



BEA WebLogic Server™

BEA WebLogic Express™

WebLogic jDriver for Oracle のインストールと使い方

BEA WebLogic Server バージョン 6.1
マニュアルの日付 : 2002 年 6 月 24 日

著作権

Copyright © 2002 BEA Systems, Inc. All Rights Reserved.

限定的権利条項

本ソフトウェアおよびマニュアルは、BEA Systems, Inc. 又は日本ビー・イー・エー・システムズ株式会社（以下、「BEA」といいます）の使用許諾契約に基づいて提供され、その内容に同意する場合にのみ使用することができ、同契約の条項通りにのみ使用またはコピーすることができます。同契約で明示的に許可されている以外の方法で同ソフトウェアをコピーすることは法律に違反します。このマニュアルの一部または全部を、BEA からの書面による事前の同意なしに、複写、複製、翻訳、あるいはいかなる電子媒体または機械可読形式への変換も行うことはできません。

米国政府による使用、複製もしくは開示は、BEA の使用許諾契約、および FAR 52.227-19 の「Commercial Computer Software-Restricted Rights」条項のサブパラグラフ (c)(1)、DFARS 252.227-7013 の「Rights in Technical Data and Computer Software」条項のサブパラグラフ (c)(1)(ii)、NASA FAR 補遺 16-52.227-86 の「Commercial Computer Software--Licensing」条項のサブパラグラフ (d)、もしくはそれらと同等の条項で定める制限の対象となります。

このマニュアルに記載されている内容は予告なく変更されることがあり、また BEA による責務を意味するものではありません。本ソフトウェアおよびマニュアルは「現状のまま」提供され、商品性や特定用途への適合性を始めとする（ただし、これらには限定されない）いかなる種類の保証も与えません。さらに、BEA は、正当性、正確さ、信頼性などについて、本ソフトウェアまたはマニュアルの使用もしくは使用結果に関していかなる確約、保証、あるいは表明も行いません。

商標または登録商標

BEA、Jolt、Tuxedo、および WebLogic は BEA Systems, Inc. の登録商標です。BEA Builder、BEA Campaign Manager for WebLogic、BEA eLink、BEA Manager、BEA WebLogic Collaborate、BEA WebLogic Commerce Server、BEA WebLogic E-Business Platform、BEA WebLogic Enterprise、BEA WebLogic Integration、BEA WebLogic Personalization Server、BEA WebLogic Process Integrator、BEA WebLogic Server、E-Business Control Center、How Business Becomes E-Business、Liquid Data、Operating System for the Internet、および Portal FrameWork は、BEA Systems, Inc. の商標です。

その他の商標はすべて、関係各社がその権利を有します。

WebLogic jDriver for Oracle のインストールと使い方

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	2002 年 6 月 24 日	BEA WebLogic Server バージョン 6.1

目次

このマニュアルの内容

対象読者	vii
e-docs Web サイト	viii
このマニュアルの印刷方法	viii
関連情報	viii
サポート情報	ix
表記規則	x

1. 概要

WebLogic jDriver の概要	1-1
WebLogic jDriver for Oracle	1-2
Oracle 共有ライブラリ	1-2
WebLogic jDriver for Oracle/XA での分散トランザクション	1-3

2. WebLogic jDriver for Oracle のインストール

WebLogic jDriver for Oracle のインストール準備	2-1
WebLogic jDriver for Oracle のソフトウェア要件のチェック	2-1
サポートされるプラットフォーム	2-2
JDBC 2.0 の動作要件	2-2
WebLogic jDriver for Oracle の使用環境の設定	2-2
Windows NT	2-3
Solaris	2-4
IBM AIX	2-6
HP-UX 11	2-6
SGI IRIX	2-8
Siemens MIPS	2-8
Compaq Tru64 UNIX	2-9
WebLogic jDriver for Oracle のインストール手順	2-9
WebLogic jDriver for Oracle のインストール後の作業	2-11
ライセンス機能	2-11
Oracle データベースとの接続の確認	2-11

接続プールの設定	2-12
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	2-12
アプリケーションでの接続プールの使い方	2-13
WebLogic jDriver での IDE またはデバッガの使い方	2-13
開発環境を設定して WebLogic jDriver for Oracle を使用する準備	2-14

