BEA WebLogic Server™ and WebLogic Express®

WebLogic JDBC プログラマーズ ガイド

BEA WebLogic Server バージョン 7.0
マニュアルの改訂 : 2004 年 10 月 13 日

著作権

Copyright © 2002, 2003 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 Commerce Server、BEA WebLogic Enterprise、BEA WebLogic Enterprise Platform、BEA WebLogic Express、BEA WebLogic Integration、BEA WebLogic Personalization Server、BEA WebLogic Platform、BEA WebLogic Portal、BEA WebLogic Server、BEA WebLogic Workshop および How Business Becomes E-Business 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 JDBC プログラマーズ ガイド

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	2004 年 4 月 8 日	BEA WebLogic Server バージョン 7.0

目次

このマニュアルの内容

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

1. WebLogic JDBC の概要

JDBC の概要 1-1
JDBC ドライバと WebLogic Server の使用 1-2
JDBC ドライバのタイプ 1-2
WebLogic Server JDBC ドライバの表 1-3
WebLogic Server 2 層 JDBC ドライバ 1-4
WebLogic jDriver for Oracle 1-4
WebLogic jDriver for Microsoft SQL Server 1-4
WebLogic Server JDBC 多層ドライバ 1-5
WebLogic RMI ドライバ 1-5
WebLogic Pool ドライバ 1-6
WebLogic JTS ドライバ 1-6
サードパーティ ドライバ 1-6
Sybase jConnect ドライバ 1-7
Oracle Thin ドライバ 1-7
接続プールの概要 1-7
サーバサイド アプリケーションでの接続プールの使い方 1-9
クライアントサイド アプリケーションでの接続プールの使い方 1-10
マルチプールの概要 1-10
クラスタ化された JDBC の概要 1-11
DataSource の概要 1-11
JDBC API 1-12
JDBC 2.0 1-12

プラットフォーム..1-12

2. WebLogic JDBC のコンフィグレーションと管理

接続プールのコンフィグレーションと使い方 2-2
接続プールを使用するメリット 2-2
起動時の接続プールの作成 2-3
正しい接続数によるサーバのロックアップの回避..........................2-3
接続プールのコンフィグレーションにおけるデータベース パスワード..2-4
接続プールの制限事項 2-5
JDBC 接続プールの接続の更新に関する注意事項 2-6
JDBC 接続プールのテスト機能の強化..2-6
データベース接続消失後の接続テストでの遅延を最小限に抑える . . 2-7
接続テスト失敗後の接続要求の遅延を最小限に抑える 2-8
secondsToTrustAnIdlePoolConnection を使用して接続要求の遅延を最小限に抑える 2-9
接続プールの動的作成 2-10
動的接続プールのサンプル コード...2-11
パッケージをインポートする 2-12
管理 MBeanHome をルックアップする 2-12
サーバ MBean を取得する 2-12
接続プール MBean を作成する...2-12
接続プール プロパティを設定する 2-12
対象を追加する 2-13
DataSource を作成する 2-13
動的接続プールと DataSource を削除する 2-14
接続プールの管理 2-15
プールに関する情報の取得...2-15
接続プールの無効化 2-16
接続プールの縮小 2-16
接続プールの停止 2-17
プールのリセット 2-17
weblogic.jdbc.common.JdbcServices と weblogic.jdbc.common.Pool クラス (非推奨) の使用 2-18
アプリケーション スコープの JDBC 接続プール......................................2-19

マルチプールのコンフィグレーションと使い方 2-20
マルチプールの機能 2-21
マルチプール アルゴリズムの選択 2-21
高可用性 2-21
ロード バランシング 2-22
マルチプールのフェイルオーバの拡張 2-22
接続プールが失敗したときの接続要求の転送の改良 2-23
マルチプール内の失敗した接続プールが回復するときの自動的な再有効化 2-23
マルチプールの使用されている接続プールのフェイルオーバの有効化 2-24
コールバックによるマルチプールのフェイルオーバの制御 2-25
コールバックによるマルチプールのフェイルバックの制御 2-29
マルチプール フェイルオーバの制限事項と要件 2-31
フェイルオーバを有効にするための予約時の接続のテスト 2-31
使用中の接続ではフェイルオーバは行われない 2-31
DataSource のコンフィグレーションと使い方 2-32
DataSource オブジェクトにアクセスするパッケージのインポート....2-33
DataSource を使用したクライアント接続の取得 2-33
コード例 2-34
JDBC データ ソース ファクトリ 2-34

3. JDBC アプリケーションのパフォーマンス チューニング

JDBC パフォーマンスの概要 3-1
WebLogic のパフォーマンス向上機能 3-1
接続プールによるパフォーマンスの向上 3-2
Prepared Statement とデータのキャッシング..3-2
ベスト パフォーマンスのためのアプリケーション設計 3-3
1. データをできるだけデータベースの内部で処理する 3-3
2. 組み込み DBMS セットベース処理を使用する 3-4
3. クエリを効率化する 3-4
4. トランザクションを単一バッチにする 3-6
5. DBMS トランザクションがユーザ入力に依存しないようにする 3-7
6. 同位置更新を使用する...3-8
7. 操作データをできるだけ小さくする 3-8

8. パイプラインと並行処理を使用する 3-9

4. WebLogic 多層 JDBC ドライバの使い方

WebLogic RMI ドライバの使い方 4-1
WebLogic RMI ドライバを使用するための WebLogic Server の設定...4-2
RMI ドライバを使用するサンプル クライアント コード 4-2
必要なパッケージをインポートする..4-3
データベース接続を取得する 4-3
JNDI ルックアップを使用した接続の取得......................................4-3
WebLogic RMI ドライバだけを使用して接続を取得する..............4-4
WebLogic RMI ドライバによる行キャッシング 4-5
WebLogic RMI ドライバによる行キャッシングの重要な制限事項 . . 4-6
WebLogic JTS ドライバの使い方 4-7
JTS ドライバを使用するサンプル クライアント コード........................4-8
WebLogic Pool ドライバの使い方 4-10

5. WebLogic Server でのサードパーティ ドライバの使い方

サードパーティ JDBC ドライバの概要 5-1
サードパーティの JDBC ドライバに対する環境設定...................................5-4
Windows でのサードパーティ JDBC ドライバの CLASSPATH 5-4
UNIX でのサードパーティ JDBC ドライバの CLASSPATH.................5-5
Oracle Thin Driver の変更または更新..5-5
Oracle Thin Driver 9.x および 10g でのパッケージの変更 5-6
nls_charset12.zip による文字セットのサポート 5-7
Sybase jConnect Driver の更新 5-7
IBM Infomix JDBC Driver のインストールと使い方 5-8
IBM Infomix JDBC Driver 使用時の接続プール属性........................5-9
IBM Infomix JDBC Driver のプログラミング上の注意 5-11
Microsoft SQL Server 2000 Driver for JDBC のインストールと使い方 . . . 5-11
Microsoft SQL Server Driver for JDBC の Windows システムへのインストール 5-12
Microsoft SQL Server Driver for JDBC の UNIX システムへのインストール 5-12
Microsoft SQL Server Driver for JDBC 使用時の接続プール属性.5-13

サードパーティ ドライバを使用した接続の取得 5-14
サードパーティ ドライバでの接続プールの使い方 5-14
接続プールと DataSource の作成 5-14
JNDI を使用した接続の取得...5-15
接続プールからの物理的な接続の取得 5-16
物理的な接続を取得するサンプル コード 5-17
物理的な接続を使用する際の制限事項...5-19
Oracle 拡張機能と Oracle Thin Driver の使用 5-19
Oracle JDBC 拡張機能の使用時の制限 5-20
Oracle 拡張機能から JDBC インタフェースにアクセスするサンプル コード...5-21
Oracle 拡張機能へアクセスするパッケージをインポートする 5-21
接続を確立する 5-22
デフォルトの行プリフェッチ値を取得する 5-22
ARRAY によるプログラミング..5-23
ARRAY を取得する 5-24
データベースで ARRAY を更新する 5-24
Oracle Array 拡張機能メソッドを使用する 5-25
STRUCT によるプログラミング 5-25
STRUCT を取得する 5-26
OracleStruct 拡張機能メソッドを使用する...................................5-27
STRUCT 属性を取得する 5-27
STRUCT によってデータベース オブジェクトを更新する 5-29
データベース オブジェクトを作成する..5-29
STRUCT 属性を自動バッファリングする 5-30
REF によるプログラミング 5-30
REF を取得する 5-31
OracleRef 拡張機能メソッドを使用する 5-32
値を取得する..5-32
REF 値を更新する 5-33
データベースで REF を作成する 5-35
BLOB と CLOB によるプログラミング 5-36
DBMS から BLOB ロケータを選択するクエリを実行する 5-36
WebLogic Server java.sql オブジェクトを宣言する 5-36
SQL 例外ブロックを開始する 5-36

PreparedStatement を使用した CLOB 値の更新 5-37
Oracle 仮想プライベート データベースによるプログラミング 5-37
Oracle 拡張機能インタフェースとサポートされるメソッドの表 5-39

6. dbKona (非推奨) の使い方

dbKona の概要 6-1
多層コンフィグレーションでの dbKona..6-2
dbKona と JDBC ドライバの相互作用 6-2
dbKona と WebLogic Event の相互作用 6-3
dbKona アーキテクチャ 6-3
dbKona API 6-4
dbKona API リファレンス 6-4
dbKona オブジェクトとそれらのクラス 6-5
dbKona のデータ コンテナ オブジェクト.......................................6-5
DataSet 6-6
QueryDataSet 6-6
TableDataSet 6-7
EventfulTableDataSet (非推奨) 6-10
Record 6-12
Value 6-13
dbKona のデータ記述オブジェクト 6-14
Schema 6-14
Column...6-15
KeyDef 6-15
SelectStmt 6-16
dbKona のその他オブジェクト 6-17
例外 6-17
定数 6-17
エンティティの関係..6-18
継承関係 6-18
所有関係 6-18
dbKona の実装 6-19
dbKona を使用した DBMS へのアクセス 6-19
手順 1. パッケージのインポート 6-19
手順 2. 接続確立用のプロパティの設定...6-20

