8 Connection Manager は、Oracle Net パケットを受信して別のサーバーに再転送できる、軽量でスケーラブルなプログラムです。Oracle Net を実行するクライアントからは、Connection Manager はちょうどデータベース・サーバーのように見えます。JDBC Thin ドライバを使用するアプレットは、Web サーバー・ホスト上で実行中の Connection Manager に接続し、Connection Manager を使用して、Oracle Net パケットを異なるホスト上で動作する Oracle Server にリダイレクトできます。

図 18-1 に、アプレット、Oracle8 Connection Manager およびデータベースの関係を示します。

図 18-1 アプレット、Connection Manager およびデータベースの関係



Oracle8 Connection Manager を使用するには、次のような 2 つの手順を実行する必要があります。

- Connection Manager のインストールおよび実行。
- Connection Manager をターゲットにした接続文字列の記述。

複数の Connection Manager を使用して接続する方法についても説明します。

Oracle8 Connection Manager のインストールおよび実行 Oracle8 配布媒体に格納されている Connection Manager を Web サーバー・ホストにインストールする必要があります。インス

トールに関する指示については、『Oracle9*i* Net Services 管理者ガイド』を参照してください。

Web サーバー・ホストでは、[ORACLE_HOME]/NET8/ADMIN ディレクトリに CMAN.ORA ファイルを作成します。CMAN.ORA ファイル内には、ファイアウォールおよび接続プーリングのサポートなどのオプションも宣言できます。

次に、非常に単純な CMAN.ORA ファイルの例を示します。<*web-server-host*> を使用する Web サーバー・ホストの名前に置き換えてください。ファイルの 4 行目は、Connection Manager がポート 1610 をリスニングしていることを示しています。このポート番号を JDBC の接続文字列に使う必要があります。

```
cman = (ADDRESS_LIST =
        (ADDRESS = (PROTOCOL=TCP)
        (HOST=<web-server-host>)
        (PORT=1610)))

cman_profile = (parameter_list =
        (MAXIMUM_RELAYS=512)
        (LOG_LEVEL=1)
    (TRACING=YES)
        (RELAY_STATISTICS=YES)
        (SHOW_TNS_INFO=YES)
        (USE_ASYNC_CALL=YES)
        (AUTHENTICATION_LEVEL=0)
        )
```

JDBC Thin ドライバ内の Java Oracle Net バージョンは、認証サービスをサポートしていません。つまり、CMAN.ORA ファイル内の AUTHENTICATION_LEVEL 構成パラメータを 0（ゼロ）に設定する必要があります。

ファイルの作成後、オペレーティング・システムのプロンプトで次のコマンドを使用して Oracle8 Connection Manager を起動します。

```
cmctl start
```

アプレットを使用するには、この時点でアプレット用の接続文字列を記述する必要があります。

Oracle8 Connection Manager をターゲットにした接続文字列の記述 アプレットが Connection Manager に接続し、Connection Manager がデータベースに接続するようにするための接続文字列を記述する方法について説明します。接続文字列には、Connection Manager が実行されている Web サーバー・ホストのプロトコル、ポートおよびホスト名の後に、データベースが実行されているホストのプロトコル、ポートおよびホスト名を指定します。

次の例では、図 18-1 で示した構成について説明します。Connection Manager を実行している Web サーバーは、ホスト webHost にあり、ポート 1610 をリスニングしています。接続対象のデータベースは、ポート 1521 をリスニングし、SID が ORCL であるホスト oraHost 上にあります。接続文字列を TNS キーワード値形式で次のように記述します。

```
Connection conn =
   DriverManager.getConnection ("jdbc:oracle:thin:" +
   "@(description=(address_list=" +
   "(address=(protocol=tcp)(host=webHost)(port=1610))" +
   "(address=(protocol=tcp)(host=oraHost)(port=1521)))" +
   "(source_route=yes)" +
   "(connect_data=(sid=orcl)))", "scott", "tiger");
```

address_list エントリの最初の要素は、Connection Manager への接続を表します。2 番目の要素は、接続するデータベースを表します。アドレスをリストする順序は重要です。

同じ接続文字列を、次の形式でも記述できます。

```
String connString =
   "jdbc:oracle:thin:@(description=(address_list=
   (address=(protocol=tcp)(port=1610)(host=webHost))
   (address=(protocol=tcp)(port=1521)(host=oraHost)))
   (connect_data=(sid=orcl))
   (source_route=yes))";
Connection conn = DriverManager.getConnection(connString, "scott", "tiger");
```

アプレットで前述のような接続文字列を使用した場合、アプレットはホスト oraHost 上のデータベースに直接接続したかのように動作します。

接続文字列内に指定するパラメータの詳細は、『Oracle9*i* Net Services 管理者ガイド』を参照してください。

複数の Connection Manager 経由の接続 アプレットは、複数の Connection Manager を経由してからでもターゲット・データベースに接続できます（たとえば、Connection Manager が代理連鎖を形成している場合など）。これを行うには、Connection Manager のアドレスをアクセスする順番にアドレス・リストに追加します。データベース・リスナーは、このリストの最後のアドレスにする必要があります。source_route アドレスの詳細は、『Oracle9*i* Net Services 管理者ガイド』を参照してください。

署名付きアプレットの使用方法

JDK 1.2.x ベース・ブラウザまたは JDK 1.1.x ベース・ブラウザのどちらかで、アプレットはソケット接続権限を要求し、Web サーバー・ホストとは異なるホストで動作しているデータベースに接続できます。Netscape 4.0 では、アプレットに署名（署名付きアプレットを記述）することによって、これを実行できます。次の手順を実行する必要があります。

1. アプレットに署名します。アプレットに署名する手順の詳細は、次のサイトにある Sun 社の「Signed Applet Example」を参照してください。

   ```
   http://java.sun.com/security/signExample12/index.html
   ```

2. ソケットをオープンする前に適切なパーミッションを要求するアプレット・コードを含めます。

   Netscape を使用している場合は、コードに次のような文を含めます。

   ```
   netscape.security.PrivilegeManager.enablePrivilege("UniversalConnect");
   connection = DriverManager.getConnection
                ("jdbc:oracle:thin:scott/tiger@dlsun511:1721:orcl");
   ```

3. オブジェクト署名証明書を取得する必要があります。次のサイトにある Netscape 社の「Object-Signing Resources」を参照してください。

   ```
   http://developer.netscape.com/software/signedobj/index.html
   ```

   このサイトには、証明書の取得とインストールについての情報があります。

パーミッションを要求するアプレット・コードの記述の詳細は、次のサイトにある Netscape 社の「Introduction to Capabilities Classes」を参照してください。

```
http://developer.netscape.com/docs/manuals/signedobj/capabilities/contents.htm
```

JDK 1.2.x および 1.1.x での署名付きアプレットの例を含む Java Security API については、次の Sun 社のサイトを参照してください。

```
http://java.sun.com/security
```

ファイアウォールとアプレットの使用方法

通常の状況下では、JDBC Thin ドライバを使用したアプレットは、ファイアウォールを通過してデータベースにアクセスできません。一般的に、ファイアウォールは、権限を与えられていないクライアントによるサーバーへのアクセスを防ぐことを目的としています。データベースに接続しようとするアプレットの場合、ファイアウォールはデータベースに対する TCP/IP ソケットのオープンを防止します。

ファイアウォールでは規則を使用します。規則のリストには、接続が可能なクライアントとそうでないクライアントが定義されています。ファイアウォールはこの規則とクライアントのホスト名を比較し、この比較に基づいてクライアントにアクセスを許可または禁止します。ホスト名の検索が失敗した場合は、ファイアウォールはもう一度検索を試みます。今度は、クライアントの IP アドレスを抽出して、それを規則と比較します。ファイアウォールはこのように設計されているので、ユーザーはホスト名と IP アドレスを含んだ規則を指定できます。

ファイアウォールに関する問題は、Oracle Net に準拠したファイアウォールおよびそのファイアウォール構成に応じた接続文字列を使用することによって解決できます。Oracle Net に準拠したファイアウォールは、多数の主要なベンダーから入手できます。このマニュアルでは、ファイアウォールに関する詳細は取り上げません。

署名付きでないアプレットは、アプレットのダウンロード元ホストにのみアクセスできます。この場合、そのホストには、Oracle Net に準拠したファイアウォールがインストールされている必要があります。これに対し、署名付きアプレットは任意のホストに接続できます。この場合、ターゲット・ホスト上のファイアウォールがアクセスを制御します。

ファイアウォールを通過して接続するには、次の項で説明する 2 つの処理を行う必要があります。

- JDBC Thin ドライバを使用するアプレット用のファイアウォールの構成
- ファイアウォールを通過する接続のための接続文字列の記述

JDBC Thin ドライバを使用するアプレット用のファイアウォールの構成

この項の説明では、Oracle Net に準拠したファイアウォールを実行していることを前提としています。

セキュリティ上の制限のため、Java アプレットはローカル・システムにはアクセスできません（ローカルにホスト名または環境変数を取得できません）。この結果、JDBC Thin ドライバはドライバが動作するホストのホスト名にアクセスできません。ファイアウォールにはホスト名が提供されません。JDBC Thin クライアントからの要求がファイアウォールを通過することを許可するには、ファイアウォールの規則のリストに対して次の 2 つの操作が必要です。

- JDBC アプレットが実行されているホストの IP アドレス（ホスト名ではない）を追加します。
- ファイアウォールの規則内にホスト名「__jdbc__」が記述されていないことを確認します。このホスト名は、IP アドレスの検索を強制的に行うために、ドライバ内に模擬ホ

スト名としてハードコードされています。このホスト名を規則のリスト内に入力した場合は、Oracle の JDBC Thin ドライバを使用するすべてのアプレットがファイアウォールを通過できるようになります。

Thin ドライバのホスト名を含めないことによって、ファイアウォールは必ず IP アドレスを検索し、アクセスの決定をホスト名でなく IP アドレスに基づいて行うようになります。

ファイアウォールを通過する接続のための接続文字列の記述

ファイアウォールを通過して接続できる接続文字列を記述するには、接続するファイアウォール・ホスト名およびデータベース・ホスト名を指定する必要があります。

たとえば、ポート 1521 をリスニングし、SID が ORCL であるホスト oraHost 上にあるデータベースに接続し、ポート 1610 をリスニングしているホスト fireWallHost 上にあるファイアウォールを通過する場合は、次の接続文字列を使用します。

```
Connection conn =
      DriverManager.getConnection ("jdbc:oracle:thin:" +
      "@(description=(address_list=" +
      (address=(protocol=tcp)(host=<firewall-host>)(port=1610))" +
      "(address=(protocol=tcp)(host=oraHost)(port=1521)))" +
      "(source_route=yes)" +
      "(connect_data=(sid=orcl)))", "scott", "tiger");
```

注意： ファイアウォールを通過して接続するために、host:port:sid 構文内に接続文字列を指定できません。たとえば、次のように指定した接続文字列は動作しません。

```
String connString =
      "jdbc:oracle:thin:@ixta.us.oracle.com:1521:orcl";
conn = DriverManager.getConnection (connString, "scott",
      "tiger");
```

---

address_list の最初の要素は、ファイアウォールへの接続を表します。2 番目の要素は、接続するデータベースを表します。アドレスを指定する順序が重要なので注意してください。

前述の接続文字列を、次の形式でも記述できます。

```
String connString =
      "jdbc:oracle:thin:@(description=(address_list=
      (address=(protocol=tcp)(port=1600)(host=fireWallHost))
      (address=(protocol=tcp)(port=1521)(host=oraHost)))
      (connect_data=(sid=orcl))
      (source_route=yes))";
Connection conn = DriverManager.getConnection(connString, "scott", "tiger");
```

アプレットが前述のような接続文字列を使用する場合は、アプレットがホスト oraHost 上のデータベースに接続されているかのように動作します。

注意： 前述の例に示したすべてのパラメータが必要です。address_list では、ファイアウォール・アドレスは、データベース・サーバー・アドレスよりも先に指定する必要があります。

前述の例で使用したパラメータの詳細は、『Oracle9*i* Net Services 管理者ガイド』を参照してください。ファイアウォールの構成方法の詳細は、ファイアウォールのドキュメントを参照するか、またはファイアウォールのベンダーにお問い合せください。

アプレットのパッケージ化

アプレットのコーディング後は、パッケージ化してユーザーが利用できるようにする必要があります。アプレットをパッケージ化するには、アプレット・クラス・ファイルと JDBC ドライバ・クラス・ファイルが必要です（これらのファイルは JDK 1.2.x バージョンを組み込んでいるブラウザをターゲットにする場合は classes12.zip、JDK 1.1.x バージョンを組み込んでいるブラウザの場合は classes111.zip に含まれています）。

次の手順を実行します。

1. JDBC ドライバのクラス・ファイル classes12.zip（または classes111.zip）を空のディレクトリに移動します。

   アプレットが US7ASCII または WE8ISO8859P1 以外のキャラクタ・セットのデータベースに接続する場合は、nls_charset12.zip ファイルまたは nls_charset11.zip ファイルも同じディレクトリに移動します。

2. JDBC ドライバ・クラス ZIP ファイル（および必要に応じて、各国語キャラクタ・セット ZIP ファイル）を解凍します。

3. アプレットのクラス・ファイルおよびアプレットで必要になる可能性のあるファイルをすべてこのディレクトリに追加します。

4. アプレットのクラスおよびドライバのクラスをまとめて 1 つの ZIP ファイルまたは JAR ファイルにします。この 1 つの圧縮ファイルには、次のものを入れてください。

   - classes12.zip または classes111.zip からのクラス・ファイル（およびアプレットでグローバリゼーション・サポートが必要な場合は、nls_charset12.zip または nls_charset11.zip から要求されたクラス・ファイル）
   - アプレットのクラス

   アプレットで DatabaseMetaData エントリ・ポイントを使用している場合は、oracle/jdbc/driver/OracleDatabaseMetaData.class ファイルを含めます。このファイルは非常に大きいため、パフォーマンスに悪影響を及ぼす可能性があるので

注意してください。DatabaseMetaData メソッドを使用しない場合は、このファイルを省略します。

5. ZIP ファイルまたは JAR ファイルが圧縮されていないことを確認します。

これでユーザーがアプレットを使用できるようになります。アプレットを利用するには、アプレットを実行する HTML ページに APPLET タグを追加する方法があります。たとえば、次のようになります。

```
<APPLET WIDTH=500 HEIGHT=200 CODE=JdbcApplet ARCHIVE=JdbcApplet.zip
      CODEBASE=Applet_Samples
</APPLET>
```

APPLET、CODE、ARCHIVE、CODEBASE、WIDTH および HEIGHT パラメータの説明は、次の項を参照してください。

HTML ページでのアプレットの指定

APPLET タグは、HTML ページのコンテキスト内で実行されるアプレットを指定します。APPLET タグでは、次のパラメータを使用できます。APPLET タグにはアプレットの名前と位置およびアプレット表示領域の高さと幅を指定するためのパラメータ CODE、ARCHIVE、CODEBASE、WIDTH および HEIGHT があります。次の項で、これらのパラメータについて説明します。

CODE、HEIGHT および WIDTH

アプレットを実行する HTML ページには、アプレット表示領域のサイズを指定する初期の幅および高さとともに、APPLET タグが必要です。ピクセル単位のサイズを指定するには、HEIGHT および WIDTH パラメータを使用します。サイズには、アプレットがオープンするウィンドウまたはダイアログのサイズは含めません。

APPLET タグでは、アプレットのコンパイル済み Applet サブクラスを含むファイルの名前も指定する必要があります。このファイル名は CODE パラメータで指定します。パスはすべてアプレットのベース URL の相対パスにする必要があります。絶対パスは使用できません。

次の例では、JdbcApplet.class は Applet のコンパイルされたアプレット・サブクラスの名前です。

```
<APPLET CODE="JdbcApplet" WIDTH=500 HEIGHT=200>
</APPLET>
```

この形式の CODE タグを使用する場合は、アプレットのためのクラスおよび JDBC Thin ドライバのためのクラスがこの HTML ページと同じディレクトリ内にある必要があります。

CODE の指定には、ファイル拡張子 .class は含めません。

CODEBASE

CODEBASE パラメータは省略可能で、アプレットのベース URL（アプレットのコードがあるディレクトリの名前）を指定します。このパラメータが指定されない場合は、ドキュメントの URL が使用されます。つまり、アプレットおよび JDBC Thin ドライバのためのクラスが、HTML ページと同じディレクトリ内にある必要があります。たとえば、カレント・ディレクトリが my_Dir であるとします。

```
<APPLET WIDTH=500 HEIGHT=200 CODE=JdbcApplet CODEBASE="."
</APPLET>
```

エントリ CODEBASE="." は、このアプレットがカレント・ディレクトリ（my_Dir）内にあることを示します。codebase の値が次のように Applet_Samples に設定されたとします。

```
CODEBASE="Applet_Samples"
```

これは、アプレットが my_Dir/Applet_Samples ディレクトリ内にあることを示します。

ARCHIVE

ARCHIVE パラメータは省略可能で、アプレットのクラスおよびアプレットが必要とするリソースが格納されているアーカイブ・ファイル（.zip または .jar ファイル）がある場合に、その名前を指定します。.zip ファイルまたは .jar ファイルの使用をお薦めします。これらのファイルを使用すると、サーバーとの余計なラウンドトリップを大幅に省くことができます。

.zip（または .jar）ファイルはあらかじめロードされます。リストに複数のアーカイブを指定する場合は、カンマで区切ります。次の例では、クラス・ファイルはアーカイブ・ファイル JdbcApplet.zip に格納されています。

```
<APPLET CODE="JdbcApplet" ARCHIVE="JdbcApplet.zip" WIDTH=500 HEIGHT=200>
</APPLET>
```

注意： バージョン 3.0 のブラウザでは、ARCHIVE パラメータはサポートされていません。

サーバー上の JDBC: サーバー側内部ドライバ

この項の項目は次のとおりです。

ターゲット・データベース内で実行される Java プログラム、Enterprise JavaBean（EJB）または Java ストアド・プロシージャは、サーバー側内部ドライバを使用してローカル SQL エンジンにアクセスする必要があります。

このドライバは Oracle データベースと Java Virtual Machine (JVM) に結び付けられています。このドライバは、データベースと同じプロセスの一部として動作します。また、デフォルトのセッション（JVM が起動されたのと同じセッション）内で動作します。

サーバー側内部ドライバはデータベース・サーバー内で動作するよう最適化されており、このドライバを使用するとローカル・データベース上の SQL データおよび PL/SQL サブプログラムに直接アクセスできます。JVM 全体は、データベースおよび SQL エンジンと同じアドレス空間内で動作します。SQL エンジンへのアクセスはファンクション・コールであり、ネットワークは使用されません。これにより、JDBC プログラムのパフォーマンスが向上し、SQL エンジンへのアクセスにリモート Oracle Net コールを実行するよりも高速になります。

サーバー側内部ドライバは、クライアント側ドライバと同じ機能、API および Oracle 拡張機能をサポートします。これにより、アプリケーションのパーティション化が非常に簡単になります。たとえば、データ集中処理型の Java アプリケーションがある場合、アプリケーション固有のコールを修正しなくても、パフォーマンスを向上させるために簡単にデータベース・サーバーに移動できます。

Oracle Java プラットフォームのサーバー側構成または機能の詳細は、『Oracle9*i* Java Developer's Guide』を参照してください。

サーバー側内部ドライバを使用したデータベースへの接続

前の項で説明したように、サーバー側内部ドライバはデフォルトのセッション内で動作します。つまり、すでに接続された状態になっています。デフォルト接続にアクセスするために使用できる 2 つのメソッドがあります。

- URL 文字列として jdbc:oracle:kprb または jdbc:default:connection を指定して、静的な DriverManager.getConnection() メソッドを使用します。
- OracleDriver クラスの Oracle 固有 defaultConnection() メソッドを使用します。

通常は defaultConnection() の使用をお薦めします。

注意： サーバー側内部ドライバと接続するために OracleDriver クラスを登録する必要はなくなりましたが、登録しても問題は生じません。これは接続のために getConnection() または defaultConnection() のどちらを使用している場合にも当てはまります。

OracleDriver クラスの defaultConnection() メソッドによる接続

oracle.jdbc.OracleDriver クラスの defaultConnection() メソッドは、Oracle 拡張機能で、常に同じ接続オブジェクトを戻します。発生した接続オブジェクトを別の変数名に割り当てて、このメソッドを複数回も呼び出したとしても、1 つの接続オブジェクトのみが再利用されます。

defaultConnection() コールに接続文字列を含める必要はありません。たとえば、次のようになります。

```
import java.sql.*;
import oracle.jdbc.*;

class JDBCConnection
{
  public static Connection connect() throws SQLException
  {
    Connection conn = null;
    try {
      // connect with the server-side internal driver
         OracleDriver ora = new OracleDriver();
         conn = ora.defaultConnection();
      }

    } catch (SQLException e) {...}
    return conn;
  }
}
```

この例には conn.close() コールがないことに注意してください。JDBC コードがターゲット・サーバー内で実行されている場合、接続は暗黙的なデータ・チャネルで、クライアントからの場合のように明示的な接続インスタンスではありません。通常はこれをクローズしないでください。

close() メソッドをコールする場合は、次の点に注意してください。

- 実際に、同じ接続オブジェクトを参照する defaultConnection() メソッドによって取得されたすべての接続インスタンスは、必要に応じて状態とリソースのクリーン・アップによってクローズされ、それ以降使用できなくなります。後に defaultConnection() を実行すると、新しい接続オブジェクトが生成されます。
- 接続オブジェクトをクローズしても、データベースへの暗黙的な接続はクローズされません。

DriverManager.getConnection() メソッドによる接続

ターゲット・サーバー内で実行中のコードから内部サーバー接続に接続するには、次のいずれかの接続文字列とともに、静的な DriverManager.getConnection() メソッドを使用できます。

```
DriverManager.getConnection("jdbc:oracle:kprb:");
```

または

```
DriverManager.getConnection("jdbc:default:connection:");
```

URL 文字列内で指定したユーザー名またはパスワードは、サーバー・デフォルト接続への接続では無視されます。

DriverManager.getConnection() メソッドをコールする度に、このメソッドは新しい Java Connection オブジェクトを戻します。このメソッドは、新しい物理接続を作成しませんが（1 つの暗黙的接続のみを使用します）、新しいオブジェクトを戻すことに注意してください。

オブジェクト・マップ（型マップ）の操作をしている場合は、DriverManager.getConnection() をコールする度にこのメソッドが新しい接続オブジェクトを戻すということに重要な意味があります。型マップは、特定の Connection オブジェクトおよびそのオブジェクトの一部である状態に対応付けられます。プログラムの一部として複数の型マップを使用する場合は、getConnection() をコールして、各型マップに対して新しい Connection オブジェクトを作成できます。

サーバー側内部ドライバの例外処理拡張要素

サーバー側内部ドライバは、(3-33 ページの「SQL 例外の処理」で説明するように) `getMessage()`、`getErrorCode()` および `getSQLState()` などの標準例外処理機能に加えて、`oracle.jdbc.driver.OracleSQLException` クラスを通じて拡張機能も提供します。このクラスは、標準 `java.sql.SQLException` クラスのサブクラスで、クライアント側 JDBC ドライバまたはサーバー側 Thin ドライバからは使用できません。

サーバーでエラー状態が発生した場合は、内部エラー・スタックに一連の関連するエラーが格納されることがあります。JDBC サーバー側内部ドライバは、スタックからエラーを取り出して、`OracleSQLException` オブジェクトの連鎖に格納します。

例外を処理するために次のメソッドを使用できます。

- `SQLException getNextException()` (標準メソッド)

  このメソッドは、連鎖内の次の例外 (次の例外がない場合は、`null`) を戻します。受け取った最初の例外から初めて、連鎖全体を処理できます。

- `int getNumParameters()` (Oracle 拡張機能)

  サーバーからのエラーには、通常エラー・メッセージの一部であるパラメータまたは変数が含まれます。これらは発生したエラーの種類、試行されていた操作の種類、無効な値または影響を受けた値を示すことがあります。

  このメソッドは、このエラーに含まれるいくつかのパラメータを戻します。

- `Object[] getParameters()` (Oracle 拡張機能)

  このメソッドは、このエラーに含まれるパラメータを格納した Java `Object[]` 配列を戻します。

例 : 次に、サーバー側エラー処理の例を示します。

```
try
{
   // should get "ORA-942: table or view does not exist"
   stmt.execute("drop table no_such_table");
}
catch (OracleSQLException e)
{
   System.out.println(e.getMessage());
   // prints "ORA-942: table or view does not exist"

   System.out.println(e.getNumParameters());
   // prints "1"

   Object[] params = e.getParameters();
   System.out.println(params[0]);
   // prints "NO_SUCH_TABLE"
}
```

サーバー側内部ドライバのセッション・コンテキストとトランザクション・コンテキスト

サーバー側ドライバは、デフォルト・セッションおよびデフォルト・トランザクションのコンテキストで動作します。デフォルト・セッションは、JVM が起動されたセッションです。サーバー上では、事実上データベースにすでに接続されています。これは、デフォルト・セッションがないクライアント側とは異なります。クライアント側では、明示的にデータベースに接続する必要があります。

サーバーでは、自動コミット・モードは無効になっています。接続オブジェクトで適切なメソッドを使用して、明示的にトランザクションの COMMIT および ROLLBACK 操作を管理する必要があります。

```
conn.commit();
```

または

```
conn.rollback();
```

サーバー上での JDBC のテスト

クライアント上で実行できる JDBC プログラムはほとんどすべてサーバー上でも実行できます。samples ディレクトリ内のすべてのプログラムは、少し修正するのみでサーバー上で実行できます。通常、修正は接続文に関するもののみです。

たとえば、2-8 ページの「JDBC およびデータベース接続のテスト : JdbcCheckup」で説明したテスト・プログラム JdbcCheckup.java について考えてみます。このプログラムをサーバー上で実行して DriverManager.getConnection() メソッドを使用して接続する場合は、任意のテキスト・エディタでこのファイルをオープンし、接続文字列のドライバ名を oci から kprb に変更します。たとえば、次のようになります。

```
Connection conn = DriverManager.getConnection
                  ("jdbc:oracle:kprb:@" + database, user, password);
```

このメソッドを使用する利点は、元のプログラムで短い文字列を 1 つ変更するのみで済むことです。欠点は、ドライバが、ユーザー、パスワードおよびデータベースの情報を破棄してもそれらを指定する必要がある点です。さらに、getConnection() メソッドを再度発行した場合は、ドライバはもう 1 つ新しい（不必要な）接続オブジェクトを作成します。

しかし、defaultConnection()（サーバー側内部ドライバからデータベースへの接続に使用することが好ましいメソッド）を使用して接続した場合は、ユーザー、パスワードまたはデータベースの情報を入力する必要はありません。これらの文はプログラムから削除できます。

接続文は次のように記述します。

```
Connection conn =  new oracle.jdbc.OracleDriver().defaultConnection();
```

次の例は、JdbcCheckup.java プログラムを書き直したもので、defaultConnection() 接続文を使用しています。接続文は太字で記述されています。不必要なユーザー、パスワードおよびデータベースの情報文は、標準の入力から読み込むユーティリティ機能とともに削除されています。

```
/*
 * This sample can be used to check the JDBC installation.
 * Just run it and provide the connect information.  It will select
 * "Hello World" from the database.
 */
// You need to import the java.sql package to use JDBC
import java.sql.*;
// We import java.io to be able to read from the command line
import java.io.*;

class JdbcCheckup
{
   public static void main (String args []) throws SQLException, IOException
   {
      // Load the Oracle JDBC driver
      DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

      Connection conn =
           new oracle.jdbc.OracleDriver ().defaultConnection ();

       // Create a statement
       Statement stmt = conn.createStatement ();

       // Do the SQL "Hello World" thing
       ResultSet rset = stmt.executeQuery ("SELECT 'Hello World' FROM dual");

       while (rset.next ())
          System.out.println (rset.getString (1));
       System.out.println ("Your JDBC installation is correct.");
   }
}
```

サーバーへのアプリケーションのロード

サーバーにアプリケーションをロードする場合は、クライアントでコンパイル済みの .class ファイルをロードしたり、.java ソース・ファイルをロードして、サーバーで自動的にコンパイルさせることができます。

どちらの場合にも Oracle loadjava クライアント側ユーティリティを使用して、ファイルをロードします。コマンドラインでソース・ファイル名を指定するか（コマンドラインではワイルド・カードを使用できます）、JAR ファイルにそのファイルを格納して、コマンドラインでその JAR ファイル名を指定します。loadjava ユーティリティの詳細は、『Oracle9*i* Java Developer's Guide』を参照してください。

実際のユーティリティを実行する loadjava スクリプトは、[Oracle Home] ディレクトリの bin サブディレクトリにあります。このディレクトリは、Oracle をインストールしたパスにあります。

注意： loadjava ユーティリティは圧縮ファイルをサポートしています。

サーバーへのクラス・ファイルのロード

アプリケーションに 3 つのクラス・ファイル（Foo1.class、Foo2.class および Foo3.class）がある場合について検討します。次の 3 つの例では、次のことを実行します。1）個々のクラス・ファイル名を指定します。2）ワイルド・カードを使用したクラス・ファイル名を指定します。3）クラス・ファイルが入っている JAR ファイルを指定します。

各クラスは、サーバーで独自のクラス・スキーマ・オブジェクトに書き込まれます。

次の 3 つの例では、ファイルをロードするためにデフォルト OCI ドライバを使用しています。

```
loadjava -user scott/tiger Foo1.class Foo2.class Foo3.class
```

または

```
loadjava -user scott/tiger Foo*.class
```

または

```
loadjava -user scott/tiger Foo.jar
```

または、Thin ドライバでロードするには次のコマンドを使用します（-thin オプションと適切な URL を指定します）。

```
loadjava -thin -user scott/tiger@localhost:1521:ORCL Foo.jar
```

（クラスをロードするために OCI ドライバまたは Thin ドライバのどちらを使用するかは、使用する環境とユーザーにとってどちらが使用しやすいかによります。）

注意： サーバー側埋込み JVM は、JDK 1.2.x を使用しているため、サーバーにロードする場合に、JDK 1.2.x でクラスをコンパイルすることをお薦めします。これによって、実行時ではなく、コンパイル時に非互換性が捕捉されます（たとえば、JDK 1.1.x で開発した `oracle.jdbc2` パッケージの使用するようなアプリケーションなど）。

サーバーへのソース・ファイルのロード

`.java` ソース・ファイルのロードに関して、`loadjava -resolve` オプションを有効にした場合は、サーバー側コンパイラはロード時にアプリケーションをコンパイルし、その結果オリジナル・ソース・コードのソース・スキーマ・オブジェクトとコンパイル済み出力の 1 つ以上のクラス・スキーマ・オブジェクトの両方が生成されます。

`-resolve` を指定しない場合は、ソースはコンパイルされずに、ソース・スキーマ・オブジェクトにロードされます。ただし、この場合はソースで定義されたクラスを最初に使用しようとしたときに、ソースは暗黙的にコンパイルされます。

たとえば、デフォルト OCI ドライバを使用して、`Foo.java` をロードしてコンパイルするには、次のように `loadjava` を実行します。

```
loadjava -user scott/tiger -resolve Foo.java
```

または、Thin ドライバでロードするには次のコマンドを使用します（`-thin` オプションと適切な URL を指定します）。

```
loadjava -thin -user scott/tiger@localhost:1521:ORCL -resolve Foo.java
```

どちらの場合にも、ソース・スキーマ・オブジェクトに加えて、適切なクラス・スキーマ・オブジェクトが作成されます。

注意： 通常はできるかぎりクライアントでソースをコンパイルし、ソース・ファイルではなく、`.class` ファイルをサーバーにロードすることをお薦めします。

oracle.sql.CHAR データのサーバー側キャラクタ・セット変換

サーバー側内部ドライバは、C 言語で `oracle.sql.CHAR` のキャラクタ・セット変換を実行します。これは Java で `oracle.sql.CHAR` のキャラクタ・セット変換を実行する、クライアント側ドライバとは異なり、よりパフォーマンスが高い実装です。`oracle.sql.CHAR` クラスの詳細は、5-28 ページの「クラス oracle.sql.CHAR」を参照してください。

19

コーディングのヒントおよびトラブルシューティング

この章では、JDBC アプリケーションまたはアプレットの最適化の方法およびトラブルシューティングについて説明します。次の項目が含まれます。

- JDBC とマルチスレッド
- パフォーマンスの最適化
- 一般的な問題
- 基本的なデバッグ処理
- トランザクション分離レベルとアクセス・モード

JDBC とマルチスレッド

Oracle JDBC ドライバは、Java マルチスレッドを使用するプログラムを完全にサポートします。次の例では、指定された数のスレッドを作成し、スレッドで接続を共有するかどうかを判断できるようにします。接続を共有する場合、すべてのスレッドで同じ JDBC 接続オブジェクトが使用されます（ただし、各スレッドは、独自の文オブジェクトを使用します）。

Oracle JDBC API のメソッドはすべて同期化されるので、2 つのスレッドが同時に接続オブジェクトを使用しようとした場合、どちらか一方はもう一方が使用し終わるまで待機させられます。

プログラムは、スレッド ID と、そのスレッドに対応付けられた従業員名および従業員 ID を表示します。

次のコマンドを入力してプログラムを実行します。

```
java JdbcMTSample [number_of_threads] [share]
```

number_of_threads は、作成するスレッドの数です。`share` は、スレッドによる接続の共有を指定します。スレッド数を指定しなかった場合は、デフォルトで 10 スレッドが作成されます。

