BEA WebLogic Server™ および WebLogic Express®

WebLogic jDriver for Oracle のコンフィグレーションと使い方

BEA WebLogic Server バージョン 7.0
マニュアルの改訂 : 2004 年 6 月 6 日

著作権

限定的権利条項

商標または登録商標

WebLogic jDriver for Oracle のコンフィグレーションと使い方

パート番号	マニュアルの日付	ソフトウェアのバージョン
なし	2003 年 1 月 17 日	BEA WebLogic Server バージョン 7.0

目次

このマニュアルの内容

1. 概要

2. WebLogic jDriver for Oracle のコンフィグレーション

3. WebLogic jDriver for Oracle の使い方

4. 分散トランザクションでの WebLogic jDriver for

Oracle/XA の使い方

5. Oracle の高度な機能

このマニュアルの内容

このマニュアルでは、BEA の Oracle データベース管理システム用 Type 2 Java Database Connectivity (JDBC) ドライバである WebLogic jDriver for Oracle のインストール方法と、このドライバを使用してローカルおよび分散トランザクションに対応するアプリケーションを開発する方法について説明します。

このマニュアルの内容は以下のとおりです。

- 第 1 章「概要」
- 第 2 章「WebLogic jDriver for Oracle のコンフィグレーション」
- 第 3 章「WebLogic jDriver for Oracle の使い方」
- 第 4 章「分散トランザクションでの WebLogic jDriver for Oracle/XA の使い方」
- 第 5 章「Oracle の高度な機能」

対象読者

このマニュアルは、Sun Microsystems の Java 2 Platform、Enterprise Edition (J2EE) を使用して e- コマース アプリケーションを構築するアプリケーション開発者向けのマニュアルです。SQL、データベースの一般的な概念、および Java プログラミングに読者が精通していることを前提として書かれています。

e-docs Web サイト

BEA 製品のドキュメントは、BEA の Web サイトで入手できます。BEA のホーム ページで [製品のドキュメント] をクリックするか、または WebLogic Server 製品ドキュメント ページ (http://edocs.beasys.co.jp/e-docs/index.html) を直接表示してください。

このマニュアルの印刷方法

Web ブラウザの [ファイル | 印刷] オプションを使用すると、Web ブラウザからこのマニュアルを一度に 1 章ずつ印刷できます。

このマニュアルの PDF 版は、WebLogic Server の Web サイトで入手できます。PDF を Adobe Acrobat Reader で開くと、マニュアルの全体 (または一部分) を書籍の形式で印刷できます。PDF を表示するには、WebLogic Server ドキュメントのホーム ページを開き、[ドキュメントのダウンロード] をクリックして、印刷するマニュアルを選択します。

Adobe Acrobat Reader は Adobe の Web サイト (http://www.adobe.co.jp) で無料で入手できます。

関連情報

BEA の Web サイトでは、WebLogic Server の全マニュアルを提供しています。

サポート情報

BEA のドキュメントに関するユーザからのフィードバックは弊社にとって非常に重要です。質問や意見などがあれば、電子メールで docsupport-jp@beasys.com までお送りください。寄せられた意見については、WebLogic Server のドキュメントを作成および改訂する BEA の専門の担当者が直に目を通します。

電子メールのメッセージには、ご使用のソフトウェアの名前とバージョン、およびドキュメントのタイトルと日付をお書き添えください。本バージョンの BEA WebLogic Server について不明な点がある場合、または BEA WebLogic Server のインストールおよび動作に問題がある場合は、BEA WebSupport (www.bea.com) を通じて BEA カスタマ サポートまでお問い合わせください。カスタマ サポートへの連絡方法については、製品パッケージに同梱されているカスタマ サポート カードにも記載されています。

カスタマ サポートでは以下の情報をお尋ねしますので、お問い合わせの際はあらかじめご用意ください。

- お名前、電子メール アドレス、電話番号、ファクス番号
- 会社の名前と住所
- お使いの機種とコード番号
- 製品の名前とバージョン
- 問題の状況と表示されるエラー メッセージの内容

表記規則

このマニュアルでは、全体を通して以下の表記規則が使用されています。

表記法	適用
〔Ctrl〕+ 〔Tab〕	複数のキーを同時に押すことを示す。
斜体	強調または書籍のタイトルを示す。

等幅テキスト	コード サンプル、コマンドとそのオプション、データ構造体とそのメンバー、データ型、ディレクトリ、およびファイル名とその拡張子を示す。等幅テキストはキーボードから入力するテキストも示す。
	例:
	`import java.util.Enumeration;`
	`chmod u+w *`
	`config/examples/applications`
	`.java`
	`config.xml`
	`float`

斜体の等幅テキスト	コード内の変数を示す。 例: `String *CustomerName*;`
すべて大文字のテキスト	デバイス名、環境変数、および論理演算子を示す。 例: LPT1 BEA_HOME OR
{ }	構文の中で複数の選択肢を示す。

表記法	適用
[ ]	構文の中で任意指定の項目を示す。 例： `java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]`
|	構文の中で相互に排他的な選択肢を区切る。 例： `java weblogic.deploy [list|deploy|undeploy|update] password {application} {source}`
...	コマンドラインで以下のいずれかを示す。 ■ 引数を複数回繰り返すことができる。 ■ 任意指定の引数が省略されている。 ■ パラメータや値などの情報を追加入力できる。
. . .	コード サンプルまたは構文で項目が省略されていることを示す。

1　概要

このマニュアルでは、BEA の Oracle データベース管理システム (DBMS) 用 JDBC ドライバをコンフィグレーションして、WebLogic Server で使用する方法について説明します。また、*MultiPools* (マルチプール) 機能についても説明します。

このマニュアルでは、ユーザが Java、DBMS の全般的な概念、および Structured Query Language (SQL) について理解していることを前提にしています。

この章では次の内容について説明します。

- WebLogic jDriver の概要
- WebLogic jDriver for Oracle

WebLogic jDriver の概要

BEA は、WebLogic Server ソフトウェアで使用する以下の WebLogic jDriver を提供します。

- 分散トランザクション機能を含む Oracle 用 Type 2 ネイティブ JDBC ドライバ
- Microsoft SQL Server 用の Type 4 JDBC ドライバ

Type 2 ドライバはデータベース ベンダが提供するクライアント ライブラリを使用しますが、Type 4 ドライバは pure-Java であり、通信レベルでデータベース サーバに接続するので、ベンダ固有のクライアント ライブラリが不要です。

これらのドライバを使用して、接続プールに物理データベース接続を作成します。他のベンダの JDBC ドライバも使用できます。『WebLogic JDBC プログラマーズ ガイド』の「JDBC ドライバと WebLogic Server の使用」を参照してください。

WebLogic jDriver for Oracle

Oracle DBMS 用 Type 2 JDBC ドライバである WebLogic jDriver for Oracle は、WebLogic Server ソフトウェアに付属しています。このドライバを使用するには、必要なすべてのライブラリを含む Oracle クライアントを、Oracle DBMS のクライアントとなるマシン上に完全インストールしなければなりません。この Oracle クライアントのインストール内容には、WebLogic Server で必要とされるベンダ提供のクライアント ライブラリおよび関連ファイルが含まれていなければなりません。

注意： 同じバージョンの WebLogic jDriver for Oracle、Oracle クライアント、およびデータベース管理システムを使用する必要があります。たとえば、Oracle DBMS のバージョンが 8.1.7 である場合、バージョン 8.1.7 の Oracle クライアントおよび WebLogic jDriver for Oracle を使用する必要があります。

Oracle 共有ライブラリ

WebLogic Server 配布キットには、WebLogic Server 用に BEA が提供するネイティブ ライブラリが入っています。どのライブラリを選ぶかは、クライアント マシンにインストールされている Oracle クライアントのバージョンと、Oracle サーバにアクセスするために使用する Oracle API のバージョンによって決まります。このドライバを使用する前に、BEA のネイティブ ライブラリと Oracle のクライアント ライブラリの両方を、クライアントの `PATH` (Windows) または `shared library path` (UNIX) に入れる必要があります。詳細については、2-1 ページの「WebLogic jDriver for Oracle のコンフィグレーション」を参照してください。

WebLogic jDriver for Oracle/XA での分散トランザクション

WebLogic Server は、Oracle Corporation の Oracle8i および Oracle9i データベース管理システム用のマルチスレッド JDBC/XA ドライバを提供します。WebLogic jDriver for Oracle/XA は、WebLogic jDriver for Oracle のトランザクション対応バージョンです。WebLogic jDriver for Oracle/XA は、X/Open Distributed Transaction Processing (DTP) モデルのトランザクション マネージャとリソース マネージャとの双方向システムレベル インタフェースである XA を完全サポートします。

2 WebLogic jDriver for Oracle のコンフィグレーション

この章では次の内容について説明します。

- WebLogic jDriver for Oracle の使用準備
- WebLogic jDriver for Oracle の使用環境の設定
- Oracle データベースとの接続の確認
- 接続プールの設定
- WebLogic jDriver での IDE またはデバッガの使い方
- 開発環境を設定して WebLogic jDriver for Oracle を使用する準備

WebLogic jDriver for Oracle の使用準備

WebLogic jDriver for Oracle を使用する前に、以下の作業が必要です。

- WebLogic jDriver for Oracle のソフトウェア要件のチェック
- ライセンス機能のチェック
- WebLogic jDriver for Oracle の使用環境の設定

WebLogic jDriver for Oracle のソフトウェア要件のチェック

この節では、以下のソフトウェア要件について説明します。

- サポートされるプラットフォーム

- JDBC 2.0 の動作要件

サポートされるプラットフォーム

WebLogic jDriver がサポートするプラットフォーム、オペレーティング システム、JVM、DBMS バージョン、およびクライアント ライブラリの詳細については、「動作確認状況」を参照してください。

JDBC 2.0 の動作要件

WebLogic Server 7.0 は JDK 1.3.1 プラットフォーム環境で動作し、分散トランザクション機能を含む JDBC 2.0 API (JDBC 2.0 コア API と JDBC オプション パッケージ API) をサポートします。さらに、Oracle Call Interface のバージョン 8 API のドライバを使用する必要があります。

ライセンス機能のチェック

WebLogic jDriver for Oracle を使用するには、適切なライセンスが必要です。WebLogic jDriver for Oracle のライセンス機能は、この WebLogic Server をインストールした BEA ホーム ディレクトリ内のライセンス ファイルに含まれています。次に例を示します。

```
c:\bea\license.bea
```

WebLogic Server ライセンスのインストール時または最終更新時に WebLogic jDriver for Oracle がそのライセンスに含まれていた場合は、これ以上の作業は不要です。この機能を追加する場合は、BEA の販売代理店から更新ライセンスを入手する必要があります。ライセンス ファイルの更新方法については、WebLogic Server の『インストール ガイド』の「license.bea ファイルの更新」を参照してください。

注意： WebLogic Server が実行されていないときに WebLogic jDriver for Oracle を使用する場合は、`license.bea` のあるフォルダへのパスを `CLASSPATH` に入れる必要があります。

WebLogic jDriver for Oracle の使用環境の設定

WebLogic jDriver を使用する環境を設定するには、パス変数の設定に以下の情報を入れる必要があります。

- WebLogic jDriver for Oracle が入っているディレクトリ (ドライバ ファイルは、使用しているオペレーティング システムにより異なり、ネイティブ dll、so、または sl ファイルになります)。ドライバが入っているファイルは、WebLogic Server クライアントで使用できなければなりません。パス変数の名前は、使用しているシステムによって異なります。
  - Windows システムでは、PATH を設定します。
  - ほとんどの UNIX システムでは、LD_LIBRARY_PATH を設定します。
  - HP-UX システムでは、SHLIB_PATH を設定します。

  ドライバ ファイルが入ったディレクトリは、以下で説明する要素によって異なります。
- Oracle が提供するライブラリが入ったディレクトリ。Oracle クライアント ライブラリの入ったディレクトリの場所は、インストールによって異なります。Windows NT では、Oracle インストーラはこれらのライブラリをシステム パスに配置します。ディレクトリ名は、オペレーティング システムとそのアーキテクチャ (32 ビットまたは 64 ビット) に応じて異なります。

WebLogic Server は、Oracle Call Interface (OCI) バージョン 8 API で作成された dll ファイル、so ファイル、または sl ファイルを、Oracle DBMS にアクセスするためのネイティブ インタフェースとして使用します。

プラットフォームごとにまとめた以降の節の表は、Oracle クライアント バージョンに基づいて、システム PATH に指定する必要があるディレクトリの一覧です。

Windows NT

WebLogic 共有ライブラリ (.dll) ディレクトリのパス名と Oracle クライアントのインストール先ディレクトリを、次のように、PATH に追加します。

構文

次の構文で指定します。

- WL_HOME\server\bin\ と適切な WebLogic Server 共有ライブラリ ディレクトリを以下の表を参照して PATH に追加します。WL_HOME は、WebLogic Server のインストール ディレクトリを表します。次に例を示します。

  %WL_HOME%\server\bin\oci*xxxx*

  xxxx は Oracle 8.1.7 では 817_8、Oracle 9.0.1 では 901_8、Oracle 9.2.0 では 920_8 になります。

- ORACLE_HOME\bin を PATH に追加します。ORACLE_HOME は、Oracle クライアントのインストール先ディレクトリを表します。WebLogic jDriver for Oracle および Oracle ホーム情報は、常に PATH の**先頭**に追加します。次に例を示します。

  %ORACLE_HOME%\bin;%PATH%

例

上記の構文に従って Oracle 8.1.7 用の実際の例を作成すると、パスは次のようになります。

```
$set PATH=%WL_HOME%\server\bin\oci817_8;%ORACLE_HOME%\bin;%PATH%
```