3. WebLogic jDriver for Oracle の使い方

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	3-2
JDBC パッケージのインポート	3-2
CLASSPATH の設定	3-3
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	3-3
Oracle DBMS への接続	3-4
2 層コンフィグレーションの WebLogic Server を使用した接続	3-4
多層コンフィグレーションの WebLogic Server を使用した接続	3-5
接続のサンプル	3-6
DataSource オブジェクトを用いた接続	3-6
WebLogic JDBC を使用するためのプロパティ設定	3-7
Connection オブジェクトについて	3-7
自動コミットの設定	3-7
簡単な SQL クエリの作り方	3-8
レコードの挿入、更新、および削除	3-9
ストアード プロシージャおよび関数の作り方と使い方	3-10
接続の切断とオブジェクトのクローズ	3-13
ストアード プロシージャからの ResultSets の処理	3-14
WebLogic JDBC による行キャッシング	3-14
コード例	3-14
未サポートの JDBC 2.0 メソッド	3-18

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

WebLogic jDriver for Oracle の XA モードと非 XA モードの違い	4-2
JDBC XA および非 XA リソースのコンフィグレーション	4-3
JDBC/XA リソース	4-3
XA 非対応の JDBC リソース	4-3

WebLogic jDriver for Oracle XA の制限	4-4
分散トランザクションの実装	4-4
パッケージをインポートする	4-4
JNDI を介したデータソースの検索	4-5
分散トランザクションの実行	4-6

5. Oracle の高度な機能

大文字 / 小文字を区別せずにメタデータを扱う方法	5-2
データ型	5-2
WebLogic Server と Oracle の NUMBER カラム	5-4
Oracle の Long raw データ型の使い方	5-5
Oracle リソース上の待機	5-5
自動コミット	5-6
トランザクションのアイソレーション レベル	5-7
コードセットのサポート	5-7
Oracle 配列フェッチのサポート	5-10
ストアド プロシージャの使い方	5-11
Oracle カーソルへのパラメータのバインド	5-11
CallableStatement の使用上の注意	5-13
DatabaseMetaData メソッド	5-13
JDBC 拡張 SQL のサポート	5-14
Oracle 用 JDBC 2.0 の概要	5-15
JDBC 2.0 のサポートに必要なコンフィグレーション	5-16
BLOB と CLOB	5-16
トランザクション境界	5-17
BLOB	5-17
Connection プロパティ	5-17
Import 文	5-18
BLOB フィールドの初期化	5-18
BLOB へのバイナリ データの書き込み	5-19
BLOB オブジェクトの書き込み	5-20
Prepared Statement を使用した CLOB 値の更新	5-20
BLOB データの読み取り	5-21
その他のメソッド	5-22
CLOB	5-22

コードセットのサポート	5-22
CLOB フィールドの初期化.....	5-23
CLOB への ASCII データの書き込み	5-24
CLOB への Unicode データの書き込み	5-24
CLOB オブジェクトの書き込み	5-25
CLOB データの読み取り	5-26
その他のメソッド	5-27
文字と ASCII ストリーム	5-27
Unicode 文字ストリーム.....	5-27
ASCII 文字ストリーム.....	5-28
バッチ更新.....	5-28
バッチ更新の使い方	5-29
バッチ処理文の消去	5-29
更新件数	5-29
新しい日付関連メソッド	5-30

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 のインストール」
- 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) を使った 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を対象としています。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 ホームページの [製品のドキュメント] をクリックするか、WebLogic Server 製品ドキュメント ページ (<http://edocs.beasys.co.jp/e-docs/wls61>) 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のメイン トピックを一度に 1 つずつ印刷できます。

このマニュアルの PDF 版は、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>) から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@bea.com までお送りください。寄せられた意見については、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名とバージョン名、およびマニュアルのタイトルと作成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 }	同時に押すキーを示す。
斜体	強調または本のタイトルを示す。
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、Java クラス、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： <pre>import java.util.Enumeration; chmod u+w * config/examples/applications .java config.xml float</pre>
斜体の等幅テキスト	コード内の変数を示す。 例： <pre>String CustomerName;</pre>
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： <pre>LPT1 BEA_HOME OR</pre>
{ }	構文内の複数の選択肢を示す。