この例は、20-19 ページの「マルチスレッド : JdbcMTSample.java」と同じものです。

```
/*
 * This sample is a  multi-threaded JDBC program.
 */

import java.sql.*;
import oracle.jdbc.OracleStatement;

public class JdbcMTSample extends Thread
{
  // Default no of threads to 10
  private static int NUM_OF_THREADS = 10;

  int m_myId;

  static  int c_nextId = 1;
  static  Connection s_conn = null;
  static  boolean   share_connection = false;

  synchronized static int getNextId()
  {
      return c_nextId++;
  }
```

```
public static void main (String args [])
{
  try
  {
    /* Load the JDBC driver */
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    // If NoOfThreads is specified, then read it
    if ((args.length > 2)  ||
         ((args.length > 1) && !(args[1].equals("share"))))
    {
       System.out.println("Error: Invalid Syntax. ");
       System.out.println("java JdbcMTSample [NoOfThreads] [share]");
       System.exit(0);
    }

    if (args.length > 1)
    {
       share_connection = true;
       System.out.println
              ("All threads will be sharing the same connection");
    }

    // get the no of threads if given
    if (args.length > 0)
       NUM_OF_THREADS = Integer.parseInt (args[0]);

    // get a shared connection
    if (share_connection)
        s_conn = DriverManager.getConnection
                              ("jdbc:oracle:" +args[1], "scott","tiger");

    // Create the threads
    Thread[] threadList = new Thread[NUM_OF_THREADS];

    // spawn threads
    for (int i = 0; i < NUM_OF_THREADS; i++)
    {
        threadList[i] = new JdbcMTSample();
        threadList[i].start();
    }

    // Start everyone at the same time
    setGreenLight ();
```

```
    // wait for all threads to end
    for (int i = 0; i < NUM_OF_THREADS; i++)
    {
        threadList[i].join();
    }

    if (share_connection)
    {
        s_conn.close();
        s_conn = null;
    }

  }
  catch (Exception e)
  {
     e.printStackTrace();
  }

}

public JdbcMTSample()
{
   super();
   // Assign an Id to the thread
   m_myId = getNextId();
}

public void run()
{
  Connection conn = null;
  ResultSet     rs   = null;
  Statement  stmt = null;

  try
  {
    // Get the connection

    if (share_connection)
      stmt = s_conn.createStatement (); // Create a Statement
    else
    {
      conn = DriverManager.getConnection("jdbc:oracle:oci8:@",
                                         "scott","tiger");
      stmt = conn.createStatement (); // Create a Statement
    }
```

```
      while (!getGreenLight())
        yield();

      // Execute the Query
      rs = stmt.executeQuery ("select * from EMP");

      // Loop through the results
      while (rs.next())
      {
        System.out.println("Thread " + m_myId +
                           " Employee Id : " + rs.getInt(1) +
                           " Name : " + rs.getString(2));
        yield();  // Yield To other threads
      }

      // Close all the resources
      rs.close();
      rs = null;

      // Close the statement
      stmt.close();
      stmt = null;

      // Close the local connection
      if ((!share_connection) && (conn != null))
      {
         conn.close();
         conn = null;
      }
      System.out.println("Thread " + m_myId +  " is finished. ");
    }
    catch (Exception e)
    {
      System.out.println("Thread " + m_myId + " got Exception: " + e);
      e.printStackTrace();
      return;
    }
  }
  static boolean greenLight = false;
  static synchronized void setGreenLight () { greenLight = true; }
  synchronized boolean getGreenLight () { return greenLight; }
}
```

パフォーマンスの最適化

次の機能を使用して、JDBC プログラムのパフォーマンスを大幅に向上させることができます。

- 自動コミット・モードの無効化
- 標準フェッチ・サイズと Oracle 行プリフェッチ
- 標準バッチ更新と Oracle バッチ更新

自動コミット・モードの無効化

自動コミット・モードは、SQL 操作を実行するたびに自動的に COMMIT 操作を発行するかどうかを、データベースに対して指示します。自動コミット・モードにすると、異なるバインド変数で同じ文を繰り返すような場合などに、時間と処理能力の面で大きな負荷がかかる場合があります。

デフォルトでは、新規の接続オブジェクトは自動コミット・モードが有効になります。ただし、接続オブジェクト（java.sql.Conection または oracle.jdbc.OracleConnection のどちらか）の setAutoCommit() メソッドで自動コミット・モードを無効化できます。

自動コミット・モードでは、文が完了した時点または次の実行が発生した時点のうち、どちらか早い方で COMMIT 操作が発生します。ResultSet を戻す文の場合は、ResultSet の最後の行が取り出されたとき、または ResultSet がクローズしたときに文が完了します。より複雑なケースでは、1 つの文で出力パラメータ値や複数の結果が返されることがあります。この場合、すべての結果および出力パラメータ値が取り出された時点で COMMIT が発生します。

setAutoCommit(false) をコールして自動コミット・モードを無効化した場合、接続オブジェクトの commit() または rollback() メソッドを使用して、操作のグループを手動でコミットまたはロールバックする必要があります。

例 : 自動コミットの無効化 次に、ドライバをロードしてデータベースに接続する例を示します。新しい接続はデフォルトで自動コミット・モードが有効になるので、この例では自動コミットを無効化する方法を示します。この例では、conn は Connection オブジェクトを、stmt は Statement オブジェクトを表します。

```
// Load the Oracle JDBC driver
DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

// Connect to the database
// You can put a database hostname after the @ sign in the connection URL.
Connection conn =
      DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");
```

```
// It's faster when auto commit is off
conn.setAutoCommit (false);

// Create a Statement
Statement stmt = conn.createStatement ();
...
```

標準フェッチ・サイズと Oracle 行プリフェッチ

問合せ中に結果セットが移入されるときに、データベースへの 1 回のラウンドトリップでクライアントにプリフェッチされる行数は、Oracle JDBC の接続オブジェクトおよび文オブジェクトで指定できます。接続オブジェクトに値を設定すると、その接続で作成される文すべてに適用できます。また、その値を特定の文オブジェクトでオーバーライドすることもできます。接続オブジェクトのデフォルト値は 10 です。データをクライアントにプリフェッチすると、サーバーへのラウンドトリップの回数を削減できます。

JDBC 2.0 では、同様、かつさらに柔軟に、文オブジェクトにも（後続の問合せに影響）、結果セット・オブジェクトにも（行の再フェッチに影響）、1 回のラウンドトリップでフェッチする行の数を指定できます。デフォルトでは、その結果セットを作成した文オブジェクトの値が、結果セットで使用されます。JDBC 2.0 フェッチ・サイズを設定しないと、Oracle 接続行プリフェッチ値がデフォルトとして使用されます。

詳細は、12-19 ページの「Oracle 行プリフェッチ」および 11-20 ページの「フェッチ・サイズ」を参照してください。

標準バッチ更新と Oracle バッチ更新

Oracle JDBC ドライバでは、プリコンパイルされた SQL 文の `INSERT`、`DELETE` および `UPDATE` 操作をクライアントで蓄積し、一回でサーバーに送信することができます。この機能を使用すると、サーバーとのラウンドトリップを減少できます。Oracle バッチ更新または標準バッチ更新のどちらかを使用できます。Oracle バッチ更新では、通常、あらかじめ設定されているバッチ値に達すると、暗黙的にバッチが実行されます。標準バッチ更新では、バッチは明示的に実行されます。

バッチ更新モデルの説明および使用方法については、12-2 ページの「バッチ更新」を参照してください。

一般的な問題

この項では、Oracle JDBC ドライバの使用中に発生する可能性のある、一般的な問題について説明します。たとえば、次の問題があります。

- OUT または IN/OUT 変数として定義された CHAR 列に対する空白の埋込み
- メモリー・リークおよびカーソルの不足
- PL/SQL ストアド・プロシージャのブール型パラメータ
- 1 プロセスで可能な OCI 接続のオープン数について

OUT または IN/OUT 変数として定義された CHAR 列に対する空白の埋込み

PL/SQL では、OUT または IN/OUT 変数として定義された CHAR 列は、必要に応じて空白を埋め込まれた長さ 32767 バイトのレコードで返されます。VARCHAR2 列ではこのようにはなりません。

この問題を避けるには、Statement オブジェクトで setMaxFieldSize() メソッドを使用し、すべての列に最大データ長を設定できるようにします。データ長は setMaxFieldSize() に指定した値になります。必要に応じて空白が埋め込まれます。このメソッドは文固有で、CHAR、RAW、LONG、LONG RAW および VARCHAR2 列のすべての長さに影響するので、setMaxFieldSize() の値を選択するときには注意が必要です。

この機能を使用可能にするには、OUT 変数を登録する前に、setMaxFieldSize() メソッドを起動する必要があります。

メモリー・リークおよびカーソルの不足

カーソルまたはメモリーが不足しているというメッセージを受け取った場合は、すべての Statement および ResultSet オブジェクトを明示的にクローズしてください。Oracle JDBC ドライバには、ファイナライザ・メソッドがありません。クリーン・アップ・ルーチンは、ResultSet および Statement クラスの close() メソッドで実行されます。結果セットおよび文オブジェクトを明示的にクローズしておかないと、重大なメモリー・リークが発生します。また、データベースのカーソルが不足します。結果セットまたは文をクローズすると、データベース内の対応するカーソルが解放されます。

同様に、サーバー側でのリークおよびカーソル不足を避けるには、Connection オブジェクトも明示的にクローズしておく必要があります。接続をクローズすると、その接続に対応付けられたオープン中の文オブジェクトが、JDBC ドライバによってクローズされ、サーバー側のカーソル・オブジェクトが解放されます。

PL/SQL ストアド・プロシージャのブール型パラメータ

OCI レイヤーの制限により、JDBC ドライバでは、PL/SQL ストアド・プロシージャに BOOLEAN パラメータを渡せません。PL/SQL プロシージャに BOOLEAN 値が含まれている場合、その PL/SQL プロシージャを、引数を INT として受け取る 2 番目の PL/SQL プロシージャでラップし、最初のストアド・プロシージャに渡します。2 番目のプロシージャがコールされると、サーバーによって INT から BOOLEAN に変換されます。

ストアド・プロシージャの例を示します。BOOLPROC は、BOOLEAN パラメータを渡そうとします。2 番目のプロシージャ BOOLWRAP は、BOOLEAN 値と INT 値の置換えを実行します。

```
CREATE OR REPLACE PROCEDURE boolproc(x boolean)
AS
BEGIN
[...]
END;

CREATE OR REPLACE PROCEDURE boolwrap(x int)
AS
BEGIN
IF (x=1) THEN
  boolproc(TRUE);
ELSE
  boolproc(FALSE);
END IF;
END;

// Create the database connection
Connection conn = DriverManager.getConnection
                  ("jdbc:oracle:oci8:@<...hoststring...>", "scott", "tiger");
CallableStatement cs = conn.prepareCall ("begin boolwrap(?); end;");
cs.setInt(1, 1);
cs.execute ();
```

1 プロセスで可能な OCI 接続のオープン数について

任意の時点で 1 プロセスで約 17 以上の JDBC-OCI 接続をオープンできないことがあります。サーバー上のプロセス数が初期化ファイルに指定されている制限を超えたためか、またはプロセスごとのファイル記述子の制限を超えた可能性があります。1 つの JDBC-OCI 接続で、複数のファイル記述子が使用される（3 ～ 4 個のファイル記述子が使用されます）ことがあります。

サーバーで 17 以上のプロセスを使用可能にしている場合は、プロセスごとのファイル記述子の制限が原因である可能性があります。この制限を上げると、解決する可能性があります。

基本的なデバッグ処理

JDBC プログラムのデバッグ方法について説明します。

- ネットワーク・イベントをトラップするための Oracle Net トレース
- サード・パーティのデバッグ・ツール

デバッグに役立つスタック・トレースの出力など、SQL 例外の処理については、3-33 ページの「SQL 例外の処理」を参照してください。

ネットワーク・イベントをトラップするための Oracle Net トレース

クライアントおよびサーバーで Oracle Net トレースを使用可能にすると、Oracle Net を介して送信されたパケットをトラップできます。クライアント側でのトレースは、JDBC OCI ドライバのみ使用できます。JDBC Thin ドライバでは使用できません。トレースおよびトレース・ファイルの読み方の詳細は、『Oracle9*i* Net Services 管理者ガイド』を参照してください。

トレース機能を使用すると、ネットワーク・イベントが実行されるたびにそのイベントについて記述される、一連の詳細な文が生成されます。操作をトレースすることにより、イベントの内部操作に関する詳細な情報を取り出すことができます。この情報は読込み可能ファイルに出力され、エラーの原因となったイベントを特定できます。トレース情報の収集は、`SQLNET.ORA` ファイルにあるいくつかの Oracle Net パラメータによって制御されます。`SQLNET.ORA` のパラメータを設定した後に、トレースを実行するために新しい接続を作成する必要があります。

トレース・レベルが高いほど、より詳細な情報がトレース・ファイルに書き込まれます。トレース・ファイルの内容が複雑になることがあるので、トレースを使用可能にするときはトレース・レベル 4 から始めてください。トレース・ファイルの最初の部分には、接続ハンドシェイク情報が書き込まれます。JDBC プログラムに関連する SQL 文およびエラー・メッセージについては、接続ハンドシェイク情報以降を参照してください。

注意： トレース機能ではディスク領域が大量に使用されるため、システムのパフォーマンスが大幅に低下する可能性があります。トレースは、必要なときにのみ使用可能にしてください。

クライアント側でのトレース

クライアント・システムの SQLNET.ORA ファイルに、次のパラメータを設定します。

TRACE_LEVEL_CLIENT

目的 トレースを一定の指定レベルでオン / オフにします。

デフォルト値 0 または OFF

有効な値
- 0 または OFF ― トレースの出力なし
- 4 または USER ― ユーザー・トレース情報
- 10 または ADMIN ― 管理トレース情報
- 16 または SUPPORT ― カスタマ・サポート・トレース情報

例 : TRACE_LEVEL_CLIENT=10

TRACE_DIRECTORY_CLIENT

目的 トレース・ファイルの書込み先ディレクトリを指定します。

デフォルト値 $ORACLE_HOME/network/trace

例 : UNIX の場合 : TRACE_DIRECTORY_CLIENT=/oracle/traces

Windows NT の場合 : TRACE_DIRECTORY_CLIENT=C:¥ORACLE¥TRACES

TRACE_FILE_CLIENT

目的 クライアント・トレース・ファイルの名前を指定します。

デフォルト値 SQLNET.TRC

例 : TRACE_FILE_CLIENT=cli_Connection1.trc

注意 : TRACE_FILE_CLIENT ファイルには、TRACE_FILE_SERVER ファイルとは異なる名前を付けてください。

TRACE_UNIQUE_CLIENT

目的 クライアント側の各トレースに一意の名前を付け、各トレース・ファイルが次に発生したクライアント・トレースによって上書きされないようにします。ファイル名の最後に PID が付加されます。

デフォルト値 OFF

例 : TRACE_UNIQUE_CLIENT = ON

サーバー側のトレース

サーバー・システムの SQLNET.ORA ファイルに次のパラメータを設定します。接続ごとに、一意のファイル名を持つ個別のファイルが生成されます。

TRACE_LEVEL_SERVER

目的 トレースを一定の指定レベルでオン / オフにします。

デフォルト値 0 または OFF

有効な値

- 0 または OFF ― トレースの出力なし
- 4 または USER ― ユーザー・トレース情報
- 10 または ADMIN ― 管理トレース情報
- 16 または SUPPORT ― カスタマ・サポート・トレース情報

例 : TRACE_LEVEL_SERVER=10

TRACE_DIRECTORY_SERVER

目的 トレース・ファイルの書込み先ディレクトリを指定します。

デフォルト値 $ORACLE_HOME/network/trace

例 : TRACE_DIRECTORY_SERVER=/oracle/traces

TRACE_FILE_SERVER

目的 サーバー・トレース・ファイルの名前を指定します。

デフォルト値 SERVER.TRC

例 : TRACE_FILE_SERVER= svr_Connection1.trc

注意 : TRACE_FILE_SERVER には、TRACE_FILE_CLIENT ファイルとは異なる名前を付けてください。

サード・パーティのデバッグ・ツール

Intersolv 社の JDBCSpy および JDBCTest などのツールを使用して、JDBC API レベルで問題に対処することができます。これらのツールは、ODBCSpy や ODBCTest と類似しています。

トランザクション分離レベルとアクセス・モード

読取り専用接続は、Oracle Server ではサポートされていますが、Oracle JDBC ドライバではサポートされていません。

Oracle Server では、トランザクションに対して、TRANSACTION_READ_COMMITTED および TRANSACTION_SERIALIZABLE トランザクション分離レベルのみがサポートされています。デフォルトは TRANSACTION_READ_COMMITTED です。レベルの取出しおよび設定を行うには、oracle.jdbc.OracleConnection クラスの次のメソッドを使用します。

- getTransactionIsolation(): この接続のカレント・トランザクション分離レベルを取得します。
- setTransactionIsolation(): TRANSACTION_* 値の 1 つを使用してトランザクション分離レベルを変更します。

20

サンプル・アプリケーション

この章では、標準 JDBC 機能および Oracle 固有 JDBC 機能の両方について、サンプル・アプリケーションを示します。次の項目が含まれます。

- 基本サンプル
- JDBC の PL/SQL のサンプル
- 中級レベルのサンプル
- JDBC 2.0 型のサンプル
- Oracle 型拡張要素のサンプル
- カスタム・オブジェクト・クラスのサンプル
- JDBC 2.0 結果セット拡張のサンプル
- パフォーマンス強化のサンプル
- 接続プーリングと分散トランザクションのサンプル
- サンプル・アプレット
- JDBC サンプル・コードと SQLJ サンプル・コード

これらのサンプルは、次のディレクトリの各サブディレクトリにあります。

```
[Oracle Home]/jdbc/demo/samples
```

注意： Thin ドライバを必要とするサンプル・アプレットを除き、この章のサンプルはすべて、どの JDBC ドライバでも動作します。ほとんどのサンプルは、製品 CD の `oci8` ディレクトリの下にありますが、誤解しないよう注意してください。

基本サンプル

表から従業員情報を出力し、表に従業員情報を挿入する、基本サンプルを示します。

- EMP 表からの名前のリストの取出し : Employee.java
- EMP 表への名前の挿入 : InsertExample.java

これらのサンプルは、次のディレクトリにあります。

```
[Oracle Home]/jdbc/demo/samples/oci8/basic-samples
```

基本 JDBC 機能の段階的な説明は、3-2 ページの「JDBC での最初の処理」を参照してください。

EMP 表からの名前のリストの取出し : Employee.java

EMP 表からすべての従業員名を取り出して出力します。

> **注意 :** この Employee.java と、後で使用するカスタム Java クラスの ORAData 実装例の Employee.java を混同しないでください。

```
/*
 * This sample shows how to list all the names from the EMP table
 */

// You need to import the java.sql package to use JDBC
import java.sql.*;

class Employee
{
  public static void main (String args [])
       throws SQLException
  {
    // Load the Oracle JDBC driver
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    // Connect to the database
    // You can put a database name after the @ sign in the connection URL.
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

    // Create a Statement
    Statement stmt = conn.createStatement ();
```

```
    // Select the ENAME column from the EMP table
    ResultSet rset = stmt.executeQuery ("select ENAME from EMP");

    // Iterate through the result and print the employee names
    while (rset.next ())
      System.out.println (rset.getString (1));

    // Close the RseultSet
    rset.close();

    // Close the Statement
    stmt.close();

    // Close the connection
    conn.close();
  }
}
```

EMP 表への名前の挿入 : InsertExample.java

この例では、プリコンパイルされた SQL 文を使用して、新しい従業員の行を EMP 表に挿入します。

```
/*
 * This sample shows how to insert data in a table.
 */

// You need to import the java.sql package to use JDBC
import java.sql.*;

class InsertExample
{
  public static void main (String args [])
       throws SQLException
  {
    // Load the Oracle JDBC driver
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    // Connect to the database
    // You can put a database name after the @ sign in the connection URL.
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");
```

```
    // Prepare a statement to cleanup the emp table
    Statement stmt = conn.createStatement ();
    try
    {
      stmt.execute ("delete from EMP where EMPNO = 1500");
    }
    catch (SQLException e)
    {
      // Ignore an error here
    }

    try
    {
      stmt.execute ("delete from EMP where EMPNO = 507");
    }
    catch (SQLException e)
    {
      // Ignore an error here too
    }

    // Close the statement
    stmt.close();

    // Prepare to insert new names in the EMP table
    PreparedStatement pstmt =
      conn.prepareStatement ("insert into EMP (EMPNO, ENAME) values (?, ?)");

    // Add LESLIE as employee number 1500
    pstmt.setInt (1, 1500);          // The first ? is for EMPNO
    pstmt.setString (2, "LESLIE");   // The second ? is for ENAME
    // Do the insertion
    pstmt.execute ();

    // Add MARSHA as employee number 507
    pstmt.setInt (1, 507);           // The first ? is for EMPNO
    pstmt.setString (2, "MARSHA");   // The second ? is for ENAME
    // Do the insertion
    pstmt.execute ();

    // Close the statement
    pstmt.close();

    // Close the connecion
    conn.close();

  }
}
```

JDBC の PL/SQL のサンプル

次の例で、標準 SQL92 コール構文と Oracle PL/SQL ブロック構文を対比させ、PL/SQL と JDBC の相互運用性を示します。

- PL/SQL ブロックでのプロシージャの実行 : PLSQL.java
- PL/SQL ストアド・プロシージャのコール : PLSQLExample.java

これらのサンプルは、次のディレクトリにあります。

```
[Oracle Home]/jdbc/demo/samples/oci8/basic-samples
```

この例に関する説明は、3-31 ページの「PL/SQL ストアド・プロシージャ」を参照してください。

PL/SQL ストアド・プロシージャのコール : PLSQLExample.java

このサンプルでは、ストアド・ファンクションを定義し、コール可能文で SQL92 CALL 構文を使用してそれを実行します。ファンクションは、入力として従業員名と給与を取り、設定された金額分給与を上げます。

```
/*
 * This sample shows how to call a PL/SQL stored procedure using the SQL92
 * syntax.  See also the other sample PLSQL.java.
 */

import java.sql.*;
import java.io.*;

class PLSQLExample
{
  public static void main (String args [])
       throws SQLException, IOException
  {
    // Load the driver
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    // Connect to the database
    // You can put a database name after the @ sign in the connection URL.
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

    // Create a statement
    Statement stmt = conn.createStatement ();
```

```
    // Create the stored function
    stmt.execute ("create or replace function RAISESAL (name CHAR, raise NUMBER)
                return NUMBER is begin return raise + 100000; end;");

    // Close the statement
    stmt.close();

    // Prepare to call the stored procedure RAISESAL.
    // This sample uses the SQL92 syntax
    CallableStatement cstmt = conn.prepareCall ("{? = call RAISESAL (?, ?)}");

    // Declare that the first ? is a return value of type Int
    cstmt.registerOutParameter (1, Types.INTEGER);

    // We want to raise LESLIE's salary by 20,000
    cstmt.setString (2, "LESLIE");  // The name argument is the second ?
    cstmt.setInt (3, 20000);        // The raise argument is the third ?

    // Do the raise
    cstmt.execute ();

    // Get the new salary back
    int new_salary = cstmt.getInt (1);

    System.out.println ("The new salary is: " + new_salary);

    // Close the statement
    cstmt.close();

    // Close the connection
    conn.close();
  }
}
```

PL/SQL ブロックでのプロシージャの実行 : PLSQL.java

このサンプルでは、PL/SQL ストアド・プロシージャおよびファンクションを定義し、コール可能文の Oracle PL/SQL BEGIN...END ブロックの内側からそれらを実行します。入力、出力、入出力、戻り値の各パラメータを持つ、ストアド・プロシージャおよびファンクションが示されます。

```
/*
 * This sample shows how to call PL/SQL blocks from JDBC.
 */

import java.sql.*;

class PLSQL
{
  public static void main (String args [])
       throws SQLException, ClassNotFoundException
  {
    // Load the driver
    Class.forName ("oracle.jdbc.OracleDriver");

    // Connect to the database
    // You can put a database name after the @ sign in the connection URL.
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

    // Create the stored procedures
    init (conn);

    // Cleanup the plsqltest database
    Statement stmt = conn.createStatement ();
    stmt.execute ("delete from plsqltest");

    // Close the statement
    stmt.close();

    // Call a procedure with no parameters
    {
      CallableStatement procnone = conn.prepareCall ("begin procnone; end;");
      procnone.execute ();
      dumpTestTable (conn);
      procnone.close();
    }
```

```
// Call a procedure with an IN parameter
{
  CallableStatement procin = conn.prepareCall ("begin procin (?); end;");
  procin.setString (1, "testing");
  procin.execute ();
  dumpTestTable (conn);
  procin.close();
}

// Call a procedure with an OUT parameter
{
  CallableStatement procout = conn.prepareCall ("begin procout (?); end;");
  procout.registerOutParameter (1, Types.CHAR);
  procout.execute ();
  System.out.println ("Out argument is: " + procout.getString (1));
  procout.close();
}

// Call a procedure with an IN/OUT prameter
{
  CallableStatement procinout = conn.prepareCall
                               ("begin procinout (?); end;");
  procinout.registerOutParameter (1, Types.VARCHAR);
  procinout.setString (1, "testing");
  procinout.execute ();
  dumpTestTable (conn);
  System.out.println ("Out argument is: " + procinout.getString (1));
  procinout.close();
}

// Call a function with no parameters
{
  CallableStatement funcnone = conn.prepareCall
                              ("begin ? := funcnone; end;");
  funcnone.registerOutParameter (1, Types.CHAR);
  funcnone.execute ();
  System.out.println ("Return value is: " + funcnone.getString (1));
  funcnone.close();
}
```

```
  // Call a function with an IN parameter
  {
    CallableStatement funcin = conn.prepareCall
                               ("begin ? := funcin (?); end;");
    funcin.registerOutParameter (1, Types.CHAR);
    funcin.setString (2, "testing");
    funcin.execute ();
    System.out.println ("Return value is: " + funcin.getString (1));
    funcin.close();
  }

  // Call a function with an OUT parameter
  {
    CallableStatement funcout = conn.prepareCall
                                ("begin ? := funcout (?); end;");
    funcout.registerOutParameter (1, Types.CHAR);
    funcout.registerOutParameter (2, Types.CHAR);
    funcout.execute ();
    System.out.println ("Return value is: " + funcout.getString (1));
    System.out.println ("Out argument is: " + funcout.getString (2));
    funcout.close();
  }

  // Close the connection
  conn.close();
}

// Utility function to dump the contents of the PLSQLTEST table and
// clear it
static void dumpTestTable (Connection conn)
  throws SQLException
{
  Statement stmt = conn.createStatement ();
  ResultSet rset = stmt.executeQuery ("select * from plsqltest");
  while (rset.next ())
    System.out.println (rset.getString (1));
  stmt.execute ("delete from plsqltest");
  rset.close();
  stmt.close();
}
```

```
  // Utility function to create the stored procedures
  static void init (Connection conn)
    throws SQLException
  {
    Statement stmt = conn.createStatement ();
    try { stmt.execute ("drop table plsqltest"); } catch (SQLException e) { }
    stmt.execute ("create table plsqltest (x char(20))");
    stmt.execute ("create or replace procedure procnone
                   is begin insert into plsqltest values ('testing'); end;");
    stmt.execute ("create or replace procedure procin (y char)
                   is begin insert into plsqltest values (y); end;");
    stmt.execute ("create or replace procedure procout (y out char)
                   is begin y := 'tested'; end;");
    stmt.execute ("create or replace procedure procinout (y in out varchar)
                  is begin insert into plsqltest values (y);
                  y := 'tested'; end;");

    stmt.execute ("create or replace function funcnone return char
                   is begin return 'tested'; end;");
    stmt.execute ("create or replace function funcin (y char) return char
                  is begin return y || y; end;");
    stmt.execute ("create or replace function funcout (y out char) return char
                 is begin y := 'tested'; return 'returned'; end;");
    stmt.close();
  }
}
```

JDBC からの PL/SQL 索引付き表へのアクセス : PLSQLIndexTab.java

このコード例は、IN、OUT（ファンクション戻り値を含む）および IN OUT パラメータ・モードでの、特殊な入力バインドおよび出力登録を示しています。

この項目の詳細は、16-19 ページの「PL/SQL 索引付き表へのアクセス」を参照してください。

```
/*
 * This sample demonstrates how to make PL/SQL calls with index-by table
 * parameters
 */

// You need to import java.sql, oracle.sql and oracle.jdbc packages to use
import java.sql.*;
import oracle.jdbc.*;
import oracle.sql.*;
```

```
class PLSQLIndexTab
{
  public static void main (String args [])
       throws SQLException
  {

    String [] plSqlIndexArrayIn = {"string1","string2","string3"};
    int currentLen = plSqlIndexArrayIn.length;
    int maxLen = currentLen;
    int elementMaxLen = 20;

    // Load the Oracle JDBC driver
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    String url = "jdbc:oracle:oci8:@";
    try {
      String url1 = System.getProperty("JDBC_URL");
      if (url1 != null)
        url = url1;
    } catch (Exception e) {
      // If there is any security exception, ignore it
      // and use the default
    }

    // Connect to the database
    Connection conn =
      DriverManager.getConnection (url, "scott", "tiger");

    // Create the procedures which use Index-by Table as IN/OUT parameters
    createProc_Func(conn);

    // Call a procedure with an IN parameter
    System.out.println ("Call a procedure with an IN parameter");
    OracleCallableStatement cs =
        (OracleCallableStatement) conn.prepareCall ("begin proc_in (?, ?); end;");

    // Use setPlsqlIndexTable() to set the Index-by Table parameter
    cs.setPlsqlIndexTable (1, plSqlIndexArrayIn, maxLen,
                              currentLen, OracleTypes.VARCHAR, elementMaxLen);

    // Register OUT paramater
    cs.registerOutParameter (2, Types.CHAR);

    // Call the procedure
    cs.execute ();
```

```
    // Display the Status
    System.out.println ("Status = " + cs.getString (2));

    // Call a procedure with an OUT parameter
    System.out.println ("Call a procedure with an OUT parameter");
    cs = (OracleCallableStatement) conn.prepareCall ("begin proc_out (?); end;");

    // Use setPlsqlIndexTable() to set the Index-by Table parameter
    cs.registerIndexTableOutParameter (1, maxLen, OracleTypes.VARCHAR,
    elementMaxLen);

    // Call the procedure
    cs.execute ();

    // Display the OUT value
    Datum[] val = cs.getOraclePlsqlIndexTable (1);
    for (int i = 0; i < val.length; i++)
        System.out.println ("Value = " + val[i].stringValue());

    // Call a procedure with IN/OUT parameter
    System.out.println ("Call a procedure with IN/OUT parameter");

    cs = (OracleCallableStatement) conn.prepareCall ("begin proc_inout (?, ?);
end;");

    // Use setPlsqlIndexTable() to set the Index-by Table parameter
    cs.setPlsqlIndexTable (1, plSqlIndexArrayIn, maxLen,
                               currentLen, OracleTypes.VARCHAR, elementMaxLen);

    // Register OUT paramater
    cs.registerIndexTableOutParameter (1, maxLen, OracleTypes.VARCHAR,
    elementMaxLen);
    cs.registerOutParameter (2, Types.CHAR);

    // Call the procedure
    cs.execute ();

    // Display the Status
    System.out.println ("Status = " + cs.getString (2));

    // Display the OUT value
    val = cs.getOraclePlsqlIndexTable (1);
    for (int i = 0; i < val.length; i++)
        System.out.println ("Value = " + val[i].stringValue());

    // Call the Function
    System.out.println ("Call the function");
```

```
    cs = (OracleCallableStatement) conn.prepareCall ("begin ? := func (?); end;");

    // Use setPlsqlIndexTable() to set the Index-by Table parameter
    cs.setPlsqlIndexTable (2, plSqlIndexArrayIn, maxLen,
                               currentLen, OracleTypes.VARCHAR, elementMaxLen);

    // Register OUT paramater
    cs.registerIndexTableOutParameter (1, maxLen, OracleTypes.VARCHAR,
elementMaxLen);

    // Call the procedure
    cs.execute ();

    val = cs.getOraclePlsqlIndexTable (1);
    for (int i = 0; i < val.length; i++)
        System.out.println ("Value = " + val[i].stringValue());

    // Close the Callable Statement
    cs.close();

    // Dump the contents of the demo_tab
    System.out.println ("Dump the demo_tab table");
    dumpTable(conn);

    // Clean up the schema
    cleanup(conn);

    // Close the connection
    conn.close();
  }

  private static void createProc_Func (Connection conn)
      throws SQLException
  {
    // Cleanup the schema
    cleanup(conn);

    // Create a Statement
    Statement stmt = conn.createStatement ();

    // Create the Table
    stmt.execute("CREATE TABLE demo_tab (col1 VARCHAR2(20))");
```

```
    // Use PL/SQL to create the Package
    String plsql1 = "CREATE OR REPLACE PACKAGE pkg AS " +
                    "  TYPE indexByTab IS TABLE OF VARCHAR2(20) INDEX BY
                       BINARY_INTEGER; " +
                    "END;";

    stmt.execute(plsql1);

    // Create a procedure to use the Index-by Table as IN paramater
    String plsql2 = "CREATE OR REPLACE PROCEDURE proc_in (p1 IN pkg.indexByTab,
status OUT VARCHAR2) IS " +
                    "BEGIN " +
                    "  FOR i in 1..3 LOOP " +
                    "    INSERT INTO demo_tab VALUES (p1(i)); " +
                    "  END LOOP; " +
                    "  IF ((p1(1)='string1') AND (p1(2)='string2') AND
                          (p1(3)='string3')) " +
                    "     THEN status := 'Values passed in correctly'; " +
                    "  ELSE " +
                    "      status := 'Values passed in are incorrect'; " +
                    "  END IF; " +
                    "END;";

    stmt.execute(plsql2);

    // Create a procedure to use the Index-by Table as OUT paramater
    String plsql3 = "CREATE OR REPLACE PROCEDURE proc_out (p1 OUT
                     pkg.indexByTab) IS " +
                    "BEGIN " +
                    "  p1(1) := 'string1'; " +
                    "  p1(2) := 'string2'; " +
                    "  p1(3) := 'string3'; " +
                    "END;";

    stmt.execute(plsql3);

    // Create a procedure to use the Index-by Table as both IN and OUT paramater
    String plsql4 = "CREATE OR REPLACE PROCEDURE proc_inout (p1 IN OUT
                     pkg.indexByTab, status OUT VARCHAR2) IS " +
                    "BEGIN " +
                    "  FOR i in 1..3 LOOP " +
                    "    INSERT INTO demo_tab VALUES (p1(i)); " +
                    "  END LOOP; " +
                    "  IF ((p1(1)='string1') AND (p1(2)='string2') AND
                        (p1(3)='string3')) " +
                    "     THEN status := 'Values passed in correctly'; " +
                    "  ELSE " +
```

```
                    "      status := 'Values passed in are incorrect'; " +
                    "  END IF; " +
                    "  p1(1) := 'string4'; " +
                    "  p1(2) := 'string5'; " +
                    "  p1(3) := 'string6'; " +
                    "END;";

  stmt.execute(plsql4);

  String plsql5 = "CREATE OR REPLACE FUNCTION func (p1 pkg.indexByTab) RETURN
                   pkg.indexByTab IS " +
                  "  n pkg.indexByTab; " +
                  "BEGIN " +
                  "  FOR i in 1..3 LOOP " +
                  "    INSERT INTO demo_tab VALUES (p1(i)); " +
                  "  END LOOP; " +
                  "  IF ((p1(1)='string1') AND (p1(2)='string2') AND
                        (p1(3)='string3')) THEN " +
                  "     n(1) := 'p1(1) correct'; " +
                  "     n(2) := 'p1(2) correct'; " +
                  "     n(3) := 'p1(3) correct'; " +
                  "  ELSE " +
                  "     n(1) := 'p1(1) wrong'; " +
                  "     n(2) := 'p1(2) wrong'; " +
                  "     n(3) := 'p1(3) wring'; " +
                  "  END IF; " +
                  "  RETURN n; " +
                  "END;";

  stmt.execute(plsql5);

  // Close the statement
  stmt.close();
}

/**
 * Utility function to dump the contents of the "demo_tab" table
 */
static void dumpTable (Connection conn) throws SQLException
{
  Statement stmt = conn.createStatement ();
  ResultSet rset = stmt.executeQuery ("select * from demo_tab");
  while (rset.next ())
    System.out.println (rset.getString (1));
  rset.close();
  stmt.close();
}
```

```
  /**
   * Cleanup data structures created in this example
   */
  static void cleanup (Connection conn) throws SQLException
  {
    Statement stmt = conn.createStatement ();

    try {
      stmt.execute ("DROP TABLE demo_tab");
    } catch (SQLException e) {}

    try {
      stmt.execute ("DROP PROCEDURE proc_in");
    } catch (SQLException e) {}

    try {
      stmt.execute ("DROP PROCEDURE proc_out");
    } catch (SQLException e) {}

    try {
      stmt.execute ("DROP PROCEDURE proc_inout");
    } catch (SQLException e) {}

    try {
      stmt.execute ("DROP PROCEDURE func");
    } catch (SQLException e) {}

    try {
      stmt.execute ("DROP PACKAGE pck");
    } catch (SQLException e) {}

    stmt.close ();
  }

}
```