次の表は、Windows 用のディレクトリと Oracle クライアントのバージョンを示します。

表 2-1 Windows NT 上の Oracle

Oracle クライアント バージョン	OCI API バージョン	共有ライブラリ (.dll) ディレクトリ	メモ
8.1.7	8	oci817_8	Oracle 8 と JDBC 2.0 コア API およびオプション パッケージ API (分散トランザクション機能を含む) にアクセス可能。
9.0.1	8	oci901_8	Oracle 9.0 と JDBC 2.0 コア API およびオプション パッケージ API (分散トランザクション機能を含む) にアクセス可能。
9.2.0	8	oci920_8	Oracle 9.2 と JDBC 2.0 コア API およびオプション パッケージ API (分散トランザクション機能を含む) にアクセス可能。

Solaris

Solaris 環境を設定して WebLogic jDriver をサポートするには、ネイティブ インタフェース ファイル (ドライバ ファイル) が入っているディレクトリと、Oracle クライアントをインストールしたディレクトリを、環境変数 **LD_LIBRARY_PATH** の設定に入れる必要があります。

構文

次の構文で指定します。

- ネイティブ インタフェース **libweblogicoci38.so** と **libweblogicoxa38.so** が入っているディレクトリ。次に例を示します。

  ```
  $WL_HOME/server/lib/solaris/ociXXXX
  ```

  xxxx は Oracle 8.1.7 では 817_8、Oracle 9.0.1 では 901_8、Oracle 9.2.0 では 920_8 になります。

- Oracle が提供するライブラリが入ったディレクトリ。Oracle クライアント ライブラリの入ったディレクトリの場所は、インストールによって異なります。次に例を示します。

  ```
  $ ORACLE_HOME/lib
  ```

例

上記の構文に従って Oracle 8.1.7 用の実際のパスを作成すると次のようになります。

```
export
LD_LIBRARY_PATH=$WL_HOME/server/lib/solaris/oci817_8:$ORACLE_HOME
/lib:$LD_LIBRARY_PATH
```

次の表は、Solaris 用のディレクトリと Oracle クライアントのバージョンを示します。

表 2-2 Solaris 上の Oracle

Oracle クライアント バージョン	OCI API バージョン	共有ライブラリ (.so) ディレクトリ	メモ
8.1.7	8	oci817_8	Oracle 8 と JDBC 2.0 コア API およびオプション パッケージ API (分散トランザクション機能を含む) にアクセス可能。
9.0.1	8	oci901_8	Oracle 9 と JDBC 2.0 コア API およびオプション パッケージ API (分散トランザクション機能を含む) にアクセス可能。

表 2-2 Solaris 上の Oracle

Oracle クライアント バージョン	OCI API バージョン	共有ライブラリ (.so) ディレクトリ	メモ
9.2.0	8	oci920_8	Oracle 9.2 と JDBC 2.0 コア API およびオプション パッケージ API (分散トランザクション機能を含む) にアクセス可能。

次の表に、Solaris の 32 ビットおよび 64 ビット インストールの、ベンダ提供の Oracle ライブラリがあるディレクトリを示します。

表 2-3 Solaris にインストールされた Oracle ライブラリへのパス

Oracle クライアント バージョン	アーキテクチャ	Oracle ライブラリへのパス
8.1.7	32 ビット	`ORACLE_HOME/lib`
8.1.7	64 ビット	`ORACLE_HOME/lib64`
9.0.1	32 ビット	`ORACLE_HOME/lib32`
9.0.1	64 ビット	`ORACLE_HOME/lib`
9.2.0	32 ビット	`ORACLE_HOME/lib32`
9.2.0	64 ビット	`ORACLE_HOME/lib`

IBM AIX

ご使用のプラットフォームがサポートされているかどうかを確認するには、「動作確認状況」を参照してください。

HP-UX 11

HP 環境を設定して WebLogic jDriver をサポートするには、ネイティブ インタフェース ファイル (ドライバ ファイル) が入っているディレクトリと、Oracle クライアントをインストールしたディレクトリを、環境変数 **SHLIB_PATH** の設定に入れる必要があります。

注意： HP-UX 用の Oracle 9 は、Oracle クライアントを含め 64 ビット バージョンでのみ利用できます。WebLogic jDriver for Oracle は Type-2 JDBC ドライバであるため、データベース アクセス用の Oracle クライアントが必要です。したがって、Oracle 9 で WebLogic jDriver for Oracle を使用するには、WebLogic Server を 64 ビット マシンで実行する必要があります。

構文

Oracle 8 では、次の構文を使用します。

- Oracle 8i 用のネイティブ インタフェース (ドライバ ファイル) **libweblogicoci38.sl** と **libweblogicoxa38.so** が入っているディレクトリ。次に例を示します。

  ```
  $WL_HOME/server/lib/hpux11/oci817_8
  ```

- Oracle が提供するライブラリが入ったディレクトリ。Oracle クライアント ライブラリの入ったディレクトリの場所は、インストールによって異なります。次に例を示します。

  ```
  $ORACLE_HOME/lib
  ```

Oracle 9i では、次の構文を使用します。

- Oracle 9i 用のネイティブ インタフェース (ドライバ ファイル) **libweblogicoci38.sl** と **libweblogicoxa38.so** が入っているディレクトリ。次に例を示します。

  ```
  $WL_HOME/server/lib/hpux11/oci901_8
  ```

- Oracle が提供するライブラリが入ったディレクトリ。Oracle クライアント ライブラリの入ったディレクトリの場所は、インストールによって異なります。次に例を示します。

  ```
  $ORACLE_HOME/lib32
  ```

例

上記の構文に従って Oracle 8.1.7 用の実際のパスを作成すると次のようになります。

```
export SHLIB_PATH=
$WL_HOME/server/lib/hpux11/oci817_8:$ORACLE_HOME/lib:$SHLIB_PATH
```

Oracle 9.0.1 では、パスは次のようになります。

```
export SHLIB_PATH=
$WL_HOME/server/lib/hpux11/oci901_8:$ORACLE_HOME/lib32:$SHLIB_PAT
H
```

次の表は、HP-UX 用のディレクトリと Oracle クライアントのバージョンを示します。

表 2-4 HP 上の Oracle

Oracle クライアント バージョン	OCI API バージョン	共有ライブラリ (.sl) ディレクトリ	メモ
8.1.7	8	oci817_8	Oracle 8 と JDBC 2.0 コア API およびオプション パッケージ API (分散トランザクション機能を含む) にアクセス可能。
9.0.1	8	oci901_8	Oracle 9 と JDBC 2.0 コア API およびオプション パッケージ API (分散トランザクション機能を含む) にアクセス可能。
9.2.0	8	oci920_8	Oracle 9.2 と JDBC 2.0 コア API およびオプション パッケージ API (分散トランザクション機能を含む) にアクセス可能。

次の表に、HP の 32 ビットおよび 64 ビット インストールの、ベンダ提供の Oracle ライブラリがあるディレクトリを示します。

表 2-5 HP にインストールされた Oracle ライブラリへのパス

Oracle クライアント バージョン	アーキテクチャ	Oracle ライブラリへのパス
8.1.7	32 ビット	ORACLE_HOME/lib
8.1.7	64 ビット	ORACLE_HOME/lib64
9.0.1	32 ビット	ORACLE_HOME/lib32
9.0.1	64 ビット	ORACLE_HOME/lib
9.2.0	32 ビット	ORACLE_HOME/lib32
9.2.0	64 ビット	ORACLE_HOME/lib

SGI IRIX

ご使用のプラットフォームがサポートされているかどうかを確認するには、「動作確認状況」を参照してください。

Siemens MIPS

ご使用のプラットフォームがサポートされているかどうかを確認するには、「動作確認状況」を参照してください。

Compaq Tru64 UNIX

ご使用のプラットフォームがサポートされているかどうかを確認するには、「動作確認状況」を参照してください。

Oracle データベースとの接続の確認

WebLogic jDriver for Oracle をインストールしたら、このドライバを使ってデータベースに接続できるかどうか確認します。確認するには、WebLogic Server ソフトウェアに付属の dbping を使用します。

環境を設定し dbping を使用するには、次のコマンドをコマンド ラインに入力します。

```
WL_HOME\server\bin\setWLSEnv.cmd
set path=WL_HOME\server\bin\oci817_8;%PATH%
java utils.dbping ORACLE user password server
```

WL_HOME は、WebLogic プラットフォームがインストールされているディレクトリで、通常は c:\bea\weblogic700 です。

dbping ユーティリティの使用方法の詳細については、『管理者ガイド』の「WebLogic Java ユーティリティの使い方」を参照してください。

問題がある場合は、『WebLogic JDBC プログラマーズ ガイド』の「JDBC 接続のテストとトラブルシューティング」を参照してください。

接続プールの設定

WebLogic Server または WebLogic Express で WebLogic jDriver for Oracle を使用している場合、WebLogic Server の起動時に Oracle DBMS との接続を確立する接続プールを設定できます。接続はユーザ間で共有されるので、接続プールを使用すると、ユーザごとに新規のデータベース接続を開くオーバーヘッドをなくすことができます。

アプリケーションは次に、JNDI ツリーで DataSource をルックアップし、接続プールに接続を要求します。データベース接続の終了時には、アプリケーションがその接続を接続プールに返します。

WebLogic Server ソフトウェアでの接続プールのコンフィグレーション

1. ベンダ提供のネイティブ ライブラリと、WebLogic Server 用の WebLogic ネイティブ ライブラリを、WebLogic Server を起動するシェルの PATH (Windows) またはロード ライブラリ パス (UNIX) に入れます。詳細については、『管理者ガイド』の「WebLogic Server の起動と停止」を参照してください。
2. Administration Console を使用して、接続プールを設定します。接続プールの詳細については、『管理者ガイド』の「JDBC コンポーネント (接続プール、データソース、マルチプール)」および Administration Console オンライン ヘルプの「JDBC 接続プールの作成とコンフィグレーション」を参照してください。

アプリケーションでの接続プールの使い方

表 2-6

接続プールを使用するアプリケーションのタイプ	データベース接続に使用するドライバ	詳細の参照先
クライアントサイド	JNDI ツリー上のデータソース	『WebLogic JDBC プログラマーズ ガイド』の「DataSource のコンフィグレーションと使い方」
サーバサイド (サーブレットとして使用)	WebLogic の RMI、Pool、および JTS ドライバ、または JNDI ツリー上のデータソース	『WebLogic HTTP サーブレット プログラマーズ ガイド』の「JDBC 接続プールを用いたデータベースへの接続」

クライアント アプリケーションでの JDBC アクティビティのロギング

WebLogic jDriver for Oracle を使用してデータベース接続を作成している接続プールからの接続をクライアント アプリケーションで使用している場合、クライアント上の JDBC アクティビティはサーバ上の JDBC ログに自動的には含まれません。JDBC のロギングを有効にし、クライアント上の JDBC アクティビティをサーバ上の JDBC ログに含めるには、次の手順に従います。

1. JDBC のロギングを有効にします。Administration Console で次の手順に従います。
   a. 左ペイン内で [サーバ] のノードをクリックします。
   b. 左ペインで特定のサーバを選択します。
   c. [ログ] タブを選択します。
   d. [JDBC] タブを選択します。
   e. [JDBC ログ記録を有効化] を選択します。

2. 接続プールのプロパティに JDBCDebug=true を追加します。Administration Console で次の手順に従います。

   a. 左ペインで、[JDBC] ノード、[接続プール] ノードを順にクリックして展開し、クライアントの JDBC アクティビティのログを記録する接続プールを選択します

   b. 右ペインで、[コンフィグレーション | 一般] タブを選択します。

   c. [プロパティ] ボックスで、新しい行に次のテキストを追加します。

      ```
      JDBCDebug=true
      ```

      [適用] をクリックします。

   d. サーバを再起動します。

WebLogic jDriver での IDE またはデバッガの使い方

統合開発環境 (IDE) またはデバッガを使用している場合、WebLogic 付属のネイティブ ライブラリ (ドライバ ファイル) を新しいファイルにコピーし、ファイル拡張子の前が _g で終わるファイル名に変えてください。次に例を示します。

- UNIX システムでは、libweblogicoci38.so を libweblogicoci38_g.so にコピーします。分散トランザクションの場合は、libweblogicoxa38.so を libweblogicoxa38_g.so にコピーします。
- Windows NT プラットフォームでは、weblogicoci38.dll を weblogicoci38_g.dll にコピーします。分散トランザクションの場合は、weblogicoxa38.dll を weblogicocoxa38_g.dll にコピーします。

開発環境を設定して WebLogic jDriver for Oracle を使用する準備

詳細については、以下を参照してください。

表 2-7

内容	参照先
JDBC クライアントを実行する開発環境の設定	『WebLogic Server アプリケーションの開発』の「開発環境の構築」
ドライバの使い方	『WebLogic jDriver for Oracle のコンフィグレーションと使い方』(このガイド) の「WebLogic jDriver for Oracle の使い方」

3 WebLogic jDriver for Oracle の使い方

この章では、簡単なアプリケーションに関する基本作業について説明します。この章には、コード例やサポートされていないメソッドのリストも含まれています。

- ローカル トランザクションと分散トランザクションの比較
- JDBC パッケージのインポート
- CLASSPATH の設定
- Oracle クライアント ライブラリのバージョン、URL、およびドライバ クラス名
- Oracle DBMS への接続
- 簡単な SQL クエリの作り方
- レコードの挿入、更新、および削除
- ストアド プロシージャおよび関数の作り方と使い方
- 接続の切断とオブジェクトのクローズ
- ストアド プロシージャからの ResultSets の処理
- WebLogic JDBC による行キャッシング
- コード例
- サポートされていない JDBC 2.0 メソッド

ローカル トランザクションと分散トランザクションの比較

WebLogic Server でトランザクションを実行する場合は、ローカル トランザクションと分散トランザクションのどちらを使うかによって、一部の基本作業が異なります。トランザクションは以下のように分けられます。

- ローカル トランザクション ー WebLogic jDriver for Oracle を使用します。
- 分散トランザクション、またはグローバル トランザクション ー XA モードの WebLogic jDriver for Oracle (WebLogic jDriver for Oracle/XA) を使用します。

