Oracle8*i*

JDBC 開発者ガイドおよびリファレンス

リリース 8.1

ORACLE

Oracle8*i* JDBC 開発者ガイドおよびリファレンス リリース 8.1

部品番号：A62737-1

第 1 版 1999 年 5 月（第 1 刷）

原本名：Oracle8*i* JDBC Developer's Guide and Reference, Release 8.1.5

原本部品番号：A64685-01

原本著者：Thomas Pfaeffle

原本協力者：Prabha Krishna, Bernie Harris, Ana Hernandez, Anthony Lau, Paul Lo, Jack Melnick, Janice Wong, Brian Wright, Joyce Yang

目次

3 基本機能

4 Oracle エクステンション

5 上級トピック

6 コーディングのヒントおよびトラブルシューティング

7 サンプル・アプリケーション

8 リファレンス情報

A JDBC エラー・メッセージ

索引

はじめに

次の項が含まれます。

対象とする読者

このマニュアルは、読者が経験を積んだプログラマで、Oracle データベース、SQL、Java プログラミング言語、および JDBC の原理を理解していることを前提とします。

マニュアルの構成

JDBC 開発者ガイドおよびリファレンスには、8 つの章と 1 つの付録が含まれます。

第 1 章「概要」	この章では、Oracle による JDBC の実現方法および 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」	この章では、Oracle JDBC ドライバおよびその使用方法を示すシナリオを紹介します。また、実行したインストールと設定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」	この章では、任意の JDBC アプリケーションを作成する際の基本的な手順を説明します。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「Oracle エクステンション」	この章では、Oracle の提供する JDBC エクステンションである、パッケージ、クラス、データ型について説明します。また、エクステンションの提供する LOB、オブジェクト、およびコレクションのサポートについても説明します。
第 5 章「上級トピック」	この章では、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 の上級トピックを扱います。
第 6 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」	この章では、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 7 章「サンプル・アプリケーション」	この章では、JDBC の先進機能と Oracle エクステンションを際立たせた、サンプル・アプリケーションについて説明します。
第 8 章「リファレンス情報」	この章では、JDBC 参照情報の詳細を説明します。
付録 A「JDBC エラー・メッセージ」	ここでは、JDBC ドライバの発行するエラーの一覧を示します。

関連資料

このマニュアルには、Oracle の提供する次の資料が参照されています。

- 『Oracle8*i* JPublisher User's Guide』

  この資料は、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 クラスに翻訳する方法を説明します。オブジェクト型、varray 型、ネストした表タイプ、または REF 型を使用する SQLJ または JDBC アプリケーションを開発している場合、これらの型に対応した Java クラスが必要となります。JPublisher は、オブジェクト型と Java クラス間のマップ、およびオブジェクト属性タイプと対応する Java タイプ間のマップを作成することにより、ユーザーの作業を支援します。

- 『Oracle8*i* SQLJ 開発者ガイドおよびリファレンス』

  この資料は、SQLJ を使って静的 SQL 操作を直接 Java コードに埋め込む方法を説明します。標準 SQLJ 機能と Oracle 独自の SQLJ 機能の両方の説明を含みます。

- 『Oracle8*i* Java ストアド・プロシージャ開発者ガイド』

  この資料は、Java プログラマが Oracle RDBMS へアクセスする際に使用する、Java ストアド・プロシージャについて説明します。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・トリガおよび SQL メソッド）を使って、ビジネス・ロジックをサーバー・レベルでインプリメントすることにより、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。

- 『Oracle8*i* Enterprise JavaBeans と CORBA 開発者ガイド』

  この資料は、JavaBeans および CORBA 仕様に対する Oracle エクステンションについて説明します。

- 『Net8 管理者ガイド』

  ANO（Advanced Network Option）、Oracle8 Connection Manager、および Net8 を使ったネットワーク管理全般の詳細は、このマニュアルを参照してください。

- 『Oracle8*i* エラー・メッセージ』

  Oracle データベースおよび Oracle JDBC ドライバからのエラー・メッセージの詳細は、このドキュメント・セットを参照してください。

- 『Oracle8*i* NLS ガイド』

  NLS 環境変数、キャラクタ・セット、地域およびローカル・データの詳細は、このマニュアルを参照してください。さらに、このマニュアルには、NLS 共通の問題、標準的ないくつかのシナリオ、および NLS に関して OCI や SQL プログラマが考慮すべき点も含まれます。

- 『Oracle8*i* アプリケーション開発者ガイド ラージ・オブジェクト』および『Oracle8*i* Application Developer's Reference - Packages』

これらの資料は、PL/SQL コードおよび DBMS_LOB パッケージを使用した、ラージ・オブジェクト（LOB）へのアクセスおよび操作方法について説明します。

- 『Oracle8*i* SQL リファレンス』

  このリファレンスは、Oracle データベースの情報管理に使用する SQL（Structured Query Language）の内容および構文について説明します。

- 『PL/SQL ユーザーズ・ガイドおよびリファレンス』

  PL/SQL は、SQL に対する Oracle のプロシージャ・エクステンションです。先進の第 4 世代プログラミング言語（4GL）である PL/SQL は、シームレスな SQL アクセス、Oracle サーバーやツールとの緊密な統合、移植性、セキュリティ、およびデータのカプセル化やオーバーロード、例外処理、情報隠ぺいなどの最新のソフトウェア工学機能を提供します。このガイドでは、PL/SQL の背後にある設計思想を説明し、この言語の各要素を例証します。

- Oracle8*i* Application Server 関連マニュアル

  Oracle8*i* Application Server による JDBC のサポート情報の詳細は、このマニュアルを参照してください。

- Oracle8 JDeveloper Suite 関連マニュアル

  Oracle8 JDeveloper Suite による JDBC のサポート情報の詳細は、このマニュアルを参照してください。

このマニュアルで使用する表記規則

本書では、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、各行の最後ではキャリッジ・リターン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、本書で使用します。

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。
[ ]	角カッコは、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。

1
概要

この章では、Oracle による JDBC の実現方法の概要を説明します。次のトピックが含まれます。

- JDBC とは何か
- JDBC 対 SQLJ
- 基本ドライバ・アーキテクチャ
- JDBC 標準に対する Oracle エクステンション
- サポートする JDK および JDBC のバージョン
- JDBC と Oracle Application Server
- JDBC と IDE

JDBC とは何か

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 Microsystems の定義による JDBC 標準は、各プロバイダが独自の JDBC ドライバで標準をインプリメント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。

標準 JDBC API に加え、Oracle ドライバにはプロパティ、タイプ、およびパフォーマンスの拡張機能があります。

JDBC 対 SQLJ

この項に含まれるサブセクションは、次のとおりです。

- 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）レイヤに精通している開発者は、OCI が C や C++ プログラマに提供するようなパワーと柔軟性を、JDBC が Java プログラマに提供することを認めるでしょう。OCI の場合と同様、JDBC を使って、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行えます。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 文を使う 1 つの方法です。コール・プログラムは、JDBC を使って実行時に 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 を使って静的 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（つまり Java プログラムの「本文内で明白」）しています。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・ステップでは、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・コードと共に 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、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、『Oracle8*i* SQLJ 開発者ガイドおよびリファレンス』を参照してください。

SQLJ と JDBC の構文と方法は、実行時の構成に依存しないため、クライアントやデータベース側または中間層でのインプリメントが可能です。

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いです。
- SQLJ は、静的 SQL コードのタイプ・チェックにデータベース接続を使用します。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得または設定（あるいはその両方）したり、位置番号でバインドを指定する必要があります。
- SQLJ は、問合せ出力と戻りパラメータの強力な型付けを提供し、コール時の型チェックが可能です。JDBC は、コンパイル時の型チェックなしで SQL と値の受渡しを行います。
- 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

次の場合は、SQLJ を使ってプログラムを記述します。

- 実行時ではなく翻訳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実行可能なアプリケーションを記述する場合。SQLJ を使うと、実行時に、対象のデータベースに合わせて静的 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。

次の場合は、JDBC を使ってプログラムを記述します。

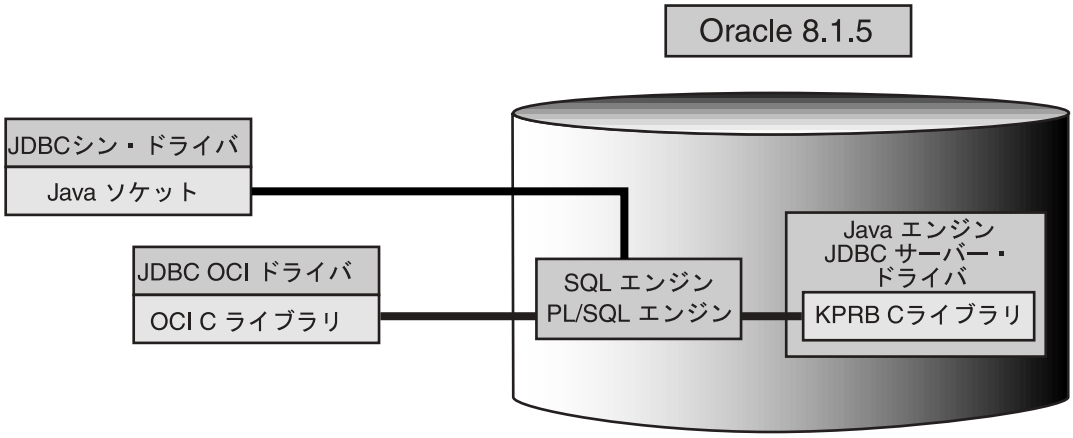
- プログラムで動的 SQL を使用する場合。たとえば、問合せを迅速に構築したり、対話的コンポーネントを使用するプログラムでは、JDBC を使用します。
- 実行時や開発時に SQLJ レイヤを必要としない場合。たとえば、低速接続でのダウンロード時間を最小にするため、SQLJ ランタイム・ライブラリをダウンロードせずに JDBC Thin ドライバだけをダウンロードする場合があります。

基本ドライバ・アーキテクチャ

この項に含まれるサブセクションは、次のとおりです。

図 1-1 に、JDBC Thin、OCI、およびサーバー・ドライバ用のドライバ / データベース・アーキテクチャを示します。

図 1-1 ドライバ / データベース・アーキテクチャ



JDBC Thin クライアント側ドライバ・アーキテクチャ

Oracle JDBC Thin ドライバは、アプレット開発者を対象とした Type IV ドライバです。このドライバは、100% Pure Java で記述され、JDBC 1.22 標準に準拠しています。

データベースとの通信用に、ドライバは、Oracle の TTC プレゼンテーション・プロトコルおよび Net8 セッション・プロトコルと等価な機能を Java で実装しています。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。Net8 プロトコルは、TCP/IP だけで動作します。このドライバを使用するために、クライアントに Oracle 独自のソフトウェアをインストールする必要はありません。

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 OCI クライアント側ドライバ・アーキテクチャ

JDBC OCI ドライバは、クライアント / サーバー型の Java アプリケーション・プログラマおよび Java ベースの中間層開発者を対象とした、Type II ドライバです。JDBC OCI ドライバは、JDBC 起動を Oracle コール・インタフェース（OCI）へのコールに変換します。その後これらのコールは、Net8 経由で Oracle データベース・サーバーへ送信されます。

JDBC OCI ドライバは、OCI ライブラリをコールする必要があるため、Java と C の両方を使って記述されています。ドライバが機能するには、OCI ライブラリ、Net8、CORE ライブラリ、および他の必要なファイルが、各クライアント・マシンまたはドライバがインストールされた中間層のアプリケーション・サーバーに存在する必要があります。

JDBC サーバー・ドライバ・アーキテクチャ

JDBC サーバー・ドライバは、Oracle 8.1.5 Java Virtual Machine（VM）を使用してデータベース内で稼動する Java プログラムが、SQL エンジンと通信することを可能にします。サーバー・ドライバ、Java VM、データベース、KPRB（サーバー側）C ライブラリ、および SQL エンジンすべては、同一のアドレス空間内で動作します。ネットワーク・ラウンドトリップは関係しません。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。

JDBC 標準に対する Oracle エクステンション

Oracle JDBC ドライバは、JDBC 2.0 標準の機能の多くをサポートします。サポートは、Oracle データ型、オブジェクト型、およびそれらの Java へのマッピングに対する Oracle 定義のエクステンションという形式で提供されます。これらのエクステンションの詳細は、第 4 章の「Oracle エクステンション」を参照してください。

サポートする JDK および JDBC のバージョン

Oracle の JDBC ドライバ、リリース 8.1.5 は、JDK バージョン 1.0.2 および 1.1.*x* をサポートします。これらのドライバは、JDBC バージョン 1.22 にも準拠しており、さらに JDBC バージョン 2.0 の大半の機能をインプリメントしています。

注意 : アプレットのコンテキストで、Thin ドライバを JDK 1.0.2 および 1.1.1 と共に使用する場合、特に考慮すべき点があります。このトピックの詳細は、5-6 ページの「アプレットの操作」を参照してください。

JDBC と Oracle Application Server

Oracle Application Server 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。Oracle Application Server は、Hypertext Transfer Protocol（HTTP）を使用する Web クライアント（ブラウザ）、および Common Object Request Broker Architecture（CORBA）と Internet Inter-ORB Protocol （IIOP）を使用する CORBA クライアントの両方からのアプリケーションへのアクセスをサポートしています。

中間層の JDBC OCI ドライバを、Oracle Web Application Server バージョン 3.0 以降と共に使用できます。Oracle Web Application Server には、JDBC がバンドルされています。JDBC および Oracle Web Application Server の使用方法の詳細は、Oracle Web 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ビルド、デバッグおよび実行するための、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 Java およびネイティブ Oracle8 ドライバを含む、JDBC の統合サポートが含まれます。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動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、インストールのテストと確認、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。説明は、次のトピックごとに行います。

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

Oracle JDBC ドライバ

この項には次の項目が含まれます。

Oracle は、2 つのクライアント用ドライバ（1 つは中間層でも使用可能）と、1 つのサーバー用ドライバを提供しています。続く章では、クライアント側が使用するドライバに焦点を当てて説明します。サーバー側が使用するドライバの詳細は 5-20 ページの「サーバー上の JDBC: サーバー・ドライバ」を参照してください。

Oracle JDBC ドライバの紹介

この項では、Oracle JDBC ドライバとその使用例について説明します。Oracle は、クライアント用とサーバー用の JDBC ドライバを提供しています。クライアント側のドライバは、クライアントまたは 3 層構造の中間層で実行される Java アプリケーションや Java アプレットで使用できます。サーバー側のドライバは、Java VM と SQL エンジンとの通信を可能にするサーバー側の JDBC サポートを提供します。

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。

- JDBC 1.22
- JDBC 2.0 が備えるほとんどの機能
- 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

ドライバ間で異なるのは、データベースへの接続方法とデータの転送方法だけです。

JDBC Thin ドライバ

Oracle JDBC Thin ドライバは、JDBC 1.22 の標準規格に準拠した 100% Pure Java インプリメンテーションです。JDBC Thin ドライバは、Java ソケットを使って Oracle Server に直接接続します。JDBC Thin ドライバは Java アプリケーションからも使用できますが、通常は 2 層構造または 3 層構造の Java アプレットで使用されます。JDBC Thin ドライバは、TCP/IP バージョンの Oracle の Net8 を独自に実現しています。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。JDBC Thin ドライバをアプレットと共に使用する場合、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。

JDBC Thin ドライバは、クライアント側に Oracle ソフトウェアを必要とせず、実行する Java アプレットと同時にブラウザにダウンロードできます。クライアント（通常はブラウザ）の Java アプレット・タグを含む HTML ページで、URL を選択します。Web サーバーは、Java アプレットと JDBC Thin ドライバをクライアントにダウンロードします。次に、JDBC Thin ドライバが、Java ソケットを使ってデータベース・サーバーへの接続を直接確立します。

JDBC Thin ドライバは、リリース 7.2.3 以降の任意の Oracle データベースに接続可能です。JDBC Thin ドライバを使用すると、Java ソケットの上で Net8 と TTC（OCI が使用する接続プロトコル）をエミュレートすることにより、データベースに直接接続できます。このドライバでは、TCP/IP プロトコルだけがサポートされます。また TNS リスナーがデータベース・サーバー上の TCP/IP ソケットをリスニングする必要もあります。

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についての説明は、5-6 ページの「アプレットの操作」を参照してください。

JDBC OCI ドライバ

JDBC OCI ドライバは、Oracle コール・インタフェース（OCI）を使用して JDBC インタフェースをインプリメントします。OCI ドライバは、OCI キャッシュ、OCI への C エントリ・ポイントおよび OCI ライブラリを利用します。C エントリ・ポイントのコールにネイティブ・メソッドを使用するため、プラットフォーム固有のドライバを使用します。JDBC OCI ドライバには、Net8 を含む、Oracle クライアントのインストールが必要です。

JDBC OCI ドライバは、OCI 経由で Oracle データベースへのインタフェースを提供するため、全バージョンの Oracle データベースと互換性があります。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 Net8 アダプタをサポートします。

JDBC OCI ドライバには C のコードが含まれるため、アプレットでの使用には適しません。ただし、Java アプリケーションや Oracle Web Application Server などの Java 中間層での使用には最適です。次のような構成で、JDBC OCI ドライバを使用できます。

- 2 層構造のクライアント・マシンで稼動する Java アプリケーション
- 3 層構造の中間層で稼動する Java アプリケーション
- 3 層構造の中間層で稼動する Java サーブレット

JDBC サーバー・ドライバ

Oracle の JDBC サーバー・ドライバを使用できるのはサーバー側だけです。サーバー・ドライバは、データベースで使用される Java プログラム、Java ストアド・プロシージャ、Enterprise Java Beans（EJB）、および SQL と PL/SQL プログラムとの通信用などサーバー側の JDBC サポートを提供します。サーバー・ドライバは、クライアント側のドライバ機能との一貫性および同一機能をサポートします。サーバー側のドライバの詳細は、5-20 ページの「サーバー上の JDBC: サーバー・ドライバ」を参照してください。

適切なドライバの選択

アプリケーションやアプレットで使用する JDBC ドライバを選択する際に、次の 4 つの点を考慮してください。

- 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 に基づくドライバ・クラスは、（C 言語の）ネイティブ・メソッドをコールするため、Web ブラウザにダウンロードすることができません。

  注意：

  - JDBC Thin ドライバは、Net8 プロトコルのサブセットを使用します。このドライバは、Java で記述され、TCP/IP プロトコルを使って接続します。
  - JDBC ドライバの選択に加え、アプレットにはさらに制限があります。これらの制限の詳細は、5-19 ページの「ブラウザのセキュリティと JDK バージョンの考慮点」を参照してください。

- 移植性を最大限にする場合は、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 Oracle8 データ・サーバーに接続できます。
- 記述するアプリケーションのパフォーマンスを最も重視する場合は、JDBC OCI ドライバを選択してください。
- Oracle 8.1.5 Java VM を使って Oracle データベース・サーバーを稼動させている場合は、JDBC サーバー・ドライバを選択してください。

Oracle JDBC ドライバの必要条件および互換性

次の表 2-1 では Oracle データベースと JDBC ドライバのバージョン間の互換性を示します。

表 2-1 JDBC ドライバとデータベースの互換性

データベースのバージョン	ドライバのバージョン	備考
8.1.5	JDBC Thin ドライバ JDBC OCI ドライバ JDBC サーバー・ドライバ	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完全なオブジェクト・サポートを提供します。
8.1.4	JDBC Thin ドライバ JDBC OCI ドライバ JDBC サーバー・ドライバ	リリース 8.1.4 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完全なオブジェクト・サポートを提供します。
8.0.*x*	JDBC Thin ドライバ JDBC OCI ドライバ **注意 :** JDBC サーバー・ドライバは、リリース 8.0.x では使用できません。	リリース 8.0.*x* のデータベースで使用する場合、JDBC OCI および Thin ドライバはオブジェクトをサポートしません。これは、JDBC がリリース 8.0.*x* 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。
7.*x*	JDBC Thin ドライバ JDBC OCI ドライバ **注意** : JDBC サーバー・ドライバは、バージョン 7.x では使用できません。	バージョン 7.*x* のデータベースで使用する場合、JDBC OCI および Thin ドライバは、オブジェクトをサポートしません。これは、JDBC がバージョン 7.*x* 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。 JDBC OCI ドライバは、LOB をサポートしていません。

JDBC クライアント・インストールの検証

この項には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の必要はありません。(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、Net8 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、Sun Microsystems の Java Developer's 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 JDBC ドライバは、JDK バージョン 1.0.2 および 1.1.x と互換性があります。バージョン 8.1.5 用の Oracle JDBC ドライバは JDK 1.2 をサポートしていません。

JDBC のディレクトリ

Oracle Java サーバー製品をインストールすると、`[ORACLE_HOME]` に `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` ディレクトリには、SQL92 や Oracle SQL の構文や、PL/SQL ブロック、ストリーム、および Oracle JDBC 型や、パフォーマンス拡張の使用法を示すサンプル・プログラムが含まれます。`demo` ディレクトリには、`samples` サブディレクトリだけがあります。
- `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。`classes111.zip`（JDK 1.1.1 用）および `classes102.zip`（JDK 1.0.2 用）
- `readme.txt`: `readme.txt` ファイルには、マニュアルで説明されていない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、JDBC OCI ドライバと JDBC Thin ドライバ用に設定する環境変数について説明します。

Solaris および Windows NT プラットフォーム

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 1.0.2 または 1.1.1 のいずれを使用しているかに応じて、CLASSPATH に次の中から適切な値を指定してください。

- `[Oracle Home]/jdbc/lib/classes102.zip`

または

- `[Oracle Home]/jdbc/lib/classes111.zip`

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

- `[Oracle Home]/lib`

Solaris の場合、このディレクトリには共有オブジェクト・ライブラリ libocijdbc8.so 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、ほか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さらに確認するには、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。

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.driver.*;

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たことになります。

JdbcCheckup.java は簡単なプログラムですが、次の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 での最初のステップ」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.driver.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、基本的な手順を説明します。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。次のトピックが含まれます。

- JDBC での最初のステップ
- サンプル : 接続、クエリーおよび結果処理
- データ型マッピング
- JDBC での Java ストリームの使用方法
- JDBC プログラムでのストアド・プロシージャの使用方法
- エラー・メッセージと JDBC
- サーバー側の基本
- アプリケーションの基本およびアプレットの基本

JDBC での最初のステップ

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. クエリーの実行と結果セット・オブジェクトの復帰
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. 接続のクローズ

これら 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 ドライバの種類に関わらず、次の `import` 文をプログラムの最初に記述する必要があります。

`import java.sql.*`	JDBC パッケージ。
`import java.math.*`	Java math パッケージ。これらのパッケージは、`BigDecimal` クラス等に必要です。

Oracle ドライバの提供する拡張機能を利用する場合、次の Oracle パッケージをプログラムに追加する必要があります。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。

`oracle.jdbc.driver.*` および `oracle.sql.*`	JDBC への任意の Oracle 固有エクステンションをプログラムで使用する場合、これらのパッケージを追加します。Oracle エクステンションの詳細は、第 4 章の「Oracle エクステンション」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()` メソッドを使って行います。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。次に例を示します。

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

ただし、このメソッドは、JDK に準拠した Java 仮想マシンでのみ有効です。Microsoft Java 仮想マシン上では、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 1 度だけ登録します。

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

注意 : Thin ドライバをアプレットに登録する場合、この例で指定したドライバ文字列とは異なる文字列を入力する必要があります。アプレット用 Thin ドライバの登録の詳細は、5-6 ページの「アプレットのコーディング」を参照してください。

データベースへの接続のオープン

データベースへの接続を、JDBC `DriverManager` クラスの静的な `getConnection()` メソッドを使ってオープンします。このメソッドは、JDBC `Connection` クラスのオブジェクトを返します。このオブジェクトには、ユーザー ID、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。このメソッドに精通していない場合は、次の「getConnection() のフォームについて」を参照してください。

`getConnection()` メソッドに精通している場合は、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。

- 3-6 ページの「JDBC OCI ドライバ用の接続のオープン」
- 3-6 ページの「JDBC Thin ドライバ使用時の接続のオープン」

注意 : この項の説明は、クライアント側ドライバだけに当てはまります。サーバー側ドライバを使用してデータベース接続をオープンする方法は、3-24 ページの「サーバー側の基本」を参照してください。

getConnection() のフォームについて

`getConnection()` メソッドはオーバーロードされるメソッドで、この項で説明する方法で宣言します。

- 3-4 ページの「データベース URL、ユーザー ID、およびパスワードの指定」
- 3-5 ページの「ユーザー ID とパスワードを含むデータベース URL の指定」
- 3-5 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」

注意 : デフォルト接続が存在する場合は、データベース名を指定する必要はありません。デフォルト接続の詳細は、5-20 ページの「サーバー・ドライバを使ったデータベースへの接続」を参照してください。

接続でデータベース名を指定する場合の書式は、次のいずれかでなければなりません。

- Net8 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` エントリの詳細は、『Net8 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 ID、およびパスワードの指定

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

URL の書式は、次のとおりです。

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

次の例では、Thin ドライバを使って、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、Net8 キーワード値ペアを使ってデータベースを指定することもできます。Net8 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 ID とパスワードを含むデータベース URL の指定

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

URL の書式は、次のとおりです。

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

次の例では、OCI ドライバを使って、パスワード tiger のユーザー scott を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);
```

データベース URL とプロパティ・オブジェクトの指定

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

URL の書式は、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。次に例を示します。

```
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);
```

接続プロパティ・オブジェクトへの Oracle エクステンション Oracle は、Oracle JDBC ドライバがサポートする接続プロパティへのエクステンションを複数定義しています。getConnection() メソッドおよび Properties オブジェクトに対する Oracle エクステンションの書式の詳細は、4-98 ページの「接続プロパティの Oracle エクステンション」を参照してください。

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 に格納されています。UNIX システムでは、\var\opt\oracle 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 ドライバの場合（Thin ドライバの場合と同様）、Net8 キーワード値ペアで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。Net8 キーワード値ペアは、その他の JDBC ドライバに対しても機能します。

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、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");
```

JDBC Thin ドライバ使用時の接続のオープン

Oracle クライアント・インストレーション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 エントリを使用することはできません。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");
```

Net8 キーワード値ペアを使っ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** : アプレット用の接続文を記述する場合、これらの例で使用した接続文字列とは異なる接続文字列を入力する必要があります。アプレットによるデータベースへの接続の詳細は、5-6 ページの「アプレットのコーディング」を参照してください。

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返します。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 独自の部分がないことに注意してください。

クエリーの実行と結果セット・オブジェクトの復帰

データベースへの問合せを行う場合、Statement オブジェクトで 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 クラスのオブジェクトを返します。

この例の、Statement オブジェクト stmt 作成後の次のステップは、クエリーを実行し、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 独自の部分はなく、標準 JDBC 構文に準拠しています。

注意 : JDBC ドライバは、実際に OracleResultSet オブジェクトを返しますが、その対象は標準 ResultSet 出力変数です。Oracle エクステンションを使って結果セットを処理する場合は、出力を OracleResultSet にキャストする必要があります。この詳細は、4-20 ページの「oracle.jdbc.driver パッケージのクラス」を参照してください。

結果セットの処理

クエリーの実行後は、ResultSet オブジェクトの next() メソッドを使っ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ループして、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 オブジェクトのさまざまな getXXX() メソッドを使用します。ここで、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、各従業員名の取得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 を返します。従業員名は、Java 文字列として実装されます。

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() メソッドを使って実行します。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。

データベース内のカーソルを解放することもできます。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、次の行を使っ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。

接続のクローズ

作業を完了したら、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、Connection クラスの close() メソッドを使って実行します。たとえば、Connection オブジェクトが conn の場合、次の文で接続をクローズします。

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

サンプル : 接続、クエリーおよび結果処理

次の例は、前の項で説明したステップ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クエリーを実行し、結果セットを処理します。

Statement オブジェクトの作成、クエリーの実行、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.driver.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。

注意: アプレット用のコードを作成する場合、getConnection() および registerDriver() 文字列は異なります。詳細は、5-6 ページの「アプレットのコーディング」を参照してください。

データ型マッピング

Oracle JDBC ドライバは、JDBC 1.22 が必要とする SQL データ型をサポートします。さらに、Oracle JDBC ドライバは、Oracle 独自の ROWID データ型および REF CURSOR カテゴリのユーザー定義型もサポートします。

次の表は、JDBC データ型、ネイティブの Java データ型、SQL データ型と、Oracle エクステンションにより定義される、対応する Java データ型とのデフォルトのマッピングを示します。

標準 JDBC データ型列は、JDBC 1.22 標準がサポートするデータ型のリストです。これらのデータ型は、すべて java.sql.Types クラス内で定義されます。

Java ネイティブ・データ型列は、Java 言語により定義されたデータ型のリストです。

SQL データ型列は、データベースに存在する SQL データ型のリストです。

SQL データ型に相当する Oracle エクステンション一 Java クラス列は、データベース内の各 SQL データ型に対応する oracle.sql.* Java 型のリストです。これらは、oracle.sql.* Java 型の SQL データすべての取得を可能にする Oracle エクステンションです。SQL データ型を oracle.sql データ型にマップすることにより、データの格納と取得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、4-6 ページの「oracle.sql Package のクラス」を参照してください。

SQL データ型の有効なマップが可能な Java データ型すべてのリストは、8-2 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。

表 3-1 JDBC、Java ネイティブ、Oracle データ型間のマップ

標準 JDBC データ型	Java ネイティブ・データ型	SQL データ型	Oracle エクステンションー Java クラス SQL データ型に相当
java.sql.Types.CHAR	java.lang.String	CHAR	oracle.sql.CHAR
java.sql.Types.VARCHAR	java.lang.String	VARCHAR2	oracle.sql.CHAR
java.sql.Types.LONGVARCHAR	java.lang.String	LONG	oracle.sql.CHAR
java.sql.Types.NUMERIC	java.math.BigDecimal	NUMBER	oracle.sql.NUMBER
java.sql.Types.DECIMAL	java.math.BigDecimal	NUMBER	oracle.sql.NUMBER
java.sql.Types.BIT	boolean	NUMBER	oracle.sql.NUMBER
java.sql.Types.TINYINT	byte	NUMBER	oracle.sql.NUMBER
java.sql.Types.SMALLINT	short	NUMBER	oracle.sql.NUMBER
java.sql.Types.INTEGER	int	NUMBER	oracle.sql.NUMBER
java.sql.Types.BIGINT	long	NUMBER	oracle.sql.NUMBER
java.sql.Types.REAL	float	NUMBER	oracle.sql.NUMBER
java.sql.Types.FLOAT	double	NUMBER	oracle.sql.NUMBER
java.sql.Types.DOUBLE	double	NUMBER	oracle.sql.NUMBER
java.sql.Types.BINARY	byte[]	NUMBER	oracle.sql.NUMBER
java.sql.Types.VARBINARY	byte[]	RAW	oracle.sql.RAW
java.sql.Types.LONGVARBINARY	byte[]	LONGRAW	oracle.sql.NUMBER
java.sql.Types.DATE	java.sql.Date	DATE	oracle.sql.DATE
java.sql.Types.TIME	java.sql.Time	DATE	oracle.sql.DATE
java.sql.Types.TIMESTAMP	javal.sql.Timestamp	DATE	oracle.sql.DATE

Oracle JDBC エクステンション型

さらに、SQL データ型に対応する次の JDBC エクステンション（その大半は JDBC 2.0 標準に準拠）がサポートされています。ここでは参考用に紹介するにとどめ、詳細は第 4 章の「Oracle エクステンション」で説明します。表 3-2 に、JDBC エクステンションの Oracle データ型へのマップを示します。

SQL データ型列は、データベースに存在する SQL データ型のリストです。

SQL データ型に対応する JDBC エクステンション列は、Oracle データ型が JDBC 2.0 標準に従ってマップする型のリストです。クラス `oracle.jdbc.driver.OracleTypes.*` は、

JDBC 標準には存在しない Oracle 独自の型の定義を含む、oracle.sql.* のスーパーセットです。

SQL データ型に対応する Oracle エクステンション― Java クラス列は、データベース内の各 SQL データ型に対応する oracle.sql.* Java 型のリストです。これらは、oracle.sql.* Java 型の SQL データすべての取得を可能にする Oracle エクステンションです。oracle.sql.* パッケージの詳細は、4-6 ページの「oracle.sql Package のクラス」を参照してください。

SQL データ型の有効なマップが可能な Java データ型すべてのリストは、8-2 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。

表 3-2 Oracle エクステンション JDBC 型の Oracle データ型へのマップ

SQL データ型	JDBC エクステンション SQL データ型に対応	Oracle エクステンション― Java クラス SQL データ型に対応
ROWID	oracle.jdbc.driver.OracleTypes.ROWID	oracle.sql.ROWID
REF CURSOR カテゴリのユーザー定義型	oracle.jdbc.driver.OracleTypes.CURSOR	java.sql.ResultSet
BLOB	oracle.jdbc.driver.OracleTypes.BLOB	oracle.sql.BLOB
CLOB	oracle.jdbc.driver.OracleTypes.CLOB	oracle.sql.CLOB
BFILE	oracle.jdbc.driver.OracleTypes.BFILE	oracle.sql.BFILE
オブジェクト値	oracle.jdbc.driver.OracleTypes.STRUCT	型マップにオブジェクト値のエントリがない場合 ■ oracle.sql.STRUCT 型マップにオブジェクト値のエントリがある場合 ■ カスタマイズした Java クラス
オブジェクト参照	oracle.jdbc.driver.OracleTypes.REF	oracle.sql.REF を継承するクラス
コレクション（varray およびネストした表）	oracle.jdbc.driver.OracleTypes.ARRAY	oracle.sql.ARRAY

型マッピングの詳細は、第 4 章の「Oracle エクステンション」を参照してください。第 4 章では、次の詳細を知ることもできます。

- パッケージ oracle.sql、oracle.jdbc.driver および oracle.jdbc2
- Oracle ROWID データ型に対応した型エクステンション、および REF CURSOR カテゴリのユーザー定義型
- 型マップをオブジェクト値やコレクションと共に使用する方法

JDBC での Java ストリームの使用方法

この項に含まれるサブセクションは、次のとおりです。

この項では、Oracle JDBC ドライバがさまざまなデータ型の Java ストリームを処理する方法を説明します。データ・ストリームを使うことにより、2 ギガバイトまでの LONG 型の列データを読み取れます。ストリームに関連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。

- バイナリ・ストリームは、データの RAW バイトを返します。これは、`getBinaryStream()` メソッドに対応します。
- ASCII ストリームは、ASCII バイトを ISO-Latin-1 エンコーディングで返します。これは、`getAsciiStream()` メソッドに対応します。
- Unicode ストリームは、Unicode バイトを UCS-2 エンコーディングで返します。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・データを返します。これらのメソッドの詳細は、第 4 章の「Oracle エクステンション」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 ドライバは文字列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 列のデータは読取り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()` メソッドを使います。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、任意のものを使って、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、5-2 ページの「NLS の使用」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 がその状態のまま返されます。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 コードで返されます。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 バイトが返されます。

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

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

たとえば、LONG RAW の値 20 は、16 進では、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返します。データベースに US7ASCII と WE8ISO8859P1 のいずれの文字セットも使用されていない場合、getAsciiStream() へのコールに意味不明のコードが返されます。

getUnicodeStream() を使って 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返されます。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() を使って 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、クライアントのキャラクタ・セットが US7ASCII と WE8ISO8859P1 のいずれでもない場合、getBinaryStream() へのコールには UTF-8 が返されます。クライアントのキャラクタ・セットが US7ASCII または WE8ISO8859P1 の場合は、コールには US7ASCII のバイト・ストリームが返されます。
- ドライバが JDBC Thin で、データベースのキャラクタ・セットが US7ASCII と WE8ISO8859P1 のいずれでもない場合、getBinaryStream() へのコールには UTF-8

が返されます。サーバー側の文字セットが US7ASCII または WE8ISO8859P1 の場合は、コールには US7ASCII のバイト・ストリームが返されます。

キャラクタ・セットに基づく、ドライバのデータ復帰方法の詳細は、5-2 ページの「NLS の使用」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、3-18 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 文字を表すバイト。次の場合に、このバイトは US7ASCII または WE8ISO8859P1 文字を表すこともあります。 ■ クライアントの NLS_LANG の値が US7ASCII または WE8ISO8859P1 の場合。 または ■ データベースのキャラクタ・セットは、US7ASCII または WE8ISO8859P1 です。	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() メソッドを使って 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() を使った LONG RAW データ列の取得 Java を使ったこの例では、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取得します。

次のコードは、名前 LESLIE に関連付けられた 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、データベースとクライアント間の chunk で指定されたサイズの部分に増分的に転送されます。getBinaryStream() へのコールにより返される 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取ります。

getBytes() を使った LONG RAW データ列の取得 この例では、getBinaryStream() のかわりに getBytes() を使って、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、2 ギガバイト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、一度のコールで返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() メソッドを使って 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() を使って定義する場合、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.driver.OracleStatement 型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 往復せずに済みます。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。データはストリーム処理されずに、バイト配列へ書き込まれて返されます。