表記法	適用
[]	構文内の任意指定の項目を示す。 例： <pre>java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]</pre>
	構文の中で相互に排他的な選択肢を区切る。 例： <pre>java weblogic.deploy [list deploy undeploy update] password {application} {source}</pre>
...	コマンドラインで以下のいずれかを示す。 <ul style="list-style-type: none"> ■ 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 ドライバをインストールして、WebLogic Server で使用する方法について説明します。WebLogic Server バージョン 6 で追加された新機能であるマルチプール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 では、WebLogic Server ソフトウェアで使用する 3 つの 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
- Informix 用と 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。

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。このドライバをインストールする前に、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 NT) または共有ライブラリパス (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 のインストール」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 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 のインストール

この章では次の内容について説明します。

- WebLogic jDriver for Oracle のインストール準備
- WebLogic jDriver for Oracle のインストール手順
- WebLogic jDriver for Oracle のインストール後の作業

WebLogic jDriver for Oracle のインストール準備

WebLogic jDriver for Oracle をインストールする前に、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
- 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、「[Weblogic jDriver JDBC ドライバのプラットフォーム サポート](#)」を参照してください。

JDBC 2.0 の動作要件

WebLogic Server 6.1 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。

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。

- ドライバが入っているディレクトリ (ドライバ ファイルは `dll`、`so`、または `sl` ファイル)。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 NT 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。

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\bin\` と適切な WebLogic Server 共有ライブラリ (以下の表参照) を `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%\bin\oci
```

- `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%\bin\oci817_8;c:\ORANT817\bin;%PATH%
```

`oci817_8` は `oci817_8`、`ORACLE_HOME` は `c:\ORANT817` と表されます。

Oracle 9.0.1 では、パスは次のようになります。

```
$set PATH=%WL_HOME%\bin\oci901_8;c:\ORANT901\bin;%PATH%
```

次の表は、ディレクトリと Windows NT 用 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 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 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、Oracle クライアントをインストールしたディレクトリと、ネイティブ インタフェース ファイルが入っているディレクトリを、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 `libweblogicocixx.so` が入っているディレクトリ。次に例を示します。

```
$WL_HOME/lib/solaris/oci
```

- 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が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=
$WL_HOME/lib/solaris/oci817_8:ORACLE/lib:$LD_LIBRARY_PATH
```

ocixx は oci817_8、ORACLE_HOME は /ORACLE/ と表されます。

次の表は、ディレクトリと Solaris 用 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。
9.2.0	8	oci920_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。

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、「[BEA WebLogic Server プラットフォーム サポート](#)」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、Oracle クライアントをインストールしたディレクトリと、ネイティブ インタフェース ファイルが入っているディレクトリを、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、HP-UX 用の 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。

- ネイティブ インタフェース ファイル `weblogicocixx.sl` が入っているディレクトリ。次に例を示します。

```
$WL_HOME/lib/hpux11/ocixx
```

- 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 ライブラリが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。

- ネイティブ インタフェース ファイル `weblogicocixx.sl` が入っているディレクトリ。次に例を示します。

```
$WL_HOME/lib/hpux11/ocixx
```

- 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が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=  
$WL_HOME/lib/hpux11/oci817_8:ORACLE/lib:$SHLIB_PATH
```

ocixx は oci817_8、ORACLE_HOME は ORACLE と表されます。

Oracle 9.0.1 では、パスは次のようになります。

```
export SHLIB_PATH=  
$WL_HOME/lib/hpux11/oci901_8:ORACLE/lib32:$SHLIB_PATH
```

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

表 2-3 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 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。

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、[「BEA WebLogic Server プラットフォーム サポート」](#)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、[「BEA WebLogic Server プラットフォーム サポート」](#)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、「[BEA WebLogic Server プラットフォーム サポート](#)」を参照してください。

WebLogic jDriver for Oracle のインストール手順

このマニュアルの手順では、WebLogic Server を Microsoft Windows オペレーティング システムが稼働しているコンピュータにインストールし、アーカイブを C ドライブのルート ディレクトリに復元することを前提にしています。オペレーティング システムまたはディレクトリがこの前提と異なる場合は、システムのマニュアルを参照して、手順の中のコマンドを環境に合わせて変更する方法を確認してください。