手順 3. DBMS との接続の確立 6-20
クエリの準備、およびデータの検索と表示 6-21
手順 1. データ検索用のパラメータの設定....................................6-21
手順 2. クエリ結果用の DataSet の生成 6-22
手順 3. 結果の取り出し 6-23
手順 4. TableDataSet の Schema の検査 6-24
手順 5. htmlKona を使用したデータの検査...................................6-24
手順 6. htmlKona を使用した結果の表示.......................................6-25
手順 7. DataSet および接続のクローズ 6-26
SelectStmt オブジェクトを使用したクエリの作成.............................6-28
手順 1. SelectStmt パラメータの設定 6-28
手順 2. QBE を使用したパラメータの修正 6-29
SQL 文を使用した DBMS データの変更 6-29
手順 1. SQL 文の記述 6-30
手順 1. SQL 文の記述 6-30
手順 2. 各 SQL 文の実行 6-30
手順 3. htmlKona を使用した結果の表示.......................................6-31
KeyDef を使用した DBMS データの変更..6-34
手順 1. KeyDef とその属性の作成...6-34
手順 2. KeyDef を使用した TableDataSet の作成..........................6-35
手順 3. TableDataSet へのレコードの挿入.....................................6-35
手順 4. TableDataSet でのレコードの更新.....................................6-36
手順 5. TableDataSet からのレコードの削除.................................6-36
手順 6. TableDataSet の保存の詳細 6-37
保存前の Record 状態の確認..6-37
手順 7. 変更内容の検証 6-38
コードのまとめ..6-38
dbKona での JDBC PreparedStatement の使い方 6-40
dbKona でのストアドプロシージャの使い方 6-41
手順 1. ストアドプロシージャの作成 6-41
手順 2. パラメータの設定 6-42
手順 3. 結果の検査...6-42
画像およびオーディオ用バイト配列の使い方 6-42
手順 1. 画像データの検索と表示..6-43
手順 2. データベースへの画像の挿入 6-43

Oracle シーケンス用の dbKona の使い方 6-44
手順 1. dbKona Sequence オブジェクトの作成 6-44
手順 2. dbKona からの Oracle サーバのシーケンスの作成と破棄 6-44
手順 3. Sequence の使い方 6-45
コードのまとめ 6-45

7. JDBC 接続のテストとトラブルシューティング

JDBC 接続のモニタ 7-1
コマンドラインからの DBMS 接続の有効性の検証 7-2
構文 7-2
引数 7-2
サンプル 7-3
JDBC のトラブルシューティング 7-4
JDBC 接続 7-4
Windows 7-4
UNIX 7-5
コードセットのサポート 7-5
UNIX での Oracle に関わる他の問題 7-5
UNIX でのスレッド関連の問題 7-6
JDBC オブジェクトを閉じる 7-7
JDBC オブジェクトの破棄 7-7
UNIX での共有ライブラリに関連する問題のトラブルシューティング 7-8
WebLogic jDriver for Oracle 7-8
Solaris 7-9
HP-UX 7-9
不適切なファイル パーミッションの設定 7-9
不適切な SHLIB_PATH 7-10

このマニュアルの内容

このマニュアルでは、WebLogic Server™ における JDBC の使い方について説明します。

このマニュアルの構成は次のとおりです。

- 第 1 章「WebLogic JDBC の概要」では、JDBC コンポーネントと JDBC API の概要について説明します。
- 第 2 章「WebLogic JDBC のコンフィグレーションと管理」では、WebLogic Server Java アプリケーションでの JDBC の使い方について説明します。
- 第 3 章「JDBC アプリケーションのパフォーマンス チューニング」では、JDBC アプリケーションから最高のパフォーマンスを得る方法について説明します。
- 第 4 章「WebLogic 多層 JDBC ドライバの使い方」では、WebLogic Server を使用するための WebLogic RMI ドライバおよび JDBC クライアントの設定方法について説明します。
- 第 5 章「WebLogic Server でのサードパーティ ドライバの使い方」では、WebLogic Server でサードパーティ製ドライバを設定および使用する方法について説明します。
- 第 6 章「dbKona (非推奨) の使い方」では、アプリケーションで dbKone クラスを使用する方法について説明します。
- 第 7 章「JDBC 接続のテストとトラブルシューティング」では、WebLogic Server で JDBC を使用する際のトラブルシューティングのヒントを紹介します。

対象読者

このマニュアルは、Sun Microsystems, Inc. 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を対象としています。Web 技術、オブジェクト指向プログラミング技術、および Java プログラミング言語に読者が精通していることを前提として書かれてます。

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] をクリックします。

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。JDBC の詳細については、Sun Microsystems Javasoft Web サイトにある JDBC セクションを参照してください。

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 WebLogic JDBC の概要

以下の節では JDBC コンポーネントと JDBC API の概要について説明します。

- 1-1 ページの「JDBC の概要」
- 1-2 ページの「JDBC ドライバと WebLogic Server の使用」
- 1-7 ページの「接続プールの概要」
- 1-10 ページの「マルチプールの概要」
- 1-11 ページの「クラスタ化された JDBC の概要」
- 1-11 ページの「DataSource の概要」
- 1-12 ページの「JDBC API」
- 1-12 ページの「JDBC 2.0」
- 1-12 ページの「プラットフォーム」

JDBC の概要

Java Database Connectivity (JDBC) とは、Java プログラミング言語で作成された一連のクラスとインタフェースで構成された標準 Java API のことです。アプリケーション、ツール、およびデータベースの開発者は、JDBC を使用することにより、データベース アプリケーションを作成したり、SQL 文を実行したりできます。

JDBC は低レベルインタフェースであり、SQL コマンドを直接起動する (呼び出す) ために使用されます。 また、JDBC は Java Messaging Service (JMS) や Enterprise JavaBean (EJB) などの高レベルのインタフェースとツールを構築するための基盤でもあります。

JDBC ドライバと WebLogic Server の使用

JDBC ドライバは、JDBC API のインタフェースとクラスを実装します。以下の節では、WebLogic Server で使用できる JDBC ドライバ オプションについて説明します。

JDBC ドライバのタイプ

WebLogic Server は、以下のタイプの JDBC ドライバを使用します。これらのドライバが互いに連携することによって、データベース アクセスが提供されます。

- 2 層ドライバー接続プールとデータベース間の直接的なデータベース アクセスを提供します。WebLogic Server は、DBMS ベンダ固有の JDBC ドライバ (WebLogic jDriver for Oracle、および WebLogic jDriver for Microsoft SQL Server など) を使用してバックエンド データベースに接続します。
- 多層ドライバ - ベンダに依存しないデータベース アクセスを提供します。Java クライアント アプリケーションは多層ドライバを使用して、WebLogic Server でコンフィグレーションされた任意のデータベースにアクセスできます。BEA は、RMI、Pool、および JTS という 3 種類の多層ドライバを提供しています。WebLogic Server システムでは、JNDI ルックアップを使用して接続プールとデータ ソースを接続するときに、その背後でこれらのドライバを使用します。

WebLogic Server の中間層アーキテクチャ (データ ソース、および接続プール) により、データベース リソースを集中管理できます。ベンダに依存しない多層 JDBC ドライバを使用すれば、購入したコンポーネントを自社の DBMS 環境により簡単に適合させ、より移植性の高いコードを記述できます。

WebLogic Server JDBC ドライバの表

次の表に、WebLogic Server で使用するドライバの一覧を示します。

表 1-1 JDBC ドライバ

ドライバ層	ドライバのタイプと名前	データベース接続性	ドキュメント ソース
2 層 分散トランザクションのサポートなし (XA 非対応)	Type 2 (ネイティブ ライブラリが必要) ■ WebLogic jDriver for Oracle ■ サードパーティ ドライバ Type 4 (pure Java) ■ WebLogic jDriver for Microsoft SQL Server ■ 以下のものを含むサードパーティ ドライバ Oracle Thin Sybase jConnect	ローカル トランザクションでの WebLogic Server と DBMS 間	『WebLogic JDBC プログラマーズ ガイド』(このマニュアル) 『管理者ガイド』の「JDBC 接続の管理」 『WebLogic jDriver for Oracle のコンフィグレーションと使い方』 『WebLogic jDriver for Microsoft SQL Server のコンフィグレーションと使い方』
2 層 分散トランザクションのサポート付き (XA 非対応)	Type 2 (ネイティブ ライブラリが必要) ■ WebLogic jDriver for Oracle XA	分散トランザクションでの WebLogic Server と DBMS 間	『WebLogic JTA プログラマーズ ガイド』 『管理者ガイド』の「JDBC 接続の管理」 『WebLogic jDriver for Oracle のコンフィグレーションと使い方』

表 1-1 JDBC ドライバ

ドライバ層	ドライバのタイプと名前	データベース接続性	ドキュメント ソース
多層	Type 3 ■ WebLogic RMI ドライバ ■ WebLogic Pool ドライバ ■ WebLogic JTS (Type 3 以外)	クライアントと WebLogic Server (接続プール) 間。RMI ドライバは非推奨の t3 ドライバの代わりに使用される。JTS ドライバは分散トランザクションで使用される。Pool および JTS ドライバはサーバサイドのみ。	『WebLogic JDBC プログラマーズ ガイド』(このマニュアル)

WebLogic Server 2 層 JDBC ドライバ

以下の節では、ベンダ固有の DBMS に接続するために WebLogic Server で使用される Type 2 および Type 4 の BEA 2 層ドライバについて説明します。

WebLogic jDriver for Oracle

BEA の WebLogic jDriver for Oracle WebLogic Server の配布キットの中に入っています。このドライバを使用するには、Oracle クライアントがインストールされている必要があります。WebLogic jDriver for Oracle XA ドライバは、WebLogic jDriver for Oracle を分散トランザクション用に拡張します。詳細については、『WebLogic jDriver for Oracle のコンフィグレーションと使い方』を参照してください。

WebLogic jDriver for Microsoft SQL Server

WebLogic Server 配布キットに同梱の BEA WebLogic jDriver for Microsoft SQL Server は、Microsoft SQL Server への接続を提供する pure-Java、Type 4 JDBC ドライバです。詳細については、『WebLogic jDriver for Microsoft SQL Server のコンフィグレーションと使い方』を参照してください。

WebLogic Server JDBC 多層ドライバ

以下の節では、アプリケーションへのデータベース アクセスを提供する WebLogic 多層 JDBC ドライバについて簡単に説明します。これらのドライバは、サーバ サイドのアプリケーション (および RMI ドライバのクライアント アプリケーション) で使用できますが、JNDI ツリーからデータ ソースをルックアップして、接続プールからデータベース接続を確立することをお勧めします。

各ドライバの使用については、第 4 章「WebLogic 多層 JDBC ドライバの使い方」を参照してください。

WebLogic RMI ドライバ

WebLogic RMI ドライバは WebLogic Server 内で動作する多層 Type 3 Java Database Connectivity (JDBC) ドライバです。WebLogic RMI ドライバを使用して、データベースと接続プールを接続することもできますが、これはお勧めできる方法ではありません。BEA では、JNDI ツリーのデータ ソースをルックアップして、接続プールからデータベース接続を確立することをお勧めします。以降、データ ソースでは内部的に RMI ドライバを使用するようになります。いずれの方法でも、WebLogic RMI ドライバは WebLogic と WebLogic JTS の各ドライバを内部的に使用して、接続プールを接続します。

また、WebLogic Server クラスタ内でコンフィグレーションされている場合は、クラスタ化 JDBC 用に使用できます。これにより、JDBC クライアントは WebLogic クラスタのロード バランシング機能とファイルオーバ機能を活用できます。

WebLogic RMI は、サーバ サイド アプリケーション、またはクライアント アプリケーションと共に使用できます。

WebLogic ドライバの使用については、4-1 ページの「WebLogic RMI ドライバの使い方」を参照してください。

WebLogic Pool ドライバ

WebLogic Pool ドライバを使用すると、HTTP サーブレットや EJB などのサーバサイド アプリケーションから接続プールを利用できます。WebLogic Pool ドライバは、サーバ サイド アプリケーションで直接使用できますが、JNDI のルックアップにより、接続プールからデータベース接続を確立することをお勧めします。WebLogic Server のデータ ソースでは、WebLogic Pool ドライバを内部的に使用して、接続プールを接続します。