```
//cast the statement stmt to an OracleStatement
oracle.jdbc.driver.OracleStatement ostmt =
   (oracle.jdbc.driver.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 エクステンションを使って 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 を返しますが、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() または next() 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込むストリームを返します。

注意：リリース 8.1.5 では、CHAR、VARCHAR および RAW データ型には setXXXStream() メソッドは利用できません。

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、通常、ストリームの読込みが終了するまで使用できません。これは、データベースが各行を、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するためです。このため、ストリーム列の後のデータは、ストリームの読込み終了後に読込み可能になります。

たとえば、次の問合せについて考えてみましょう。

```
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>
```

rset.next() をコールすると、JDBC ドライバは、LONG 列の最初の文字の直前で行データの読込みを中止します。次に、ドライバは rset.getAsciiStream() を使用して、データベース接続から LONG 列の文字を直接 Java ストリームとして読み込みます。ドライバは、ストリームからデータの最終バイトを読み込んだ後にだけ、NUMBER データを第 3 列から読み込みます。

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、3-21 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 列のデータだけを回復します。

```
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);
    }
```

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、事前に注意すべき点を説明します。現行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。次の項で説明する、2 つの一般的な注意点は次のとおりです。

- ストリーム・データはアクセス後に使用します。
- SELECT のリスト順にストリーム列をコールします。

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回復するには、列を get するだけでは十分ではありません。列の内容を読み込み、格納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。 問合せを使っ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込む前に、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・データを読み込んだ後、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" エラーを返します。次の例は、この点を例証しています。

```
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
```

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。

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ても、ストリームのクローズおよび廃棄を行えます。データ・ストリームに実行する close() メソッドの詳細は、3-19 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の詳細は 3-20 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。

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。JDBC ドライバは、BLOB（非構造化バイナリ・データ）、CLOB（シングルバイト文字データ）および BFILE（外部ファイル）という、3 タイプの LOB をサポートします。Oracle JDBC ドライバは、CLOB、BLOB および BFILE データのストリームをサポートします。

LOB 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、LOB 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、たいていの 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、接続が有効な間、LOB データには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 により取得した CLOB または BLOB データを操作するには、Oracle エクステンション・クラス oracle.sql.BLOB 内および 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への読込み、出力ストリームから CLOB または BLOB への書込み、CLOB または BLOB の長さの判断、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、4-44 ページの「BLOB および CLOB データの読取りと書込み」を参照してください。

ストリーム BFILE 外部ファイルである BFILE は、ロケータを、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のに使用されます。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 により取得した BFILE を操作するには、Oracle エクステンション・クラス oracle.sql.BFILE のメソッドを使用します。これらのクラスは、BFILE から入力ストリームへの読込み、出力ストリームから BFILE への書込み、BFILE の長さの判断、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 データの使用法の詳細は、4-52 ページの「BFILE データの読取り」を参照してください。

JDBC プログラムでのストアド・プロシージャの使用方法

この項では、Oracle JDBC ドライバがストアド・プロシージャをサポートする方法を説明します。次のサブセクションを含みます。

- 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 エスケープ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 (?,?)}" ) ;
CallableStatement cs2 = conn.prepareCall
                          ( "{? = call func (?,?)}" ) ;
// Oracle Syntax
CallableStatement cs3 = conn.prepareCall
                          ( "begin proc (:1, :2); end;" ) ;
CallableStatement cs4 = conn.prepareCall
                          ( "begin :1 := func(:2,:3); end;" ) ;
```

Oracle 構文の例として、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 = proc.getString(1);
```

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構文は、PL/SQL ストアド・プロシージャをコールする構文と同じです。Java ストアド・プロシージャの使用法の詳細は、『Oracle8*i* Java ストアド・プロシージャ開発者ガイド』を参照してください。

エラー・メッセージと JDBC

例外を処理するため、Oracle JDBC ドライバは `java.sql.SQLException()` を発行します。返されるエラーには、2 つのタイプがあります。最初のタイプは Oracle データベース・エラーで、Oracle データベース自体から返され、エラー番号とエラー内容を説明するテキスト・メッセージで構成されます。これらのエラーは、『Oracle8*i* エラー・メッセージ』で解説されています。

別のタイプは、JDBC ドライバ自体から返されるエラーです。これらのメッセージは、テキスト・メッセージで構成され、エラー番号は含まれません。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。

次のメソッドでエラーを返せます。

- `getMessage()`: 例外を発行したオブジェクトに関連付けられたエラー・メッセージを返します。
- `printStackTrace()`: このオブジェクトの名前および指定された印刷ストリームへのスタックトレースを出力します。

次の例は、`getMessage()` と `printStackTrace()` の両方を使用してエラーを返します。

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

すべてのエラー・メッセージのテキストが国際化されています。つまり、これらは Oracle がサポートするすべての言語および文字セットで使用可能です。これらのエラー・メッセージの一覧は、付録 A の「JDBC エラー・メッセージ」を参照してください。

サーバー側の基本

この項に含まれるサブセクションは、次のとおりです。

- 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
- データベースへの接続

クライアント側ドライバを使ったデータベースへの接続と問合せについては、3-2 ページの「JDBC での最初のステップ」のチュートリアルで説明しました。次の項では、サーバー側ドライバを使ってチュートリアルを実行する場合の基本的な相違点を説明します。サーバー側ドライバの詳細は、5-20 ページの「サーバー上の JDBC: サーバー・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機能します。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、5-21 ページの「サーバー・ドライバのセッションおよびトランザクション・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。データベースへの接続に、`DriverManager.getConnection()` メソッドまたは Oracle 独自の API `defaultConnection()` メソッドを使用できます。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、5-20 ページの「サーバー・ドライバを使ったデータベースへの接続」を参照してください。

アプリケーションの基本およびアプレットの基本

この項に含まれるサブセクションは、次のとおりです。

アプリケーションの基本

Oracle JDBC Thin ドライバまたは JDBC OCI ドライバを使用して、アプリケーションを作成できます。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。

クライアント上で動作可能なアプリケーションは、JDBC サーバー・ドライバを使用することにより、サーバー上でも動作可能です。

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、アプリケーションは Net8 とクライアント・ライブラリのインストールを必要とします。

アプリケーションと暗号化

Oracle OCI ドライバを使用するアプリケーションでは、Net8 ANO（Advanced Networking Option）を利用して暗号化されたデータが扱えます。ANO の詳細は、『Net8 管理者ガイド』を参照してください。

Applet 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。

アプレットとセキュリティ

アプレットは、ダウンロード元のホスト・マシンを除き、ネットワーク接続をオープンできません。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。別のマシン上で稼動中のデータベースに接続する場合は、次のいずれかを実行する必要があります。

- 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します。ブラウザが JDK 1.1.*x* をサポートしている場合、署名付きアプレットを使用できます。署名付きアプレットは、他のマシンへのソケット接続権を要求できます。

これらのトピックは、5-8 ページの「データベースへのアプレットの接続」で詳細に論じられています。

アプレットとファイアウォール

JDBC Thin ドライバは、ファイアウォール経由でデータベースに接続可能です。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詳細は、5-13 ページの「ファイアウォールとアプレットの使用」を参照してください。

アプレットと暗号化

JDBC Thin ドライバを使用するアプレットは、データ暗号化をサポートしません。

アプレットのパッケージ化と実行

アプレットをパッケージ化および実行するには、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、5-16 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。

4

Oracle エクステンション

この章では、標準 JDBC への Oracle エクステンションについて説明します。次のトピックが含まれます。

- Oracle エクステンションの概要
- Oracle JDBC パッケージとクラス
- データ・アクセスとデータの操作 Oracle 型対 Java 型
- LOB の使用
- Oracle エクステンションの追加情報
- Oracle JDBC の注意および制限事項

Oracle エクステンションの概要

Oracle による JDBC のインプリメントは、Sun Microsystems の JDK バージョン 1.0.2 と 1.1.*x* をサポートし、JDBC 1.22（これらのバージョンの JDK に含まれている）に準拠しています。この章では、JDBC 1.22 に対応した Oracle エクステンションについて説明します。内容は、次の 2 つのカテゴリに分かれています。

- JDBC 2.0 のサブセットに準拠した型エクステンション。これは JDK 1.2 の一部です。
- Oracle 固有の型エクステンションおよびパフォーマンス・エクステンション

この項では、Oracle JDBC がこれらのエクステンションをサポートする方法を説明します。作成される Java パッケージおよびデータ型のサポートに関する考慮すべき問題も扱います。

注意 : JDBC OCI、Thin、およびサーバー・ドライバは、同一の機能およびすべての Oracle エクステンションをサポートします。

パッケージ Oracle リリース 8.1.5 は、JDK 1.2 をサポートしません。JDBC 2.0 のインタフェースは、JDK 1.2 に同梱されている `java.sql` パッケージの一部です。JDBC 2.0 タイプを追加 Oracle エクステンションと同様にサポートするため、Oracle JDBC の配布には次の Java パッケージが含まれています。

- `oracle.jdbc2`（標準 JDBC 2.0 インタフェースのサブセット）
- `oracle.sql`（すべての Oracle 型エクステンションをサポートするクラス）
- `oracle.jdbc.driver`（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）

4-5 ページの「Oracle JDBC パッケージとクラス」に、これらのパッケージとそのクラスの詳細な説明があります。

Oracle データ型のサポート Oracle JDBC エクステンションの主要な特色は、`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、ロー 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 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、4-6 ページの「oracle.sql Package のクラス」を参照してください。

特定の oracle.sql.* データ型クラスに固有の情報は、4-41 ページの「LOB の使用」および 4-100 ページの「追加の型エクステンション」を参照してください。

Oracle オブジェクトのサポート Oracle8 の中で最も注目すべき型は、Oracle オブジェクトです。Oracle8 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。構造化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（Java 形式と oracle.sql.* 形式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

使用する Oracle オブジェクトに対応する Java クラスを手動で作成する場合、Oracle8*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、コマンド行オプションを指定することで、この作業をシームレスに実行します。

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・データを取得する際に、どの Java クラスのインスタンシエートおよび移入を行うかを決定します。

Oracle オブジェクト・データ型に対応して作成された各 Java クラスは、サポートする 2 つのインタフェース（JDBC 標準 SQLData インタフェースまたは Oracle CustomDatum インタフェース）のどちらかをインプリメントする必要があります。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。現在、JPublisher は CustomDatum インタフェースだけをサポートしています。

JPublisher は、Java クラスの get 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 アプリケーション内に取得します。JPublisher ユーティリティの詳細は、『Oracle8*i* Jpublisher User's Guide』を参照してください。

4-56 ページの「Oracle オブジェクト型の使用」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。型名が現行の名前領域内（つまり現行のスキーマ内）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 ドライバは現行の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現行のスキーマに属している場合、（スキーマを指定せずに）型名だけを指定できます。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。
- JDBC ドライバは、スキーマ名にはドット（"."）が含まれないと仮定します。
- JDBC ドライバは、スキーマ名または型名に二重引用符を使用することを許可しません。

Oracle JDBC パッケージとクラス

この項では、Oracle JDBC エクステンションをサポートする 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。この項に含まれるサブセクションは、次のとおりです。

- oracle.jdbc2 パッケージのクラス
- oracle.sql Package のクラス
- oracle.jdbc.driver パッケージのクラス

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

oracle.jdbc2 パッケージのクラス

`oracle.jdbc2` パッケージには、Oracle による標準 JDBC 2.0 インタフェースの実装が含まれます。JDBC 2.0 インタフェースは、JDK 1.2 に含まれる `java.sql` パッケージの一部です。ただし、現在ドライバが JDK 1.2 をサポートしていないため、これらのインタフェースは Oracle 1.0.2 および 1.1.*x* ドライバから `oracle.jdbc2` パッケージとして利用可能です。このパッケージには、Oracle ドライバがサポートする、JDK 1.2`java.sql` パッケージの JDBC 2.0 機能が含まれます。

次のインタフェースは、JDBC 2.0 に準拠した Oracle 型エクステンション対応の `oracle.sql.*` 型クラスによりインプリメントされています。これらのインタフェースは、Sun Microsystems が公開したインタフェースと等価であり、`oracle.jdbc2` バージョンでの新機能の追加はありません。

- `oracle.jdbc2.Array` は、`oracle.sql.ARRAY` によりインプリメントされます。
- `oracle.jdbc2.Struct` は、`oracle.sql.STRUCT` によりインプリメントされます。
- `oracle.jdbc2.Ref` は、`oracle.sql.REF` によりインプリメントされます。
- `oracle.jdbc2.Clob` は、`oracle.sql.CLOB` によりインプリメントされます。
- `oracle.jdbc2.Blob` は、`oracle.sql.BLOB` によりインプリメントされます。

さらに、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って作成するユーザーのために、Oracle は次の標準 JDBC 2.0 インタフェースも含めました。

- Oracle オブジェクトにマップするクラスによりインプリメントされた `oracle.jdbc2.SQLData`。ユーザーはこのインプリメンテーションを提供する必要があります。
- オブジェクト・データを読み込むクラスによりインプリメントされた `oracle.jdbc2.SQLInput`。Oracle は JDBC ドライバが使用する `SQLInput` クラスを提供します。

- オブジェクト・データを書き込むクラスによりインプリメントされた oracle.jdbc2.SQLOutput。Oracle は JDBC ドライバが使用する 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。他の機能は、Oracle CustomDatum インタフェースで、これは oracle.sql パッケージに含まれます。SQLData、SQLInput、および SQLOutput の詳細は、4-63 ページの「SQLData インタフェース」を参照してください。

注意 : SQLData インタフェースのかわりに CustomDatum インタフェースを使用することをお薦めします。CustomDatum は、JPublisher ユーティリティ（Oracle オブジェクトに対応した CustomDatum クラスを自動生成する）や SQLJ 等の Oracle Java 製品が提供する他の機能とよりスムーズに連携動作します。

---

oracle.sql Package のクラス

oracle.sql パッケージは、SQL 形式のデータへのダイレクト・アクセスをサポートします。また、主に Oracle SQL データ型にマップするクラスで構成されます。

これらのクラスは、Oracle SQL 型への Java マッピングを提供する、ロー SQL データ用のラッパー・クラスです。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 プリミティブ型の場合、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 データ型のクラスは、すべてのデータ型に共通した機能をカプセル化したスーパークラスである oracle.sql.Datum を継承します。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 4-1 に示すように、oracle.sql.Datum を継承すると同時に、oracle.jdbc2 パッケージの標準 JDBC 2.0 インタフェースをインプリメントしています。

表 4-1 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 4-1 Oracle データ型クラス

Java クラス	Oracle SQL 型（およびその説明）および JDBC 2.0 対応の場合のインプリメントされたインタフェース
oracle.sql.STRUCT	STRUCT（オブジェクト） JDBC 2.0、oracle.jdbc2.Struct をインプリメント
oracle.sql.REF	REF（オブジェクト参照） JDBC 2.0、oracle.jdbc2.Ref をインプリメント

表 4-1 Oracle データ型クラス (続き)

Java クラス	Oracle SQL 型（およびその説明）および JDBC 2.0 対応の場合のインプリメントされたインタフェース
oracle.sql.ARRAY	varray またはネストされた表（コレクション） JDBC 2.0、oracle.jdbc2.Array をインプリメント
oracle.sql.BLOB	BLOB（ラージ・バイナリ・オブジェクト） JDBC 2.0、oracle.jdbc2.Blob をインプリメント
oracle.sql.CLOB	CLOB（ラージ文字列オブジェクト） JDBC 2.0、oracle.jdbc2.Clob をインプリメント
oracle.sql.BFILE	BFILE（外部ファイル）
oracle.sql.CHAR	CHAR、VARCHAR2
oracle.sql.DATE	DATE
oracle.sql.NUMBER	NUMBER
oracle.sql.RAW	RAW
oracle.sql.ROWID	ROWID（行識別子）

次の項で、表 4-1 で示した各クラスについて説明します。Oracle 拡張型の詳細は、（STRUCT、REF、ARRAY、BLOB、CLOB、BFILE、および ROWID）は、4-41 ページの「LOB の使用」、4-76 ページの「Oracle オブジェクト参照の使用」、4-79 ページの「配列の使用」、および 4-100 ページの「追加の型エクステンション」で説明します。

注意：

- オブジェクトだけを対象として使用される STRUCT クラスと、一般的な用語である構造化オブジェクト（オブジェクトまたはコレクションのいずれかを指すことが多い）を混同しないように注意してください。ARRAY クラスは、コレクションをサポートします。コレクションは、varray またはネストされた表のいずれかの機能を果たします。
- 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.* 型にデータを取得する場合の詳細は、4-29 ページの「データ・アクセスとデータの操作 Oracle 型対 Java 型」を参照してください。
- LONG、LONG RAW、または REF CURSOR SQL 型には、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 データは、標準 JDBC メソッドである getAsciStream()、getBinaryStream()、および getUnicodeStream() を使って入力ストリームとして取得します。REF CURSOR 型の場合は、getCursor() 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、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。詳細は、4-13 ページの「クラス oracle.sql.ARRAY」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。詳細は、4-9 ページの「クラス oracle.sql.STRUCT」を参照してください。
- oracle.sql.CharacterSet および oracle.sql.CharacterSetFactory クラス：キャラクタ・セット・オブジェクトの作成に使用します。キャラクタ・セット・オブジェクトは、oracle.sql.CHAR オブジェクトの作成に使用します。詳細は、4-17 ページの「クラス oracle.sql.CHAR」を参照してください。
- oracle.sql.CustomDatum および oracle.sql.CustomDatumFactory インタフェース：Oracle オブジェクト・サポートの Oracle CustomDatum シナリオをインプリメントする 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。CustomDatum の詳細は、4-68 ページの「CustomDatum インタフェース」を参照してください。）

これらのクラスに関する追加情報は、Javadoc を参照してください。この項では、さらに oracle.sql.* クラスについても説明します。

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
- SQL データをバイト配列として返す `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返します。たとえば、Oracle `ROWID` を `oracle.sql.ROWID` として返します。
- `stringValue()` または `intValue()` メソッド。これらのメソッドは、適切な場所で、SQL データを `String` または `int` に変換します。
- 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得する LOB クラスのメソッドやオブジェクト参照を使ってオブジェクト・データの取得や設定を行う `REF` クラスのメソッドなど）に適した `get` および `set` メソッドを利用します。

これらのクラスの詳細は、Javadoc を参照してください。

クラス oracle.sql.STRUCT

指定された任意の Oracle オブジェクト型で、接続の型マップで Java クラスへのマッピングを指定しない場合、オブジェクト型から取得したデータは、`oracle.sql.STRUCT` クラスのインスタンス内で Java により実体化されます。

`STRUCT` クラスは、標準 JDBC 2.0 `oracle.jdbc2.Struct` クラスをインプリメントし、`oracle.sql.Datum` を継承します。

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・バイト用ラッパーで、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を含みます。`STRUCT` オブジェクトも、Oracle オブジェクトの SQL 型名を含みます。

`STRUCT` クラスを使用するだけで十分な場合があるとしても、大抵はカスタムの Java 型定義クラスを作成して、使用する Oracle オブジェクトにマップすることになるでしょう（4-57 ページの「STRUCT オブジェクトの使用方法」を参照）。`STRUCT` の属性は、`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、また `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実体化できます。`oracle.sql.*` 形式を使用すると、一般に `oracle.sql.*` データ型クラスを使用した場合と同じ利点があります。

- `STRUCT` クラスは、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。これは、データを操作しても表示する必要がない場合に有用です。

- これにより、Java アプリケーションは自由度の高い方法でデータを復元できます。

注意：

- 値配列の要素は、一般的な Datum 型の要素であっても、特定の属性に適した oracle.sql.* 型（たとえば、CHAR データの場合は oracle.sql.CHAR）と関連付けられたデータを含みます。要素は、適切な oracle.sql.* 型としてキャストできます。
- JDBC ドライバは、STRUCT オブジェクトの値配列内でネストしたオブジェクトを、STRUCT 自体のインスタンスとして実体化します。
- oracle.sql.STRUCT クラスの特定の機能およびメソッドの詳細は、Javadoc を参照してください。

手動で STRUCT オブジェクトを作成して、準備済みの文やコール可能文へ渡したい場合もあります。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。STRUCT オブジェクトの作成の詳細は、4-11 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 クラスには、次のメソッドが含まれます。

- getAttributes(): 型マップを使って（定義済みの場合）データの実体化に使用する Java クラスを決定し、値配列から値を取得します。理論的には、getAttributes() は属性値を含む Java 配列を返します。属性値の型は、基礎となる型で getObject() へ実行したコールが返す型と同じです。つまり、基礎となる型に対応したデフォルトの JDBC 型です。

  たとえば、定義済みの SQL 型 PERSON があり、CHAR 型の名前属性および NUMBER 型の年齢属性が含まれるとします。getAttributes() を使って PERSON のオブジェクト属性を取得する場合、名前は Java の String 型で、年齢は Java の BigDecimal 型で返されます。

  ネストしたオブジェクト上で getAttributes() をコールする場合、接続のデフォルト型マップを使用しなくても、オプションで型マップ（java.util.Map オブジェクト）を指定できます。

- getOracleAttributes(): 値配列の値を oracle.sql.* オブジェクトとして取得します。
- getSQLTypeName(): この STRUCT が表す Oracle オブジェクトの完全修飾型名（*schema.sql_type_name*）を返します。
- getDescriptor(): この STRUCT オブジェクトに対応した StructDescriptor オブジェクトを返します（StructDescriptor クラスに関しては 4-11 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）。
- getConnection(): 現行の接続を返します。

- getDescriptor(): Oracle オブジェクト型を識別する OracleType を返します。
- getMap(): 現行の型マップを返します。
- isConvertibleTo(Class): データ・オブジェクトが特定のクラスに変換可能かどうかを決定します。
- makeJdbcArray(int): データの JDBC 配列表現を返します。
- setDatumArray(Datum[]): Datum 型の配列を設定します。
- setDescriptor(StructDescriptor): 記述子を設定します。
- stringValue(): データ・オブジェクトの String 型表現に変換します。
- toBytes(): 属性を表すバイトを、実際にデータベースで使用される形式にパックします。
- toClass(Class): SQL 構造化型で使用する通常のアルゴリズムを特定の Java クラスに適用します。
- toJdbc(): 現行のマップを参考にして変換先のクラスを決定し、次いで toClass() を使用します。
- toJdbc(Dictionary): マップを参考にして、変換先のクラスを決定し、次いで toClass() を使用します。
- toSTRUCT(Object, OracleConnection): 入力用の Java オブジェクトから、対応する STRUCT オブジェクトを返します。

STRUCT オブジェクトと記述子の作成 oracle.sql.STRUCT オブジェクトを作成するには、使用する Oracle オブジェクト型に対応した STRUCT 記述子がまず存在する必要があります。この記述子は、oracle.sql.StructDescriptor クラスのオブジェクトです。

StructDescriptor は、SQL 構造化オブジェクト（Oracle）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 つだけです。

ドライバは、STRUCT 記述子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。Oracle JDBC エクステンションは、StructDescriptor を新規に作成するか、既存のものを返す、静的な createDescriptor() メソッドを提供します。

StructDescriptor オブジェクトを作成するには、Java 文字列パラメータに Oracle オブジェクト型の SQL 型名を指定して渡し、接続オブジェクトを StructDescriptor.createDescriptor() メソッドに渡します。

```
StructDescriptor structdesc = StructDescriptor.createDescriptor(sql_type_name,
connection);
```

sql_type_name には Oracle オブジェクト型の名前（EMPLOYEE など）を含む Java 文字列が、connection には使用する接続オブジェクトが当てはまります。

STRUCT オブジェクトを新規に作成する場合は、StructDescriptor オブジェクトをコールすることもできます。StructDescriptor オブジェクトを新規作成する場合は、Java 文字列パラメータに Oracle オブジェクト型の SQL 型名と使用する接続オブジェクトを指定します。

```
StructDescriptor structdesc = new StructDescriptor(sql_type_name, connection);
```

STRUCT オブジェクトを作成するには、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 get メソッドの使用方法 STRUCT 記述子は、型オブジェクトとして参照可能です。これは、STRUCT 記述子に、オブジェクト型のタイプ・コードと型名、および指定した型と相互変換を行う方法に関する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 つだけ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、getName() メソッドが含まれます。このメソッドは、Oracle オブジェクトの完全修飾された SQL 型名を（つまり、CORPORATE.EMPLOYEE のような *schema.sql_type_name* の形式で）返します。

埋込みオブジェクト 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得する場合、同じ変換規則に従い、型マップ（利用可能な場合）またはデフォルトのマッピングを使用します。

クラス oracle.sql.REF

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、oracle.sql.* データ型のクラスすべてと同様、oracle.sql.Datum のサブクラスです。このクラスは、標準 JDBC 2.0 oracle.jdbc2.Ref インタフェースをインプリメントします。

REF を選択すると、オブジェクトへのポインタだけが取得されます。オブジェクトの実体化は行いません。ただし、実体化を行うメソッドは存在します。

oracle.sql.REF クラスには、次のメソッドが含まれます。

- getValue(): オブジェクトの属性を（必要に応じて型マップを使用して）取得します。
- setValue(): オブジェクトの属性を（必要に応じて型マップを使用して）設定します。
- getBaseTypeName(): 参照先の項目の完全修飾された SQL 構造化型名を返します。

OraclePreparedStatement および OracleCallableStatement クラスの setREF() および setRef() メソッドでは、REF オブジェクトを、準備済みの文へ入力パラメータとして渡すことができます。同様に、OracleCallableStatement、OracleResultSet の getREF() および getRef() メソッドでは、REF オブジェクトを出力パラメータとして渡すことができます。

JDBC を使って REF オブジェクトを作成することはできません。

REF オブジェクトの使用法の詳細は、4-76 ページの「Oracle オブジェクト参照の使用」を参照してください。

クラス oracle.sql.ARRAY

oracle.sql.ARRAY クラスは、Oracle コレクション、varray またはネストした表のいずれかをサポートします。データベースから varray またはネストした表を選択すると、JDBC ドライバは ARRAY クラスのオブジェクトとして実体化します。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 を継承し（すべての oracle.sql.* クラスの場合と同様）、oracle.jdbc2.Array 標準 JDBC 2.0 配列インタフェースをインプリメントします。

ARRAY オブジェクトを手動で作成して、準備済みの文またはコール可能文に渡し、データベースへの挿入等の操作を行う場合も考えられます。このような場合、ArrayDescriptor オブジェクトを使用します。このオブジェクトについては、4-13 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。

ARRAY クラスには、次のメソッドが含まれます。

- 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。
- getArrayDescriptor(): この配列に属する ArrayDescriptor オブジェクトを返します（ArrayDescriptor クラスの詳細は、4-13 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）。
- getBaseType(): 配列要素に対応した SQL タイプ・コードを返します（タイプ・コードの詳細は、4-26 ページの「クラス oracle.jdbc.driver.OracleTypes」を参照してください）。
- getSQLTypeName(): 配列要素の SQL 型名を返します。
- getBaseTypeName(): 名前の付いた型（Oracle オブジェクトなど）の場合、特定の型名（たとえば、EMPLOYEE）を返します。
- getResultSet(): 配列を結果セットとして実体化します。

ARRAY オブジェクトと記述子の作成 OraclePreparedStatement または OracleCallableStatement クラスの setARRAY() メソッドを使用すると、配列を準備

済みの文に入力パラメータとして渡すことができます。まず、oracle.sql.ArrayDescriptor オブジェクトである *array descriptor* を作成する必要があります。次に、配列の引渡し用に oracle.sql.ARRAY オブジェクトを作成します。

ArrayDescriptor オブジェクトは、配列の SQL 型を記述します。ただし、任意の SQL 型 1 つに対し、必要な配列記述子は 1 つだけです。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、強く型付けされています。Oracle は、名前付き配列だけ、つまり SQL 型名の指定された配列をサポートします。たとえば、配列を CREATE TYPE 文を使って次のように作成します。

```
CREATE TYPE num_varray AS varray(22) OF NUMBER(5,2);
```

コレクション型に対応した SQL 型名は、num_varray です。

注意：コレクション型の名前は、要素の型名と関係がありません。

例：CREATE TYPE person AS object (c1 NUMBER(5), c2 VARCHAR2(30));

CREATE TYPE array_of_persons AS varray(10) OF person;