分散トランザクションの詳細については、「分散トランザクションでの WebLogic jDriver for Oracle/XA の使い方」を参照してください。

JDBC パッケージのインポート

アプリケーションにインポートしなければならないクラスは以下のとおりです。

```
import java.sql.*;
import java.util.Properties; \\ 接続パラメータ設定用に Properties
                             \\ オブジェクトを使用する場合にだけ必要
import weblogic.common.*;

import javax.sql.Datasource; \\ 接続の取得に DataSource API を
                             \\ 使用する場合にだけ必要

import javax.naming.*;       \\ DataSource オブジェクトのルックアップに
                             \\ JNDI を使用する場合にだけ必要
```

WebLogic Server ドライバは、`java.sql interface` を実装します。アプリケーションを作成するには、`java.sql` クラスを使用します。JDBC ドライバ クラスをインポートする必要はありませんが、代わりに、アプリケーション内でドライバをロードします。これにより、適切なドライバを実行時に選択できるようになります。接続先となる DBMS をプログラムのコンパイル後に決めることもできます。

CLASSPATH の設定

WebLogic Server に付属のドライバを使用して WebLogic Server クライアントを実行する場合は、CLASSPATH に次のディレクトリを追加しなければなりません。

```
%WL_HOME%\server\lib\weblogic.jar
```

(%WL_HOME% は、WebLogic プラットフォームがインストールされているディレクトリで、通常は c:\bea\weblogic700)

CLASSPATH の設定と環境設定に関する問題の詳細については、『WebLogic Server アプリケーションの開発』の「開発環境の構築」を参照してください。

Oracle クライアント ライブラリのバージョン、URL、およびドライバ クラス名

使用するドライバ クラス名と URL は、以下の要素によって決まります。

- 使用するプラットフォーム
- 使用する Oracle クライアント ライブラリのバージョン

また、システムのパスに正しいドライバ バージョンを指定しなければなりません。詳細については、「WebLogic jDriver for Oracle の使用環境の設定」を参照してください。

ドライバを通常 (非 XA) モードで使用する場合 :

- ドライバ クラス : `weblogic.jdbc.oci.Driver`
- URL : `jdbc:weblogic:oracle`

ドライバを XA モードで使用する場合 :

- ドライバ クラス : `weblogic.jdbc.oci.xa.XADataSource`

- URL : 不要

Oracle DBMS への接続

以下の節で説明するように、2 層接続または多層接続を使用して、アプリケーションから Oracle DBMS への接続を確立します。

WebLogic Server を使用したデータベースへの 2 層コンフィグレーションによる接続

WebLogic Server を使用して、アプリケーションから Oracle DBMS に 2 層接続するには、次の手順を実行します。接続の詳細については、2-12 ページの「WebLogic Server ソフトウェアでの接続プールのコンフィグレーション」を参照してください。

1. WebLogic Server **JDBC ドライバ クラス**をロードし、`java.sql.Driver` オブジェクトにキャストします。XA ドライバを使用している場合は Datasource API を使用します。`java.sql.Driver` API は使用しません。次に例を示します。

   ```
   Driver myDriver = (Driver)Class.forName
   ("weblogic.jdbc.oci.Driver").newInstance();
   ```

2. 接続を記述する `java.util.Properties` オブジェクトを作成します。このオブジェクトは、ユーザ名、パスワード、データベース名、サーバ名、およびポート番号などの情報が入った名前と値の組み合わせを格納します。次に例を示します。

   ```
   Properties props = new Properties();
   props.put("user",         "scott");
   props.put("password",     "secret");
   props.put("server",           "DEMO" );
   ```

   サーバ名 (上の例では `DEMO`) は、Oracle クライアントのインストール先ディレクトリにある `tnsnames.ora` ファイル内のエントリを参照します。サーバ名によって、ホスト名と Oracle データベースについてのその他の情報が定義されます。サーバ名を提供しなかった場合、システムは環境変数 (Oracle の

場合は ORACLE_SID) を探します。次のフォーマットに従って、サーバ名を URL に追加することもできます。

```
"jdbc:weblogic:oracle:DEMO"
```

この構文でサーバを指定する場合、*server* プロパティを提供する必要はありません。

PowerSoft's PowerJ などの製品で使用するために、単一の URL 内にプロパティを設定することもできます。

3. `Driver.connect()` メソッドを呼び出すことで、JDBC の操作で不可欠となる JDBC 接続オブジェクトを作成します。このメソッドは、パラメータとして**ドライバの URL** と手順 2 で作成した `java.util.Properties` オブジェクトを取ります。次に例を示します。

```
Connection conn =
  myDriver.connect("jdbc:weblogic:oracle", props);
```

手順 1 と 3 では、JDBC ドライバを記述します。手順 1 では、ドライバの完全パッケージ名を使用します。ドットを使って区切ります。手順 3 では、URL (コロンで区切ります) を使ってドライバを識別します。URL には、`jdbc:weblogic:oracle` という文字列が入っていなければなりません。このほかに、サーバのホスト名やデータベース名などの情報を入れてもかまいません。

多層コンフィグレーションの WebLogic Server を使用した接続

WebLogic Server の多層コンフィグレーションで、アプリケーションから Oracle DBMS に接続するには、次の手順を実行します。

1. JNDI を使用して WebLogic RMI ドライバにアクセスするには、DataSource オブジェクトの JNDI 名をルックアップすることで、JNDI ツリーから Context オブジェクトを取得します。たとえば、Administration Console で定義した「myDataSource」という JNDI 名の DataSource にアクセスする手順は次のとおりです。

```
try {
  Context ctx = new InitialContext();
  javax.sql.DataSource ds
    = (javax.sql.DataSource) ctx.lookup ("myDataSource");
```

```
} catch (NamingException ex) {

  // ルックアップに失敗
}
```

2. DataSource オブジェクトから JDBC 接続を取得する手順は次のとおりです。

```
try {
 java.sql.Connection conn = ds.getConnection();
} catch (SQLException ex) {
  // 接続の取得に失敗
}
```

詳細については、『』「DataSource のコンフィグレーションと使い方」を参照してください。

接続のサンプル

このサンプルは、myDB というデータベースに接続するために Properties オブジェクトをどのように使用するかを示します。

```
Properties props = new Properties();
props.put("user",          "scott");
props.put("password",      "secret");
props.put("db",            "myDB");

Driver myDriver = (Driver)
  Class.forName("weblogic.jdbc.oci.Driver").newInstance();
Connection conn =
  myDriver.connect("jdbc:weblogic:oracle", props);
```

Connection オブジェクトについて

Connection オブジェクトはアプリケーションの重要な部分です。Connection クラスは、アプリケーションで使用する多くの基本的なデータベースに対するコンストラクタを持ちます。たとえば次のサンプルでは、Connection オブジェクト conn が何度も使われています。データベースに接続すると、アプリケーションの初期部分は終了します。

Connection オブジェクトを使った処理が終了したら、直ちにこのオブジェクトに対して close() メソッドを呼び出す必要があります。通常は、クラスの最後で呼び出します。

自動コミットの設定

自動コミットのデフォルト設定は、次の表のとおりです。

表 3-1 自動コミットのデフォルト設定

トランザクション タイプ	自動コミットのデフォルト設定	デフォルト設定の変更	結果
ローカル トランザクション	true	変更する	デフォルト設定を false に変更するとパフォーマンスが向上することがある。
分散トランザクション	false	変更しない	デフォルト設定を変更してはならない。true に変更すると、SQLException が発生する。

簡単な SQL クエリの作り方

データベース アクセスにおける最も基本的な作業は、データを検索することです。WebLogic Server を使ってデータを検索するには、次の 3 段階の手順を実行します。

1. SQL クエリを DBMS に送る Statement を作成します。
2. 作成した Statement を実行します。
3. 実行した結果を `ResultSet` に入れます。このサンプルでは、従業員テーブル (エリアス名 `emp`) に対して簡単なクエリを実行し、3 つのカラムのデータを表示します。また、データの検索先のテーブルに関するメタデータにアクセスして表示します。最後に `Statement` を閉じます。

```
Statement stmt = conn.createStatement();
stmt.execute("select * from emp");
ResultSet rs = stmt.getResultSet();

while (rs.next()) {
  System.out.println(rs.getString("empid") + " - " +
```

```
                    rs.getString("name")  + " - " +
                    rs.getString("dept"));
  }

ResultSetMetaData md = rs.getMetaData();

System.out.println("Number of columns: " +
      md.getColumnCount());
for (int i = 1; i <= md.getColumnCount(); i++) {
    System.out.println("Column Name: "     +
      md.getColumnName(i));
    System.out.println("Nullable: "       +
      md.isNullable(i));
    System.out.println("Precision: "      +
      md.getPrecision(i));
    System.out.println("Scale: "          +
      md.getScale(i));
    System.out.println("Size: "           +
      md.getColumnDisplaySize(i));
    System.out.println("Column Type: "    +
      md.getColumnType(i));
    System.out.println("Column Type Name: "+
      md.getColumnTypeName(i));
    System.out.println("");
  }

stmt.close();
```

レコードの挿入、更新、および削除

この手順では、データベース テーブルのレコードの挿入、更新、および削除という、データベースに関する 3 つの一般的な作業を示します。これらの処理には、JDBC PreparedStatement を使います。まず、PreparedStatement を作成してから、それを実行し、閉じます。

PreparedStatemen (JDBC Statement のサブクラス) を使用すると、同じ SQL を値を変えて何度でも実行できます。PreparedStatement では、JDBC の「?」構文を使用します。

```
String inssql =
   "insert into emp(empid, name, dept) values (?, ?, ?)";
PreparedStatement pstmt = conn.prepareStatement(inssql);
for (int i = 0; i < 100; i++) {
  pstmt.setInt(1, i);
  pstmt.setString(2, "Person " + i);
  pstmt.setInt(3, i);
  pstmt.execute():
```

```
}
  pstmt.close();
```

PreparedStatement を使用してレコードを更新することもできます。次のサンプルでは、カウンタ「i」の値を「dept」フィールドの現在の値に追加します。

```
String updsql =
   "update emp set dept = dept + ? where empid = ?";
PreparedStatement pstmt2 = conn.prepareStatement(updsql);
 for (int i = 0; i < 100; i++) {
  pstmt2.setInt(1, i);
  pstmt2.setInt(2, i);
  pstmt2.execute();
}
pstmt2.close();
```

最後に、PreparedStatement を使用して、さきほど追加および更新されたレコードを削除します。

```
String delsql = "delete from emp where empid = ?";
PreparedStatement pstmt3 = conn.prepareStatement(delsql);
for (int i = 0; i < 100; i++) {
  pstmt3.setInt(1, i);
  pstmt3.execute();
}
pstmt3.close();
```

注意： varchar2 の値を挿入または更新するときに空の文字列 (“”) を挿入しようとすると、Oracle はその値を NULL として解釈します。値を挿入するカラムに NOT NULL 制約がある場合、データベースは「ORA-01400 : カラム名には NULL を挿入できません」エラーを送出します。

ストアド プロシージャおよび関数の作り方と使い方

WebLogic Server で使用するトランザクションのタイプによって、ストアド プロシージャとストアド関数の使い方が決まります。

- ローカル トランザクションの場合、ストアド プロシージャとストアド関数の作成、使用、および削除ができます。

- 分散トランザクション (XA モードのドライバ) の場合、ストアド プロシージャとストアド関数を実行できます。ただし、ストアド プロシージャとストアド関数を削除および作成することはできません。

まず、一連の文を実行して、ストアド プロシージャとストアド関数をデータベースから削除します。

```
Statement stmt = conn.createStatement();
try {stmt.execute("drop procedure proc_squareInt");}
catch (SQLException e) {// ここに例外処理をコーディング ;}
try {stmt.execute("drop procedure func_squareInt");}
catch (SQLException e) {// ここに例外処理をコーディング ;}
try {stmt.execute("drop procedure proc_getresults");}
catch (SQLException e) {// ここに例外処理をコーディング ;}
stmt.close();
```

JDBC Statement を使用してストアド プロシージャまたはストアド関数を作成してから、JDBC の「?」構文で JDBC CallableStatement (Statement のサブクラス) を使用して、`IN` および `OUT` パラメータを設定します。

ネイティブ Oracle では SQL 文中で「?」値のバインディングをサポートしていません。代わりに、「`:1`」、「`:2`」等を使用します。WebLogic Server では SQL 文の中でどちらの構文を使用することもできます。

ストアド プロシージャの入力パラメータは、JDBC の `IN` パラメータにマップされており、`setInt()` などの `CallableStatement.setXXX()` メソッドと JDBC PreparedStatement「?」構文で使われます。ストアド プロシージャの出力パラメータは、JDBC の `OUT` パラメータにマップされており、`CallableStatement.registerOutParameter()` メソッドと JDBC PreparedStatement「?」構文で使われます。`IN` と `OUT` の両方のパラメータを使って、`setXXX()` と `registerOutParameter()` の呼び出しが両方とも同じパラメータ番号で行われるようにしてもかまいません。

このサンプルでは、JDBC Statement を使用して Oracle ストアド プロシージャを 1 つ作成してから、そのプロシージャを CallableStatement. を使用して実行しています。`registerOutParameter()` メソッドを使用して、2 乗された値を入れるための出力パラメータを設定しています。

```
Statement stmt1 = conn.createStatement();
stmt1.execute
  ("CREATE OR REPLACE PROCEDURE proc_squareInt " +
  "(field1 IN OUT INTEGER, field2 OUT INTEGER) IS " +
  "BEGIN field2 := field1 * field1; field1 := " +
  "field1 * field1; END proc_squareInt;");
```

```
stmt1.close();

// ネイティブ Oracle SQL をここにコメントアウト
// String sql = "BEGIN proc_squareInt(?, ?); END;";

// これは JDBC で指定された正しい構文
String sql = "{call proc_squareInt(?, ?)}";
CallableStatement cstmt1 = conn.prepareCall(sql);

// 出力パラメータを登録する
cstmt1.registerOutParameter(2, java.sql.Types.INTEGER);
for (int i = 0; i < 5; i++) {
  cstmt1.setInt(1, i);
  cstmt1.execute();
  System.out.println(i + " " + cstmt1.getInt(1) + " "
   + cstmt1.getInt(2));
} cstmt1.close();
```