Pool ドライバの使い方については、『WebLogic HTTP サーブレット プログラマーズ ガイド』の「プログラミング タスク」の「データベースへのアクセス」を参照してください。

WebLogic JTS ドライバ

WebLogic JTS は WebLogic Pool ドライバに似た多層 JDBC ドライバです。ただし、1 つのデータベース インスタンスを使用する複数のサーバ間での分散トランザクションで使用されます。JTS ドライバは 2 フェーズ コミットを回避するため、単一のデータベース インスタンスを使用する場合に限り WebLogic jDriver for Oracle XA ドライバより効率的に動作します。このドライバは、サーバ サイド アプリケーションでのみ使用します。

WebLogic JTS ドライバの使用については、4-7 ページの「WebLogic JTS ドライバの使い方」を参照してください。

サードパーティ ドライバ

WebLogic Server は、以下の要件を満たすサードパーティ JDBC ドライバと連携して動作します。

- スレッドセーフ
- JDBC API をサポートしているドライバでは、API の拡張もサポートされますが、最低条件として JDBC API がサポートされていなければなりません。
- JDBC で EJB トランザクション呼び出しを実装する

これらのドライバは通常、WebLogic Server をコンフィグレーションして、接続プールに物理的なデータベース接続を作成するときに使用します。

Sybase jConnect ドライバ

2 層 Sybase jConnect Type 4 ドライバは、WebLogic Server 配布キットに付属しています。このドライバは、Sybase Web サイトから最新版を入手して、使用してください。WebLogic Server でのこのドライバの使い方の詳細については、5-1 ページの「WebLogic Server でのサードパーティ ドライバの使い方」を参照してください。

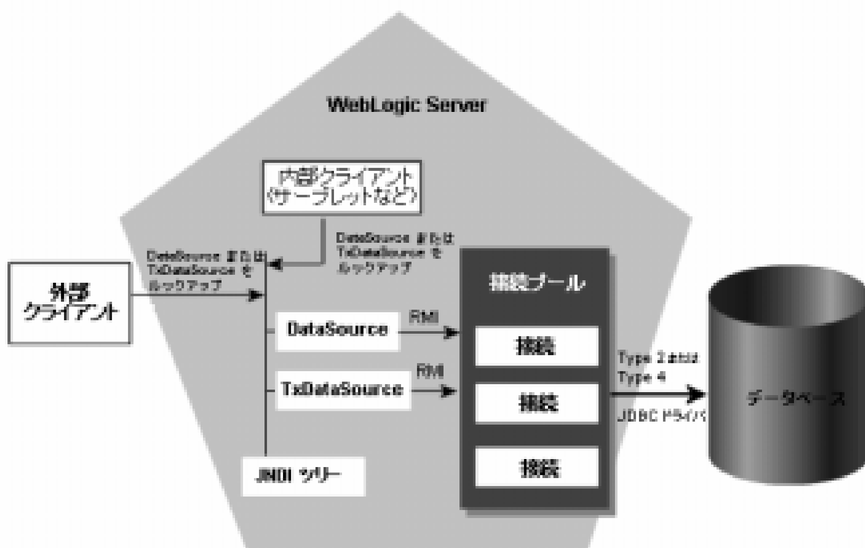
Oracle Thin ドライバ

WebLogic Server に付属する 2 層 Oracle Thin Type 4 ドライバは、WebLogic Server から Oracle DBMS への接続を提供します。Oracle Thin ドライバは、Oracle Web サイトから最新版を入手して、使用してください。WebLogic Server でのこのドライバの使い方の詳細については、5-1 ページの「WebLogic Server でのサードパーティ ドライバの使い方」を参照してください。

接続プールの概要

WebLogic Server では、DBMS への接続にそのまま利用できる接続プールをコンフィグレーションできます。クライアント、およびサーバ サイドのアプリケーションでは、JNDI ツリーにある DataSource を介して接続プールから接続を利用すること (推奨方法) も、多層 WebLogic ドライバを介して接続を利用することもできます。接続が完了すると、アプリケーションにより、接続プールに接続が返されます。

図 1-1 接続プール アーキテクチャへの WebLogic Server の接続



接続プールが起動されると、指定された数の物理的なデータベース接続が作成されます。起動時に接続を確立しておくことにより、接続プールでは個々のアプリケーションごとにデータベース接続を作成するためのオーバーヘッドを解消できます。

接続プールを使って WebLogic Server から DBMS へ物理的に接続するには、2 層 JDBC ドライバが必要になります。2 層ドライバには、WebLogic jDrivers、または JDBC ドライバ (Sybase jConnect ドライバ、または Oracle Thin ドライバ) のいずれかを使用できます。次の表に、接続プールを使用する場合のメリットをまとめています。

表 1-2 接続プールを使用するメリット

接続プールのメリット	実現される機能
時間の節約、オーバーヘッドの削減	DBMS 接続は非常に低速である。接続プールを使用すると、接続が確立されてユーザが利用できるようになる。代わりの手段は、アプリケーションが必要に応じて独自の JDBC 接続を行うことである。DBMS は、実行時にユーザからの接続試行を処理する場合より専用接続を使う方が高速に動作する。
DBMS ユーザの管理	システム上で複数の並列 DBMS 接続を管理できる。これは、DBMS 接続にライセンス上の制限があるか、またはリソースに不安がある場合に重要となる。 アプリケーションは DBMS ユーザ名、パスワード、および DBMS の場所を認識する必要も、伝送する必要もない。
DBMS 永続性オプションを使用できる	DBMS 永続性オプションと EJB のような API を使う場合、WebLogic Server が JDBC 接続を管理するにはプールが必須となる。これにより、EJB トランザクションが正確かつ完全にコミットまたはロール バックされる。

この節は、接続プールの概要です。詳細については、2-2 ページの「接続プールのコンフィグレーションと使い方」を参照してください。

サーバサイド アプリケーションでの接続プールの使い方

サーバ サイド アプリケーションからデータベースにアクセスする場合は、Java Naming and Directory Interface (JNDI) ツリーの DataSource、または WebLogic Pool ドライバのいずれかを使用します。2 フェーズ コミット トランザクションの場合は、JNDI ツリーの TxDataSource を使用するか、WebLogic Server JDBC/XA ドライバ (WebLogic jDriver for Oracle/XA) を使用します。1 つのデー

タベース インスタンスを使う複数のサーバ間での分散トランザクションの場合は、JNDI ツリーの TxDataSource、または JTS ドライバを使用します。WebLogic 多層ドライバではなく、JNDI ツリー、および DataSource を使用して、接続プールにアクセスすることをお勧めします。

クライアントサイド アプリケーションでの接続プールの使い方

BEA は、クライアントサイドの多層 JDBC 用に RMI ドライバを提供しています。RMI ドライバでは、Java 2 Enterprise Edition (J2EE) 仕様を使って標準ベースのアプローチが利用できます。新しいデプロイメントの場合は、RMI ドライバの代わりに、JNDI ツリーの DataSource を使用することをお勧めします。

WebLogic RMI ドライバは Type 3、多層 JDBC ドライバで、RMI と DataSource オブジェクトを使ってデータベース接続を作成します。このドライバはクラスタ化された JDBC にも対応し、WebLogic Server クラスタのロード バランシングおよびフェイルオーバ機能を活用します。DataSource オブジェクトを定義して、トランザクション サポートを有効にしたり無効にしたりできます。

マルチプールの概要

JDBC マルチプールは、「接続プールのプール」であり、高可用性、またはロード バランス アルゴリズムのいずれかに従って設定できます。マルチプールは、接続プールと同じ方法で使用できます。アプリケーションで接続が必要になる場合、マルチプールでは選択されたアルゴリズムに従って、どの接続プールから接続を提供するかを特定します。マルチプールは、複数サーバ コンフィグレーションまたは分散トランザクションではサポートされません。

WebLogic Server 構成で使用するマルチプールごとに、次のいずれかのアルゴリズム オプションを選択できます。

- 高可用性 - 接続プールは順序付けされたリストとして設定され、順番に使用されます。

- ロード バランシング - リストされたすべてのプールはラウンドロビン方式でアクセスされます。

詳細については、2-20 ページの「マルチプールのコンフィグレーションと使い方」を参照してください。

クラスタ化された JDBC の概要

WebLogic Server を使用すると、データ ソース、接続プール、マルチプールなどの JDBC オブジェクトをクラスタ化して、クラスタにホストされるアプリケーションの可用性を高めることができます。クラスタ用にコンフィグレーションする各 JDBC オブジェクトは、クラスタ内の各管理対象サーバに存在する必要があります。JDBC オブジェクトをコンフィグレーションするときに、それらをクラスタに割り当てます。

クラスタ環境での JDBC オブジェクトについては、『WebLogic Server クラスタ ユーザーズ ガイド』の「J DBC 接続」を参照してください。

DataSource の概要

クライアント、およびサーバ サイドの JDBC アプリケーションでは、DataSource を使用して、DBMS 接続を確立することができます。DataSource は、アプリケーションと接続プールの間のインタフェースです。各データ ソース (DBMS インスタンスなど) には、独自の DataSource オブジェクトが必要です。DataSource オブジェクトは、分散トランザクションをサポートする DataSource クラスとして実装できます。詳細については、2-32 ページの「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。

JDBC API

JDBC アプリケーションを作成するには、*java.sql* API を使用して、データ ソースへの接続の確立、クエリの送信、データ ソースへの文の更新、および結果の処理に必要なクラス オブジェクトを作成します。各 JDBC インタフェースの詳細については、`java.sql` Javadoc の標準 JDBC インタフェースを参照してください。また次の WebLogic の Javadoc も参照してください。

- `weblogic.jdbc.pool`
- `weblogic.management.configuration` (データ ソース、接続プール、およびマルチプールを作成する Mbean)

JDBC 2.0

WebLogic Server は、JDBC 2.0 をサポートする JDK 1.3.1 を使用します。

プラットフォーム

サポートされるプラットフォームは、ベンダ固有の DBMS とドライバによって異なります。現時点での情報については、「動作確認状況」を参照してください。

2 WebLogic JDBC のコンフィグレーションと管理

WebLogic Server Administration Console を使用して、JDBC などの WebLogic Server の機能を有効化したり、コンフィグレーションやモニタを行ったりできます。

以下の節では、JDBC 接続コンポーネントのプログラミング方法について説明します。

- 2-2 ページの「接続プールのコンフィグレーションと使い方」
- 2-19 ページの「アプリケーション スコープの JDBC 接続プール」
- 2-20 ページの「マルチプールのコンフィグレーションと使い方」
- 2-32 ページの「DataSource のコンフィグレーションと使い方」

詳細については、以下を参照してください。

- 『管理者ガイド』の「JDBC 接続の管理」。Administration Console およびコマンド ライン インタフェースを使用して、接続をコンフィグレーションおよび管理する方法について説明します。
- Administration Console オンラインヘルプ。Administration Console を使用して特定のコンフィグレーション タスクを行う方法について説明します。