前の文では、コレクション型の SQL 型名は、array_of_persons です。コレクションの要素の SQL 型名は、person です。

ArrayDescriptor オブジェクトを組み立てるために、コレクション型と 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・データを収集する）の SQL 型名をコンストラクタに渡します。

```
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。

ARRAY オブジェクトを組み立てるには、配列記述子、接続オブジェクト、および配列に含め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 にはオブジェクトの 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 プリミティブの配列たとえば、int[]。

- Java オブジェクトの配列（たとえば、*xxx*[]。*xxx* には Java オブジェクト型の名前が当てはまります。）たとえば、Integer[]。

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。
- ARRAY および ArrayDescriptor クラスの機能の詳細は、Javadoc を参照してください。

ArrayDescriptor の get メソッドの使用方法 配列記述子は、型オブジェクトとして参照可能です。これは、配列記述子が、配列の SQL 型名、配列要素のタイプ・コード、および配列が STRUCT である場合は要素の型名の情報を含んでいることを意味しています。配列記述子は、指定された型との相互変換方法の情報も含んでいます。任意の型 1 つにつき、必要な配列記述子は 1 つだけです。その後、記述子を使用してその型の配列を必要な数だけ作成できます。

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。

- getBaseType(): この配列記述子に関連付けられた整数型を返します。（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、4-20 ページの「oracle.jdbc.driver パッケージのクラス」で説明されています。）
- getBaseName(): この配列要素が STRUCT、REF、またはコレクションである場合は、文字列を、配列要素と関連付けられた型名と共に返します。

注意: 配列の要素を ARRAY 型にすることはできません。コレクションに collection 型の要素を含めることはできません。ただし、Oracle オブジェクトおよび STRUCTS には、Java 型の ARRAY（SQL 型の collection）を含めることができます。

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

BLOB、CLOB、および BFILE（これらはすべて LOB として参照される）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に対応しています。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、LOB を次のいくつかの方法でサポートします。

- BLOB は、非構造化ラージ・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 文を使って、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、ロケータだけであることに留意してください。データの取得には、追加のステップが必要です。これは、4-41 ページの「LOB の使用」で説明されています。

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。

---

oracle.sql.BLOB クラスには、次のメソッドが含まれます。

- getBinaryOutputStream(): BLOB データを返します。
- getBinaryStream(): この Blob のインスタンスによりバイト・ストリームとして指定された BLOB を返します。
- getBytes(): BLOB データ内の指定された位置から読取りを開始し、提供されたバッファ内に格納します。
- length(): BLOB の長さをバイトで返します。
- position(): BLOB 内で指定されたパターンが始まるバイト位置を決定します。
- 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。

oracle.sql.CLOB クラスには、次のメソッドが含まれます。

- getAsciiOutputStream(): ASCII ストリームから CLOB データの書込みを行います。
- getAsciiStream(): Clob オブジェクトにより指定された CLOB 値を Ascii バイト・ストリームとして返します。
- getCharacterOutputStream(): Unicode ストリームから CLOB データの書込みを行います。
- getCharacterStream(): CLOB データを Unicode 文字のストリームとして返します。
- getChars(): CLOB データの指定された位置から文字を取得し、文字配列に格納します。
- length(): CLOB の長さを文字数で返します。
- position(): CLOB 内で、指定されたサブ文字列の始まる文字位置を決定します。
- 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サブ文字列を取得します。
- 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。

oracle.sql.BFILE クラスには、次のメソッドが含まれます。

- openFile(): 外部ファイルをオープンします。
- 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・ストリームとして返します。
- getBytes(): 外部ファイルの指定された位置から読取りを開始し、提供されたバッファ内に格納します。
- getName(): 外部ファイルの名前を取得します。
- getDirAlias(): 外部ファイルのディレクトリ別名を取得します。
- length(): BFILE の長さをバイトで返します。
- position(): 指定されたバイト・パターンの始まるバイト位置を決定します。

注意 : BFILE への書込みはできません。読取りだけが可能です。

クラス oracle.sql.CHAR

CHAR クラスには、文字データを NLS 変換する特殊機能があります。CHAR クラスの主要属性、および CHAR オブジェクトの構築時に常に渡されるパラメータは、文字データの提示に使用される NLS キャラクタ・セットです。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。

ドライバが構築する、または返す 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。Oracle8 オブジェクトである CHAR オブジェクトは、データベース・キャラクタ・セットで返されます。

データベースから文字データの読取りが行われると、JDBC は CHAR オブジェクトの構築および移入を実行します。さらに、CHAR オブジェクトを独自に作成する場合もありえます (たとえば、準備済みの文への引渡しを行う場合)。

CHAR オブジェクトを構築する場合、キャラクタ・セット情報を CHAR オブジェクトに oracle.sql.CharacterSet クラスのインスタンス経由で提供する必要があります。CharacterSet クラスの各インスタンスは、Oracle がサポートする NLS 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、『Oracle8*i* NLS ガイド』を参照してください。

Oracle がサポートしないキャラクタ・セットに基づく CHAR オブジェクトを使用する場合、JDBC ドライバはそのオブジェクトに対してキャラクタ・セットの変換を行うことはできま

せん。たとえば、OraclePreparedStatement.setOracleObject() コールでは CHAR オブジェクトは使用できません。

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 インスタンスを作成します。このメソッドは、キャラクタ・セット・クラスのファクトリです。このメソッドは、Oracle がサポートするキャラクタ・セットに対応する整数 OracleId を入力として取ります。

   例 :

   ```
   int oracleId = CharacterSet.JA16SJIS_CHARSET; // this is character set 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 OracleId を保持します。無効な OracleId を入力しても、例外は発行されません。そのキャラクタ・セットを使用しようとすると、予期しない結果を受け取ります。キャラクタ・セットおよびキャラクタ・セット ID の詳細は、『Oracle8*i* NLS ガイド』を参照してください。

2. CHAR オブジェクトを構築します。コンストラクタに文字列（または文字列を表すバイト）および CharacterSet オブジェクトを渡します。CharacterSet オブジェクトは、キャラクタ・セットに基づくバイトの解釈方法を指定します。

   例 :

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

   CHAR クラスには、複数のコンストラクタが含まれます。これらのコンストラクタは、CharacterSet オブジェクトに加え、文字列、バイト配列、オブジェクトを入力として取ることができます。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、Javadoc の CHAR クラスに関する説明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唯一の方法は、make() メソッドを使用することです。
- 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 クラスを継承することを、オラクル社は意図しておらず、推奨もしていません。

CHAR クラスは、文字データの文字列への翻訳用としてこれらのメソッドを提供しています。

- 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 オブジェクトを返します。キャラクタ・セットが認識されない（つまり、無効な OracleID を入力した）場合、getString() は SQLException を発行します。
- toString():getString() と同一ですが、キャラクタ・セットが認識されない（つまり、無効な OracleID を入力した）場合、toString() は CHAR データの 16 進値表現を返し、SQLException を発行することはありません。
- getStringWithReplacement():getString() と同一ですが、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。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、大抵の場合クエスチョン・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、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。このクラスのメソッドを使っ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・セットに変換（またはその逆）する必要があります。データを変換する際、ドライバは Oracle の各国語サポート（NLS）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、5-2 ページの「NLS の使用」を参照してください。NLS の詳細は、『Oracle8*i* NLS ガイド』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 の一部ではない、プリミティブな SQL データ型にマップします。これらのクラスは、対応する JDBC Java 型との相互変換を行います。詳細は、Javadoc を参照してください。

クラス oracle.sql.ROWID

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 を更新することはできません。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インプリメントしていません。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 進値表現を返す 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、4-88 ページの「Oracle エクステンションの追加情報」を参照してください。

oracle.jdbc.driver パッケージのクラス

oracle.jdbc.driver パッケージには、oracle.sql 形式でのデータ・アクセスを可能にする拡張機能を提供するクラスが含まれています。また、これらのクラスが提供する Oracle 固有のエクステンションは、oracle.sql.* オブジェクトを使った未加工の SQL 形式データへのアクセスを可能にします。

表 4-2 に、このパッケージに含まれる接続、文、および結果セット用の主要クラスの一覧を示します。

表 4-2 接続、文、および結果セット・クラス

クラス	主要機能
OracleDriver	java.sql.Driver のインプリメンテーション
OracleConnection	Oracle statement オブジェクトを返すメソッド。現行の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・エクステンションを設定するメソッド（java.sql.Connection をインプリメント）。
OracleStatement	文ごとに Oracle パフォーマンス・エクステンション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 をインプリメント）。
OraclePreparedStatement	oracle.sql.* 型を準備済みの文にバインドする set メソッド（java.sql.PreparedStatement をインプリメントし、OracleStatement を継承）。

表 4-2 接続、文、および結果セット・クラス (続き)

クラス	主要機能
OracleCallableStatement	データを oracle.sql 形式で取得する get メソッド。oracle.sql.* 型をコール可能な文にバインドする set メソッド（OraclePreparedStatement を継承)、(java.sql.CallableStatement をインプリメントし、PreparedStatement を継承)。
OracleResultSet	oracle.sql 形式でデータを取得する get メソッド (java.sql.ResultSet をインプリメント)。
OracleResultSetMetaData	Oracle 結果セットの情報を取得するメソッド (java.sql.ResultSetMetaData をインプリメント)。

oracle.jdbc.driver パッケージには、次のクラスも含まれます。

- ストリーム・クラス
- OracleTypes クラス

ストリーム・クラスは、標準 Java ストリーム・クラスを継承し、Oracle LOB、LONG、および LONG RAW データの読取りと書込みを行います。

OracleTypes は、SQL 型を識別する整数定数を定義します。標準型の場合、OracleTypes は同じ値を標準 java.sql.Types 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。

この項の残りの部分で、oracle.jdbc.driver パッケージのクラスについて説明します。これらのクラスを使った Oracle 型エクステンションへのアクセスの詳細は、4-29 ページの「データ・アクセスとデータの操作 Oracle 型対 Java 型」を参照してください。

クラス oracle.jdbc.driver.OracleDriver

このクラスを使って、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 インタフェースをインプリメントするクラス）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 クラスを使っ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 での最初のステップ」を参照してください。

クラス oracle.jdbc.driver.OracleConnection

このクラスは、標準 JDBC 接続機能を継承して、Oracle 文オブジェクトの作成と復帰、Oracle パフォーマンス・エクステンション用のフラグやオプションの設定、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、バッチ処理の更新、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・エクステンションの詳細は、4-88 ページの「パフォーマンス・エクステンション」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。
- getTransactionIsolation(): この接続の現行の分離モードを取得します。
- setTransactionIsolation(): TRANSACTION_* 値を使って、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.driver.OracleConnection メソッドは、Oracle 定義のエクステンションです。

- getDefaultExecuteBatch(): この接続に対応したデフォルトの更新バッチ処理値を取得します。
- setDefaultExecuteBatch(): この接続に対応したデフォルトの更新バッチ処理値を設定します。
- 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得します。
- 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 を返します。
- setRemarksReporting(): TABLE_REMARKS のレポート処理を有効または無効にします。
- getTypeMap(): この接続の型マップを取得します（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）。

クラス oracle.jdbc.driver.OracleStatement

このクラスは、標準 JDBC 文の機能を拡張した、OraclePreparedStatement および OracleCallableStatement クラスのスーパークラスです。拡張機能には、文単位で Oracle パフォーマンス・エクステンション用のフラグおよびオプションの設定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 クラスと対照的な役割を果たします。

4-88 ページの「パフォーマンス・エクステンション」パフォーマンス・エクステンションを記述します。行のプリフェッチおよび列型の定義が含まれます。

主要メソッドは、次の通りです。

- executeQuery(): データベースへの問合せを実行し、OracleResultSet オブジェクトを返します。
- getResultSet(): OracleResultSet オブジェクトを取得します。
- close(): 現行の文をクローズします。

これらの oracle.jdbc.driver.OracleStatement メソッドは、Oracle 定義のエクステンションです。

- defineColumnType(): 特定のデータベース表の列からデータを取得する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得します。
- setRowPrefetch(): この文の行プリフェッチ値を設定します。

クラス oracle.jdbc.driver.OraclePreparedStatement

このクラスは、標準 JDBC 準備済み文の機能を拡張します。また OracleStatement クラスのサブクラスであると共に、OracleCallableStatement クラスのスーパークラスでもあります。拡張機能には、oracle.sql.* 型およびオブジェクトを準備済みの文にバインドする set メソッド、および文単位で Oracle パフォーマンス・エクステンションをサポートするメソッドが含まれます。

4-88 ページの「パフォーマンス・エクステンション」で、データベース更新バッチ処理を含むパフォーマンス・エクステンションについて説明します。

主要メソッドは、次の通りです。

- getExecuteBatch(): この文の更新バッチ処理値を取得します。
- setExecuteBatch(): この文の更新バッチ処理値を設定します。
- setOracleObject(): oracle.sql.* データを準備済みの文に oracle.sql.Datum オブジェクトとしてバインドする、汎用の set メソッド。

- setXXX(): setBLOB() 等の、特定の oracle.sql.* 型を準備済みの文にバインドする、set メソッド。oracle.sql.* 型で利用可能なすべての setXXX() メソッドの詳細は、Javadoc を参照してください。
- setCustomDatum(): CustomDatum 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）準備済みの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。
- close(): 現行の文をクローズします。

クラス oracle.jdbc.driver.OracleCallableStatement

このクラスは、標準 JDBC コール可能文の機能を拡張する、OracleStatement および OraclePreparedStatement クラスのサブクラスです。拡張機能には、構造化オブジェクトと oracle.sql.* オブジェクトを現行の文にバインドする set メソッド、およびデータを oracle.sql.* オブジェクト内に取得する get メソッドが含まれます。

主要メソッドは、次のとおりです。

- getOracleObject(): データを oracle.sql.Datum 内に取得する、汎用の get メソッド。必要に応じて特定の oracle.sql.* 型にキャストできます。
- getXXX(): データを特定の oracle.sql.* オブジェクト内に取得する、getCLOB() などの get メソッド。oracle.sql.* 型で使用可能なすべての getXXX() メソッドの詳細は、Javadoc を参照してください。
- setOracleObject(): oracle.sql.* データを oracle.sql.Datum オブジェクトとしてコール可能文にバインドする、汎用の set メソッド。
- setXXX(): OraclePreparedStatement から継承した set メソッド（setBLOB() など）。特定の oracle.sql.* オブジェクトをコール可能文にバインドします。oracle.sql.* 型で利用可能なすべての setXXX() メソッドの詳細は、Javadoc を参照してください。
- 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（*schema.type*）名が当てはまります。
- 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（文の出力変数の、他のパラメータを元にした相対位置）および整数値の SQL 型（oracle.jdbc.driver.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。このメソッドには、名前付きタイプ（つまり、SQL タイプ・コードが OracleTypes.REF、STRUCT、または ARRAY である

場合）だけに使用するバージョンがあります。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 オブジェクト型の名前）を取ります。

- close(): 現行の結果セット（存在する場合）および現行の文をクローズします。

クラス oracle.jdbc.driver.OracleResultSet

このクラスは、標準 JDBC 結果セットの機能を拡張し、get メソッドをインプリメントしてデータを oracle.sql.* オブジェクト内に取得します。

主要メソッドは、次のとおりです。

- getOracleObject(): データを oracle.sql.Datum 内に取得する、汎用の get メソッド。必要に応じて特定の oracle.sql.* 型にキャストできます。
- getXXX(): データを oracle.sql.* オブジェクト内に取得する、getCLOB() などの get メソッド。
- next(): 結果セットの次の行へ進みます。

クラス oracle.jdbc.driver.OracleResultSetMetaData

このクラスは、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。

主要メソッドは次のとおりです。

- getColumnCount(): Oracle 結果セット内の列数を返します。
- getColumnName(): Oracle 結果セット内の指定された列名を返します。
- getColumnType(): Oracle 結果セット内の指定された列の SQL 型を返します。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 を返します。
- getColumnTypeName(): 列内に格納されたデータの SQL 型名を返します。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 型名を返します。列が REF データを格納している場合、このメソッドは REF が指すオブジェクトの SQL 型名を返します。
- getTableName(): 選択された Oracle 結果セット列を含む表の名前を返します。

Oracle ストリーム・クラス

Oracle は、標準 Java ストリーム・クラスを継承する多くのストリーム・クラスを使用して、Oracle データベースへの直接書込み等の特殊機能を提供します。JDBC ドライバは、oracle.jdbc.driver パッケージに含まれるこれらのクラスを使用しますが、Java アプリケーションのプログラマがこれらのクラスを使用することは意図されていません。Java スト

リームの詳細は、3-13 ページの「JDBC での Java ストリームの使用方法」を参照してください。

クラス oracle.jdbc.driver.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.driver.OracleTypes クラスには、標準 Java java.sql.Types クラスのコピー、および次の補足的な Oracle 型エクステンションが含まれます。

- OracleTypes.STRUCT
- OracleTypes.REF
- OracleTypes.ARRAY
- OracleTypes.BLOB
- OracleTypes.CLOB
- OracleTypes.BFILE
- OracleTypes.ROWID

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。

OracleType と出力パラメータの登録 OracleTypes クラスの SQL 型は、java.sql.CallableStatement および oracle.jdbc.driver.OracleCallableStatement クラスの registerOutParameter() メソッドで使用する出力パラメータの 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。

```
CallableStatement.registerOutParameter(int index, int sqlType)

CallableStatement.registerOutParameter(int index, int sqlType, int scale)

OracleCallableStatement.registerOutParameter(int index, int sqlType, String sql_
name)
```

これらのプロトタイプでは、index はパラメータの索引を表し、sqlType は SQL データ型（ここでは OracleTypes の 1 つ）を表し、sql_name そのデータ型に指定された名前（つまり名前付きタイプ）を表し、scale は sqlType が NUMERIC または DECIMAL データ型である場合の小数点以下の桁数を表します。

STRUCT、ARRAY、または REF を除く、任意の出力パラメータ・データ型では、CallableStatement.registerOutParameter() の 2 つのフォームを使用できます。

registerOutParameter() の OracleCallableStatement フォームは、出力パラメータが STRUCT、ARRAY、または REF 型である場合、または名前付きタイプの名前を提供する必要がある場合だけ、使用可能です。

次の例では、CallableStatement を使って procout 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 データ型を返します。registerOutParameter() メソッドで、OracleTypes.CHAR SQL 名を使用していることに注目してください。

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

次の例では、CallableStatement を使って procout をコールします。このプロシージャは、STRUCT データ型を返します。registerOutParameter() のフォームでは、SQL 型、OracleTypes.STRUCT の名前、および SQL 型名（名前付きタイプ）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 データ型内に取得されます。EMPLOYEE の値をデフォルトの STRUCT データ型内に取得する場合、文オブジェクト procout は OracleCallableStatement にキャストされ、getSTRUCT() が適用されます。

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

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

OracleTypes および setNull() メソッド OracleTypes クラス内の SQL 型はオブジェクトを識別します。そのオブジェクトは、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 および oracle.jdbc.driver.OraclePreparedStatement クラス内に存在します。

setNull() が PreparedStatement および OraclePreparedStatement クラスに対して使用可能なフォームを次に示します。

```
PreparedStatement.setNull(int index, int sqlType)

OraclePreparedStatement.setNull(int index, int sqlType, String sql_name)
```

これらのプロトタイプでは、index にはパラメータ索引が、sqlType には SQL データ型（ここでは OracleTypes のいずれか）が、sql_name にはデータ型に指定される名前（名

前付きタイプの名前）が、それぞれ当てはまります。無効な sqlType を入力すると、「パラメータ・タイプ矛盾」のエラーが発生します。

setNull() の PreparedStatement フォームを使って、STRUCT、ARRAY、REF を除く、任意のデータ型の値を NULL に設定できます。

setNull() の OraclePreparedStatement フォームを使用できるのは、データ型 STRUCT、ARRAY、または REF のオブジェクトの値を NULL に設定する場合だけです。

次の例では、PreparedStatement を使って数値 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。

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

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

この例では、準備済みの文を使って、EMPLOYEE 型の NULL STRUCT オブジェクトをデータベースに挿入します。STRUCT オブジェクトを NULL に設定するために、OraclePreparedStatement が必要なことに注意してください。このため、準備済みの文 pstmt を OraclePreparedStatement にキャストする必要があります。

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

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

データ・アクセスとデータの操作 Oracle 型対 Java 型

この項に含まれるサブセクションは、次のとおりです。

この項では、oracle.sql.* 形式でのデータ・アクセスについて、Java 形式と対比させて説明します。この章の冒頭で説明したように、oracle.sql.* 形式は Oracle JDBC エクステンションの主要要素であり、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。これらのクラスやメソッドの詳細は、Javadoc を参照してください。

データ変換での考慮事項

JDBC プログラムが SQL データを Java 変数内に取得する場合、SQL データは取得先の変数の Java データ型に変換されます。Java データ型は、java.lang や java.sql.Types パッケージのメンバーとして表現するかわりに、oracle.sql パッケージのメンバーとして表現できます。処理速度と成果の面で、oracle.sql.* クラスは SQL データを表現する最も効率的な手段を提供します。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だけです。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.Types.* または java.lang.* のメンバーとして表現します。同様に、使用する解析機能が Java 形式のデータを前提としている場合、データを oracle.sql.* 形式ではなく、いずれかの Java 形式で表現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、スカラー型の固定セット（byte、int、float など）およびオブジェクト型の固定セット（オブジェ

クトや配列など）の 2 つに分類されます。Java スカラー型で null を表現することはできません。かわりに、null をゼロ値（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 を表現可能な各スカラー型に対応したオブジェクト・ラッパー（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。

結果セットと文エクステンションの使用方法

JDBC Statement オブジェクトは、java.sql.ResultSet として型付けされた OracleResultSet オブジェクトを返します。標準 JDBC メソッドだけをオブジェクトに適用する場合は、ResultSet 型を維持してください。ただし、オブジェクト上で Oracle エクステンションを使用する場合、Oracle エクステンションを OracleResultSet 型にキャストする必要があります。オブジェクトへの変更は行われません。Java コンパイラがオブジェクトの識別に使用する型は、変更されません。

Java アプリケーションで標準 JDBC Statement オブジェクトを使って SELECT 文を実行すると、Oracle の JDBC ドライバは java.sql.ResultSet オブジェクトを返します。標準 JDBC ResultSet メソッドだけを必要とする場合、この標準 ResultSet オブジェクトを使用できます。ただし、Oracle エクステンションを使用するには、結果セットを OracleResultSet オブジェクトにキャストする必要があります。

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

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

Oracle エクステンションが JDBC に提供する拡張機能を必要とする場合、上記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、コール可能文を使ってストアド・プロシージャを実行する場合、JDBC ドライバは java.sql.CallableStatement として型付けされた OracleCallableStatement オブジェクトを返します。標準 JDBC メソッドだけをオブジェクトに適用する場合は、CallableStatement 型を維持してください。ただし、オブジェクトで Oracle エクステンションを使用する場合は、オブジェクトを OracleCallableStatement 型にキャストする必要があります。オブジェクトへの変更は行われません。Java コンパイラがオブジェクトの識別に使用する型は、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 メソッドだけ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 エクステンションを使用する場合、オブジェクトを OraclePreparedStatement 型にキャストする必要があります。オブジェクトへの変更は行われません。Java コンパイラがオブジェクトの識別に使用する型は、変更されません。

結果セットおよび statement クラスへの主要エクステンションには、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うと、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、次の項「oracle.sql.* 形式および Java 形式の get と set メソッドの比較」を参照してください。

oracle.sql.* 形式および Java 形式の get と set メソッドの比較

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（4-34 ページの「他の getXXX() メソッド」で説明したように）、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 メソッドを使用することもできます。

Standard 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 オブジェクト内に返します。返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、getObject() は列の SQL 型に対応するデフォルトの Java 型を返します。
- Oracle 固有のデータ型（4-100 ページの「追加の型エクステンション」で説明した ROWID など）の場合、getObject() は適切な oracle.sql.* クラス（oracle.sql.ROWID など）のオブジェクトを返します。
- Oracle オブジェクトの場合、getObject() は、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返します。（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、4-60 ページの「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 オブジェクトを返します。

getObject() 戻り型の詳細は、4-33 ページの表 4-3 の「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得する場合、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、Datum オブジェクト内に返されます。メソッドのプロトタイプは次の通りです。

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

データを Datum オブジェクト内に取得した場合は、標準 Java instanceOf() 演算子を使って、実際の oracle.sql.* 型を決定します。

getOracleObject() 戻り型の詳細は、4-33 ページの表 4-3 の「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得します。getOracleObject() は CHAR データを char_datum 変数に、BFILE ロケータを bfile_datum 変数に取得します。getOracleObject() は Datum オブジェクトを返すため、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 string_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）を日付と関連付けるプロシージャ myGetDate() へのコールを準備します。このプログラムは、文字列 SCOTT を解析して準備済みの文に渡し、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 と関連付けられた日付を取得します。getOracleObject() は Datum オブジェクトを返すため、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 4-3 は、前出の項 4-31 ページの「Standard getObject() メソッド」および 4-31 ページの「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()` は、常にデータを `java.lang.Object` に返します。
- `getOracleObject()` は、常にデータを `oracle.sql.Datum` に返します。

いずれかの特殊機能を使用するためには、返されたオブジェクトをキャストする必要があります。（4-35 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 4-3 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 またはネストした表）	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、8-2 ページの表 8-1 の「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）を正確に返します。

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 を返します。たとえば、getROWID() は、oracle.sql.ROWID を返します。

これらのエクステンションには、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.* 型ではなく、oracle.jdbc2.* 型のオブジェクトを返します。たとえば、プロトタイプを比較してみましょう。

```
oracle.jdbc2.Blob getBlob(int parameter_index)
```

これは、BLOB に対して oracle.jdbc2 型を返します。

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

一方、この文は BLOB に対して oracle.sql 型を返します。

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返すため、キャストによるトラブルを防ぐことができます。

表 4-4 に、各 getXXX() メソッドに対応した、基礎となる戻り型とシグネチャ型をまとめます。Oracle 固有のメソッドを使用するには、OracleResultSet または OracleCallableStatement にキャストする必要があります。

表 4-4 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	Oracle 固有
getArray()	oracle.sql.ARRAY	oracle.jdbc2.Array	Yes
getARRAY()	oracle.sql.ARRAY	oracle.sql.ARRAY	Yes
getBfile()	oracle.sql.BFILE	oracle.sql.BFILE	Yes
getBFILE()	oracle.sql.BFILE	oracle.sql.BFILE	Yes
getBigDecimal()	BigDecimal	BigDecimal	No
getBlob()	oracle.sql.BLOB	oracle.jdbc2.Blob	Yes
getBLOB	oracle.sql.BLOB	oracle.sql.BLOB	Yes

表 4-4 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	Oracle 固有
getBoolean()	boolean	boolean	No
getByte()	byte	byte	No
getBytes()	byte[]	byte[]	No
getCHAR()	oracle.sql.CHAR	oracle.sql.CHAR	Yes
getClob()	oracle.sql.CLOB	oracle.jdbc2.Clob	Yes
getCLOB()	oracle.sql.CLOB	oracle.sql.CLOB	Yes
getDate()	java.sql.Date	java.sql.Date	No
getDATE()	oracle.sql.DATE	oracle.sql.DATE	Yes
getDouble()	double	double	No
getFloat()	float	float	No
getInt()	int	int	No
getLong()	long	long	No
getNUMBER()	oracle.sql.NUMBER	oracle.sql.NUMBER	Yes
getRAW()	oracle.sql.RAW	oracle.sql.RAW	Yes
getRef()	oracle.sql.REF	oracle.jdbc2.Ref	Yes
getREF()	oracle.sql.REF	oracle.sql.REF	Yes
getROWID()	oracle.sql.ROWID	oracle.sql.ROWID	Yes
getShort()	short	short	No
getString()	String	String	No
getSTRUCT()	oracle.sql.STRUCT.	oracle.sql.STRUCT	Yes
getTime()	java.sql.Time	java.sql.Time	No
getTimestamp	java.sql.Timestamp	java.sql.Timestamp	No

get メソッドの戻り値のキャスト

4-31 ページの「Standard getObject() メソッド」で説明したように、Oracle によるインプリメンテーションでは、`getObject()` は常に `java.lang.Object` を返し、`getOracleObject()` は常に `oracle.sql.Datum` を返します。通常、返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() メソッドを使用することもできます。getXXX() の戻り型は、返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 ではなく、oracle.sql.CLOB を返します。

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（ここでは列 1）にフェッチしたものとします。CHAR データの正確性を保持したまま操作するため、結果セットを OracleResultSet ors にキャストし、getOracleObject() を使って CHAR データを返します。（結果セットをキャストしない場合、getObject() を使う必要があります。このメソッドは、文字データを Java String に返すため、SQL データの正確性がいくらか失われます。）結果セットをキャストすることにより、getOracleObject() を使用し、データを oracle.sql.* 形式で返すことが可能になります。

getOracleObject() メソッドは、oracle.sql.CHAR オブジェクトを oracle.sql.Datum 戻り変数内に返します。CHAR 戻り変数を使用し、後でそのクラスの特殊機能（文字表現に使用されるキャラクタ・セットを返す 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 戻り変数内に返しておき、後で CHAR getCharacterSet() メソッドを使用する際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();
```

ここでは、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得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 の準備済みの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。

setObject() メソッドを使用して、標準 Java 型を準備済みの文またはコール可能文にバインドできます。このメソッドは java.lang.Object を入力として取ります。setOracleObject() メソッドを使用して、oracle.sql.* 型をバインドできます。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。

setObject() メソッドは、いくつかの oracle.sql.* 型をサポートします。次の注意を参照してください。その他の oracle.sql.* 型の場合、setOracleObject() を使用する必要があります。

setOracleObject() を使用するには、準備済みの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。

注意 : setObject() 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インプリメントされています。oracle.sql.* クラスは、JDBC 2.0 準拠の Oracle エクステンションである BLOB、CLOB、BFILE、STRUCT、REF、および ARRAY に対応します。

例 : 準備済みの文での 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 型を返す）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 型のデータを返すため）。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);
...
```

準備済みの文 ps では、setOracleObject() メソッドは、charVal 変数で表現された oracle.sql.CHAR データを準備済みの文にバインドします。oracle.sql.* データをバインドするには、準備済みの文を 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。

さらに、JDBC 2.0 標準との互換性を維持するため、OraclePreparedStatement および 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、oracle.jdbc2.Blob 入力パラメータを取る setBlob() メソッド、および oracle.sql.BLOB 入力パラメータを取る setBLOB() メソッドがあります。

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*)

  動作は、標準 Java java.sql.PreparedStatement.setNull() と同様です。このメソッドは、java.sql.Types により定義された、パラメータ索引および SQL タイプ・コードを取ります。このメソッドを使ってオブジェクト（REF、ARRAY、STRUCT を除く）を NULL に設定します。

- void setNull(int *parameterIndex*, int *sqlType*, String *sql_type_name*)

  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 REF、ARRAY、または STRUCT の場合のみ使用します。

同様に、OracleCallableStatement.registerOutParameter() メソッドも、REF、ARRAY、または STRUCT で使用する、オーバーロードされたメソッドを持ちます。

void registerOutParameter(int *parameterIndex*, int *sqlType*, String *sql_type_name*)

Oracle 固有型をバインドする setXXX() メソッドを使用しても、標準 Java 型をバインドするメソッドに比べてパフォーマンス面のメリットはありません。

表 4-5 に、すべての setXXX() メソッドの入力型をまとめます。Oracle 固有のメソッドを使用するには、文を OraclePreparedStatement または OracleCallableStatement にキャストする必要があります。

表 4-5 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	Oracle 固有
setArray()	oracle.jdbc2.Array	Yes
setARRAY()	oracle.sql.ARRAY	Yes
setBfile()	oracle.sql.BFILE	Yes
setBFILE()	oracle.sql.BFILE	Yes
setBigDecimal()	BigDecimal	No
setBlob()	oracle.jdbc2.Blob	Yes
setBLOB()	oracle.sql.BLOB	Yes
setBoolean()	boolean	No
setByte()	byte	No
setBytes()	byte[]	No
setCHAR()	oracle.sql.CHAR	Yes

表 4-5 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	Oracle 固有
setClob()	oracle.jdbc2.Clob	Yes
setCLOB()	oracle.sql.CLOB	Yes
setDate()	java.sql.Date	No
setDATE()	oracle.sql.DATE	Yes
setDouble()	double	No
setFloat()	float	No
setInt()	int	No
setLong()	long	No
setNUMBER()	oracle.sql.NUMBER	Yes
setRAW()	oracle.sql.RAW	Yes
setRef()	oracle.jdbc2.Ref	Yes
setREF()	oracle.sql.REF	Yes
setROWID()	oracle.sql.ROWID	Yes
setShort()	short	No
setString()	String	No
setSTRUCT()	oracle.sql.STRUCT	Yes
setTime()	java.sql.Time	No
setTimestamp()	java.sql.Timestamp	No

すべての SQL および Java 型間の型互換性については、8-2 ページの表 8-1 の「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。

結果セット・メタデータ・エクステンションの使用方法

`oracle.jdbc.driver.OracleResultSetMetaData` クラスは、結果セット・メタデータ取得用の JDBC 2.0 API すべてをインプリメントしてはいませんが、Oracle 結果セットに関する情報を取得するための多くのメソッドを提供しています。

`getColumnTypeName()` メソッドは、列番号を受け取り、`REF`、`STRUCT`、または `ARRAY` 型列の SQL 型名を返します。一方、`getColumnType()` メソッドは、列番号を受け取り、SQL 型を返します。列が Oracle オブジェクトまたはコレクションを格納する場合、列は `OracleTypes.STRUCT` または `OracleTypes.ARRAY` を返します。

OracleResultSetMetadata により提供される主要メソッドの一覧は、4-25 ページの「クラス oracle.jdbc.driver.OracleResultSetMetaData」を参照してください。

次の例は、OracleResultSetMetadata クラスのメソッドをいくつか使って、EMP 表から列数、各列の数値型および SQL 型名を取得します。

```
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返します。

```
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
```

LOB の使用

この項に含まれるサブセクションは、次の通りです。

- BLOB および CLOB ロケータの取得
- BLOB および CLOB ロケータの引渡し
- BLOB および CLOB データの読取りと書込み
- BLOB または CLOB 列の作成と移入
- BLOB および CLOB データのアクセスと操作
- BFILE ロケータの取得
- BFILE ロケータの引渡し
- BFILE データの読取り
- BFILE 列の作成と移入
- BFILE データへのアクセスと操作

LOB は、内部と外部のいずれの場合もあります。内部 LOB は、その名前が示す通り、データベースの表領域内に格納され、領域を最適化し効率的なアクセスを可能にします。JDBC ドライバは、BLOB（非構造化バイナリ・データ）と CLOB（シングルバイト文字データ）の 2 種類の内部 LOB をサポートします。BLOB および CLOB データは、データベース表内に格納されたロケータを使ってアクセスおよび参照が行われ、BLOB または CLOB データを指します。

外部 LOB（BFILES）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・データです。これらのファイルは、リファレンス・セマンティックスを利用します。これらのファイルはまた、ハード・ディスク、CD-ROM、PhotoCD、および DVD 等の補助記憶デバイス上に配置することも可能です。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。

この項では、LOB を使って作業する際の、JDBC および oracle.sql.* クラスの使用法を説明します。LOB データを使用する場合、まず表からロケータを取得する必要があります。その後、LOB に対してデータの読取りや書込み、またはさまざまなデータ操作を実行できます。この項では、表内で LOB 列を作成および移入する方法を説明します。

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 oracle.jdbc2.Blob および Clob インタフェースをインプリメントします。一方、BFILE には、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だけを含み、データは含みません。ロケータへのアクセス後に、データにアクセスするための追加的な手順を実行する必要があります。これらの手順については、4-44 ページの「BLOB および CLOB データの読取りと書込み」および 4-52 ページの「BFILE データの読取り」を参照してください。

BLOB および CLOB ロケータの取得

BLOB または CLOB ロケータを含む標準 JDBC 結果セットまたはコール可能文オブジェクトがある場合、標準 ResultSet.getObject() メソッドを使って、ロケータにアクセスできます。このメソッドは、適切な oracle.sql.BLOB オブジェクトまたは oracle.sql.CLOB オブジェクトを返します（ただし、このメソッドは、BLOB または CLOB を oracle.jdbc2.Blob または oracle.jdbc2.Clob 型の変数内に返すことに留意してください）。

結果セットを OracleResultSet にキャストするか、またはコール可能文を OracleCallableStatement にキャストし、その後適切な getOracleObject()、getBLOB()、または getCLOB() メソッドを使用することにより、ロケータにアクセスできます。

OracleResultSet および OracleCallableStatement クラスでは、getBlob() は oracle.jdbc2.Blob を返し、getBLOB() は oracle.sql.BLOB を返します。同様に、getCLOB() は oracle.jdbc2.CLOB を返し、getClob() は oracle.sql.Clob を返します。

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、4-35 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。
- oracle.sql.BLOB および oracle.sql.CLOB クラスの特定機能の詳細は、Javadoc を参照してください。

例：結果セットからの BLOB および CLOB ロケータの取得 データベースに lob_table という表があり、その中に BLOB ロケータである blob_col 用の列、および CLOB ロケータである clob_col 用の列があるとします。この例では、Statement オブジェクトである 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で選択します。その後、LOB データを適切な Java クラス内に取得します。