中級レベルのサンプル

この項のサンプルでは、中級レベルの JDBC 機能を示します。

- ストリーム : StreamExample.java
- マルチスレッド : JdbcMTSample.java

これらのサンプルは、次のディレクトリにあります。

```
[Oracle Home]/jdbc/demo/samples/oci8/basic-samples
```

ストリーム : StreamExample.java

JDBC ドライバは、クライアントとサーバー間での双方向のデータ・ストリーム操作をサポートします。この項のコード・サンプルでは、標準 JDBC ストリーム API を使用した、データベースへの接続および LONG データの挿入とフェッチを示します。

この項目の詳細は、3-19 ページの「JDBC 内の Java ストリーム」を参照してください。

```
/*
 * This example shows how to stream data from the database
 */

import java.sql.*;
import java.io.*;

class StreamExample
{
  public static void main (String args [])
       throws SQLException, IOException
  {
    // Load the driver
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    // Connect to the database
    // You can put a database name after the @ sign in the connection URL.
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

    // It's faster when you don't commit automatically
    conn.setAutoCommit (false);

    // Create a Statement
    Statement stmt = conn.createStatement ();
```

```
// Create the example table
try
{
  stmt.execute ("drop table streamexample");
}
catch (SQLException e)
{
  // An exception would be raised if the table did not exist
  // We just ignore it
}

// Create the table
stmt.execute ("create table streamexample
               (NAME varchar2 (256), DATA long)");

// Let's insert some data into it.  We'll put the source code
// for this very test in the database.
File file = new File ("StreamExample.java");
InputStream is = new FileInputStream ("StreamExample.java");
PreparedStatement pstmt =
  conn.prepareStatement ("insert into streamexample
                          (data, name) values (?, ?)");
pstmt.setAsciiStream (1, is, (int)file.length ());
pstmt.setString (2, "StreamExample");
pstmt.execute ();

// Do a query to get the row with NAME 'StreamExample'
ResultSet rset =
  stmt.executeQuery ("select DATA from streamexample where
                      NAME='StreamExample'");

// Get the first row
if (rset.next ())
{
  // Get the data as a Stream from Oracle to the client
  InputStream gif_data = rset.getAsciiStream (1);

  // Open a file to store the gif data
  FileOutputStream os = new FileOutputStream ("example.out");

  // Loop, reading from the gif stream and writing to the file
  int c;
  while ((c = gif_data.read ()) != -1)
    os.write (c);
```

```
      // Close the file
      os.close ();
    }

    // Close all the resources
    if (rset != null)
      rset.close();

    if (stmt != null)
      stmt.close();

    if (pstmt != null)
      pstmt.close();

    if (conn != null)
      conn.close();
  }
}
```

マルチスレッド : JdbcMTSample.java

Oracle JDBC ドライバは、Java マルチスレッドを使用するプログラムを完全にサポートします。次のサンプル・プログラムでは、指定された数のスレッドを作成し、スレッドで接続を共有するかどうかを判断できるようにします。接続を共有する場合、すべてのスレッドで同じ JDBC 接続オブジェクトが使用されます（ただし、各スレッドは、独自の文オブジェクトを使用します）。

Oracle JDBC API のメソッドはすべて（`cancel()` メソッドは除きます）同期化されるので、2 つのスレッドが同時に接続オブジェクトを使用しようとした場合、どちらか一方はもう一方が使用し終わるまで待機させられます。

プログラムは、スレッド ID と、そのスレッドに対応付けられた従業員名および従業員 ID を表示します。

このサンプルは、19-2 ページの「JDBC とマルチスレッド」のサンプルと同じです。

```
/*
 * This sample is a  multi-threaded JDBC program.
 */

import java.sql.*;
import oracle.jdbc.OracleStatement;

public class JdbcMTSample extends Thread
{
  // Default no of threads to 10
  private static int NUM_OF_THREADS = 10;
```

```
int m_myId;

static  int c_nextId = 1;
static  Connection s_conn = null;
static  boolean   share_connection = false;

synchronized static int getNextId()
{
    return c_nextId++;
}

public static void main (String args [])
{
  try
  {
    /* Load the JDBC driver */
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    // If NoOfThreads is specified, then read it
    if ((args.length > 2)  ||
         ((args.length > 1) && !(args[1].equals("share"))))
    {
       System.out.println("Error: Invalid Syntax. ");
       System.out.println("java JdbcMTSample [NoOfThreads] [share]");
       System.exit(0);
    }

    if (args.length > 1)
    {
       share_connection = true;
       System.out.println
              ("All threads will be sharing the same connection");
    }

    // get the no of threads if given
    if (args.length > 0)
       NUM_OF_THREADS = Integer.parseInt (args[0]);

    // get a shared connection
    if (share_connection)
        s_conn = DriverManager.getConnection
                              ("jdbc:oracle:oci8:@", "scott","tiger");

    // Create the threads
    Thread[] threadList = new Thread[NUM_OF_THREADS];
```

```
    // spawn threads
    for (int i = 0; i < NUM_OF_THREADS; i++)
    {
        threadList[i] = new JdbcMTSample();
        threadList[i].start();
    }

    // Start everyone at the same time
    setGreenLight ();

    // wait for all threads to end
    for (int i = 0; i < NUM_OF_THREADS; i++)
    {
        threadList[i].join();
    }

    if (share_connection)
    {
        s_conn.close();
        s_conn = null;
    }

  }
  catch (Exception e)
  {
     e.printStackTrace();
  }

}

public JdbcMTSample()
{
   super();
   // Assign an Id to the thread
   m_myId = getNextId();
}

public void run()
{
  Connection conn = null;
  ResultSet     rs   = null;
  Statement  stmt = null;

  try
  {
    // Get the connection
```

```
    if (share_connection)
      stmt = s_conn.createStatement (); // Create a Statement
    else
    {
      conn = DriverManager.getConnection("jdbc:oracle:oci8:@",
                                         "scott","tiger");
      stmt = conn.createStatement (); // Create a Statement
    }

    while (!getGreenLight())
      yield();

    // Execute the Query
    rs = stmt.executeQuery ("select * from EMP");

    // Loop through the results
    while (rs.next())
    {
      System.out.println("Thread " + m_myId +
                         " Employee Id : " + rs.getInt(1) +
                         " Name : " + rs.getString(2));
      yield();  // Yield To other threads
    }

    // Close all the resources
    rs.close();
    rs = null;

    // Close the statement
    stmt.close();
    stmt = null;

    // Close the local connection
    if ((!share_connection) && (conn != null))
    {
       conn.close();
       conn = null;
    }
    System.out.println("Thread " + m_myId +  " is finished. ");
  }
  catch (Exception e)
  {
    System.out.println("Thread " + m_myId + " got Exception: " + e);
    e.printStackTrace();
    return;
  }
}
```

```
  static boolean greenLight = false;
  static synchronized void setGreenLight () { greenLight = true; }
  synchronized boolean getGreenLight () { return greenLight; }

}
```

JDBC 2.0 型のサンプル

この項には、標準 JDBC 2.0 型の、Oracle 実装のサンプル・コードが含まれています。

- BLOB および CLOB: LobExample.java
- 弱い型指定のオブジェクト : PersonObject.java
- 弱い型指定のオブジェクト参照 : StudentRef.java
- 弱い型指定の配列 : ArrayExample.java

これらのサンプルは、次のディレクトリにあります。

```
[Oracle Home]/jdbc/demo/samples/oci8/object-samples
```

BLOB および CLOB: LobExample.java

このサンプルでは、JDBC による基本的な LOB のサポートを示します。LOB 列を含む表の作成方法を示します。また、LOB からの読込み、LOB への書込みおよび LOB の内容のダンプを行うユーティリティ・プログラムが含まれます。LOB の詳細は、7-3 ページの「BLOB と CLOB の操作」を参照してください。

```
/*
 * This sample demonstrate basic LOB support.
 */

import java.sql.*;
import java.io.*;
import java.util.*;
import oracle.jdbc.driver.*;

//needed for new CLOB and BLOB classes
import oracle.sql.*;
```

```
public class LobExample
{
  public static void main (String args [])
       throws Exception
  {
    // Register the Oracle JDBC driver
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    // Connect to the database
    // You can put a database name after the @ sign in the connection
URL.
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott",
"tiger");

    // Need to set auto commit off to update LOBs
    conn.setAutoCommit (false);

    // Create a Statement
    Statement stmt = conn.createStatement ();

    try
    {
      stmt.execute ("drop table basic_lob_table");
    }
    catch (SQLException e)
    {
      // An exception could be raised here if the table did not exist
already.
    }

    // Create a table containing a BLOB and a CLOB
    stmt.execute ("create table basic_lob_table (x varchar2 (30), b
blob, c clob)");

    // Populate the table
    stmt.execute ("insert into basic_lob_table values ('one',
'01010101010101010101010101010101', 'onetwothreefour')");
    stmt.execute ("insert into basic_lob_table values ('two',
'02020202020202020202020202020202', 'twothreefourfivesix')");

    // commit set up
    conn.commit();

    System.out.println ("Dumping lobs");
```

```
    // Select the lobs
    // note that the FOR UPDATE clause is needed for updating LOBs
    ResultSet rset = stmt.executeQuery ("select * from basic_lob_table
for update");
    while (rset.next ())
    {
      // Get the lobs
      BLOB blob = ((OracleResultSet)rset).getBLOB (2);
      CLOB clob = ((OracleResultSet)rset).getCLOB (3);

      // Print the lob contents
      dumpBlob (conn, blob);
      dumpClob (conn, clob);

      // Change the lob contents
      fillClob (conn, clob, 2000);
      fillBlob (conn, blob, 4000);

    }
    // You could rollback the changes made by fillClob() and fillBlob()
    // by issuing a rollback here
    // conn.rollback();

    System.out.println ("Dumping lobs again");

    // No need to have FOR UPDATE clause just to do selects
    rset = stmt.executeQuery ("select * from basic_lob_table");
    while (rset.next ())
    {
      // Get the lobs
      BLOB blob = ((OracleResultSet)rset).getBLOB (2);
      CLOB clob = ((OracleResultSet)rset).getCLOB (3);

      // Print the lobs contents
      dumpBlob (conn, blob);
      dumpClob (conn, clob);
    }
    // Close all resources
    rset.close();
    stmt.close();
    conn.close();
  }
```

```
// Utility function to dump Clob contents
static void dumpClob (Connection conn, CLOB clob)
  throws Exception
{
  // get character stream to retrieve clob data
  Reader instream = clob.getCharacterStream();

  // create temporary buffer for read
  char[] buffer = new char[10];

  // length of characters read
  int length = 0;

  // fetch data
  while ((length = instream.read(buffer)) != -1)
  {
    System.out.print("Read " + length + " chars: ");

    for (int i=0; i<length; i++)
      System.out.print(buffer[i]);
    System.out.println();
  }

  // Close input stream
  instream.close();
}

// Utility function to dump Blob contents
static void dumpBlob (Connection conn, BLOB blob)
  throws Exception
{
  // Get binary output stream to retrieve blob data
  InputStream instream = blob.getBinaryStream();

  // Create temporary buffer for read
  byte[] buffer = new byte[10];

  // length of bytes read
  int length = 0;

  // Fetch data
  while ((length = instream.read(buffer)) != -1)
  {
    System.out.print("Read " + length + " bytes: ");
```

```
    for (int i=0; i<length; i++)
      System.out.print(buffer[i]+" ");
    System.out.println();
  }

  // Close input stream
  instream.close();
}

// Utility function to put data in a Clob
static void fillClob (Connection conn, CLOB clob, long length)
  throws Exception
{
  Writer outstream = clob.getCharacterOutputStream();

  int i = 0;
  int chunk = 10;

  while (i < length)
  {
    outstream.write(i + "hello world", 0, chunk);

    i += chunk;
    if (length - i < chunk)
    chunk = (int) length - i;
  }
  outstream.close();
}

// Utility function to put data in a Blob
static void fillBlob (Connection conn, BLOB blob, long length)
  throws Exception
{
  OutputStream outstream = blob.getBinaryOutputStream();

  int i = 0;
  int chunk = 10;

  byte [] data = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

  while (i < length)
  {
    data [0] = (byte)i;
    outstream.write(data, 0, chunk);
```

```
      i += chunk;
      if (length - i < chunk)
      chunk = (int) length - i;
    }
    outstream.close();
  }
}
```

弱い型指定のオブジェクト : PersonObject.java

このサンプルでは、SQL 構造型オブジェクトの弱い型指定をサポートする Oracle クラス oracle.sql.STRUCT および oracle.sql.StructDescriptor の機能を示します。この列では、SQL オブジェクト型 PERSON および ADDRESS（PERSON の属性）が定義されます。

弱い型指定の STRUCT クラスの機能の詳細は、8-3 ページの「Oracle オブジェクト用のデフォルト STRUCT クラスの使用方法」を参照してください。

```
/*
 * This sample demonstrate basic Object support
 */

import java.sql.*;
import java.io.*;
import java.util.*;
import java.math.BigDecimal;
import oracle.sql.*;
import oracle.jdbc.*;

public class PersonObject
{
  public static void main (String args [])
       throws Exception
  {
    // Register the Oracle JDBC driver
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    // Connect to the database
    // You need to put your database name after the @ sign in
    // the connection URL.
    //
    // The sample retrieves an object of type "STUDENT",
    // materializes the object as an object of type ADT.
    // The Object is then modified and inserted back into the database.
```

```
Connection conn =
  DriverManager.getConnection ("jdbc:oracle:oci8:@",
                               "scott", "tiger");

// It's faster when auto commit is off
conn.setAutoCommit (false);

// Create a Statement
Statement stmt = conn.createStatement ();

try
{
  stmt.execute ("drop table people");
  stmt.execute ("drop type PERSON FORCE");
  stmt.execute ("drop type ADDRESS FORCE");
}
catch (SQLException e)
{
  // the above drop and create statements will throw exceptions
  // if the types and tables did not exist before
}

stmt.execute ("create type ADDRESS as object
               (street VARCHAR (30), num NUMBER)");
stmt.execute ("create type PERSON as object
               (name VARCHAR (30), home ADDRESS)");
stmt.execute ("create table people (empno NUMBER, empid PERSON)");

stmt.execute ("insert into people values
               (101, PERSON ('Greg', ADDRESS ('Van Ness', 345)))");
stmt.execute ("insert into people values
               (102, PERSON ('John', ADDRESS ('Geary', 229)))");

ResultSet rs = stmt.executeQuery ("select * from people");
showResultSet (rs);
rs.close();

//now insert a new row

// create a new STRUCT object with a new name and address
// create the embedded object for the address
Object [] address_attributes = new Object [2];
address_attributes [0] = "Mission";
address_attributes [1] = new BigDecimal (346);
```

```
    StructDescriptor addressDesc =
      StructDescriptor.createDescriptor ("ADDRESS", conn);
    STRUCT address = new STRUCT (addressDesc, conn, address_attributes);

    Object [] person_attributes = new Object [2];
    person_attributes [0] = "Gary";
    person_attributes [1] = address;

    StructDescriptor personDesc =
      StructDescriptor.createDescriptor("PERSON", conn);
    STRUCT new_person = new STRUCT (personDesc, conn, person_attributes);

    PreparedStatement ps =
      conn.prepareStatement ("insert into people values (?,?)");
    ps.setInt (1, 102);
    ps.setObject (2, new_person);

    ps.execute ();
    ps.close();

    rs = stmt.executeQuery ("select * from people");
    System.out.println ();
    System.out.println (" a new row has been added to the people table");
    System.out.println ();
    showResultSet (rs);

    rs.close();
    stmt.close();
    conn.close();
  }

  public static void showResultSet (ResultSet rs)
    throws SQLException
  {
    while (rs.next ())
    {
      int empno = rs.getInt (1);
      // retrieve the STRUCT
      STRUCT person_struct = (STRUCT)rs.getObject (2);
      Object person_attrs[] = person_struct.getAttributes();

      System.out.println ("person name:  " + (String) person_attrs[0]);

      STRUCT address = (STRUCT) person_attrs[1];

      System.out.println ("person address: ");
```

```
      Object address_attrs[] = address.getAttributes();

      System.out.println ("street:  " + (String) address_attrs[0]);
      System.out.println ("number:  " +
                          ((BigDecimal) address_attrs[1]).intValue());
      System.out.println ();
    }
  }
}
```

弱い型指定のオブジェクト参照 : StudentRef.java

このサンプルでは、SQL オブジェクト参照の弱い型指定をサポートする Oracle クラス oracle.sql.REF の機能を示します。この例では、SQL オブジェクト型 STUDENT が定義され、このオブジェクト型への参照が使用されます。

弱い型指定の REF クラスの機能の詳細は、第 9 章「Oracle オブジェクト参照の操作」を参照してください。

```
/*
 * This sample demonstrate basic Ref support
 */

import java.sql.*;
import java.io.*;
import java.util.*;
import java.math.BigDecimal;
import oracle.sql.*;
import oracle.jdbc.*;

public class StudentRef
{
  public static void main (String args [])
       throws Exception
  {
    // Register the Oracle JDBC driver
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    // Connect to the database
    // You need to put your database name after the @ sign in
    // the connection URL.
    //
    // The sample retrieves an object of type "person",
    // materializes the object as an object of type ADT.
    // The Object is then modified and inserted back into the database.
```

```
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:oci8:@",
                                   "scott", "tiger");

    // It's faster when auto commit is off
    conn.setAutoCommit (false);

    // Create a Statement
    Statement stmt = conn.createStatement ();

    try
    {
      stmt.execute ("drop table student_table");
      stmt.execute ("drop type STUDENT");
    }
    catch (SQLException e)
    {
      // the above drop and create statements will throw exceptions
      // if the types and tables did not exist before
    }

    stmt.execute ("create type STUDENT as object
                   (name VARCHAR (30), age NUMBER)");
    stmt.execute ("create table student_table of STUDENT");
    stmt.execute ("insert into student_table values ('John', 20)");

    ResultSet rs = stmt.executeQuery ("select ref (s) from student_table s");
    rs.next ();

    // retrieve the ref object
    REF ref = (REF) rs.getObject (1);

    //retrieve the object value that the ref points to in the
    // object table

    STRUCT student = (STRUCT) ref.getValue ();
    Object attributes[] = student.getAttributes();

    System.out.println ("student name: " + (String) attributes[0]);
    System.out.println ("student age:  " + ((BigDecimal)
                         attributes[1]).intValue());

    rs.close();
    stmt.close();
    conn.close();
  }
}
```

弱い型指定の配列 : ArrayExample.java

このサンプル・プログラムは、JDBC を使用して、VARRAY を含む表を作成します。このプログラムは、新しい配列オブジェクトを表に挿入してから、表の内容を出力します。配列の詳細は、第 10 章「Oracle コレクションの操作」を参照してください。

```
import java.sql.*;
import oracle.sql.*;
import oracle.jdbc.oracore.Util;
import oracle.jdbc.*;
import java.math.BigDecimal;


public class ArrayExample
{
  public static void main (String args[])
    throws Exception
  {
    // Register the Oracle JDBC driver
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    // Connect to the database
    // You need to put your database name after the @ sign in
    // the connection URL.
    //
    // The sample retrieves an varray of type "NUM_VARRAY",
    // materializes the object as an object of type ARRAY.
    // A new ARRAY is then inserted into the database.

    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:oci8:@",
                                   "scott", "tiger");

    // It's faster when auto commit is off
    conn.setAutoCommit (false);

    // Create a Statement
    Statement stmt = conn.createStatement ();
```

```
    try
    {
      stmt.execute ("DROP TABLE varray_table");
      stmt.execute ("DROP TYPE num_varray");
    }
    catch (SQLException e)
    {
      // the above drop statements will throw exceptions
      // if the types and tables did not exist before. Just ingore it.
    }

    stmt.execute ("CREATE TYPE num_varray AS VARRAY(10) OF NUMBER(12, 2)");
    stmt.execute ("CREATE TABLE varray_table (col1 num_varray)");
    stmt.execute ("INSERT INTO varray_table VALUES (num_varray(100, 200))");

    ResultSet rs = stmt.executeQuery("SELECT * FROM varray_table");
    showResultSet (rs);

    //now insert a new row

    // create a new ARRAY object
    int elements[] = { 300, 400, 500, 600 };
    ArrayDescriptor desc = ArrayDescriptor.createDescriptor("NUM_VARRAY", conn);
    ARRAY newArray = new ARRAY(desc, conn, elements);

    PreparedStatement ps =
      conn.prepareStatement ("insert into varray_table values (?)");
    ((OraclePreparedStatement)ps).setARRAY (1, newArray);

    ps.execute ();

    rs = stmt.executeQuery("SELECT * FROM varray_table");
    showResultSet (rs);

    // Close all the resources
    rs.close();
    ps.close();
    stmt.close();
    conn.close();

  }
```

```
  public static void showResultSet (ResultSet rs)
    throws SQLException
  {
    int line = 0;
    while (rs.next())
    {
      line++;
      System.out.println("Row "+line+" : ");
      ARRAY array = ((OracleResultSet)rs).getARRAY (1);

      System.out.println ("Array is of type "+array.getSQLTypeName());
      System.out.println
                ("Array element is of typecode "+array.getBaseType());
      System.out.println ("Array is of length "+array.length());

      // get Array elements
      BigDecimal[] values = (BigDecimal[]) array.getArray();

      for (int i=0; i<values.length; i++)
      {
        BigDecimal value = (BigDecimal) values[i];
        System.out.println(">> index "+i+" = "+value.intValue());
      }
    }
  }
}
```

Oracle 型拡張要素のサンプル

この項には、Oracle 型拡張要素のサンプル・コードが含まれています。

- REF CURSOR: RefCursorExample.java
- BFILE: FileExample.java

REF CURSOR のサンプルは、次のディレクトリにあります。

```
[Oracle Home]/jdbc/demo/samples/oci8/basic-samples
```

BFILE の例は、`object-samples` ディレクトリにあります。

REF CURSOR: RefCursorExample.java

このサンプル・プログラムでは、REF CURSOR 型を戻すストアド・ファンクションを含む PL/SQL パッケージを作成し、Oracle JDBC REF CURSOR 機能を示します。サンプルは、REF CURSOR を結果セット・オブジェクトに取り込みます。REF CURSOR の詳細は、5-33 ページの「Oracle REF CURSOR 型カテゴリ」を参照してください。

```
/*
 * This sample shows how to call a PL/SQL function that opens
 * a cursor and get the cursor back as a Java ResultSet.
 */

import java.sql.*;
import java.io.*;
import oracle.jdbc.*;

class RefCursorExample
{
  public static void main (String args [])
       throws SQLException
  {
    // Load the driver
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    // Connect to the database
    // You can put a database name after the @ sign in the connection URL.
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

    // Create the stored procedure
    init (conn);

    // Prepare a PL/SQL call
    CallableStatement call =
      conn.prepareCall ("{ ? = call java_refcursor.job_listing (?)}");

    // Find out all the SALES person
    call.registerOutParameter (1, OracleTypes.CURSOR);
    call.setString (2, "SALESMAN");
    call.execute ();
    ResultSet rset = (ResultSet)call.getObject (1);

    // Dump the cursor
    while (rset.next ())
      System.out.println (rset.getString ("ENAME"));
```

```
    // Close all the resources
    rset.close();
    call.close();
    conn.close();

  }

  // Utility function to create the stored procedure
  static void init (Connection conn)
       throws SQLException
  {
    Statement stmt = conn.createStatement ();

    stmt.execute ("create or replace package java_refcursor as " +
                  "  type myrctype is ref cursor return EMP%ROWTYPE; " +
                  "  function job_listing (j varchar2) return myrctype; " +
                  "end java_refcursor;");

    stmt.execute ("create or replace package body java_refcursor as " +
              "  function job_listing (j varchar2) return myrctype is " +
              "    rc myrctype; " +
              "  begin " +
              "    open rc for select * from emp where job = j; " +
              "    return rc; " +
              "  end; " +
              "end java_refcursor;");
    stmt.close();
  }
}
```

BFILE: FileExample.java

このサンプルでは、Oracle JDBC BFILE のサポートを示します。このサンプルは、表に BFILE を格納する方法を示します。また、BFILE の内容をダンプするためのユーティリティが含まれています。BFILE の詳細は、7-19 ページの「BFILE の操作」を参照してください。

```
/*
 * This sample demonstrate basic File support
 */

import java.sql.*;
import java.io.*;
import java.util.*;

//including this import makes the code easier to read
import oracle.jdbc.*;
```

```
// needed for new BFILE class
import oracle.sql.*;

public class FileExample
{
  public static void main (String args [])
       throws Exception
  {
    // Register the Oracle JDBC driver
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    // Connect to the database
    // You can put a database name after the @ sign in the connection URL.
    //
    // The sample creates a DIRECTORY and you have to be connected as
    // "system" to be able to run the test.
    // I you can't connect as "system" have your system manager
    // create the directory for you, grant you the rights to it, and
    // remove the portion of this program that drops and creates the directory.
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:oci8:@", "system", "manager");

    // It's faster when auto commit is off
    conn.setAutoCommit (false);

    // Create a Statement
    Statement stmt = conn.createStatement ();

    try
    {
      stmt.execute ("drop directory TEST_DIR");
    }
    catch (SQLException e)
    {
      // An error is raised if the directory does not exist.  Just ignore it.
    }
    stmt.execute ("create directory TEST_DIR as '/tmp/filetest'");

    try
    {
      stmt.execute ("drop table test_dir_table");
    }
    catch (SQLException e)
    {
      // An error is raised if the table does not exist.  Just ignore it.
    }
```

```
  // Create and populate a table with files
  // The files file1 and file2 must exist in the directory TEST_DIR created
  // above as symbolic name for /private/local/filetest.
  stmt.execute ("create table test_dir_table (x varchar2 (30), b bfile)");
  stmt.execute ("insert into test_dir_table values
                 ('one', bfilename ('TEST_DIR', 'file1'))");
  stmt.execute ("insert into test_dir_table values
                 ('two', bfilename ('TEST_DIR', 'file2'))");

  // Select the file from the table
  ResultSet rset = stmt.executeQuery ("select * from test_dir_table");
  while (rset.next ())
  {
    String x = rset.getString (1);
    BFILE bfile = ((OracleResultSet)rset).getBFILE (2);
    System.out.println (x + " " + bfile);

    // Dump the file contents
    dumpBfile (conn, bfile);
  }

  // Close all resources
  rset.close();
  stmt.close();
  conn.close();
}

// Utility function to dump the contents of a Bfile
static void dumpBfile (Connection conn, BFILE bfile)
  throws Exception
{
  System.out.println ("Dumping file " + bfile.getName());
  System.out.println ("File exists: " + bfile.fileExists());
  System.out.println ("File open: " + bfile.isFileOpen());

  System.out.println ("Opening File: ");

  bfile.openFile();

  System.out.println ("File open: " + bfile.isFileOpen());

  long length = bfile.length();
  System.out.println ("File length: " + length);

  int chunk = 10;
```

```
    InputStream instream = bfile.getBinaryStream();

    // Create temporary buffer for read
    byte[] buffer = new byte[chunk];

    // Fetch data
    while ((length = instream.read(buffer)) != -1)
    {
      System.out.print("Read " + length + " bytes: ");

      for (int i=0; i<length; i++)
        System.out.print(buffer[i]+" ");
      System.out.println();
    }

    // Close input stream
    instream.close();

    // close file handler
    bfile.closeFile();
  }
}
```

カスタム・オブジェクト・クラスのサンプル

この項では、SQL 構造化オブジェクトからマップするためのカスタム Java クラスの機能を示します。標準 SQLData 実装と Oracle ORAData 実装の両方の例が含まれています。

- SQLData 実装 : SQLDataExample.java
- ORAData 実装 : ORADataExample.java

この例には、Oracle オブジェクトのカスタム Java クラスを定義するために用意する必要があるコードの例と、これらのカスタム Java クラス定義を利用したサンプル・アプリケーションが含まれています。カスタム・クラスを作成するには、標準 java.sql.SQLData インタフェースまたは Oracle oracle.sql.CustomDatum インタフェースのいずれかを実装する必要があります。これらのインタフェースは、Oracle オブジェクトとその属性に対応したカスタム Java クラスを作成および移入するための方法を提供します。

SQLData および CustomDatum はどちらも SQL オブジェクトから Java オブジェクトを移入します。SQLData インタフェースは移植性が高く、CustomDatum インタフェースはデータを表現するのに便利で柔軟性が高くなっています。

SQLData インタフェースは、JDBC 標準です。このインタフェースの詳細は、8-15 ページの「SQLData インタフェース」を参照してください。

CustomDatum インタフェースは、Oracle により提供されます。CustomDatum インタフェースの詳細は、8-21 ページの「ORAData インタフェース」を参照してください。

独自のコードを作成し、どちらかのインタフェースを実装するカスタム Java クラスを作成することもできますが、Oracle JPublisher ユーティリティで、どちらかのインタフェースを実装するクラスを生成することもできます。

JPublisher の詳細は、8-44 ページの「JPublisher を使用したカスタム・オブジェクト・クラスの作成」および『Oracle9*i* JPublisher ユーザーズ・ガイド』を参照してください。

この項のサンプル・アプリケーションおよびカスタム Java クラス定義は、次のディレクトリにあります。

```
[Oracle Home]/jdbc/demo/samples/oci8/object-samples
```

SQLData 実装 : SQLDataExample.java

この項のコードでは、SQLData 実装を使用して、指定された SQL オブジェクト型に対応するカスタム Java 型を定義および使用する方法を示します。

SQL オブジェクト定義

次に、EMPLOYEE オブジェクトの SQL 定義を示します。このオブジェクトには、VARCHAR2 属性 EMPNAME（従業員名）と INTEGER 属性 EMPNO（従業員番号）の、2 つの属性があります。

```
 -- SQL definition
CREATE TYPE employee AS OBJECT
(
     empname VARCHAR2(50),
     empno   INTEGER
);
```