接続プールのコンフィグレーションと使い方

接続プールとは、データベースへの同一の JDBC 接続の集まりです。データベースへの接続は、接続プールを登録すると実行されます。この実行は、WebLogic Server の起動時または実行時に動的に行われます。アプリケーションはプールから接続を「借り」、使用後に接続を閉じることでプールに接続を返します。1-7 ページの「接続プールの概要」も参照してください。

接続プールを使用するメリット

接続プールには、数多くの性能およびアプリケーション設計上の利点があります。

- 接続プールの使用は、データベースへのアクセスが必要になるたびにクライアント別に新しい接続を確立するよりもはるかに効率的です。
- アプリケーションで DBMS パスワードなどの詳細をハードコード化する必要がありません。
- DBMS への接続数を制限できます。DBMS への接続数に対するライセンス制限を管理する場合に便利です。
- アプリケーションのコードを変更せずに、使用中の DBMS を変更できます。

接続プールのコンフィグレーションの属性は Administration Console オンラインヘルプで定義されます。WebLogic Server の実行中に接続プールをプログラムで作成する場合に使用できる API もあります。2-10 ページの「接続プールの動的作成」を参照してください。コマンド ラインを使用することもできます。『管理者ガイド』の「WebLogic Server コマンドライン インタフェース リファレンス」を参照してください。

起動時の接続プールの作成

起動 (静的) 接続プールを作成するには、Administration Console で属性とパーミッションを定義してから、WebLogic Server を起動します。WebLogic Server は、起動処理中にデータベースに対する JDBC 接続を開き、接続をプールに追加します。

Administration Console で接続プールを設定するには、左ペインに表示されるナビゲーション ツリーでサービスと JDBC ノードを展開し、接続プールを選択します。右ペインには、既存の接続プールのリストが表示されます。[新しい JDBC Connection Pool のコンフィグレーション] テキスト リンクをクリックして、接続プールを作成します。

手順説明、および接続プール属性の説明については、Administration Console オンライン ヘルプ (Administration Console の右上の疑問符をクリックしたときに表示される) を参照してください。詳細については、『管理者ガイド』の「JDBC 接続の管理」を参照してください。

正しい接続数によるサーバのロックアップの回避

アプリケーションで、接続プールから接続を取得しようとしたが使用できる接続がない場合、使用できる接続がないという旨の例外が送出されます。接続プールでは、接続要求のキューは作成されません。このエラーを避けるには、接続プールのサイズを接続要求の最大負荷に対応できるサイズに拡大するようにしてください。

Administration Console で接続プールの最大接続数を設定するには、左ペインのナビゲーション ツリーを展開して [サービス | JDBC | 接続プール] ノードを表示し、接続プールを選択します。次に右ペインで、[コンフィグレーション | 接続] タブを選択して、[最大容量] の値を指定します。

接続プールのコンフィグレーションにおけるデータベース パスワード

接続プールを作成する場合、一般にはデータベースへの接続用として、1 つ以上のパスワードを組み込みます。オープン文字列を使用して、XA を有効にする場合は、パスワードを 2 つ使用できます。パスワードは、名前と値のペアとして [プロパティ] フィールドに入力するか、名前と値をそれぞれ対応するフィールドに入力します。

- **[パスワード]**。このフィールドを使用して、データベース パスワードを設定します。物理的なデータベース接続を作成した時点で、2 層 JDBC ドライバに送る [プロパティ] で定義されたパスワードの値がすべてオーバーライドされます。値は、`config.xml` ファイルで暗号化 (`JDBCConnectionPool` タグの `Password` 属性として保存) され、Administration Console では表示されなくなります。
- **[オープン文字列のパスワード]**。このフィールドを使用して、WebLogic Server のトランザクション マネージャがデータベース接続を開くときに使用するオープン文字列内にパスワードを設定します。この値によって、[プロパティ] フィールド内のオープン文字列の一部として定義されたすべてのパスワードがオーバーライドされます。値は、`config.xml` ファイルで暗号化 (`JDBCConnectionPool` タグの `XAPassword` 属性として保存) され、Administration Console では表示されなくなります。実行時に、WebLogic Server はこのフィールドで指定されたオープン文字列を再構築します。[プロパティ] フィールドのオープン文字列は、次のフォーマットにする必要があります。

  ```
  openString=Oracle_XA+Acc=P/userName/+SesTm=177+DB=demoPool+Threads=true=Sqlnet=dvi0+logDir=
  ```

  `userName` の後、パスワードは表示されなくなります。

接続プールを初めてコンフィグレーションするときに [プロパティ] フィールドにパスワードを指定した場合、次の起動時に WebLogic Server は、パスワードを `Properties` 文字列から取り除き、その値を暗号化して `Password` の値として設定します。接続プールの `Password` 属性に値が指定されている場合、WebLogic Server は値を変更しません。ただし、`Password` 属性の値は `Properties` 文字列内のパスワード値をオーバーライドします。この動作は、オープン文字列の一部

として定義するすべてのパスワードに対して同じように適用されます。たとえば、接続プールを初めてコンフィグレーションするときに次のプロパティを指定したとします。

```
user=scott;
password=tiger;
openString=Oracle_XA+Acc=p/scott/tiger+SesTm=177+db=jtaXaPool+Thr
eads=true+Sqlnet=lcs817+logDir=.+dbgFl=0x15;server=lcs817
```

WebLogic Server を次に起動すると、データベース パスワードは [パスワード] 属性に、オープン文字列内のパスワードは [オープン文字列のパスワード] 属性に送られ、[プロパティ] フィールドには次の値が残ります。

```
user=scott;
openString=Oracle_XA+Acc=p/scott/+SesTm=177+db=jtaXaPool+Threads=
true+Sqlnet=lcs817+logDir=.+dbgFl=0x15;server=lcs817
```

[パスワード] または [オープン文字列のパスワード] 属性の値が確定したら、これらの属性の値は [プロパティ] フィールドのそれぞれの値をオーバーライドします。前述の例で説明すると、`tiger2` をデータベース パスワードとして [プロパティ] 属性に指定した場合、WebLogic Server はこの値を無視し、暗号化された [パスワード] 属性の現在の値である `tiger` をデータベース パスワードとして使い続けます。データベース パスワードを変更するには、[パスワード] 属性を変更しなければなりません。

注意 : [パスワード] と [オープン文字列のパスワード] の値は、同じでなくてもかまいません。

接続プールの制限事項

接続プールを使用する場合、データベース接続プロパティを変更する DBMS 固有の SQL コードや、WebLogic Server および JDBC ドライバでは認識されない SQL コードを実行することが可能です。接続が接続プールに戻されたときに、接続の特性が有効な状態に戻らない場合があります。たとえば、Sybase DBMS では、`set rowcount 3 select * from y` という文を使用すると、接続から返される行は最大で 3 行です。この接続が接続プールに戻されて再利用される際、選択しているテーブルが 500 行だったとしても、クライアントに返されるのは 3 行のみとなります。ほとんどの場合は、同じ結果を実現でき、WebLogic Server または JDBC ドライバによって接続がリセットされる標準の (DBMS 固有でない) SQL コードが用意されています。この例であれば、`set rowcount` の代わりに `setMaxRows()` を使用できます。

接続を変更する DBMS 固有の SQL コードを使用する場合は、接続プールに戻す前に接続を適切な状態に設定しなおす必要があります。

JDBC 接続プールの接続の更新に関する注意事項

更新プロセスで置換できない不良なデータベース接続が見つかると、そのプロセスは現在のサイクルを停止します。残っている壊れた接続は、接続プールから削除されません。それらの接続は、新しい接続で置換できるまで接続プールに留まります。この動作設計は、DBMS がアクセス不能な場合にシステムのサイクルを利用してデータベース接続を更新することによるパフォーマンスの低下を防止するためでした。

更新プロセスでは、アプリケーション コードで現在使用されている接続をテストまたは更新することはできません。テストされるのは、現時点で予約されていない接続だけです。したがって、たとえ見つかった不良接続を置換できる場合であっても、アプリケーションが接続を要求している場合には更新サイクルで接続プールのすべての接続がテストされることはありません。

更新プロセスでは使用されていない接続しかテストされないので、一部の接続はまったくテストされることがない、ということも考えられます。`testConnsOnReserve` を有効にしない限りは、クライアントは常に壊れた接続を取得するリスクを負います。実際に、たとえアプリケーションに渡される前に接続がテストを受けていても、その接続はテストの成功の直後に不良化することがあります。

JDBC 接続プールのテスト機能の強化

WebLogic Server 7.0SP5 では、プールされた接続に対するデータベース接続テストの機能を改良するために、JDBC 接続プールに以下の属性が追加されました。

- `CountOfTestFailuresTillFlush`— 指定する回数のテストが失敗した後で接続プール内のすべての接続を閉じ、それ以上データベース テストを行うことで発生する遅延を最小限に抑えます。「データベース接続消失後の接続テストでの遅延を最小限に抑える」を参照してください。
- `CountOfRefreshFailuresTillDisable`— 指定する回数のテストが失敗した後で接続プールを無効にし、データベースの障害発生後に接続要求を処理す

る際の遅延を最小限に抑えます。「接続テスト失敗後の接続要求の遅延を最小限に抑える」を参照してください。

データベース接続消失後の接続テストでの遅延を最小限に抑える

DBMS への接続が一時的にでも失われると、接続プール内の一部または全部の JDBC 接続は停止した状態になります。接続プールが予約時に接続をテストするようにコンフィグレーションされている場合 (推奨)、アプリケーションがデータベース接続を要求すると、WebLogic Server は接続をテストします。接続が停止していることを検出すると、要求に応じるためにその接続を新しい接続で置き換えます。通常、DBMS がオンラインに復帰すると更新処理が行われます。ただし、特定の状況や失敗の状態によっては、停止した接続をテストすると長い遅延が発生することがあります。この遅延は、接続プール内の停止した接続ごとに発生し、すべての接続が置き換えられるまで続きます。

停止したデータベース接続のテスト中に発生する遅延を最小限に抑えるために、接続プールで CountOfTestFailuresTillFlush 属性を設定できます。この接続を設定する場合、WebLogic Server では、テストが指定された回数続けて失敗すると、接続プール内のすべての接続を停止した状態と見なして、接続プール内のすべての接続を閉じます。

アプリケーションが接続を要求したときに、接続プールは停止した接続のテストを最初に行わずに、接続を作成することができます。この動作により、接続プールがフラッシュされた後は、接続要求の遅延を最小限に抑えられます。

CountOfTestFailuresTillFlush 属性は config.xml ファイルの JDBCConnectionPool エントリ内に指定します。TestConnectionsOnReserve も true に設定する必要があります。次に例を示します。

```
<JDBCConnectionPool
  CapacityIncrement="1"
  DriverName="com.pointbase.xa.xaDataSource"
  InitialCapacity="2" MaxCapacity="10"
  Name="demoXAPool" Password="password"
  Properties="user=examples;
    DatabaseName=jdbc:pointbase:server://localhost/demo"
  Targets="examplesServer"
  TestConnectionsOnReserve="true"
  CountOfTestFailuresTillFlush="1"
  URL="jdbc:pointbase:server://localhost/demo"
/>
```