```
// 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.
   oracle.jdbc2.Blob blob = (oracle.jdbc2.Blob)rs.getObject(1);
   oracle.jdbc2.Clob clob = (oracle.jdbc2.Clob)rs.getObject(2);
   [...process...]
}
```

出力は、oracle.jdbc2.Blob および Clob にキャストされます。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、上のコードを次のように書き直して、LOB ロケータを取得することもできます。

```
   // 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 ロケータの取得 LOB 取得用のコール可能文は、結果セットのメソッドと同一です。コール可能文の場合、出力パラメータを OracleTypes.BLOB または OracleTypes.CLOB として登録します。

たとえば、出力パラメータ CLOB 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。

```
OracleCallableStatement ocs =
   (OracleCallableStatement)conn.prepareCall("{? = call func()}")
ocs.registerOutParameter(1, OracleTypes.CLOB);
ocs.executeQuery()
oracle.sql.CLOB clob = ocs.getCLOB(1);
```

BLOB および CLOB ロケータの引渡し

LOB ロケータを準備済みの文またはコール可能文に渡すには（データベース内の LOB ロケータを更新するなどの理由で）、汎用の setObject() メソッドを使用するか、文を OraclePreparedStatement または OracleCallableStatement にキャストして setOracleObject()、setBLOB()、または setCLOB() のうち適切なメソッドを使用します。これらのメソッドは、パラメータ索引と BLOB オブジェクト、または CLOB オブジェクトを入力として受け取ります。

例 : BLOB ロケータの準備済みの文への引渡し 最初のパラメータが my_blob という名の BLOB である OraclePreparedStatement ops がある場合、BLOB を準備済みの文に次のように入力します。

```
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 CLOB を最初のパラメータとする OracleCallableStatement ocs がある場合、CLOB をコール可能文に次のように入力します。

```
OracleCallableStatement ocs =
   (OracleCallableStatement)conn.prepareCall("{? := call func()}")
ocs.setClob(1, my_clob)
ocs.execute();
```

BLOB および CLOB データの読取りと書込み

SQL SELECT 文は、LOB ロケータの問合せを実行します。ロケータがあれば、JDBC から LOB データの読取りおよび書込みができます。LOB データは、Java ストリームとして実体化されます。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、接続が有効な間、LOB データにはいつでもアクセスできます。

LOB データの読取り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取り、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 :

- データ・アクセス API のインプリメンテーションは、JDBC OCI およびサーバー・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 8.1.5 JDBC ドライバの LOB クラスで、同じ API を使用できます。
- JDBC Thin ドライバのみの場合、データ・アクセス API のインプリメンテーションは、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、『Oracle8*i* アプリケーション開発者ガイド ラージ・オブジェクト』および『Oracle8i Application Developer's Reference - Packages』を参照してください。

LOB データの読取り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 から読み取るには、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得します。これにより、java.io.InputStream オブジェクトが返されます。

すべての 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 つを使って 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 を書き込む場合、oracle.sql.BLOB オブジェクトの getBinaryOutputStream() メソッドを使って、BLOB を出力ストリームとして取得します。これにより、java.io.OutputStream オブジェクトが BLOB に書き戻されます。

  すべての 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 つを使って、LOB データを更新し、完了時に close() メソッドを使用します。

- CLOB から読み取るには、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得します。getAsciiStream() メソッドは、java.io.InputStream オブジェクトの ASCII 入力ストリームを返します。getCharacterStream() メソッドは、java.io.Reader オブジェクトの Unicode 入力ストリームを返します。

  すべての 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 つを使って、LOB データを読み込み、完了時に close() を使用します。

  oracle.sql.CLOB オブジェクトの getSubString() メソッドを使って、CLOB のサブセットを java.lang.String 型の文字列として取得できます。

- CLOB への書込みを実行する場合、oracle.sql.CLOB オブジェクトの getAsciiOutputStream() または getCharacterOutputStream() メソッドを使って、CLOB を出力ストリームとして取得してから、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 出力ストリームを返します。getCharacterOutputStream() メソッドは、java.io.Writer オブジェクトの Unicode 出力ストリームを返します。

  すべての OutputStream や 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 つを使って、LOB データを更新し、完了時に 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/COMMIT を実行する必要はありません。
- CLOB に対して書込みまたは読取りを実行する際、JDBC ドライバがすべてのキャラクタ・セット変換を処理します。

例 : BLOB データの読取り oracle.sql.BLOB クラスの getBinaryStream() メソッドを使用して、BLOB データの読取りを行います。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、その後バイト・ストリームを読み取ってバイト配列に格納します。(読み取ったバイト数も返します。)

```
// 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 データの読取り 次の例では、getCharacterStream() メソッドを使って、CLOB データを Unicode 文字ストリーム内に読み込みます。次に、文字ストリームを文字配列に書き込みます。(読み取った文字数も返します。)

```
// 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() メソッドを使って、CLOB データを ASCII 文字ストリーム内に読み込みます。その後、ASCII ストリームをバイト配列に読み込みます。(読み取った文字数も返します。)

```
// 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() メソッドを使って、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() メソッドを使って、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() メソッドを使って、データを CLOB に書き込みます。getCharacterOutputStream() メソッドは Unicode 出力ストリームを返し、getAsciiOutputStream() メソッドは ASCII 出力ストリームを返します。

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() メソッドを使って文字データの配列を CLOB に書き込みます。getCharacterOutputStream() メソッドは、oracle.jdbc2.Clob ではなく、oracle.sql.CLOB の java.io.Writer オブジェクトを返します。

```
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() メソッドを使って、ASCII データの配列を CLOB に書き込みます。getAsciiOutputStream() は ASCII 出力ストリームを返すため、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

表内で BLOB または CLOB 列を作成する場合、SQL 文を使用します。

注意 : 表内で BLOB または CLOB 列を作成する場合、SQL 文を使用する必要があります。Java の new（new BLOB や new CLOB など）は、動作しません。

表内で BLOB または CLOB 列を作成する場合は、SQL CREATE TABLE 文を使用します。その後、LOB を移入します。これには、表内での LOB エントリの作成、LOB ロケータの取得、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 文を作成します。次の例では、新しい表に BLOB 列を作成します。この例では、Connection オブジェクトである conn および Statement オブジェクトである 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 は one や two などの行番号を、blob 列は BLOB データのロケータを格納します。

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 オブジェクトである conn および Statement オブジェクトである 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 文を使って BLOB エントリを表の中に作成します。empty_blob 構文を使って、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 ロケータを取得します。

   ```
   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得します。

```
File binaryFile = new File("john.gif");
System.out.println("john.gif length = " + binaryFile.length());
    FileInputStream instream = new FileInputStream(binaryFile);
    OutputStream outstream = blob.getBinaryOutputStream();
```

4. getChunkSize() をコールして、BLOB に書き込むための理想的なチャンク・サイズを決定し、次に buffer バイト配列を作成します。

```
    int chunk = blob.getChunkSize();
    byte[] buffer = new byte[chunk];
   int length = -1;
```

5. read() メソッドを使って GIF ファイルをバイト配列 buffer 内に読み込み、次に write() メソッドを使ってそれを 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「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、4-42 ページの「BLOB および CLOB ロケータの取得」を参照してください。

ロケータの選択後に、BLOB または CLOB データを取得できます。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.* 形式で取得できるようにします。BLOB または CLOB データの取得後、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 文を使って、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.getlength();

// 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(0, length);
printBytes(bytes, length);
```

BFILE ロケータの取得

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れば、標準 ResultSet.getObject() メソッドを使ってロケータへのアクセスが可能です。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 を返します。BFILE 用の oracle.jdbc2 クラスはありません。
- getObject() または getOracleObject() を使用する場合、必要に応じ出力をキャストすることに留意してください。詳細は、4-35 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 ロケータの取得 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 bfile_col 用の列が 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、標準結果セットに格納します。結果セットを OracleResultSet にキャストする場合、getBFILE() を使って次のように BFILE データを取得できます。

```
// 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() を使って BFILE ロケータを返すこともできます。この場合、getObject() は java.lang.Object を返すため、結果を BFILE にキャストしてください。次に例を示します。

```
       oracle.sql.BFILE my_bfile = (BFILE)rs.getObject(1);
```

例 : コール可能文からの BFILE ロケータの取得 BFILE 出力パラメータを持つ関数 func をコールする、OracleCallableStatement ocs があるものとします。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、そして BFILE ロケータを取得します。

```
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 ロケータを準備済みの文またはコール可能文に渡すには（BFILE ロケータを更新するなどの理由で）、汎用の setObject() メソッドを使用するか、文を OraclePreparedStatement または OracleCallableStatement にキャストして、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 ロケータの準備済みの文への引渡し BFILE ロケータを表に挿入する方法を説明します。最初のパラメータが文字列（行番号を示す）で、2 番目のパラメータが BFILE である OraclePreparedStatement ops、および有効な oracle.sql.BFILE オブジェクト（bfile）がすでに存在するものとします。BFILE を準備済みの文に次のように入力します。

```
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、準備済みの文に渡す方法と同じです。この場合、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 の長さを数値で返します。

```
OracleCallableStatement cstmt =
      (OracleCallableStatement)
        conn.prepareCall ("begin ? := myGetFileLength (?); end;");
    try
    {
      cstmt.registerOutParameter (1, Types.NUMERIC);
      cstmt.setBFILE (2, bfile);
      cstmt.execute ();
      return cstmt.getLong (1);
    }
    finally
    {
      cstmt.close ();
    }
  }
```

BFILE データの読取り

BFILE データを読み取るには、まず BFILE ロケータを取得する必要があります。ロケータは、コール可能文からも結果セットからも取得可能です。詳細は、4-50 ページの「BFILE ロケータの取得」を参照してください。

ロケータを取得すれば、BFILE をオープンしなくても、BFILE に対して数多くのメソッドを実行できます。たとえば、oracle.sql.BFILE メソッド fileExists() や isFileOpen() を使って、BFILE の存在やオープンしているかどうかを確認できます。ただし、データの読取りおよび操作を行うには、BFILE. をオープンする必要があります。BFILE データは、Java ストリームとして実体化されます。JDBC から BFILE に対し、次のような操作を行います。

- BFILE から読み取るには、oracle.sql.BFILE オブジェクトの getBinaryStream() メソッドを使って、ファイル全体を入力ストリームとして取得します。これにより、java.io.InputStream クラスが返されます。

  すべての 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 つを使ってファイル・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。

例 : BFILE データの読取り 次の例では、oracle.sql.BFILE オブジェクトの getBinaryStream() メソッドを使って、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

SQL 文を使って表内に BFILE 列を作成し、BFILE の格納場所を指定します。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 の場所を指定します。BFILE の場所を指定するには、SQL CREATE DIRECTORY...AS 文を使って 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 文を使って 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 列は BFILE データのロケータを格納します。

BFILE 列の移入

SQL INSERT INTO...VALUES 文を使って 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 列は、ロケータと共に BFILE データに移入されます。BFILE 列を移入するには、bfilename キーワードを使っ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、one 行の bfile 列にロードされます。BFILE jdbcTest.data のロケータは、two 行の 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 を指定します。

```
cmd ="INSERT INTO my_bfile_table VALUES ('three', null)";
stmt.execute(cmd);
```

ここでは、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。プログラムの後の部分で、準備済みの文を使って BFILE ロケータを表に挿入します。

まず、有効な BFILE ロケータを 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(2);
```

その後、準備済みの文を作成します。この例では setBFILE() メソッドを使って BFILE を識別しているため、次のように準備済みの文を OraclePreparedStatement にキャストする必要があります。

```
OraclePreparedStatement ops = (OraclePreparedStatement)conn.prepareStatement(INSERT
? INTO my_bfile_table)
                WHERE (x = 'three');
ops.setBFILE(2, bfile);
ops.execute();
```

これで、two 行と three 行には、同じ BFILE が格納されました。

表の中で利用可能な BFILE ロケータの準備完了後に、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「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 ロケータを配置できたら、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。

次の例では、BFILE のロケータを、表の行 two から結果セット内に取得します。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。読取り、長さの取得、表示などの 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。BFILE の例の詳細は、7-10 ページの「BFILE のサンプル」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] );
      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();
```

Oracle オブジェクト型の使用

この項のトピックは次のとおりです。

- Oracle オブジェクト用のデフォルト Java クラスの使用方法
- Oracle オブジェクト用のカスタム Java クラスの作成
- JDBC での JPublisher の使用方法

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）、住所（CHAR 型）、電話番号（CHAR 型）、および従業員番号（NUMBER 型）などの属性をもつ Person 型を定義することができます。

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。カスタム Java 型定義クラスを作成すると、SQL 型を Java クラスにマップする方法をカスタマイズすることができます。Oracle では、かなり柔軟性のあるマッピング方法が提供されます。このマニュアルでは、Oracle オブジェクトにマップするクラスとして作成された Java クラスをカスタム Java クラスと呼びます。

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実体化します。JDBC を使って 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 つのステップを実行します。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 として実体化します。詳細は 4-57 ページの「Oracle オブジェクト用のデフォルト Java 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 Java クラ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、SQLData インタフェースまたは CustomDatum インタフェースのいずれかに従ってクラスを定義します。詳細は、4-59 ページの「Oracle オブジェクト用のカスタム Java クラスの作成」を参照してください。

Oracle オブジェクト用のデフォルト Java クラスの使用方法

型マップを提供して Oracle オブジェクトに対する Java クラスを明示的に指定しない場合、Oracle JDBC によりオブジェクトを Struct として実体化することができます。

通常は、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操作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することができます。4-9 ページの「クラス oracle.sql.STRUCT」で説明されているように、STRUCT ではデータが SQL 形式で保持されるため、データを完全に保存できます。ユーザーにわかりやすい形で情報を保存する必要がない場合、Struct オブジェクトを使えば、データをより効率的に、より正確に保存できます。

コードが JDBC 2.0 に完全に準拠する必要がある場合、oracle.jdbc2.Struct インタフェースの次の機能を使ってください。

- getAttributes(map): 値配列から値を java.lang.Object オブジェクトとして取得。型マップ（定義されている場合）のエントリによってデータの実体化を行う Java クラスを決定します。
- getAttributes(): 値配列の値を java.lang.Object オブジェクトとして取得。
- getSQLTypeName(): この Struct が表す Oracle オブジェクトの完全修飾型名（*schema.sql_type_name*）を表す JavaString を返します。

JDBC 2.0 と互換性を保つ必要がない場合、Oracle 定義メソッドによる拡張機能を利用したいときは、出力を oracle.sql.STRUCT にキャストしてください。

oracle.sql.STRUCT クラスでは、oracle.jdbc2.Struct インタフェースがインプリメントされ、JDBC 2.0 規格以上の拡張機能が提供されます。上記のメソッドと 4-9 ページの「クラス oracle.sql.STRUCT」に記載されている oracle.sql.STRUCT のメソッドを比較してください。

STRUCT オブジェクトの使用方法

getObject() などの標準 JDBC 機能を使用すると、データベースから Oracle オブジェクトを oracle.jdbc2.Struct のインスタンスとして取得できます。getObject() により java.lang.Object が返されるため、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します。

```
oracle.jdbc2.Struct  myStruct = (oracle.jdbc2.Struct)rs.getObject(1);
```

前の項で説明したとおり、oracle.jdbc2.Struct クラスは oracle.sql.STRUCT によってインプリメントされます。Struct オブジェクトを STRUCT にキャストすると、Oracle が提供する拡張機能を使用できます。たとえば、getOracleAttributes() メソッドを使って Struct の属性を返す場合、次のように myStruct を oracle.sql.STRUCT にキャストします。

```
oracle.sql.STRUCT STRUCTattribute=s
((oracle.sql.STRUCT)myStruct).getOracleAttributes()
```

getOracleAttributes() メソッドにより myStruct の属性が oracle.sql.* 形式で返されます。

オブジェクトを取り出して直接 oracle.sql.STRUCT に入れることもできます。たとえば、getObject() を使って表 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 型が返されるため、結果が oracle.sql.STRUCT にキャストされます。次の例は Statement オブジェクトの stmt がすでに生成されていることを前提にしています。

```
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 test_Struct");
oracle.sql.STRUCT struct_obj=(oracle.sql.STRUCT) rs.getObject(1);
```

oracle.sql.STRUCT オブジェクトを使ってデータに対するアクセス、操作、または更新を行う場合、setOracleObject() メソッドを使ってオブジェクトを準備済みの文またはコール可能文にバインドできます。この場合、準備済みの文またはコール可能文を、OraclePreparedStatement オブジェクトまたは OracleCallableStatement オブジェクトにキャストする必要があります。

```
PreparedStatement ps= conn.prepareStatement("text_of_prepared_statement");
STRUCT mySTRUCT = new STRUCT (...)
((OraclePreparedStatement)ps).setOracleObject(1, mySTRUCT);
```

同様にデータベースからデータを取得するには、OracleCallableStatement および OracleResultSet クラスに getSTRUCT() メソッドを含める必要があります。このメソッドは Oracle オブジェクトを oracle.sql.STRUCT として返します。

```
ResultSet rset = stmt.executeQuery (...);
oracle.sql.STRUCT mySTRUCT = ((OracleResultSet)rs).getSTRUCT();
```

Oracle オブジェクト用のカスタム Java クラスの作成

Oracle オブジェクト用にカスタム Java クラスを定義する場合、型マップ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 Java クラスを生成します。

Oracle オブジェクトとその属性データからカスタム Java クラスを生成し、移入する手段も提供する必要があります。ドライバから Java カスタム・クラスの読取りおよび移入を実行できなければなりません。さらにカスタム Java クラスは、必要に応じて Oracle オブジェクトの属性に対応する get および set メソッドを提供することもできます。カスタム・クラスの作成と移入、およびドライバの読取り / 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 が提供する SQLData インタフェース
- Oracle が提供する CustomDatum インタフェース

作成するカスタム Java クラスでは、これらのインタフェースのいずれかをインプリメントする必要があります。

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。型マップを使って EMPLOYEE オブジェクトが JEmployee というカスタム Java クラスにマップされるように指定します。JEmployee クラスによって SQLData または CustomDatum インタフェースのいずれかをインプリメントできます。

カスタム Java クラスを作成する最も便利な方法は、JPublisher ユーティリティを活用することです。ただし、JPublisher は CustomDatum インタフェースしかサポートしていません。自分でカスタム Java クラスを作成することもできます。実際、SQLData インタフェースをインプリメントする場合には自分で作成する必要があります。

次の項では、CustomDatum および SQLData の利点について説明します。

CustomDatum と SQLData の利点 2 つのインタフェースのうちどちらをインプリメントするか決定する場合、次の点を考慮してください。

CustomDatum の利点

- Oracle エクステンションに対応しています。
- oracle.sql.STRUCT から CustomDatum を構築できます。ネイティブ Java 型への変換が最小限で済むため、より効率的です。
- toDatum() メソッドを使って、CustomDatum オブジェクトから対応する Datum オブジェクト（oracle.sql 形式）を取得できます。
- 型マップが必要ありません。

- パフォーマンスが向上します。CustomDatum は、Datum 型を直接使って動作します。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われる内部形式です。
- JPublisher によってサポートされています。Jpublisher により作成された Custom Java クラスでは、CustomDatum のインプリメンテーションが使われます。リリース 8.1.5 では、SQLData は Jpublisher ではサポートされません。
- Oracle SQLJ によってサポートされています。リリース 8.1.5 では、Oracle SQLJ インプリメンテーションによって SQLData はサポートされません。

SQLData の利点

- JDBC 規格であるため、コードの移植が容易です。

SQLData インタフェースでは、SQL オブジェクトから Java オブジェクトへのデータの移入以外はできませんが、CustomDatum インタフェースでは、はるかに強力です。CustomDatum を使うと、Java オブジェクトにデータを移入できるほか、不要なオブジェクトの SQL 型からオブジェクトを実体化できます。このため、Oracle データベースに格納されているデータ型から CustomDatum オブジェクトを作成できます。この方法は、オブジェクトを直列化できる RAW データの場合に特に有効です。

型マップ

SQLData インタフェースを使って Java カスタム・クラスを作成する場合は、Java カスタム・クラスを指定する型マップを作成する必要があります。この Java カスタム・クラスは、データベースの Oracle オブジェクトに対応しています。SQLData を使ってカスタム Java クラスを作成する方法については、4-59 ページの「Oracle オブジェクト用のカスタム Java クラスの作成」を参照してください。

型マップにオブジェクトおよびそのマッピングを含めなかった場合は、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。このクラスの詳細は、4-9 ページの「クラス oracle.sql.STRUCT」を参照してください。

型マップを使って、Java クラスを Oracle オブジェクトの SQL 型名に関連付けます。このマップは、1 対 1 マッピングで、ハッシュ表にキー - 値の組合せで格納されています。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、SQL オブジェクト型のデータの実体化に使われる 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 型名が取得されます。SQL と Java 間の実際の変換は、ドライバにより処理されます。

Oracle オブジェクトに対応している Java クラスの属性では、Java ネイティブ型または Oracle ネイティブ型（oracle.sql.* クラスのインスタンス）を使って属性を格納できます。

型マップ・クラスの作成

Java アプリケーションのプログラマは、java.util.Dictionary をインプリメントする型マップ・クラスを設定する必要があります。たとえば、java.util.Hashtable によって、Dictionary がインプリメントされます。

型マップ・クラスには、put() メソッドをインプリメントする必要があります。このメソッドは、Java クラスを Oracle オブジェクト型に関連付ける、各マッピング・エントリを入力するときに使われます。put() メソッドは、キーワード - 値の組合せを受け取るためにインプリメントする必要があります。このキーは Oracle SQL 型名で、その値は Java クラス・オブジェクトです。

型マップ・オブジェクトの作成とマッピングの定義

各接続オブジェクトには、関連付けられている型マップ・オブジェクトの属性が設定されています。型マップ機能を使うときは、接続を OracleConnection オブジェクトにキャストします。

型マップを作成するには、次の項で説明する方法を使用します。

- 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 既存の型マップにエントリを追加するには、次のステップに従ってください。

1. OracleConnection オブジェクトの getTypeMap() メソッドを使って、接続の Map オブジェクトを取得します。getTypeMap() メソッドにより、java.util.Dictionary オブジェクトが返されます。たとえば、

   ```
   java.util.Dictionary myMap = oraconn.getTypeMap();
   ```

   この例では、OracleConnection オブジェクトである oraconn の getMapType() メソッドによって myMap Dictionary オブジェクトが返されます。

   > **注意**：OracleConnection オブジェクトの型マップが初期化されていない場合、getTypeMap() への最初のコールでは null が返されます。

2. Dictionary オブジェクトの put() メソッドを使ってマップに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・オブジェクト名を指定します。

   ```
   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。

class.forName() メソッドを使って、クラス・オブジェクトを取得します。put() メソッドのかわりに、次の文を実行することもできます。

```
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 PERSONJava 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 PERSONSQL 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。

3. エントリをマップに追加したら、OracleConnection オブジェクトの setTypeMap() メソッドを使って、接続の既存の型マップを上書きします。たとえば、

```
oraconn.setTypeMap(myMap);
```

この例では、setTypeMap() によって、oraconn 接続の元のマップが myMap で上書きされます。

新しい型マップの作成 新しい型マップを作成するには、次のステップに従ってください。

1. 空のマップ・オブジェクトを作成します。空のマップ・オブジェクトは、java.util.Dictionary クラスをインプリメントする任意のオブジェクトです。たとえば、java.util.Hashtable クラスは、Dictionary クラスをインプリメントできます。
2. Map オブジェクトの put() メソッドを使って、マップにエントリを追加します。put() メソッドの詳細は、前の項のステップ 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、次の文を使って、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ら、OracleConnection オブジェクトの setTypeMap() メソッドを使って、接続の既存の型マップを上書きします。たとえば、

```
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 :

- getXXX() および setXXX() メソッドに型マップ・オブジェクトを明示的に指定すると、接続のカスタムまたはデフォルト・マップを上書きできます。
- Oracle オブジェクト型に対する Java クラスのマッピングを型マップで指定しなかった場合は、デフォルトで oracle.sql.STRUCT クラスのインスタンス内で、Java によって実体化されたオブジェクト・タイプのデータにマップされます。このクラスの詳細は、4-9 ページの「クラス oracle.sql.STRUCT」を参照してください。
- 型マップを使って、データベースにカスタム・オブジェクトを挿入しないでください。

STRUCTS と型マップ 特定の SQL オブジェクト型を型マップに指定しなかった場合は、そのオブジェクトはドライバによって oracle.jdbc2.Struct クラスのインスタンスとして実体化されます。SQL オブジェクトに埋込みオブジェクトが格納され、そのオブジェクトが型マップに指定されていない場合は、埋込みオブジェクトは、ドライバによって oracle.sql.Struct のインスタンスとして実体化されます。その埋込みオブジェクトが型マップに指定されている場合は、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返されます。

SQLData インタフェース

Java アプリケーションで利用可能な Oracle オブジェクトとその属性データを作成するには、SQLData インタフェースをインプリメントするオブジェクトに対して、カスタム Java クラスを作成します。このインタフェースを使う場合は、データベース内の Oracle オブジェクト、およびそのオブジェクトに対して作成するカスタム Java クラスに対応する名前を指定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。標準的な JDBC の oracle.jdbc2 パッケージには、SQLData インタフェース、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 をインプリメントするカスタム Java クラスを作成する場合は、SQLData インタフェースの定義に従って、readSQL() メソッドおよび writeSQL() メソッドを指定する必要があります。

JDBC ドライバでは、readSQL() メソッドをコールして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 Java クラスのインスタンスを移入します。通常、ドライバでは、OracleResultSet.getObject() コールの一部としてこのメソッドが使われます。

同様に、JDBC ドライバでは、writeSQL() メソッドをコールし、カスタム Java クラスのインスタンス・データの順序値を読み込み、データベースに書込みできるストリームに書き込

みます。通常、ドライバでは OraclePreparedStatement setObject() コールの一部としてこのメソッドが使われます。

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 インタフェースをインプリメントするクラスが含まれます。SQLOutput または SQLInput オブジェクトをインプリメントする必要はありません。JDBC ドライバによってインプリメントされます。

SQLInput インプリメンテーションは入力ストリーム・クラスで、readSQL() に渡す必要があるインスタンスです。SQLInput には、readObject()、readInt()、readLong()、readFloat()、readBlob() など、Oracle オブジェクトの属性と Java 型の変換に使われるすべての 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返されます。たとえば、readInt() では、整数が返されます。

SQLOutput インプリメンテーションは出力ストリーム・クラスで、そのインスタンスは writeSQL() 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() メソッドのインプリメント SQLData をインプリメントするカスタム Java クラスを作成するときは、readSQL() および writeSQL() メソッドもインプリメントする必要があります。

次のように、readSQL() をインプリメントします。

```
public void readSQL(SQLInput stream, String sql_type_name) throws SQLException
```

- readSQL() では、SQLInput ストリームおよびデータの SQL 型名を示す文字列（EMPLOYEE などの Oracle オブジェクト型名）を入力として指定する必要があります。

  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っ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。

- Oracle オブジェクトの属性にマップする Java データ型ごとに、readSQL() では、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 変数として従業員名、および NUMBER 変数として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() では、readXXX() メソッドによって読み込まれ、カスタム Java クラスの適切な要素またはフィールドに変換されたデータを割り当てます。

次のように、writeSQL() をインプリメントする必要があります。

```
public void writeSQL(SQLOutput stream) throws SQLException
```

- writeSQL() では、SQLOutput ストリームを入力として指定する必要があります。

  Java アプリケーションから setObject() をコールすると、JDBC ドライバでは SQLOutput 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ってそのオブジェクトを移入します。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。

- Oracle オブジェクトの属性にマップする Java データ型ごとに、writeSQL() では、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 変数として従業員名、および NUMBER 変数として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() では、writeXXX() メソッドにより変換されたデータを読み込み、SQLOutput ストリームに書き込みます。この結果、準備済みの文を実行したときにデータベースに変換データが書き込まれます。

注意：SQLData 、SQLInput 、および SQLOutput インタフェースの詳細は、Javadoc を参照してください。

Oracle オブジェクトの特定の SQL 定義に対して、SQLData インタフェースをインプリメントする例については、7-19 ページの「Oracle オブジェクト用にカスタマイズした Java クラス」を参照してください。

SQLData クラスを使った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 をインプリメントしている場合に、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。

SQLData インタフェースを使った Oracle オブジェクトからのデータの読込み　この項では、カスタム Java クラスに対して SQLData インプリメンテーションを選択したときに、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ステップについて説明します。

このステップでは、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 を定義していることを前提にしています。

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");
rs.next();
```

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() メソッドを使って、カスタム Java クラスのインスタンスを結果セットの行のデータによって移入します。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 オブジェクトが返されます。

```
Employee emp = (Employee)rs.getObject(1);
```

型マップにオブジェクトのエントリが定義されていない場合、getObject() によって oracle.sql.STRUCT オブジェクトが返されます。この場合、出力を oracle.sql.STRUCT にキャストする必要があります。

```
Struct empstruct = (oracle.sql.STRUCT)rs.getObject(1);
...
```

すでに説明したように、getObject() コールによって readSQL() および readXXX() コールが行われます。

注意: 型マップを使わないようにするには、getSTRUCT() メソッドを使います。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 オブジェクトが返されます。

---

3. カスタム Java クラスに get メソッドが定義されている場合は、そのメソッドを使っ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 を返す getEmpName() メソッドおよび整数値（int）を返す getEmpNum() メソッドを定義します。次に、そのメソッドを Java アプリケーションでコールします。

```
String empname = emp.getName();
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 Statement オブジェクトを使って、データをフェッチします。

SQLData オブジェクトを OUT パラメータとしてコール可能文に渡す PL/SQL 関数 getEmployee(?) をコールする OracleCallableStatement ocs を定義していると仮定します。プログラムでは、従業員番号（empnumber）を関数に渡し、その関数から対応する Employee オブジェクトが返されます。

1. getEmployee(?) 関数をコールするために、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、OUT パラメータとして登録します。Employee オブジェクトの SQL 型は、OracleTypes.STRUCT です。次に、文を実行します。

```
ocs.setInt(2,empnumber);
ocs.registerOutParameter(1, OracleTypes.STRUCT, "EMP_OBJECT");
ocs.execute();
```

3. getObject() メソッドを使って、employee オブジェクトを取得します。このオブジェクトは STRUCT として返されるため、getObject() の出力を Employee オブジェクトにキャストします。

```
Employee emp = (Employee) ocs.getObject(1);
```

SQLData オブジェクトを IN パラメータとしてコール可能文に渡す 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 関数 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(?) 関数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() を使って、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 インタフェースを使ってデータを Oracle オブジェクトに書き込む この項では、カスタム Java クラスに対して SQLData インプリメンテーションを選択したときに、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ステップを説明します。

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 Java クラスに set メソッドを定義している場合、そのメソッドを使ってアプリケーションの Java 変数のデータを読み込み、Java データ型オブジェクト属性に書き込みます。

   ```
   emp.setEmpName(empname);
   emp.setEmpNum(empnumber);
   ```

   この文では、4-65 ページの「SQLData インタフェースを使った Oracle オブジェクトからのデータの読込み」で定義されている emp オブジェクト、empname 変数、および empnumber 変数が使われます。

2. Java データ型オブジェクトのデータを使って、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. 準備済みの文の setObject() メソッドを使って、Java データ型オブジェクトを準備済みの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();
   ```

注意 : Java データ型オブジェクトは、IN バインド変数または OUT バインド変数として使えます。

CustomDatum 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には、oracle.sql.CustomDatum および oracle.sql.CustomDatumFactory インタフェースをインプリメントするオブジェクトに対して、カスタム Java クラスを作成します。CustomDatum および CustomDatumFactory インタフェースは、Oracle から提供されていますが、JDBC 規格には準拠していません。

注意 : JPublisher ユーティリティは、CustomDatum および CustomDatumFactory インタフェースをインプリメントするクラスの生成をサポートしています。

CustomDatum インタフェースには、次の利点もあります。

- JDBC の Oracle エクステンションを識別します。CustomDatum では、oracle.sql.Datum 型が直接使われます。
- 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。
- パフォーマンスが改善します。CustomDatum は、Datum 型直接使って動作します。Datum は、Oracle オブジェクトを保持するためにドライバにより使われる内部形式です。

CustomDatum および CustomDatumFactory インタフェースでは次の処理が実行されます。

- CustomDatum 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。
- CustomDatumFactory では、カスタム Java クラスのコンストラクタと等価の create() メソッドを指定します。このメソッドでは、CustomDatum インスタンスの作成および返却を行います。JDBC ドライバでは、create() メソッドを使って、カスタム Java 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返します。oracle.sql.Datum オブジェクト、および OracleTypes クラスに指定された対応する SQL 型コードを示す整数を、入力として指定します。