次のインストール手順は、Oracle および WebLogic Server が動作するサーバ上で WebLogic jDriver for Oracle をインストールする場合と、スタンドアロンの type-2 JDBC ドライバとして (WebLogic Server から独立して) WebLogic jDriver for Oracle をインストール場合の両方の手順です。WebLogic Server でのインストールの場合には手順 1 および 2 を、WebLogic Server を使用してアプレット経由でデータベースにアクセスする場合には手順 5 に従います。スタンドアロンのインストールの場合には、すべての手順に従います。

1. 環境に合わせて適切なネイティブ ライブラリを選択します。

WebLogic Server には、さまざまな Oracle クライアント バージョン、API、およびプラットフォーム用の dll、s1、または so ファイルが付属しています。適切なファイルをシステムパスまたは共有ライブラリパスに入れておかないと、このドライバは機能しません。デフォルトパス上に常にファイルが 1 つだけあった旧バージョンの WebLogic Server とは違い、現バージョンではパスを明示的に設定する必要があります。これらのオプションの設定の詳細については、「[WebLogic jDriver for Oracle の使用環境の設定](#)」を参照してください。

2. Oracle のインストール先からクライアント ライブラリをシステム PATH (Windows NT) またはロード ライブラリ パス (UNIX の場合。この変数の名前は UNIX システムによって異なります) に追加します。これらのライブラリの場所については、Oracle のマニュアルを参照してください。Windows NT プラットフォームでは、通常、`c:\ORANT\bin` にあります。

3. 配布キットを復元します。

WebLogic Server をすでに購入されている場合、WebLogic jDriver for Oracle は配布キットに含まれています。配布キットを復元するために、これ以上の手順は不要です。これ以降のインストール手順は、スタンドアロンバージョンの WebLogic jDriver のユーザの場合にのみ必要となります。

ダウンロードしたアーカイブをコンピュータのルート ディレクトリに復元していない場合は、ここで復元します。アーカイブ内のディレクトリ構造を再現するには、[WinZip](#) などのプログラムを使用しなければなりません。ファイルは、`weblogic` ディレクトリに復元されます。

4. WebLogic Server クラスのディレクトリを CLASSPATH 変数に追加します。

注意： WebLogic Server ソフトウェアに付属の WebLogic jDriver for Oracle を使用している場合は、CLASSPATH の設定は WebLogic Server 用の設定と同じです。WebLogic Server 配布キットに付属の標準スタートアップスクリプトは、この処理を実行します。

CLASSPATH を一時的に変更するには、Windows コマンド プロンプトで、次のコマンドを使います。

```
$set CLASSPATH=%CLASSPATH%;c:\weblogic\oracle\classes
```

CLASSPATH を永久に変更するには、次の手順に従います。

- a. [コントロール パネル] で、[システム] アイコンをダブルクリックします。
- b. [環境] タブをクリックします。
- c. 下側のパネルで、CLASSPATH 変数を選択します。CLASSPATH 変数が定義されていない場合は、追加します。
- d. `%WL_HOME%\oracle\classes` を CLASSPATH 変数に追加します。すでに変数の値があれば、セミコロン (;) を使って新しいパスと分けます。

[システム] コントロール パネルで CLASSPATH 変数を更新した後に起動したプログラムは、新しい値にアクセスします。

CLASSPATH の設定の詳細については、『WebLogic Server アプリケーションの開発』の「[WebLogic Server J2EE アプリケーションの開発](#)」を参照してください。Symantec Cafe や JBuilder のような IDE 内から WebLogic Server を使用している場合は、CLASSPATH にクラスを追加する手順が異なります。

5. ライセンス ファイルをインストールします。

WebLogic jDriver for Oracle のインストール後の作業

この節では、WebLogic jDriver for Oracle ソフトウェアをインストールした後に必要な作業について説明します。

- ライセンス機能
- Oracle データベースとの接続の確認
- 接続プールの設定

ライセンス機能

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
```

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。

dbping を使用する環境を設定するには、次のコマンドラインを 1 行で入力します。