注意： `CountOfTestFailuresTillFlush` 属性は、Administration Console では使用できません。

ネットワークに小さな障害が発生したり、ファイアウォールが唯一のソケットまたは接続を停止することがよくある場合は、テストの失敗数を 2 または 3 に設定することができます。ただし、データベースの可用性の問題を解決した場合は、値を 1 に設定すると最良のパフォーマンスが得られます。

接続テスト失敗後の接続要求の遅延を最小限に抑える

DBMS が使用不能になると、接続プールは接続要求に応じようとして、永続的にテストを行い、停止した接続を置き換えようとします。接続プールはデータベースが使用可能になるとすぐに反応できるため、この動作は有益です。ただし、停止したデータベース接続のテストはネットワークがタイムアウトするまで行われることがあり、その結果クライアントで遅延が発生します。

データベースが使用できない場合にクライアント アプリケーションで発生する遅延を最小限に抑えるために、接続プールで `CountOfRefreshFailuresTillDisable` 属性を設定できます。この属性を設定すると、WebLogic Server は、停止した接続の置き換えに特定の回数続けて失敗した後、接続プールを無効にします。アプリケーションが無効な接続プールの接続を要求した場合、WebLogic Server は直ちに `ConnectDisabledException` を送出して、クライアントに接続が使用できないことを通知します。

この方法で無効になった接続プールに対して、WebLogic Server は定期的に更新処理を実行します。更新処理によって新しいデータベース接続が正常に作成されると、WebLogic Server は接続プールを再び有効にします。`weblogic.Admin ENABLE_POOL` コマンドを使用して、接続プールを手動で最有効化することもできます。

`CountOfRefreshFailuresTillDisable` 属性は `config.xml` ファイルの JDBCConnectionPool エントリ内に指定します。`TestConnectionsOnReserve` も `true` に設定する必要があります。次に例を示します。

```
<JDBCConnectionPool
  CapacityIncrement="1"
  DriverName="com.pointbase.xa.xaDataSource"
  InitialCapacity="2" MaxCapacity="10"
  Name="demoXAPool" Password="password"
  Properties="user=examples;
    DatabaseName=jdbc:pointbase:server://localhost/demo"
  Targets="examplesServer"
```

```
  TestConnectionsOnReserve="true"
  CountOfRefreshFailuresTillDisable="1"
  URL="jdbc:pointbase:server://localhost/demo"
/>
```

注意： `CountOfRefreshFailuresTillDisable` 属性は、Administration Console では使用できません。

ネットワークに小さな障害が発生したり、ファイアウォールが唯一のソケットまたは接続を停止することがよくある場合は、更新の失敗数を 2 または 3 に設定することができます。ただし、値を 1 に設定すると最良のパフォーマンスが得られます。

secondsToTrustAnIdlePoolConnection を使用して接続要求の遅延を最小限に抑える

大量のトラフィック発生時にデータベース接続をテストすると、アプリケーションのパフォーマンスが低下するおそれがあります。接続テストへの影響を最小限に抑えるために、JDBC 接続プールのコンフィグレーションで `secondsToTrustAnIdlePoolConnection` 接続プロパティを設定して、最近使用された、またはテストされたデータベース接続の有効性を信頼することで、接続テストを省略できます。

使用する接続プールが、予約時に接続をテストするようにコンフィグレーションされている場合 (推奨)、アプリケーションがデータベース接続を要求すると、WebLogic Server ではそのデータベース接続をテストしてからアプリケーションに渡します。ただし、接続がテストされたか正常に使用されて接続プールに返されたときから、secondsToTrustAnIdlePoolConnection で指定された時間が経過するまでの間に行われた要求については、接続をアプリケーションに渡す前の接続テストが省略されます。

使用する接続プールが、プール内で使用可能な接続を定期的にテストするようにコンフィグレーションされている場合 (RefreshMinutes を指定)、正常に使用されて接続プールに返された接続については、secondsToTrustAnIdlePoolConnection で指定された時間内の接続テストが同様に省略されます。

secondsToTrustAnIdlePoolConnection を設定するには、Administration Console の [JDBC 接続プール | コンフィグレーション | 一般] タブにあるプロパティのリストに secondsToTrustAnIdlePoolConnection を追加します。Administration Console

オンライン ヘルプの「[JDBC 接続プール] --> [コンフィグレーション] --> [一般]」を参照してください。また、このプロパティは config.xml ファイルで直接設定することもできます。次に例を示します。

```
<JDBCConnectionPool
  CapacityIncrement="1"
  DriverName="com.pointbase.xa.xaDataSource"
  InitialCapacity="2" MaxCapacity="10"
  Name="demoXAPool" Password="password"
  Properties="user=examples;
    secondsToTrustAnIdlePoolConnection=15;
    DatabaseName=jdbc:pointbase:server://localhost/demo"
  Targets="examplesServer"
  TestConnectionsOnReserve="true"
  TestTableName="SYSTABLES"
  URL="jdbc:pointbase:server://localhost/demo"
/>
```

secondsToTrustAnIdlePoolConnection は、(特に大量のトラフィック発生時の) データベース接続テストによる遅延を最小限に抑えることにより、アプリケーションのパフォーマンスを向上できるチューニング機能です。ただし、値を高く設定しすぎると、接続テストの有効性が低くなることがあります。適切な値は、使用する環境と接続の切断が発生しやすいかどうかによって決まります。

接続プールの動的作成

JDBCConnectionPool 管理 MBean は WebLogic Server 管理アーキテクチャ (JMX) の一部として提供されます。JDBCConnectionPool MBean を使用して、Java アプリケーションの内部から、接続プールを動的に作成、およびコンフィグレーションできます。すなわち、クライアント、またはサーバ アプリケーション コードを使用して、既に実行中の WebLogic Server で接続プールを作成できます。

WebLogic Server コマンド ライン インタフェースで CREATE_POOL コマンドを使用しても、接続プールを動的に作成できます。『管理者ガイド』の「CREATE_POOL」を参照してください。

JDBCConnectionPool 管理 MBean を使用して接続プールを動的に作成するには、次の手順に従います。

1. 必要なパッケージをインポートします。

2. JNDI ツリーで、管理 MBeanHome をルックアップします。
3. サーバ MBean を取得します。
4. 接続プール MBean を作成します。
5. 接続プールのプロパティを設定します。
6. 対象を追加します。
7. DataSource オブジェクトを作成します。

注意： 動的に作成した接続プールでは必ず、動的に作成した DataSource オブジェクトを使用します。DataSource を終了するには、これが特定の接続プールに関連付けられている必要があります。また、WebLogic Server 中の DataSource オブジェクトと接続プールの間は、1 対 1 の関係になっています。したがって、接続プールに対応して使用する DataSource の作成が必要です。

`JDBCConnectionPool` MBean を使用して接続プールを作成すると、サーバ コンフィグレーションに接続プールが追加され、サーバを起動 / 停止した後も、引き続き有効になります。接続プールを持続させる必要がない場合は、プログラムで削除します。

また、動的に作成した接続プールは一時的に無効にすることができます。無効にすると、プール中のどの接続でもデータベース サーバとの通信がサスペンドされます。いったん無効にしたプールを再度有効にすると、各接続ではプールを無効にした時点と同じ状態が復元されるため、クライアントは中断された地点からデータベースの操作を続行できます。

MBean を使用して WebLogic Server を管理する方法については、『WebLogic JMX Service プログラマーズ ガイド』を参照してください。
`JDBCConnectionPool` MBean の詳細については、WebLogic クラスの Javadoc を参照してください。

動的接続プールのサンプル コード

接続プールを動的に作成するときの各手順を実行するコード サンプルを以下の節で示します。

パッケージをインポートする

```
import java.sql.*;
import java.util.*;
import javax.naming.Context;
import javax.sql.DataSource;
import weblogic.jndi.Environment;
import weblogic.management.configuration.JDBCConnectionPoolMBean;
import weblogic.management.runtime.JDBCConnectionPoolRuntimeMBean;
import weblogic.management.configuration.JDBCDataSourceMBean;
import weblogic.management.configuration.ServerMBean;
import weblogic.management.MBeanHome;
import weblogic.management.WebLogicObjectName;
```

管理 MBeanHome をルックアップする

```
mbeanHome = (MBeanHome)ctx.lookup(MBeanHome.ADMIN_JNDI_NAME);
```

サーバ MBean を取得する

```
serverMBean = (ServerMBean)mbeanHome.getAdminMBean(serverName, "Server");
//Server MBean の WebLogic オブジェクト名を作成する
//JDBCConnectionPoolRuntime MBean の名前を作成する場合に使用
WebLogicObjectName pname = new WebLogicObjectName("server1", "ServerRuntime",
mbeanHome.getDomainName(),"server1");
//JDBCConnectionPoolRuntime MBean の WebLogic オブジェクト名を作成する
//JDBCConnectionPoolRuntime MBean の名前を作成する場合に使用する
WebLogicObjectName oname = new WebLogicObjectName(cpName,
"JDBCConnectionPoolRuntime", mbeanHome.getDomainName(),"server1", pname);
JDBCConnectionPoolRuntimeMBean cprmb =
(JDBCConnectionPoolRuntimeMBean)mbeanHome.getMBean(oname);
```

接続プール MBean を作成する

```
// ConnectionPool MBean の作成
cpMBean = (JDBCConnectionPoolMBean)mbeanHome.createAdminMBean(
  cpName, "JDBCConnectionPool",
      mbeanHome.getDomainName());
```

接続プール プロパティを設定する

```
Properties pros = new Properties();
pros.put("user", "scott");
  pros.put("server", "lcdbnt1");
```

```
// DataSource 属性の設定
cpMBean.setURL("jdbc:weblogic:oracle");
cpMBean.setDriverName("weblogic.jdbc.oci.Driver");
cpMBean.setProperties(pros);
cpMBean.setPassword("tiger");
cpMBean.setLoginDelaySeconds(1);
cpMBean.setInitialCapacity(1);
cpMBean.setMaxCapacity(10);
cpMBean.setCapacityIncrement(1);
cpMBean.setShrinkingEnabled(true);
cpMBean.setShrinkPeriodMinutes(10);
cpMBean.setRefreshMinutes(10);
cpMBean.setTestTableName("dual");
```

注意： この例では、[プロパティ] にユーザ名とサーバ名で指定する代わりに、setPassword(String) メソッドを使ってデータベース パスワードを設定します。setPassword(String) メソッドを使用すると、WebLogic Server は config.xml ファイル中のパスワード、および管理コンソールに表示されるパスワードを暗号化します。このメソッドを使用して、config.xml ファイル中でデータベース パスワードをクリア テキストで格納するのを避けるようにお勧めします。