CustomDatum および CustomDatumFactory は、次のように定義されています。

```
public interface CustomDatum
{
    Datum toDatum (OracleConnection conn) throws SQLException;
}

public interface CustomDatumFactory
{
    CustomDatum 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 型コードを表します。

注意 : Datum オブジェクト型と SQL 型コードが矛盾する場合の処理方法は、開発者が決定します。

オブジェクト・データを CustomDatum のインスタンスとして取出し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います。

オブジェクト・データを取り出すには

- Oracle エクステンションの OracleResultSet.getCustomDatum () メソッドを使用します。

  ```
  OracleResultSet.getCustomDatum (int col_index, CustomDatumFactory 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 CustomDatumFactory 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 Java クラスの getFactory() メソッドをインプリメントすると、CustomDatumFactory インスタンスが作成され、getCustomDatum() の入力になります。CustomDatum をインプリメントする Java クラスを使うときは、型マップは必要ありません。

または

- 標準的な ResultSet.getObject(index, map) メソッドを使って、CustomDatum 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われるファクトリ・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

- Oracle エクステンションの OraclePreparedStatement.setCustomDatum () メソッドを使用します。

  ```
  OraclePreparedStatement.setCustomDatum (int bind_index, CustomDatum 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

- 標準 JDBC PreparedStatement.setObject() メソッドを使います。このメソッドはさまざまな方法で使うことができます。また、型マップを使わずに CustomDatum インスタンスを挿入することができます。

次の項以降では、getCustomDatum() および setCustomDatum() メソッドについて説明します。

Oracle オブジェクトのサンプル・オブジェクト EMPLOYEE を引き続き使うには、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。

```
CustomDatum datum = ors.getCustomDatum(1, Employee.getFactory());
```

この例では、ors は Oracle 結果セット、getCustomDatum() は CustomDatum オブジェクトの取出しに使われる OracleResultSet クラス、および EMPLOYEE は結果セットの列 1 です。Employee.getFactory() コールによって、CustomDatumFactory が JDBC ドライバに返されます。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返します。

注意：

- CustomDatum および CustomDatumFactory 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）からインプリメントできます。
- カスタム Java クラスに oracle.sql.*（CustomDatum、CustomDatumFactory、Datum は必須です）、oracle.jdbc.driver.*（OracleConnection、OracleTypes は必須です）、および java.sql.SQLException をインポートする必要があります。
- CustomDatum および CustomDatumFactory クラスの詳細は、Javadoc を参照してください。

CustomDatum と SQLData: 直列可能オブジェクトとの比較

CustomDatum 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、SQL オブジェクト型（Oracle8 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 インタフェースをインプリメントすると、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・データとカスタマイズされた Java クラスのフィールド間の移入を、JDBC ドライバを使って行います。

CustomDatum インタフェースの場合は、SQL オブジェクト型から Java 型へのカスタマイズ以上のことが行えます。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。

たとえば、CustomDatum を使うと、データベースの特定の SQL Oracle8 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。CustomDatumFactory 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インプリメントする必要があります。CustomDatum の toDatum() メソッドは、Java オブジェクトから oracle.sql.RAW への変換をインプリメントする必要があります。このインプリメントのために Java の直列化を使うことができます。

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、CustomDatumFactory 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だけでデータベースに格納できます。JDBC ドライバでは、CustomDatum.toDatum() メソッドを透過的にコールし、Java オブジェクトを oracle.sql.RAW オブジェクトに変換します。このオブジェクトは、データベースの型 RAW の列に格納されます。

CustomDatum インタフェースをサポートすると、これらの変換が oracle.sql.* 形式を使って動作するように設計されており、oracle.sql.* 形式は JDBC ドライバで使わ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、CustomDatum をインプリメントする Java クラスを使うときに、SQLData インタフェースで必要な型マップが必要ありません。CustomDatum をインプリメントするクラスに型マップが不要な理由については、4-68 ページの「CustomDatum インタフェース」を参照してください。

CustomDatum インタフェースを使った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 CustomDatum をインプリメントする場合に、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。

CustomDatum インタフェースを使って Oracle オブジェクトのデータを読み込む 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ステップについて説明します。このステップは、CustomDatum を手動でインプリメントする場合と、JPublisher を使ってカスタム Java クラスを作成する場合の両方に適用されます。

このステップは、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 クラスを手動でまたは JPublisher を使って作成し、Statement オブジェクト stmt を定義していることを前提にしています。

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");
   ors.next();
   ```

   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 getCustomDatum() メソッドを使って、カスタム Java 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。getCustomDatum() メソッドから oracle.sql.CustomDatum オブジェクトが返されます。このオブジェクトは特定のカスタム Java オブジェクトにキャストできます。

   ```
   Employee emp = (Employee)ors.getCustomDatum(1, Employee.getFactory());
   ```

   または

```
CustomDatum datum = ors.getCustomDatum(1, Employee.getFactory());
```

この例では、Employee はカスタム Java クラス名、ors は OracleResultSet オブジェクト名です。

getCustomDatum() を使わないときは、JDBC ドライバの場合、標準 JDBC ResultSet.getObject() メソッドを使って CustomDatum データを取り出します。ただし、型マップにエントリがなければなりません。型マップでは、指定されたオブジェクト型で使われるファクトリ・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、CustomDatum がインプリメントされます。対応するファクトリ・クラスは EmployeeFactory で、CustomDatumFactory がインプリメントされます。

次の文を使って、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() の形式を使い、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 型名に対して使われるファクトリ・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 Java クラスに get メソッドを定義している場合、そのメソッドを使っ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、型 CHAR の Name および型 NUMBER の EmpNum（従業員番号）が定義されている場合、Java 文字列を返す getName() メソッドおよび整数値を返す 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.getName();
int empnumber = emp.getEmpNum();
```

注意：または、コール可能 Statement オブジェクトにデータをフェッチすることもできます。OracleCallableStatement クラスには、getCustomDatum() メソッドも定義されています。

CustomDatum インタフェースを使ってデータを Oracle オブジェクトに書き込む この項では、JPublisher を使ってカスタム Java クラスを作成したとき、または他の方法で CustomDatum インプリメンテーションを選択したときに、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ステップを説明します。

このステップは、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 クラスを手動でまたは JPublisher を使って作成していることを前提にしています。

注意 : データベースの INSERT および UPDATE の実行時には、型マップは使えません。

---

1. カスタム Java クラスに set メソッドを定義している場合、そのメソッドを使ってアプリケーションの Java 変数のデータから Java データ型オブジェクトの属性に書き込みます。

   ```
   emp.setName(empname);
   emp.setEmpNum(empnumber);
   ```

   この文で使われている emp オブジェクト、empname 変数、empnumber 変数の詳細は、4-72 ページの「CustomDatum インタフェースを使って Oracle オブジェクトのデータを読み込む」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って、Oracle オブジェクトを更新する準備済みの Oracle 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. 準備済みの Oracle 文の setCustomDatum() メソッドを使って、Java データ型オブジェクトを準備済みの文にバインドします。

   ```
   opstmt.setCustomDatum(1, emp);
   ```

   setCustomDatum() メソッドでは、カスタム Java クラスの toDatum() メソッドをコールし、データベースに書き込むことができる oracle.sql.STRUCT オブジェクトを取り出します。

   このステップでは、setObject() メソッドを使って Java データ型をバインドすることもできます。たとえば

   ```
   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うことができます。

JDBC での JPublisher の使用方法

JPublisher とは、Oracle オブジェクトにマップする Java クラスを作成する Oracle ユーティリティのことです。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シエートすることにより、Oracle オブジェクトのデータを保持できます。JPublisher が生成したクラスには、SQL と Java 間でデータを変換するメソッド、およびクラス属性の getter および setter メソッドが含まれます。

機能を追加する場合は、必要に応じてサブクラスを作成し、機能を追加できます。JPublisher には、元のクラスを再生成する必要がある場合に、記述したコードへの参照を作成する機能があります。メソッドの追加によって生成されたクラスを編集することもできますが、JPublisher を実行してクラスを再生成する予定がある場合にはお薦めしません。この方法で修正したクラスを、JPublisher を実行して再生成した場合、変更（追加したメソッド）は上書きされます。JPublisher の出力を別のファイルに切り替えた場合でも、変更をそのファイルにマージする必要があります。

JPublisher を使わずにカスタム Java クラスを作成することもできますが、通常は使った方が便利です。JPublisher の詳細は、『Oracle8*i* Jpublisher User's Guide』を参照してください。

JPublisher マップ・オプション

JPublisher を使ってカスタム Java クラスをインプリメントする場合は、次の 3 つから属性のマッピング方法を選択できます。

- Oracle マッピング
- JDBC マップ
- オブジェクト JDBC マップ

JPublisher のコマンド行オプションを使って、この 3 つのマッピング・オプションのいずれかを選択できます。マップ・オプションの詳細は、『Oracle8*i* Jpublisher User's Guide』を参照してください。

Oracle オブジェクト参照の使用

この項のトピックは次のとおりです。

Oracle オブジェクト参照を、オブジェクト表に格納されているオブジェクトに定義できます。ただし、表の列に格納されているオブジェクト値に対して、オブジェクト参照を定義できません。

SQL では、オブジェクト参照（REF）は完全に表示されます。たとえば、EMPLOYEE オブジェクトへの参照は、REF だけではなく、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、oracle.sql.REF クラスのインスタンスとして実体化され、完全に表示されません。このため、EMPLOYEE REF を選択すると、oracle.sql.REF オブジェクトが返されます。REF の種類を確認するには、そのオブジェクトの getBaseTypeName() メソッドを使います。このメソッドを使うと、オブジェクトの SQL 型が返されます。この場合は、EMPLOYEE が返されます。

オブジェクト参照は、SQL 型の基本形の 1 つです。オブジェクト参照のアクセスおよび操作のステップは、SQL 型の他の基本形で行うステップと同じです。

注意 : 配列はオブジェクトの同様に構造体型ですが、参照できません。

JDBC では、REF に対して、次のサポートを提供しています。

- SELECT リストの列
- IN または OUT バインド変数
- Oracle8 オブジェクト属性
- コレクション（配列）型オブジェクトの要素

JPublisher を使ってカスタム Java クラスを生成すると、参照クラスも生成されます。これらの参照クラスは、oracle.sql.REF のエクステンションで、oracle.sql.REF クラスと違って完全に表示されます。たとえば、Oracle オブジェクト EMPLOYEE を定義すると、JPublisher により Employee クラスおよび EmployeeRef クラスが生成されます。

オブジェクト参照の取り出し

次の例では、REF を取り出す方法を示すため、最初に Oracle オブジェクト型 ADDRESS を定義しています。

```
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、次のステップに従ってください。

1. 標準 SQL SELECT 文を使って、データベース表の REF 列から参照を取り出します。
2. getREF() を使って、結果セットから Address 参照を取得し、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() メソッドを使って、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 3 つのステップ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た Statement 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");
rs.next();
REF ref = rs.getREF(1);
Address a = (Address)(ref.getValue());
```

他の SQL 型と同様に、結果セットの getObject() メソッドを使っ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、

```
REF ref = (REF)rs.getObject(1);
```

getREF() のかわりに getObject() を使うこともできます。

オブジェクト参照をコール可能文に渡す

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、次のステップに従って、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() メソッドを使って登録します。

```
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 型コード（この場合は OracleTypes.REF）です。sql_type_name は、このオブジェクト参照がポイントする STRUCT 名です。たとえば、OUT パラメータが前項の例と同様に ADDRESS オブジェクトへの REF の場合は、ADDRESS には、sql_type_name が渡される必要があります。

3. コールを実行します。

```
ocs.execute()
```

オブジェクト参照を介したオブジェクト値に対するアクセスと更新

次の参照を介して、Java Address オブジェクトの作成およびデータベースの ADDRESS オブジェクトの更新を行うことができます。（Address クラスのコンストラクタに必要なものは省略されています。）この例は、有効な REF オブジェクトがあらかじめ取り出してあることを前提としています。

```
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 オブジェクトが更新されます。

オブジェクト参照を準備済みの文に渡す

他の SQL 型を渡すときと同様に、オブジェクト参照を準備済みの文に渡します。準備済みの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。

前の例に続けて次の準備済みの文を使い、ROWID に基づく Address 参照を更新します。

```
PreparedStatement pstmt =
   conn.prepareStatement ("update PEOPLE set ADDR_REF = ? where ROWID = ?");
pstmt.setREF (1, addr_ref);
pstmt.setROWID (2, rowid);
```

配列の使用

この項のトピックは次のとおりです。

- 配列とその要素の取り出し
- 配列を準備済みの文に渡す
- 配列をコール可能文に渡す
- 型マップを使って配列要素をマップする

oracle.sql.ARRAY クラスを使うと、JDBC プログラムで配列とそのデータに対するアクセスおよび操作を行うことができます。oracle.sql.ARRAY クラスによって、oracle.jdbc2.Array インタフェースがインプリメントされます。

JDBC では、次の配列をサポートしています。

- 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

> **注意** : JDBC 2.0 の配列は、Oracle のコレクションに相当します。

配列には、varray（可変長配列）およびネストした表が含まれます。oracle.sql.ARRAY クラスのメソッドを使うと、配列とそのデータに対するアクセスおよび操作が行うことができます。この配列には varray またはネストした表を使うことができます。つまり、varray アクセスする場合も、ネストした表にアクセスする場合も、コードを変更する必要はありません。このメソッドでは、varray またはネストした表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。

Oracle では、名前付き配列だけをサポートします。このため、SQL 型名を指定して配列型を記述する必要があります。次の SQL 構文では、配列の作成時に SQL 型名が割当てられます。

```
CREATE TYPE <sql_type_name> AS <datatype>
```

配列には、ネストした表または varray を使うこともできます。

varray はサイズが変更可能な配列なので "varray" という名前が付けられています。varray には順序付けられたデータ要素が設定されています。指定された varray の要素のデータ型は、すべて同じです。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。配列型の宣言時に、サイズの最大値を指定する必要があります。たとえば、

```
CREATE TYPE myNumType AS VARRAY(10) OF NUMBER;
```

この文では、10 要素以下の NUMBER 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。

ネストした表は、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。この表は 1 列で構成され、列の形式はビルトイン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。次の構文を使って、ネストした表を作成します。

```
CREATE TYPE myNumList AS TABLE OF integer;
```

この文は、型 integer のネストした表に使われる表型が定義された SQL 型名として、myNumList を指定しています。

この項の残りの部分では、配列データに対するアクセスおよび更新方法について説明します。配列オブジェクトを手動で作成する方法など、oracle.sql.ARRAY クラスについては 4-13 ページの「クラス oracle.sql.ARRAY」を参照してください。配列形式の表を作成し、内容の操作および印刷を行うコード例については、7-16 ページの「配列のサンプル」を参照してください。

配列とその要素の取り出し

配列を取り出すと、oracle.sql.ARRAY オブジェクトを取得し、各配列要素は実体化された Java 配列オブジェクトまたは結果セット・オブジェクトとして返されます。

結果セットを OracleResultSet オブジェクトにキャストし、getARRAY() メソッドを使うと、選択した SQL 配列を取り出して結果セットに格納できます。このとき、oracle.sql.ARRAY が返されます。結果セットをキャストしない場合は、oracle.sql.ResultSet クラスの getObject() メソッドを使ってデータを取得し、出力を oracle.sql.ARRAY にキャストできます。

ARRAY オブジェクトに配列を作成すると、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って、データを取り出すことができます。

- getArray()
- getOracleArray()
- getResultSet()

これらのメソッドをさまざまな方法で使って型マップを指定し、SQL データ型を Java データ型にマップする方法を選択できます。また、配列の要素またはサブセットをすべて取り出すことができるメソッドも提供しています。(ただし、配列全体を取り出すときと比べて、配列のサブセットを取り出すときはパフォーマンスが低下することもあります。)

注意：リリース 8.1.5 から、配列の索引は 1 から始まるようになりました。以前のリリースでは、索引は 0 から始まっていました。

---

getArray() メソッド： `getArray()` メソッドでは、配列の要素値を `java.lang.Object[]` 配列に取り出します。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。

`getArray()` では、データを `oracle.sql.*` オブジェクトの配列として実体化し、型マップは使いません。Oracle が提供する `getArray(map)` メソッドを使うと、型マップを指定できます。また、`getArray(index,count)` メソッドを使うと、配列のサブセットを取得できます。

getOracleArray() メソッド： `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。要素は、元の配列の SQL 型データに対応する `oracle.sql.*` データ型に変換されます。

注意：`getOracleArray()` メソッドは、Oracle 固有のエクステンションなので、`oracle.jdbc2.ARRAY` JDBC 2.0 インタフェースには含まれません。

`getOracleArray()` メソッドでは、データを `oracle.sql.*` オブジェクトの配列として実体化し、型マップは使いません。Oracle では、`getOracleArray(index,count)` も提供しています。

getResultSet() メソッド： `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返します。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、索引は配列における要素の位置を表します。順序が定義されていないネストした表の索引は、特定の問合せによって返された要素の順序です。

ネストした表からデータを取り出すときは、`getResultSet()` を使うことをお薦めします。ネストした表の要素数には、制限はありません。メソッドから返された `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()` メソッドを使うと、ネストした表の内容を取得できます。また、`getArray()` を使うと、ネストした表のすべての内容が一度に返されます。

`getResultSet()` メソッドでは接続のデフォルト型マップを使って Oracle オブジェクトの SQL 型と対応する Java データ型とのマップを決定します。接続のデフォルト型マップを使わない場合は、`getResultSet(map)` を使っ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)` メソッドを使うと、配列のサブセットを取り出すことができます。

配列のすべての要素を取り出す

getArray() を使ってデータ型基本型を取り出すと、要素の値を含む java.lang.Object が返されます。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、

```
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。values 配列には、型 java.math.BigDecimal の要素の配列が含まれています。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。

同様に、getResultSet() を使ってデータ型基本形の配列を返すと、JDBC ドライバにより ResultSet オブジェクトが返されます。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、

```
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 BigDecimal 要素値が格納されます。

型マップに従って配列要素を取り出す

デフォルトでは、getArray() または getResultSet() を使うと、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っ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) メソッドを使って、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

```
Object[] object = (Object[])objArray.getArray(map);
```

この例の objArray は oracle.sql.ARRAY オブジェクト、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 として返されます。

getResultSet(map) メソッドは、getArray(map) と似ています。

配列で型マップを使う方法の詳細は 4-86 ページの「型マップを使って配列要素をマップする」を参照してください。

配列要素のサブセットを取り出す

配列のサブセットを取り出すには、索引と件数を渡して、取り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って、Java 型に変換します。たとえば、

```
Object object = arr.getArray(index, count, map);
Object object = arr.getArray(index, count);
```

getResultSet() を使った例です。

```
ResultSet rset = arr.getResultSet(index, count, map);
ResultSet rset= arr.getResultSet(index, count);
```

getOracleArray() を使った例です。

```
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。

oracle.sql.Datum として配列を取り出す

oracle.sql.Datum[] 配列を返すには、getOracleArray() を使います。返される配列の要素は、SQL 配列要素の SQL データ型に対応する oracle.sql.* 型です。たとえば、

```
Datum arraydata[] = arr.getOracleArray();
```

arr は、oracle.sql.ARRAY オブジェクトです。配列とその内容を取り出す例については、7-16 ページの「配列のサンプル」を参照してください。

例 : 結果セットからデータ型基本型の配列を取得し印刷する 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 型オブジェクトである my_array に配列データが取り出されます。

my_array は型 oracle.sql.ARRAY 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印刷されます。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、標準出力に印刷されます。

```
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 type code " + 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() を使っ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() メソッドを使っ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って、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));
};
```

配列を準備済みの文に渡す

次のステップに従って、配列を準備済みの文に渡します。(配列をコール可能文に渡すときと同様のステップを使います。)

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まだ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、4-13 ページの「クラス oracle.sql.ARRAY」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、4-79 ページの「配列の使用」を参照してください。

2. 準備済みの文に 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, elements);
```

descriptor はステップ 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 を表しています。これらのオブジェクトは、適切な SQL 型のロー・バイトに変換されます。

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. 準備済みの文を OraclePreparedStatement にキャストし、OraclePreparedStatement オブジェクトの setARRAY() メソッドを使って配列を準備済みの文に渡します。

```
(OraclePreparedStatement)stmt.setARRAY(parameterIndex, array);
```

parameterIndex はパラメータ索引、array はステップ 2 で作成した oracle.sql.ARRAY オブジェクトを表します。

5. 準備済みの文を実行します。

注意 : 配列は、IN または OUT バインド変数を使うことができます。

配列をコール可能文に渡す

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、次のステップに従って 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 regsiterOutParameter() 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 型コード、および sql_type_name は配列型名を表します。sql_type は OracleTypes.ARRAY です。

3. 問合せを実行します。

```
ocs.executeQuery()
```

型マップを使って配列要素をマップする

配列に Oracle オブジェクトが含まれる場合は、型マップを使って、配列の各オブジェクトを対応する Java クラスと関連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` として要素が返されます。

型マップを使って、配列の Oracle オブジェクトおよび対応する Java クラスのマッピングを決定するときは、それらが型マップに定義されていない場合、型マップにその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、4-60 ページの「型マップ」を参照してください。

次の例では、型マップを使っ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、ネストした表です。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` オブジェクトのネストした表型です。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 部に属する全従業員をカスタム Java オブジェクト `EmployeeObj` として選択するには、型マップに `EMPLOYEE`SQL 型と `EmployeeObj` カスタム Java オブジェクト・クラス間のマッピングを作成する必要があります。

作成するには、まず、文および結果セット・オブジェクトを作成し、次に、`SALES` 部に関連付けられた `EMPLOYEE_LIST` を選択し、結果セットに格納します。結果セットを `OracleResultSet` にキャストすると、`getARRAY()` メソッドによって `EMPLOYEE_LIST` オブジェクトが `employeeArray` オブジェクトに取り出されます。

> **注意**：この例の `EmployeeObj` カスタム Java オブジェクト型は、`SQLData` インタフェースをインプリメントしています。`EmployeeObj` 型を作成するコード例については、7-20 ページの「カスタム Java クラスの作成」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、EMPLOYEE SQL 型を EmployeeObjJava 型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"
Dictionary map = conn.getTypeMap();
map.put("EMPLOYEE", Class.forName("EmployeeObj"));
```

EMPLOYEE_LIST から SQL EMPLOYEE オブジェクトを取り出します。オブジェクトを取り出すには、oracle.jdbc2.Array クラスの getArray() メソッドを employeeArray に適用します。このメソッドにより、オブジェクト配列が返されます。getArray() メソッドでは、employees オブジェクト配列に EMPLOYEE オブジェクトが返されます。

```
// 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];
     ...
   }
```

Oracle エクステンションの追加情報

この項のトピックは次のとおりです。

この項では、JDBC 2.0 仕様のデータ型以外の Oracle エクステンションについて説明します。また、追加のデータ型エクステンションおよびパフォーマンス・エクステンションについて説明します。

パフォーマンス・エクステンション

Oracle JDBC ドライバでは、これらのエクステンションをサポートすることにより、データベースへのラウンドトリップが減少し、パフォーマンスを向上しています。

- 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。処理されなかったデータは、クライアント側バッファに格納され、後でクライアントからアクセスされます。プリフェッチの行数は、目的に応じて設定できます。
- バッチ更新を行うことにより、データベースへのラウンドトリップが減少します。クライアント側に更新するデータを一定数保存してから、データベースに転送し、すべての更新を実行します。
- 列型を指定することにより、通常の JDBC プロトコルで、問合せの実行およびその結果の取り出しを行うときの効率が向上しました。
- データベース・メタデータ `TABLE_REMARKS` 列を廃止したので、高度な外部結合操作をする必要がなくなりました。

Oracle では、これらのパフォーマンス・エクステンションをサポートするために、接続プロパティ・オブジェクトのエクステンションの一部をサポートしています。プロパティ・オブジェクト・エクステンションを使って、プリフェッチとバッチ更新の `remarksReporting` フラグおよびデフォルト値を設定できます。詳細は、4-98 ページの「接続プロパティの Oracle エクステンション」を参照してください。

注意 : プリフェッチおよびバッチ更新のエクステンションは、JDBC 2.0 標準が発表される前に開発されました。このため、JDBC 2.0 には対応していません。

行のプリフェッチ

Oracle JDBC ドライバを使って、問合せ中に結果セットが移入されるときに、クライアントにプリフェッチする行数を設定することができます。この機能を使うと、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。

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を使うと、整数行プリフェッチ設定と指定された Statement オブジェクトを関連付けることができます。JDBC では、問合せ時に複数の行を一度にフェッチできます。つまり、プリフェッチ設定が N のときは、JDBC によって問合せ条件と一致する N 行がフェッチされ、すべての行が一度でクライアントに返されます。次に、その N 行から `next()` コールを実行すると、問合せ条件と一致する次の 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。接続の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` オブジェクトでない場合は、Statement オブジェクトをいずれかのオブジェクトにキャストします。
2. Statement オブジェクトの `setRowPrefetch()` メソッドを使って、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のプリフェッチ数を確認するには、Statement オブジェクトの `getRowPrefetch()` メソッドを使います。このメソッドでは整数が返されます。

次の方法で、接続の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. `OracleConnection` オブジェクトの `setDefaultRowPrefetch()` メソッドを使って、プリフェッチするデフォルト行数を設定し、目的のデフォルトを指定する整数を渡します。デフォルトの現在の設定を確認する場合は、`OracleConnection` オブジェクトの `getDefaultRowPrefetch()` メソッドを使います。このメソッドでは整数が返されます。

行の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、作成時に関連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）の場合は、JDBC では、どちらの型の値も実際に読み込んでいない場合でも、文のプリフェッチ設定が 1 に変更されます。

`Properties` オブジェクトを引数として指定する、`DriverManager` クラスの `getConnection()` メソッドの形式を使う場合は、この方法で接続の行プリフェッチのデフォルト値を設定します。オブジェクトおよび接続プロパティの詳細は、3-5 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」および 4-98 ページの「接続プロパティの Oracle エクステンション」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。この例は、`oracle.jdbc.driver.*` クラスをインポートしていることを前提としています。

```
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();
```

データベースのバッチ更新

Oracle JDBC ドライバを使うと、準備済みの文の挿入および更新をクライアント側で蓄積し、サーバーに一括して送ることにより、サーバーへのラウンドトリップを減らすことができます。さまざまなバインド変数で同じ文を繰り返し使う場合に効果的です。

通常 JDBC では、文の `executeUpdate()` メソッドがコールされるたびに、データベースへのラウンドトリップを行い、準備済みの文を実行します。しかし、Oracle バッチ更新機能を使うと、特定のバッチ値を各準備済みの Statement オブジェクトに関連付けることができ

ます。Oracle JDBC では、準備済みの文の実行要求を蓄積します。バッチ値に到達したら、すべての要求をデータベースに自動的に渡し、実行します。

バッチ更新の制限事項 CallableStatement に OUT パラメータが設定されている場合を除き、CallableStatement を使ってバッチ更新を実行できます。OUT パラメータが設定されている場合は、直前のバッチ値がドライバによって自動的に上書きされ、1 にリセットされます。

JDBC 2.0 PreparedStatement インタフェースの addBatch() および executeBatch() メソッドは使わないでください。これらのメソッドは、OraclePreparedStatement に関連付けられたメソッドによって提供される機能と互換性がありません。

Oracle 準備済みの文のバッチ値に関わらず、文のバインド変数がストリーミング型である（またはストリーミング型に変換される）場合は、JDBC によりバッチ値が 1 に設定され、実行待ちの要求すべてがデータベースに送られて実行されます。

JDBC では、接続が commit 要求または close 要求を受け取ると、または文が close 要求を受け取ると、自動的に文の sendBatch() メソッドが実行されます。

Properties オブジェクトを引数として指定する DriverManager.getConnection() メソッドの形式を使う場合は、接続のデフォルト・バッチ値をオブジェクトに設定できます。Properties オブジェクトの詳細は、4-98 ページの「接続プロパティの Oracle エクステンション」を参照してください。

デフォルトのバッチ更新値は 1 です。

各文にバッチ更新値を設定する バッチ更新値を OraclePreparedStatement オブジェクトに適用すると、各 Oracle 準備済みの文にバッチ値が設定されます。各文に設定したバッチ値により、接続に設定された値が上書きされます。また、Oracle 接続のすべての Oracle 準備済みの文に適用されるデフォルト・バッチ値を設定するには、OracleConnection オブジェクトにデフォルト・バッチ値を適用します。

特定の準備済みの文に Oracle バッチ値機能を適用するには、次のステップを実行します。

1. 準備済みの文を記述し、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. 準備済みの文を OraclePreparedStatement オブジェクトにキャストし、setDefaultExecuteBatch() メソッドを適用します。この例では、文のデフォルト・バッチ・サイズが 2 に設定されています。

```
((OraclePreparedStatement)ps).setDefaultExecuteBatch(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.outPrintln (" Statement Execute Batch Value " +
    ((OraclePreperedStatement)ps).getExecuteBatch());
```

3. この時点でデータベースに更新の実行文を送った場合は、データはデータベースに送られません。かわりに、executeUpdate() コールから 0 が返されます。

```
// 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 行に入力値のセットおよび更新の実行を入力した場合は、executeUpdate() へのバッチ・コール回数は、バッチ値 2 と等しくなります。データがデータベースに送られ、1 回のラウンドトリップに 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 ();
```

デフォルト・バッチ更新値を上書きする バッチ値に到達する前に蓄積された文を実行する場合は、OraclePreparedStatement オブジェクトの sendBatch() メソッドを使用します。たとえば、

1. 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。この例では、接続のすべての文のデフォルト・バッチを 50 に設定します。

```
((OracleConnection)conn).setDefaultExecuteBatch (50);
```

2. 準備済みの文を記述し、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 ());
```

この時点では更新の実行は発生しません。ps.executeUpdate() メソッドにより "0" が返されます。

3. 第 2 行および executeUpdate() に入力値を入力しても、データはデータベースに送られません。文と接続のバッチ・デフォルト値が同じ 50 であるためです。

```
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);
```

println 文の rows の値が "0" である点に注意してください。

4. この時点で sendBatch() を適用すると、これまでバッチされた 2 つの実行が、1 回のラウンドトリップでデータベースに送られます。sendBatch() メソッドから更新された行数が返されます。sendBatch() のプロパティを println で使うと、更新された行数が印刷されます。

```
// 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 ();
```

接続のバッチ更新値を設定する Oracle 接続では、任意の Oracle 準備済みの文にデフォルト・バッチ値を指定できます。指定するには、OracleConnection オブジェクトに setDefaultExecute() メソッドを設定します。たとえば、次の文では、conn 接続オブジェクトに属するすべての準備済みの文に対して、デフォルト・バッチ値に 20 が設定されます。

```
((OracleConnection)conn).setDefaultExecuteBatch(20);
```

この例では、接続に属するすべての準備済みの文に対してデフォルト・バッチ値が設定されますが、各文で setDefaultBatch() をコールするとその設定値を上書きできます。

バッチ値の確認 getExecuteBatch() メソッドを使うと、特定の Oracle 準備済みの文または Oracle 接続に属するすべての準備済みの文の、現在のデフォルト・バッチ値を確認できます。たとえば、

```
Integer batch_val = ((OraclePreparedStatement)ps).getExecuteBatch();
```

または

```
Integer batch_val = ((OracleConnection)conn).getDefaultExecuteBatch();
```

例：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.jdbc.driver.*` クラ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");

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
ps.close();
```

注意：

- 各文には独自のバッチ数が指定されています。特定の文の実行だけがバッチ数に追加されます。
- バッチによって遅延した更新は、他の問合せの結果に反映されます。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返されます。

```
UPDATE emp SET name = "Sue" WHERE name = "Bob";
SELECT name FROM emp WHERE name = "Sue";
```

列型の再定義

Oracle JDBC ドライバを使うと、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。表を記述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って、結果セットの列に使われる型を決定します。次に、JDBC では、問い合わ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。

列型を再定義する場合の制限事項 この機能を使用するには、必要な結果セットの各列に対して、データ型を指定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数と一致しない場合は、`SQLException` で処理が失敗します。

オブジェクトまたはオブジェクト参照の列型は定義できません。

問合せに対して列型を再定義する 問合せに対して列型を再定義するには、次のステップを実行します。

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()` メソッドを使って、この `Statement` 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で次の要素を決定します。
   - 列索引（位置）
   - 必要な戻りデータ型のコード（このデータ型は列型と異なることがあります。）

Oracle 固有型の oracle.jdbc.driver.OracleTypes および標準型の java.sql.Types または OracleTypes に従います。（Types および OracleTypes では、標準型の定数は同じ値です。）

4. 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に渡します。

   - 列索引（整数）
   - 型コード（整数）

     java.sql.Types クラスの静的定数または Oracle 固有型では、oracle.jdbc.driver.OracleTypes クラスの静的定数（Types.INTEGER、Types.FLOAT、Types.VARCHAR、OracleTypes.VARCHAR、OracleTypes.ROWID など）を使います。

   - （オプション）最大フィールド・サイズ（整数）

   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, type);
   ```

   または

   ```
   stmt.defineColumnType(column_index, type, max_size);
   ```

   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() メソッドを使って、最大フィールド・サイズが小さく設定した場合、またはデータ型の元の最大サイズが小さい場合は、この最大サイズよりも小さなデータが返されます。つまり、返されるデータのサイズは、次の値の最小値になります。

   - defineColumnType() に設定された最大フィールド・サイズ、または
   - setMaxFieldSize() に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

このステップが終了したら、文の executeQuery() メソッドを使って問合せを実行します。

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.driver.* クラ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返されます。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 インプリメンテーション以外では提供されません。

定義のエクステンションでは、JDBC 型を使っ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、" 固有の " JDBC 型に定義でき、多くの場合、Types.CHAR または Types.VARCHAR で定義されます。