次のサンプルでは、同様のコードを使用して、整数を 2 乗するストアド関数を作成して実行します。

```
Statement stmt2 = conn.createStatement();
stmt2.execute("CREATE OR REPLACE FUNCTION func_squareInt " +
              "(field1 IN INTEGER) RETURN INTEGER IS " +
              "BEGIN return field1 * field1; " +
              "END func_squareInt;");
stmt2.close();

// ネイティブ Oracle SQL をここにコメントアウト
// sql = "BEGIN ?:= func_squareInt(?); END;";

// これは JDBC で指定された正しい構文
sql = "{ ?= call func_squareInt(?)}";
CallableStatement cstmt2 = conn.prepareCall(sql);

cstmt2.registerOutParameter(1, Types.INTEGER);
for (int i = 0; i < 5; i++) {
  cstmt2.setInt(2, i);
  cstmt2.execute();
  System.out.println(i + " " + cstmt2.getInt(1) +
                     " " + cstmt2.getInt(2));
}
cstmt2.close();
```

次に、sp_getmessages というストアド プロシージャを使用します (このストアド プロシージャのコードはこのサンプルには含まれていません)。このストアド プロシージャは、入力パラメータとしてメッセージ番号を取り、メッセージ テキストの入ったテーブルからメッセージ番号に対応するメッセージ テキストを探し、そのメッセージ テキストを出力パラメータとして ResultSet に返して格

納します。Statement.execute() および Statement.getResult() メソッドを使ってストアド プロシージャから返された ResultSets をすべて処理してからでないと、OUT パラメータと戻りステータスは使用可能になりません。

まず、CallableStatement に対する 3 つのパラメータを設定します。

1. パラメータ 1 (出力のみ) はストアド プロシージャの戻り値
2. パラメータ 2 (入力のみ) は sp_getmessage への msgno 引数
3. パラメータ 3 (出力のみ) はメッセージ番号に対応して返されたメッセージテキスト

```
String sql = "{ ? = call sp_getmessage(?, ?)}";
 CallableStatement stmt = conn.prepareCall(sql);

 stmt.registerOutParameter(1, java.sql.Types.INTEGER);
 stmt.setInt(2, 18000);            // メッセージ番号 18000
 stmt.registerOutParameter(3, java.sql.Types.VARCHAR);
```

次に、ストアド プロシージャを実行し、戻り値をチェックして、ResultSet が空かどうかを調べます。空でない場合は、ループを使用して、その内容を取り出して表示するという処理を繰り返します。

```
boolean hasResultSet = stmt.execute();
while (true)
{
  ResultSet rs = stmt.getResultSet();
  int updateCount = stmt.getUpdateCount();
  if (rs == null && updateCount == -1) // 他に結果がない場合
    break;
  if (rs !== null) {
    // 空になるまで ResultSet オブジェクトを処理する
    while (rs.next()) {
      System.out.println
      ("Get first col by id:"+ rs.getString(1));
    }
  } else {
    // 更新件数がある
    System.out.println("Update count = " +
     stmt.getUpdateCount());
  }
  stmt.getMoreResults();
}
```

ResultSet の処理が終了すると、OUT パラメータと戻りステータスが使用可能になります。

```
int retstat = stmt.getInt(1);
String msg = stmt.getString(3);
```

```
System.out.println("sp_getmessage:status = " +
                   retstat + " msg = " + msg);
stmt.close();
```

接続の切断とオブジェクトのクローズ

接続を閉じる前に、データベースに対する変更をコミットするために commit() メソッドを呼び出すと便利な場合があります。

自動コミットが true (デフォルトの JDBC トランザクション モード) に設定されている場合、各 SQL 文がそれぞれトランザクションになります。しかし、このサンプルでは、Connection を作成した後に、自動コミットを false に設定しました。このモードでは、Connection は関連する暗黙的なトランザクションを常に持っており、rollback() または commit() メソッドを呼び出すと、現在のトランザクションが終了し、新しいトランザクションが開始されます。close() の前に commit() を呼び出すと、Connection を閉じる前にすべてのトランザクションが必ず完了します。

Statement、PreparedStatement、および CallableStatement を使う作業が終了したときにこれらのオブジェクトを閉じるように、アプリケーションの最後のクリーンアップとして、Connection オブジェクトの close() メソッドを try {} ブロック内で必ず呼び出し、例外を補足して適切な処理を行います。このサンプルの最後の 2 行では、commit を呼び出してから close を呼び出して接続を閉じます。

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

ストアド プロシージャからの ResultSets の処理

ストアド プロシージャを実行すると、複数の ResultSets が返されることがあります。ストアド プロシージャから返された ResultSets を、`Statement.execute()` および `Statement.getResultSet()` メソッドを使って処理する場合は、返された ResultSets をすべて処理してからでないと、OUT パラメータまたは戻りステータスは使用できません。

WebLogic JDBC による行キャッシング

Oracle はクライアントに配列フェッチ機能も提供しており、jDriver for Oracle はこの機能をサポートしています。デフォルトでは、jDriver for Oracle は最大 100 行の配列を DBMS から取得します。この数字は、`weblogic.oci.cacheRows` プロパティを使って変更できます。

上記のメソッドを使用すると、100 行の WebLogic JDBC クエリは、クライアントから WebLogic へ 4 つの呼び出しを実行するだけで済む上に、実際に WebLogic がデータを要求するために DBMS に送る呼び出しは 1 つだけです。詳細については、5-10 ページの「Oracle 配列フェッチのサポート」を参照してください。

コード例

次のコードでは、JDBC アプリケーションの構造を示します。ここに示すサンプル コードの内容は、データの検索、メタデータの表示、データの挿入、削除、および更新、さらに、ストアド プロシージャおよびストアド関数の呼び出しです。JDBC 関連の各オブジェクトに対して `close()` を明示的に呼び出すだけでなく、`try {}` ブロックでラップした `close()` を呼び出して、Connection 自体を `finally {}` ブロックで閉じてください。

```
package examples.jdbc.oracle;

import java.sql.*;
import java.Properties;
import weblogic.common.*;

public class test {
  static int i;
  Statement stmt = null;

  public static void main(String[] argv) {
    try {
      Properties props = new Properties();
      props.put("user",              "scott");
      props.put("password",          "tiger");
      props.put("server",            "DEMO");

    Driver myDriver = (Driver) Class.forName
      ("weblogic.jdbc.oci.Driver").newInstance();

    Connection conn =
      myDriver.connect("jdbc:weblogic:oracle", props);

    }
    catch (Exception e)
      e.printStackTrace();
    }

    try {
      // これにより Oracle のパフォーマンスを向上する
      // 後で commit() を明示的に呼び出す必要がある
      conn.setAutoCommit(false);

      stmt = conn.createStatement();
      stmt.execute("select * from emp");
      ResultSet rs = stmt.getResultSet();

      while (rs.next()) {
        System.out.println(rs.getString("empid") + " - " +
                           rs.getString("name")  + " - " +
                           rs.getString("dept"));
      }

      ResultSetMetaData md = rs.getMetaData();

      System.out.println("Number of Columns: " +
         md.getColumnCount());
      for (i = 1; i <= md.getColumnCount(); i++) {
        System.out.println("Column Name: "     +
          md.getColumnName(i));
        System.out.println("Nullable: "       +
          md.isNullable(i));
        System.out.println("Precision: "      +
          md.getPrecision(i));
        System.out.println("Scale: "          +
          md.getScale(i));
```

```
  System.out.println("Size: "            +
    md.getColumnDisplaySize(i));
  System.out.println("Column Type: "     +
    md.getColumnType(i));
  System.out.println("Column Type Name: "+
    md.getColumnTypeName(i));
  System.out.println("");
}
rs.close();
stmt.close();

Statement stmtdrop = conn.createStatement();
try {stmtdrop.execute("drop procedure proc_squareInt");}
catch (SQLException e) {;}
try {stmtdrop.execute("drop procedure func_squareInt"); }
catch (SQLException e) {;}
try {stmtdrop.execute("drop procedure proc_getresults"); }
catch (SQLException e) {;}
stmtdrop.close();

// ストアド プロシージャを作成する
Statement stmt1 = conn.createStatement();
stmt1.execute
 ("CREATE OR REPLACE PROCEDURE proc_squareInt " +
 "(field1 IN OUT INTEGER, " +
 "field2 OUT INTEGER) IS " +
 "BEGIN field2 := field1 * field1;" +
 "field1 := field1 * field1; " +
 "END proc_squareInt;");
stmt1.close();

CallableStatement cstmt1 =
  conn.prepareCall("BEGIN proc_squareInt(?, ?); END;");
cstmt1.registerOutParameter(2, Types.INTEGER);
for (i = 0; i < 100; i++) {
  cstmt1.setInt(1, i);
  cstmt1.execute();
  System.out.println(i + " " + cstmt1.getInt(1) +
                     " " + cstmt1.getInt(2));
}
cstmt1.close();

// ストアド関数を作成する
Statement stmt2 = conn.createStatement();
stmt2.execute
 ("CREATE OR REPLACE FUNCTION func_squareInt " +
 "(field1 IN INTEGER) RETURN INTEGER IS " +
 "BEGIN return field1 * field1; END func_squareInt;");
stmt2.close();

CallableStatement cstmt2 =
  conn.prepareCall("BEGIN ?:= func_squareInt(?); END;");
cstmt2.registerOutParameter(1, Types.INTEGER);
for (i = 0; i < 100; i++) {
  cstmt2.setInt(2, i);
  cstmt2.execute();
```

```
      System.out.println(i + " " + cstmt2.getInt(1) +
                         " " + cstmt2.getInt(2));
    }
    cstmt2.close();

    // レコードを 100 件挿入する


    System.out.println("Inserting 100 records...");
    String inssql =
      "insert into emp(empid, name, dept) values (?, ?, ?)";
    PreparedStatement pstmt = conn.prepareStatement(inssql);

    for (i = 0; i < 100; i++) {
      pstmt.setInt(1, i);
      pstmt.setString(2, "Person " + i);
      pstmt.setInt(3, i);
      pstmt.execute():
    }
    pstmt.close();

    // レコードを 100 件更新する
    System.out.println("Updating 100 records...");
    String updsql =
     "update emp set dept = dept + ? where empid = ?";
    PreparedStatement pstmt2 = conn.prepareStatement(updsql);

    for (i = 0; i < 100; i++) {
      pstmt2.setInt(1, i);
      pstmt2.setInt(2, i);
      pstmt2.execute();
    }
    pstmt2.close();

    // レコードを 100 件削除する
    System.out.println("Deleting 100 records...");
    String delsql = "delete from emp where empid = ?";
    PreparedStatement pstmt3 = conn.prepareStatement(delsql);

    for (i = 0; i < 100; i++) {
      pstmt3.setInt(1, i);
      pstmt3.execute();
    }
    pstmt3.close();

    conn.commit();
  }
  catch (Exception e) {
    // 失敗を適切に処理する
  }
  finally {
    try {conn.close();}
    catch (Exception e) {
      // 例外を捕捉して処理する
    }
  }
```

```
  }

}
```

これ以外の Oracle サンプル コードについては、samples\examples\jdbc\oracle ディレクトリを参照してください。

サポートされていない JDBC 2.0 メソッド

WebLogic Server はすべての JDBC 2.0 メソッドをサポートしていますが、WebLogic jDriver for Oracle ではサポートしていない JDBC 2.0 メソッドもあります。これらのメソッドを使用する必要がある場合は、Oracle Thin ドライバなどの別の JDBC ドライバを使用してデータベースに接続することができます。表 3-2 に、WebLogic jDriver for Oracle でサポートされていない JDBC 2.0 メソッドを示します。

表 3-2 WebLogic jDriver for Oracle でサポートされていない JDBC 2.0 メソッド

クラス またはインタフェース	サポートされていないメソッド
java.sql.Blob	public long position(Blob blob, long l) public long position(byte abyte0[], long l)
java.sql.CallableStatement	public Array getArray(int i) public Date getDate(int i, Calendar calendar) public Object getObject(int i, Map map) public Ref getRef(int i) public Time getTime(int i, Calendar calendar) public Timestamp getTimestamp(int i, Calendar calendar) public void registerOutParameter(int i, int j, String s)
java.sql.Clob	public long position(String s, long l) public long position(java.sql.Clob clob, long l)

表 3-2 WebLogic jDriver for Oracle でサポートされていない JDBC 2.0 メソッド (続き)

クラス またはインタフェース	サポートされていないメソッド
java.sql.Connection	public java.sql.Statement createStatement(int i, int j) public Map getTypeMap() public CallableStatement prepareCall(String s, int i, int j) public PreparedStatement prepareStatement(String s, int i, int j) public void setTypeMap(Map map)
java.sql.DatabaseMetaData	public Connection getConnection() public ResultSet getUDTs(String s, String s1, String s2, int ai[]) public boolean supportsBatchUpdates()
java.sql.PreparedStatement	public void addBatch() public ResultSetMetaData getMetaData() public void setArray(int i, Array array) public void setNull(int i, int j, String s) public void setRef(int i, Ref ref)

表 3-2 WebLogic jDriver for Oracle でサポートされていない JDBC 2.0 メソッド (続き)

クラス またはインタフェース	サポートされていないメソッド
java.sql.ResultSet	public boolean absolute(int i) public void afterLast() public void beforeFirst() public void cancelRowUpdates() public void deleteRow() public boolean first() public Array getArray(int i) public Array getArray(String s) public int getConcurrency() public int getFetchDirection() public int getFetchSize() public Object getObject(int i, Map map) public Object getObject(String s, Map map) public Ref getRef(int i) public Ref getRef(String s) public int getRow() public Statement getStatement() public int getType() public void insertRow()