対象を追加する

```
cpMBean.addTarget(serverMBean);
```

DataSource を作成する

```
public void createDataSource() throws SQLException {
  try {
    // コンテキストの取得
    Environment env = new Environment();
    env.setProviderUrl(url);
    env.setSecurityPrincipal(userName);
    env.setSecurityCredentials(password);
    ctx = env.getInitialContext();

      // DataSource  MBean の作成
      dsMBeans = (JDBCDataSourceMBean)mbeanHome.createAdminMBean(
        cpName, "JDBCDataSource",
        mbeanHome.getDomainName());

      // DataSource 属性の設定
      dsMBeans.setJNDIName(cpJNDIName);
      dsMBeans.setPoolName(cpName);

      // DataSource の起動
      dsMBeans.addTarget(serverMBean);
```

```
    } catch (Exception ex) {
      ex.printStackTrace();
    throw new SQLException(ex.toString());

    }

  }
```

動的接続プールと DataSource を削除する

動的に作成した接続プールを削除する方法を以下のコード サンプルで示します。動的に作成した接続プールは削除しない限り、サーバを停止 / 再起動した後も、有効です。

```
public void deleteConnectionPool() throws SQLException {
  try {
    // 動的に作成した接続プールをサーバから削除
    cpMBean.removeTarget(serverMBean);
    // 動的に作成した接続プールをコンフィグレーションから削除
    mbeanHome.deleteMBean(cpMBean);
  } catch (Exception ex) {
    throw new SQLException(ex.toString());
  }
}


  public void deleteDataSource() throws SQLException {

    try {

      // 動的に作成した DataSource をサーバから削除

      dsMBeans.removeTarget(serverMBean);

      // 動的に作成した DataSource をコンフィグレーションから削除

      mbeanHome.deleteMBean(dsMBeans);

    } catch (Exception ex) {

      throw new SQLException(ex.toString());

    }

  }
```

接続プールの管理

JDBCConnectionPool と JDBCConnectionPoolRuntime の MBean には、接続プールを管理し、これに関する情報を収集するためのメソッドがあります。メソッドの目的は以下のとおりです。

- プールに関する情報を取得します。
- 接続プールを無効にして、そこからクライアントが接続を取得できないようにします。
- 無効にされているプールを有効にします。
- 未使用の接続を解放して、指定された最小サイズまでプールを縮小します。
- プールをリフレッシュします (接続をクローズして再び開く)。
- プールを停止します。

JDBCConnectionPool MBean と JDBCConnectionPoolRuntime MBean によって、weblogic.jdbc.common.JdbcServices と weblogic.jdbc.common.Pool は非推奨の機能として置き換えられています。

JDBCConnectionPool MBean で使用できるメソッドの詳細については、Javadoc を参照してください。JDBCConnectionPoolRuntime MBean の詳細については、Javadoc を参照してください。

プールに関する情報の取得

```
boolean x = JDBCConnectionPoolRuntimeMBean.poolExists(cpName);
```

```
props = JDBCConnectionPoolRuntimeMBean.getProperties();
```

poolExists() メソッドは、指定された名前の接続プールが WebLogic Server に存在するかどうかを調べます。このメソッドを使用すると、動的接続プールが既に作成されているかどうかを調べ、作成する動的接続プールに固有の名前を付けることができます。

getProperties() メソッドは、接続プールのプロパティを取得します。

接続プールの無効化

```
JDBCConnectionPoolRuntimeMBean.disableDroppingUsers()
JDBCConnectionPoolRuntimeMBean.disableFreezingUsers()
JDBCConnectionPoolRuntimeMBean.enable()
```

接続プールを一時的に無効にして、クライアントがそのプールから接続を取得するのを防ぐことができます。接続プールを有効または無効にできるのは、「system」ユーザか、またはそのプールに関連付けられている ACL によって「admin」パーミッションが与えられたユーザだけです。

`disableFreezingUsers()` を呼び出すと、プールの接続を現在使っているクライアントは中断状態に置かれます。データベース サーバと通信しようとすると、例外が送出されます。ただし、クライアントは接続プールが無効になっている間に自分の接続をクローズできます。その場合、接続はプールに返され、プールが有効になるまでは別のクライアントから予約することはできません。

`disableDroppingUsers()` を使用すると、接続プールが無効になるだけでなく、そのプールに対するクライアントの JDBC 接続が破棄されます。その接続で行われるトランザクションはすべてロールバックされ、その接続が接続プールに返されます。クライアントの JDBC 接続コンテキストは無効になります。

`disableFreezingUsers()` で無効にしたプールを再び有効にした場合、使用中だった各接続の JDBC 接続状態はその接続プールが無効にされたときと同じなので、クライアントはちょうど中断したところから JDBC 操作を続行できます。

さらに、`weblogic.Admin` クラスの `disable_pool` コマンドと `enable_pool` コマンドを使用して、プールを無効にしたり有効にしたりできます。

接続プールの縮小

```
JDBCConnectionPoolRuntimeMBean.shrink()
```

接続プールは、プール内の接続の初期数と最大数を定義する一連のプロパティ (`initialCapacity` と `maxCapacity`) と、接続がすべて使用中のときにプールに追加される接続の数を定義するプロパティ (`capacityIncrement`) を備えています。プールがその最大容量に達すると、最大数の接続が開くことになり、プールを縮小しない限りそれらの接続は開いたままになります。

接続の使用がピークを過ぎれば、接続プールから接続をいくつか削除して、WebLogic Server と DBMS のリソースを解放してもかまいません。

接続プールの停止

`JDBCConnectionPoolRuntimeMBean.shutdownSoft()`

`JDBCConnectionPoolRuntimeMBean.shutdownHard()`

これらのメソッドは、接続プールを破棄します。接続はクローズされてプールから削除され、プールに残っている接続がなくなればプールは消滅します。接続プールを破棄できるのは、「system」ユーザか、またはそのプールに関連付けられている ACL によって「admin」パーミッションが与えられたユーザだけです。

`shutdownSoft()` は、接続がプールに返されるのを待って、それらの接続をクローズします。

`shutdownHard()` メソッドは、すべての接続を即座に破棄します。プールから取得した接続を使っているクライアントは、`shutdownHard()` が呼び出された後で接続を使おうとすると、例外を受け取ることになります。

さらに、`weblogic.Admin` クラスの `destroy_pool` コマンドを使って、プールを破棄することもできます。

プールのリセット

`JDBCConnectionPoolRuntimeMBean.reset()`

接続プールは、定期的に、あるいは接続が予約または解放されるたびに接続をテストするようにコンフィグレーションすることができます。WebLogic Server がプール接続の一貫性を自動的に保てるようにすることで、DBMS 接続に関する問題の大半は防げるはずです。さらに、WebLogic には、アプリケーションから呼び出してプール内のすべての接続、またはプールから予約した単一の接続をリフレッシュできるメソッドが用意されています。

`JDBCConnectionPoolRuntimeMBean.reset()` メソッドは、接続プール内に割り当てられている接続をすべてクローズしてから開き直します。これは、たとえば、DBMS が再起動されたあとに必要になることがあります。接続プール内の 1 つの接続が失敗した場合は、プール内のすべての接続が不良であることが往々にしてあります。

接続プールをリセットするには、以下のいずれかの方法を使用します。

- Administration Console
- weblogic.Admin コマンド。管理者 (管理特権を持ったユーザ) として接続プールをリセットします。次にそのパターンを示します。

```
$ java weblogic.Admin WebLogicURL RESET_POOL poolName system passwd
```

  コマンドラインからこの方法を使うことはめったにないかもしれません。他にも、次に説明するようにプログラムを使用したより効率的な方法があります。

- クライアント アプリケーションにある JDBCConnectionPoolRuntimeMBean の reset() メソッド

  最後のケースは、行うべき作業が最も多くなりますが、その反面、最初の 2 つの方法よりも柔軟性があります。次に、reset() メソッドを使用してプールをリセットする方法を示します。

  a. try ブロックの中で、DBMS への有効な接続が存在する限りどのような状況でも必ず成功する SQL 文を使用して、接続プールの接続をテストします。たとえば、「select 1 from dual」という SQL 文は、Oracle DBMS の場合には必ず成功します。

  b. SQLException を取得します。

  c. catch ブロックの中で reset() メソッドを呼び出します。

weblogic.jdbc.common.JdbcServices と weblogic.jdbc.common.Pool クラス (非推奨) の使用

旧バージョンの WebLogic Server には、プログラム上で接続プールの作成や管理のための次のクラスが組み込まれていました。
weblogic.jdbc.common.JdbcServices and weblogic.jdbc.common.Pool. これらのクラスは非推奨となっています。これらのクラスは現在でも利用できますが、代わりに JDBCConnectionPool MBean を使用して、接続プールを動的に作成、および管理することをお勧めします。

JDBCConnectionPool MBean を使用して、管理対象のサーバ上の接続プールを作成、または変更すると、JMX サービスにより管理サーバに変更が直ちに通知されます。weblogic.jdbc.common.JdbcServices と weblogic.jdbc.common.Pool を使用して、接続プールを作成、または変更すると、次のアクションは管理サーバには伝達されません。

- 停止
- 取り出し
- 更新
- 有効化
- 無効化

これらのアクションを実行すると、関連する接続プールを使用した管理対象サーバ上のアプリケーションが失敗することがあります。

weblogic.jdbc.common.JdbcServices と weblogic.jdbc.common.Pool の詳細については、WebLogic Server バージョン 6.1 の『WebLogic JDBC プログラマーズ ガイド』の「WebLogic JDBC のコンフィグレーションと管理」を参照してください。

アプリケーション スコープの JDBC 接続プール

エンタープライズ アプリケーションをパッケージ化するときに、weblogic-application.xml 補足デプロイメント記述子を含めることができます。この記述子を使用して、アプリケーション スコーピングをコンフィグレーションします。weblogic-application.xml ファイルでは、エンタープライズ アプリケーションをデプロイするときに作成される JDBC 接続プールをコンフィグレーションできます。

接続プールのインスタンスは、アプリケーションのインスタンスごとに作成されます。つまり、プールのインスタンスは、アプリケーションの対象となっている各ノード上のアプリケーションで作成されます。プールのサイズを検討する場合は、この点に留意してください。

この方法で作成された接続プールは「アプリケーション スコープの接続プール」(アプリケーション スコープ プール、アプリケーション ローカル プール、または ローカル プール) と呼ばれ、そのエンタープライズ アプリケーションに対してだけスコーピングされます。つまり、各接続プールがエンタープライズ アプリケーションごとに分離されます。

アプリケーション スコーピング、およびアプリケーション スコープ リソースの詳細については、以下を参照してください。

- 『WebLogic Server アプリケーションの開発』の「weblogic-application.xml デプロイメント記述子の要素」
- 『WebLogic Server アプリケーションの開発』の「エンタープライズ アプリケーションのパッケージ化」
- 『WebLogic Server アプリケーションの開発』の「2 フェーズ デプロイメント」

マルチプールのコンフィグレーションと使い方

マルチプールは「プールのプール」です。まず接続プールを作成し、次に Administration Console または WebLogic 管理 API を使用してマルチプールを作成した後、マルチプールに接続プールを割り当てることにより、マルチプールを作成します。

Administration Console を使用してマルチプールを作成する手順については、Administration Console オンライン ヘルプを参照してください。`JDBCMultiPoolMBean` の詳細については、WebLogic Server Javadoc を参照してください。