カスタム・オブジェクト・クラス : SQLData 実装

次のコードでは、SQL 型 EMPLOYEE に対応するカスタム Java クラス EmployeeObj が定義されます（EmployeeObj.java で定義されます）。EmployeeObj の定義には、文字列 empName（従業員名）属性および整数 empNo（従業員番号）属性が含まれることに注意してください。また、EmployeeObj カスタム Java クラスの Java 定義は、SQLData インタフェースを実装し、get メソッドおよび必要な readSQL() メソッドと writeSQL() メソッドの実装を含みます。

```
import java.sql.*;
import oracle.jdbc.*;

public class EmployeeObj implements SQLData
{
  private String sql_type;

  public String empName;
  public int empNo;

  public EmployeeObj()
  {
  }

  public EmployeeObj (String sql_type, String empName, int empNo)
  {
    this.sql_type = sql_type;
    this.empName = empName;
    this.empNo = empNo;
  }

  ////// implements SQLData //////

  public String getSQLTypeName() throws SQLException
  {
    return sql_type;
  }

  public void readSQL(SQLInput stream, String typeName)
    throws SQLException
  {
    sql_type = typeName;

    empName = stream.readString();
    empNo = stream.readInt();
  }
```

```
  public void writeSQL(SQLOutput stream)
    throws SQLException
  {
    stream.writeString(empName);
    stream.writeInt(empNo);
  }
}
```

SQLData カスタム・オブジェクト・クラスを使用するサンプル・アプリケーション

独自の EmployeeObj Java クラスの作成後に、プログラム中でそれを使用できます。次のプログラムは、従業員名および番号データを格納する表を作成します。このプログラムは、EmployeeObj オブジェクトを使用して新しい従業員オブジェクトを作成し、表に挿入します。次に SELECT 文を適用して表の内容を取り出し、その内容を出力します。

SQLData 実装を使用した SQL オブジェクト・データへのアクセスおよび操作の詳細は、8-17 ページの「SQLData 実装によるデータの読込みおよび書込み」を参照してください。

```
import java.sql.*;
import oracle.jdbc.*;
import oracle.sql.*;
import java.math.BigDecimal;
import java.util.Dictionary;


public class SQLDataExample
{

  public static void main(String args []) throws Exception
  {

    // Connect
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver ());
    OracleConnection conn = (OracleConnection)
      DriverManager.getConnection("jdbc:oracle:oci8:@",
                                  "scott", "tiger");

    Dictionary map = (Dictionary)conn.getTypeMap();
    map.put("EMPLOYEE", Class.forName("EmployeeObj"));
```

```
// Create a Statement
Statement stmt = conn.createStatement ();
try
{
  stmt.execute ("drop table EMPLOYEE_TABLE");
  stmt.execute ("drop type EMPLOYEE");
}
catch (SQLException e)
{
  // An error is raised if the table/type does not exist. Just ignore it.
}

// Create and populate tables
stmt.execute ("CREATE TYPE EMPLOYEE AS OBJECT
               (EmpName VARCHAR2(50),EmpNo INTEGER)");
stmt.execute ("CREATE TABLE EMPLOYEE_TABLE (ATTR1 EMPLOYEE)");
stmt.execute ("INSERT INTO EMPLOYEE_TABLE VALUES
               (EMPLOYEE('Susan Smith', 123))");
stmt.close();

// Create a SQLData object
EmployeeObj e = new EmployeeObj("SCOTT.EMPLOYEE", "George Jones", 456);

// Insert the SQLData object
PreparedStatement pstmt
  = conn.prepareStatement ("insert into employee_table values (?)");

pstmt.setObject(1, e, OracleTypes.STRUCT);
pstmt.executeQuery();
System.out.println("insert done");
pstmt.close();

// Select now
Statement s = conn.createStatement();
OracleResultSet rs = (OracleResultSet)
  s.executeQuery("select * from employee_table");

while(rs.next())
{
   EmployeeObj ee = (EmployeeObj) rs.getObject(1);
   System.out.println("EmpName: " + ee.empName + " EmpNo: " + ee.empNo);
}
rs.close();
s.close();
```

```
    if (conn != null)
    {
      conn.close();
    }
  }
}
```

ORAData 実装 : ORADataExample.java

この項のコードでは、ORAData 実装を使用して、指定された SQL オブジェクト型に対応するカスタム Java 型を定義および使用する方法を示します。

SQL オブジェクト定義

次に、EMPLOYEE オブジェクトの SQL 定義を示します。このオブジェクトには、VARCHAR2 属性 EMPNAME（従業員名）と INTEGER 属性 EMPNO（従業員番号）の、2 つの属性があります。

```
CREATE TYPE employee AS OBJECT
(
     empname VARCHAR2(50),
     empno   INTEGER
);
```

カスタム・オブジェクト・クラス : CustomDatum 実装

次のコードでは、SQL 型 EMPLOYEE に対応するカスタム Java クラス Employee が定義されます（Employee.java で定義されます）。Employee の定義には、文字列 empname（従業員名）および整数 empno（従業員番号）のアクセッサ・メソッドが含まれることに注意してください。また、Employee カスタム Java クラスの Java 定義で、ORAData および ORADataFactory インタフェースが実装されることにも注意してください。ORAData を実装するカスタム Java クラスには、ORADataFactory オブジェクトを戻す静的な getFactory() メソッドがあります。JDBC ドライバは、ORADataFactory オブジェクトの create() メソッドを使用し、CustomDatum インスタンスを戻します。

カスタム Java クラスを独自に記述するかわりに、JPublisher ユーティリティを使用して、ORAData および ORADataFactory インタフェースを実装するクラス定義を生成します。実際に JPublisher で生成された Employee.java のコードを示します。

```
import java.sql.SQLException;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.OracleTypes;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;

public class Employee implements CustomDatum, CustomDatumFactory
{
  public static final String _SQL_NAME = "SCOTT.EMPLOYEE";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

  MutableStruct _struct;

  static int[] _sqlType =
  {
    12, 4
  };

  static CustomDatumFactory[] _factory = new CustomDatumFactory[2];

  static final Employee _EmployeeFactory = new Employee();
  public static CustomDatumFactory getFactory()
  {
    return _EmployeeFactory;
  }

  /* constructor */
  public Employee()
  {
    _struct = new MutableStruct(new Object[2], _sqlType, _factory);
  }

  /* CustomDatum interface */
  public Datum toDatum(OracleConnection c) throws SQLException
  {
    return _struct.toDatum(c, _SQL_NAME);
  }
```

```
  /* CustomDatumFactory interface */
  public CustomDatum create(Datum d, int sqlType) throws SQLException
  {
    if (d == null) return null;
    Employee o = new Employee();
    o._struct = new MutableStruct((STRUCT) d, _sqlType, _factory);
    return o;
  }

  /* accessor methods */
  public String getEmpname() throws SQLException
  { return (String) _struct.getAttribute(0); }

  public void setEmpname(String empname) throws SQLException
  { _struct.setAttribute(0, empname); }

  public Integer getEmpno() throws SQLException
  { return (Integer) _struct.getAttribute(1); }

  public void setEmpno(Integer empno) throws SQLException
  { _struct.setAttribute(1, empno); }
}
```

CustomDatum カスタム・オブジェクト・クラスを使用するサンプル・アプリケーション

このサンプル・プログラムでは、JPublisher で生成された Employee クラスの使用方法を示します。サンプル・コードは、新しい Employee オブジェクトを作成してデータを格納してから、データベースに挿入します。次に、データベースから Employee データを取り出します。

CustomDatum 実装を使用した SQL オブジェクト・データへのアクセスおよび操作の詳細は、8-23 ページの「ORAData 実装によるデータの読込みおよび書込み」を参照してください。

```
import java.sql.*;
import oracle.jdbc.*;
import oracle.sql.*;
import java.math.BigDecimal;


public class CustomDatumExample
{

  public static void main(String args []) throws Exception
  {
```

```
// Connect
DriverManager.registerDriver(new oracle.jdbc.OracleDriver ());
OracleConnection conn = (OracleConnection)
  DriverManager.getConnection("jdbc:oracle:oci8:@",
                              "scott", "tiger");

// Create a Statement
Statement stmt = conn.createStatement ();
try
{
  stmt.execute ("drop table EMPLOYEE_TABLE");
  stmt.execute ("drop type EMPLOYEE");
}
catch (SQLException e)
{
  // An error is raised if the table/type does not exist. Just ignore it.
}

// Create and populate tables
stmt.execute ("CREATE TYPE EMPLOYEE AS OBJECT
               (EmpName VARCHAR2(50),EmpNo INTEGER)");
stmt.execute ("CREATE TABLE EMPLOYEE_TABLE (ATTR1 EMPLOYEE)");
stmt.execute ("INSERT INTO EMPLOYEE_TABLE VALUES
               (EMPLOYEE('Susan Smith', 123))");
stmt.close();

// Create a CustomDatum object
Employee e = new Employee("George Jones", new BigDecimal("456"));

// Insert the CustomDatum object
PreparedStatement pstmt
  = conn.prepareStatement ("insert into employee_table values (?)");

pstmt.setObject(1, e, OracleTypes.STRUCT);
pstmt.executeQuery();
System.out.println("insert done");
pstmt.close();

// Select now
Statement s = conn.createStatement();
OracleResultSet rs = (OracleResultSet)
  s.executeQuery("select * from employee_table");
```

```
    while(rs.next())
    {
       Employee ee = (Employee) rs.getCustomDatum(1, Employee.getFactory());
       System.out.println("EmpName: " + ee.empName + " EmpNo: " + ee.empNo);
    }
    rs.close();
    s.close();

    if (conn != null)
    {
      conn.close();
    }
  }
}
```

JDBC 2.0 結果セット拡張のサンプル

この項のサンプルでは、JDBC 2.0 で使用可能な結果セット拡張の機能を示します。この機能には、スクロール可能な結果セットでの位置指定、結果セットの更新、外部更新を自動的に参照できる scroll-sensitive 結果セットの使用および結果セットへの明示的なデータの再フェッチが含まれます。

- 結果セットでの位置の指定 : ResultSet2.java
- 結果セットでの行の挿入および削除 : ResultSet3.java
- 結果セットでの行の更新 : ResultSet4.java
- Scroll-Sensitive 結果セット : ResultSet5.java
- 結果セットでの行の再フェッチ : ResultSet6.java

この項のサンプル・アプリケーションは、次のディレクトリにあります。

```
[Oracle Home]/jdbc/demo/samples/oci8/jdbc20-samples
```

結果セットでの位置の指定 : ResultSet2.java

この項では、スクロール可能な結果セットの機能を示します。相対行位置および絶対行位置への移動および結果セットの後方への反復を行います。

この項目の詳細は、11-11 ページの「スクロール可能な結果セットの位置指定および処理」を参照してください。

```
/**
 * A simple sample to demonstrate previous(), absolute() and relative().
 */

import java.sql.*;

public class ResultSet2
{
  public static void main(String[] args) throws SQLException
  {
    // Load the Oracle JDBC driver
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    // Connect to the database
    // You can put a database name after the @ sign in the connection URL.
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

    // Create a Statement
    Statement stmt = conn.createStatement (ResultSet.TYPE_SCROLL_INSENSITIVE,
                                           ResultSet.CONCUR_UPDATABLE);

    // Query the EMP table
    ResultSet rset = stmt.executeQuery ("select ENAME from EMP");

    // iterate through the result using next()
    show_resultset_by_next(rset);

    // iterate through the result using previous()
    show_resultset_by_previous(rset);

    // iterate through the result using absolute()
    show_resultset_by_absolute(rset);

    // iterate through the result using relative()
    show_resultset_by_relative(rset);

    // Close the ResultSet
    rset.close();
```

```
  // Close the Statement
  stmt.close();

  // Close the connection
  conn.close();
}

/**
 * Iterate through the result using next().
 *
 * @param rset a result set object
 */
public static void show_resultset_by_next(ResultSet rset)
  throws SQLException
{
  System.out.println ("List the employee names using ResultSet.next():");

  // Make sure the cursor is placed right before the first row
  if (!rset.isBeforeFirst())
  {
    // Place the cursor right before the first row
    rset.beforeFirst ();
  }

  // Iterate through the rows using next()
  while (rset.next())
    System.out.println (rset.getString (1));

  System.out.println ();
}

/**
 * Iterate through the result using previous().
 *
 * @param rset a result set object
 */
public static void show_resultset_by_previous(ResultSet rset)
  throws SQLException
{
  System.out.println ("List the employee names using ResultSet.previous():");

  // Make sure the cursor is placed after the last row
  if (!rset.isAfterLast())
  {
    // Place the cursor after the last row
    rset.afterLast ();
  }
```

```
  // Iterate through the rows using previous()
  while (rset.previous())
    System.out.println (rset.getString (1));

  System.out.println ();
}

/**
 * Iterate through the result using absolute().
 *
 * @param rset a result set object
 */
public static void show_resultset_by_absolute (ResultSet rset)
  throws SQLException
{
  System.out.println ("List the employee names using ResultSet.absolute():");

  // The begin index for ResultSet.absolute (idx)
  int idx = 1;

  // Loop through the result set until absolute() returns false.
  while (rset.absolute(idx))
  {
    System.out.println (rset.getString (1));
    idx ++;
  }
  System.out.println ();
}

/**
 * Iterate through the result using relative().
 *
 * @param rset a result set object
 */
public static void show_resultset_by_relative (ResultSet rset)
  throws SQLException
{
  System.out.println ("List the employee names using ResultSet.relative():");

  // getRow() returns 0 if there is no current row
  if (rset.getRow () == 0 || !rset.isLast())
  {
    // place the cursor on the last row
    rset.last ();
  }
```

```
    // Calling relative(-1) is similar to previous(), but the cursor
    // has to be on a valid row before calling relative().
    do
    {
      System.out.println (rset.getString (1));
    }
    while (rset.relative (-1));

    System.out.println ();
  }
}
```

結果セットでの行の挿入および削除 : ResultSet3.java

このサンプルでは、更新可能な結果セットの機能をいくつか示します。行を挿入および削除し、次に、その行をデータベースに挿入またはデータベースから削除します。

この項目の詳細は、11-18 ページの「結果セットでの INSERT 操作実行」および 11-15 ページの「結果セットでの DELETE 操作実行」を参照してください。

```
/**
 * A simple sample to to demonstrate ResultSet.insertRow() and
 * ResultSet.deleteRow().
 */

import java.sql.*;

public class ResultSet3
{
  public static void main(String[] args) throws SQLException
  {
    // Load the Oracle JDBC driver
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    // Connect to the database
    // You can put a database name after the @ sign in the connection URL.
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

    // Cleanup
    cleanup (conn);

    // Create a Statement
    Statement stmt = conn.createStatement (ResultSet.TYPE_SCROLL_INSENSITIVE,
                                           ResultSet.CONCUR_UPDATABLE);
```

```
// Query the EMP table
ResultSet rset = stmt.executeQuery ("select EMPNO, ENAME from EMP");

// Add three new employees using ResultSet.insertRow()
addEmployee (rset, 1001, "PETER");
addEmployee (rset, 1002, "MARY");
addEmployee (rset, 1003, "DAVID");

// Close the result set
rset.close ();

// Verify the insertion
System.out.println ("\nList EMPNO and ENAME in the EMP table: ");

rset = stmt.executeQuery ("select EMPNO, ENAME from EMP");
while (rset.next())
{
  // We expect to see the three new employees
  System.out.println (rset.getInt(1)+" "+rset.getString(2));
}
System.out.println ();

// Delete the new employee 'PETER' using ResultSet.deleteRow()
removeEmployee (rset, 1001);
rset.close ();

// Verify the deletion
System.out.println ("\nList EMPNO and ENAME in the EMP table: ");
rset = stmt.executeQuery ("select EMPNO, ENAME from EMP");
while (rset.next())
{
  // We expect "PETER" is removed
  System.out.println (rset.getInt(1)+" "+rset.getString(2));
}
System.out.println ();

// Close the RseultSet
rset.close();

// Close the Statement
stmt.close();

// Cleanup
cleanup(conn);
```

```
  // Close the connection
  conn.close();
}

/**
 * Add a new employee to EMP table.
 */
public static void addEmployee (ResultSet rset,
                                int employeeId,
                                String employeeName)
  throws SQLException
{
  System.out.println ("Adding new employee: "+employeeId+" "+employeeName);

  // Place the cursor on the insert row
  rset.moveToInsertRow();

  // Assign the new values
  rset.updateInt (1, employeeId);
  rset.updateString (2, employeeName);

  // Insert the new row to database
  rset.insertRow();
}

/**
 * Remove the employee from EMP table.
 */
public static void removeEmployee (ResultSet rset,
                                   int employeeId)
  throws SQLException
{
  System.out.println ("Removing the employee: id="+employeeId);

  // Place the cursor right before the first row if it doesn't
  if (!rset.isBeforeFirst())
  {
    rset.beforeFirst();
  }
```

```
    // Iterate the result set
    while (rset.next())
    {
      // Place the cursor the row with matched employee id
      if (rset.getInt(1) == employeeId)
      {
        // Delete the current row
        rset.deleteRow();
        break;
      }
    }
  }

  /**
   * Generic cleanup.
   */
  public static void cleanup (Connection conn)
    throws SQLException
  {
    Statement stmt = conn.createStatement ();
    stmt.execute
         ("DELETE FROM EMP WHERE EMPNO=1001 OR EMPNO=1002 OR EMPNO=1003");
    stmt.execute ("COMMIT");
    stmt.close ();
  }
}
```

結果セットでの行の更新 : ResultSet4.java

このサンプルでは、更新可能な結果セットの機能をいくつか示します。行を更新し、次に、データベースを更新します。

この項目の詳細は、11-16 ページの「結果セットでの UPDATE 操作実行」を参照してください。

```
/**
 * A simple sample to demonstrate ResultSet.udpateRow().
 */

import java.sql.*;
```

```
public class ResultSet4
{
  public static void main(String[] args) throws SQLException
  {
    // Load the Oracle JDBC driver
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    // Connect to the database
    // You can put a database name after the @ sign in the connection URL.
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

    // Create a Statement
    Statement stmt = conn.createStatement (ResultSet.TYPE_SCROLL_INSENSITIVE,
                                           ResultSet.CONCUR_UPDATABLE);

    // Query the EMP table
    ResultSet rset = stmt.executeQuery ("select EMPNO, ENAME, SAL from EMP");

    // Give everybody a $500 raise
    adjustSalary (rset, 500);

    // Verify the sarlary changes
    System.out.println ("Verify the changes with a new query: ");
    rset = stmt.executeQuery ("select EMPNO, ENAME, SAL from EMP");
    while (rset.next())
    {
      System.out.println (rset.getInt(1)+" "+rset.getString(2)+" "+
                          rset.getInt(3));
    }
    System.out.println ();

    // Close the RseultSet
    rset.close();

    // Close the Statement
    stmt.close();

    // Cleanup
    cleanup(conn);

    // Close the connection
    conn.close();
  }
```

```
  /**
   * Update the ResultSet content using updateRow().
   */
  public static void adjustSalary (ResultSet rset, int raise)
    throws SQLException
  {
    System.out.println ("Give everybody in the EMP table a $500 raise\n");

    int salary = 0;

    while (rset.next ())
    {
      // save the old value
      salary = rset.getInt (3);

      // update the row
      rset.updateInt (3, salary + raise);

      // flush the changes to database
      rset.updateRow ();

      // show the changes
      System.out.println (rset.getInt(1)+" "+rset.getString(2)+" "+
                          salary+" -> "+rset.getInt(3));
    }
    System.out.println ();
  }

  /**
   * Generic cleanup.
   */
  public static void cleanup (Connection conn) throws SQLException
  {
    Statement stmt = conn.createStatement ();
    stmt.execute ("UPDATE EMP SET SAL = SAL - 500");
    stmt.execute ("COMMIT");
    stmt.close ();
  }
}
```

Scroll-Sensitive 結果セット : ResultSet5.java

このサンプルでは、scroll-sensitive 結果の機能を示します。この結果セットでは、外部的に加えられたデータベースへの変更を、暗黙的に参照できます。

scroll-sensitive 結果セットとその実装方法の詳細は、11-27 ページの「Scroll-Sensitive 結果セットの Oracle 実装」を参照してください。

```
/**
 * A simple sample to demonstrate scroll sensitive result set.
 */

import java.sql.*;

public class ResultSet5
{
  public static void main(String[] args) throws SQLException
  {
    // Load the Oracle JDBC driver
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    // Connect to the database
    // You can put a database name after the @ sign in the connection URL.
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

    // Create a Statement
    Statement stmt = conn.createStatement (ResultSet.TYPE_SCROLL_SENSITIVE,
                                           ResultSet.CONCUR_UPDATABLE);

    // Set the statement fetch size to 1
    stmt.setFetchSize (1);

    // Query the EMP table
    ResultSet rset = stmt.executeQuery ("select EMPNO, ENAME, SAL from EMP");

    // List the result set's type, concurrency type, ..., etc
    showProperty (rset);

    // List the query result
    System.out.println ("List ENO, ENAME and SAL from the EMP table: ");
    while (rset.next())
    {
      System.out.println (rset.getInt(1)+" "+rset.getString(2)+" "+
                          rset.getInt(3));
    }
    System.out.println ();
```

```
    // Do some changes outside the result set
    doSomeChanges (conn);

    // Place the cursor right before the first row
    rset.beforeFirst ();

    // List the employee information again
    System.out.println ("List ENO, ENAME and SAL again: ");
    while (rset.next())
    {
      // We expect to see the changes made in "doSomeChanges()"
      System.out.println (rset.getInt(1)+" "+rset.getString(2)+" "+
                          rset.getInt(3));
    }

    // Close the RseultSet
    rset.close();

    // Close the Statement
    stmt.close();

    // Cleanup
    cleanup(conn);

    // Close the connection
    conn.close();
  }

  /**
   * Update the EMP table.
   */
  public static void doSomeChanges (Connection conn)
    throws SQLException
  {
    System.out.println ("Update the employee salary outside the result set\n");

    Statement otherStmt = conn.createStatement ();
    otherStmt.execute ("update emp set sal = sal + 500");
    otherStmt.execute ("commit");
    otherStmt.close ();
  }
```

```
/**
 * Show the result set properties like type, concurrency type, fetch
 * size,..., etc.
 */
public static void showProperty (ResultSet rset) throws SQLException
{
  // Verify the result set type
  switch (rset.getType())
  {
    case ResultSet.TYPE_FORWARD_ONLY:
      System.out.println ("Result set type: TYPE_FORWARD_ONLY");
      break;
    case ResultSet.TYPE_SCROLL_INSENSITIVE:
      System.out.println ("Result set type: TYPE_SCROLL_INSENSITIVE");
      break;
    case ResultSet.TYPE_SCROLL_SENSITIVE:
      System.out.println ("Result set type: TYPE_SCROLL_SENSITIVE");
      break;
    default:
      System.out.println ("Invalid type");
      break;
  }

  // Verify the result set concurrency
  switch (rset.getConcurrency())
  {
    case ResultSet.CONCUR_UPDATABLE:
      System.out.println
                ("Result set concurrency: ResultSet.CONCUR_UPDATABLE");
      break;
    case ResultSet.CONCUR_READ_ONLY:
      System.out.println
                ("Result set concurrency: ResultSet.CONCUR_READ_ONLY");
      break;
    default:
      System.out.println ("Invalid type");
      break;
  }

  // Verify the fetch size
  System.out.println ("fetch size: "+rset.getFetchSize ());

  System.out.println ();
}
```

```
  /**
   * Generic cleanup.
   */
  public static void cleanup (Connection conn) throws SQLException
  {
    Statement stmt = conn.createStatement ();
    stmt.execute ("UPDATE EMP SET SAL = SAL - 500");
    stmt.execute ("COMMIT");
    stmt.close ();
  }
}
```

結果セットでの行の再フェッチ : ResultSet6.java

このサンプルでは、データベースからデータを明示的に再フェッチして、結果セットを更新する方法を示します。この機能は、scroll-sensitive な結果セットおよび scroll-insensitive で更新可能な結果セットで使用可能です。

詳細は、11-22 ページの「行の再フェッチ」を参照してください。

```
/**
 * A simple sample to demonstrate ResultSet.refreshRow().
 */

import java.sql.*;

public class ResultSet6
{
  public static void main(String[] args) throws SQLException
  {
    // Load the Oracle JDBC driver
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    // Connect to the database
    // You can put a database name after the @ sign in the connection URL.
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

    // Create a Statement
    Statement stmt = conn.createStatement (ResultSet.TYPE_SCROLL_INSENSITIVE,
                                           ResultSet.CONCUR_UPDATABLE);
```

```
// Set the statement fetch size to 1
stmt.setFetchSize (1);

// Query the EMP table
ResultSet rset = stmt.executeQuery ("select EMPNO, ENAME, SAL from EMP");

// List the result set's type, concurrency type, ..., etc
showProperty (rset);

// List the query result
System.out.println ("List ENO, ENAME and SAL from the EMP table: ");
while (rset.next())
{
  System.out.println (rset.getInt(1)+" "+rset.getString(2)+" "+
                      rset.getInt(3));
}
System.out.println ();

// Do some changes outside the result set
doSomeChanges (conn);

// Place the cursor right before the first row
rset.beforeFirst ();

// List the employee information again
System.out.println ("List ENO, ENAME and SAL again: ");
int salary = 0;
while (rset.next())
{
  // save the original salary
  salary = rset.getInt (3);

  // refresh the row
  rset.refreshRow ();

  // We expect to see the changes made in "doSomeChanges()"
  System.out.println (rset.getInt(1)+" "+rset.getString(2)+" "+
                      salary+" -> "+rset.getInt(3));
}

// Close the RseultSet
rset.close();

// Close the Statement
stmt.close();
```

```
  // Cleanup
  cleanup(conn);

  // Close the connection
  conn.close();
}

/**
 * Update the EMP table.
 */
public static void doSomeChanges (Connection conn)
  throws SQLException
{
  System.out.println ("Update the employee salary outside the result set\n");

  Statement otherStmt = conn.createStatement ();
  otherStmt.execute ("update emp set sal = sal + 500");
  otherStmt.execute ("commit");
  otherStmt.close ();
}

/**
 * Show the result set properties like type, concurrency type, fetch
 * size,..., etc.
 */
public static void showProperty (ResultSet rset) throws SQLException
{
  // Verify the result set type
  switch (rset.getType())
  {
    case ResultSet.TYPE_FORWARD_ONLY:
      System.out.println ("Result set type: TYPE_FORWARD_ONLY");
      break;
    case ResultSet.TYPE_SCROLL_INSENSITIVE:
      System.out.println ("Result set type: TYPE_SCROLL_INSENSITIVE");
      break;
    case ResultSet.TYPE_SCROLL_SENSITIVE:
      System.out.println ("Result set type: TYPE_SCROLL_SENSITIVE");
      break;
    default:
      System.out.println ("Invalid type");
      break;
  }
```

```
    // Verify the result set concurrency
    switch (rset.getConcurrency())
    {
      case ResultSet.CONCUR_UPDATABLE:
        System.out.println
                    ("Result set concurrency: ResultSet.CONCUR_UPDATABLE");
        break;
      case ResultSet.CONCUR_READ_ONLY:
        System.out.println
                    ("Result set concurrency: ResultSet.CONCUR_READ_ONLY");
        break;
      default:
        System.out.println ("Invalid type");
        break;
    }

    // Verify the fetch size
    System.out.println ("fetch size: "+rset.getFetchSize ());

    System.out.println ();
  }

  /**
   * Generic cleanup.
   */
  public static void cleanup (Connection conn) throws SQLException
  {
    Statement stmt = conn.createStatement ();
    stmt.execute ("UPDATE EMP SET SAL = SAL - 500");
    stmt.execute ("COMMIT");
    stmt.close ();
  }
}
```

パフォーマンス強化のサンプル

この項には、バッチ更新など、パフォーマンス強化機能のサンプル・アプリケーションが含まれています。

- 標準バッチ更新 : BatchUpdates.java
- 暗黙的に実行される Oracle バッチ更新 : SetExecuteBatch.java
- 明示的に実行される Oracle バッチ更新 : SendBatch.java
- 接続で指定する Oracle 行プリフェッチ : RowPrefetch_connection.java
- 文で指定する Oracle 行プリフェッチ : RowPrefetch_statement.java
- Oracle 列型定義 : DefineColumnType.java

Oracle 固有のパフォーマンス強化のサンプル・アプリケーションは、次のディレクトリにあります。

```
[Oracle Home]/jdbc/demo/samples/oci8/basic-samples
```

標準バッチ更新のサンプルは、`jdbc20-samples` ディレクトリにあります。

標準バッチ更新 : BatchUpdates.java

このサンプルで、JDBC 2.0 で指定されている標準バッチ更新の使用方法を示します。詳細は、12-10 ページの「標準バッチ更新」を参照してください。

標準バッチ更新モデルと Oracle 固有バッチ更新モデルの比較については、12-2 ページの「バッチ更新モデルの概要」を参照してください。

```
/**
 * A simple sample to demonstrate standard JDBC 2.0 update batching.
 */

import java.sql.*;

public class BatchUpdates
{
  public static void main(String[] args)
  {
    Connection          conn = null;
    Statement           stmt = null;
    PreparedStatement   pstmt = null;
    ResultSet           rset = null;
    int                 i = 0;
```

```
try
{
  DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

  conn = DriverManager.getConnection(
           "jdbc:oracle:oci8:@", "scott", "tiger");

  stmt = conn.createStatement();
  try { stmt.execute(
        "create table mytest_table (col1 number, col2 varchar2(20))");
  } catch (Exception e1) {}

  //
  // Insert in a batch.
  //
  pstmt = conn.prepareStatement("insert into mytest_table values (?, ?)");

  pstmt.setInt(1, 1);
  pstmt.setString(2, "row 1");
  pstmt.addBatch();

  pstmt.setInt(1, 2);
  pstmt.setString(2, "row 2");
  pstmt.addBatch();

  pstmt.executeBatch();

  //
  // Select and print results.
  //
  rset = stmt.executeQuery("select * from mytest_table");
  while (rset.next())
  {
    System.out.println(rset.getInt(1) + ", " + rset.getString(2));
  }
}
catch (Exception e)
{
  e.printStackTrace();
}
finally
{
  if (stmt != null)
  {
    try { stmt.execute("drop table mytest_table"); } catch (Exception e) {}
    try { stmt.close(); } catch (Exception e) {}
  }
```

```
      if (pstmt != null)
      {
        try { pstmt.close(); } catch (Exception e) {}
      }
      if (conn != null)
      {
        try { conn.close(); } catch (Exception e) {}
      }
    }
  }
}
```

暗黙的に実行される Oracle バッチ更新 : SetExecuteBatch.java

このサンプルでは、Oracle バッチ更新の使用方法を示します。バッチは、バッチ値（データベースに送信するまでに収集する文の数）に達したときに暗黙的に実行されます。

Oracle バッチ更新の詳細は、12-4 ページの「Oracle バッチ更新」を参照してください。

標準バッチ更新モデルと Oracle 固有バッチ更新モデルの比較については、12-2 ページの「バッチ更新モデルの概要」を参照してください。

```
/*
 * This sample shows how to use the batching extensions.
 * In this example, we set the defaultBatch value from the
 * connection object. This affects all statements created from
 * this connection.
 * It is possible to set the batch value individually for each
 * statement. The API to use on the statement object is setExecuteBatch().
 *
 */

// You need to import the java.sql package to use JDBC
import java.sql.*;

// You need to import oracle.jdbc.* in order to use the
// API extensions.
import oracle.jdbc.*;

class SetExecuteBatch
{
  public static void main (String args [])
       throws SQLException
  {
    // Load the Oracle JDBC driver
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
```

```
    // Connect to the database
    // You can put a database name after the @ sign in the connection URL.
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

    // Default batch value set to 2 for all prepared statements belonging
    // to this connection.
    ((OracleConnection)conn).setDefaultExecuteBatch (2);

    PreparedStatement ps =
      conn.prepareStatement ("insert into dept values (?, ?, ?)");

    ps.setInt (1, 12);
    ps.setString (2, "Oracle");
    ps.setString (3, "USA");

    // No data is sent to the database by this call to executeUpdate
    System.out.println ("Number of rows updated so far: "
                        + ps.executeUpdate ());

    ps.setInt (1, 11);
    ps.setString (2, "Applications");
    ps.setString (3, "Indonesia");

    // The number of batch calls to executeUpdate is now equal to the
    // batch value of 2.  The data is now sent to the database and
    // both rows are inserted in a single roundtrip.
    int rows = ps.executeUpdate ();
    System.out.println ("Number of rows updated now: " + rows);

    ps.close ();
    conn.close();
  }
}
```

明示的に実行される Oracle バッチ更新 : SendBatch.java

このサンプルでは、Oracle バッチ更新の使用方法を示します。バッチは、sendBatch() コールで明示的に実行されます。

Oracle バッチ更新の詳細は、12-4 ページの「Oracle バッチ更新」を参照してください。

標準バッチ更新モデルと Oracle 固有バッチ更新モデルの比較については、12-2 ページの「バッチ更新モデルの概要」を参照してください。

```
/*
 * This sample shows how to use the batching extensions.
 * In this example, we demonstrate the use of the "sendBatch" API.
 * This allows the user to actually execute a set of batched
 * execute commands.
 *
 */

// You need to import the java.sql package to use JDBC
import java.sql.*;

// You need to import oracle.jdbc.* in order to use the
// API extensions.
import oracle.jdbc.*;

class SendBatch
{
  public static void main (String args [])
       throws SQLException
  {
    // Load the Oracle JDBC driver
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    // Connect to the database
    // You can put a database name after the @ sign in the connection URL.
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

    Statement stmt = conn.createStatement ();

    // Default batch value set to 50 for all prepared statements belonging
    // to this connection.
    ((OracleConnection)conn).setDefaultExecuteBatch (50);

    PreparedStatement ps =
      conn.prepareStatement ("insert into dept values (?, ?, ?)");
```

```
    ps.setInt (1, 32);
    ps.setString (2, "Oracle");
    ps.setString (3, "USA");

    // this execute does not actually happen at this point
    System.out.println (ps.executeUpdate ());

    ps.setInt (1, 33);
    ps.setString (2, "Applications");
    ps.setString (3, "Indonesia");

    // this execute does not actually happen at this point
    int rows = ps.executeUpdate ();

    System.out.println ("Number of rows updated before calling sendBatch: "
                                        + rows);

    // Execution of both previously batched executes will happen
    // at this point. The number of rows updated will be
    // returned by sendBatch.
    rows = ((OraclePreparedStatement)ps).sendBatch ();

    System.out.println ("Number of rows updated by calling sendBatch: "
                                          + rows);

    ps.close ();
    conn.close ();
  }
}
```