表 3-2 WebLogic jDriver for Oracle でサポートされていない JDBC 2.0 メソッド (続き)

クラス またはインタフェース	サポートされていないメソッド
java.sql.ResultSet (続き)	public boolean isAfterLast() public boolean isBeforeFirst() public boolean isFirst() public boolean isLast() public boolean last() public void moveToCurrentRow() public void moveToInsertRow() public boolean previous() public void refreshRow() public boolean relative(int i) public boolean rowDeleted() public boolean rowInserted() public boolean rowUpdated() public void setFetchDirection(int i) public void setFetchSize(int i) public void updateAsciiStream(int i, InputStream inputstream, int j) public void updateAsciiStream(String s, InputStream inputstream, int i) public void updateBigDecimal(int i, BigDecimal bigdecimal) public void updateBigDecimal(String s, BigDecimal bigdecimal) public void updateBinaryStream(int i, InputStream inputstream, int j) public void updateBinaryStream(String s, InputStream inputstream, int i) public void updateBoolean(int i, boolean flag) public void updateBoolean(String s, boolean flag) public void updateByte(int i, byte byte0) public void updateByte(String s, byte byte0) public void updateBytes(int i, byte abyte0[]) public void updateBytes(String s, byte abyte0[])

表 3-2 WebLogic jDriver for Oracle でサポートされていない JDBC 2.0 メソッド (続き)

クラス またはインタフェース	サポートされていないメソッド
java.sql.ResultSet (続き)	public void updateCharacterStream(int i, Reader reader, int j) public void updateCharacterStream(String s, Reader reader, int i) public void updateDate(int i, Date date) public void updateDate(String s, Date date) public void updateDouble(int i, double d) public void updateDouble(String s, double d) public void updateFloat(int i, float f) public void updateFloat(String s, float f) public void updateInt(int i, int j) public void updateInt(String s, int i) public void updateLong(int i, long l) public void updateLong(String s, long l) public void updateNull(int i) public void updateNull(String s) public void updateObject(int i, Object obj) public void updateObject(int i, Object obj, int j) public void updateObject(String s, Object obj) public void updateObject(String s, Object obj, int i) public void updateRow() public void updateShort(int i, short word0) public void updateShort(String s, short word0) public void updateString(int i, String s) public void updateString(String s, String s1) public void updateTime(int i, Time time) public void updateTime(String s, Time time) public void updateTimestamp(int i, Timestamp timestamp) public void updateTimestamp(String s, Timestamp timestamp)
java.sql.ResultSetMetaData	public String getColumnClassName(int i)

4 分散トランザクションでの WebLogic jDriver for Oracle/XA の使い方

以下の節では、WebLogic jDriver for Oracle/XA を使用し、BEA WebLogic Server 環境で動作する EJB および RMI アプリケーションにトランザクションを統合する方法について説明します。

- WebLogic jDriver for Oracle の XA モードと 非 XA モードの違い
- JDBC XA および 非 XA リソースのコンフィグレーション
- WebLogic jDriver for Oracle XA の制限
- 分散トランザクションの実装

WebLogic jDriver for Oracle の XA モードと 非 XA モードの違い

WebLogic jDriver for Oracle は、分散トランザクションについて JDBC 2.0 オプション パッケージ API を完全サポートしています。分散トランザクション (XA) モードでこのドライバを使用するアプリケーションは、以下の例外を除いて、ローカル トランザクション (非 XA) モードの場合と同じように JDBC 2.0 コア API を使用できます。

- 接続を取得するには、非推奨になった java.sql.DriverManager または java.sql.Driver API ではなく、JDBC 2.0 javax.sql.DataSource API を使用しなければなりません。

- 接続プール内の物理的なデータベース接続の扱いが異なります。4-2 ページの「WebLogic XA jDriver での接続の動作」を参照してください。
- WebLogic Server で使用する場合は、TxDataSource をコンフィグレーションしなければなりません。TxDataSource と接続プールをコンフィグレーションする手順については、『管理者ガイド』の「JDBC 接続の管理」を参照してください。
- 自動コミットはデフォルトで false に設定されます。Connection オブジェクトで `java.sql.Connection.setAutoCommit` メソッドを呼び出して自動コミット モードを有効化しようとすると、SQLException が発生します。
- `java.sql.Connection.commit` または `java.sql.Connection.rollback` メソッドを呼び出して分散トランザクションを終了しようとすると、SQLException が発生します。

最後の 2 つの理由が違うのは、WebLogic jDriver for Oracle/XA が分散トランザクションを構成する場合、分散トランザクションの境界を決めたり、調整したりするのが外部のトランザクション マネージャだからです。

詳細については、「JDBC 2.0 Standard Extension API spec」(バージョン 1.0、98/12/7 付、Section 7.1 の最後の 2 パラグラフ) を参照してください。

WebLogic XA jDriver での接続の動作

WebLogic XA jDriver for Oracle は内部的に Oracle C/XA スイッチを使用するため、`xa_open` と `xa_start` は、SQL 呼び出しを行う各スレッド上で呼び出される必要があります。また、`xa_open` を呼び出すと、新しい物理的な XA データベース接続が作成されます。これらの制限に対処するために、WebLogic XA jDriver for Oracle では、スレッドが接続を使用するまでは物理的なデータベース接続を作成しません。スレッドが接続を使用するときに、XA jDriver は `xa_open` (および `xa_start`) を呼び出して接続を作成し、その接続をスレッドに関連付けます。データベース接続が作成された後も、接続はスレッドにそのまま関連付けられており、ドライバは `xa_close` を呼び出しません。以降にスレッドでデータベース接続が必要になると、スレッドは関連付けられている同じデータベース接続を使用します。ただし、これは JDBC 接続プールから接続を取得して返しているように見えます。

このドライバの動作によって、JDBC 接続プールの動作も若干変わります。

- 接続プールがデプロイされるとき (作成時またはサーバ起動時)、論理的な JDBC 接続オブジェクトは作成されますが、物理的なデータベース接続は作成されません。
- 接続プール内の論理的な接続の数は、ほとんどの場合、物理的なデータベース接続の数と同じにはなりません。物理的なデータベース接続の数は、SQL 呼び出しが行われる実行スレッドの数と同じになります。
- JDBC 接続プール内の論理的な接続オブジェクトの数によって、データベース作業を同時に行えるスレッドの数が制限されます。接続プールの最大容量は、システム内の実行スレッドの数に設定する必要があります。
- 物理的なデータベース接続は、JDBC 接続プールが破棄されるとき (アンデプロイ時またはサーバ停止時) に閉じられます。サーバの停止後、または JDBC 接続が破棄された後も、物理的なデータベース接続はデータベース上に残り、DBMS によって最終的にクリーンアップされます。

注意： これらの動作の変更点は、物理的なデータベース接続の作成に WebLogic XA jDriver を使用する JDBC 接続プールにのみ適用されます。他の XA ドライバを使用する接続プールには適用されません。

JDBC XA および 非 XA リソースのコンフィグレーション

Administration Console を使用して、以下の節で説明するように JDBC リソースをコンフィグレーションします。

JDBC/XA リソース

XA 対応 JDBC ドライバを分散トランザクションに参加させるには、以下のように JDBC 接続プールをコンフィグレーションします。

- `DriverName` プロパティに、`javax.sql.XADataSource` インタフェースをサポートしているクラスの名前を指定します。つまり、

`weblogic.jdbc.oci.xa.XADataSource` を `DriverName` プロパティ (Administration Console の `Driver Classname`) として使用します。

- データベース プロパティが指定されていることを確認します。WebLogic jDriver for Oracle のデータ ソース プロパティの詳細については、『管理者ガイド』の「分散トランザクション用の XA 対応 JDBC ドライバのコンフィグレーション」を参照してください。

手順や属性の定義については、JDBC 接続プールのパネルで Administration Console オンライン ヘルプを参照してください。

XA 非対応の JDBC リソース

XA 非対応の JDBC リソースをサポートするには、JDBC トランザクション データ ソースをコンフィグレーションするときに、`enableTwoPhaseCommit` データベース プロパティ (Administration Console の [`Emulate Two-Phase Commit for non-XA Driver`]) を選択します。このプロパティの詳細については、『管理者ガイド』の「分散トランザクション用の XA 非対応 JDBC ドライバのコンフィグレーション」を参照してください。

WebLogic jDriver for Oracle XA の制限

XA モードの WebLogic jDriver for Oracle は以下の動作をサポートしていません。

- ローカル トランザクションとグローバル トランザクションの混合。このため、グローバル トランザクションを使用せずに SQL の処理が試みられると、SQLException が送出されます。
- DDL 処理の実行 (テーブルの作成または削除、ストアド プロシージャなど)。DDL 処理を実行する場合は、次のような 2 つの異なる接続プールを定義する必要があります。
  - DDL 処理で使用できる XA 非対応の接続プール。
  - 分散トランザクションで DML 処理に使用できる XA 対応の接続プール。

- トランザクションのトランザクション アイソレーション レベルの設定。トランザクションは、接続に設定されたトランザクション アイソレーション レベル、またはデータベースに対するデフォルトのトランザクション アイソレーション レベルを使用します。

分散トランザクションの実装

ここでは以下について説明します。

- パッケージのインポート
- JNDI を介したデータ ソースの検索
- 分散トランザクションの実行

パッケージのインポート

コード リスト 4-1 は、アプリケーションがインポートするパッケージを示します。特に以下の点に注意してください。

- `java.sql.*` および `javax.sql.*` パッケージは、データベース操作で不可欠です。
- `javax.naming.*` パッケージは、サーバの起動時にコマンドライン パラメータとして渡されるプール名についての JNDI ルックアップの実行に不可欠です。プール名は、そのサーバ グループで登録する必要があります。

コード リスト 4-1 必要なパッケージのインポート

```
import java.sql.*;
import javax.sql.*;
import javax.naming.*;
```

JNDI を介したデータ ソースの検索

コード リスト 4-2 は、JNDI を介したデータ ソースの検索方法を示します。

コード リスト 4-2　JNDI を介したデータ ソースの検索

```
static DataSource pool;

...

public void get_connpool(String pool_name)
    throws Exception
  {
    try {
      javax.naming.Context ctx = new InitialContext();
      pool = (DataSource)ctx.lookup("jdbc/" + pool_name);
    }
    catch (javax.naming.NamingException ex){
      TP.userlog("Couldn't obtain JDBC connection pool: " +
pool_name);
      throw ex;
    }
  }
}
```

分散トランザクションの実行

コード リスト 4-3 は、2 つのデータベース接続を使用し、セッション Bean 内のビジネス メソッドとして実装された分散トランザクションを示します。

コード リスト 4-3　分散トランザクションの実行

```
public class myEJB implements SessionBean {
     EJBContext ejbContext;

     public void myMethod(...) {
          javax,transaction.UserTransaction usertx;
          javax.sql.DataSource data1;
          javax.sql.DataSource data2;
          java.sql.Connection conn1;
          java.sql.Connection conn2;
          java.sql.Statement stat1;
          java.sql.Statement stat2;
```

```
        InitialContext initCtx = new InitialContext();

        //
        // ユーザ トランザクション オブジェクトを初期化する
        //
        usertx = ejbContext.getUserTransaction();

        // 新規のユーザ トランザクションを開始する
        usertx.begin();

        // 最初のデータベースとの接続を確立し、
        // トランザクションの処理を準備する
        data1 = (javax.sql.DataSource)
             initCtx.lookup("java:comp/env/jdbc/DataBase1");
        conn1 = data1.getConnection();

        stat1 = conn1.getStatement();

        // 2 番目のデータベースとの接続を確立し、
        // トランザクションの処理を準備する
        data2 = (javax.sql.DataSource)
             initCtx.lookup("java:comp/env/jdbc/DataBase2");
        conn2 = data1.getConnection();

        stat2 = conn2.getStatement();

        // conn1 および conn2 の両方を更新する。EJB コンテナは
        // 関連するリソースを自動的にリスト表示する
        stat1.executeQuery(...);
        stat1.executeUpdate(...);
        stat2.executeQuery(...);
        stat2.executeUpdate(...);
        stat1.executeUpdate(...);
        stat2.executeUpdate(...);

        // トランザクションをコミットする
        // 関連するデータベースに変更を適用する
        usertx.commit();

        // すべての接続と文を解放する
        stat1.close();
        stat2.close();
        conn1.close();
        conn2.close();
    }
    ...
}
```

5　Oracle の高度な機能

この章では、WebLogic jDriver for Oracle で使用する以下の Oracle の高度な機能について説明します。

- 大文字 / 小文字を区別せずにメタデータを扱う方法
- データ型
- WebLogic Server と Oracle の NUMBER カラム
- Oracle の Long raw データ型の使い方
- Oracle リソース上の待機
- JDBC 拡張 SQL のサポート
- Oracle 用 JDBC 2.0 の概要
- JDBC 2.0 のサポートに必要なコンフィグレーション
- BLOB と CLOB
- 文字と ASCII ストリーム
- 新しい日付関連メソッド

注意： WebLogic Server では、PreparedStatement、CallableStatement、配列、Struct、REF に対しても Oracle 拡張メソッドをサポートしています。これらの拡張メソッドを使用するためには、Oracle Thin ドライバを使用してデータベースに接続する必要があります。

大文字 / 小文字を区別せずにメタデータを扱う方法

WebLogic Server では、allowMixedCaseMetaData プロパティを設定できます。このプロパティをブール値 true に設定すると、このプロパティは、DatabaseMetaData メソッドの呼び出しで大文字 / 小文字を区別しないように、Connection オブジェクトを設定します。このプロパティを false に設定した場合、Oracle は、データベース メタデータについて大文字をデフォルトで使用します。

次のサンプル コードは、この機能を利用するためのプロパティの設定方法を示します。

```
Properties props = new Properties();
props.put("user",                    "scott");
props.put("password",                "tiger");
props.put("server",                  "DEMO");
props.put("allowMixedCaseMetaData", "true");

Driver myDriver = (Driver)
  Class.for.Name(weblogic.jdbc.oci.Driver).newInstance();

Connection conn =
   myDriver.connect("jdbc:weblogic:oracle", props);
```