マルチプールの機能

マルチプールとは、接続プールのプールです。特定の接続プール内のすべての接続は、同じデータベース、同じユーザ、同じ接続プロパティを使って作成されます。つまり、それらは単一のデータベースに関連付けられます。しかし、マルチプール内の接続プールには、それぞれ異なる DBMS を関連付けることができます。

マルチプールのデータベース接続はローカル トランザクションでのみ使用されます。分散トランザクションでの使用は、WebLogic Server ではサポートされていません。

マルチプール アルゴリズムの選択

マルチプールを設定する前に、その主要な目的、つまり高可用性またはロードバランシングのいずれかを指定する必要があります。アルゴリズムは、各自のニーズに合わせて選択できます。

高可用性

高可用性アルゴリズムでは、接続プールの順序付けされたリストを提供します。通常、このタイプのマルチプールへのすべての接続リクエストは、リストの最初のプールによって処理されます。このプールを通したデータベース接続が失敗すると、そのリストの次のプールから順番に接続が選択されます。

注意： マルチプール内の接続プールに対して `TestConnectionsOnReserve=true` を設定することで、リスト内の次の接続プールにフェイルオーバするタイミングをマルチプールが決定できるようにする必要があります。

デフォルトでは、接続プール内のすべての接続が使用中の場合、高可用性アルゴリズムを備えたマルチプールではリスト内の次のプールから接続が提供されることはありません。これは、接続プールの容量を設定できるようにするためです。このシナリオでフェイルオーバを有効にするには、マルチプールのコンフィグレーションで

`FailoverRequestIfBusy` 属性を `true` に設定します。詳細については、2-24 ページの「マルチプールの使用されている接続プールのフェイルオーバの有効化」を参照してください。

ロード バランシング

ロード バランシング マルチプールへの接続リクエストは、リスト内の任意の接続プールから処理されます。プールはリストの順に追加され、ラウンドロビン方式でアクセスされます。アプリケーションが接続を要求すると、マルチプールはリスト内の次の接続プールから接続を提供しようとします。

マルチプールのフェイルオーバの拡張

WebLogic Server 7.0SP5 では、マルチプールに以下の拡張が施されました。

- マルチプール内で自動的に無効化されている (非アクティブな) 接続プールに接続を要求しないようにするための、接続要求の転送の改良。「接続プールが失敗したときの接続要求の転送の改良」を参照してください。
- マルチプール内の失敗した接続プールが回復したときの自動フェイルバック。「マルチプール内の失敗した接続プールが回復するときの自動的な再有効化」を参照してください。
- マルチプール内の使用中の接続プールのフェイルオーバ。「マルチプールの使用されている接続プールのフェイルオーバの有効化」を参照してください。
- 高可用性アルゴリズムを備えたマルチプールのフェイルオーバ コールバック。「コールバックによるマルチプールのフェイルオーバの制御」を参照してください。
- マルチプール (アルゴリズムはどちらでも同じ) のフェイルバック コールバック 「コールバックによるマルチプールのフェイルバックの制御」を参照してください。

接続プールが失敗したときの接続要求の転送の改良

マルチプール内の接続プールが失敗したときのパフォーマンスを向上させるために、WebLogic Server はプールの接続が接続テストに失敗したときにその接続プールを自動的に無効にします。接続プールが無効化された後、WebLogic Server は接続要求をアプリケーションからその接続プールに転送しません。その代わりに、接続要求はマルチプールのリストにある次に利用可能な接続プールに転送されます。

この機能を利用するには、マルチプールのすべての接続プールで接続プール テスト オプションがコンフィグレーションされていなければなりません (特に TestTableName と TestConnectionsOnReserve)。

マルチプールに対してコールバック ハンドラが登録されている場合、WebLogic Server はリストの次の接続プールにフェイルオーバする前にコールバック ハンドラを呼び出します。詳細については、2-25 ページの「コールバックによるマルチプールのフェイルオーバの制御」を参照してください。

マルチプール内の失敗した接続プールが回復するときの自動的な再有効化

接続が接続テストに失敗したことが原因で接続プールが自動的に無効化された後、WebLogic Server は定期的に無効化された接続プールの接続をテストして接続プール (または基盤のデータベース) がいつ再び利用可能になるのかを判断します。接続プールが利用可能になると、WebLogic Server は自動的に接続プールを再び有効化し、マルチプールのアルゴリズムと、接続プールのリストにおける位置に基づいて、その接続プールへの接続要求の転送を再開します。

マルチプールで自動的に無効化されている接続プールが WebLogic Server によってチェックされる頻度を制御するには、config.xml ファイルのマルチプールのコンフィグレーションに HealthCheckFrequencySeconds 属性の値を追加します。次に例を示します。

```
<JDBCMultiPool
AlgorithmType="High-Availability"
Name="demoMultiPool"
PoolList="demoPool2,demoPool"
HealthCheckFrequencySeconds="240"
Targets="examplesServer" />
```

注意 : この属性は、Administration Console では表示されません。この機能を実装するには、`config.xml` ファイルのマルチプールのコンフィグレーションに手動でこの属性を追加する必要があります。

WebLogic Server は、無効化されている各接続プールの接続テストを、指定された期間は行いません。デフォルト値は、300 秒です。値を指定しない場合、自動的に無効化された接続プールは 300 秒おきにテストされます。

この機能を利用するには、マルチプールのすべての接続プールで接続プール テスト オプションがコンフィグレーションされていなければなりません (特に `TestTableName` と `TestConnectionsOnReserve`)。

手動で無効化された接続プールについては、WebLogic Server はテストと自動的な再有効化を行いません。テストするのは、自動的に無効化された接続プールだけです。

マルチプールに対してコールバック ハンドラが登録されている場合、WebLogic Server は接続プールを再び有効にする前にコールバック ハンドラを呼び出します。詳細については、2-29 ページの「コールバックによるマルチプールのフェイルバックの制御」を参照してください。

マルチプールの使用されている接続プールのフェイルオーバの有効化

デフォルトでは、マルチプールのアルゴリズムが高可用性である場合、データベース接続に対する要求の数がマルチプールの現在の接続プールで利用可能な接続の数を超えると、それ以降の接続要求が失敗します。

現在の接続プールのすべての接続が使用中のときにマルチプールがフェイルオーバするようにするには、`config.xml` ファイルのマルチプールのコンフィグレーションで `FailoverRequestIfBusy` 属性の値を設定する必要があります。`true` に設定すると、現在の接続プールのすべての接続が使用中のときに、接続に対するアプリケーションの要求がマルチプールの次の利用可能な接続プールに転送されます。`false` (デフォルト) に設定した場合、接続要求はフェイルオーバされません。

`FailoverRequestIfBusy` 属性を `config.xml` ファイルに追加した後、マルチプールのエントリは次のようになります。

```
<JDBCMultiPool
AlgorithmType="High-Availability"
Name="demoMultiPool"
PoolList="demoPool2,demoPool"
FailoverRequestIfBusy="true"
Targets="examplesServer" />
```

注意 : FailoverRequestIfBusy 属性は、Administration Console では表示されません。この機能を実装するには、config.xml ファイルのマルチプールのコンフィグレーションに手動でこの属性を追加する必要があります。

ConnectionPoolFailoverCallbackHandler がマルチプールのコンフィグレーションに含まれている場合、WebLogic Server はフェイルオーバの前にコールバック ハンドラを呼び出します。詳細については、2-25 ページの「コールバックによるマルチプールのフェイルオーバの制御」を参照してください。

コールバックによるマルチプールのフェイルオーバの制御

WebLogic Server にはコールバック ハンドラを登録できます。コールバック ハンドラは、高可用性アルゴリズムのマルチプールが、その中の JDBC 接続プールからリストの次の接続プールにどのタイミングで接続要求をフェイルオーバするかを制御します。

コールバック ハンドラを使用するとフェイルオーバを行うかどうか、および行う場合にいつ行うのかを制御できるので、フェイルオーバの前にシステム上の他の準備 (データベースの準備や高可用性フレームワークとの通信) を行うことができます。

コールバック ハンドラは、config.xml ファイルにあるマルチプールの属性を介して、マルチプールごとに登録されます。したがって、コールバック ハンドラはコールバックを適用するマルチプールごとに登録する必要があります。マルチプールごとに異なるコールバック ハンドラを登録することもできます。

コールバック ハンドラの要件

マルチプール内のフェイルオーバとフェイルバックを制御するために使用するコールバック ハンドラには、weblogic.jdbc.extensions.ConnectionPoolFailoverCallback インタフェースの実装が必要です。マルチプールがリスト内の次の接続プールにフェイルオーバする必要があるとき、または以前に無効化された接続プールが利用可能になるときに、WebLogic Server は ConnectionPoolFailoverCallback インタ

フェースの allowPoolFailover() メソッドを呼び出し、3 つのパラメータ currPool、nextPool、および opcode (いずれも定義は後述) の値を渡します。その後、WebLogic Server はコールバック ハンドラからの戻り値を待ってからタスクを完了します。

アプリケーションは、下記に定義するように OK、RETRY_CURRENT、または DONOT_FAILOVER を返す必要があります。アプリケーションでは、フェイルオーバとフェイルバックのケースを処理します。

詳細については、weblogic.jdbc.extensions.ConnectionPoolFailoverCallback インタフェースの Javadoc を参照してください。

注意 : フェイルオーバのコールバック ハンドラは省略可能です。マルチプールのコンフィグレーションでコールバック ハンドラが指定されていない場合、WebLogic Server はフェイルオーバか、無効化されている接続プールを再び有効にする処理に進みます。

コールバック ハンドラのコンフィグレーション

フェイルオーバおよびフェイルバック機能に関連するマルチプールのコンフィグレーション属性には、以下の 2 つがあります。

- ConnectionPoolFailoverCallbackHandler— マルチプールのフェイルオーバ コールバック ハンドラを登録するには、この属性の値を config.xml ファイルのマルチプールのコンフィグレーションに追加します。値は、com.bea.samples.wls.jdbc.MultiPoolFailoverCallbackApplication のような絶対名でなければなりません。
- HealthCheckFrequencySeconds—WebLogic Server がマルチプールで無効化されている (非アクティブな) 接続プールをチェックして利用可能かどうかを確認する頻度を制御するには、この属性の値を config.xml ファイルのマルチプールのコンフィグレーションに追加します。詳細については、2-23 ページの「マルチプール内の失敗した接続プールが回復するときの自動的な再有効化」を参照してください。

これらの属性を config.xml ファイルに追加した後、マルチプールのエントリは次のようになります。

```
<JDBCMultiPool
AlgorithmType="High-Availability"
```

```
Name="demoMultiPool"
ConnectionPoolFailoverCallbackHandler="com.bea.samples.wls.jdbc
.MultiPoolFailoverCallbackApplication"
PoolList="demoPool2,demoPool"
HealthCheckFrequencySeconds="120"
Targets="examplesServer" />
```

注意： これらの属性は、Administration Console では表示されません。この機能を実装するには、config.xml ファイルのマルチプールのコンフィグレーションに手動でこれらの属性を追加する必要があります。