接続で指定する Oracle 行プリフェッチ : RowPrefetch_connection.java

この項では、Oracle 行プリフェッチ機能の使用方法を示します。接続オブジェクトに行プリフェッチ値を設定し、その接続から作成されるすべての文に影響を与えます。

Oracle 行プリフェッチは、基本的には、JDBC 2.0 フェッチ・サイズ機能に似ています。

Oracle 行プリフェッチの詳細は、12-19 ページの「Oracle 行プリフェッチ」を参照してください。JDBC 2.0 フェッチ・サイズの詳細およびフェッチ・サイズと行プリフェッチの比較については、11-20 ページの「フェッチ・サイズ」を参照してください。

```
/*
 * This sample shows how to use the Oracle performance extensions
 * for row-prefetching. This allows the driver to fetch multiple
 * rows in one round-trip, saving unecessary round-trips to the database.
 *
 * This example shows how to set the rowPrefetch for the connection object,
 * which will be used for all statements created from this connection.
 * Please see RowPrefetch_statement.java for examples of how to set
 * the rowPrefetch for statements individually.
 *
 */

// You need to import the java.sql package to use JDBC
import java.sql.*;

// You need to import oacle.jdbc.driver in order to use the oracle extensions.
import oracle.jdbc.*;

class RowPrefetch_connection
{
  public static void main (String args [])
       throws SQLException
  {
    // Load the Oracle JDBC driver
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    // Connect to the database
    // You can put a database name after the @ sign in the connection URL.
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

    // set the RowPrefetch value from the Connection object
    // This sets the rowPrefetch for *all* statements belonging
    // to this connection.
    // The rowPrefetch value can be overriden for specific statements by
    // using the setRowPrefetch API on the statement object. Please look
    // at RowPrefetch_statement.java for an example.


    // Please note that any statements created *before* the connection
    // rowPrefetch was set, will use the default rowPrefetch.

    ((OracleConnection)conn).setDefaultRowPrefetch (30);

    Statement stmt = conn.createStatement ();
```

```
    // Check to verify statement rowPrefetch value is 30.
    int row_prefetch = ((OracleStatement)stmt).getRowPrefetch ();
    System.out.println ("The RowPrefetch for the statement is:  "
                                        + row_prefetch + "\n");

    ResultSet rset = stmt.executeQuery ("select ename from emp");

    while(rset.next ())
    {
      System.out.println (rset.getString (1));
    }
    rset.close ();
    stmt.close ();
    conn.close ();
  }
}
```

文で指定する Oracle 行プリフェッチ : RowPrefetch_statement.java

この項では、Oracle 行プリフェッチ機能の使用方法を示します。特定の文オブジェクトに行プリフェッチ値を設定し、文を作成した接続オブジェクトの値をオーバーライドします。

Oracle 行プリフェッチは、基本的には、JDBC 2.0 フェッチ・サイズ機能に似ています。

Oracle 行プリフェッチの詳細は、12-19 ページの「Oracle 行プリフェッチ」を参照してください。JDBC 2.0 フェッチ・サイズの詳細と行プリフェッチの比較については、11-20 ページの「フェッチ・サイズ」を参照してください。

```
/*
 * This sample shows how to use the Oracle performance extensions
 * for row-prefetching. This allows the driver to fetch multiple
 * rows in one round-trip, saving unecessary round-trips to the database.
 *
 * This example shows how to set the rowPrefetch for individual
 * statements.
 *
 */

// You need to import the java.sql package to use JDBC
import java.sql.*;

// You need to import oracle.jdbc in order to use the
// Oracle extensions
import oracle.jdbc.*;
```

```
class RowPrefetch_statement
{
  public static void main (String args [])
       throws SQLException
  {
    // Load the Oracle JDBC driver
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    // Connect to the database
    // You can put a database name after the @ sign in the connection URL.
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

    // get the value of the default row prefetch from the connection object

    int default_row_prefetch =
      ((OracleConnection)conn).getDefaultRowPrefetch ();
    System.out.println ("The Default RowPrefetch for the connection is: "
                                       + default_row_prefetch);

    Statement stmt = conn.createStatement ();

    // set the RowPrefetch value from the statement object
    // This sets the rowPrefetch only for this particular statement.
    // All other statements will use the default RowPrefetch from the
    // connection.

    ((OracleStatement)stmt).setRowPrefetch (30);

    // Check to verify statement rowPrefetch value is 30.
    int row_prefetch = ((OracleStatement)stmt).getRowPrefetch ();
    System.out.println ("The RowPrefetch for the statement is: "
                                        + row_prefetch + "\n");

    ResultSet rset = stmt.executeQuery ("select ename from emp");

    while(rset.next ())
    {
      System.out.println (rset.getString (1));
    }
    rset.close ();
    stmt.close ();
    stmt.close ();
  }
}
```

Oracle 列型定義 : DefineColumnType.java

このサンプルでは、Oracle 拡張機能を使用して結果セット列型を事前定義し、問合せごとのデータベースへのラウンドトリップを削減します。

列型定義の詳細は、12-22 ページの「列型の定義」を参照してください。

```
/*
 * This sample shows how to use the "define" extensions.
 * The define extensions allow the user to specify the types
 * under which to retrieve column data in a query.
 *
 * This saves round-trips to the database (otherwise necessary to
 * gather information regarding the types in the select-list) and
 * conversions from native types to the types under which the user
 * will get the data.
 *
 * This can also be used to avoid streaming of long columns, by defining
 * them as CHAR or VARCHAR types.
 */

// You need to import the java.sql package to use JDBC
import java.sql.*;

// You need to import oracle.jdbc.* in order to use the
// API extensions.
import oracle.jdbc.*;

class DefineColumnType
{
  public static void main (String args [])
       throws SQLException
  {
    // Load the Oracle JDBC driver
    DriverManager.registerDriver( new oracle.jdbc.OracleDriver());

    // Connect to the database
    // You can put a database name after the @ sign in the connection URL.
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

    Statement stmt = conn.createStatement ();
```

```
    // Call DefineColumnType to specify that the column will be
    // retrieved as a String to avoid conversion from NUMBER to String
    // on the client side.  This also avoids a round-trip to the
    // database to get the column type.
    //
    // There are 2 defineColumnType API.  We use the one with 3 arguments.
    // The 3rd argument allows us to specify the maximum length
    // of the String.  The values obtained for this column will
    // not exceed this length.

    ((OracleStatement)stmt).defineColumnType (1, Types.VARCHAR, 7);

    ResultSet rset = stmt.executeQuery ("select empno from emp");
    while (rset.next ())
    {
      System.out.println (rset.getString (1));
    }

    // Close the resultSet
    rset.close();

    // Close the statement
    stmt.close ();

    // Close the connection
    conn.close();
  }
}
```

暗黙的文キャッシュ : StmtCache1.java

このサンプル・アプリケーションでは、暗黙的文キャッシュを使用してデータベースから結果セットを作成し、従業員名などの様々な情報を出力します。

暗黙的文キャッシュの詳細は、13-8 ページの「暗黙的文キャッシュの使用方法」および第 13 章「文キャッシュ」のその他の関連項目を参照してください。

```
/*
 * This sample to demonstrate Implicit Statement Caching. This can be
 * enabled by calling setStmtCacheSize on the Connection Object.
 */

// You need to import the java.sql package to use JDBC
import java.sql.*;
import oracle.jdbc.*;
```

```
class StmtCache1
{
  public static void main (String args [])
       throws SQLException
  {
    // Load the Oracle JDBC driver
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    // Connect to the database
    // You can put a database name after the @ sign in the connection URL.
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

    ((OracleConnection)conn).setStmtCacheSize(1);

    Connection sysconn = DriverManager.getConnection("jdbc:oracle:oci8:@",
                                                     "system", "manager");
    String sql = "select ENAME from EMP";

    System.out.println("Beging of 1st execution");
    getOpenCursors (sysconn);

    // Create a Statement
    PreparedStatement stmt = conn.prepareStatement (sql);
    System.out.println("1. Stmt is " + stmt);

    // Select the ENAME column from the EMP table
    ResultSet rset = stmt.executeQuery ();

    // Iterate through the result and print the employee names
    while (rset.next ())
      System.out.println (rset.getString (1));

    // Close the RseultSet
    rset.close();

    // Close the Statement
    stmt.close();

    System.out.println("End of 1st execution");
    getOpenCursors (sysconn);

    System.out.println("Reexecuting the same SQL");

    stmt = conn.prepareStatement (sql);

    System.out.println("2. Stmt is " + stmt);
```

```
    // Select the ENAME column from the EMP table
    rset = stmt.executeQuery ();

    // Iterate through the result and print the employee names
    while (rset.next ())
      System.out.println (rset.getString (1));

    // Close the RseultSet
    rset.close();

    // Close the Statement
    stmt.close();

    System.out.println("End of 2nd execution");
    getOpenCursors (sysconn);

    // Close the connection
    conn.close();

    System.out.println("After close of connection");
    getOpenCursors (sysconn);

    sysconn.close();
  }

  private static void getOpenCursors (Connection conn)
     throws SQLException
  {
     System.out.println("Open Cusrors are : ");
     Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery ("select SQL_TEXT from V$OPEN_CURSOR");
     while (rs.next())
       System.out.println("Cursor's sql text is " + rs.getString(1));
     rs.close();
     rs = null;
     stmt.close();
     stmt = null;
  }
}
```

明示的文キャッシュ : StmtCache2.java

このサンプル・アプリケーションでは、明示的文キャッシュを使用してデータベースから結果セットを作成し、従業員名などの様々な情報を出力します。

明示的文キャッシュの詳細は、13-10 ページの「明示的文キャッシュの使用方法」および第 13 章「文キャッシュ」のその他の関連項目を参照してください。

```
/*
 * This sample to demonstrate Explicit Statement Caching. This can be
 * enabled by calling Oracle Specific calls like closeWithKey,
 * prepareStatementWithKey etc.
 */

// You need to import the java.sql package to use JDBC
import java.sql.*;
import oracle.jdbc.*;

class StmtCache2
{
  public static void main (String args [])
       throws SQLException
  {
    // Load the Oracle JDBC driver
    DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

    // Connect to the database
    // You can put a database name after the @ sign in the connection URL.
    Connection conn =
      DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

    ((OracleConnection)conn).setStmtCacheSize(1);

    Connection sysconn = DriverManager.getConnection("jdbc:oracle:oci8:@",
                                                     "system", "manager");
    String sql = "select ENAME from EMP";

    System.out.println("Beging of 1st execution");
    getOpenCursors (sysconn);

    // Create a Statement
    PreparedStatement stmt = conn.prepareStatement (sql);
    System.out.println("1. Stmt is " + stmt);

    // Select the ENAME column from the EMP table
    ResultSet rset = stmt.executeQuery ();
```

```
  // Iterate through the result and print the employee names
  while (rset.next ())
    System.out.println (rset.getString (1));

  // Close the RseultSet
  rset.close();

  // Close the Statement
  //stmt.close();
  ((OracleStatement)stmt).closeWithKey ("mysql");

  System.out.println("End of 1st execution");
  getOpenCursors (sysconn);

  System.out.println("Reexecuting the same SQL");

  stmt = ((OracleConnection)conn).prepareStatementWithKey ("mysql");

  System.out.println("2. Stmt is " + stmt);

  // Select the ENAME column from the EMP table
  rset = stmt.executeQuery ();

  // Iterate through the result and print the employee names
  while (rset.next ())
    System.out.println (rset.getString (1));

  // Close the RseultSet
  rset.close();

  // Close the Statement
  stmt.close();

  System.out.println("End of 2nd execution");
  getOpenCursors (sysconn);

  // Close the connection
  conn.close();

  System.out.println("After close of connection");
  getOpenCursors (sysconn);

  sysconn.close();
}
```

```
  private static void getOpenCursors (Connection conn)
     throws SQLException
  {
     System.out.println("Open Cusrors are : ");
     Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery ("select SQL_TEXT from V$OPEN_CURSOR");
     while (rs.next())
       System.out.println("Cursor's sql text is " + rs.getString(1));
     rs.close();
     rs = null;
     stmt.close();
     stmt = null;
  }
}
```

接続プーリングと分散トランザクションのサンプル

この項には、JDBC 2.0 拡張要素機能のサンプルが含まれています。JDBC 2.0 拡張要素機能には、次のように、データ・ソース、接続プーリング、接続キャッシュおよび分散トランザクション（XA）があります。

- JNDI を使用しないデータ・ソース : DataSource.java
- JNDI を使用するデータ・ソース : DataSourceJNDI.java
- プーリングされた接続 : PooledConnection.java
- Oracle 接続キャッシュ（動的）: CCache1.java
- Oracle 接続キャッシュ（待機なし固定）: CCache2.java
- 保留と再開 XA: XA2.java
- 2 フェーズ・コミットの操作 XA: XA4.java

JNDI を使用しないデータ・ソース : DataSource.java

この例では、JNDI を使用せずに JDBC 2.0 データ・ソースを使用する方法を示します。JNDI を使用するデータ・ソースの使用方法、または JNDI を使用しないデータ・ソースの使用方法など、データ・ソースに関する一般的な情報については、14-2 ページの「データ・ソース」を参照してください。

```
/**
 * A Simple DataSource sample without using JNDI.
 */

// You need to import the java.sql package to use JDBC
import java.sql.*;
import javax.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.OracleDataSource;

public class DataSource
{
  public static void main (String args [])
    throws SQLException
  {
    // Create a OracleDataSource instance explicitly
    OracleDataSource ods = new OracleDataSource();

    // Set the user name, password, driver type and network protocol
    ods.setUser("scott");
    ods.setPassword("tiger");
    ods.setDriverType("oci8");
    ods.setNetworkProtocol("ipc");

    // Retrieve a connection
    Connection conn = ods.getConnection();
    getUserName(conn);
    // Close the connection
    conn.close();
    conn = null;
  }

  static void getUserName(Connection conn)
      throws SQLException
  {
    // Create a Statement
    Statement stmt = conn.createStatement ();

    // Select the ENAME column from the EMP table
    ResultSet rset = stmt.executeQuery ("select USER from dual");
```

```
    // Iterate through the result and print the employee names
    while (rset.next ())
      System.out.println ("User name is " + rset.getString (1));

    // Close the RseultSet
    rset.close();
    rset =  null;

    // Close the Statement
    stmt.close();
    stmt = null;
  }
}
```

JNDI を使用するデータ・ソース : DataSourceJNDI.java

この例では、JNDI を使用して JDBC 2.0 データ・ソースを使用する方法を示します。JNDI を使用するデータ・ソースの使用方法、または JNDI を使用しないデータ・ソースの使用方法など、データ・ソースに関する一般的な情報については、14-2 ページの「データ・ソース」を参照してください。

このクラスには、`getUserName()` メソッドの他に、JNDI 機能の `do_bind()` および `do_lookup()` メソッドが含まれています。

```
/**
 * A Simple DataSource sample with JNDI.
 * This is tested using File System based reference
 * implementation of JNDI SPI driver from JavaSoft.
 * You need to download fscontext1_2beta2.zip from
 * JavaSoft site.
 * Include providerutil.jar & fscontext.jar extracted
 * from the above ZIP in the classpath.
 * Create a directory /tmp/JNDI/jdbc
 */

// You need to import the java.sql package to use JDBC
import java.sql.*;
import javax.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.OracleDataSource;
import javax.naming.*;
import javax.naming.spi.*;
import java.util.Hashtable;
```

```
public class DataSourceJNDI
{
  public static void main (String args [])
    throws SQLException, NamingException
  {
    // Initialize the Context
    Context ctx = null;
    try {
      Hashtable env = new Hashtable (5);
      env.put (Context.INITIAL_CONTEXT_FACTORY,
               "com.sun.jndi.fscontext.RefFSContextFactory");
      env.put (Context.PROVIDER_URL, "file:/tmp/JNDI");
      ctx = new InitialContext(env);
    } catch (NamingException ne)
    {
      ne.printStackTrace();
    }

    do_bind(ctx, "jdbc/sampledb");
    do_lookup(ctx, "jdbc/sampledb");

  }

  static void do_bind (Context ctx, String ln)
    throws SQLException, NamingException
  {
    // Create a OracleDataSource instance explicitly
    OracleDataSource ods = new OracleDataSource();

    // Set the user name, password, driver type and network protocol
    ods.setUser("scott");
    ods.setPassword("tiger");
    ods.setDriverType("oci8");
    ods.setNetworkProtocol("ipc");

    // Bind it
    System.out.println ("Doing a bind with the logical name : " + ln);
    ctx.bind (ln,ods);
  }

  static void do_lookup (Context ctx, String ln)
    throws SQLException, NamingException
  {

    System.out.println ("Doing a lookup with the logical name : " + ln);
    OracleDataSource ods = (OracleDataSource) ctx.lookup (ln);
```

```
    // Retrieve a connection
    Connection conn = ods.getConnection();
    getUserName(conn);
    // Close the connection
    conn.close();
    conn = null;
  }

  static void getUserName(Connection conn)
       throws SQLException
  {
    // Create a Statement
    Statement stmt = conn.createStatement ();

    // Select the ENAME column from the EMP table
    ResultSet rset = stmt.executeQuery ("select USER from dual");

    // Iterate through the result and print the employee names
    while (rset.next ())
      System.out.println ("User name is " + rset.getString (1));

    // Close the RseultSet
    rset.close();
    rset =  null;

    // Close the Statement
    stmt.close();
    stmt = null;
  }
}
```

プーリングされた接続 : PooledConnection.java

JDBC 2.0 のプールされた接続機能の使用方法を示す簡単な例です。接続プーリングの詳細は、14-12 ページの「接続プーリング」を参照してください。

```
/*
 * A simple Pooled Connection Sample
 */

import java.sql.*;
import javax.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;

class PooledConnection1
{
  public static void main (String args [])
       throws SQLException
  {

    // Create a OracleConnectionPoolDataSource instance
    OracleConnectionPoolDataSource ocpds =
                               new OracleConnectionPoolDataSource();

    // Set connection parameters
    ocpds.setURL("jdbc:oracle:oci8:@");
    ocpds.setUser("scott");
    ocpds.setPassword("tiger");

    // Create a pooled connection
    PooledConnection pc  = ocpds.getPooledConnection();

    // Get a Logical connection
    Connection conn = pc.getConnection();

    // Create a Statement
    Statement stmt = conn.createStatement ();

    // Select the ENAME column from the EMP table
    ResultSet rset = stmt.executeQuery ("select ENAME from EMP");

    // Iterate through the result and print the employee names
    while (rset.next ())
      System.out.println (rset.getString (1));

    // Close the RseultSet
    rset.close();
    rset = null;
```

```
    // Close the Statement
    stmt.close();
    stmt = null;

    // Close the logical connection
    conn.close();
    conn = null;

    // Close the pooled connection
    pc.close();
    pc = null;
  }
}
```

OCI 接続プール : OCIConnectionPool.java

このサンプル・コードは、OCI 接続プーリングを示します。

OCI 接続プーリングの詳細は、16-2 ページの「OCI ドライバ接続プーリング」を参照してください。

```
/*
 * A simple OCI Connection Pool Sample
 */

import java.sql.*;
import javax.sql.*;
import java.util.Properties;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;
import oracle.jdbc.oci.*;

class OCIConnectionPool
{
  public static void main (String args [])
       throws SQLException
  {
```

```
    String url = "jdbc:oracle:oci:@";
    try {
      String url1 = System.getProperty("JDBC_URL");
      if (url1 != null)
        url = url1;
    } catch (Exception e) {
      // If there is any security exception, ignore it
      // and use the default
    }

    // Create an OracleOCIConnectionPool instance with default configuration
    OracleOCIConnectionPool cpool = new OracleOCIConnectionPool("scott",
      "tiger", url, null);

    // Print out the default configuration for the OracleOCIConnectionPool
    System.out.println ("-- The default configuration for the
  OracleOCIConnectionPool --");
    displayPoolConfig(cpool);

    // Get a connection from the pool
    OracleOCIConnection conn1 = (OracleOCIConnection) cpool.getConnection("scott",
"tiger");

    // Create a Statement
    Statement stmt = conn1.createStatement ();

    // Select the ENAME column from the EMP table
    ResultSet rset = stmt.executeQuery ("select ENAME from EMP");

    // Iterate through the result and print the employee names
    System.out.println ("-- Use the connection from the OracleOCIConnectionPool
      --");
    while (rset.next ())
      System.out.println (rset.getString (1));

    System.out.println ("-- Use another connection from the OracleOCIConnectionPool
--");

    // Get another connection from the pool with different userID and password
    OracleOCIConnection conn2 = (OracleOCIConnection) cpool.getConnection("system",
"manager");

    // Create a Statement
    stmt = conn2.createStatement ();

    // Select the USER from DUAL to test the connection
    rset = stmt.executeQuery ("select USER from DUAL");
```

```
    // Iterate through the result and print the employee names
    rset.next ();
    System.out.println (rset.getString (1));

    // Reconfigure the OracleOCIConnectionPool in case the performance is too bad.
    // This might happen when many users are trying to connect at the same time.
    // In this case, increase MAX_LIMIT to some larger number, and also increase
    // INCREMENT to a positive number.

    Properties p  = new Properties();
    p.put (OracleOCIConnectionPool.CONNPOOL_MIN_LIMIT,
Integer.toString(cpool.getMinLimit()));
    p.put (OracleOCIConnectionPool.CONNPOOL_MAX_LIMIT,
Integer.toString(cpool.getMaxLimit() * 2)) ;
    if (cpool.getConnectionIncrement() > 0)
        // Keep the old value
        p.put (OracleOCIConnectionPool.CONNPOOL_INCREMENT,
Integer.toString(cpool.getConnectionIncrement()));
    else
        // Set it to a number larger than 0
        p.put (OracleOCIConnectionPool.CONNPOOL_INCREMENT, "1") ;

    // Enable the new configuration
    cpool.setPoolConfig(p);

    // Print out the current configuration for the OracleOCIConnectionPool
    System.out.println ("-- The new configuration for the OracleOCIConnectionPool
--");
    displayPoolConfig(cpool);

    // Close the RseultSet
    rset.close();
    rset = null;

    // Close the Statement
    stmt.close();
    stmt = null;

    // Close the connections
    conn1.close();
    conn2.close();
    conn1 = null;
    conn2 = null;
```

```
    // Close the OracleOCIConnectionPool
    cpool.close();
    cpool = null;
  }

  // Display the current status of the OracleOCIConnectionPool
  private static void displayPoolConfig (OracleOCIConnectionPool cpool)
      throws SQLException
  {
    System.out.println (" Min poolsize Limit: " + cpool.getMinLimit());
    System.out.println (" Max poolsize Limit: " + cpool.getMaxLimit());
    System.out.println (" Connection Increment: " + cpool.getConnectionIncrement());
    System.out.println (" NoWait: " + cpool.getNoWait());
    System.out.println (" Timeout: " + cpool.getTimeout());
    System.out.println (" PoolSize: " + cpool.getPoolSize());
    System.out.println (" ActiveSize: " + cpool.getActiveSize());
  }
}
```

中間層認証 : NtierAuth.java

このサンプル・コードは、プロキシ接続を介する中間層認証を示します。

中間層認証の詳細は、16-12 ページの「プロキシ接続を介する中間層認証」を参照してください。

```
/*
 * A Ntier Authentication Sample
 *
 */

import java.sql.*;
import javax.sql.*;
import java.util.Properties;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;
import oracle.jdbc.oci.*;

class NtierAuth
{
  public static void main (String args [])
       throws SQLException
  {
    // Step 1: Connect as system/manager to create the users, setup roles and
proxies.
```

```
// Load the Oracle JDBC driver
DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

String url = "jdbc:oracle:oci8:@";
try {
  String url1 = System.getProperty("JDBC_URL");
  if (url1 != null)
    url = url1;
} catch (Exception e) {
  // If there is any security exception, ignore it
  // and use the default
}

// Connect to the database as system/manager
Connection sysConn = DriverManager.getConnection (url, "system", "manager");

// Do some cleanup
trySQL (sysConn, "drop role role1");
trySQL (sysConn, "drop role role2");
trySQL (sysConn, "drop user client cascade");
trySQL (sysConn, "drop user proxy cascade");

// Create a Statement
Statement sysStmt = sysConn.createStatement ();

// Create client and proxy
sysStmt.execute("create user proxy identified by mehul");
sysStmt.execute("create user client identified by ding");

// Grant privilages to client and proxy
sysStmt.execute("grant create session, connect, resource to proxy");
sysStmt.execute("grant create session, connect, resource to client");

// Create and setup roles with system connection
sysStmt.execute("create role role1");
sysStmt.execute("create role role2");

// Connect to the database as proxy
Connection proxyConn = DriverManager.getConnection (url, "proxy", "mehul");

// Create a table with proxy connection
Statement proxyStmt = proxyConn.createStatement ();
proxyStmt.execute("create table account (purchase number)");
proxyStmt.execute("insert into account values (6)");
```

```
    // Grant privilages to role1, role2
    proxyStmt.execute("grant select on account to role1");
    proxyStmt.execute("grant insert on account to role2");

    // Close the proxy statement and connection
    proxyStmt.close();
    proxyConn.close();

    // Grant role1, role2 to client
    sysStmt.execute("grant role1, role2 to client");

    // Grant proxy privilage to connect as client
    sysStmt.execute("alter user client grant connect through proxy with role
role1");

    // Step 2: Use OCIConnectionPool to get the proxy connection

    // Create an OracleOCIConnectionPool instance with default configuration using
proxy/mehul
    OracleOCIConnectionPool cpool = new OracleOCIConnectionPool("proxy", "mehul",
url, null);

    Properties prop = new Properties();
    String[] roles = {"role1"};
    prop.put(OracleOCIConnectionPool.PROXY_USER_NAME,"client" );
    prop.put(OracleOCIConnectionPool.PROXY_ROLES, roles);

    // Get the proxy connection
    OracleOCIConnection conn = (OracleOCIConnection)
         cpool.getProxyConnection(OracleOCIConnectionPool.PROXYTYPE_USER_NAME,
prop);

    // Create a Statement
    Statement stmt = conn.createStatement ();

    // Select the ENAME column from the EMP table
    ResultSet rset = stmt.executeQuery ("select * from proxy.account");

    // Iterate through the result and print the purchase number
    System.out.println ("-- Do a Select from the proxy connection --");
    while (rset.next ())
      System.out.println (rset.getString (1));

    // Close the RseultSet
    rset.close();
    rset = null;
```

```
  // Now, try to do an Insert. This shouldn't be authorized
  System.out.println ("-- Now, try to do an Insert with the proxy connection --");
  try {
    stmt.execute("insert into proxy.account values (9)");
  } catch (SQLException e) {
    System.out.println ("Exception thrown: " + e.getMessage());
  }
  finally {
      if (stmt != null)
         stmt.close();
  }

  // Close the connection
  conn.close();
  conn = null;

  // Close the OracleOCIConnectionPool
  cpool.close();
  cpool = null;

  // Make the cleanup
  trySQL (sysConn, "drop role role1");
  trySQL (sysConn, "drop role role2");
  trySQL (sysConn, "drop user client cascade");
  trySQL (sysConn, "drop user proxy cascade");

  // Close the system statement and connection
  sysStmt.close();
  sysConn.close();

}

// Used for Cleaning up the database
private static void trySQL (Connection conn, String sqlString)
    throws SQLException
{
  // Create a Statement
  Statement stmt = conn.createStatement ();
```

```
    try {
        stmt.execute(sqlString);
        stmt.close();
    } catch (SQLException e) {
        // In case the user or role hasn't been created, ignore it.
    }
    finally {
        if (stmt != null)
           stmt.close();
    }
  }
}
```

JDBC OCI アプリケーション・フェイルオーバー・コールバック：OCIFailOver.java

このサンプルは、JDBC OCI アプリケーション・フェイルオーバー・コールバックの登録および操作方法を示します。

透過的アプリケーション・フェイルオーバー（TAF）およびフェイルオーバー・イベントに関する詳細は、16-15 ページの「OCI ドライバの透過的アプリケーション・フェイルオーバー」を参照してください。

```
/*
 * This sample demonstrates the registration and operation of
 * JDBC OCI application failover callbacks
 *
 * Note: Before you run this sample, set up the following
 *       service in tnsnames.ora:
 *       inst_primary=(DESCRIPTION=
 *             (ADDRESS=(PROTOCOL=tcp)(Host=hostname)(Port=1521))
 *             (CONNECT_DATA=(SERVICE_NAME=ORCL)
 *                           (FAILOVER_MODE=(TYPE=SELECT)(METHOD=BASIC))
 *            )
 *          )
 *       Please see the Oracle Net Administrator's Guide for more detail about
 *       failover_mode
 *
 * To demonstrate the the functionality, first compile and start up the sample,
 *    then log into sqlplus and connect /as sysdba. While the sample is still
 *    running, shutdown the database with "shutdown abort;". At this moment,
 *    the failover callback functions should be invoked. Now, the database can
 *    be restarted, and the interupted query will be continued.
 */
```

```
// You need to import java.sql and oracle.jdbc packages to use
// JDBC OCI failover callback

import java.sql.*;
import java.net.*;
import java.io.*;
import java.util.*;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.OracleOCIFailover;


public class OCIFailOver {

  static final String user = "scott";
  static final String password = "tiger";
  static final String driver_class = "oracle.jdbc.OracleDriver";

  static final String URL = "jdbc:oracle:oci8:@inst_primary";


  public static void main (String[] args) throws Exception {

    Connection conn = null;
    CallBack   fcbk= new CallBack();
    String     msg = null;
    Statement  stmt = null;
    ResultSet rset = null;

    // Load JDBC driver
    try {
      Class.forName(driver_class);
    }
    catch(Exception e) {
      System.out.println(e);
    }

    // Connect to the database
    conn = DriverManager.getConnection(URL, user, password);

    // register TAF callback function
    ((OracleConnection) conn).registerTAFCallback(fcbk, msg);

    // Create a Statement
    stmt = conn.createStatement ();
```

```
    for (int i=0; i<30; i++) {
      // Select the ENAME column from the EMP table
      rset = stmt.executeQuery ("select ENAME from EMP");

      // Iterate through the result and print the employee names
      while (rset.next ())
        System.out.println (rset.getString (1));

      // Sleep one second to make it possible to shutdown the DB.
      Thread.sleep(1000);
    } // End for

    // Close the RseultSet
    rset.close();

    // Close the Statement
    stmt.close();

    // Close the connection
    conn.close();


  } // End Main()

} // End class jdemofo


/*
 * Define class CallBack
 */
class CallBack implements OracleOCIFailover {

   // TAF callback function
   public int callbackFn (Connection conn, Object ctxt, int type, int event) {
```

```
/**********************************************************************
 * There are 7 possible failover event
 *   FO_BEGIN = 1   indicates that failover has detected a
 *                  lost conenction and faiover is starting.
 *   FO_END = 2     indicates successful completion of failover.
 *   FO_ABORt = 3   indicates that failover was unsuccessful,
 *                  and there is no option of retrying.
 *   FO_REAUTH = 4  indicates that a user handle has been re-
 *                  authenticated.
 *   FO_ERROR = 5   indicates that failover was temporarily un-
 *                  successful, but it gives the apps the opp-
 *                  ortunity to handle the error and retry failover.
 *                  The usual method of error handling is to issue
 *                  sleep() and retry by returning the value FO_RETRY
 *   FO_RETRY = 6
 *   FO_EVENT_UNKNOWN = 7  It is a bad failover event
 **********************************************************************/
String failover_type = null;

switch (type) {
    case FO_SESSION:
           failover_type = "SESSION";
           break;
    case FO_SELECT:
           failover_type = "SELECT";
           break;
    default:
           failover_type = "NONE";
}

switch (event) {

  case FO_BEGIN:
       System.out.println(ctxt + ": "+ failover_type + " failing over...");
       break;
  case FO_END:
       System.out.println(ctxt + ": failover ended");
       break;
  case FO_ABORT:
       System.out.println(ctxt + ": failover aborted.");
       break;
  case FO_REAUTH:
       System.out.println(ctxt + ": failover.");
       break;
  case FO_ERROR:
       System.out.println(ctxt + ": failover error gotten. Sleeping...");
       // Sleep for a while
```

```
            try {
              Thread.sleep(100);
            }
            catch (InterruptedException e) {
               System.out.println("Thread.sleep has problem: " + e.toString());
            }
            return FO_RETRY;
       default:
            System.out.println(ctxt + ": bad failover event.");
            break;

     }

     return 0;

   }
}
```

Oracle 接続キャッシュ（動的）: CCache1.java

クラス OracleConnectionCacheImpl で使用可能な Oracle 実装例を使用する、2 つの接続キャッシュの例の最初の 1 つです。

この例では、プーリングされた接続がすでに最大数に達している場合に、動的スキームを使用します。新しくプーリングされる接続インスタンスは、必要に応じて作成されます。しかし、JDBC アプリケーションがプーリングされた接続インスタンスによって提供される論理接続インスタンスの使用を終了するとすぐに、自動的にクローズされ、解放されます。