```
%WL_HOME%\config\mydomain\setEnv.cmd  
java utils.dbping ORACLE user password server
```

%WL_HOME% を、WebLogic Server または WebLogic jDriver for Oracle のインストール先ディレクトリと置き換えます。

DBMS との接続を確認する手順の詳細については、「接続のテスト」を参照してください。

接続できない場合は、『WebLogic JDBC プログラミング ガイド』の「[JDBC の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、WebLogic Pool または RMI ドライバなどの多層 (Type 3) JDBC ドライバを使用して、WebLogic Server ソフトウェアに接続します。

WebLogic Server ソフトウェアは、WebLogic Server とプールの中の 1 つの接続を使用して、アプリケーションの代わりに Oracle データベースに接続します。

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。

- 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「[接続プール](#)」またはオンラインヘルプの「[JDBC 接続プールのコンフィグレーション](#)」の手順を参照してください。
- WebLogic Server ソフトウェアを起動します。

アプリケーションでの接続プールの使い方

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	WebLogic JDBC/RMI ドライバ	『WebLogic JDBC プログラミング ガイド』の「 WebLogic 多層 JDBC ドライバの使い方 」および「 クラスタ化された JDBC の概要 」
サーバサイド（サーブレットとして使用）	WebLogic Pool、JTS、および WebLogic jDriver for Oracle/XA ドライバ	『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 」 『管理者ガイド』の「 JDBC 接続の管理 」

WebLogic jDriver での IDE またはデバッガの使い方

Symantec Cafe、その他の 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 ライブラリを新しいファイルにコピーし、（拡張子を除いて）_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、libweblogicoci37.so を libweblogicoci37_g.so にコピーします。分散トランザクションの場合は、libweblogicoxa37.so を libweblogicoxa37_g.so にコピーします。

- Windows NT プラットフォームでは、weblogicoci37.dll を weblogicoci37_g.dll にコピーします。分散トランザクションの場合は、weblogicoxa37.dll を weblogicocoxa37_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。

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「 WebLogic Server J2EE アプリケーションの開発 」
ドライバの使い方	「 WebLogic jDriver for Oracle の使い方 」

3 WebLogic jDriver for Oracle の使い方

この節では、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、最後にサンプルコードを示します。

- 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
- 未サポートの 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 - 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 - 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%\lib\weblogic.jar
```

(%WL_HOME% は WebLogic Server のインストール先ディレクトリです)

スタンドアロンバージョンの WebLogic Server を実行している場合は、CLASSPATH に次のパス名を指定します。

```
%WL_HOME%\oci\classes
```

%WL_HOME% を、WebLogic Server のインストール先ディレクトリの名前と置き換えます。

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。

2 層コンフィギュレーションの WebLogic Server を使用した接続

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、2-11 ページの「WebLogic jDriver for Oracle のインストール後の作業」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 の 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指定します。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
}
```

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);
```

DataSource オブジェクトを用いた接続

DataSource オブジェクトを使用して接続することもできます。詳細については、『WebLogic JDBC プログラミング ガイド』の「[DataSources の概要](#)」を参照してください。

WebLogic JDBC を使用するためのプロパティ設定

WebLogic Server のドライバと一緒に多層環境内で WebLogic JDBC を使用する場合は、若干異なる方法で接続プロパティを設定することになります。詳細については、『WebLogic JDBC プログラミング ガイド』を参照してください。

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（エイリアス名 `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。

PreparedStatement (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();
```

ストアード 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、ストアード プロシージャとストアー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、ストアード プロシージャとストアード関数の作成、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、ストアード プロシージャとストアード関数を実行できます。ただし、ストアード プロシージャとストアー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、ストアード プロシージャとストアー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 を使用してストアード プロシージャまたはストアー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 文にどちらかの構文を使用できます。

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();
```

この次の例では、`sp_getmessages` というストアード プロシージャを使用します (このストアード 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)。このストアード 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 `ResultSet` に格納して返します。 `Statement.execute()` および `Statement.getResult()` メソッドを使ってストアード プロシージャから返された `ResultSet` 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) はストアード 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、ストアード 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。

配列フェッチの詳細については、[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

以下に示すコードは、JDBC アプリケーションの全体的な構造がわかるように、これまでのサンプルで使われたコードを抜き出したものです。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.util.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。