表 4-6 は、defineColumnType() メソッドで使うことができる有効な列定義引数の一覧です。

表 4-6 有効な列型

列内の Oracle SQL 型	defineColumnType() を使って再定義できる列定義引数
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() メソッドを使って TABLE_REMARKS 列をレポートすると、高度な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、JDBC ドライバでは、デフォルトでは TABLE_REMARKS 列がレポートされません。

OracleConnection オブジェクトの setRemarksReporting() メソッドに TRUE 引数を渡すと、TABLE_REMARKS レポートが実行使用可能になります。

標準 java.sql.Connection オブジェクトを使っている場合、setRemarksReporting() を使うには、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。次の文を使うと TABLE_REMARKS レポートが使用可能になります。

```
( (oracle.jdbc.driver.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() メソッドでは、catalog、schemaPattern、columnNamePattern、および procedureNamePattern パラメータは同様に処理されます。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 にはカタログは複数ありませんが、パッケージが複数あります。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（返された ResultSet 内の catalog 列）の両方に適用されます。入力では、コンストラクト "" （空白文字列）はプロシージャおよび引数をパッケージなしで受け取ります。つまり、スタンドアロン・オブジェクトです。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が返されます。("%" 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パターン（必要に応じて、SQL ワイルド・カードが使えます。）になります。
- schemaPattern: Oracle のすべてのオブジェクトには、スキーマを設定する必要があります。スキーマが設定されていないオブジェクトの情報を返しても意味を持ちません。このため、コンストラクト "" （空白文字列）は、入力時に現在のスキーマ（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・カードと共にパターンとしても使うことができます。
- 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、コンストラクト "" 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。

接続プロパティの Oracle エクステンション

DriverManager.getConnection() メソッドの特定の形式を使うと、URL およびプロパティ・オブジェクトを指定できます。

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

URL の形式は次のとおりです。

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

URL のほかにも標準 Java Properties クラスのオブジェクトを入力として使用できます。たとえば、

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

表 4-7 は、Oracle JDBC ドライバがサポートする接続プロパティの一覧です。defaultRowPrefetch、remarksReporting、および defaultBatchValue の Oracle エクステンションも含まれます。

表 4-7 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	データベースへの接続文字列 setDefaultRowPrefetch() を使う場合と等価
defaultRowPrefetch	prefetch	整数型	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 です。
remarksReporting	remarks	ブール型	getTables() および getColumns() が TABLE_REMARKS をレポートする場合は真。setRemarksReporting() を使う場合と等価。デフォルト値は偽です。
defaultBatchValue	batchvalue	整数型	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 です。

次の例では、データベースに接続する前に、java.util.Properties.put() メソッドを使ってパフォーマンスのエクステンション・オプション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.driver.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);
```

追加の型エクステンション

Oracle JDBC ドライバでは、Oracle 固有の ROWID および REF CURSOR データ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。

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、ROWID SQL データのラッパーとして提供されます。

ROWID では、java.sql.ResultSet.getCursorName() および java.sql.Statement.setCursorName() JDBC メソッドと同様の機能が提供されます。これらのメソッドは Oracle インプリメンテーションではサポートされません。

問合せに ROWID 擬似列を含めた場合は、ResultSet.getString() メソッドを使って（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() メソッドを使うと、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 クラスを使います。このクラスは、以前のリリースの Oracle JDBC で使われていました。
- ROWID クラスの機能の詳細は、Javadoc を参照してください。

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 タイプ・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の位置（アドレス）が保持されます。このため、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は多数存在し、さまざま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 カテゴリに属するユーザー定義型を識別します。たとえば、

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

次に、カーソル変数をユーザー定義型の DeptCursorTyp 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、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 カテゴリのユーザー定義型またはカーソル変数が返されます。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実体化され、次の方法でアクセスできます。

1. JDBC コール可能文を使って、ストアド・プロシージャをコールします。(出力パラメータがあるため、準備済みの文ではなくコール可能文を使う必要があります。)
2. ストアド・プロシージャにより REF CURSOR が返されます。
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() メソッドを使って REF CURSOR が JDBC ResultSet として実体化されます。
4. 結果セットは要求どおりに処理されます。

例 : REF CURSOR データにアクセスする 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.driver.*;
...
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));}
```

この例では

- CallableStatement オブジェクトは、接続クラスの prepareCall() メソッドを使って作成されます。
- コール可能文によって、REF CURSOR を返す PL/SQL プロシージャがインプリメントされます。
- 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 で使われる Oracle タイプ・コードは、OracleTypes.CURSOR です。
- コール可能文が実行され、REF CURSOR が返されます。
- 標準 JDBC API の Oracle エクステンションである 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 オブジェクトに返されます。

REF CURSOR を使った完全なサンプル・アプリケーションについては、7-14 ページの「REF CURSOR のサンプル」を参照してください。

Oracle JDBC の注意および制限事項

Oracle JDBC インプリメンテーションには次の制限事項があります。ただし、それほど重要な制限でないか、制限を回避する方法があります。

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、4-100 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 "(+)" を使ってください。SQL92 構文の詳細は、5-24 ページの「埋込み SQL92 構文」を参照してください。

PL/SQL TABLE、BOOLEAN、および RECORD 型

Oracle JDBC ドライバでは、PL/SQL TABLE、BOOLEAN、または RECORD 型の引数のコールまたは戻り値がサポートされません。これは、OCI レイヤーの制限事項です。

ブール型のかわりに、BOOLEAN 引数を CHAR または NUMBER として受け取り、BOOLEAN として元のストアド・プロシージャに渡す、追加の PL/SQL ストアド・プロシージャを定義することができます。このトピックの詳細は、6-7 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、ゼロ（0）、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すことができます。

10^{-30} ～（$1 - 10^{-38}$）* 10^{126} を 38 桁の完全精度で表すことができます。

読取り専用接続

読取り専用接続は、サポートされません。読取り専用接続に相当する機能は Oracle にはありません。

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、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、4-97 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報告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、SQLWarning がサポートされていません。

Oracle JDBC ドライバによるエラー処理の詳細は、3-23 ページの「エラー・メッセージと JDBC」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することができました。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);
```

この機能は、JDBC1.0 または 2.0 の仕様にはありません。また、Oracle ではこの機能をサポートしていません。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かったため、2 回目の実行で 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。

5
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。この章は、次の項で構成されています。

- NLS の使用
- アプレットの操作
- サーバー上の JDBC: サーバー・ドライバ
- 埋込み SQL92 構文

NLS の使用

この項は、次のトピックで構成されています。

Oracle の JDBC ドライバは、NLS（National Language Support）をサポートしています。NLS 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・セットを使う場合、ドライバは、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・セットの間の変換をサポートしています。

NLS、NLS 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、『Oracle8*i* NLS 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、『Oracle8*i* リファレンス・マニュアル』を参照してください。

次に、NLS 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()`: `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()`: `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 NLS 変換を行う方法

Oracle のドライバが 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、データベースが使うキャラクタ・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・セットを使っている場合です。この場合、ドライバはデータをデータベース・キャラクタ・セットから Java アプリケーションで使われる UCS-2 に直接変換します。

US7ASCII でも WE8ISO8859P1 でもないキャラクタ・セット（日本語または韓国語）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 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 NLS

JDBC OCI ドライバのインストレーションの場合、データベース・キャラクタ・セットの他にクライアント側のキャラクタ・セットがあることに注意してください。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 NLS_LANG 環境変数の値によって決まります。データベース・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。クライアントが使うキャラクタ・セットは、サーバー上のデータベースで使うキャラクタ・セットと異なる可能性があります。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 3 つの要素を考慮する必要があります。

- データベースのキャラクタ・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・セットのクライアント・データに変換します。NLS_LANG 環境変数の値により、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 の値が指定されていないか、US7ASCII または WE8ISO8859P1 キャラクタ・セットである場合は、JDBC OCI ドライバは、Java を使ってキャラクタ・セットを US7ASCII または WE8ISO8859P1 から直接 UCS-2 に変換します。
- NLS_LANG の値が 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 の値を使ってデータベース・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 ドライバが UTF-8 データを UCS-2 に変換します。

注意 :

- ドライバは NLS_LANG の値を UTF-8 に設定して、Java で実行する変換の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、もっぱら JDBC アプリケーション処理のためだけに行います。
- NLS_LANG パラメータの詳細は、『Oracle8*i* NLS ガイド』を参照してください。

JDBC Thin ドライバおよび NLS

アプリケーションまたはアプレットが JDBC Thin ドライバを使う場合は、Oracle クライアントのインストレーションはありません。このため、C 言語の OCI クライアント変換ルーチンは利用できません。この場合、クライアント変換ルーチンは、JDBC OCI ドライバとは異なります。

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 の UCS-2 に変換します。

データベース・キャラクタ・セットが US7ASCII でも WE8ISO8859P1 でもない場合は、サーバーはクライアントにデータを転送する前にまず UTF-8 に変換します。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。

注意 : OCI ドライバと Thin ドライバは、どちらも同じ NLS の透過的なサポートを提供します。

サーバー・ドライバおよび NLS

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・ドライバがデータベース・キャラクタ・セットを基にキャラクタ・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。

JDBC サーバー・ドライバは ASCII（US7ASCII）および ISO-Latin-1（WE8ISO8859P1）キャラクタ・セットだけをサポートします。

注意 : Java VM は英語（US7ASCII）および ISO-Latin-1（WE8ISO8859P1）キャラクタ・セットだけをサポートします。

NLS の制限

NLS 変換に対するデータ・サイズの制限

CHAR および VARCHAR2 データ型をバインド・コール内で使う場合は、これらのデータ型の最大サイズに制限があります。この制限は、データの破損を防ぐために必要です。この問題は、マルチバイトのキャラクタ・セット・データベースに接続している場合にバインドと共にのみ発生し（定義に対してではない）、CHAR および VARCHAR2 データ型だけに影響があります。

バインドの最大長は次のように制限されます。

CHAR および VARCHAR2 に対しては、データのバイト長が増える可能性のあるキャラクタ・セット変換が行われます。変換の前と後のデータ・サイズの割合は、NLS 比と呼ばれています。変換後のバインド値を 4 KB（Oracle8）または 2 KB（Oracle7）よりも大きくすることはできません。

表 5-1 クライアント側ドライバに対する新しい最大バインド長の制限

ドライバ	サーバー・バージョン	データ型	以前の最大バインド長（バイト）	新しい最大バインド長の制限（バイト）
Thin および OCI	V8	CHAR	2000	min(2000,4000 / NLS_Ratio)
		VARCHAR2	4000	(4000 / NLS_Ratio)

たとえば、Oracle8 Server に接続している場合、次の値を超えてバインドすることはできません。

- CHAR 型の場合、min (2000, 4000 / NLS_RATIO)
- VARCHAR2 型の場合、4000 / NLS_RATIO

表 5-2 に、一般的なサーバー・キャラクタ・セットの NLS 比と最大バインド値の例を示します。

表 5-2 一般的なサーバー・キャラクタ・セットの NLS 比とサイズ制限

サーバー・キャラクタ・セット	NLS 比	Oracle8 Server 上の最大バインド値（バイト）
WE8DEC	1	4000
US7ASCII	1	4000
ISO 8859-1 から 10	1	4000
JA16SJIS	2	2000
JA16EUC	3	1333

アプレットの操作

この項では、JDBC Thin ドライバを使うアプレットを操作する際の基本について説明します。まず、JDBC アプレットの簡単な例から始め、次にアプレットをデータベースに接続するために必要な操作について説明します。ここでは、Web サーバーと同じホスト上で動作していないデータベースに接続している場合の Oracle8 Connection Manager や署名付きアプレットの使い方なども説明します。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実行する方法について説明します。

アプレットのコーディング

JDBC インタフェースをインポートして JDBC エントリ・ポイントにアクセスする場合以外は、他の Java アプレットと同じように JDBC アプレットを記述します。JDK 1.1.1 ブラウザ用と JDK 1.0.2 ブラウザ用のどちらのアプレットをコーディングするかによって、使うコードが多少異なります。どちらの場合も、アプレットは JDBC Thin ドライバを使う必要があります。このドライバは、TCP/IP プロトコルを使ってデータベースに接続します。

JDK 1.1.1 ブラウザ（Netscape 4.*x*、Internet Explorer 4.*x* など）をターゲットにしている場合は、次の操作が必要です。

- `java.sql` パッケージをプログラムにインポートします。`java.sql` パッケージには、標準の JDBC 1.22 インタフェースおよび JDK 1.1.1 クラス・ライブラリの一部が含まれています。
- `oracle.jdbc.driver.OracleDriver()` クラスにドライバを登録して、接続文字列にドライバ名を `thin` として指定します。

JDK 1.0.2 ブラウザ（Netscape 3.*x*、Internet Explorer 3.*x* など）をターゲットにしている場合は、次の操作が必要です。

- `jdbc.sql` パッケージをプログラムにインポートします。

  `jdbc.sql` パッケージは標準の JDK 1.0.2 クラス・ライブラリの一部ではありません。JDBC の一部としてダウンロードする別のライブラリです。JDK 1.0.2 ブラウザでは文字列 "`java`" で始まるパッケージをダウンロードできなかったため、`jdbc.sql` パッケージが作成されました。次善策として、`java.sql` パッケージが `jdbc.sql` という名前

になりました。この名前が変更されたパッケージは、Oracle JDBC 製品と共に出荷されています。

- `oracle.jdbc.dnlddriver.OracleDriver()` クラスにドライバを登録して、接続文字列にドライバ名を `dnldthin` と指定します。

以下の項では、JDK 1.1.1 ブラウザ用アプレットと JDK 1.0.2 ブラウザ用アプレットのコーディングの違いを示します。

- JDK 1.1.1 ブラウザ用アプレットのコーディング
- JDK 1.0.2 ブラウザ用アプレットのコーディング

JDK 1.1.1 ブラウザ用アプレットのコーディング

JDK 1.1.1 ブラウザ用のアプレットをコーディングする場合は、`java.sql` パッケージから JDBC インタフェースをインポートして、Oracle JDBC Thin ドライバをロードします。

```
import java.sql.*;
public class JdbcApplet extends java.applet.Applet
{
 Connection conn; // Hold the connection to the database
 public void init()
 {
 // Register the driver.
 DriverManager.registerDriver (new oracle.jdbc.driver.OracleDriver());
 // Connect to the database.
 conn = DriverManager.getConnection
 ("jdbc:oracle:thin:scott/tiger@www-aurora.us.oracle.com:1521:orcl");
 ...
 }
}
```

この例では、接続文字列にはユーザー名およびパスワードが含まれていますが、これらは、ユーザーから取得した後に `getConnection()` の引数として渡すこともできます。データベース接続の詳細は、3-3 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。

JDK 1.0.2 ブラウザ用アプレットのコーディング

JDK 1.0.2 ブラウザ用のアプレットをコーディングする場合は、`jdbc.sql` パッケージから JDBC インタフェースをインポートして、`oracle.jdbc.dnlddriver.OracleDriver()` クラスからドライバをロードし、接続文字列内に `dnldthin` サブプロトコルを使います。

```
import jdbc.sql.*;
public class JdbcApplet extends java.applet.Applet
{
 Connection conn; // Hold the connection to the database
 public void init ()
```

```
 {
 // Register the driver
 DriverManager.registerDriver (new oracle.jdbc.dnlddriver.OracleDriver());
 // Connect to the database
 conn = DriverManager.getConnection
 ("jdbc:oracle:dnldthin:scott/tiger@www-aurora.us.oracle.com:1521:orcl");
 ...
 }
}
```

データベースへのアプレットの接続

この項は、次のトピックで構成されています。

- Web サーバーと同じホスト上のデータベースへの接続
- 異なるホスト上のデータベースへの接続
- Oracle8 Connection Manager の使用
- 署名付きアプレットの使用

JDBC ドライバを使っ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、アプレットはそのアプレットのダウンロード元であるホスト（Web サーバーが実行されているホスト）に対する TCP/IP ソケットしかオープンできません。つまり、アプレットは Web サーバーと同じホスト上で動作するデータベースに対してだけ接続できます。この場合、アプレットはデータベースに直接接続でき、それ以上の手順は必要ありません。

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、アプレットはサーバーが実行するデータベースではなく、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 を使ってデータベースに接続できます。

または

- Web ブラウザが JDK 1.1.*x* をサポートしている場合は、署名付きアプレットを使ってデータベースに直接接続できます。

この項ではまず、アプレットのダウンロード元のホストと同じホスト（Web サーバーと同じホスト）上のデータベースに接続するという最も単純な例について説明します。その後、異なるホスト上で動作するデータベースに接続する 2 つの方法について説明します。

Web サーバーと同じホスト上のデータベースへの接続

データベースがアプレットのダウンロード元のホストと同じホスト上で動作している場合は、アプレットにデータベースを指定することによってそのデータベースに接続できます。

データベースは DriverManager クラスの getConnection() メソッドの接続文字列に指定します。

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 にあり、SID が ORCL で、ユーザー名 scott、パスワード tiger で接続する場合は、次の 2 つのいずれかの接続文字列を使います。

host:port:sid 構文を使う場合 :

```
String connString="jdbc:oracle:thin:@prodHost:1521:ORCL";
conn = DriverManager.getConnection(connString, "scott", "tiger");
```

TNS キーワード - 値構文を使う場合 :

```
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 - 値のペアを使って 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。

異なるホスト上のデータベースへの接続

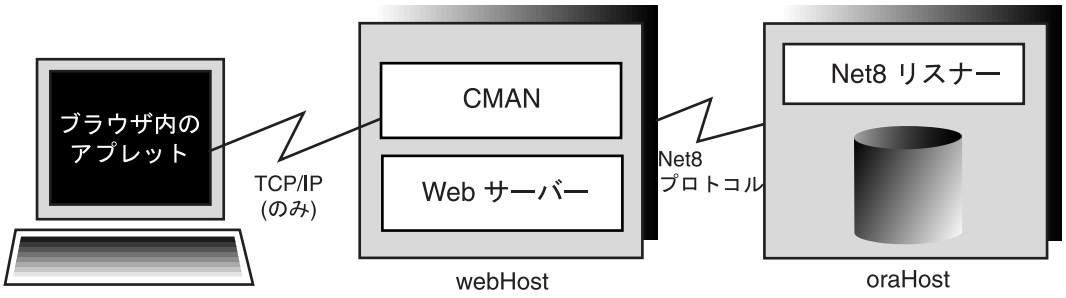
Web サーバーが動作しているホスト以外のホスト上のデータベースに接続する場合は、アプレットのセキュリティの制限に対処する必要があります。これは、Oracle8 Connection Manager または署名付きアプレットを使うことで可能です。

Oracle8 Connection Manager の使用

Oracle8 Connection Manager は、Net8 パケットを受信して別のサーバーに再転送できる、軽量で非常にスケーラブルなプログラムです。Net8 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 ドライバを使うアプレットは、Web サーバー・ホスト上で実行中の Connection Manager に接続し、Connection Manager を使って、Net8 パケットを異なるホスト上で動作する Oracle サーバーにリダイレクトすることができます。

図 5-1 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 5-1 アプレット、Connection Manager およびデータベースの関係



Oracle8 Connection Manager を使うには、この項で説明する 2 つの手順を実行する必要があります。

- Oracle8 Connection Manager のインストールおよび実行
- Oracle8 Connection Manager をターゲットにした接続文字列の記述

Oracle8 Connection Manager のインストールおよび実行 Connection Manager を Web サーバー・ホスト上にインストールする必要があります。Connection Manager は、Oracle8 の配布媒体からインストールします。Connection Manager のインストールの詳細は、『Net8 管理者ガイド』を参照してください。

Web サーバー・ホスト上には、`[ORACLE_HOME]/NET8/ADMIN` ディレクトリに `CMAN.ORA` ファイルを作成する必要があります。`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。`CMAN.ORA` ファイルに入力できるオプションの詳細は、『Net8 管理者ガイド』を参照してください。

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 Net8 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 に設定する必要があります。

CMAN.ORA ファイルにリストされているオプションの説明は、『Net8 管理者ガイド』を参照してください。

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って Oracle8 Connection Manager を起動します。

```
cmctl start
```

アプレットを使うには、ここ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 この項では、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 5-1 について説明します。Connection Manager が実行されている 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");
```

アプレットで上記のような接続文字列を使った場合、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、『Net8 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、まず複数の 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（例 : 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、『Net8 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

ブラウザが JDK 1.1.*x*（Netscape 4.0 など）をサポートしている場合は、署名付きアプレットを使うことができます。署名付きアプレットは、他のマシンへのソケット接続権限を要求できます。設定には、次の操作が必要です。

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 Microsystem 社の「Signed Applet Example」を参照してください。

   ```
   http://java.sun.com/security/signExample/index.html
   ```

2. ソケットをオープンする前に適切なアクセス権を要求するアプレット・コードを含めます。

   Netscape を使っている場合は、コードに次のような文を含めます。

   ```
   netscape.security.PrivilegeManager.enablePrivilege("UniversalConnect");
       Connection conn = DriverManager.getConnection(...);
   ```

   アクセス権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
```

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」ページを参照してください。

   ```
   http://developer.netscape.com/software/signedobj/index.html
   ```

   証明書の取得とインストールについての情報が得られます。

Netscape Capabilities クラスを使った署名付きアプレットの完全な例は、7-30 ページの「署名付きアプレットの作成」を参照してください。

ファイアウォールとアプレットの使用

通常の状況下では、JDBC Thin ドライバを使ったアプレットは、ファイアウォールを通過してデータベースにアクセスすることはできません。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。

この問題は、Net8 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うことによって解決できます。Net8 に準拠したファイアウォールは、多くの優れたベンダーから入手できます。これらのファイアウォールの詳細は、このマニュアルでは扱っていません。

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だけアクセスできます。この場合、そのホストには、Net8 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。

以下の項では、次のトピックについて説明します。

- ファイアウォールの仕組み
- JDBC Thin ドライバを使うアプレットに対する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

ファイアウォールの仕組み

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接続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 1 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。

ファイアウォールを通過して接続するには、以下の項で説明する 2 つの手順を実行する必要があります。

- JDBC Thin ドライバを使うアプレットに対する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 ドライバを使うアプレットに対するファイアウォールの構成

この項の説明では、Net8 に準拠したファイアウォールが実行され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（ローカルでホスト名または環境変数が取得できません）。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、ドライバ内に模造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 ドライバを使う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います。

```
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することはできません。たとえば、次のように指定され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。

上記の接続文字列を、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");
```

アプレットが上記のような接続文字列を使う場合は、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 : 上記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。

上記の例で使われたパラメータの詳細は、『Net8 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問い合せてください。

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、アプレットのクラス・ファイルおよびJDBC ドライバのクラス・ファイル（アプレットのターゲットが JDK 1.1.1 を実行しているブラウザの場合は `classes111.zip`、JDK 1.0.2 を実行しているブラウザの場合は `classes102.zip`）が必要です。

次の手順を実行します。

1. JDBC ドライバのクラス・ファイル `classes111.zip`（または `classes102.zip`）を空のディレクトリに移動します。
2. ドライバのクラスの zip ファイルを解凍します。

   JDK 1.0.2 を実行するブラウザをターゲットにする場合は、以下のテーブルの左側の列にリストされたパッケージを削除します。次に、右側の列にリストされたパッケージがあることを確認します。テーブル内にリストされたパッケージはすべて JDBC に含まれています。

削除するパッケージ :	**あることを確認するパッケージ :**
java.sql	jdbc.sql
java.math	jdbc.math
oracle.jdbc.driver	oracle.jdbc.dnlddriver
oracle.jdbc.dbaccess	oracle.jdbc.dnlddbaccess
oracle.jdbc.oracore	oracle.jdbc.dnldoracore
oracle.jdbc.util	oracle.jdbc.dnldutil
oracle.jdbc.ttc7	oracle.jdbc.dnldttc7
oracle.sql	oracle.sdnldql
oracle.jdbc2	oracle.dnldjdbc2
java.io.Reader	jdbc.io.Reader
java.io.Writer	jdbc.io.Writer

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（または `.jar`）ファイルにします。

   JDK 1.1.1 を実行するブラウザをターゲットにするには、この zip ファイルに次のものが含まれている必要があります。

- classes111.zip から解凍したファイル
- アプレットのクラス
- アプレットで DatabaseMetaData エントリ・ポイントを使っている場合は、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 エントリ・ポイントを使わない場合は、このファイルを省略します。

JDK 1.0.2 を実行するブラウザをターゲットにするには、この zip ファイルに次のものが含まれている必要があります。

- classes102.zip から解凍したファイル（手順 2 で削除したファイル以外）
- アプレットのクラス
- JDBC の classes/jdbc/sql ディレクトリ内の jdbc.sql パッケージにある jdbc インタフェース・ファイル

注意 : JDK 1.0.2 を実行するブラウザをアプレットのターゲットにする場合は、アプレットを zip ファイルでパッケージ化する必要があります。JDK 1.0.2 を実行するブラウザは、.jar ファイルをサポートしていません。

---

5. zip（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えるようにできます。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、次のようにします。

```
<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。以下の項で、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共に、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 パラメータを使います。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された 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 に対する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 タグを使う場合は、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 が使われます。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、現行のディレクトリが my_Dir であるとします。

```
<APPLET WIDTH=500 HEIGHT=200 CODE=JdbcApplet CODEBASE="."
</APPLET>
```

エントリ CODEBASE="." は、このアプレットが現行のディレクトリ（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）の名前を指定します。オラクル社では .zip ファイルの使用をお薦めします。.zip ファイルを使うと、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。

ブラウザのセキュリティと JDK バージョンの考慮点

JDBC Thin ドライバを使うアプレットと Oracle データベース間の通信は、Java TCP/IP ソケット上で行われます。

Netscape 3.0 などの JDK 1.0.2 ベースの Web ブラウザでは、アプレットはそのアプレットのダウンロード元のホストに対するソケットしかオープンできません。Oracle8 の場合、アプレットは、Web サーバーと同じホスト上で動作するデータベースに対してだけ接続できるということになります。異なるホスト上で動作するデータベースに接続する場合は、Oracle8 Connection Manager 経由で接続する必要があります。詳細は、5-9 ページの「Oracle8 Connection Manager の使用」を参照してください。

Netscape 4.0 などの JDK 1.1.1 ベースの Web ブラウザでは、アプレットはソケット接続権限を要求して、Web サーバー・ホストと異なるホスト上で動作す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）してから次のように接続をオープンすることによって、これを実行できます。

```
netscape.security.PrivilegeManager.enablePrivilege
        ("UniversalConnect");
connection = DriverManager.getConnection
        ("jdbc:oracle:thin:scott/tiger@dlsun511:1721:orcl");
```

署名付きアプレットの操作方法の詳細は、ブラウザのドキュメントを参照してください。また、5-12 ページの「署名付きアプレットの使用」も参照できます。

サーバー上の JDBC: サーバー・ドライバ

この項は、次のトピックで構成されています。

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、すべてサーバー・ドライバを使って SQL エンジンにアクセスできます。

サーバー・ドライバは、本来 8.1 データベースおよび Java VM 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（Java VM 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・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、このドライバを使うとローカル・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。Java VM 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関数コールであり、ネットワークは使いません。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 Net8 コールを実行するよりもはるかに高速です。

サーバー側ドライバは、クライアント側ドライバと同じ機能、API および Oracle 拡張要素をサポートします。これにより、アプリケーションのパーティション化が単純になります。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。

サーバー・ドライバを使っ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・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。デフォルトの接続にアクセスするには、Oracle 固有の API `defaultConnection()` メソッドか、または標準 Java `DriverManager.getConnection()` メソッドを使うことができます。

defaultConnection() を使った接続

`oracle.jdbc.driver.OracleServerDriver` クラスの `defaultConnection()` メソッドは、Oracle 拡張要素で常に同じ接続オブジェクトを戻します。文に接続文字列を含める必要はありません。たとえば、次のようにします。

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

```
class JDBCConnection {
  public static Connection connect() throws SQLException {
    Connection conn = null;
    try {
      // connect with the Server driver
         OracleDriver ora = new OracleDriver();
         conn = ora.defaultConnection();
      }

    } catch (SQLException e)
    return conn;
  }
}
```

conn.close 文がないことに注意してください。サーバー・ドライバによって確立されたデフォルトの接続はクローズできません。この接続に対して close() をコールしても何も行われません。

DriverManager.getConnection() を使った接続

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、新しい接続を作成しなくても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関連付けられます。プログラムの一部として複数の型マップを使う場合は、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。

データベースに DriverManager.getConnection() メソッドを使って接続した場合は、接続文字列 jdbc:oracle:kprb: を使います。たとえば、次のようにします。

```
DriverManager.getConnection("jdbc:oracle:kprb:");
```

この文字列にユーザー名およびパスワードを含めることもできますが、サーバーから接続しているため、それらは無視されます。

サーバー・ドライバのセッションおよびトランザクション・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、Java VM 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。

サーバー内の Java アプリケーション・コードを実行した場合は、トランザクション（COMMIT および ROLLBACK）を明示的に実行できます。

Server 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、少し修正するだけ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だけです。

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() メソッドを使って接続する場合は、好みのテキスト・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 "oci8" から "kprb" に変更します。たとえば、次のようにします。

```
Connection conn =
    DriverManager.getConnection ("jdbc:oracle:kprb:@" +
                                 database, user, password);
```

このメソッドを使う利点は、元のプログラムの短い文字列を 1 つ変更するだけでよいことです。欠点は、ドライバが破棄するにもかかわらず、ユーザー、パスワードおよびデータベースの情報を指定しなければならない点です。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・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）を使って接続した場合は、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.driver.OracleDriver ().defaultConnection ();
```

次の例は、JdbcCheckup.java プログラムを書き直したもので、defaultConnection() 接続文を使っています。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共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
        {
            Connection conn =  new oracle.jdbc.driver.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・ドライバの NLS のサポート

Java プログラムに対してサーバー・ドライバがデータベース・キャラクタ・セットの変換処理を行う方法の説明は、5-4 ページの「サーバー・ドライバおよび NLS」を参照してください。

oracle.sql.CHAR データのキャラクタ・セット変換

サーバー・ドライバは、oracle.sql.CHAR のキャラクタ・セットの変換を C 言語で行います。これは、クライアント側ドライバのインプリメンテーションとは異なります。クライアント側ドライバは、oracle.sql.CHAR のキャラクタ・セットの変換を Java で行います。oracle.sql.CHAR クラスの詳細は、4-17 ページの「クラス oracle.sql.CHAR」を参照してください。

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。現行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
- 関数コール構文

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、選択可能な次善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、デフォルトでは有効になっています。JDBC ドライバは、データベースに SQL コードを送信する前にエスケープ置換を実行します。SQL92 構文でない通常の Oracle SQL 構文をドライバで使う場合は、次の文を使います。

```
stmt.setEscapeProcessing(false)
```

注意 : 通常、準備済みの文は `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ないはずです。

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う構文は、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だけがサポートされています。この項では、SQL 文内で使う必要のある日付、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。たとえば、
{d '1998-10-22'} のようになります。JDBC ドライバは、このエスケープ句を同じ意味の Oracle 表現 "22 OCT 1998" 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。たとえば、{t '05:10:45'} のようになります。JDBC ドライバは、このエスケープ句を同じ意味の Oracle 表現 "05:10:45" に置換します。時刻が {t '14:20:50'} として指定された場合は、サーバーが 24 時間制のクロックを使っていると仮定すると、同じ意味の Oracle 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、{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、スカラー関数のすべてをサポートしてはいません。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.driver.OracleDatabaseMetaData インタフェースおよび標準 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います。

- 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、関数キーワード '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おうとしたとします。

```
{fn concat ("Oracle", "8i") }
```

Java アプリケーションを実行すると、エラー "Non supported SQL92 token at position xx: fn" が戻されます。次善策は、Oracle SQL 構文を使うことです。

たとえば、次のような埋込み SQL92 構文で fn キーワードを使うかわりに、

```
Statement stmt = conn.createStatement ();
stmt.executeUpdate("UPDATE emp SET ename = {fn CONCAT('My', 'Name')}");
```

次のように Oracle SQL 構文を使います。

```
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 文字だけの一致に使います）。これらの文字を文字列内で文字どおりに使う場合は、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う場合は、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: 円記号（\）をエスケープ文字として使う場合は、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。次善策は、Oracle 外部結合構文を使うことです。

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 構文を使います。

```
Statement stmt = conn.createStatement ();
ResultSet rset = stmt.executeQuery
        ("SELECT ename, dname
         FROM emp a, dept b WHERE a.deptno = b.deptno(+)
         ORDER BY ename");
```

関数コール構文

Oracle の JDBC ドライバは、次に示す関数コール構文をサポートしています。

戻り値のないコール:

```
{ call procedure_name (argument1, argument2,...) }
```

戻り値のあるコール:

```
{ ? = call procedure_name (argument1, argument2,...) }
```

SQL92 から SQL への構文変換例

SQL92 を標準の 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、関数コール、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.driver.OracleSql.parse()` メソッドが変換を実行します。

```
import oracle.jdbc.driver.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));
      }
}
```

次のコードは、対応する SQL 構文を書き出す出力です。

```
{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')
```

6

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、以下のトピックで説明します。

- JDBC およびマルチスレッド
- 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・プロシージャ
- トランザクション分離レベルと Oracle サーバー

JDBC およびマルチスレッド

Oracle JDBC ドライバはマルチスレッドを使うプログラムを完全にサポートしています。次のプログラム例では、デフォルトのオラクル従業員データベース emp を使います。このプログラムでは複数のスレッドが作成されます。各スレッドは接続をオープンし、emp テーブルの内容に対する問合せをデータベースに送信します。次に、スレッド、従業員名、およびその従業員に割り当てられている従業員 ID を表示します。

次のコードを入力してプログラムを実行します。

```
java JdbcMTSample [number_of_threads]
```

コマンド行の *number_of_threads* には、作成するスレッドの数を入力します。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。