一般的な接続キャッシュの詳細および特定の Oracle 実装例については、14-16 ページの「接続キャッシュ」を参照してください。

```
/**
 *  JDBC 2.0 Spec doesn't mandate that JDBC vendors implement a
 *  Connection Cache. However, we implemented a basic one with two
 *  schemes as an example.
 *  A Sample demo to illustrate DYNAMIC_SCHEME of OracleConnectionCacheImpl.
 *  Dynamic Scheme :  This is the default scheme. New connections could be
 *  created beyond the Max limit upon request but closed and freed when the
 *  logical connections are closed. When all the connections are active and
 *  busy, requests for new connections willend up creating new physical
 *  connections. But these physical connections are closed when the
 *  corresponding logical connections are closed. A typical grow and shrink
 *  scheme.
 */
```

```
import java.sql.*;
import javax.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;

class CCache1
{
  public static void main (String args [])
       throws SQLException
  {
    OracleConnectionCacheImpl ods = new OracleConnectionCacheImpl();
    ods.setURL("jdbc:oracle:oci8:@");
    ods.setUser("scott");
    ods.setPassword("tiger");

     // Set the Max Limit
    ods.setMaxLimit (3);

    Connection conn1 = null;
    conn1 = ods.getConnection();
    if (conn1 != null)
      System.out.println("Connection 1 " + " Succeeded!");
    else
      System.out.println("Connection 1 " + " Failed !!!");

    Connection conn2 = null;
    conn2 = ods.getConnection();
    if (conn2 != null)
      System.out.println("Connection 2 " + " Succeeded!");
    else
      System.out.println("Connection 2 " + " Failed !!!");

    Connection conn3 = null;
    conn3 = ods.getConnection();
    if (conn3 != null)
      System.out.println("Connection 3 " + " Succeeded!");
    else
      System.out.println("Connection 3 " + " Failed !!!");

    Connection conn4 = null;
    conn4 = ods.getConnection();
    if (conn4 != null)
      System.out.println("Connection 4 " + " Succeeded!");
    else
      System.out.println("Connection 4 " + " Failed !!!");
```

```
    Connection conn5 = null;
    conn5 = ods.getConnection();
    if (conn5 != null)
      System.out.println("Connection 5 " + " Succeeded!");
    else
      System.out.println("Connection 5 " + " Failed !!!");


    System.out.println("Active size : " + ods.getActiveSize());
    System.out.println("Cache Size is " + ods.getCacheSize());

    // Close 3 logical Connections
    conn1.close();
    conn2.close();
    conn3.close();

    System.out.println("Active size : " + ods.getActiveSize());
    System.out.println("Cache Size is " + ods.getCacheSize());

    // close the Data Source
    ods.close();

    System.out.println("Active size : " + ods.getActiveSize());
    System.out.println("Cache Size is " + ods.getCacheSize());
  }
}
```

Oracle 接続キャッシュ（待機なし固定）: CCache2.java

クラス `OracleConnectionCacheImpl` で使用可能な Oracle 実装例を使用する、2 つの接続キャッシュの例の 2 番目です。

この例では、プーリングされた接続がすでに最大数に達している場合に、待機なし固定のスキームを使用します。接続が要求されると、`null` が返されます。

一般的な接続キャッシュの詳細および特定の Oracle 実装例については、14-16 ページの「接続キャッシュ」を参照してください。

```
/**
 *  JDBC 2.0 Spec doesn't mandate that JDBC vendors implement a
 *  Connection Cache. However, we implemented a basic one with 2
 *  schemes as an Example.
 *  A Sample demo to illustrate FIXED_RETURN_NULL_SCHEME of
 *  OracleConnectionCacheImpl.
 *  Fixed with NoWait :  At no instance there will be more active
 *  connections than the Maximum limit. Request for new connections
 *  beyond the max limit will return null.
 */

// You need to import the java.sql package to use JDBC
import java.sql.*;
import javax.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;

public class CCache2 {

  public static void main (String args [])
       throws SQLException
  {

    // Create a OracleConnectionPoolDataSource as an factory
    // of PooledConnections for the Cache to create.
    OracleConnectionPoolDataSource ocpds =
                               new OracleConnectionPoolDataSource();
    ocpds.setURL("jdbc:oracle:oci8:@");
    ocpds.setUser("scott");
    ocpds.setPassword("tiger");

    // Associate it with the Cache
    OracleConnectionCacheImpl ods = new OracleConnectionCacheImpl(ocpds);

    // Set the Max Limit
    ods.setMaxLimit (3);

    // Set the Scheme
    ods.setCacheScheme (OracleConnectionCacheImpl.FIXED_RETURN_NULL_SCHEME);
```

```
    Connection conn = null;
    for (int i=0; i < 5; ++i )
    {
      conn = ods.getConnection();
      if (conn != null)
        System.out.println("Connection " + i + " Succeeded!");
      else
        System.out.println("Connection " + i + " Failed !!!");
    }

    System.out.println("Active size : " + ods.getActiveSize());
    System.out.println("Cache Size is " + ods.getCacheSize());

    // close the Data Source
    ods.close();

    System.out.println("Active size : " + ods.getActiveSize());
    System.out.println("Cache Size is " + ods.getCacheSize());

  }
}
```

保留と再開 XA: XA2.java

このサンプルでは、トランザクションを保留および再開する方法を示します。トランザクションを保留および再開するには、標準 XA リソース機能を使用しますが、かわりに Oracle 拡張機能 `suspend()` および `resume()` メソッドを使用する方法についてのコメントも含まれています。

このクラスには、この例で使用するトランザクション ID を形成する `createXid()` メソッドが含まれています。

分散トランザクションおよび XA 機能の詳細は、第 15 章「分散トランザクション」を参照してください。

```
/*
    A simple XA demo with suspend and resume. Opens 2 global
    transactions each of one branch. Does some DML on the first one
    and suspends it and does some DML on the 2nd one and resumes the
    first one and commits. Basically, to illustrate interleaving
    of global transactions.
    Need a java enabled 8.1.6 database to run this demo.
 */
```

```
// You need to import the java.sql package to use JDBC
import java.sql.*;
import javax.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;
import oracle.jdbc.xa.OracleXid;
import oracle.jdbc.xa.OracleXAException;
import oracle.jdbc.xa.client.*;
import javax.transaction.xa.*;

class XA2
{
  public static void main (String args [])
       throws SQLException
  {

    try
    {
        DriverManager.registerDriver(new OracleDriver());

        // You can put a database name after the @ sign in the connection URL.
        Connection conn =
          DriverManager.getConnection ("jdbc:oracle:oci8:@", "scott", "tiger");

        // Prepare a statement to create the table
        Statement stmt = conn.createStatement ();

        try
        {
          // Drop the test table
          stmt.execute ("drop table my_table");
        }
        catch (SQLException e)
        {
          // Ignore an error here
        }

        try
        {
          // Create a test table
          stmt.execute ("create table my_table (col1 int)");
        }
        catch (SQLException e)
        {
          // Ignore an error here too
        }
```

```
try
{
  // Drop the test table
  stmt.execute ("drop table my_tab");
}
catch (SQLException e)
{
  // Ignore an error here
}

try
{
  // Create a test table
  stmt.execute ("create table my_tab (col1 int)");
}
catch (SQLException e)
{
  // Ignore an error here too
}

// Create a XADataSource instance
OracleXADataSource oxds = new OracleXADataSource();
oxds.setURL("jdbc:oracle:oci8:@");
oxds.setUser("scott");
oxds.setPassword("tiger");

// get a XA connection
XAConnection pc  = oxds.getXAConnection();
// Get a logical connection
Connection conn1 = pc.getConnection();

// Get XA resource handle
XAResource oxar = pc.getXAResource();
Xid xid1 = createXid(111,111);

// Start a transaction branch
oxar.start (xid1, XAResource.TMNOFLAGS);

// Create a Statement
Statement stmt1 = conn1.createStatement ();

// Do some DML
stmt1.executeUpdate ("insert into my_table values (2727)");

// Suspend the first global transaction
// ((OracleXAResource)oxar).suspend (xid1); or
oxar.end (xid1, XAResource.TMSUSPEND);
```

```
Xid xid2 = createXid(222,222);
oxar.start (xid2, XAResource.TMNOFLAGS);
Statement stmt2 = conn1.createStatement ();
stmt2.executeUpdate ("insert into my_tab values (7272)");
oxar.commit (xid2, true);
stmt2.close();
stmt2 = null;

// Close the Statement
stmt1.close();
stmt1 = null;

// Resume the first global transaction
// ((OracleXAResource)oxar).resume (xid1); or
oxar.start (xid1, XAResource.TMRESUME);

// End the branch
oxar.end(xid1, XAResource.TMSUCCESS);

// Do a 1 phase commit
oxar.commit (xid1, true);

// Close the connection
conn1.close();
conn1 = null;

// close the XA connection
pc.close();
pc = null;

ResultSet rset = stmt.executeQuery ("select col1 from my_table");
while (rset.next())
  System.out.println("Col1 is " + rset.getInt(1));

rset.close();
rset = null;

rset = stmt.executeQuery ("select col1 from my_tab");
while (rset.next())
  System.out.println("Col1 is " + rset.getString(1));

rset.close();
rset = null;

stmt.close();
stmt = null;
```

```
        conn.close();
        conn = null;

      } catch (SQLException sqe)
      {
        sqe.printStackTrace();
      } catch (XAException xae)
      {
        if (xae instanceof OracleXAException)
        {
          System.out.println("XA error is " +
                             ((OracleXAException)xae).getXAError());
          System.out.println("SQL error is " +
                             ((OracleXAException)xae).getOracleError());
        }
        xae.printStackTrace();
      }
    }

    static Xid createXid(int gd, int bd)
      throws XAException
    {
      byte[] gid = new byte[1]; gid[0]= (byte) gd;
      byte[] bid = new byte[1]; bid[0]= (byte) bd;
      byte[] gtrid = new byte[64];
      byte[] bqual = new byte[64];
      System.arraycopy (gid, 0, gtrid, 0, 1);
      System.arraycopy (bid, 0, bqual, 0, 1);
      Xid xid = new OracleXid(0x1234, gtrid, bqual);
      return xid;
    }
  }
```

2 フェーズ・コミットの操作 XA: XA4.java

この例では、分散トランザクションの基本的な 2 フェーズ・コミット機能を示します。

このクラスには、この例で使用するトランザクション ID を形成する `createXid()` メソッドが含まれています。SQL 操作を実行する `doSomeWork1()` および `doSomeWork2()` メソッドも含まれています。

分散トランザクションおよび XA 機能の詳細は、第 15 章「分散トランザクション」を参照してください。

```
/*
    A simple 2 phase XA demo. Both the branches talk to different RMS
    Need 2 java enabled 8.1.6 databases to run this demo.
      -> start-1
      -> start-2
      -> Do some DML on 1
      -> Do some DML on 2
      -> end 1
      -> end 2
      -> prepare-1
      -> prepare-2
      -> commit-1
      -> commit-2
    Please change the URL2 before running this.
 */

// You need to import the java.sql package to use JDBC
import java.sql.*;
import javax.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;
import oracle.jdbc.xa.OracleXid;
import oracle.jdbc.xa.OracleXAException;
import oracle.jdbc.xa.client.*;
import javax.transaction.xa.*;

class XA4
{
  public static void main (String args [])
       throws SQLException
  {
```

```
try
{
    String URL1 = "jdbc:oracle:oci8:@";
    String URL2 =
                "jdbc:oracle:thin:@
                (description=(address=(host=dlsun991)(protocol=tcp)
                (port=5521))(connect_data=(sid=rdbms2)))";

    DriverManager.registerDriver(new OracleDriver());

    // You can put a database name after the @ sign in the connection URL.
    Connection conna =
      DriverManager.getConnection (URL1, "scott", "tiger");

    // Prepare a statement to create the table
    Statement stmta = conna.createStatement ();

    Connection connb =
      DriverManager.getConnection (URL2, "scott", "tiger");

    // Prepare a statement to create the table
    Statement stmtb = connb.createStatement ();

    try
    {
      // Drop the test table
      stmta.execute ("drop table my_table");
    }
    catch (SQLException e)
    {
      // Ignore an error here
    }

    try
    {
      // Create a test table
      stmta.execute ("create table my_table (col1 int)");
    }
    catch (SQLException e)
    {
      // Ignore an error here too
    }
```

```
try
{
  // Drop the test table
  stmtb.execute ("drop table my_tab");
}
catch (SQLException e)
{
  // Ignore an error here
}

try
{
  // Create a test table
  stmtb.execute ("create table my_tab (col1 char(30))");
}
catch (SQLException e)
{
  // Ignore an error here too
}

// Create a XADataSource instance
OracleXADataSource oxds1 = new OracleXADataSource();
oxds1.setURL("jdbc:oracle:oci8:@");
oxds1.setUser("scott");
oxds1.setPassword("tiger");

OracleXADataSource oxds2 = new OracleXADataSource();

oxds2.setURL
      ("jdbc:oracle:thin:@(description=(address=(host=dlsun991)
      (protocol=tcp)(port=5521))(connect_data=(sid=rdbms2)))");
oxds2.setUser("scott");
oxds2.setPassword("tiger");

// Get a XA connection to the underlying data source
XAConnection pc1  = oxds1.getXAConnection();

// We can use the same data source
XAConnection pc2  = oxds2.getXAConnection();

// Get the Physical Connections
Connection conn1 = pc1.getConnection();
Connection conn2 = pc2.getConnection();

// Get the XA Resources
XAResource oxar1 = pc1.getXAResource();
XAResource oxar2 = pc2.getXAResource();
```

```
// Create the Xids With the Same Global Ids
Xid xid1 = createXid(1);
Xid xid2 = createXid(2);

// Start the Resources
oxar1.start (xid1, XAResource.TMNOFLAGS);
oxar2.start (xid2, XAResource.TMNOFLAGS);

// Do  something with conn1 and conn2
doSomeWork1 (conn1);
doSomeWork2 (conn2);

// END both the branches -- THIS IS MUST
oxar1.end(xid1, XAResource.TMSUCCESS);
oxar2.end(xid2, XAResource.TMSUCCESS);

// Prepare the RMs
int prp1 =  oxar1.prepare (xid1);
int prp2 =  oxar2.prepare (xid2);

System.out.println("Return value of prepare 1 is " + prp1);
System.out.println("Return value of prepare 2 is " + prp2);

boolean do_commit = true;

if (!((prp1 == XAResource.XA_OK) || (prp1 == XAResource.XA_RDONLY)))
   do_commit = false;

if (!((prp2 == XAResource.XA_OK) || (prp2 == XAResource.XA_RDONLY)))
   do_commit = false;

System.out.println("do_commit is " + do_commit);
System.out.println("Is oxar1 same as oxar2 ? " + oxar1.isSameRM(oxar2));

if (prp1 == XAResource.XA_OK)
  if (do_commit)
     oxar1.commit (xid1, false);
  else
     oxar1.rollback (xid1);

if (prp2 == XAResource.XA_OK)
  if (do_commit)
     oxar2.commit (xid2, false);
  else
     oxar2.rollback (xid2);
```

```
 // Close connections
conn1.close();
conn1 = null;
conn2.close();
conn2 = null;

pc1.close();
pc1 = null;
pc2.close();
pc2 = null;

ResultSet rset = stmta.executeQuery ("select col1 from my_table");
while (rset.next())
  System.out.println("Col1 is " + rset.getInt(1));

rset.close();
rset = null;

rset = stmtb.executeQuery ("select col1 from my_tab");
while (rset.next())
  System.out.println("Col1 is " + rset.getString(1));

rset.close();
rset = null;

stmta.close();
stmta = null;
stmtb.close();
stmtb = null;

conna.close();
conna = null;
connb.close();
connb = null;
```

```
  } catch (SQLException sqe)
  {
    sqe.printStackTrace();
  } catch (XAException xae)
  {
    if (xae instanceof OracleXAException) {
      System.out.println("XA Error is " +
                  ((OracleXAException)xae).getXAError());
      System.out.println("SQL Error is " +
                  ((OracleXAException)xae).getOracleError());
    }
  }
}

static Xid createXid(int bids)
  throws XAException
{
  byte[] gid = new byte[1]; gid[0]= (byte) 9;
  byte[] bid = new byte[1]; bid[0]= (byte) bids;
  byte[] gtrid = new byte[64];
  byte[] bqual = new byte[64];
  System.arraycopy (gid, 0, gtrid, 0, 1);
  System.arraycopy (bid, 0, bqual, 0, 1);
  Xid xid = new OracleXid(0x1234, gtrid, bqual);
  return xid;
}

private static void doSomeWork1 (Connection conn)
 throws SQLException
{
  // Create a Statement
  Statement stmt = conn.createStatement ();

  int cnt = stmt.executeUpdate ("insert into my_table values (4321)");

  System.out.println("No of rows Affected " + cnt);

  stmt.close();
  stmt = null;
}

private static void doSomeWork2 (Connection conn)
  throws SQLException
{
  // Create a Statement
  Statement stmt = conn.createStatement ();
```

```
    int cnt = stmt.executeUpdate ("insert into my_tab values ('test')");

    System.out.println("No of rows Affected " + cnt);

    stmt.close();
    stmt = null;
  }

}
```

HeteroRM XA: XA6.java

次のコードは、HeteroRM XA 機能の使用方法を示します。コミット、ロールバック、一時停止および再開など、JDBC XA を通じて使用可能な XA 固有の機能はすべて、HeteroRM XA でも使用可能です。

HeteroRM XA の詳細は、16-17 ページの「OCI HeteroRM XA」を参照してください。

```
import java.sql.*;
import javax.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;
import oracle.jdbc.xa.OracleXid;
import oracle.jdbc.xa.OracleXAException;
import oracle.jdbc.xa.client.*;
import javax.transaction.xa.*;

class XA6
{
  public static void main (String args [])
       throws SQLException
  {

    try
    {
        DriverManager.registerDriver(new OracleDriver());

        String url = "jdbc:oracle:oci8:@";
        try {
          String url1 = System.getProperty("JDBC_URL");
          if (url1 != null)
            url = url1;
        } catch (Exception e) {
          // If there is any security exception, ignore it
          // and use the default
        }
```

```
// Connect to the database
Connection conn =
  DriverManager.getConnection (url, "scott", "tiger");

// Prepare a statement to create the table
Statement stmt = conn.createStatement ();

try
{
  // Drop the test table
  stmt.execute ("drop table my_table");
}
catch (SQLException e)
{
  // Ignore an error here
}

try
{
  // Create a test table
  stmt.execute ("create table my_table (col1 int)");
}
catch (SQLException e)
{
  // Ignore an error here too
}

// Create a XADataSource instance
OracleXADataSource oxds = new OracleXADataSource();
oxds.setURL(url);

// Set the nativeXA property to use HeteroRM XA feature
oxds.setNativeXA(true);

// Set the tnsEntry property to an older DB as required
oxds.setTNSEntryName("ora805");

oxds.setUser("scott");
oxds.setPassword("tiger");

// get a XA connection
XAConnection pc  = oxds.getXAConnection();
// Get a logical connection
Connection conn1 = pc.getConnection();
```

```
// Get XA resource handle
XAResource oxar = pc.getXAResource();

Xid xid = createXid();

// Start a transaction branch
oxar.start (xid, XAResource.TMNOFLAGS);

// Create a Statement
Statement stmt1 = conn1.createStatement ();

// Do some DML
stmt1.executeUpdate ("insert into my_table values (7321)");

// Close the Statement
stmt1.close();
stmt1 = null;

// End the branch
oxar.end(xid, XAResource.TMSUCCESS);

// Do a 1 phase commit
oxar.commit (xid, true);

// Close the connection
conn1.close();
conn1 = null;

// close the XA connection
pc.close();
pc = null;

ResultSet rset = stmt.executeQuery ("select col1 from my_table");
while (rset.next())
  System.out.println("Col1 is " + rset.getInt(1));

rset.close();
rset = null;

stmt.close();
stmt = null;

conn.close();
conn = null;
```

```
    } catch (SQLException sqe)              // check for SQLExceptions
    {
      sqe.printStackTrace();
    } catch (XAException xae)               // check for XAExceptions
    {
      xae.printStackTrace();
    }
  }

  static Xid createXid()
    throws XAException
  {
    byte[] gid = new byte[1]; gid[0]= (byte) 1;
    byte[] bid = new byte[1]; bid[0]= (byte) 1;
    byte[] gtrid = new byte[64];
    byte[] bqual = new byte[64];
    System.arraycopy (gid, 0, gtrid, 0, 1);
    System.arraycopy (bid, 0, bqual, 0, 1);
    Xid xid = new OracleXid(0x1234, gtrid, bqual);
    return xid;
  }
}
```

サンプル・アプレット

この項では、データベースから Hello World および日付を選択する単純なアプレットを例にして、Oracle JDBC Thin ドライバの使用方法を示します。HTML ページとアプレット・コードの両方を示します。JDBC アプレットは、一般的なアプレットと同じように、標準 Web サーバーを使用して配置でき、標準ブラウザから実行できます。

アプレットで JDBC を使用する方法の詳細は、18-16 ページの「アプレット内の JDBC」を参照してください。

この例では、Web サーバーとデータベースは同じホストに置く必要があります。この例は、署名付きアプレットではなく、Oracle Connection Manager を使用しません。詳細は、18-18 ページの「Web サーバーとは異なるホスト上のデータベースへの接続」を参照してください。

HTML ページ : JdbcApplet.htm

アプレットのユーザー・インタフェースになる HTML コードです。

```
<html>
<head>
<title>JDBC applet</title>
</head>
<body>

<h1>JDBC applet</h1>

This page contains an example of an applet that uses the Thin JDBC
driver to connect to Oracle.<p>

The source code for the applet is in <a
href="JdbcApplet.java">JdbcApplet.java</a>.  Please check carefully
the driver class name and the connect string in the code.<p>

The Applet tag in this file contains a CODEBASE entry that must be set
to point to a directory containing the Java classes from the Thin JDBC
distribution *and* the compiled JdbcApplet.class.<p>

As distributed it will *not* work because the classes*.zip files are not
in this directory.<p>

<hr>
<applet codebase="." archive="classes111.zip"
code="JdbcApplet" width=500 height=200>
</applet>
<hr>
```

アプレット・コード : JdbcApplet.java

アプレットのソース・コードです。

```
/*
 * This sample applet just selects 'Hello World' and the date from the database
 */

// Import the JDBC classes
import java.sql.*;

// Import the java classes used in applets
import java.awt.*;
import java.io.*;
import java.util.*;

public class JdbcApplet extends java.applet.Applet
{

  // The connect string
  static final String connect_string =
                 "jdbc:oracle:thin:scott/tiger@langer:5521:rdbms";

  /* This is the kind of string you would use if going through the
   * Oracle connection manager which lets you run the database on a
   * different host than the Web Server. See the Oracle Net Administrator's
   * Guide for more information.
   * static final String connect_string = "jdbc:oracle:thin:scott/tiger@
   *                (description=(address_list=(address=(protocol=tcp)
   *                (host=dlsun511)(port=1610))(address=(protocol=tcp)
   *                (host=pkrishna-pc2)(port=1521)))
   *                (source_route=yes)(connect_data=(sid=orcl)))";
   */

  // The query we will execute
  static final String query = "select 'Hello JDBC: ' || sysdate from dual";


  // The button to push for executing the query
  Button execute_button;

  // The place where to dump the query result
  TextArea output;

  // The connection to the database
  Connection conn;
```

```
// Create the User Interface
public void init ()
{
  this.setLayout (new BorderLayout ());
  Panel p = new Panel ();
  p.setLayout (new FlowLayout (FlowLayout.LEFT));
  execute_button = new Button ("Hello JDBC");
  p.add (execute_button);
  this.add ("North", p);
  output = new TextArea (10, 60);
  this.add ("Center", output);
}

// Do the work
public boolean action (Event ev, Object arg)
{
  if (ev.target == execute_button)
  {
    try
    {

    // See if we need to open the connection to the database
    if (conn == null)
    {
      // Load the JDBC driver
      DriverManager.registerDriver (new oracle.jdbc.OracleDriver());

      // Connect to the databse
      output.appendText ("Connecting to " + connect_string + "\n");
      conn = DriverManager.getConnection (connect_string);
      output.appendText ("Connected\n");
    }

    // Create a statement
    Statement stmt = conn.createStatement ();

    // Execute the query
    output.appendText ("Executing query " + query + "\n");
    ResultSet rset = stmt.executeQuery (query);

    // Dump the result
    while (rset.next ())
      output.appendText (rset.getString (1) + "\n");
```

```
      // We're done
      output.appendText ("done.\n");
      }
      catch (Exception e)
      {
      // Oops
      output.appendText (e.getMessage () + "\n");
      }
      return true;
    }
    else
      return false;
  }
}
```

JDBC サンプル・コードと SQLJ サンプル・コード

この項では、Oracle CustomDatum 機能を使用する、同一のサンプル・コードの 2 つのバージョンを並べて比較します。一方のバージョンは JDBC で記述されており、もう一方は SQLJ で記述されています。この項の目的は、コードを記述する際の、SQLJ と JDBC との要求事項の違いを指摘することです。

サンプルでは、2 つのメソッドが定義されています。getEmployeeAddress() は、表の中で選択を行い、従業員番号に基づいて従業員の住所を戻します。updateAddress() は、取り出された住所を取り、ストアド・プロシージャをコールし、更新された住所をデータベースに戻します。

どちらのバージョンのサンプル・コードにも、次の前提事項があります。

- ObjectDemo.sql スクリプト（後で説明します）の実行により、必要なデータベース・エンティティはすでに作成されています。
- PL/SQL ストアド・ファンクション UPDATE_ADDRESS は、指定された住所が存在する場合にそれを更新します。
- 接続オブジェクト（JDBC の場合）およびデフォルト接続コンテキスト（SQLJ の場合）が、コール側によりすでに作成されています。
- 例外は、コール側が処理します。
- updateAddress メソッドに渡されるアドレス引数（addr）の値は、NULL の場合もあります。

注意： コードは、JDBC と SQLJ のどちらのバージョンも、部分的なサンプルです。これらのコードのみを独立して実行できません。

表とオブジェクト作成用の SQL プログラム

次に、2 つのサンプル・コードが参照する表とオブジェクトを作成する ObjectDemo.sql スクリプトを示します。ObjectDemo.sql スクリプトは、PERSON オブジェクト、ADDRESS オブジェクト、PERSON オブジェクトの型付けされた表（PERSONS）および従業員データ用のリレーショナル表（EMPLOYEES）を生成します。

```
/*** Using objects in SQLJ ***/
SET ECHO ON;
/**

/*** Clean up ***/
DROP TABLE EMPLOYEES
/
DROP TABLE PERSONS
/
DROP TYPE PERSON FORCE
/
DROP TYPE ADDRESS FORCE
/

/*** Create an address object ***/
CREATE TYPE address AS OBJECT
(
  street        VARCHAR(60),
  city          VARCHAR(30),
  state         CHAR(2),
  zip_code      CHAR(5)
)
/

/*** Create a person object containing an embedded Address object ***/
CREATE TYPE person AS OBJECT
(
  name    VARCHAR(30),
  ssn     NUMBER,
  addr    address
)
/

/*** Create a typed table for person objects ***/
CREATE TABLE persons OF person
/

/*** Create a relational table with two columns that are REFs
     to person objects, as well as a column which is an Address object.***/
```

```
CREATE TABLE  employees
( empnumber            INTEGER PRIMARY KEY,
  person_data     REF  person,
  manager         REF  person,
  office_addr          address,
  salary               NUMBER
)

/
/*** insert code for UPDATE_ADDRESS stored procedure here
/

/*** Now let's put in some sample data
     Insert 2 objects into the persons typed table ***/

INSERT INTO persons VALUES (
            person('Wolfgang Amadeus Mozart', 123456,
            address('Am Berg 100', 'Salzburg', 'AU','10424')))
/
INSERT INTO persons VALUES (
            person('Ludwig van Beethoven', 234567,
            address('Rheinallee', 'Bonn', 'DE', '69234')))
/

/** Put a row in the employees table **/

INSERT INTO employees (empnumber, office_addr, salary) " +
            " VALUES (1001, address('500 Oracle Parkway', " +
            " 'Redwood City', 'CA', '94065'), 50000)
/

/** Set the manager and person REFs for the employee **/

UPDATE employees
  SET manager =
    (SELECT REF(p) FROM persons p WHERE p.name = 'Wolfgang Amadeus Mozart')
/

UPDATE employees
  SET person_data =
    (SELECT REF(p) FROM persons p WHERE p.name = 'Ludwig van Beethoven')
/

COMMIT
/
QUIT
```

JDBC バージョンのサンプル・コード

次に、JDBC バージョンのサンプル・コードを示します。このコードにより定義されたメソッドは、データベースから従業員の住所を取り出し、住所を更新してデータベースに戻します。コメント行の TO DO は、その部分にコードを追加してサンプル・コードの機能を拡張できることを示します。

```
import java.sql.*;
import oracle.jdbc.*;

/**
  This is what we have to do in JDBC
  **/
public class SimpleDemoJDBC                                    // line 7
{

//TO DO: make a main that calls this

  public Address getEmployeeAddress(int empno, Connection conn)
    throws SQLException                                        // line 13
  {
    Address addr;
    PreparedStatement pstmt =                                  // line 16
      conn.prepareStatement("SELECT office_addr FROM employees" +
       " WHERE empnumber = ?");
    pstmt.setInt(1, empno);
    OracleResultSet rs = (OracleResultSet)pstmt.executeQuery();
    rs.next();                                                 // line 21
     //TO DO: what if false (result set contains no data)?
    addr = (Address)rs.getCustomDatum(1, Address.getFactory());
    //TO DO: what if additional rows?
    rs.close();                                                // line 25
    pstmt.close();
    return addr;                                               // line 27

  }

  public Address updateAddress(Address addr, Connection conn)
    throws SQLException                                        // line 30

  {
    OracleCallableStatement cstmt = (OracleCallableStatement)
      conn.prepareCall("{ ? = call UPDATE_ADDRESS(?) }");    //line 34
    cstmt.registerOutParameter(1, Address._SQL_TYPECODE, Address._SQL_NAME);
                                                               // line 36
```

```
    if (addr == null) {
      cstmt.setNull(2, Address._SQL_TYPECODE, Address._SQL_NAME);
    } else {
      cstmt.setCustomDatum(2, addr);
    }

    cstmt.executeUpdate();                                    // line 43
    addr = (Address)cstmt.getCustomDatum(1, Address.getFactory());
    cstmt.close();                                            // line 45
    return addr;
  }
}
```

Line 12: `getEmployeeAddress()` メソッド定義で、接続オブジェクトを明示的にメソッド定義に渡す必要があります。

Lines 16-20: 従業員番号に基づき、`EMPLOYEES` 表から従業員の住所を選択する文を準備します。従業員番号は、マーカー変数で表現されます。マーカー変数は `setInt()` メソッドを使用して設定します。プリコンパイルされた SQL 文は、20-121 ページの「表とオブジェクト作成用の SQL プログラム」で使用した `INTO` 構文を認識しないため、住所（`addr`）変数を移入するコードを独自に提供する必要があります。プリコンパイルされた SQL 文はカスタム・オブジェクトを戻すため、出力を Oracle 結果セットにキャストします。

Lines 21-23: Oracle 結果セットには `Address` 型のカスタム・オブジェクトが含まれるため、`getCustomDatum()` メソッドを使用してカスタム・オブジェクトを取り出します。（`Address` オブジェクトは JPublisher で作成できます。）`getCustomDatum()` メソッドを使用する場合、静的ファクトリ・メソッド `Address.getFactory()` を使用して、`Address` オブジェクトのインスタンスをインスタンス化する必要があります。`getCustomDatum()` は `Datum` を戻すため、出力を `Address` オブジェクトにキャストします。

このルーチンは、結果セットが 1 行であることを前提としています。コメント文の `TO DO` は、結果セットに行が含まれていない場合または 2 行以上含まれている場合に実行される追加コードを記述する必要があることを示しています。

Lines 25-27: 結果セットとプリコンパイルされた SQL 文オブジェクトをクローズして、`addr` 変数を戻します。

Line 29: `updateAddress()` 定義では、接続オブジェクトと `Address` オブジェクトを明示的に渡す必要があります。

`updateAddress()` メソッドは、住所をデータベースに渡して、住所を更新し、再度フェッチします。実際の住所更新は、`UPDATE_ADDRESS` ストアド・プロシージャにより行われます。（このプロシージャ用のコードはサンプル・コードには含まれていません。）

Line 33-43: 住所オブジェクト（Address）を取得し、それを UPDATE_ADDRESS ストアド・プロシージャに渡す、Oracle コール可能文を準備します。オブジェクトを出力パラメータとして登録するには、オブジェクトの SQL タイプ・コードおよび SQL 型名が必要です。

住所オブジェクト（addr）を入力パラメータとして渡す前に、プログラムは addr が値を持っているか、それとも null かを判断する必要があります。addr の値に基づいて、プログラムは異なる set メソッドをコールします。addr が null の場合、プログラムは setNull() をコールします。値を持つ場合、プログラムは setCustomDatum() をコールします。

Line 44: 返された結果 addr をフェッチします。Oracle コール可能文は Address 型のカスタム・オブジェクトを戻すため、getCustomDatum() メソッドを使用してカスタム・オブジェクトを取り出します。（Address オブジェクトは JPublisher で作成できます。）getCustomDatum() メソッドを使用する場合、静的ファクトリ・メソッド Address.getFactory() を使用して、Address オブジェクトのインスタンスをインスタンス化する必要があります。getCustomDatum() は Datum を戻すため、出力を Address オブジェクトにキャストします。

Lines 45、46: Oracle コール可能文をクローズし、addr 変数を戻します。

JDBC バージョンのコーディング要件

JDBC バージョンのサンプル・コードに関する、次のコーディング要件に留意してください。

- getEmployeeAddress() および updateAddress() 定義では、接続オブジェクトを明示的に含める必要があります。
- 長い SQL 文字列は、SQL 連結文字（+）で連結する必要があります。
- リソースは明示的に管理する（結果セットや文オブジェクトのクローズなど）必要があります。
- 必要に応じて、データ型をキャストします。
- 出力パラメータとして登録するファクトリ・オブジェクトの _SQL_TYPECODE および _SQL_NAME を知っている必要があります。
- null データは、明示的に処理する必要があります。
- ホスト変数は、コール可能文およびプリコンパイルされた SQL 文のパラメータ・マーカーで表現する必要があります。