未サポートの JDBC 2.0 メソッド

WebLogic Server は JDBC 2.0 のすべてのメソッドをサポートしますが、WebLogic jDriver for Oracle では JDBC 2.0 の一部のメソッドはサポートされません。そのようなメソッドを使う必要がある場合、Oracle Thin Driver などの別の JDBC ドライバを使用してデータベースに接続できます。表 3-2 に、WebLogic jDriver for Oracle でサポートされていない JDBC 2.0 メソッドの一覧を示します。

表 3-2 WebLogic jDriver for Oracle で未サポートの JDBC 2.0 メソッド

クラスまたはインターフェイス	未サポートのメソッド
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)
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 で未サポートの JDBC 2.0 メソッド

クラスまたはインターフェイス	未サポートのメソッド
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 で未サポートの JDBC 2.0 メソッド

クラスまたはインターフェイス	未サポートのメソッド
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 で未サポートの JDBC 2.0 メソッド

クラスまたはインターフェイス	未サポートのメソッド
java.sql.ResultSet (続き)	<pre> 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) </pre>
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 オブジェクト 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。
- 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、`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、`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。

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。
- 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』を参照してください。サードパーティ製ドライバのデータソース プロパティについては、そのベンダのドキュメント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。

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。

- パッケージをインポートする
- 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

パッケージをインポートする

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、
        // and prepare it for handling a transaction.
        data1 = (javax.sql.DataSource)
            initCtx.lookup("java:comp/env/jdbc/DataBase1");
        conn1 = data1.getConnection();

        stat1 = conn1.createStatement();

        // 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.createStatement();

        // Update both conn1 and conn2. The EJB Container
        // 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、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

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.forName(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せん。

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。

図 5-1 Oracle と WebLogic Server の型の対応表

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らず、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。

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のですが、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、短時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` 内の値を調べて、変換に使用する JDK のコードセットを選択します。コードセットのマッピングには、表 5-1 が使用されます。

`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 では、Java コード内からコードセットを設定できます。接続プロパティの `weblogic.codeset` を設定すると、`NLS_LANG` 環境変数内の値をオーバーライドできます。たとえば、次のサンプルのように、`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。

表 5-1 NLS_LANG 設定と JDK コードセットのマッピング対応表

NLS_LANG	JDK codeset
al24utfss	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 内のカラムが LONG、BLOB、または CLOB 型の場合、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、ストアー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ません)。
- 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っ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。

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 6.1 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、「[WebLogic プラットフォーム サポート](#)」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。

注意： 次の制限に注意してください。WebLogic jDriver for Oracle と組み合わせる RMI ドライバを使用する場合、BLOB と CLOB を使用することはできません。BLOB および CLOB はシリアライズできないため、WebLogic Server 6.x で使用する JDBC RMI ドライバではサポートされません。

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 は対象となる connection 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 ストリームヘ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();
```

- 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。

- 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.prepareStatement(
    "UPDATE myOtherTable SET myOtherBlobColumn = ? WHERE id = 12");
pstmt.setBlob(1, myBlob);
```

Prepared Statement を使用した CLOB 値の更新

Prepared Statement を使用して CLOB を更新する場合、新しい値が古い値より短いと、更新の際に明示的に置換されなかった文字が CLOB に残ります。たとえば、現在の値が `abcdefghij` である CLOB を Prepared Statement で `zxyw` という値に更新すると、更新後の CLOB の値は `zxywefghij` になります。Prepared

Statement で更新した結果を正しい値にするには、`dbms_lob.trim` プロシージャを使って、更新後に残っている余分な文字を削除する必要があります。`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。

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.CallableStatement` 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。このコードセットは、Oracle 環境変数 `NLS_LANG` に指定されたコードセットの値をオーバーライドします。

`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。このコードセットは、Oracle 環境変数 `NLS_NCHAR` に指定されたコードセットの値をオーバーライドします。

```
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。

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。

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.prepareStatement(  
    "UPDATE myOtherTable SET myOtherClobColumn = ? WHERE id = 12");  
  
pstmt.setClob(1, myClob);
```

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 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.CallableStatement 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)
```

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