このプロパティを設定しなかった場合、WebLogic Server は Oracle のデフォルト設定を使用するので、データベース メタデータについては大文字が使用されます。

データ型

次の表は、Oracle データ型と Java 型との推奨マッピングを示します。この他にも、Oracle データ型を Java で表現する方法はあります。結果セットの処理中に getObject() メソッドを呼び出した場合、クエリ対象の Oracle カラムに対するデフォルトの Java データ型が返されます。

表 5-1

Oracle	WebLogic Server
Varchar	String
Number	Tinyint
Number	Smallint
Number	Integer
Number	Long
Number	Float
Number	Numeric
Number	Double
Long	Longvarchar
RowID	String
Date	Timestamp
Raw	(var)Binary
Long raw	Longvarbinary
Char	(var)Char
Boolean*	Number OR Varchar
MLS label	String
Blob	Blob
Clob	Clob

* PreparedStatement.setBoolean( ) を呼び出すと、VARCHAR 型は 1 または 0 (文字列) に、NUMBER 型は 1 または 0 (数字) に変換されます。

WebLogic Server と Oracle の NUMBER カラム

Oracle には NUMBER というカラム タイプがあります。このカラム タイプは、オプションとして NUMBER(P) および NUMBER(P,S) の形式で精度とスケールを指定できます。修飾されていない単純な NUMBER 形式でも、このカラムは、小さな整数値から非常に大きな浮動小数点までのすべての数値タイプを高い精度で保持できます。

WebLogic Server アプリケーションがこうしたカラムの値を要求すると、WebLogic Server は、カラム内の値を要求された Java 型に変換します。getInt() で 123.456 という値が要求された場合、当然、値は丸められます。

ただし、getObject() メソッドの場合は、これより若干複雑になります。WebLogic Server は、NUMBER カラムの値を同様の精度で表現する Java オブジェクトで必ず返します。つまり、値 1 は Integer として返されますが、123434567890.123456789 のような値は BigDecimal でのみ返されます。

カラムの値の最大精度を Oracle から報告するメタデータはありません。したがって、WebLogic Server は、それぞれの値に基づいて、どのような種類のオブジェクトを返すかを判断する必要があります。つまり、1 つの ResultSet が、NUMBER カラムに対して getObject() から複数の Java 型を返す場合があるということです。整数値だけのテーブルはすべて Integer として getObject() から返されることもあり、浮動小数点単位のテーブルは主に Double で返されますが、「123.00」のような値は Integer として返される場合があります。Oracle からは、NUMBER 値の「1」と「1.0000000000」を識別するための情報は提供されていません。

修飾された NUMBER カラム、つまり、特定の精度が定義されているカラムは、動作の信頼性が高くなります。Oracle のメタデータはこれらのパラメータをドライバに提供するため、WebLogic Server はテーブルの値にかかわりなく、常に特定の精度とスケールに合わせて適切な Java オブジェクトを返します。

表 5-2

カラム定義	getObject() の戻り値の型
NUMBER(P <= 9)	Integer
NUMBER(P <= 18)	Long
NUMBER(P = 19)	BigDecimal
NUMBER(P <=16, S 0)	Double
NUMBER(P = 17, S 0)	BigDecimal

Oracle の Long raw データ型の使い方

WebLogic Server では、BLOB、CLOB、Long、および Long raw といった Oracle のデータ型を使用する場合に備えて、2 つのプロパティを提供しています。BLOB および CLOB データ型は、Oracle バージョン 8 と JDBC 2.0 でサポートされているだけですが、これらのプロパティは、Oracle バージョン 7 で使用可能な Oracle の Long raw データ型に対しても適用できます。

Oracle リソース上の待機

注意： waitOnResources() メソッドは、Oracle 8 API 使用時にはサポートされません。

WebLogic Server のドライバは、Oracle の oopt() C 関数をサポートしています。これは、リソースが使用可能になるまでクライアントが待機できるようにする機能です。Oracle C 関数は、要求されたリソースが使用可能でない場合のオプション (ロックを待機するかどうかなど) を設定します。

開発者は、クライアントが DBMS リソースを待機するか、または直ちに例外を受け取るかを指定できます。次のコードは、サンプル コード ファイル (examples\jdbc\oracle\waiton.java) からの抜粋です。

```
java.util.Properties props = new java.util.Properties();
props.put("user",     "scott");
props.put("password", "tiger");
props.put("server",   "myserver");

Driver myDriver = (Driver)
  Class.forName("weblogic.jdbc.oci.Driver").newInstance();

// この拡張機能を利用するために
// Connection を weblogic.jdbc.oci.Connection としてキャストする必要がある

Connection conn =(weblogic.jdbc.oci.Connection)
  myDriver.connect("jdbc:weblogic:oracle", props);

// オブジェクトの作成後、直ちに
// waitOnResources メソッドを呼び出す

conn.waitOnResources(true);
```

このメソッドを使用すると、短期間ロックされる内部リソースを待つ間に、いくつかのエラー リターン コードが発生する場合があります。

この機能を使用するには、次の処理を行う必要があります。

1. Connection オブジェクトを `weblogic.jdbc.oci.Connection` としてキャストします。
2. `waitOnResources()` メソッドを呼び出します。

この関数については、『The OCI Functions for C』のセクション 4-97 に記載されています。

自動コミット

JDBC WebLogic Server のデフォルト トランザクション モードでは、自動コミットを true と仮定します。Connection オブジェクトの作成後、次の文で自動コミットを false に設定することで、プログラムの性能を改善できます。

```
Connection.setAutoCommit(false);
```

トランザクションのアイソレーション レベル

WebLogic Server は、以下のトランザクションのアイソレーション レベルをサポートしています。

- `SET TRANSACTION ISOLATION LEVEL READ COMMITTED`
- `SET TRANSACTION ISOLATION LEVEL SERIALIZABLE`

Oracle DBMS はこれら 2 つのアイソレーション レベルのみをサポートしています。他の JDBC ドライバと違い、WebLogic Server は、開発者がサポートされていないアイソレーション レベルを使おうとした場合に例外を生成します。一部のドライバは、サポートされていないアイソレーション レベルを設定しようとした場合に、例外を生成することなく無視します。

`READ_UNCOMMITTED` トランザクション アイソレーション レベルはサポートされていません。

コードセットのサポート

JDBC と WebLogic Server は、Java 内の文字列を Unicode 文字列として扱います。Oracle DBMS は別のコードセットを使用するため、Unicode から Oracle が使用するコードセットに変換する必要があります。WebLogic Server は、Oracle の環境変数 `NLS_LANG` 内の値を調べ、表 5-3 に示すマッピングを使用して、変換に使用する JDK のコードセットを選択します。`NLS_LANG` 変数が設定されていない場合、または JDK に認識できないコードセットに設定されていた場合、ドライバは正しいコードセットを選択できません (`NLS_LANG` の正しい構文の詳細については、Oracle のドキュメントを参照してください)。

コードセットを変換する場合は、コード内で接続を確立するときに `Driver.connect()` メソッドを使用して、次のプロパティを WebLogic Server に渡す必要があります。

```
props.put("weblogic.oci.min_bind_size", 660);
```

このプロパティは、バインドされるバッファの最小サイズを定義します。デフォルトでは 2000 バイトで、これは最大値でもあります。コードセットを変換する場合は、このプロパティを使用して、バインド サイズを最大 2000 バイトの 1/3 の最大 660 に減らす必要があります。この理由は、Oracle コード変換では、拡張を考慮してバッファが 3 倍に拡大されるからです。

WebLogic Server には、Java コード内からコードセットを設定するための `weblogic.codeset` プロパティがあります。たとえば、次の例のように、`cp863` コードセットを使用するには、`Driver.connect()` を呼び出す前に、Properties オブジェクトを作成し、`weblogic.codeset` プロパティを設定します。

```
java.util.Properties props = new java.util.Properties();
props.put("weblogic.codeset", "cp863");
props.put("user", "scott");
props.put("password", "tiger");

String connectUrl = "jdbc:weblogic:oracle";

Driver myDriver = (Driver)
 Class.forName("weblogic.jdbc.oci.Driver").newInstance();

Connection conn =
     myDriver.connect(connectUrl, props);
```

コードセット サポートは、JVM の種類によって異なります。特定のコードセットがサポートされているかどうかについては、使用している JDK のドキュメントをチェックしてください。

注意： Oracle クライアントの `NLS_LANG` 環境変数も、同じまたは対応するコードセットに設定する必要があります。

表 5-3 NLS_LANG 設定と JDK コードセットのマッピング対応表

NLS_LANG	JDK コードセット
al24utffss	UTF8
al32utf8	UTF8
ar8iso8859p6	ISO8859_6
cdn8pc863	Cp863
cl8iso8859p5	ISO8859_5
cl8maccyrillic	MacCyrillic

cl8mswin1251	Cp1251
ee8iso8859p2	ISO8859_2
ee8macce	MacCentralEurope
ee8maccroatian	MacCroatian
ee8mswin1250	Cp1250
ee8pc852	Cp852
el8iso8859p7	ISO8859_7
el8macgreek	MacGreek
el8mswin1253	Cp1253
el8pc737	Cp737
is8macicelandic	MacIceland
is8pc861	Cp861
iw8iso8859p8	ISO8859_8
ja16euc	EUC_JP
ja16sjis	SJIS
ko16ksc5601	EUC_KR
lt8pc772	Cp772
lt8pc774	Cp774
n8pc865	Cp865
ne8iso8859p10	ISO8859_10
nee8iso8859p4	ISO8859_4
ru8pc855	Cp855
ru8pc866	Cp866
se8iso8859p3	ISO8859_3
th8macthai	MacThai

tr8macturkish	MacTurkish
tr8pc857	Cp857
us7ascii	ASCII
us8pc437	Cp437
utf8	UTF8
we8ebcdic37	Cp1046
we8ebcdic500	Cp500
we8iso8859p1	ISO8859_1
we8iso8859p15	ISO8859_15_FDIS
we8iso8859p9	ISO8859_9
we8macroman8	MacRoman
we8pc850	Cp850
we8pc860	Cp860
zht16big5	Big5

Oracle 配列フェッチのサポート

WebLogic Server は、Oracle 配列フェッチをサポートしています。`ResultSet.next()` を最初に呼び出したときには、1 行を取り出すのではなく、行の配列を取得して、それをメモリに格納します。それ以降の `next()` に対する各呼び出しは、メモリに格納した行をそれぞれ 1 行読み取ります。この操作はメモリ内の行がなくなるまで続き、行がなくなると、`next()` への呼び出しはデータベースに戻ります。

配列フェッチのサイズを制御するには、プロパティ (`java.util.Property`) を設定します。このプロパティは `weblogic.oci.cacheRows` で、デフォルトで 100 に設定されています。このプロパティを 300 に設定する例を次に示します。これは、`next()` への呼び出しは、クライアントが取り出す 300 行につき 1 回だけデータベースをヒットすることを意味します。

```
Properties props = new Properties();
props.put("user",      "scott");
props.put("password",  "tiger");
props.put("server",    "DEMO");
props.put("weblogic.oci.cacheRows", "300");

Driver myDriver = (Driver)
  Class.forName("weblogic.jdbc.oci.Driver").newInstance();

Connection conn = myDriver.connect("jdbc:weblogic:oracle", props);
```

この JDBC 拡張機能を利用することで、クライアントの性能を改善し、データベース サーバの負荷を緩和できます。ただし、クライアントでの行のキャッシングは、クライアント リソースを必要とします。ネットワーク コンフィグレーションとアプリケーションに応じて、性能とクライアント リソースのバランスが最適になるようにアプリケーションを調整してください。

SELECT 内のいずれかのカラムのタイプが Long、BLOB、または CLOB の場合、WebLogic Server はその Select 文に関連付けられている ResultSet のキャッシュ サイズを一時的に 1 にリセットします。

ストアド プロシージャの使い方

この節では、Oracle に固有のストアド プロシージャのさまざまな実装について説明します。

- Oracle カーソルへのパラメータのバインド
- CallableStatement の使用上の注意

Oracle カーソルへのパラメータのバインド

WebLogic は、ストアド プロシージャのパラメータを Oracle カーソルにバインドできるようにする JDBC の拡張機能(`weblogic.jdbc.oci.CallableStatement`)を作成しました。ストアド プロシージャの結果を使って、JDBC ResultSet オブジェクトを作成できます。これによって、複数の ResultSet を整理して返すことができます。ResultSet は実行時にストアド プロシージャ内で決定されます。

次に例を示します。まず、次のようにストアド プロシージャを定義します。

```
create or replace package
curs_types as
type EmpCurType is REF CURSOR RETURN emp%ROWTYPE;
end curs_types;
/

create or replace procedure
single_cursor(curs1 IN OUT curs_types.EmpCurType,
ctype in number) AS BEGIN
  if ctype = 1 then
    OPEN curs1 FOR SELECT * FROM emp;
  elsif ctype = 2 then
    OPEN curs1 FOR SELECT * FROM emp where sal  2000;
  elsif ctype = 3 then
    OPEN curs1 FOR SELECT * FROM emp where deptno = 20;
  end if;
END single_cursor;
/
create or replace procedure
multi_cursor(curs1 IN OUT curs_types.EmpCurType,
             curs2 IN OUT curs_types.EmpCurType,
             curs3 IN OUT curs_types.EmpCurType) AS
BEGIN
    OPEN curs1 FOR SELECT * FROM emp;
    OPEN curs2 FOR SELECT * FROM emp where sal  2000;
    OPEN curs3 FOR SELECT * FROM emp where deptno = 20;
END multi_cursor;
/
```