JDBC プログラムのメンテナンス

JDBC プログラムは、メンテナンス面でコストがかさむ可能性があります。たとえば、既出のコード例に、新たに WHERE 句を追加する場合、SELECT 文字列を変更する必要があります。新たなホスト変数を追加する場合は、他のホスト変数の索引値を 1 増加させる必要があります。JDBC プログラムの 1 行に簡単な変更を加えると、プログラムの他のいくつかの部分を変更する必要が生じる場合があります。

SQLJ バージョンのサンプル・コード

次に、SQLJ バージョンのサンプル・コードを示します。このコードにより定義されたメソッドは、データベースから従業員の住所を取り出し、住所を更新してデータベースに戻します。

```
import java.sql.*;

/**
  This is what we have to do in SQLJ
  **/
public class SimpleDemoSQLJ                                    // line 6
{
  //TO DO: make a main that calls this?

  public Address getEmployeeAddress(int empno)                 // line 10
    throws SQLException
  {
    Address addr;                                              // line 13
    #sql { SELECT office_addr INTO :addr FROM employees
           WHERE empnumber = :empno };
    return addr;
  }
                                                               // line 18
  public Address updateAddress(Address addr)
    throws SQLException
  {
    #sql addr = { VALUES(UPDATE_ADDRESS(:addr)) };             // line 23
    return addr;
  }
}
```

Line 10: `getEmployeeAddress()` メソッドには、接続オブジェクトは不要です。SQLJ は、デフォルト接続コンテキストのインスタンスを使用します。これは、アプリケーション内ですでに定義されています。

Lines 13-15: `getEmployeeAddress()` メソッドは、従業員番号に基づいて従業員の住所を取り出します。従業員番号が `getEmployeeAddress()` に渡された従業員番号（`empno`）に一致すると、標準 SQLJ `SELECT INTO` 構文を使用して、従業員表から従業員の住所を選択します。この動作には、データを受け取る `Address` オブジェクト（`addr`）の宣言が必要です。`empno` および `addr` 変数が、入力ホスト変数として使用されます。（ホスト変数は、バインド変数として参照されることもあります。）

Line 16: `getEmployeeAddress()` メソッドは、`addr` オブジェクトを戻します。

Line 19: updateAddress() メソッドも、デフォルト接続コンテキストのインスタンスを使用します。

Lines 19-23: 住所は、updateAddress() メソッドに渡されます。このメソッドは住所をデータベースに渡します。データベースは住所を更新し、戻します。実際の住所更新は、UPDATE_ADDRESS ストアド・ファンクション（このプロシージャ用のコードはサンプル・コードには含まれていません）により行われます。標準 SQLJ ファンクション・コール構文を使用して、UPDATE_ADDRESS により出力された住所オブジェクト（addr）を受け取ります。

Line 24: updateAddress() メソッドは、addr オブジェクトを戻します。

SQLJ バージョンのコーディング要件

SQLJ バージョンのサンプル・コードでは、次のコーディング要件に注意してください。

- 明示的な接続は不要です。デフォルト接続コンテキストは、アプリケーションであらかじめ定義されます。
- データ型のキャストは必要ありません。
- SQLJ では、_SQL_TYPECODE、_SQL_NAME またはファクトリについて知っている必要はありません。
- NULL データは、暗黙的に処理されます。
- リソース管理用の明示的なコード（文や結果セットのクローズなど）は必要ありません。
- JDBC がパラメータ・マーカーを使用するのに対し、SQLJ はホスト変数を埋め込みます。
- 長い SQL 文での文字列の連結は、必要ありません。
- 出力パラメータを登録する必要はありません。
- SQLJ 構文の方が簡単です。たとえば、SELECT INTO 構文がサポートされています。OBDC スタイルのエスケープは使用されません。

21

リファレンス情報

この章では、JDBC リファレンス詳細情報について説明します。次の項目が含まれます。

- 有効な SQL-JDBC データ型マッピング
- サポートされている SQL および PL/SQL データ型
- 埋込み SQL92 構文
- Oracle JDBC の注意および制限事項
- 関連情報

有効な SQL-JDBC データ型マッピング

第 3 章の表 3-2 は、Oracle JDBC ドライバでサポートされている Java クラスおよび SQL データ型のデフォルトのマッピングのリストです。表 3-2 の **JDBC データ型、標準 Java データ型**および **SQL データ型**の列の内容と、次の表 21-1 の内容を比較してください。

表 21-1 は、特定の SQL データ型のマッピング先として有効な、すべての Java 型のリストです。Oracle JDBC ドライバでは、これらの非デフォルト・マッピングをサポートしています。たとえば、SQL CHAR データを `oracle.sql.CHAR` オブジェクトとしてインスタンス化するには、`getCHAR()` メソッドを使用します。`java.math.BigDecimal` オブジェクトとしてインスタンス化するには、`getBigDecimal()` メソッドを使用します。

注意：

- 次の SQL データ型については、`oracle.sql.ORAData` または `oracle.sql.Datum` を Java 型としてインスタンス化できます。
- `oracle.sql.ORAData` をイタリック体で示してあるクラスは、JPublisher での生成が可能です。

表 21-1 有効な SQL データ型 ― Java クラス・マッピング

SQL データ型	実現可能な Java 型
CHAR、VARCHAR2、LONG	oracle.sql.CHAR
	java.lang.String
	java.sql.Date
	java.sql.Time
	java.sql.Timestamp
	java.lang.Byte
	java.lang.Short
	java.lang.Integer
	java.lang.Long
	java.lang.Float
	java.lang.Double
	java.math.BigDecimal
	byte, short, int, long, float, double

表 21-1 有効な SQL データ型 ー Java クラス・マッピング （続き）

SQL データ型	実現可能な Java 型
DATE	oracle.sql.DATE
	java.sql.Date
	java.sql.Time
	java.sql.Timestamp
	java.lang.String
NUMBER	oracle.sql.NUMBER
	java.lang.Byte
	java.lang.Short
	java.lang.Integer
	java.lang.Long
	java.lang.Float
	java.lang.Double
	java.math.BigDecimal
	byte, short, int, long, float, double
OPAQUE	oracle.sql.OPAQUE
RAW、LONG RAW	oracle.sql.RAW
	byte[]
ROWID	oracle.sql.CHAR
	oracle.sql.ROWID
	java.lang.String
BFILE	oracle.sql.BFILE
BLOB	oracle.sql.BLOB
	java.sql.Blob（JDK 1.1.x では oracle.jdbc2.Blob）
CLOB	oracle.sql.CLOB
	java.sql.Clob（JDK 1.1.x では oracle.jdbc2.Clob）

表 21-1 有効な SQL データ型 ― Java クラス・マッピング （続き）

SQL データ型	実現可能な Java 型
オブジェクト型および SQLJ 型	oracle.sql.STRUCT
	java.sql.Struct（JDK 1.1.x では oracle.jdbc2.Struct）
	java.sql.SqlData
	oracle.sql.ORAData
参照型	oracle.sql.REF
	java.sql.Ref（JDK 1.1.x では oracle.jdbc2.Ref）
	oracle.sql.ORAData
NESTED TABLE 型および VARRAY 型	oracle.sql.ARRAY
	java.sql.Array（JDK 1.1.x では oracle.jdbc2.Array）
	oracle.sql.ORAData

注意：

- UROWID 型はサポートされていません。
- oracle.sql.Datum は抽象クラスです。oracle.sql.Datum 型のパラメータに渡す値は、基礎となる SQL 型に対応する Java 型にする必要があります。同様に、戻り型が oracle.sql.Datum のメソッドから戻す値は、基礎となる SQL 型に対応する Java 型にする必要があります。
- SQL 形式から Java 形式に変換する必要がない場合は、oracle.sql クラスへのマッピングが適しています。

サポートされている SQL および PL/SQL データ型

この項の表は、SQL と PL/SQL のデータ型、および Oracle JDBC ドライバと SQLJ がこれらのデータ型をサポートしているかどうかのリストです。表 21-2 は、SQL データ型に対する Oracle JDBC ドライバおよび SQLJ のサポート状況のリストです。

表 21-2 SQL データ型に対するサポート

SQL データ型	JDBC ドライバによるサポート	SQLJ によるサポート
BFILE	可	可
BLOB	可	可
CHAR	可	可
CLOB	可	可
DATE	可	可
NCHAR	不可	不可
NCHAR VARYING	不可	不可
NUMBER	可	可
NVARCHAR2	不可	不可
RAW	可	可
REF	可	可
ROWID	可	可
UROWID	不可	不可
VARCHAR2	可	可

表 21-3 は、ANSI でサポートされている SQL データ型に対する Oracle JDBC ドライバおよび SQLJ のサポート状況のリストです。

表 21-3 ANSI-92 SQL データ型に対するサポート

ANSI でサポートされている SQL データ型	JDBC ドライバによるサポート	SQLJ によるサポート
CHARACTER	可	可
DEC	可	可
DECIMAL	可	可

表 21-3 ANSI-92 SQL データ型に対するサポート （続き）

ANSI でサポートされている SQL データ型	JDBC ドライバによるサポート	SQLJ によるサポート
DOUBLE PRECISION	可	可
FLOAT	可	可
INT	可	可
INTEGER	可	可
NATIONAL CHARACTER	不可	不可
NATIONAL CHARACTER VARYING	不可	不可
NATIONAL CHAR	可	可
NATIONAL CHAR VARYING	不可	不可
NCHAR	可	可
NCHAR VARYING	不可	不可
NUMERIC	可	可
REAL	可	可
SMALLINT	可	可
VARCHAR	可	可

表 21-4 は、SQL ユーザー定義型に対する Oracle JDBC ドライバおよび SQLJ のサポート状況のリストです。

表 21-4 SQL ユーザー定義型に対するサポート

SQL ユーザー定義型	JDBC ドライバによるサポート	SQLJ によるサポート
OPAQUE	可	不可
参照型	可	可
SQLJ 型（`JAVA_STRUCT`）	可	不可
オブジェクト型（`JAVA_OBJECT`）	可	可
NESTED TABLE 型および VARRAY 型	可	可

注意： SQLJ 型については、『Information Technology - SQLJ - Part 2：SQL Types using the JAVATM Programming Language』（ANSI NCITS 331.2-2000）を参照してください。

表 21-5 は、PL/SQL データ型に対する Oracle JDBC ドライバおよび SQLJ のサポート状況のリストです。PL/SQL データ型には、次のカテゴリがあります。

- スカラー型
- スカラー文字列型（ブール型および日付データ型を含みます。）
- コンポジット型
- 参照型
- LOB 型

表 21-5 PL/SQL データ型に対するサポート

PL/SQL データ型	JDBC ドライバによるサポート	SQLJ によるサポート
スカラー型		
BINARY INTEGER	可	可
DEC	可	可
DECIMAL	可	可
DOUBLE PRECISION	可	可
FLOAT	可	可
INT	可	可
INTEGER	可	可
NATURAL	可	可
NATURAL*n*	不可	不可
NUMBER	可	可
NUMERIC	可	可
PLS_INTEGER	可	可
POSITIVE	可	可
POSITIVE*n*	不可	不可
REAL	可	可

表 21-5 PL/SQL データ型に対するサポート （続き）

PL/SQL データ型	JDBC ドライバによるサポート	SQLJ によるサポート
SIGNTYPE	可	可
SMALLINT	可	可
スカラー文字列型		
CHAR	可	可
CHARACTER	可	可
LONG	可	可
LONG RAW	可	可
NCHAR	不可	不可
NVARCHAR2	不可	不可
RAW	可	可
ROWID	可	可
STRING	可	可
UROWID	不可	不可
VARCHAR	可	可
VARCHAR2	可	可
BOOLEAN	可	可
DATE	可	可
コンポジット型		
RECORD	不可	不可
TABLE	不可	不可
VARRAY	可	可
参照型		
REF CURSOR 型	可	可
オブジェクト参照型	可	可
LOB 型		
BFILE	可	可
BLOB	可	可

表 21-5 PL/SQL データ型に対するサポート （続き）

PL/SQL データ型	JDBC ドライバによるサポート	SQLJ によるサポート
CLOB	可	可
NCLOB	可	可

注意：

- NATURAL、NATURAL*n*、POSITIVE、POSITIVE*n* および SIGNTYPE 型は、BINARY INTEGER のサブタイプです。
- DEC、DECIMAL、DOUBLE PRECISION、FLOAT、INT、INTEGER、NUMERIC、REAL および SMALLINT 型は、NUMBER のサブタイプです。

埋込み SQL92 構文

Oracle の JDBC ドライバは、いくつかの埋込み SQL92 構文をサポートしています。これは、中括弧内に指定する構文です。現在のサポートは初歩的なものです。この項では、Oracle の JDBC ドライバによって提供される次の SQL92 構文のサポートについて説明します。

- 時刻および日付リテラル
- スカラー関数
- LIKE エスケープ文字
- 外部結合
- ファンクション・コール構文

ドライバのサポートが制限されている場合、これらの項では、選択可能な回避策についても説明します。

エスケープ処理の無効化 SQL92 構文のエスケープ処理は、デフォルトで有効です。そのため、JDBC ドライバは、SQL コードをデータベースに送信する前にエスケープ置換を実行します。通常の Oracle SQL 構文をドライバで使用するには、次の文を使用します。このようにすると、SQL92 構文とエスケープ処理を使用するよりも効率がよくなります。

```
stmt.setEscapeProcessing(false);
```

注意： 通常、プリコンパイルされた SQL 文は setEscapeProcessing() へのコールの前に解析済みのため、エスケープ処理を無効にしても影響はありません。

時刻および日付リテラル

日付、時刻およびタイムスタンプのリテラルに使用する構文は、データベースによって異なります。JDBC では、特定の形式で記述された日付および時刻のみがサポートされています。この項では、SQL 文内で使用する必要のある日付、時刻およびタイムスタンプのリテラルについて説明します。

日付リテラル

JDBC ドライバは、次の形式で記述された SQL 文内の日付リテラルをサポートしています。

```
{d 'yyyy-mm-dd'}
```

yyyy-mm-dd は、年、月および日を表します。例を示します。

```
{d '1995-10-22'}
```

JDBC ドライバは、このエスケープ句を等価の Oracle の表現「22 OCT 1995」に置換します。

次のコードの抜粋には、SQL 文内での日付リテラルの使用例が含まれています。

```
// Connect to the database
// You can put a database name after the @ sign in the connection URL.
Connection conn = DriverManager.getConnection
                  ("jdbc:oracle:oci8:@", "scott", "tiger");

// Create a Statement
Statement stmt = conn.createStatement ();

// Select the ename column from the emp table where the hiredate is Jan-23-1982
ResultSet rset = stmt.executeQuery
                 ("SELECT ename FROM emp WHERE hiredate = {d '1982-01-23'}");

// Iterate through the result and print the employee names
while (rset.next ())
   System.out.println (rset.getString (1));
```

時刻リテラル

JDBC ドライバは、次の形式で記述された SQL 文内の時刻リテラルをサポートしています。

```
{t 'hh:mm:ss'}
```

hh:mm:ss は、時、分および秒を表します。例を示します。

```
{t '05:10:45'}
```

JDBC ドライバは、このエスケープ句を等価の Oracle の表現「05:10:45」に置換します。

次のように時刻が指定されているとします。

```
{t '14:20:50'}
```

サーバーが 24 時間制のクロックを使用しているとすれば、等価の Oracle の表現は「14:20:50」になります。

次のコードの抜粋には、SQL 文内での時刻リテラルの使用例が含まれています。

```
ResultSet rset = stmt.executeQuery
                 ("SELECT ename FROM emp WHERE hiredate = {t '12:00:00'}");
```

タイムスタンプ・リテラル

JDBC ドライバは、次の形式で記述された SQL 文内のタイムスタンプ・リテラルをサポートしています。

```
{ts 'yyyy-mm-dd hh:mm:ss.f...'}
```

`yyyy-mm-dd hh:mm:ss.f...` は、年、月、日、時、分および秒を表します。端数秒の部分（.f...）は省略できます。たとえば、次のようになります。`{ts '1997-11-01 13:22:45'}` は、Oracle 形式では NOV 01 1997 13:22:45 と表されます。

次のコードの抜粋には、SQL 文内でのタイムスタンプ・リテラルの使用例が含まれています。

```
ResultSet rset = stmt.executeQuery
    ("SELECT ename FROM emp WHERE hiredate = {ts '1982-01-23 12:00:00'}");
```

スカラー関数

Oracle JDBC ドライバは、すべてのスカラー関数をサポートしていません。ドライバがサポートする関数を調べるには、Oracle 固有の `oracle.jdbc.OracleDatabaseMetaData` クラスおよび標準 Java の `java.sql.DatabaseMetadata` インタフェースでサポートされている次のメソッドを使用します。

- `getNumericFunctions()`: ドライバによってサポートされている数値演算関数をカンマで区切られたリストで戻します。例 : `ABS(`*`number`*`)`、`COS(`*`float`*`)`、`SQRT(`*`float`*`)`。
- `getStringFunctions()`: ドライバによってサポートされている文字列関数をカンマで区切られたリストで戻します。例 : `ASCII(`*`string`*`)`、`LOCATE(`*`string1`*`、`*`string2`*`、`*`start`*`)`。

- getSystemFunctions(): ドライバによってサポートされているシステム関数をカンマで区切られたリストで戻します。例 : DATABASE()、IFNULL(*expression*、*value*)、USER()。
- getTimeDateFunctions(): ドライバによってサポートされている時刻および日付関数をカンマで区切られたリストで戻します。例 : CURDATE()、DAYOFYEAR(*date*)、HOUR(*time*)。

Oracle の JDBC ドライバは、ファンクション・キーワード 'fn' をサポートしていません。たとえば、次のようにこのキーワードを使用しようとしたとします。

```
{fn concat ("Oracle", "8i") }
```

この場合、エラー「Non supported SQL92 token at position xx:fn」が、Java アプリケーションを実行したときに発生します。回避策は、Oracle SQL 構文を使用することです。

たとえば、次のような埋込み SQL92 構文で fn キーワードを使用するかわりに、

```
Statement stmt = conn.createStatement ();
stmt.executeUpdate("UPDATE emp SET ename = {fn CONCAT('My', 'Name')}");
```

Oracle SQL 構文を使用します。

```
stmt.executeUpdate("UPDATE emp SET ename = CONCAT('My', 'Name')");
```

LIKE エスケープ文字

SQL LIKE 句では、文字「%」および「_」には特別な意味があります（「%」は 0 文字以上の一致、「_」は 1 文字のみの一致に使用します）。これらの文字を文字列内で文字どおりに使用する場合は、その前に特別なエスケープ文字を置きます。たとえば、アンパサンド「&」をエスケープ文字として使用する場合は、SQL 文内では {escape '&'} として識別させます。

```
Statement stmt = conn.createStatement ();

// Select the empno column from the emp table where the ename starts with '_'
ResultSet rset = stmt.executeQuery
          ("SELECT empno FROM emp WHERE ename LIKE '&_%' {ESCAPE '&'}");

// Iterate through the result and print the employee numbers
while (rset.next ())
   System.out.println (rset.getString (1));
```

注意： バックスラッシュ（\）をエスケープ文字として使用する場合は、2 回入力する（\\ とする）必要があります。たとえば、次のようにします。

```
ResultSet rset = stmt.executeQuery("SELECT empno FROM emp
                  WHERE ename LIKE '\\_%' {escape '\\'}");
```

外部結合

Oracle の JDBC ドライバは、外部結合構文 {*oj outer-join*} をサポートしていません。回避策は、Oracle 外部結合構文を使用することです。

次の構文のかわりに、

```
Statement stmt = conn.createStatement ();
ResultSet rset = stmt.executeQuery
     ("SELECT ename, dname
       FROM {OJ dept LEFT OUTER JOIN emp ON dept.deptno = emp.deptno}
       ORDER BY ename");
```

Oracle SQL 構文を使用します。

```
Statement stmt = conn.createStatement ();
ResultSet rset = stmt.executeQuery
     ("SELECT ename, dname
       FROM emp a, dept b WHERE a.deptno = b.deptno(+)
       ORDER BY ename");
```

ファンクション・コール構文

Oracle の JDBC ドライバは、次のプロシージャおよびファンクション・コール構文をサポートしています。

プロシージャ・コール（戻り値なし）：

{ call *procedure_name* (*argument1*, *argument2*,...) }

ファンクション・コール（戻り値あり）：

{ ? = call *procedure_name* (*argument1*, *argument2*,...) }

SQL92 から SQL への構文変換例

SQL92 を標準 SQL 構文に変換する簡単なプログラムを記述できます。次のプログラムは、ファンクション・コール、日付リテラル、時刻リテラルおよびタイムスタンプ・リテラルのための SQL92 の文に対して、それに対応する SQL 構文を出力します。このプログラムでは、`oracle.jdbc.OracleSql` クラスの `parse()` メソッドで変換します。

```
import oracle.jdbc.OracleSql;

public class Foo
{
   public static void main (String args[]) throws Exception
   {
      show ("{call foo(?, ?)}");
      show ("{? = call bar (?, ?)}");
      show ("{d '1998-10-22'}");
      show ("{t '16:22:34'}");
      show ("{ts '1998-10-22 16:22:34'}");
   }

   public static void show (String s) throws Exception
   {
      System.out.println (s + " => " + new OracleSql().parse (s));
   }
}
```

対応する SQL 構文の出力です。

```
{call foo(?, ?)} => BEGIN foo(:1, :2); END;
{? = call bar (?, ?)} => BEGIN :1 := bar (:2, :3); END;
{d '1998-10-22'} => TO_DATE ('1998-10-22', 'YYYY-MM-DD')
{t '16:22:34'} => TO_DATE ('16:22:34', 'HH24:MI:SS')
{ts '1998-10-22 16:22:34'} => TO_DATE ('1998-10-22 16:22:34', 'YYYY-MM-DD
HH24:MI:SS')
```

Oracle JDBC の注意および制限事項

Oracle JDBC 実装には次の制限事項があります。しかし、これらすべての制限は、それほど重要でないか、あるいは容易に回避できます。

CursorName

Oracle JDBC ドライバでは、`getCursorName()` および `setCursorName()` メソッドがサポートされません。これらのメソッドを Oracle 構造体へマップする簡単な方法がないためです。かわりに `ROWID` の使用をお薦めします。ROWID の使用および操作方法の詳細は、5-32 ページの「Oracle ROWID 型」を参照してください。

SQL92 外部結合エスケープ

Oracle JDBC では、SQL92 外部結合エスケープがサポートされません。かわりに、Oracle SQL 構文で（+）を使用してください。SQL92 構文の詳細は、21-9 ページの「埋込み SQL92 構文」を参照してください。

PL/SQL 表、BOOLEAN およびレコード型

Oracle JDBC ドライバでは、PL/SQL レコード、`BOOLEAN` またはスカラー以外の要素型を持つ表の引数コールまたは戻り値をサポートできません。ただし、Oracle JDBC ドライバはスカラー要素型の PL/SQL 索引付き表をサポートします。この項目の詳細は、16-19 ページの「PL/SQL 索引付き表へのアクセス」を参照してください。

PL/SQL レコード、`BOOLEAN`、またはスカラー以外の表タイプの回避策として、データを JDBC によりサポートされる型として処理するラッパー・プロシージャを作成します。たとえば、PL/SQL の BOOLEAN を使用するストアド・プロシージャをラップするには、JDBC から文字または数値を取って、元のプロシージャに `BOOLEAN` として渡すストアド・プロシージャを作成します。出力パラメータの場合、元のプロシージャから `BOOLEAN` 引数を受け取り、`CHAR` または `NUMBER` として JDBC に渡します。同様に、PL/SQL レコードを使用するストアド・プロシージャをラップするには、レコードを独立したコンポーネント（`CHAR` や `NUMBER` など）、または構造化オブジェクト型として処理するストアド・プロシージャを作成します。PL/SQL 表を使用するストアド・プロシージャをラップするには、データをコンポーネントに分割するか、Oracle コレクション型を使用します。

`BOOLEAN` の回避方法の例は、19-9 ページの「PL/SQL ストアド・プロシージャのブール型パラメータ」を参照してください。

IEEE 754 浮動小数点との互換性

Oracle NUMBER 型の算術演算は、浮動小数演算の IEEE 754 規格に準拠していません。このため、Oracle による演算結果と Java による演算結果に小さな誤差が生じることがあります。

Oracle では、10 進数演算と互換性のある形式で数値を格納し、小数点以下 38 桁までの精度を保証しています。このため、0（ゼロ）、無限大の負数、無限大の正数を正確に表現できます。各正数に対して、同じ絶対値の負数も表せます。

10^{-30} ～ $(1 - 10^{-38}) * 10^{126}$ を 38 桁の完全精度で表せます。

DatabaseMetaData コールへの Catalog 引数

特定の DatabaseMetaData メソッドにより catalog パラメータが定義されます。このパラメータは、そのメソッドの選択条件の 1 つです。Oracle には複数カタログはありませんが、複数パッケージはあります。Oracle JDBC ドライバによる catalog 引数の処理方法の詳細は、12-25 ページの「DatabaseMetaData TABLE_REMARKS レポート」を参照してください。

SQLWarning クラス

java.sql.SQLWarning クラスから、データベース・アクセス警告についての情報が提供されます。通常、警告には、警告の説明および警告を識別するコードが含まれます。警告は、その原因になったメソッドをレポートするためにそのオブジェクトに自動的に関連付けられます。Oracle JDBC ドライバでは、通常、SQLWarning がサポートされていません。（例外的に、スクロール可能な結果セット操作は SQL 警告を生成します。しかし、SQLWarning インスタンスはデータベースではなくクライアントで作成されます。）

Oracle JDBC ドライバによるエラー処理の詳細は、3-33 ページの「SQL 例外の処理」を参照してください。

名前によるバインド

名前によるバインドはサポートされていません。特定の環境下では、以前のバージョンの Oracle JDBC ドライバは、名前によって文変数をバインドできました。次の文では、名前付き変数 EmpId が整数 314159 とバインドされます。

```
PreparedStatement p = conn.prepareStatement
                          ("SELECT name FROM emp WHERE id = :EmpId");
p.setInt(1, 314159);
```

この、名前によるバインド機能は、JDBC 仕様の 1.0 にも 2.0 にもありません。また、Oracle もこの機能をサポートしません。JDBC ドライバでは、SQLException または予期しない結果が発生します。

以前のバージョンの Oracle JDBC ドライバでは、JDBC 1.0 の仕様として、実行のコールをまたいでバインド値が保持されませんでした。現在のバージョンではバインド値が保持されます。たとえば、次のようになります。

```
PreparedStatement p = conn.prepareStatement
                      ("SELECT name FROM emp WHERE id = :? AND dept = :?");
p.setInt(1, 314159);
p.setString(2, "SALES");
ResultSet r1 = p.execute();
p.setInt(1, 425260);
ResultSet r2 = p.execute();
```

以前のバージョンでは、2 番目の引数にバインドされる値がないため、2 回目の `execute()` コールで `SQLException` が発生していました。今回のリリースでは、2 回目の実行により正しい値が返されるため、2 番目の引数のバインドが文字列 "`SALES`" に保持されます。

保持されたバインド値がストリームの場合は、Oracle JDBC ドライバではストリームをリセットしません。アプリケーション・コードによってストリームのリセット、再配置または変更が行われない限り、後続の実行のコールにより引数値として `NULL` が送られます。

関連情報

この項では、JDBC プログラマにとって役に立つ情報が掲載されている Web サイトのリストを示します。これらのサイトの多くは、このマニュアルの他の項で参照されています。このリストには、Oracle JDBC ドライバ、Oracle SQLJ、Java テクノロジ、Java Development Kit API（バージョン 1.2.x および 1.1.x）、Java Security API のリファレンスおよび署名付きアプレットの作成に役立つリソースがあります。

Oracle JDBC および SQLJ

Oracle JDBC ドライバのホーム・ページ（オラクル社）

`http://www.oracle.com/java/jdbc`

Oracle SQLJ のホーム・ページ（オラクル社）

`http://www.oracle.com/java/sqlj`

Java テクノロジ

Java テクノロジのホーム・ページ （Sun 社）

`http://www.javasoft.com`

Java Development Kit（JDK1.2.x および 1.1.x）（Sun 社）

`http://java.sun.com/products/jdk`

A

行セット

この付録には、次の項目が含まれます。

- 行セットのセットアップおよび構成
- 行セットのランタイム・プロパティ
- 行セット・リスナー
- 行の横断
- キャッシュされた行セット
- JDBC 行セット

概要

行セットとは、一連の行をカプセル化するオブジェクトです。これらの行には javax.sql.RowSet インタフェースを使用してアクセスできます。このインタフェースは、JavaBean のような開発コンポーネント・モデルをサポートし、JavaSoft の JDBC オプション・パッケージに含まれています。

JavaSoft では、次の 3 種類の行セットをサポートしています。

- キャッシュされた行セット
- JDBC 行セット
- Web 行セット

> **注意：** Oracle ではキャッシュされた行セットおよび JDBC 行セットを実装しますが、Web 行セットは実装しません。

行セットの実装はすべて、oracle.jdbc.rowset パッケージ内にあります。Web 行セットは準接続行セットです。接続がオープンしているサーブレットがあり、WebRowSet インタフェースによってユーザー・コールが HTTP を経由したサーブレットへの適切な要求に変換されます。HTTP 実装のみを含む Thin クライアントを対象にしています。

> **注意：** 行セットは、JDK 1.2 以上でのみ使用できます。

RowSet インタフェースは、共有可能な一連のプロパティを提供します。これらのプロパティにより、1 つのインタフェースを通じてデータベースのデータにアクセスすることが可能になります。このインタフェースは、JavaBean のコアを形成するプロパティおよびイベントをサポートします。接続文字列、ユーザー名、パスワード、接続のタイプ、問合せ文字列をはじめ、問合せに渡されるパラメータなどの様々なプロパティがあります。次のコードは、簡単な問合せを実行します。

```
...
rowset.setUrl ("jdbc:oracle:oci8:@");
rowset.setUsername ("SCOTT");
rowset.setPassword ("TIGER");
rowset.setCommand (
  "SELECT empno, ename, sal FROM emp WHERE empno = ?");

// empno of employee name "KING"
rowset.setInt (1, 7839);
...
```

前述の例では、従業員名や給与を取り出す問合せに必要な URL、ユーザー名、パスワード、SQL 問合せおよびバインド・パラメータがコマンド・プロパティとして設定されています。また、行セットには empno が 7389、従業員名が KING である従業員の empno、ename および sal が含まれます

行セットのセットアップおよび構成

行セット機能のクラスは、ocrs12.zip という別のアーカイブに含まれています。このファイルは、$ORACLE_HOME/jdbc ディレクトリに格納されています。 行セットを使用するには、このアーカイブを CLASSPATH に含める必要があります。

UNIX（sh）の場合は、次のコマンドを使用します。

```
CLASSPATH=$CLASSPATH:$ORACLE_HOME/jdbc/lib/ocrs12.zip
export CLASSPATH
```

Windows 2000/NT/98/95 の場合は、次のコマンドを使用します。

```
set CLASSPATH=%CLASSPATH%;%ORACLE_HOME%¥jdbc¥lib¥ocrs12.zip
```

また、JDeveloper などの IDE を使用する場合は、アーカイブをプロジェクト・プロパティに設定します。

Oracle 行セット・インタフェースは、oracle.jdbc.rowset パッケージに実装されます。Oracle 行セット実装を使用するには、このパッケージをインポートしてください。

OracleCachedRowSet および OracleJDBCRowSet クラスは、java.sql.ResultSet を拡張する javax.sql.RowSet インタフェースを実装します。行セットは結果セットのインタフェースを提供するのみでなく、java.sql.Connection および java.sql.PreparedStatement インタフェースの一部のプロパティも提供します。接続文およびプリコンパイルされた文は完全にこのインタフェースによって抽象化されます。CachedRowSet はシリアル化可能です。このクラスは、java.io.Serializable インタフェースを実装します。これにより、ネットワークまたは他の JVM セッションを通じて OracleCachedRowSet クラスを移動することが可能になります。

また、oracle.jdbc.rowset.OracleRowSetListenerAdapter クラスも使用できます。

行セットのランタイム・プロパティ

一般に、開発時には行セットにアプリケーションの静的プロパティを設定し、残りの（実行時によって変わる）動的なプロパティは実行時に設定することができます。静的プロパティには、接続 URL、ユーザー名、パスワード、接続タイプ、行セットの並行性タイプまたは問合せそのものが含まれます。問合せのバインド変数などのランタイム・プロパティは実行時にバインドできます。問合せそのものが動的プロパティとなる使用例も一般的です。

行セット・リスナー

行セット機能では、RowSet オブジェクトに複数のリスナーを登録できます。リスナーの登録には addRowSetListener() メソッドを使用し、登録を解除するには removeRowSetListener() メソッドを使用します。リスナーは、javax.sql.RowSetListener インタフェースを実装し、行セット・リスナーとして登録する必要があります。RowSet インタフェースは、次の 3 つのタイプのイベントをサポートしています。

1. cursorMoved イベント : カーソルが移動するたびに生成されます。カーソルは、next() または previous() メソッドがコールされると移動します。
2. rowChanged イベント : 行セットに新しい行が挿入されるか、行セットの行が更新されるか、または行セットから行が削除されると生成されます。
3. rowsetChanged イベント : 行セット全体が作成または変更されると、生成されます。