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

public class JdbcMTSample extends Thread
{
    // Set default number of threads to 10
    private static int NUM_OF_THREADS = 10;

    int m_myId;

    static     int c_nextId = 1;
    static  Connection s_conn = null;

    synchronized static int getNextId()
    {
        return c_nextId++;
    }

    public static void main (String args [])
    {
        try
        {
            // Load the JDBC driver //
            DriverManager.registerDriver
                     (new oracle.jdbc.driver.OracleDriver());

            // If NoOfThreads is specified, then read it
            if (args.length > 1) {
                System.out.println("Error: Invalid Syntax. ");
                System.out.println("java JdbcMTSample [NoOfThreads]");
                System.exit(0);
            }
            else if (args.length == 1)
```

```
            NUM_OF_THREADS = Integer.parseInt (args[0]);

          // 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();
          }

          // wait for all threads to end
          for (int i = 0; i < NUM_OF_THREADS; i++)
          {
                  threadList[i].join();
          }

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

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

```
            // 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));

            // Close all the resources
            rs.close();
              stmt.close();
            if (conn != null)
                conn.close();
            System.out.println("Thread " + m_myId +  " is finished. ");
        }
        catch (Exception e)
        {
            System.out.println("Thread " + m_myId + " got Exception: " + e);
            e.printStackTrace();
            return;
        }
    }

}
```

パフォーマンスの最適化

次の機能を使って、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
- 行のプリフェッチ
- バッチ更新

自動コミット・モードの無効化

自動コミット・モードによって、SQL 文の後で毎回実行およびコミットを発行するかどうかを指定します。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・モードを使用禁止にすることができます。

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、どちらか早い方でコミットが発生します。`ResultSet` を返す文の場合は、`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 つの文でアウトプット・パラメータ値や複数の結果が返されることがあります。この場合は、すべての結果およびアウトプット・パラメータ値が取り出された時点でコミットが発生します。

自動コミット・モードを使用禁止にする（`setAutoCommit(false)`）と、その接続の SQL 文は、JDBC ドライバによって、`commit()` または `rollback()` 文で終了するトランザクションとしてグループ化されます。

例 : AutoCommit を使用禁止にする 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・モードになるので、この例では自動コミットを使用禁止に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.driver.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 ();
...
```

行のプリフェッチ

Oracle JDBC ドライバでは、問合せ中に結果セットが移入されているときに、プリフェッチする行数をクライアントに設定できます。デフォルトのプリフェッチ行数は 10 です。行データをクライアントにプリフェッチしておくと、サーバーとのラウンドトリップの回数を削減できます。反対に、標準的な JDBC では、結果セットを一度に 1 行ずつフェッチするので、1 行ごとにデータベースとのラウンドトリップが必要です。

行のプリフェッチの値は、個々の文に対して設定することも、接続のすべての文に対して設定することもできます。行のプリフェッチについての説明、およびその使用方法は、4-88 ページの「行のプリフェッチ」を参照してください。

バッチ更新

Oracle JDBC ドライバでは、準備済みの文の挿入および更新をクライアントで蓄積し、指定したバッチ値に達したときに一括してサーバーに送信できます。この機能を使うと、サーバーとのラウンドトリップを減少できます。デフォルトのバッチ値は 1 です。

バッチ値は、Oracle の個々の準備済みの文または Oracle 接続のすべての準備済みの文に対して設定することもできます。バッチ更新についての説明およびその使用方法は、4-90 ページの「データベースのバッチ更新」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・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 16 以上の OCI 接続をオープンする

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 列では異なります。

この問題を避けるには、Statement オブジェクトで setMaxFieldSize() メソッドを使い、すべての列に返すことのできるように、最大データ長を設定します。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・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。ResultSet および Statement クラスの close() メソッドを使ってクリーン・アップ・ルーチンが実行されます。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、その接続に関連付けられたオーブン中の文が JDBC ドライバによってクローズされ、サーバー側でカーソル・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 レイヤーの制限事項のために、JDBC ドライバでは、PL/SQL ストアド・プロシージャにブール型パラメータを渡すことができません。PL/SQL プロシージャにブール値が含まれている場合、制限を避けるには、その PL/SQL プロシージャを 2 番目の PL/SQL プロシージャでラップし、次のプロシージャがその引数を INT として受け入れて最初のストアド・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。

次に、ストアド・プロシージャの例を示します。boolProc はブール型パラメータを渡し、2 番目のプロシージャ boolWrap はブール値の整数値の置換えを行います。

```
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 16 以上の OCI 接続をオープンする

任意の時点で、1 プロセスで約 16 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われる（3 ～ 4 個のファイル記述子が使われます）ことがあります。

サーバーで 16 以上のプロセスを使える場合は、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。解決策は、制限数を大きくすることです。

基本的なデバッグ・プロシージャ

この項では、JDBC プログラムの 4 つのデバッグ方法について説明します。

- 例外の検出
- JDBC コールのロギング
- ネットワーク・イベント検出のための Net8 トレース
- サード・パーティ・ツールの使用

例外の検出

JDBC プログラムで発生するエラーのほとんどは、例外として処理されます。Java には、例外を捕捉するための `try...catch` 文、およびスタック・トレースを表示するための `printStackTrace()` メソッドがあります。

次のコードは、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 ドライバでエラーが処理される様子を示すため、サンプル `Employee.java` には意図的に次の不正なコードが追加されています。

```
// 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 (); }
```

列索引を 5 に変更して、故意にエラーを発生させています。このプログラムを実行すると、次のエラー・メッセージが表示されます。

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

JDBC ドライバのエラー処理の方法、および SQLException() メソッドと printStackTrace() メソッドについては、3-23 ページの「エラー・メッセージと JDBC」を参照してください。

JDBC コールのロギング

java.io.PrintStream.DriverManager.setLogStream() メソッドを使うと JDBC コールをログに記録できます。このメソッドによって、DriverManager とすべてのドライバによって使用されるロギング / トレースの PrintStream が設定されます。コード中で JDBC コールのロギングを開始する位置に、次の行を挿入してください。

```
DriverManager.setLogStream(System.out);
```

ネットワーク・イベント検出のための Net8 トレース

クライアントおよびサーバーで Net8 トレースを使用可能にすると、Net8 を介して送信されたパケットを検出できます。クライアント側でのトレースは、JDBC OCI ドライバ以外では使えません。JDBC Thin ドライバに対するサポートはありません。トレースおよびトレース・ファイルの読取りについての詳細は、『Net8 管理者ガイド』に記載されています。

トレース機能を使うと、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得することができます。この情報は読取り可能ファイルに出力され、エラーの原因の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 Net8 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ら、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。これらのパラメータの詳細は、『Net8 管理者ガイド』を参照してください。

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われるため、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、必要なときにだけ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独立した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・ツールの使用

Intersolv 社の JDBCSpy および JDBCTest などのツールを使って、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 Oracle サーバー

Oracle サーバーでは、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。レベルの取得および設定を行うには、oracle.jdbc.driver.OracleConnection クラスの次のメソッドを使います。

- getTransactionIsolation(): この接続の現在のトランザクション分離レベルを取得します。
- setTransactionIsolation(): TRANSACTION_* 値の 1 つを使ってトランザクション分離レベルを変更します。

7

サンプル・アプリケーション

この章では、JDBC の先進機能と Oracle エクステンションを際立たせた、サンプル・アプリケーションについて説明します。次のトピックが含まれます。

- JDBC の基本機能を利用したサンプル・アプリケーション
- JDBC 2.0 に準拠した Oracle エクステンション対応のサンプル・アプリケーション
- 他の Oracle エクステンション用のサンプル・アプリケーション
- Oracle オブジェクト用にカスタマイズした Java クラス
- 署名付きアプレットの作成
- JDBC サンプル・コードと SQLJ サンプル・コード

JDBC の基本機能を利用したサンプル・アプリケーション

この項は、JDBC の基本機能を示すコード例を示します。

ストリーム・データ

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。この項のコード例では、JDBC OCI ドライバを使ってデータベースへ接続したり、Java ストリームを使って LONG データの挿入とフェッチを実行します。

```
import java.sql.*;                                          // line 1
import java.io.*;

class StreamExample
{
  public static void main (String args [])
       throws SQLException, IOException
  {
    // Load the driver
    DriverManager.registerDriver(new oracle.jdbc.driver.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);                             // line 18

    // 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                                     // line 34
    stmt.execute ("create table streamexample (NAME varchar2 (256),
          DATA long)");
```

```
    File file = new File ("StreamExample.java");            // line 37
    InputStream is = new FileInputStream ("StreamExample.java");
    PreparedStatement pstmt =
      conn.prepareStatement ("insert into streamexample (name, data)
           values (?, ?)");
     pstmt.setString (1, "StreamExample");
     pstmt.setAsciiStream (2, is, (int)file.length ());
     pstmt.execute ();                                      // line 44

    // 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                                    // line 51
    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 ();                                          // line 66
    }
  }
}
```

Lines 1-18: 必要なクラスをインポートします。JDBC OCI ドライバを `DriverManager.registerDriver()` メソッドを使ってロードします。`getConnection()` を使って、ユーザー `scott`、パスワード `tiger` でデータベースにアクセスします。データベース URL `jdbc:oracle:oci8:@` を使用します。オプションで、`@` 記号の後にデータベース名を入力することもできます。`AUTOCOMMIT` を使用禁止にして、パフォーマンスを向上させます。これを行わない場合、ドライバは各 SQL 文の後に実行およびコミット・コマンドを発行します。

Line 34: `VARCHAR` 型の `NAME` 列および `LONG` 型の `DATA` 列を持つ表 `STREAMEXAMPLE` を作成します。

Lines 37-44: StreamExample.java の内容を表に挿入します。そのために、Java ファイルの入力ストリーム・オブジェクトを作成します。次に、文字データを NAME 列に挿入し、ストリーム・データを DATA 列に挿入する文を準備します。setString() を使って NAME データを挿入します。また、setAsciiStream() を使ってストリーム・データを挿入します。

Line 46: 表の問合せを行い、DATA 列の内容を結果セット内に取得します。

Line 51-66: 結果セットの最初の行からデータを取得し、InputStream オブジェクト gif_data に格納します。FileOutputStream を作成し、指定されたファイル・オブジェクトに書込みを行います。次に、gif ストリームの内容を読み取り、ファイル example.out へ書き込みます。

JDBC 2.0 に準拠した Oracle エクステンション対応のサンプル・アプリケーション

この項は、次の Oracle エクステンション用のサンプル・コードを含みます。

- LOB のサンプル
- BFILE のサンプル

LOB のサンプル

このサンプルは、OCI 8 ドライバでの基本的な LOB サポート機能を例証します。LOB 列を含む表の作成方法を示します。また、LOB からの読取り、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、4-41 ページの「LOB の使用」を参照してください。

いくつかのコメントの変更を除き、次のサンプルは、Demo/samples/oci8/object-samples ディレクトリの LobExample.java プログラムと同様です。

```
import java.sql.*;                                          // line 1
import java.io.*;
import java.util.*;

// Importing the Oracle Jdbc driver package
// makes the code more readable
import oracle.jdbc.driver.*;

// Import this to get CLOB and BLOB classes
import oracle.sql.*;

public class NewLobExample1
{
```

```
public static void main (String args [])
     throws Exception
{
  // Load the Oracle JDBC driver
  DriverManager.registerDriver(new oracle.jdbc.driver.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 auto commit is off
  conn.setAutoCommit (false);                              // line 26

  // 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 but we gleefully ignore it
  }                                                        // line 38

  // Create a table containing a BLOB and a CLOB             line 40
  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',
         '0101010101010101010101010101', 'onetwothreefour')");
  stmt.execute ("INSERT INTO basic_lob_table VALUES ('two',
'02020202020202020202020202', 'twothreefourfivesix')");
                                                           // line 49
  System.out.println ("Dumping lobs");

  // Select the lobs
  ResultSet 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 lob contents
    dumpBlob (conn, blob);
    dumpClob (conn, clob);

    // Change the lob contents
    fillClob (conn, clob, 2000);
    fillBlob (conn, blob, 4000);
  }
                                                          // line 68
  System.out.println ("Dumping lobs again");

  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);
  }
}                                                         // line 82

// 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                     line 91
  char[] buffer = new char[10];

  // length of characters read
  int length = 0;

  // fetch data                                           line 98
  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();
}                                                          // line 108

// 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                                  line 120
  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();
}
                                                           // line 135
// 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);        // line 147

    i += chunk;
    if (length - i < chunk)
        chunk = (int) length - i;
```

```
    }
    outstream.close();
  }                                                              // line 154

  // Utility function to write data to a Blob
  static void fillBlob (Connection conn, BLOB blob, long length)
    throws Exception
  {
    OutputStream outstream = blob.getBinaryOutputStream();

    int i = 0;

    byte [] data = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };            // line 165
   int chunk = data.length;

    while (i < length)
    {
      data [0] = (byte)i;
      outstream.write(data, 0, chunk);

      i += chunk;
      if (length - i < chunk)
          chunk = (int) length - i;
    }
    outstream.close();
  }
}                                                                // line 175
```

Lines 1-26: 必要な java.* および oracle.* クラスをインポートします。DriverManager.registerDriver() メソッドを使ってドライバを登録し、DriverManager.getConnection() を使ってデータベースに接続します。データベース URL jdbc:oracle:oci8:@ を使用し、ユーザー scott、パスワード tiger で接続します。オプションで、@ 記号の後にデータベース名を入力することもできます。

setAutoCommit(false) を使って AUTOCOMMIT 機能を無効にし、パフォーマンスを向上させます。これを行わない場合、ドライバは各 SQL 文の後に実行およびコミット・コマンドを発行します。

Lines 27-38: 文オブジェクトを作成します。basic_lob_table という名前の既存の表を削除します。次に、basic_lob_table ディレクトリを新規に作成し、LOB をインラインに格納します。

Lines 40-49: SQL 文を使って、行番号を VARCHAR2 として格納する列、BLOB 列、および CLOB 列の 3 つを含む表を作成します。次に、データを表の 2 つの行に挿入します。

Lines 50-68: SELECT を使って表の内容を選択し、結果セットに格納します。

LOB を取得します。getBLOB() および getCLOB() メソッドは、ロケータを LOB データに返します。LOB の内容を取得するには、コードをさらに記述する必要があります。（このプログラムの後の部分で定義します。）getBLOB() および getCLOB() メソッドを使用するには、結果セットを OracleResultSet オブジェクトにキャストします。次に dump 関数をコールして LOB の内容を表示し、fill 関数をコールして LOB の内容を変更します。dump および fill 関数は、このプログラムの後の部分で定義されます。

Lines 69-82: LOB の内容変更後に、再び LOB を表示します。SELECT 文を使って、表の内容を選択して結果セットに格納し、dump 関数を適用します。dump 関数は、このプログラムの後の部分で定義されます。

Lines 84-108: ユーティリティ関数 dumpClob を定義して、CLOB の内容を表示します。CLOB の内容を文字ストリームとして読み込みます。getCharacterStream() メソッドを使用して、READER ストリーム・オブジェクトを取得します。一時文字配列を設定して、文字データを 10 文字のチャンクに読み込みます。

CLOB の内容の読込みおよび表示を行うループを設定します。CLOB の長さも表示されます。作業完了時に、入力ストリームをクローズします。

Lines 110-135: ユーティリティ関数 dumpBlob を定義して、BLOB の内容を表示します。BLOB の内容をバイナリ・ストリームとして読み込みます。getBinaryStream() メソッドを使って、InputStream ストリーム・オブジェクトを取得します。一時バイト配列を設定して、バイナリ・データを 10 文字のチャンクに読み込みます。

BLOB の内容の読込みおよび表示を行うループを設定します。BLOB の長さも表示されます。作業完了時に、入力ストリームをクローズします。

Lines 136-154: ユーティリティ関数 fillClob を定義して、データを CLOB に書き込みます。fillClob 関数には、CLOB ロケータおよび CLOB の長さが必要です。CLOB への書込みを行う場合は、getCharacterOutputStream() メソッドを使って WRITER オブジェクトを取得します。

ループを設定して、索引値および文字列 Hello World の一部を CLOB に書き込みます。作業完了時に、WRITER ストリームをクローズします。

Lines 156-175: ユーティリティ関数 fillBlob を定義して、データを BLOB に書き込みます。fillBlob 関数には、BLOB ロケータおよび BLOB の長さが必要です。BLOB への書込みを行う場合は、getBinaryOutputStream() メソッドを使って OutputStream オブジェクトを取得します。

BLOB へ書き込むデータのバイト配列を定義します。while ループにより、データのバリエーションが BLOB へ書き込まれます。作業完了時に、OutputStream オブジェクトをクローズします。

BFILE のサンプル

このサンプルは、OCI 8 ドライバでの基本的な BFILE サポート機能を例証します。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、4-41 ページの「LOB の使用」を参照してください。

いくつかのコメントの変更を除き、次のサンプルは、Demo/samples/oci8/object-samples ディレクトリの FileExample.java プログラムと同様です。

```
import java.sql.*;                                          // line 1
import java.io.*;
import java.util.*;

//including this import makes the code easier to read
import oracle.jdbc.driver.*;

// needed for new BFILE class
import oracle.sql.*;
                                                          // line 10
public class NewFileExample1
{
  public static void main (String args [])
       throws Exception
  {
    // Load the Oracle JDBC driver                            line 16
    DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());

    // Connect to the database
    // You can put a database name after the @ sign in the connection URL.
    //
    // The example creates a DIRECTORY and you have to be connected as
    // "system" to be able to run the test.
    // If 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);
                                                                  // line 32
    // Create a Statement
    Statement stmt = conn.createStatement ();

    try                                                       // line 36
    {
```

```
    stmt.execute ("DROP DIRECTORY TEST_DIR");
  }
  catch (SQLException e)
  {
    // An error is raised if the directory does not exist.  Just ignore it.
  }                                                          // line 43
  stmt.execute ("CREATE DIRECTORY TEST_DIR AS '/temp/filetest'");

  try                                                        // line 46
  {
    stmt.execute ("drop table test_dir_table");
  }
  catch (SQLException e)
  {
    // An error is raised if the table does not exist.  Just ignore it.
  }
                                                             // line 54
  // 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                          // line 64
  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);
  }
}                                                             //line 75

// Utility function to dump the contents of a Bfile          line 77
static void dumpBfile (Connection conn, BFILE bfile)
  throws Exception
{                                                           // line 80
  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: ");                  // line 84

    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();
    }                                                        // line 108

    // Close input stream
    instream.close();

    // close file handler
    bfile.closeFile();
  }                                                          // line 115
}
```

Lines 1-32: 必要な java.* および oracle.* クラスをインポートします。DriverManager.registerDriver() メソッドを使ってドライバを登録し、getConnection() メソッドを使ってデータベースに接続します。データベース URL jdbc:oracle:oci8:@ を使用し、ユーザー system、パスワード manager で接続します。オプションで、@ 記号の後にデータベース名を入力することもできます。

setAutoCommit(false) を使って AUTOCOMMIT 機能を無効にし、パフォーマンスを向上させます。これを行わない場合、ドライバは各 SQL 文の後に実行およびコミット・コマンドを発行します。

Lines 33-44: 文オブジェクトを作成します。TEST_DIR という名前の既存のディレクトリを削除します。次に、TEST_DIR ディレクトリを新規作成し、BFILE を格納します。この作業を行うユーザーまたはシステム管理者は、任意のファイル名を使用できます。

Lines 46-53: test_dir_table という名前の既存の表を削除します。

Lines 55-63: 表を作成し、ファイルに移入します。SQL 文を使って、2 つの列を持つ表 test_dir_table を作成します。列の 1 つは VARCHAR2（one や two など）で行番号を示し、もう 1 つの列は BFILE ロケータを保持します。

SQL 文を使って、データを表に挿入します。第 1 行では、行番号を第 1 列に挿入します。また、BFILENAME キーワードを使って BFILE、TEST_DIR 内の file1 を第 2 列に挿入します。第 2 列も同様の作業を行います。

Lines 64-75: SELECT を使って表の内容を選択し、結果セットに格納します。表の内容を取得するループを設定します。getString() を使って行番号データを取得し、getBFILE() を使って BFILE ロケータを取得します。BFILE は Oracle 固有のデータ型であり、getBFILE() は Oracle エクステンションであるため、結果セット・オブジェクトを OracleResultSet オブジェクトにキャストします。

dumpBfile() メソッド（プログラムの後の部分で定義）を使って、BFILE の内容および BFILE のさまざまな統計情報を表示します。

Line 77: dumpBfile() メソッドを定義して、BFILE の内容および BFILE のさまざまな統計情報を表示します。dumpBfile() メソッドは、BFILE ロケータを入力として取ります。

Lines 80-83: getName()、fileExists()、および isFileOpen() メソッドを使って、BFILE の名前、および BFILE が存在してオープンしているかどうかを返します。BFILE をオープンしていなくても、これらのメソッドを BFILE に適用できます。

Lines 84-108: BFILE の内容の読込みと表示を行います。まず、BFILE をオープンします。BFILE の内容は、バイナリ・ストリームとして読取りが可能です。getBinaryStream() メソッドを使って、入力ストリーム・オブジェクトを取得します。ストリームによる BFILE データの読込み先のチャンク・サイズを決定します。また、一時バイト配列を設定してデータを格納します。

BFILE の内容の読込みおよび表示を行うループを設定します。BFILE の長さも表示されます。

Lines 110-115: 作業完了時に、入力ストリームと BFILE をクローズします。

他の Oracle エクステンション用のサンプル・アプリケーション

この項は、次の Oracle エクステンション用のサンプル・コードを含みます。

- REF CURSOR のサンプル
- 配列のサンプル

REF CURSOR のサンプル

次に、JDBC を使ってデータ・サーバー内にストアド・パッケージを作成し、REF CURSOR 型カテゴリに対して get を使用して問合せの結果を取得する、完全なサンプル・プログラムを示します。REF CURSOR の詳細は、4-101 ページの「Oracle REF CURSOR タイプ・カテゴリ」を参照してください。

いくつかのコメントの変更を除き、次のサンプルは、Demo/samples/oci8/object-samples ディレクトリの RefCursorExample.java プログラムと同様です。

```
import java.sql.*;                                          // line 1
import java.io.*;
import oracle.jdbc.driver.*;

class RefCursorExample
{
   public static void main(String args[]) throws SQLException
   {
      //Load the driver.
      DriverManager.registerDriver(new oracle.jdbc.driver.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");
                                                            // line 16
      // Create the stored procedure.
      init(conn);

      // Prepare a PL/SQL call.                                 line 20
      CallableStatement call =
         conn.prepareCall("{ ? = call java_refcursor.job_listing (?) }");

      // Find out who all the sales people are.                 line 24
      call.registerOutParameter(1, OracleTypes.CURSOR);
      call.setString(2, "SALESMAN");
      call.execute();
```

```
      ResultSet rset = (ResultSet)call.getObject(1);

      // Output the information in the cursor.                line 30
      while (rset.next())
         System.out.println(rset.getString("ENAME"));
   }

// Utility function to create the stored procedure
                                                              // line 36
   static void init(Connection conn) throws SQLException
   {
      Statement stmt = conn.createStatement();
                                                              // line 40
      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;");
                                                              // line 45
      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_cursor;");                       // line 53
   }
}
```

Lines 1-16: 必要な java.* および oracle.* クラスをインポートします。DriverManager.registerDriver() メソッドを使ってドライバを登録し、getConnection() メソッドを使ってデータベースに接続します。データベース URL jdbc:oracle:oci8:@ を使用し、ユーザー scott、パスワード tiger で接続します。オプションで、@ 記号の後にデータベース名を入力することもできます。

Lines 18-29: java_refcursor PL/SQL プロシージャの job_listing 関数へのコール可能文を作成します。コール可能文は、job=SALESMAN の情報を含む行へのカーソルを返します。OracleTypes.CURSOR を出力パラメータとして登録します。setObject() メソッドは、値 SALESMAN をコール可能文に渡します。コール可能文を実行すると、結果セットには job=SALESMAN を満たす表の行へのカーソルが含まれます。

Lines 30-33: 結果セットを反復して、従業員オブジェクトの従業員名の部分を出力します。

Lines 40-45: java_refcursor パッケージのパッケージ・ヘッダーを定義します。パッケージ・ヘッダーは、戻り型および関数シグネチャを定義します。

Lines 46-53: java_refcursor パッケージのパッケージ本体を定義します。パッケージ本体は、job の値に基づいて行を選択するインプリメンテーションを定義します。

配列のサンプル

次に、JDBC を使って VARRAY を含む表を作成する方法を示す、完全なサンプル・プログラムを示します。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、4-79 ページの「配列の使用」を参照してください。

いくつかのコメントの変更を除き、次のサンプルは、Demo/samples/oci8/object-samples ディレクトリの ArrayExample.java プログラムと同様です。

```
import java.sql.*;                                          // line 1
import oracle.sql.*;
import oracle.jdbc.oracore.Util;
import oracle.jdbc.driver.*;
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.driver.OracleDriver());

    // Connect to the database
    // You need to put your database name after the @ symbol in
    // the connection URL.
    //
    // The sample retrieves an varray of type "NUM_VARRAY" and
    // materializes the object as an object of type ARRAY.
    // A new ARRAY is then inserted into the database.

    // Please replace hostname, port_number and sid_name with
    // the appropriate values

    Connection conn =
      DriverManager.getConnection
("jdbc:oracle:oci8:@(description=(address=(host=hostname)(protocol=tcp)(port=port_
number))(connect_data=(sid=sid_name)))", "scott", "tiger");

    // It's faster when auto commit is off
    conn.setLines (false);                                  // line 32

    // Create a Statement
```

```
  Statement stmt = conn.createStatement ();              // line 35

  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
  }                                                      // line 47

  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);                                    // line 54

  //now insert a new row

  // create a new ARRAY object
  int elements[] = { 300, 400, 500, 600 };               // line 59
  ArrayDescriptor desc = ArrayDescriptor.createDescriptor("NUM_VARRAY", conn);
  ARRAY newArray = new ARRAY(desc, conn, elements);
                                                         // line 62
  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);
}                                                        // line 70

public static void showResultSet (ResultSet rs)         // line 72
  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 type code
                          " + array.getBaseType());
      System.out.println ("Array is of length " + array.length());
                                                                    // line 86
      // get Array elements
      BigDecimal[] values = (BigDecimal[]) array.getArray();

      for (int i=0; i<values.length; i++)
      {
        BigDecimal value = values[i];
        System.out.println(">> index " + i + " = " + value.intValue());
      }
    }
  }
}                                                        // line 97
```

Lines 1-32: 必要な java.* および oracle.* クラスをインポートします。DriverManager.registerDriver() メソッドを使ってドライバを登録し、getConnection() メソッドを使ってデータベースに接続します。この getConnection() の使用例では、Net8 名前値ペアを使って host を hostname として、protocol を tcp として、port を 1521 として、sid を orcl として、user を scott として、password を tiger として指定します。

setAutoCommit(false) を使って AUTOCOMMIT 機能を無効にし、パフォーマンスを向上させます。これを行わない場合、ドライバは各 SQL 文の後に実行およびコミット・コマンドを発行します。

Lines 35-47: 文オブジェクトを作成し、任意の定義済みの表または varray_table 型や num_varray 型を削除します。

Lines 49-54: num_varray 型を、NUMBER を含む varray として作成します。1 つの列を持つ表 varray_table を作成し、num_varray 型のデータを格納します。表に 2 行のデータを挿入します。値 100 および 200 は、どちらも num_varray 型です。showResultSet() メソッド（プログラムの後の部分で定義）を使って、表に含まれた配列の情報を表示します。

Lines 59-61: まず、整数要素の配列を定義して、varray_table に挿入します。次に、新規 ARRAY オブジェクトの作成に使用する配列記述子オブジェクトを作成します。配列記述子オブジェクトを作成するには、配列型（NUM_ARRAY）の SQL 型名および接続オブジェクトを createDescriptor() メソッドに渡します。次に、配列記述子、接続オブジェクト、および整数要素の配列を配列記述子オブジェクトに渡して、新しい配列オブジェクトを作成します。

Lines 63-70: 新しい配列オブジェクトを varray_table に挿入する文を準備します。準備した文オブジェクトを OraclePreparedStatement オブジェクトにキャストして、setARRAY() メソッドを利用します。

表の配列内容を取得するために、SQL SELECT 文を記述および実行します。再び showResultSet() メソッド（プログラムの後の部分で定義）を使って、表に含まれた配列の情報を表示します。

Lines 72-85: showResultSet() メソッドを定義します。このメソッドは、結果セットをループして、中に含まれる配列の情報を返します。このメソッドは結果セット getARRAY() メソッドを使用して、配列を oracle.sql.ARRAY オブジェクトに返します。このために、結果セットを OracleResultSet オブジェクトにキャストします。ARRAY オブジェクトが使用可能になると、Oracle エクステンション getSQLTypeName()、getBaseType() および length() を適用して、配列の SQL 型名、配列要素の SQL タイプ・コード、および配列の長さの復帰および表示が可能になります。

Lines 87-97: ARRAY オブジェクトの getArray() メソッドを使用して、varray 要素にアクセスできます。varray には SQL 番号が含まれるため、getArray() の結果を java.math.BigDecimal 配列にキャストします。その後、値の配列を反復して個々の要素を引き出します。

Oracle オブジェクト用にカスタマイズした Java クラス

この項に含まれるサブセクションは、次のとおりです。

- SQLData のサンプル
- CustomDatum のサンプル

この項では、Oracle オブジェクトに対応したカスタム Java クラスの作成に必要なコードの例を示します。カスタム・クラスは、SQLData または CustomDatum インタフェースをインプリメントすることにより作成できます。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 と CustomDatum は、どちらも Java オブジェクトを SQL オブジェクトから移入しますが、CustomDatum インタフェースの方がより強力な機能を備えています。CustomDatum には、Java オブジェクトの移入機能に加え、必ずしもオブジェクトであるとは限らない SQL 型からオブジェクトを実体化する機能もあります。このため、Oracle データベース内の任意のデータ型から CustomDatum オブジェクトを作成できます。これは、オブジェクトの直列化が可能な RAW データの場合に特に有用な機能です。

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、4-63 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。独自にコードを記述して、このインタフェースをインプリメントするカスタム Java クラスを作成することもできます

が、Oracle ユーティリティの JPublisher を使うとカスタム・クラスをより簡単に作成できます。JPublisher により作成されるカスタム・クラスは、CustomDatum インタフェースをインプリメントします。

CustomDatum インタフェースの詳細は、4-68 ページの「CustomDatum インタフェース」を参照してください。JPublisher ユーティリティの詳細は、『Oracle8*i* JPublisher User's Guide』を参照してください。

SQLData のサンプル

この項のサンプル・コードは、指定された SQL 型に対応したカスタム Java 型の作成方法を示します。また、サンプル・プログラムのコンテキスト内でカスタム Java クラスを使用する方法を示します。さらに、SQL 型をカスタム Java 型にマップするコードも含まれます。

SQL オブジェクト定義の作成

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、文字列 EmpName（従業員名）、および整数 EmpNo（従業員番号）という 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 Java クラスの作成

次のプログラムは、SQL 型 EMPLOYEE に対応するカスタム Java クラス EmployeeObj をインプリメントします。EmployeeObj のインプリメンテーションには、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留意してください。また、EmployeeObj カスタム Java クラスの Java 定義は、SQLData インタフェースをインプリメントし、get メソッドおよび必要な readSQL() メソッドと writeSQL() メソッドのインプリメンテーションを含みます。

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

public class EmployeeObj implements SQLData
 {
  private String sql_type;

  public String empName;
  public int empNo;

  public EmployeeObj()
  {
```

```
  }
                                                                  // line 14
public EmployeeObj (String sql_type, String empName, int empNo)
  {
    this.sql_type = sql_type;
    this.empName = empName;
    this.empNo = empNo;
   }                                                              // line 20

  ////// implements SQLData //////

  // define a get method to return the SQL type of the object  line 24
  public String getSQLTypeName() throws SQLException
  {
    return sql_type;
  }                                                              // line 28

  // define the required readSQL() method                        line 30
  public void readSQL(SQLInput stream, String typeName)
    throws SQLException
  {
    sql_type = typeName;

    empName = stream.readString();
    empNo = stream.readInt();
  }
  // define the required writeSQL() method                       line 39
  public void writeSQL(SQLOutput stream)
    throws SQLException
  {
    stream.writeString(empName);
    stream.writeInt(empNo);
  }
}
```

Lines 1-14: 必要な `java.*` および `oracle.*` パッケージをインポートします。カスタム Java クラス `EmployeeObj` を定義して、`SQLData` インタフェースをインプリメントします。`EmployeeObj` は、後の部分で `EMPLOYEE` SQL オブジェクト型のマップ対象となるクラスです。`EmployeeObj` には、SQL 型名、従業員名および従業員番号の 3 つの属性が含まれます。SQL 型名は、カスタム Java クラスが表す Oracle オブジェクトの、完全修飾された SQL 型名（*schema.sql_type_name*）を表す Java 文字列です。

Lines 24-28: `getSqlType()` メソッドを定義して、カスタム Java オブジェクトの SQL 型を返します。

Lines 30-38: SQLData インタフェースの定義に必要な、readSQL() メソッドを定義します。readSQL() メソッドは、ストリーム SQLInput オブジェクトおよび読込み中のオブジェクト・データの SQL 型名を取ります。

Lines 39-45: SQLData インタフェースの定義に必要な、writeSQL() メソッドを定義します。writeSQL() メソッドは、ストリーム SQLOutput オブジェクトおよび読込み中のオブジェクト・データの SQL 型名を取ります。

カスタム Java クラスの使用方法

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得し、その内容を出力します。

いくつかのコメントの変更を除き、次のサンプルは、Demo/samples/oci8/object-samples ディレクトリの SQLDataExample.java プログラムと同様です。