Java コード内で、ストアド プロシージャを使用して CallableStatements を作成し、出力パラメータをデータ型 java.sql.Types.OTHER で登録します。データを ResultSet 内に取り出すときに、出力パラメータのインデックスを getResultSet() メソッドの引数として使用します。

```
  java.sql.CallableStatement cstmt = conn.prepareCall(
                       "BEGIN OPEN ? " +
                       "FOR select * from emp; END;");
  cstmt.registerOutParameter(1, java.sql.Types.OTHER);

  cstmt.execute();
  ResultSet rs = cstmt.getResultSet(1);
  printResultSet(rs);
  rs.close();
  cstmt.close();

  java.sql.CallableStatement cstmt2 = conn.prepareCall(
                       "BEGIN single_cursor(?, ?); END;");
  cstmt2.registerOutParameter(1, java.sql.Types.OTHER);

  cstmt2.setInt(2, 1);
  cstmt2.execute();
  rs = cstmt2.getResultSet(1);
  printResultSet(rs);
```

```
  cstmt2.setInt(2, 2);
  cstmt2.execute();
  rs = cstmt2.getResultSet(1);}
  printResultSet(rs);

  cstmt2.setInt(2, 3);
  cstmt2.execute();
  rs = cstmt2.getResultSet(1);
  printResultSet(rs);
  cstmt2.close();

  java.sql.CallableStatement cstmt3 = conn.prepareCall(
                        "BEGIN multi_cursor(?, ?, ?); END;");
  cstmt3.registerOutParameter(1, java.sql.Types.OTHER);
  cstmt3.registerOutParameter(2, java.sql.Types.OTHER);
  cstmt3.registerOutParameter(3, java.sql.Types.OTHER);

  cstmt3.execute();

  ResultSet rs1 = cstmt3.getResultSet(1);
  ResultSet rs2 = cstmt3.getResultSet(2);
  ResultSet rs3 = cstmt3.getResultSet(3);
```

`printResultSet()` メソッドを含むこのサンプルの全コードについては、`samples\examples\jdbc\oracle\` ディレクトリにあるサンプルを参照してください。

Oracle ストアド プロシージャの文字列のデフォルト サイズは 256K です。

CallableStatement の使用上の注意

CallableStatement の OUTPUT パラメータにバインドされる文字列のデフォルト長は 128 文字です。バインド パラメータに割り当てた値がこの長さを超えると、次のエラーが発生します。

```
ORA-6502: value or numeric error
```

バインド パラメータの値の長さは、明示的な長さを scale 引数を使って `CallableStatement.registerOutputParameter()` メソッドに渡すことによって調節できます。256 文字を超えない `VARCHAR` をバインドするコード サンプルを次に示します。

```
  CallableStatement cstmt =
    conn.prepareCall("BEGIN testproc(?); END;");

  cstmt.registerOutputParameter(1, Types.VARCHAR, 256);
  cstmt.execute();
```

```
System.out.println(cstmt.getString());
cstmt.close();
```

DatabaseMetaData メソッド

この節では、Oracle に固有の DatabaseMetaData メソッドの実装について説明します。

- 一般に、String catalog 引数は、すべての `DatabaseMetaData` メソッドで無視されます。
- `DatabaseMetaData.getProcedureColumns()` メソッドでは、
  - String catalog 引数は無視されます。
  - String schemaPattern 引数は、完全一致するものだけを受け付けます (パターン マッチングは受け付けない)。
  - String procedureNamePattern 引数は、完全一致するものだけを受け付けます (パターン マッチングは受け付けない)。
  - String columnNamePattern 引数は無視されます。

JDBC 拡張 SQL のサポート

JavaSoft JDBC 仕様には、SQL 拡張が含まれています。SQL 拡張は SQL Escape 構文とも呼ばれています。すべての WebLogic jDriver は拡張 SQL をサポートしています。拡張 SQL によって、DBMS 間で移植可能な共通の SQL 拡張機能にアクセスできます。

たとえば、日付から曜日を取り出す関数は、SQL 標準では定義されていません。Oracle の SQL では次のようになります。

```
select to_char(date_column, 'DAY') from table_with_dates
```

同等の関数は、Sybase や Microsoft SQL Server では次のようになります。

```
select datename(dw, date_column) from table_with_dates
```

拡張 SQL を使うと、どちらの DBMS に対しても、次のようにして曜日を取り出すことができます。

```
select {fn dayname(date_column)} from table_with_dates
```

次のサンプルは、拡張 SQL の機能のいくつかを示します。

```
String query =
"-- This SQL includes comments and " +
    "JDBC extended SQL syntax.\n" +
"select into date_table values( \n" +
"      {fn now()},        -- current time \n" +
"      {d '1997-05-24'},  -- a date       \n" +
"      {t '10:30:29' },   -- a time       \n" +
"      {ts '1997-05-24 10:30:29.123'},  -- a timestamp\n" +
"     '{string data with { or } will not be altered}'\n" +
"-- Also note that you can safely include" +
   " { and } in comments or\n" +
"-- string data.";
Statement stmt = conn.createStatement();
stmt.executeUpdate(query);
```

拡張 SQL は、一般の SQL と区別するために中括弧 (「{}」) で囲ってあります。コメントはダブルハイフンで始まり、改行コード (「\n」) で終わっています。コメント、SQL、および拡張 SQL を含む拡張 SQL のシーケンス全体は、二重引用符で囲み、Statement オブジェクトの execute() メソッドに渡します。CallableStatement の一部に拡張 SQL を使った例を次に示します。

```
CallableStatement cstmt =
 conn.prepareCall("{ ? = call func_squareInt(?)}");
```

次のサンプルは、拡張 SQL 式をネストできるということを示しています。

```
select {fn dayname({fn now()})}
```

サポートされている拡張 SQL 関数の一覧は、DatabaseMetaData オブジェクトから取り出すことができます。次のサンプルは、JDBC ドライバがサポートしている関数のすべてをリストする方法を示しています。

```
DatabaseMetaData md = conn.getMetaData();
System.out.println("Numeric functions:     " +
   md.getNumericFunctions());
System.out.println("\nString functions:    " +
   md.getStringFunctions());
System.out.println("\nTime/date functions: " +
   md.getTimeDateFunctions());
System.out.println("\nSystem functions:    " +
   md.getSystemFunctions());
conn.close();
```

Oracle 用 JDBC 2.0 の概要

WebLogic jDriver for Oracle で実装されている JDBC 2.0 の機能は以下のとおりです。

- BLOB (Binary Large Object) − WebLogic Server は、この Oracle データ型を扱えるようになりました。
- CLOB (Character Large Object) − WebLogic Server は、この Oracle データ型を扱えるようになりました。
- Character Streams (ASCII と Unicode の両文字コード用) − 文字列ストリームを扱う場合、文字列をバイトの配列としてではなく文字の流れ (ストリーム) として扱う方法が優れています。
- バッチ更新−複数の文でも 1 単位としてまとめてデータベースに送れるようになりました。

以前のバージョンで利用可能だった既存の JDBC 機能に加えて、上記の新機能も WebLogic Server で利用できるようになりました。前バージョンのドライバで使用していた既存のコードは、すべてこの新 WebLogic jDriver for Oracle でも動作します。

注意： WebLogic Server では、PreparedStatement、CallableStatement、配列、Struct、REF に対しても Oracle 拡張メソッドをサポートしています。しかし、これらの拡張メソッドを使用するためには、Oracle Thin ドライバを使用してデータベースに接続する必要があります。

JDBC 2.0 のサポートに必要なコンフィグレーション

WebLogic Server バージョン 7.0 は JDK 1.3.1 上で動作するので、JDBC 2.0 には Java 2 環境が必要となります。サポートされているコンフィグレーションの全リストについては、「動作確認状況」ページを参照してください。

BLOB と CLOB

BLOB (Binary Large Object) および CLOB (Character Large Object) データ型は、Oracle バージョン 8 のリリースで利用できるようになりました。JDBC 2.0 仕様と WebLogic Server もこれらのデータ型をサポートしています。この節では、これらのデータ型の使い方について説明します。

トランザクション境界

Oracle での BLOB と CLOB は、トランザクション境界 (SQL の *commit* または *rollback* 文の前に発行された文) に関しては、他のデータ型とは動作が異なります。BLOB または CLOB は、トランザクションがコミットされると直ちに非アクティブになります。AutoCommit が TRUE に設定されている場合、その接続で各コマンドが発行された後に、トランザクションはそれぞれ自動的にコミットされます。SELECT 文の場合でもコミットされます。この理由により、複数の SQL 文にまたがって BLOB または CLOB を保持する必要がある場合には、AutoCommit を false に設定しなければなりません。トランザクションを適切なタイミングで手動でコミット (またはロールバック) することが必要になります。AutoCommit を false に設定するには、次のコマンドを入力します。

```
conn.setAutoCommit(false); // conn は接続オブジェクト
```

BLOB

Oracle バージョン 8 で使用可能になった BLOB データ型を使用すると、Oracle テーブルに大きなバイナリ オブジェクトを保存したり、テーブルから取り出したりできます。BLOB は JDBC 2.0 仕様の一部として定義されていますが、仕様では、テーブル内の BLOB カラムを更新するためのメソッドが提供されていません。しかし、BEA WebLogic の BLOB の実装は、JDBC 2.0 を拡張することでこの機能を提供します。

Connection プロパティ

`weblogic.oci.selectBlobChunkSize`

このプロパティは、I/O ストリームへバイトや文字を送信する際に使われる内部バッファのサイズを設定します。指定したサイズに達したら、ドライバは暗黙的に `flush()` 処理を実行します。これにより、データは DBMS に送られます。

この値を明示的に設定することは、クライアントのメモリ使用量の制御に役立ちます。

このプロパティの値が設定されていない場合には、デフォルト値 65534 が使用されます。

このプロパティを、プロパティとして Connection オブジェクトに渡すことで設定します。たとえば、次のコードは `weblogic.oci.selectBlobChunkSize` を 1200 に設定します。

```
Properties props = new Properties();
props.put("user",         "scott");
props.put("password",     "tiger");
props.put("server",       "DEMO");

props.put ("weblogic.oci.selectBlobChunkSize","1200");

Driver myDriver = (Driver)
 Class.forName("weblogic.jdbc.oci.Driver").newInstance();

Connection conn =
 driver.connect("jdbc:weblogic:oracle:myServer", props);
```

`weblogic.oci.insertBlobChunkSize`

このプロパティは、ドライバが内部的に使用する入力ストリームのバッファ サイズ (バイト単位) を指定します。

BLOB チャンク機能を使用して、Oracle DBMS に BLOB を挿入するには、このプロパティを正の整数に設定します。デフォルトでは、このプロパティは、BLOB チャンクを使用しないことを意味するゼロ (0) に設定されています。

Import 文

この節で説明されている BLOB 機能を使用するには、クライアント コードに以下のクラスをインポートします。

```
import java.sql.*;
import java.util.*;
import java.io.*;
import weblogic.jdbc.common.*;
```

BLOB フィールドの初期化

BLOB データ型が入った行を最初に挿入するときには、実際のデータを使ってそのフィールドを更新する前に、「空の」BLOB を持つ行を挿入する必要があります。空の BLOB を挿入するには、Oracle *EMPTY_BLOB()* 関数を使用します。

BLOB フィールドを初期化する手順は以下のとおりです。

1. 1 つまたは複数のカラムを BLOB データ型として定義したテーブルを作成します。
2. Oracle EMPTY_BLOB() 関数を使用して、空の BLOB カラムを 1 つ持つ行を 1 つ挿入します。

   ```
   stmt.execute("INSERT into myTable values (1,EMPTY_BLOB()");
   ```

3. その BLOB カラムへの「ハンドル」を取得します。

   ```
   java.sql.Blob myBlob = null;
   Statement stmt2 = conn.createStatement();
   stmt2.execute("SELECT myBlobColumn from myTable
     where pk = 1 for update");
   ResultSet rs = stmt2.getResultSet();
   rs.next() {
     myBlob = rs.getBlob("myBlobColumn");
     // 取得した BLOB を使用して何かする
   }
   ```

4. 以上で、BLOB にデータを書き込めるようになりました。次の節、「BLOB へのバイナリ データの書き込み」に進みます。

BLOB へのバイナリ データの書き込み

BLOB カラムにバイナリ データを書き込む手順は以下のとおりです。

1. 上記「BLOB フィールドの初期化」の手順 3. に従って、BLOB フィールドへのハンドルを取得します。
2. バイナリ データが入る InputStream オブジェクトを作成します。

```
java.io.InputStream is = // 入力ストリームを作成する
```

3. BLOB データを書き込むための出力ストリームを作成します。BLOB オブジェクトを weblogic.jdbc.common.OracleBlob にキャストしなければなりません。

```
java.io.OutputStream os =
((weblogic.jdbc.common.OracleBlob)
myBlob).getBinaryOutputStream();
```

4. バイナリ データが入った入力ストリームを出力ストリームに書き込みます。書き込み操作は、OutputStream オブジェクトの flush() メソッドを呼び出したときに終了します。

```
byte[] inBytes = new byte[65534]; // 下記の「注意」を参照
               int numBytes = is.read(inBytes);
while (numBytes > 0) {
  os.write(inBytes, 0, numBytes);
  numBytes = is.read(inBytes);
}
os.flush();
```

注意： 上記コードの中の値 [65534] は、65534 というデフォルト値を持つ weblogic.oci.select.BlobChunkSize プロパティが未設定であると仮定したものです。このプロパティを設定してある場合、byte[ ] の値を weblogic.oci.select.BlobChunkSize property に設定した値に合わせると、データを最も効率的に扱えるようになります。このプロパティの詳細については、「Connection プロパティ」を参照してください。

5. クリーンアップします。

```
os.close();
pstmt.close();
conn.close();
```

BLOB オブジェクトの書き込み

BLOB オブジェクトをテーブルに書き込むには、Prepared Statements を使用します。たとえば、myBlob オブジェクトをテーブル myOtherTable に書き込むコードは以下のとおりです。

```
PreparedStatement pstmt = conn.preparedStatement(
    "UPDATE myOtherTable SET myOtherBlobColumn = ? WHERE id = 12");

pstmt.setBlob(1, myBlob);
```