次のコードは、行セット・リスナーを登録します。

```
MyRowSetListener rowsetListener =
  new MyRowSetListener ();
// adding a rowset listener.
rowset.addRowSetListener (rowsetListener);


// implementation of a rowset listener
public class MyRowSetListener implements RowSetListener
{
  public void cursorMoved(RowSetEvent event)
  {
   // action on cursor movement
  }

  public void rowChanged(RowSetEvent event)
  {
    // action on change of row
  }
```

```
  public void rowSetChanged(RowSetEvent event)
  {
    // action on changing of rowset
  }
}// end of class MyRowSetListener
```

少数のイベントのみを処理するアプリケーションの場合には、OracleRowSetAdapter クラスを使用して必要なイベントのみを実装できます。このクラスは、すべてのイベント処理メソッドについて空の実装を持つ抽象クラスです。

次のコードでは、rowSetChanged イベントのみが処理されます。この場合、アプリケーションは他のイベントを処理しません。

```
rowset.addRowSetListener (new OracleRowSetAdapter ()
  {
     public void rowSetChanged(RowSetEvent event)
     {
       // your action for rowsetChanged
     }
  }
 );
```

行の横断

RowSet インタフェースは、行を横断するための様々なメソッドを提供します。これらのメソッドは、java.sql.ResultSet インタフェースから直接継承されます。RowSet インタフェースは、データを取得および更新するための ResultSet インタフェースとして使用できます。結果セットの実装でスクロールおよび更新可能な結果セットが提供されない場合には、RowSet インタフェースにより、オプションでこれらの結果セットを実装することも可能です。

注意： java.sql.ResultSet インタフェースのスクロール可能なプロパティは、ResultSet の Oracle 実装でも提供されます。

キャッシュされた行セット

キャッシュされた行セットとは、行がキャッシュされ、データベースとのアクティブ接続を持たない（データベースとは切断されている）行セットの実装で、javax.sql.RowSet インタフェースと同様な標準インタフェースを提供します。OracleCachedRowSet は、Oracle の CachedRowSet の実装で、OracleCachedRowSet は CachedRowSet と同じように使用されます。

次のコードでは、OracleCachedRowSet オブジェクトを作成し、プロパティとして行セットの接続 URL、ユーザー名、パスワードおよび SQL 問合せを設定します。RowSet オブジェクトは、execute メソッドを使用して移入されます。execute をコールした後、java.sql.ResultSet オブジェクトとして RowSet オブジェクトを使用して、データの取得、スクロール、挿入、削除または更新を実行できます。

```
...
RowSet rowset = new OracleCachedRowSet ();
rowset.setUrl ("jdbc:oracle:oci8:@");
rowset.setUsername ("SCOTT");
rowset.setPassword ("TIGER");
rowset.setCommand ("SELECT empno, ename, sal FROM emp");
rowset.execute ();
while (rowset.next ())
{
  System.out.println ("empno: " +rowset.getInt (1));
  System.out.println ("ename: " +rowset.getString (2));
  System.out.println ("sal: "   +rowset.getInt (3));
}
...
```

問合せに CachedRowSet オブジェクトを移入するには、次の手順を実行します。

1. OracleCachedRowSet をインスタンス化します。
2. RowSet オブジェクトのプロパティとして、接続 Url、Username、Password、接続タイプ（オプション）および問合せ文字列を設定します。
3. execute メソッドをコールして、RowSet オブジェクトを移入します。

次のコードで示すように、populate() メソッドを使用して、既存の ResultSet オブジェクトに CachedRowSet を移入できます。

```
// Executing a query to get the ResultSet object.
ResultSet rset = pstmt.executeQuery ();

OracleCachedRowSet rowset = new OracleCachedRowSet ();
// the obtained ResultSet object is passed to the
// populate method to populate the data in the
// rowset object.
rowset.populate (rset);
```

前述の例では、問合せを実行して ResultSet オブジェクトを取得し、その ResultSet オブジェクトがキャッシュされた行セットの pupulate() メソッドに渡されて、結果セットの内容がキャッシュされた行セットに移入されています。

すでに使用可能な結果セットに CachedRowSet オブジェクトを移入するには、次の手順を実行します。

1. OracleCachedRowSet をインスタンス化します。
2. すでに使用可能な ResultSet オブジェクトを populate() メソッドに渡して、RowSet オブジェクトを移入します。

ResultSet インタフェースに含まれるすべてのインタフェースが、RowSet に実装されます。次のコードは、行セットをスクロールする方法を示します。

```
/**
 *  Scrolling forward, and printing the empno in
 *  the order in which it was fetched.
 */
// going to the first row of the rowset
rowset.beforeFirst ();
while (rowset.next ())
  System.out.println ("empno: " +rowset.getInt (1));
```

> **注意：** 既存の ResultSet オブジェクトを CachedRowSet オブジェクトの移入に使用する場合には、プロパティが適用される接続または結果セットがすでに作成されているため、トランザクションの分離や結果セットの並行性モードなどの接続プロパティおよびバインド・プロパティを設定することはできません。

前述の例では、beforeFirst() メソッドによって、カーソル位置が行セットの最初の行の前に初期化されます。行は、next() メソッドを使用して前方方向に取り出されます。

```
/**
 *  Scrolling backward, and printing the empno in
 *  the reverse order as it was fetched.
 */
//going to the last row of the rowset
rowset.afterLast ();
while (rowset.previous ())
  System.out.println ("empno: " +rowset.getInt (1));
```

前述の例では、RowSet の最後の行の後にカーソル位置が初期化されます。行は、RowSet の previous() メソッドを使用して逆方向に取り出されます。

行の挿入、更新および削除は、結果セット機能と同様に行セットでもサポートされます。次のコードは、行セットの 5 番目の位置に 1 行を挿入します。

```
/**
 * Inserting a row in the 5th position of the rowset.
 */
// moving the cursor to the 5th position in the rowset
if (rowset.absolute(5))
{
  rowset.moveToInsertRow ();
  rowset.updateInt (1, 193);
  rowset.updateString (2, "Ashok");
  rowset.updateInt (3, 7200);

  // inserting a row in the rowset
  rowset.insertRow ();

  // Synchronizing the data in RowSet with that in the
  // database.
  rowset.acceptChanges ();
}
```

前述の例では、パラメータ 5 が指定された absolute() メソッドにより、row set の 5 番目の位置にカーソルが移動され、moveToInsertRow() メソッドをコールすることにより、行セットに新しい行を挿入するための場所が作成されます。新しく作成された行を更新するには、updateXXX() メソッドを使用します。行のすべての列が更新されると、insertRow() メソッドがコールされ、その行セットが更新されます。変更は acceptChanges() メソッドを使用してコミットします。

次のコードは、OracleCachedRowSet オブジェクトをファイルに対してシリアル化してから取得します。

```
// writing the serialized OracleCachedRowSet object
{
  FileOutputStream fileOutputStream =
     new FileOutputStream ("emp_tab.dmp");
  ObjectOutputStream ostream = new
    ObjectOutputStream (fileOutputStream);
  ostream.writeObject (rowset);
  ostream.close ();
  fileOutputStream.close ();
}
```

```
// reading the serialized OracleCachedRowSet object
{
  FileInputStream fileInputStream = new
     FileInputStream ("emp_tab.dmp");
  ObjectInputStream istream = new
     ObjectInputStream (fileInputStream);
  RowSet rowset1 = (RowSet) istream.readObject ();
  istream.close ();
  fileInputStream.close ();
}
```

前述の例では、emp_tab.dmp ファイルの FileOutputStream オブジェクトがオープンされ、移入された OracleCachedRowSet オブジェクトが ObjectOutputStream を使用してこのファイルに書き込まれます。これは、FileInputStream および ObjectInputStream オブジェクトを使用して取り出されます。

InputStream、OutputStream、BLOBS および CLOBS などの非シリアル化形式のデータは、OracleCachedRowSet によってシリアル化されます。また、OracleCachedRowSets は独自のメタデータを実装するため、追加のラウンドトリップの必要なく取得できます。次のコードは、行セットのメタデータを取得する方法を示します。

```
ResultSetMetaData metaData = rowset.getMetaData ();
int maxCol = metaData.getColumnCount ();
for (int i = 1; i <= maxCol; ++i)
   System.out.println ("Column (" + i +") "
     +metaData.getColumnName (i));
```

前述の例は、ResultSetMetaData オブジェクトを取得して、RowSet に列名を出力する方法を示すものです。

OracleCachedRowSet クラスはシリアル化可能なので、Remote Method Invocation（RMI）の場合と同様に、ネットワークや別の JVM の間で渡すことができます。OracleCachedRowSet クラスを移入した後には、任意の JVM または JDBC ドライバを使用していない環境との間でも移動が可能になります。（acceptChanges() を使用して）行セットにデータをコミットするには、JDBC ドライバがインストールされている必要があります。

データを取り出して OracleCachedRowSet クラスに移入するプロセスはすべて、サーバーで実行され、移入された行セットは RMI や Enterprise JavaBeans（EJB）などの適切なアーキテクチャを使用してクライアントに渡されます。クライアントは、データベースに接続することなく、取出し、スクロール、挿入、更新および削除などのすべての操作を行セットで実行できます。データベースにデータがコミットされるたびに、acceptChanges() メソッドがコールされます。このメソッドにより、行セット内のデータがデータベース内のデータと同期化されます。このメソッドは JDBC ドライバを使用するため、JVM 環境に JDBC を実装する必要があります。このアーキテクチャは、パーソナル・デジタル・アシスタント（PDA）やネットワーク・コンピュータ（NC）などの Thin クライアントを使用するシステムに適しています。

CachedRowSet オブジェクトを移入すると、このオブジェクトを ResultSet オブジェクト、または RMI やその他の適切なアーキテクチャを使用してネットワークを通じて渡すことが可能な他のオブジェクトとして使用できます。

キャッシュされた行セットには、この他に次のような主要機能があります。

- 行セットのクローニング
- 行セットのコピーの作成
- 行セットの共有コピーの作成

CachedRowSet の制約事項

更新可能な結果セットに適用される制約事項すべてが行セットにも適用されます。ただし、OracleCachedRowSet はシリアル化可能なので、シリアライズ化は制約として適用されません。SQL 問合せには、次のような制約事項があります。

- データベース内の 1 つの表のみを参照できます。
- 結合操作は含まれません。
- 参照先の表の主キーを選択します。

また、挿入を実行するには、SQL 問合せが次の条件を満たしている必要があります。

- 基礎となる表の NULL 化可能な列すべてを選択します。
- デフォルト値のないすべての列を選択します。

注意： データはすべてメモリーにキャッシュされるため、CachedRowSet には大量のデータを格納できません。

接続に適用されるプロパティは、行セットの移入後には設定できません。これらのプロパティは、結果セットのトランザクションの分離や並列性モードと同様に、データを取り出した後には接続に適用できないためです。

JDBC 行セット

JDBC 行セットも、行セット実装の 1 つです。JDBC 行セットは、単純なシリアル化可能な行セットで、Bean インタフェース形式の JDBC インタフェースを提供します。JDBCRowSet へのコールはすべて JDBC インタフェースに直接伝播されます。JDBC インタフェースの使用方法は、他の行セット実装の場合と同じです。

表 A-1 は、JDBCRowSet インタフェースと CachedRowSet インタフェースの違いを示します。

表 A-1 JDBC 行セットとキャッシュされた行セットの比較

行セットのタイプ	シリアル化可能	データベースへの接続	JVM 内での移動が可能	データベースに対するデータの同期化	JDBC ドライバの存在
JDBC	なし	あり	なし	なし	あり
キャッシュ	あり	なし	あり	あり	なし

JDBC 行セットは、データベースに対してアクティブな接続を持つ接続行セットです。JDBC 行セットへのコールはすべて、JDBC 接続、文または結果セットのマッピング・コールに伝播されます。キャッシュされた行セットの場合は、データベースへの接続がオープンしていません。

JDBC 行セットでは JDBC ドライバが存在する必要があるのに対して、キャッシュされた行セットの場合は、行セットの移入時と行セットの変更をコミットするときを除いて、操作時に JDBC ドライバが存在する必要はありません。

次のコードは、JDBC 行セットの使用方法を示します。

```
RowSet rowset = new OracleJDBCRowSet ();
rowset.setUrl ("java:oracle:oci8:@");
rowset.setUsername ("SCOTT");
rowset.setPassword ("TIGER");
rowset.setCommand (
  "SELECT empno, ename, sal FROM emp");
rowset.execute ();
while (rowset.next ())
{
  System.out.println ("empno: " + rowset.getInt (1));
  System.out.println ("ename: "
    + rowset.getString (2));
  System.out.println ("sal: " + rowset.getInt (3));
}
```

前述の例では、接続 URL、ユーザー名、パスワードおよび SQL 問合せを行セットに対する接続プロパティとして設定し、`execute()` メソッドを使用して問合せを実行し、行セットを取り出して出力します。

B

JDBC エラー・メッセージ

この付録では、JDBC エラー・メッセージの一般構造について説明し、Oracle JDBC ドライバによって戻されることがある一般 JDBC エラー・メッセージと TTC エラー・メッセージをリストします。この付録には、次の項目が含まれます。

- JDBC エラー・メッセージの一般構造
- 一般 JDBC メッセージ
- TTC メッセージ

2 つのメッセージ・リストは、それぞれ初めに ORA 番号順、次に五十音順にソートされています。

JDBC 例外の処理についての一般情報は、3-33 ページの「SQL 例外の処理」を参照してください。

注意： 各メッセージの原因とアクションに関する情報は、将来のリリースで提供します。

JDBC エラー・メッセージの一般構造

一般 JDBC エラー・メッセージ構造では、次のようにメッセージの末尾にコロンを付けて、ランタイム情報を追加できます。

```
<error_message>:<extra_info>
```

たとえば、「closed statement」エラーは次のように出力されることがあります。

```
Closed Statement:next
```

これは（結果セット・オブジェクトの）`next()` メソッドのコール中に例外が発生したことを示します。

場合によっては、スタック・トレースになんらかの情報が見つかることがあります。

一般 JDBC メッセージ

この項では、初めに ORA 番号順、次に五十音順にソートして、一般 JDBC エラー・メッセージをリストします。

ORA 番号でソートした JDBC メッセージ

ORA-17001	内部エラー
ORA-17002	I/O 例外です。
ORA-17003	列索引が無効です。
ORA-17004	列の型が無効です。
ORA-17005	サポートされない列の型です。
ORA-17006	列名が無効です。
ORA-17007	動的列が無効です。
ORA-17008	非公開の接続です。
ORA-17009	非公開の文です。
ORA-17010	非公開の結果セットです。
ORA-17011	空の結果セットです。
ORA-17012	パラメータの型が競合します。
ORA-17014	`ResultSet.next` はコールされませんでした。
ORA-17015	文は取り消されました。
ORA-17016	文は時間切れになりました。

ORA-17017	カーソルはすでに初期化済です。
ORA-17018	無効なカーソルです。
ORA-17019	1 つの問合せのみ記述できます。
ORA-17020	行のプリフェッチが無効です。
ORA-17021	定義がありません。
ORA-17022	索引に定義がありません。
ORA-17023	サポートされない機能です。
ORA-17024	読み込むデータがありません。
ORA-17025	defines.isNull() でエラーが発生しました。
ORA-17026	数値のオーバーフローです。
ORA-17027	ストリームはすでにクローズ済です。
ORA-17028	現行の結果セットがクローズされるまで、新規結果セットを実行できません。
ORA-17029	setReadOnly: 読込み専用の接続はサポートされません。
ORA-17030	READ_COMMITTED および SERIALIZABLE のみが有効なトランザクション・レベルです。
ORA-17031	setAutoClose: 自動クローズ・モードが「オン」の場合のみサポートされます。
ORA-17032	行のプリフェッチをゼロに設定できません。
ORA-17033	不完全な SQL92 文字列です - 位置
ORA-17034	サポートされない SQL92 トークンです - 位置
ORA-17035	サポートされないキャラクタ・セットです。
ORA-17036	OracleNumber で例外が発生しました。
ORA-17037	UTF8 と UCS2 との間の変換に失敗しました。
ORA-17038	バイト配列の長さが不十分です。
ORA-17039	文字配列の長さが不十分です。
ORA-17040	接続 URL にサブ・プロトコルの指定が必要です。
ORA-17041	IN または OUT パラメータがありません - 索引 :
ORA-17042	バッチの値が無効です。
ORA-17043	ストリームの最大サイズが無効です。
ORA-17044	内部エラー : データ配列を割当てできません。

ORA-17045	内部エラー : バッチの値範囲を超えてバインド変数にアクセスしようとしました。
ORA-17046	内部エラー : データ・アクセスに対する索引が無効です。
ORA-17047	型記述子の解析でエラーが発生しました。
ORA-17048	型が定義されていません。
ORA-17049	JAVA と SQL のオブジェクト型が適合しません。
ORA-17050	ベクトルにそのような要素はありません。
ORA-17051	この API は、UDT 以外の型に使用できません。
ORA-17052	この参照は有効ではありません。
ORA-17053	サイズが無効です。
ORA-17054	LOB ロケータが無効です。
ORA-17055	無効なキャラクタが見つかりました -
ORA-17056	サポートされないキャラクタ・セットです。
ORA-17057	非公開の LOB です。
ORA-17058	内部エラー :NLS 変換率が無効です。
ORA-17059	内部表現の変換に失敗しました。
ORA-17060	記述子の構成に失敗しました。
ORA-17061	記述子がありません。
ORA-17062	参照カーソルが無効です。
ORA-17063	トランザクション中ではありません。
ORA-17064	構文が無効、またはデータベース名が NULL です。
ORA-17065	変換クラスが NULL です。
ORA-17066	アクセス・レイヤーには固有の実装が必要です。
ORA-17067	無効な Oracle URL が指定されました。
ORA-17068	コールに無効な引数があります。
ORA-17069	明示的な XA コールを使用してください。
ORA-17070	データ・サイズがこの型の最大サイズを超えています。
ORA-17071	最大 VARRAY 制限を超えました。
ORA-17072	列に対して挿入値が大きすぎます。
ORA-17073	論理ハンドルはすでに無効です。
ORA-17074	名前パターンが無効です。

ORA-17075	転送専用の結果セットに対する操作が無効です。
ORA-17076	読込み専用の結果セットに対する操作が無効です。
ORA-17077	REF 値の設定に失敗しました。
ORA-17078	接続はすでにオープンしているため、操作できません。
ORA-17079	既存のユーザー資格証明と一致しません。
ORA-17080	無効なバッチ・コマンドです。
ORA-17081	バッチ処理でエラーが発生しました。
ORA-17082	カレント行がありません。
ORA-17083	挿入行ではありません。
ORA-17085	値の競合が発生しました。
ORA-17086	挿入行の列値が未定義です。
ORA-17087	パフォーマンス・ヒントが無視されました :setFetchDirection()
ORA-17088	要求した結果セットの型と同時実行レベルの構文はサポートされていません。
ORA-17089	内部エラー
ORA-17090	操作できません。
ORA-17091	要求した型および / または同時実行レベルで結果セットを作成できません。
ORA-17092	JDBC 文でコール処理の終了を作成または実行できません。
ORA-17093	OCI 操作で、OCI_SUCCESS_WITH_INFO が戻りました。
ORA-17094	オブジェクト・タイプのバージョンが適合しません。
ORA-17095	文のキャッシュするにはこの接続オブジェクトは使用できません。
ORA-17096	文のキャッシュはこの論理接続に対して使用可能ではありません。
ORA-17097	PL/SQL の索引表の要素タイプが無効です。
ORA-17098	空の LOB 操作は無効です。
ORA-17099	PL/SQL の索引表の配列の長さが無効です。
ORA-17100	データベースの Java オブジェクトが無効です。
ORA-17101	OCI 接続プール・オブジェクトに無効なプロパティがあります
ORA-17102	Bfile は読取り専用です

ORA-17103	getConnection 経由で戻る接続タイプは無効です。かわりに getJavaSqlConnection を使用してください
ORA-17104	実行する SQL 文は空または NULL にできません
ORA-17105	接続セッションのタイム・ゾーンが設定されていません
ORA-17106	無効な接続の組合せが指定されました
ORA-17107	無効なプロキシ・タイプが指定されました

五十音順にソートした JDBC メッセージ

ORA-17019	1 つの問合せのみ記述できます。
ORA-17102	Bfile は読取り専用です
ORA-17025	defines.isNull() でエラーが発生しました。
ORA-17103	getConnection 経由で戻る接続タイプは無効です。かわりに getJavaSqlConnection を使用してください
ORA-17002	I/O 例外です。
ORA-17041	IN または OUT パラメータがありません - 索引 :
ORA-17049	JAVA と SQL のオブジェクト型が適合しません。
ORA-17092	JDBC でコール処理の終了を作成または実行できません。
ORA-17054	LOB ロケータが無効です。
ORA-17101	OCI 接続プール・オブジェクトに無効なプロパティがあります
ORA-17093	OCI 操作で OCI_SUCCESS_WITH_INFO が戻りました。
ORA-17036	OracleNumber で例外が発生しました。
ORA-17099	PL/SQL の索引表の配列の長さが無効です。
ORA-17097	PL/SQL の索引表の要素タイプが無効です。
ORA-17030	READ_COMMITTED および SERIALIZABLE のみが有効なトランザクション・レベルです。
ORA-17077	REF 値の設定に失敗しました。
ORA-17014	ResultSet.next はコールされませんでした。
ORA-17031	setAutoClose: 自動クローズ・モードが「オン」の場合のみサポートされます。
ORA-17029	setReadOnly: 読込み専用の接続はサポートされません。
ORA-17037	UTF8 と UCS2 との間の変換に失敗しました。

ORA-17066	アクセス・レイヤーには固有の実装が必要です。
ORA-17085	値の競合が発生しました。
ORA-17094	オブジェクト・タイプのバージョンが適合しません。
ORA-17017	カーソルはすでに初期化済です。
ORA-17048	型が定義されていません。
ORA-17047	型記述子の解析でエラーが発生しました。
ORA-17098	空の LOB 操作は無効です。
ORA-17011	空の結果セットです。
ORA-17082	カレント行がありません。
ORA-17061	記述子がありません。
ORA-17060	記述子の構成に失敗しました。
ORA-17079	既存のユーザー資格証明と一致しません。
ORA-17020	行のプリフェッチが無効です。
ORA-17032	行のプリフェッチをゼロに設定できません。
ORA-17028	現行の結果セットがクローズされるまで、新規結果セットを実行できません。
ORA-17064	構文が無効、またはデータベース名が NULL です。
ORA-17068	コールに無効な引数があります。
ORA-17051	この API は、UDT 以外の型に使用できません。
ORA-17052	この参照は有効ではありません。
ORA-17053	サイズが無効です。
ORA-17071	最大 VARRAY 制限を超えました。
ORA-17022	索引に定義がありません。
ORA-17034	サポートされない SQL92 トークンです - 位置
ORA-17023	サポートされない機能です。
ORA-17035	サポートされないキャラクタ・セットです。
ORA-17056	サポートされないキャラクタ・セットです。
ORA-17005	サポートされない列の型です。
ORA-17062	参照カーソルが無効です。
ORA-17104	実行する SQL 文は空または NULL にできません
ORA-17026	数値のオーバーフローです。

ORA-17043	ストリームの最大サイズが無効です。
ORA-17027	ストリームはすでにクローズ済です。
ORA-17040	接続 URL にサブ・プロトコルの指定が必要です。
ORA-17105	接続セッションのタイム・ゾーンが設定されていません
ORA-17078	接続はすでにオープンしているため、操作できません。
ORA-17090	操作できません。
ORA-17084	挿入行でコールされました。
ORA-17083	挿入行ではありません。
ORA-17086	挿入行の列値が未定義です。
ORA-17021	定義がありません。
ORA-17070	データ・サイズがこの型の最大サイズを超えています。
ORA-17100	データベースの Java オブジェクトが無効です。
ORA-17075	転送専用の結果セットに対する操作が無効です。
ORA-17007	動的列が無効です。
ORA-17063	トランザクション中ではありません。
ORA-17001	内部エラー
ORA-17089	内部エラー
ORA-17058	内部エラー :NLS 変換率が無効です。
ORA-17046	内部エラー : データ・アクセスに対する索引が無効です。
ORA-17044	内部エラー : データ配列を割当てできません。
ORA-17045	内部エラー : バッチの値範囲を超えてバインド変数にアクセスしようとしました。
ORA-17059	内部表現の変換に失敗しました。
ORA-17074	名前パターンが無効です。
ORA-17038	バイト配列の長さが不十分です。
ORA-17081	バッチ処理でエラーが発生しました。
ORA-17042	バッチの値が無効です。
ORA-17087	パフォーマンス・ヒントが無視されました :setFetchDirection()
ORA-17012	パラメータの型が競合します。
ORA-17057	非公開の LOB です。

ORA-17010	非公開の結果セットです。
ORA-17008	非公開の接続です。
ORA-17009	非公開の文です。
ORA-17033	不完全な SQL92 文字列です - 位置
ORA-17095	文のキャッシュするにはこの接続オブジェクトは使用できません。
ORA-17096	文のキャッシュはこの論理接続に対して使用可能ではありません。
ORA-17016	文は時間切れになりました。
ORA-17015	文は取り消されました。
ORA-17050	ベクトルにそのような要素はありません。
ORA-17065	変換クラスが NULL です。
ORA-17067	無効な Oracle URL が指定されました。
ORA-17018	無効なカーソルです。
ORA-17055	無効なキャラクタが見つかりました -
ORA-17106	無効な接続の組合せが指定されました
ORA-17080	無効なバッチ・コマンドです。
ORA-17107	無効なプロキシ・タイプが指定されました
ORA-17069	明示的な XA コールを使用してください。
ORA-17039	文字配列の長さが不十分です。
ORA-17091	要求した型および / または同時実行レベルで結果セットを作成できません。
ORA-17088	要求した結果セットの型と同時実行レベルの構文はサポートされていません。
ORA-17076	読込み専用の結果セットに対する操作が無効です。
ORA-17024	読み込むデータがありません。
ORA-17003	列索引が無効です。
ORA-17072	列に対して挿入値が大きすぎます。
ORA-17004	列の型が無効です。
ORA-17006	列名が無効です。
ORA-17073	論理ハンドルはすでに無効です。

HeteroRM XA メッセージ

次は、HeteroRM XA 機能に固有の JDBC エラー・メッセージです。

ORA 番号でソートした HeteroRM XA メッセージ

ORA-17200	XA open 文字列を Java から C へ正しく変換できません
ORA-17201	XA close 文字列を Java から C へ正しく変換できません
ORA-17202	RA 名を Java から C へ正しく変換できません
ORA-17203	ポインタ・タイプを jlong にキャストできません
ORA-17204	入力配列が短かすぎて OCI ハンドルを保持できません。
ORA-17205	xaoSvcCtx を使用して C-XA から OCISvcCtx ハンドルを取得するのに失敗しました。
ORA-17206	xaoEnv を使用して C-XA から OCIEnv を取得するのに失敗しました。
ORA-17207	tnsEntry プロパティが DataSource に設定されていません。
ORA-17213	xa_open で C-XA から XAER_RMERR が返されました
ORA-17215	xa_open で C-XA から XAER_INVAL が返されました
ORA-17216	xa_open で C-XA から XAER_PROTO が返されました
ORA-17233	xa_close で C-XA から XAER_RMERR が返されました
ORA-17235	xa_close で C-XA から XAER_INVAL が返されました
ORA-17236	xa_close で C-XA から XAER_PROTO が返されました

五十音順にソートした HeteroRM XA メッセージ

ORA-17202	RA 名を Java から C へ正しく変換できません
ORA-17207	tnsEntry プロパティが DataSource に設定されていません。
ORA-17201	XA close 文字列を Java から C へ正しく変換できません
ORA-17200	XA open 文字列を Java から C へ正しく変換できません
ORA-17235	xa_close で C-XA から XAER_INVAL が返されました
ORA-17236	xa_close で C-XA から XAER_PROTO が返されました
ORA-17233	xa_close で C-XA から XAER_RMERR が返されました
ORA-17215	xa_open で C-XA から XAER_INVAL が返されました

ORA-17216	xa_open で C-XA から XAER_PROTO が返されました
ORA-17213	xa_open で C-XA から XAER_RMERR が返されました
ORA-17206	xaoEnv を使用して C-XA から OCIEnv を取得するのに失敗しました。
ORA-17205	xaoSvcCtx を使用して C-XA から OCISvcCtx ハンドルを取得するのに失敗しました。
ORA-17204	入力配列が短かすぎて OCI ハンドルを保持できません。
ORA-17203	ポインタ・タイプを jlong にキャストできません

TTC メッセージ

この項では、初めに ORA 番号順に、次に五十音順にソートして、TTC エラー・メッセージをリストします。

ORA 番号でソートした TTC メッセージ

ORA-17401	プロトコル違反です。
ORA-17402	RPA メッセージは 1 つのはずです。
ORA-17403	RXH メッセージは 1 つのはずです。
ORA-17404	予定より多い RXD を受け取りました。
ORA-17405	UAC の長さはゼロではありません。
ORA-17406	最大バッファ長を超えています。
ORA-17407	型表現が無効です (setRep)。
ORA-17408	型表現が無効です (getRep)。
ORA-17409	バッファ長が無効です。
ORA-17410	ソケットから読み込むデータはこれ以上ありません。
ORA-17411	データ型の表現が適合しません。
ORA-17412	型の長さが最大を超えています。
ORA-17413	キー・サイズが大きすぎます。
ORA-17414	列名を保存するにはバッファ・サイズが不十分です。
ORA-17415	この型は未処理です。
ORA-17416	FATAL

ORA-17417	NLS の問題で、列名のデコードに失敗しました。
ORA-17418	内部構造体のフィールド長エラーです。
ORA-17419	無効な列の数が戻りました。
ORA-17420	Oracle バージョンが定義されていません。
ORA-17421	型または接続が定義されていません。
ORA-17422	ファクトリに無効なクラスがあります。
ORA-17423	IOV の定義なしで PLSQL ブロックを使用しています。
ORA-17424	異なる配列操作を試みています。
ORA-17425	PLSQL ブロックでストリームを戻しています。
ORA-17426	IN バインド、OUT バインドともに NULL です。
ORA-17427	初期化されていない OAC を使用しています。
ORA-17428	接続の後にはログオンのコールが必要です。
ORA-17429	少なくともサーバーに接続している必要があります。
ORA-17430	サーバーへのログオンが必要です。
ORA-17431	解析する SQL 文が NULL です。
ORA-17432	all7 でオプションが無効です。
ORA-17433	コールの引数が無効です。
ORA-17434	ストリーム・モードではありません。
ORA-17435	IOV で in_out_binds の数が無効です。
ORA-17436	アウトバインドの数が無効です。
ORA-17437	PLSQL ブロックの IN/OUT 引数にエラーがあります。
ORA-17438	内部 - 予期しない値です。
ORA-17439	SQL の型が無効です。
ORA-17440	DBItem/DBType が NULL です。
ORA-17441	この Oracle バージョンはサポートされません。7.2.3 以降はサポートされます。
ORA-17442	Refcursor 値が無効です。
ORA-17443	NULL のユーザーまたはパスワードは、THIN ドライバでサポートされません。
ORA-17444	サーバーから受け取った TTC プロトコル・バージョンはサポートされません。

五十音順にソートした TTC メッセージ

ORA-17432	all7 でオプションが無効です。
ORA-17440	DBItem/DBType が NULL です。
ORA-17416	FATAL
ORA-17426	IN バインド、OUT バインドともに NULL です。
ORA-17435	IOV で in_out_binds の数が無効です。
ORA-17423	IOV の定義なしで PLSQL ブロックを使用しています。
ORA-17417	NLS の問題で、列名のデコードに失敗しました。
ORA-17443	NULL のユーザーまたはパスワードは、THIN ドライバでサポートされません。
ORA-17420	Oracle バージョンが定義されていません。
ORA-17425	PLSQL ブロックでストリームを戻しています。
ORA-17437	PLSQL ブロックの IN/OUT 引数にエラーがあります。
ORA-17442	Refcursor 値が無効です。
ORA-17402	RPA メッセージは 1 つのはずです。
ORA-17403	RXH メッセージは 1 つのはずです。
ORA-17439	SQL の型が無効です。
ORA-17405	UAC の長さはゼロではありません。
ORA-17436	アウトバインドの数が無効です。
ORA-17431	解析する SQL 文が NULL です。
ORA-17412	型の長さが最大を超えています。
ORA-17408	型表現が無効です (getRep)。
ORA-17407	型表現が無効です (setRep)。
ORA-17421	型または接続が定義されていません。
ORA-17413	キー・サイズが大きすぎます。
ORA-17433	コールの引数が無効です。
ORA-17424	異なる配列操作を試みています。
ORA-17441	この Oracle バージョンはサポートされません。7.2.3 以降はサポートされます。
ORA-17415	この型は未処理です。

ORA-17444	サーバーから受け取った TTC プロトコル・バージョンはサポートされません。
ORA-17430	サーバーへのログオンが必要です。
ORA-17406	最大バッファ長を超えています。
ORA-17427	初期化されていない OAC を使用しています。
ORA-17429	少なくともサーバーに接続している必要があります。
ORA-17434	ストリーム・モードではありません。
ORA-17428	接続の後にはログオンのコールが必要です。
ORA-17410	ソケットから読み込むデータはこれ以上ありません。
ORA-17411	データ型の表現が適合しません。
ORA-17418	内部構造体のフィールド長エラーです。
ORA-17438	内部 - 予期しない値です。
ORA-17409	バッファ長が無効です。
ORA-17422	ファクトリに無効なクラスがあります。
ORA-17401	プロトコル違反です。
ORA-17419	無効な列の数が戻りました。
ORA-17404	予定より多い RXD を受け取りました。
ORA-17414	列名を保存するにはバッファ・サイズが不十分です。

索引

記号

数字

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

あ

い

う

え

お

か

き

し

す

せ

た

ち

て

と

な

に

ね

は

ふ

め

も

ゆ

よ

り

れ

ろ