```
import java.sql.*;                                          // line 1
import oracle.jdbc.driver.*;
import oracle.sql.*;
import java.math.BigDecimal;
import java.util.Dictionary;

public class SQLDataExample
{
   public static void main(String args []) throws Exception
  {

    // Connect to the database
    DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver ());
    OracleConnection conn = (OracleConnection)
      DriverManager.getConnection("jdbc:oracle:oci8:@",
                                  "scott", "tiger");        // line 16

    // in the type map, add the mapping of EMPLOYEE SQL     // line 18
    // type to the EmployeeObj custom Java type
    Dictionary map = conn.getTypeMap();
    map.put("EMPLOYEE", Class.forName("EmployeeObj"));      // line 21

    // Create a Statement                                      line 23
    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                          // line 35
    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))");                                 // line 40

    // Create a SQLData object EmployeeObj in the SCOTT schema
    EmployeeObj e = new EmployeeObj("SCOTT.EMPLOYEE", "George Jones", 456);

    // Insert the SQLData object into the database         // line 45
    PreparedStatement pstmt
      = conn.prepareStatement ("INSERT INTO employee_table VALUES (?)");

    pstmt.setObject(1, e, OracleTypes.STRUCT);
    pstmt.executeQuery();
    System.out.println("insert done");
    pstmt.close();                                         // line 52

    // Select the contents of the employee_table           // line 54
    Statement s = conn.createStatement();
    OracleResultSet rs = (OracleResultSet)
      s.executeQuery("SELECT * FROM employee_table");      // line 57

    // print the contents of the table                              // line 59
    while(rs.next())
    {
       EmployeeObj ee = (EmployeeObj) rs.getObject(1);
       System.out.println("EmpName: " + ee.empName + " EmpNo: " + ee.empNo);
    }                                                      // line 64

    // close the result set, statement, and connection     // line 66
    rs.close();
    s.close();

    if (conn != null)
    {
      conn.close();                                        // line 72
    }
   }
}
```

Lines 1-16: 必要な java.* および oracle.* パッケージをインポートします。DriverManager.registerDriver() メソッドを使ってドライバを登録し、getConnection() メソッドを使ってデータベースに接続します。データベース URL jdbc:oracle:oci8:@ を使用し、ユーザー scott、パスワード tiger で接続します。オプションで、@ 記号の後にデータベース名を入力することもできます。

Lines 18-21: getTypeMap() メソッドを使って、この接続と関連付けられたタイプ名を取得します。マップ・オブジェクトの put() メソッドを使って、SQL EMPLOYEE オブジェクトのマッピングを EmployeeObj カスタム Java 型に追加します。

Lines 23-33: 文オブジェクトを作成し、既存の表および EMPLOYEE_TABLE 型と EMPLOYEE 型を削除します。

Lines 35-40: SQL 文を使って、次の操作を実行します。

- 従業員名属性と従業員番号属性を含む EMPLOYEE オブジェクトの作成
- 1 つの EMPLOYEE を持つ、従業員オブジェクトの表（EMPLOYEE_TABLE）の作成
- 初期データ値の表への挿入

Lines 42、43: EmployeeObj オブジェクト（SQLData オブジェクト）を新規作成します。スキーマ名（SCOTT）、SQL 型名（EMPLOYEE）、従業員名（George Jones）および従業員番号（456）を識別します。スキーマ名は、getConnection() コールでのユーザー名と同じであることに注意してください。ユーザー名を変更した場合は、スキーマ名も変更する必要があります。

Lines 45-52: 新しい EMPLOYEE オブジェクトを従業員表に挿入する文を準備します。setObject() メソッドは、オブジェクトが最初の索引位置に挿入されること、および EMPLOYEE オブジェクトが oracle.sql.STRUCT 型をベースとすることを示します。

Lines 54-57: EMPLOYEE_TABLE の内容を選択します。結果を OracleResultSet にキャストし、カスタム Java オブジェクトを取得可能にします。

Lines 59-62: 結果セットを反復して、EMPLOYEE オブジェクトの内容を取得し、従業員名と従業員番号を出力します。

Lines 66-72: 結果セット、文および接続オブジェクトをクローズします。

CustomDatum のサンプル

この項では、CustomDatum および CustomDatumFactory インタフェースをインプリメントする、ユーザー作成の Java クラスについて説明します。CustomDatum 型のカスタム Java クラスには、CustomDatumFactory オブジェクトを返す静的な getFactory() メソッドがあります。JDBC ドライバは、CustomDatumFactory オブジェクトの create() メソッドを使って、CustomDatum インスタンスを返します。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って、CustomDatum および CustomDatumFactory インタフェースをインプリメントするクラス定義を生成します。

次のサンプル・プログラムは、EMPLOYEE オブジェクトの SQL クラス定義が存在する場合の、ユーザーによる記述が可能な Java クラス定義を示します。

EMPLOYEE オブジェクトの SQL 定義

次の SQL コードは、EMPLOYEE オブジェクトを定義します。EMPLOYEE オブジェクトは、従業員名（EmpName）および従業員番号（EmpNo）で構成されます。

```
create type EMPLOYEE as object
     (
          EmpName VARCHAR2(50),
          EmpNo   INTEGER
     );
```

カスタム Java オブジェクト用の Java クラス定義

次に、Employee.java ファイルの内容を示します。

```
import java.math.BigDecimal;
import java.sql.SQLException;
import oracle.jdbc.driver.OracleConnection;
import oracle.sql.CustomDatum;
import oracle.sql.CustomDatumFactory;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.sql.StructDescriptor;

public class Employee implements CustomDatum, CustomDatumFactory // line 10
{

  static final Employee _employeeFactory = new Employee(null, null); //line 13

  public static CustomDatumFactory getFactory()
  {
    return _employeeFactory;
  }                                                          // line 18

  /* constructor */                                          // line 20
```

```
  public Employee(String empName, BigDecimal empNo)
  {
    this.empName = empName;
    this.empNo = empNo;
  }                                                          // line 25

  /* CustomDatum interface */                                // line 27
  public Datum toDatum(OracleConnection c) throws SQLException
  {
    StructDescriptor sd =
       StructDescriptor.createDescriptor("SCOTT.EMPLOYEE", c);

    Object [] attributes = { empName, empNo };

    return new STRUCT(sd, c, attributes);
  }                                                          // line 36

  /* CustomDatumFactory interface */                         // line 38
  public CustomDatum create(Datum d, int sqlType) throws SQLException
  {
    if (d == null) return null;

    System.out.println(d);

    Object [] attributes = ((STRUCT) d).getAttributes();

    return new Employee((String) attributes[0],
                        (BigDecimal) attributes[1]);
  }                                                          // line 49

  /* fields */
  public String empName;
  public BigDecimal empNo;
}
```

Line 10: 必要に応じ、Employee クラスは CustomDatum および CustomDatumFactory インタフェースをインプリメントします。

Lines 13-18: JPublisher は、Employee クラスの _employeeFactory オブジェクトを定義します。このオブジェクトは、getFactory() メソッドにより返され、新しい Employee オブジェクトの作成に使用されます。getFactory() メソッドは、空の Employee オブジェクトを返します。このオブジェクトは、複数の Employee オブジェクトを新規作成する際に使用できます。

Lines 20-25: JPublisher は、SQL EMPLOYEE オブジェクトに対応する Employee Java クラスを定義します。JPublisher は、java.lang.String 型の従業員名および

java.math.BigDecimal 型の従業員番号という 2 つの属性を持つ Employee クラスを作成します。

Lines 27-36: CustomDatum インタフェースの toDatum() メソッドは、EMPLOYEE SQL データを oracle.sql.* 表現に変換します。このために、toDatum() は次の 2 つを使用します。

- スキーマ名、SQL オブジェクトまたは型名、および接続オブジェクトを引数として取る、STRUCT 記述子
- オブジェクトの従業員名および従業員番号属性の値を格納する、オブジェクト配列

toDatum() は、STRUCT 記述子、接続オブジェクトおよびオブジェクト属性を含む STRUCT を oracle.sql.Datum に返します。

Lines 38-49: CustomDatumFactory インタフェースは、使用する Employee カスタム Java クラスのコンストラクタに類似した create() メソッドを指定します。create() メソッドは、Datum オブジェクトおよび Datum オブジェクトの SQL タイプ・コードを取り、CustomDatum インスタンスを返します。

定義によると、Datum オブジェクトの値が null の場合、create() メソッドは null を返します。それ以外の場合は、Employee オブジェクトのインスタンスを従業員名および従業員番号属性と共に返します。

カスタム Java クラスの使用例

ここでは、JPublisher を使って作成した Employee クラスの簡単な使用例を示します。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 データを取得します。

いくつかのコメントの変更を除き、次のサンプルは、Demo/samples/oci8/object-samples ディレクトリの CustomDatumExample.java プログラムと同様です。

```
import java.sql.*;                                          // line 1
import oracle.jdbc.driver.*;
import oracle.sql.*;
import java.math.BigDecimal;


public class CustomDatumExample
{
  public static void main(String args []) throws Exception
  {

    // Connect
    DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver ());
    OracleConnection conn = (OracleConnection)
```

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

// Create a Statement                                    // line 18
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.
}                                                        // line 28

// Create and populate tables                            // line 30
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))");          // line 35

// Create a CustomDatum object                           // line 37
Employee e = new Employee("George Jones", new BigDecimal("456"));

// Insert the CustomDatum object                         // line 40
PreparedStatement pstmt
  = conn.prepareStatement ("INSERT INTO employee_table VALUES (?)");

pstmt.setObject(1, e, OracleTypes.STRUCT);
pstmt.executeQuery();
System.out.println("insert done");
pstmt.close();                                           // line 47

// Select now                                            // line 49
Statement s = conn.createStatement();
OracleResultSet rs = (OracleResultSet)
  s.executeQuery("SELECT * FROM employee_table");

while(rs.next())                                         // line 54
{
   Employee ee = (Employee) rs.getCustomDatum(1, Employee.getFactory());
   System.out.println("EmpName: " + ee.empName + " EmpNo: " + ee.empNo);
}                                                        // line 58
rs.close();
s.close();
```

```
    if (conn != null)
    {
      conn.close();
    }
  }
}
```

Lines 1-16: 必要な java.* および oracle.* パッケージをインポートします。DriverManager.registerDriver() メソッドを使ってドライバを登録し、getConnection() メソッドを使ってデータベースに接続します。データベース URL jdbc:oracle:oci8:@ を使用し、ユーザー system、パスワード manager で接続します。オプションで、@ 記号の後にデータベース名を入力することもできます。

Lines 18-28: 文オブジェクトを作成し、既存の表および EMPLOYEE_TABLE 型と EMPLOYEE 型を削除します。

Lines 30-35: SQL 文を使って、次の操作を実行します。

- 従業員名属性と従業員番号属性を含む Employee オブジェクトの作成
- 1 つの EMPLOYEE 列を持つ、従業員オブジェクト表の作成
- 初期データ値の表への挿入

Lines 37、38: Employee オブジェクト（CustomDatum オブジェクト）の新規作成および従業員名と従業員番号の定義

Lines 40-47: 新しい Employee オブジェクトをデータベースに挿入する文を準備します。setObject() メソッドは、オブジェクトが最初の索引位置に挿入されること、および Employee オブジェクトが oracle.sql.STRUCT 型をベースとすることを示します。

Lines 49-54: employee_table の内容を選択します。結果を OracleResultSet にキャストして、結果に対して getCustomDatum() メソッドを使用可能にします。

Lines 54-58: 結果セットを反復して、Employee オブジェクトの内容を取得し、従業員名と従業員番号を出力します。

Lines 58-62: 結果セット、文および接続オブジェクトをクローズします。

署名付きアプレットの作成

この項では、JDBC Thin ドライバを使ってデータベースへの接続と問合せを行う、署名付きアプレットの例を示します。アプレットで使用されたコードは、Oracle JDeveloper で作成されたもので、JDK 1.1.2 および JDBC 1.22 に準拠しています。署名付きのアプレットは、ブラウザ固有でもあります。この項で定義したアプレットは、Netscape 4.*x* ブラウザで動作します。

アプレットの提供するユーザー・インタフェース上で、「Local」または「Remote」のいずれかのボタンをクリックすると、ローカルまたはリモートのデータベースに接続できます。アプレットは、選択されたデータベースに対して指定行の内容の問合せを行い、結果を表示します。

使用するシステムでこのプログラムを実行するには、次の情報を提供する必要があります。

- Netscape から Capabilities クラスのコピーを取得します。またオブジェクト署名証明書も取得します。詳細は、次の Web サイトを参照してください。

```
http://developer.netscape.com/docs/manuals/signedobj/capabilities/contents.htm
```

  指示に従って、証明書の取得およびクラスのダウンロードを行います。この項の例では、Capabilities クラスとして `Principle.class`、`Privilege.class`、`PrivilegeManager.class` および `PrivilegeTable.class` が必要です。

アプレット・コード内で、次の文字列を置換します。

- *<local database connect string>* をローカル・データベース用の接続文字列と置換します。

  例 :

```
"jdbc:oracle:thin:@myServer.us.oracle.com:1521:orcl", "scott","tiger"
```

- *<select on row of local table>* を、ローカル・データベースの表のある行に格納された SQL `SELECT` 文と置換します。

  例 :

```
SELECT * FROM EMP WHERE ENAME = 'Mary'
```

- *<remote database connect string>* をリモート・データベース用の接続文字列と置換します。

  例 :

```
"jdbc:oracle:thin:@yourServer.us.oracle.com:1521:orcl", "scott","tiger"
```

- *<select on row of remote table>* を、リモート・データベースの表に格納された SQL `SELECT` 文と置換します。

  例 :

```
SELECT * FROM EMP WHERE ENAME = 'Bob'
```

このアプレットは、Java AWT コンポーネントと JDBC だけを使用します。

```
// Title:      JDBC Test Applet                                  // line 1
// Description:Sample JDK 1.1 Applet using the
// ORACLE JDBC Thin Driver
package JDBCApplet;

import java.awt.*;                                               // line 6
import java.awt.event.*;
import java.applet.*;
import java.sql.*;
import borland.jbcl.control.*;
import netscape.security.*;
                                                                 // line 12
public class MainApplet extends Applet {
  boolean isStandalone = false;
  BorderLayout borderLayout1 = new BorderLayout();
  Panel panel1 = new Panel();
  Label labelTitle = new Label();
  Panel panel2 = new Panel();
  BorderLayout borderLayout2 = new BorderLayout();
  TextArea txtArResults = new TextArea();
  Button button1 = new Button();
  BorderLayout borderLayout3 = new BorderLayout();
  Panel panel3 = new Panel();
  BorderLayout borderLayout4 = new BorderLayout();
  Label statusBar1 = new Label();
  Button button2 = new Button();

  // Get a parameter value                                       // line 28
  public String getParameter(String key, String def) {
    return isStandalone ? System.getProperty(key, def) :
      (getParameter(key) != null ? getParameter(key) : def);
  }                                                              // line 32

  // Construct the applet
  public MainApplet() {
  }

  // Initialize the applet                                          line 37
  public void init() {
    try { jbInit(); } catch (Exception e) { e.printStackTrace(); }
    try {
        PrivilegeManager.enablePrivilege("UniversalConnect");
        PrivilegeManager.enablePrivilege("UniversalListen");
        PrivilegeManager.enablePrivilege("UniversalAccept");
```

```
  } catch (Exception e) {
    e.printStackTrace();
  }
}

// Component initialization                                    line 49
public void jbInit() throws Exception{
  this.setBounds(new Rectangle(0, 0, 400, 400));
  panel1.setBackground(Color.lightGray);
  panel1.setLayout(borderLayout3);
  this.setSize(new Dimension(372, 373));
  labelTitle.setBackground(Color.lightGray);
  labelTitle.setFont(new Font("Dialog", 0, 12));
  labelTitle.setAlignment(1);
  labelTitle.setText("Oracle Thin JDBC Driver  Sample Applet");
  button1.setLabel("Local");
  panel3.setBackground(Color.lightGray);
  statusBar1.setBackground(Color.lightGray);
  statusBar1.setText("Ready");
  button2.setLabel("Remote");
  button2.addActionListener(new MainApplet_button2_actionAdapter(this));
  panel3.setLayout(borderLayout4);
  button1.addActionListener(new MainApplet_button1_actionAdapter(this));
  panel2.setLayout(borderLayout2);
  this.setLayout(borderLayout1);
  this.add(panel1, BorderLayout.NORTH);
  panel1.add(button1, BorderLayout.WEST);
  panel1.add(labelTitle, BorderLayout.CENTER);
  panel1.add(button2, BorderLayout.EAST);
  this.add(panel2, BorderLayout.CENTER);
  panel2.add(txtArResults, BorderLayout.CENTER);
  this.add(panel3, BorderLayout.SOUTH);
  panel3.add(statusBar1, BorderLayout.NORTH);
}

//Start the applet                                             line 79
public void start() {
}

//Stop the applet
public void stop() {
}

//Destroy the applet
public void destroy() {
}
```

```
  //Get Applet information
  public String getAppletInfo() {
    return "Applet Information";
  }

  //Get parameter info
  public String[][] getParameterInfo() {
    return null;
  }

  //Main method
  static public void main(String[] args) {
    MainApplet applet = new MainApplet();
    applet.isStandalone = true;
    Frame frame = new Frame();
    frame.setTitle("Applet Frame");
    frame.add(applet, BorderLayout.CENTER);
    applet.init();
    applet.start();
    frame.pack();
    Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
    frame.setLocation((d.width - frame.getSize().width) / 2, (d.height -
frame.getSize().height) / 2);
    frame.setVisible(true);
  }

  void button1_actionPerformed(ActionEvent e) {
    //
    // Handler for "Local" Button.
    //
    // Here is where we connect to local database            line 121

    StringBuffer b = new StringBuffer ();

    try {
        DriverManager.registerDriver ( new oracle.jdbc.driver.OracleDriver ());
        b.append ("DriverManager.registerDriver\r\n");
    } catch (SQLException oe) {
        statusBar1.setText("registerDriver: Caught SQLException");
    } catch (ClassNotFoundException oe) {
        statusBar1.setText("registerDriver: Caught ClassNotFoundException");
    }

    int numRows = 0;
    try {
        statusBar1.setText("Executing Query on Local Database ...");
        Connection conn = DriverManager.getConnection (
```

```
          "jdbc:oracle:thin:<local database connect string>");

        b.append ("[DriverManager.getConnection] \r\n");
        Statement stmt = conn.createStatement ();
        b.append ("[conn.createStatement] \r\n");
        ResultSet rset = stmt.executeQuery ("<select on row of
                                              local table>");
        b.append ("[stmt.executeQuery] \r\n");
        b.append("SQL> <select on row of local table>\r\n\n");
        b.append("DSCr\n-------------------------------------\r\n");

        while (rset.next ()) {
              String ename = rset.getString (1);
              b.append (ename);
              b.append ("\r\n");
              numRows++;
        } // [end while rset.next() loop]
        statusBar1.setText("Query Done.");
    } catch (SQLException SQLE) {
        statusBar1.setText ("Caught SQLException!");
        SQLE.printStackTrace();
    } finally {
        b.append("\r\n");
        b.append(String.valueOf(numRows) + " rows selected.\r\n");
        txtArResults.setText( b.toString ());
    }

    // End JDBC Code                                          line 165
}

void button2_actionPerformed(ActionEvent e) {
    //
    // Handler for the "Remote" Button                       line 170
    //
    StringBuffer b = new StringBuffer ();

    try {
        DriverManager.registerDriver ( new oracle.jdbc.driver.OracleDriver ());
        b.append ("DriverManager.registerDriver\r\n");
    } catch (SQLException oe) {
        statusBar1.setText("registerDriver: Caught SQLException");
    } catch (ClassNotFoundException oe) {
        statusBar1.setText("registerDriver: Caught ClassNotFoundException");
    }

    int numRows = 0;                                       // line 183
    try {
```

```
        statusBar1.setText("Executing Query on Remote Database ...");
        try {
                PrivilegeManager.enablePrivilege("UniversalConnect");
                b.append ("enablePrivilege(UniversalConnect)\r\n");
                PrivilegeManager.enablePrivilege("UniversalListen");
                b.append ("enablePrivilege(UniversalListen)\r\n");
                PrivilegeManager.enablePrivilege("UniversalAccept");
                b.append ("enablePrivilege(UniversalAccept)\r\n");

                Connection conn = DriverManager.getConnection (
                "jdbc:oracle:thin:<remote database connect string>"
                );
                b.append ("DriverManager.getConnection\r\n");

                Statement stmt = conn.createStatement ();
                b.append ("conn.createStatement\r\n");
                ResultSet rset = stmt.executeQuery ("<select on row
                                                of remote table>");
                b.append ("stmt.executeQuery\r\n");
                b.append("SQL> <select on row of remote table>\r\n\n");
                b.append("ENAME\r\n----------\r\n");

                while (rset.next ()) {
                      String ename = rset.getString (1);
                      b.append (ename);
                      b.append ("\r\n");
                      numRows++;
                } // [end while rset.next() loop]
                statusBar1.setText("Query Done.");
        } catch (Exception oe) {
                oe.printStackTrace();
        }
    } catch (SQLException SQLE) {
        statusBar1.setText("Caught SQLException!");
        SQLE.printStackTrace();
    } finally {
        b.append("\r\n");
        b.append(String.valueOf(numRows) + " rows selected.\r\n");
        txtArResults.setText( b.toString ());
    }

    // End JDBC Code for Button2                              line 256

  }
}
                                                          // line 260
class MainApplet_button1_actionAdapter implements java.awt.event.ActionListener {
```

```
  MainApplet adaptee;

  MainApplet_button1_actionAdapter(MainApplet adaptee) {
    this.adaptee = adaptee;
  }

  public void actionPerformed(ActionEvent e) {
    adaptee.button1_actionPerformed(e);
  }
}
                                                          // line 273
class MainApplet_button2_actionAdapter implements java.awt.event.ActionListener {
  MainApplet adaptee;

  MainApplet_button2_actionAdapter(MainApplet adaptee) {
    this.adaptee = adaptee;
  }

  public void actionPerformed(ActionEvent e) {
    adaptee.button2_actionPerformed(e);
  }
}
```

Lines 6-11: 必要なファイルをインポートします。

Lines 13-26: 2 つのボタンと出力表示用のテキスト領域を含む、GUI 用のグラフィックを設定します。

Lines 37-48: アプレットのダウンロード元以外のホストへの接続権限を要求します。

Lines 49-77: アプレットのコンポーネントを初期化します。これらのコンポーネントには、GUI の形式やレイアウトおよび GUI ボタンやテキスト領域が含まれます。

Lines 121-165: ローカル・データベースに接続します。このために、`DriverManager.registerDriver()` メソッドを使ってドライバを登録し、`DriverManager.getConnection()` を使ってデータベースに接続します。サーバー URL、ポート番号、SID、ユーザー名およびパスワードを使って接続します。

Lines 170-183: リモート・データベースに接続します。

Lines 183-256: アプレットがリモート・データベースに対する権限を保持しているかどうかをテストします。権限を保持している場合は、データベースに接続して SQL 文を実行します。

Lines 260-283: ボタン用のイベントとコールバックを設定します。

JDBC サンプル・コードと SQLJ サンプル・コード

この項では、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、SELECT 文を使って表の中に選択および格納し、従業員番号に基づいて従業員の住所を返します。updateAddress() は、住所を取得してストアド・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返します。

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 SQL スクリプト（後で説明）の実行により、データベース内のスキーマ作成および表の移入がすでに行われています。
- 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。

どちらのバージョンのサンプル・コードも、ObjectDemo.sql により生成されたオブジェクトおよび表を参照します。

注意 : 掲載されたサンプルコードは、JDBC と SQLJ のどちらのバージョンとも、コード全体の一部です。これらのコードだけを独立して実行することはできません。

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な表（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得し、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.driver.*;

/**
  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() メソッドを使って設定します。準備済みの文は 7-37 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。準備済みの文はカスタム・オブジェクトを返すため、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() メソッドを使ってカスタム・オブジェクトを取得します。（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は、ファクトリ・メソッド Address.getFactory() を使って Address オブジェクトのインスタンスを実体化する必要があります。getCustomDatum() は Datum を返すため、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。コメント行 TO DO は、結果セットに行が含まれていないか、2 行以上含まれている場合のために、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: 結果セットと準備済みの文オブジェクトをクローズして、addr 変数を返します。

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）を取り、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・オブジェクトを返すため、getCustomDatum() メソッドを使ってカスタム・オブジェクトを取得します。（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は、ファクトリ・メソッド Address.getFactory() を使って Address オブジェクトのインスタンスを実体化する必要があります。getCustomDatum() は Datum を返すため、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 変数を返します。

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（たとえば、結果セットや文オブジェクトをクローズすることにより）必要があります。
- 必要に応じ、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準備済みの文のパラメータ・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得し、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得します。従業員番号が getEmployeeAddress() に渡された従業員番号（empno）に一致したら、標準 SQLJ SELECT INTO 構文を使って、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 オブジェクトを返します。

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・コール構文を使って、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 オブジェクトを返します。

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・コードに関する、次のコーディング要件に留意してください。

- 明示的な接続は不要です。SQLJ は、デフォルト接続コンテキストがアプリケーションですでに定義されているものとして動作します。

- 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（たとえば、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。
- アウトパラメータを登録する必要はありません。
- SQLJ 構文は簡単です。たとえば、SELECT...INTO 形式の構文がサポートされており、OBDC 形式のエスケープを使用する必要はありません。

8

リファレンス情報

この章では、次の項目を含む JDBC リファレンス詳細情報について説明します。

- 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
- NLS キャラクタ・セットのサポート
- 関連情報

有効な SQL-JDBC データ型マッピング

第 3 章の表 3-1 および表 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一覧です。表 3-1 と表 3-2 の**標準 Java データ型、Java ネイティブ・データ型**および **Oracle SQL データ型**列の内容と、下の表 8-1 の内容を比較してください。

表 8-1 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 クラスの一覧です。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` として実現するには、`getCHAR()` を使います。これを `java.math.BigDecimal` として実現するには、`getBigDecimal()` を使います。

表 8-1 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 クラス
CHAR、NCHAR、VARCHAR2、	oracle.sql.CHAR
NVARCHAR2、LONG	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
DATE	oracle.sql.DATE
	java.sql.Date
	java.sql.Time
	java.sql.Timestamp
	java.lang.String
NUMBER	oracle.sql.NUMBER

表 8-1 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 クラス
	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
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
	oracle.jdbc2.Blob
CLOB、NCLOB	oracle.sql.CLOB
	oracle.jdbc2.Clob
OBJECT	oracle.sql.STRUCT
	oracle.SqljData
	oracle.jdbc2.Struct
REF	oracle.sql.REF
	oracle.jdbc2.Ref
TABLE（ネストした表）、VARRAY	oracle.sql.ARRAY
	oracle.jdbc2.Array
上記の任意の SQL データ型	oracle.sql.CustomDatum または oracle.sql.Datum

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される値は、SQL 型に対応している Java 型でなければなりません。同様に、oracle.sql.Datum 型の戻り値を持つメソッドによって返された値は、SQL 型に対応している Java 型でなければなりません。
- 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一覧です。表 8-2 は、SQL データ型に対する Oracle JDBC ドライバおよび SQLJ のサポート状況の一覧です。

表 8-2 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	あり	あり
BLOB	あり	あり
CHAR	あり	あり
CLOB	あり	あり
DATE	あり	あり
NCHAR	なし	なし
NCHAR VARYING	なし	なし
NUMBER	あり	あり
NVARCHAR2	なし	なし
RAW	あり	あり
REF	あり	あり
ROWID	あり	あり
UROWID	なし	なし
VARCHAR2	あり	あり

表 8-3 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 のサポート状況の一覧です。

表 8-3 ANSI でサポートされている SQL データ型に対するサポート

ANSI が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	あり	あり
DEC	あり	あり
DECIMAL	あり	あり
DOUBLE PRECISION	あり	あり
FLOAT	あり	あり
INT	あり	あり
INTEGER	あり	あり
NATIONAL CHARACTER	なし	なし
NATIONAL CHARACTER VARYING	なし	なし
NATIONAL CHAR	なし	なし
NATIONAL CHAR VARYING	なし	なし
NCHAR	なし	なし
NCHAR VARYING	なし	なし
NUMERIC	あり	あり
REAL	あり	あり
SMALLINT	あり	あり
VARCHAR	あり	あり

表 8-4 は、PL/SQL データ型に対する Oracle JDBC ドライバおよび SQLJ のサポート状況の一覧です。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。）
- 複合型
- 参照型
- LOB 型

表 8-4 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	あり	あり
DEC	あり	あり
DECIMAL	あり	あり
DOUBLE PRECISION	あり	あり
FLOAT	あり	あり
INT	あり	あり
INTEGER	あり	あり
NATURAL	あり	あり
NATURAL*N*	なし	なし
NUMBER	あり	あり
NUMERIC	あり	あり
PLS_INTEGER	あり	あり
POSITIVE	あり	あり
POSITIVE*N*	なし	なし
REAL	あり	あり
SIGNTYPE	あり	あり
SMALLINT	あり	あり
スカラー文字列型		
CHAR	あり	あり
CHARACTER	あり	あり
LONG	あり	あり
LONG RAW	あり	あり
NCHAR	なし	なし
NVARCHAR2	なし	なし
RAW	あり	あり
ROWID	あり	あり
STRING	あり	あり

表 8-4 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
UROWID	なし	なし
VARCHAR	あり	あり
VARCHAR2	あり	あり
BOOLEAN	あり	あり
DATE	あり	あり
複合型		
RECORD	なし	なし
TABLE	なし	なし
VARRAY	あり	あり
参照型		
REF CURSOR	あり	あり
REF オブジェクト型	あり	あり
LOB 型		
BFILE	あり	あり
BLOB	あり	あり
CLOB	あり	あり
NCLOB	なし	なし

注意 :

- NATURAL、NATURAL*N*、POSITIVE、POSITIVE*N*、および SIGNTYPE は、BINARY INTEGER のサブタイプです。
- DEC、DECIMAL、DOUBLE PRECISION、FLOAT、INT、INTEGER、NUMERIC、REAL、および SMALLINT は、NUMBER のサブタイプです。

NLS キャラクタ・セットのサポート

クライアントでは、Oracle JDBC OCI ドライバおよび Thin ドライバによって、すべての Oracle NLS キャラクタ・セットがサポートされます。サーバーでは、Oracle JDBC Server ドライバにより、2 つの Oracle NLS キャラクタ・セット、US7ASCII（ASCII 7-bit American）および WE8ISO8859P1（ISO 8859-1 West European または "ISO-Latin 1"）だけがサポートされます。

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 サイトの一覧を示します。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。この一覧には、Oracle JDBC ドライバと SQLJ、Java テクノロジ、Java Developer's Kit APIs（for versions 1.2 and 2.0）のリファレンス、およびアプレット作成に役立つリソースがあります。

Java テクノロジ

Java Technology Home Page （Sun Microsystems, Inc.）:

http://www.javasoft.com/

Java Development Kit 1.1 （JDK1.1）（Sun Microsystems, Inc.）:

http://java.sun.com/products/jdk/1.1/

Java Platform JDK1.1 Core API Specification （Sun Microsystems, Inc.）:

http://www.javasoft.com/products/jdk/1.1/docs/api/packages.html

Java Development Kit 1.2 （JDK1.2）（Sun Microsystems, Inc.）:

http://java.sun.com:80/products/jdk/1.2/index.html

Java Platform JDK1.2 Core API Specification （Sun Microsystems, Inc.）:

http://www.javasoft.com/products/jdk/1.2/docs/api/index.html

署名付きアプレット

Introduction to Capabilities Classes （Netscape Communications Corp.）：

http://developer.netscape.com/docs/manuals/signedobj/capabilities/contents.htm

Object-Signing Resources （Netscape Communications Corp.）：

http://developer.netscape.com/software/signedobj/index.html

Signed Applet Example （Sun Microsystems, Inc.）：

http://java.sun.com/security/signExample/index.html

A

JDBC エラー・メッセージ

この付録では、Oracle JDBC ドライバが戻せるエラー・メッセージをリストします。
各メッセージの " 原因 " および " 処置 " に関する情報は、将来のリリースで提供されます。

byte array not long enough

can only describe a query

cannot do new defines until the current result set is closed

cannot set row prefetch to zero

char array not long enough

character set not supported

closed connection

closed LOB

closed resultset

closed statement

cursor already initialized

error in defines.isNull ()

error in type descriptor parse

exception in OracleNumber

exhausted resultset

fail to construct descriptor

fail to convert between UTF8 and UCS2

fail to convert to internal representation

inconsistent Java and SQL object types

Internal error: attempt to access bind values beyond the batch value

Internal error: data array not allocated

Internal error: invalid index for data access

Internal error: invalid NLS Conversion ratio

invalid batch value

invalid character encountered in

invalid column name

invalid column type

invalid cursor

invalid dynamic column

invalid row prefetch

invalid stream maximum size

malformed SQL92 string at position

missing defines

missing defines at index

missing descriptor

missing IN or OUT parameter at index:

no data read

no such element in vector

non supported character set

non-supported SQL92 token at position

numeric overflow

only one RPA message is expected

only one RXH message is expected

parameter type conflict

protocol violation

received more RXDs than expected

REF cursor is invalid

resultSet.next was not called

setAutoClose: only support auto close mode on

setReadOnly: read-only connections not supported

setTransactionIsolation: only supports TRANSACTION_READ_UNCOMMITTED

statement timed out

statement was cancelled

stream has already been closed

sub protocol must be specified in connection URL

the LOB locator is not valid

the size is not valid

This API cannot be be used for non-UDT types

this ref is not valid

undefined type

unsupported column type

unsupported feature

索引

A

B

C

D

E

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

あ

い

え

お

か

て

と

な

ね

は

ふ

ま

め

も

ゆ

れ

ろ