BLOB データの読み取り

getBlob() メソッドを使用して BLOB カラムを取り出してから、SQL SELECT 文の実行結果 ResultSet を使用した場合は、BLOB データへのポインタだけが返されます。バイナリ データは実際にはクライアントに転送されていません。getBinaryStream() メソッドを呼び出して初めて、データがストリーム オブジェクトに書き込まれます。

Oracle テーブルから BLOB データを読み取る手順は以下のとおりです。

1. SELECT 文を実行します。

```
stmt2.execute("SELECT myBlobColumn from myTable");
```

2. その SELECT 文の実行結果を使用します。

```
int STREAM_SIZE = 10;
byte[] r = new byte[STREAM_SIZE];

ResultSet rs = stmt2.getResultSet();
java.sql.Blob myBlob = null;
while (rs.next) {
  myBlob = rs.getBlob("myBlobColumn");

 java.io.InputStream readis = myBlob.getBinaryStream();

 for (int i=0 ; i < STREAM_SIZE ; i++) {
      r[i] = (byte) readis.read();
     System.out.println("output [" + i + "] = " + r[i]);
 }
```

3. クリーンアップします。

```
rs.close();
stmt2.close();
```

注意： また、CallableStatement を使用して、ResultSet を生成することもできます。この ResultSet は、上記と同じように使用できます。詳細については、JDK ドキュメントの java.sql.CallableStatment の部分を参照してください。

その他のメソッド

さらに、**java.sql.Blob** インタフェースの以下のメソッドが、WebLogic Server JDBC 2.0 ドライバに実装されています。詳細については、JDK ドキュメントを参照してください。

- getBinaryStream()
- getBytes()
- length()

position() メソッドは実装されていません。

CLOB

Oracle バージョン 8 で使用可能になった CLOB データ型は、Oracle テーブル内に大きな文字列を格納できます。JDBC 2.0 の仕様には CLOB カラムを直接更新する機能は含まれていないので、CLOB を挿入したり更新したりするために、BEA では getAsciiOutputStream() メソッド (ASCII データ用) と getCharacterOutputStream() メソッド (Unicode データ用) を実装しました。

コードセットのサポート

使用する Oracle Server およびクライアントのバージョンによっては、以下のプロパティのいずれかを設定する必要があります。設定するには、DBMS 接続を確立したときにそのプロパティを Connection オブジェクトに渡すように、Java クライアントのコード中に記述します。

weblogic.codeset
: このプロパティを使用すると、Java コード内からコードセットを設定できます。Oracle クライアントの NLS_LANG 環境変数も設定する必要があります。

weblogic.oci.ncodeset
: このプロパティは、Oracle サーバが使用するナショナル コードセットを設定します。Oracle クライアントの NLS_NCHAR 環境変数も設定する必要があります。

`weblogic.oci.codeset_width`
このプロパティは、使用している文字コードセットが何バイト幅のタイプなのかを WebLogic Server に知らせます。コードセットの使用については、以下の制限があります。
指定できる値は次のとおりです。
- `0` (可変幅のコードセットを使用する場合)
- `1` (固定幅のコードセットを使用する場合。`1` はデフォルト値)
- `2` または `3` (コードセットの幅をバイト単位で指定する場合)

`weblogic.oci.ncodeset_width`
Oracle のナショナル コードセットのいずれかを使用している場合には、このプロパティを使用してコードセットの文字幅を指定します。コードセットの使用については、以下の制限があります。
指定できる値は次のとおりです。
- `0` (可変幅のコードセットを使用する場合)
- `1` (固定幅のコードセットを使用する場合。1 はデフォルト値)
- `2` または `3` (コードセットの幅をバイト単位で指定する場合)

CLOB フィールドの初期化

CLOB データ型が入った行を最初に挿入するときには、実際のデータを使ってそのフィールドを更新する前に、「空の」CLOB を持つ行を挿入する必要があります。空の CLOB を挿入するには、Oracle *`EMPTY_CLOB()`* 関数を使用します。

CLOB カラムを初期化する手順は以下のとおりです。

1. 1 つまたは複数のカラムを CLOB データ型として定義したテーブルを作成します。
2. Oracle `EMPTY_CLOB()` 関数を使用して、空の CLOB カラムを 1 つ持つ行を 1 つ挿入します。

```
stmt.execute("INSERT into myTable VALUES (1,EMPTY_CLOB()");
```

3. CLOB カラムのオブジェクトを取得します。

```
java.sql.Clob myClob = null;
Statement stmt2 = conn.createStatement();
stmt2.execute("SELECT myClobColumn from myTable
  where pk = 1 for update");
ResultSet rs = stmt2.getResultSet();
while (rs.next) {
```

```
  myClob = rs.getClob("myClobColumn");
}
```

4. 以上で、CLOB に文字データを書き込めるようになりました。書き込むデータが ASCII フォーマットの場合は、次の節「CLOB への ASCII データの書き込み」に進みます。書き込むデータが Unicode フォーマットの場合は、「CLOB への Unicode データの書き込み」を参照してください。

CLOB への ASCII データの書き込み

CLOB カラムに ASCII 文字データを書き込む手順は以下のとおりです。

1. 上記「CLOB フィールドの初期化」の手順 3. に従って、CLOB への「ハンドル」を取得します。
2. 文字データが入るオブジェクトを作成します。

   ```
   String s = // ASCII データ
   ```

3. CLOB 文字列を書き込むための出力ストリームを作成します。CLOB オブジェクトを `weblogic.jdbc.common.OracleClob` にキャストしなければなりません。

   ```
   java.io.OutputStream os =
   ((weblogic.jdbc.common.OracleClob)
   myclob).getAsciiOutputStream();
   ```

4. ASCII データが入った入力ストリームを出力ストリームに書き込みます。書き込み操作は、`OutputStream` オブジェクトの `flush()` メソッドを呼び出したときに終了します。

   ```
   byte[] b = s.getBytes("ASCII");

   os.write(b);
   os.flush();
   ```

5. クリーンアップします。

   ```
   os.close();
   pstmt.close();
   conn.close();
   ```

CLOB への Unicode データの書き込み

CLOB カラムに Unicode 文字データを書き込む手順は以下のとおりです。

1. 「CLOB フィールドの初期化」の手順 3 に従って、CLOB への「ハンドル」を取得します。
2. 文字データが入るオブジェクトを作成します。

   ```
   String s = // Unicode 文字データ
   ```

3. CLOB 文字列を書き込むための文字出力ストリームを作成します。CLOB オブジェクトを `weblogic.jdbc.common.OracleClob` にキャストしなければなりません。

   ```
   java.io.Writer wr =
   ((weblogic.jdbc.common.OracleClob)
   myclob).getCharacterOutputStream();
   ```

4. ASCII データが入った入力ストリームを出力ストリームに書き込みます。書き込み操作は、`OutputStream` オブジェクトの `flush()` メソッドを呼び出したときに終了します。

   ```
   char[] b = s.toCharArray(); // 「s」を文字配列に変換
   ```

   ```
   wr.write(b);
   wr.flush();
   ```

5. クリーンアップします。

   ```
   wr.close();
   pstmt.close();
   conn.close();
   ```

CLOB オブジェクトの書き込み

CLOB オブジェクトをテーブルに書き込むには、Prepared Statements を使用します。たとえば、`myClob` オブジェクトをテーブル `myOtherTable` に書き込むコードは以下のとおりです。

```
PreparedStatement pstmt = conn.preparedStatement(
    "UPDATE myOtherTable SET myOtherClobColumn = ? WHERE id = 12");

pstmt.setClob(1, myClob);
```

PreparedStatement を使用した CLOB 値の更新

PreparedStatement を使用して CLOB を更新し、新しい値が以前の値より短い場合、CLOB は更新中に明示的に置換されなかった文字を保持します。たとえば、CLOB の現在の値が `abcdefghij` の場合に、PreparedStatement を使用して `zxyw` で CLOB を更新すると、CLOB の値が `zxywefghij` に更新されます。PreparedStatement で更新された値を修正するには、`dbms_lob.trim` プロシージャを使用して、更新後に残った余分な文字を削除します。`dbms_lob.trim` プロシージャの詳細については、Oracle のマニュアルを参照してください。

CLOB データの読み取り

SQL SELECT 文の実行結果を使用して CLOB カラムを取り出した場合は、CLOB データへのポインタだけが返されます。実際のデータはクライアントに転送されていません。`getAsciiStream()` メソッドが呼び出されて初めて、その文字データがストリームに読み込まれます。

Oracle テーブルから CLOB データを読み取る手順は以下のとおりです。

1. SELECT 文を実行します。

```
java.sql.Clob myClob = null;
Statement stmt2 = conn.createStatement();
stmt2.execute("SELECT myClobColumn from myTable");
```

2. その SELECT の文の実行結果を使用します。

```
ResultSet rs = stmt2.getResultSet();

while (rs.next) {
  myClob = rs.getClob("myClobColumn");
  java.io.InputStream readClobis =
      myReadClob.getAsciiStream();
  char[] c = new char[26];
  for (int i=0 ; i < 26  ; i++) {
      c[i] = (char) readClobis.read();
      System.out.println("output [" + i + "] = " + c[i]);
  }
}
```

3. クリーンアップします。

```
rs.close();
stmt2.close();
```

注意： また、CallableStatement を使用して、ResultSet を生成することもできます。この ResultSet は、上記と同じように使用できます。詳細については、JDK ドキュメントの java.sql.CallableStatment の部分を参照してください。

その他のメソッド

さらに、java.sql.Clob インタフェースの以下のメソッドが、WebLogic Server (JDBC 2.0 ドライバ) に実装されています。

- getSubString()
- length()

これらのメソッドの詳細については、JDK ドキュメントを参照してください。

注意： position() メソッドは実装されていません。

文字と ASCII ストリーム

JDBC 2.0 仕様の新しいメソッドの一部では、文字と ASCII ストリームを、以前のバージョンで実装されていたようにバイト列として扱うのではなく、文字列として扱うことができます。文字と ASCII ストリームを扱うための以下のメソッドが WebLogic Server で実装されています。

Unicode 文字ストリーム

getCharacterStream()
: java.sql.ResultSet インタフェースは、Unicode ストリームを Java の java.io.Reader 型として読み込むために、このメソッドを使用します。このメソッドは、非推奨になった getUnicodeStream() メソッドに代わって採用されました。

setCharacterStream()
: java.sql.PreparedStatement インタフェースは、java.io.Reader オブジェクトを書き込むためにこのメソッドを使用します。このメソッド

は、非推奨になった setUnicodeStream() メソッドに代わって採用されました。

ASCII 文字ストリーム

getAsciiStream()
: java.sql.ResultSet インタフェースは、ASCII ストリームを Java の java.io.InputStream 型として読み込むためにこのメソッドを使用します。

setAsciiStream()
: java.sql.PreparedStatement インタフェースは、java.io.InputStream オブジェクトを書き込むためにこのメソッドを使用します。

これらのメソッドの使い方の詳細については、JDK ドキュメントを参照してください。

バッチ更新

バッチ更新は JDBC 2.0 の新しい機能で、この機能を使用すると、複数の SQL 更新文を 1 単位として DBMS に送ることができます。アプリケーションによっては、複数の更新文を個々に送るよりも性能が向上することがあります。バッチ更新機能は、Statement インタフェースで使用可能ですが、更新件数を返して結果セットを返さない SQL 文を使用することが必要となります。callableStatement または preparedStatement を使用したバッチ更新はサポートされていません。

バッチ更新で使用できる SQL 文は以下のとおりです。

- INSERT INTO
- UPDATE
- DELETE
- CREATE TABLE
- DROP TABLE
- ALTER TABLE

バッチ更新の使い方

バッチ更新の使い方の基本的な手順を以下に示します。

1. 第 3 章「WebLogic jDriver for Oracle の使い方」の「Oracle DBMS への接続」の説明に従って、WebLogic Server JDBC 2.0 ドライバを使用して接続を得ます (connection オブジェクトを取得します)。このサンプルでは、接続オブジェクトは conn です。
2. `createStatement()` メソッドを使用して、statement オブジェクトを作成します。次に例を示します。

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

3. `addBatch()` メソッドを使用して、SQL 文をバッチに追加します。これらの文は、`executeBatch()` メソッドが呼び出されるまで、DBMS に送られません。次に例を示します。

   ```
   stmt.addBatch("INSERT INTO batchTest VALUES ('JOE', 20,35)");
   stmt.addBatch("INSERT INTO batchTest VALUES ('Bob', 30,44)");
   stmt.addBatch("INSERT INTO batchTest VALUES ('Ed',  34,22)");
   ```

4. `executeBatch()` メソッドを使用して、処理のためバッチを DBMS に送ります。次に例を示します。

   ```
   stmt.executeBatch();
   ```

   文が失敗して例外が発生した場合、文は 1 行も実行されません。

バッチ処理文の消去

`clearBatch()` メソッドを使用すると、`addBatch()` メソッドを使用して作成した文の集合を消去できます。次に例を示します。

```
stmt.clearBatch();
```

更新件数

JDBC 2.0 仕様によると、`executeBatch()` メソッドは、各 Statement で更新された行の数が入った整数の配列を返すことになっています。しかし、Oracle DBMS はこの情報をドライバに提供していません。代わりに、Oracle DBMS は、すべての更新に対して `-2` を返します。

新しい日付関連メソッド

以下のメソッドは、新しい署名を使用して、java.util.Calendar オブジェクトをパラメータとして取ります。java.util.Calendar を使用すると、日付の変換に使われるタイム ゾーンやロケーションの情報を指定できます。
java.util.Calendar クラスの使い方の詳細については、JDK API ガイドを参照してください。

```
java.sql.ResultSet.getDate(int columnIndex, Calendar cal)
       (java.sql.Date オブジェクトを返す)
```

```
java.sql.PreparedStatement.setDate
  (int parameterIndex, Date x, Calendar cal)
```