仕組み — フェイルオーバ

WebLogic Server は、現在の接続プールが接続テストに失敗したとき、または FailoverRequestIfBusy が有効になっている場合で現在の接続プールのすべての接続が使用中のときに、リスト内の次の接続プールに接続要求をフェイルオーバしようとします。

コールバック機能を有効にするには、config.xml ファイルのマルチプール コンフィグレーションの ConnectionPoolFailoverCallbackHandler 属性を使用してコールバック ハンドラを Weblogic Server に登録します。

高可用性アルゴリズムの場合、接続要求に対してリスト内の最初の接続プールから接続が提供されます。最初の接続プールの接続が接続テストに失敗すると、WebLogic Server はその接続プールに非アクティブのマークを付け、それを無効化します。コールバック ハンドラが登録されている場合、WebLogic Server は以下の情報を渡してコールバック ハンドラを呼び出し、下記のいずれかの戻り値を待ちます。

- currPool— フェイルオーバの場合、これはデータベース接続を提供するために現在使用されている接続プールの名前です。これは、「フェイルオーバ元」の接続プールです。
- nextPool—マルチプールでリストされている次に利用可能な接続プールの名前です。フェイルオーバの場合、これは「フェイルオーバ先」の接続プールです。
- opcode— 呼び出しの理由を示す、以下のいずれかのコードです。
  - OPCODE_CURR_POOL_DEAD—WebLogic Server が現在の接続プールを非アクティブと判断し、無効化しました。

- OPCODE_CURR_POOL_BUSY— 接続プールのすべてのデータベース接続が使用中です (マルチプールのコンフィグレーションで FailoverIfBusy=true が設定されている必要がある。2-24 ページの「マルチプールの使用されている接続プールのフェイルオーバの有効化」を参照)。

フェイルオーバは、接続要求と同期の関係にあります。フェイルオーバは、WebLogic Server が接続要求を満たそうとしているときにのみ行われます。

コールバック ハンドラからの戻り値は、以下の 3 つのオプションのいずれかを示します。

- OK— 処理を進めます。この場合、リストの次の接続プールにフェイルオーバすることを意味します。
- RETRY_CURRENT— 現在の接続プールで接続要求を再試行します。
- DONOT_FAILOVER— 現在の接続要求を再試行せず、フェイルオーバを行いません。weblogic.jdbc.extensions.PoolUnavailableSQLException が送出されます。

WebLogic Server は、コールバック ハンドラから返された値に基づいて動作します。

2 番目の接続プールが失敗した場合、マルチプール内に次に利用可能な接続プールがあれば WebLogic Server はそれに対してフェイルオーバを試み、前回のフェイルオーバと同じようにコールバック ハンドラを再び呼び出します。

注意 : 接続プールが手動で無効化されている場合、WebLogic Server はコールバック ハンドラを呼び出しません。

ロード バランシング アルゴリズムのマルチプールの場合、WebLogic Server は接続プールが無効化されているときにコールバック ハンドラを呼び出しません。ただし、無効化されている接続プールを再び有効化するときにはコールバック ハンドラを呼び出します。詳細については、次の節を参照してください。

コールバックによるマルチプールのフェイルバックの制御

マルチプールのフェイルオーバ コールバック ハンドラを登録すると、WebLogic Server は自動的に無効化された接続プールを再び有効化するときに同じコールバック ハンドラを呼び出します。コールバックを使用すると無効化されている接続プールが再び有効化されるかどうか、および有効化される場合はいつ有効化されるかを制御できるので、接続プールの再有効化の前にシステム上の他の準備 (データベースの準備や高可用性フレームワークとの通信) を行うことができます。

コールバック ハンドラは、`config.xml` ファイルにあるマルチプールの属性を介して、マルチプールごとに登録されます。したがって、コールバック ハンドラはコールバックを適用するマルチプールごとに登録する必要があります。マルチプールごとに異なるコールバック ハンドラを登録することもできます。

コールバック ハンドラの詳細については、以下の節を参照してください。

- 2-25 ページの「コールバック ハンドラの要件」
- 2-26 ページの「コールバック ハンドラのコンフィグレーション」

仕組み — フェイルバック

WebLogic Server は、自動的に無効化されているマルチプール内の接続プールの状態を定期的にチェックします (2-23 ページの「マルチプール内の失敗した接続プールが回復するときの自動的な再有効化」を参照)。無効化されている接続プールが利用可能になり、フェイルオーバ コールバック ハンドラが登録されている場合、WebLogic Server は以下の情報を渡してコールバック ハンドラを呼び出し、戻り値を待ちます。

- `currPool`— フェイルバックの場合、これは以前に無効化され、現在は利用可能で再有効化できる接続プールの名前です。
- `nextPool`— フェイルバックの場合、これは null です。
- `opcode`— 呼び出しの理由を示す、以下のいずれかのコードです。フェイルバックの場合、コードは常に `OPCODE_REENABLE_CURR_POOL` です。 このコードは、`currPool` の接続プールが現時点で利用可能であることを示します。

フェイルバック (無効化されている接続プールの自動的な再有効化) は、フェイルオーバとは異なります。フェイルオーバは接続要求と同期的な関係にありますが、フェイルオーバは非同期です。

コールバック ハンドラは、以下のいずれかの値を返します。

- OK— 処理を進めます。この場合は、指定された接続プールを再び有効化することを意味します。WebLogic Server は、マルチプールのアルゴリズムと、接続プールのリストにおける位置に基づいて、その接続プールへの接続要求の転送を再開します。
- DONOT_FAILOVER—currPool の接続プールを再有効化しません。使用中の接続プールから引き続き接続要求に対処します。

WebLogic Server は、コールバック ハンドラから返された値に基づいて動作します。

コールバック ハンドラが DONOT_FAILOVER を返す場合、WebLogic Server はマルチプール コンフィグレーションの HealthCheckFrequencySeconds 属性で決められたとおりに次のテスト サイクルで接続プールを再び有効化しようとし、そのプロセスの過程でコールバック ハンドラを呼び出します。

マルチプールで接続プールがリストされる順序はとても重要です。高可用性アルゴリズムのマルチプールは常に、接続プールのリストの最初に利用可能な接続プールから接続要求に対応しようとします。以下のシナリオを検討してください。

MultiPool_1 は高可用性アルゴリズムを使用し、ConnectionPoolFailoverCallbackHandler が登録されていて、3 つの接続プール CP1、CP2、および CP3 をこの順序で保持しています。

CP1 が無効になり、MultiPool_1 は接続要求を CP2 にフェイルオーバします。

次に CP2 が無効になり、MultiPool_1 は接続要求を CP3 にフェイルオーバします。

しばらくして、CP1 が再び利用可能になり、コールバック ハンドラによって WebLogic Server が接続プールを再有効化できるようになります。それ以降の接続要求には、CP1 から接続が提供されるようになります。これは、CP1 がマルチプールで最初にリストされている接続プールであるためです。

`CP2` が続いて利用可能になり、コールバック ハンドラが WebLogic Server による接続プールの再有効化を可能にしても、接続要求は引き続き `CP1` から接続を提供されます。これは、`CP1` が接続プールのリストで `CP2` より前に位置しているためです。

マルチプール フェイルオーバの制限事項と要件

WebLogic Server はマルチプール用の高可用性アルゴリズムを備えており、接続プールで障害 (データベース管理システムのクラッシュなど) が発生しても、システムをそのまま稼動させることができます。ただし、システムをコンフィグレーションするときには以下の制限と要件を考慮する必要があります。

フェイルオーバを有効にするための予約時の接続のテスト

接続プールでは、いつデータベース接続が失われたかを識別するために `TestConnectionsOnReserve` 機能を使用します。接続は、アプリケーションによって予約されるまで、自動的にはテストされません。マルチプール内の接続プールに `TestConnectionsOnReserve=true` を設定する必要があります。この機能をオンにすると、各接続はアプリケーションに戻される前にテストされます。これは、高可用性アルゴリズムにおいて不可欠な処理です。高可用性アルゴリズムでは、マルチプール内の次の接続プールにフェイルオーバするタイミングが、予約時の接続テストの結果に基づいて決定されます。テストが失敗すると、接続プールによって接続が再作成されます。この再作成にも失敗すると、マルチプールが次の接続プールにフェイルオーバします。マルチプールのフェイルオーバの拡張の詳細については、2-22 ページの「マルチプールのフェイルオーバの拡張」を参照してください。

使用中の接続ではフェイルオーバは行われない

予約後に接続が失敗する可能性もありますが、これについてはアプリケーションで処理する必要があります。WebLogic Server では、アプリケーションで使用中に失敗した接続をフェイルオーバさせることはできません。接続の使用中に障害が発生した場合は、トランザクションをやり直す必要があり、こうした障害が処理できるようにコーディングしておく必要があります。

DataSource のコンフィグレーションと使い方

接続プールやマルチプールと同様に、DataSource オブジェクトは Administration Console または WebLogic 管理 API を使用して作成できます。DataSource オブジェクトを定義して、トランザクション サービスを有効または無効にできます。DataSource のプール名属性を定義する前に、接続プールとマルチプールをコンフィグレーションします。

DataSource オブジェクトを JNDI と組み合わせると、データベースへの接続を提供する接続プールにアクセスできます。個々の DataSource は、1 つの接続プール、またはマルチプールを参照できます。ただし、単一の接続プールを使用する複数の DataSource を定義できます。これにより、同じデータベースを共有するトランザクション対応 DataSource オブジェクトとトランザクション非対応 DataSource オブジェクトの両方を定義できます。

WebLogic Server では、2 種類の DataSource オブジェクトがサポートされます。

- DataSource (ローカル トランザクションのみ)
- TxDataSource (分散トランザクション)

アプリケーションで次のいずれかの基準が該当する場合は、WebLogic Server で TxDataSource を使用してください。

- Java Transaction API (JTA) を使用している
- WebLogic Server EJB コンテナを使用して、トランザクションを管理している
- 単一のトランザクション中に、何度もデータベースが更新されている

TxDataSource の使い方、およびコンフィグレーションの方法については、『管理者ガイド』の「接続プール、マルチプール、およびデータソースの JDBC コンフィグレーション ガイドライン」を参照してください。

アプリケーションで DataSource を使用して、接続プールからデータベース接続を取得する (推奨) 場合は、アプリケーションを実行する前に、Administration Console で DataSource 定義します。DataSource の作成手順については、Administration Console オンライン ヘルプを参照してください。TxDataSource の作成手順については、Administration Console オンライン ヘルプを参照してください。

DataSource オブジェクトにアクセスするパッケージのインポート

アプリケーションで DataSource オブジェクトを使用するには、以下のクラスをクライアント コードにインポートします。

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

DataSource を使用したクライアント接続の取得

JDBC クライアントから接続を取得するには、以下のコードに示すように、Java Naming and Directory Interface (JDNI) ルックアップを使用して DataSource オブジェクトを見つけます。

```
Context ctx = null;
  Hashtable ht = new Hashtable();
  ht.put(Context.INITIAL_CONTEXT_FACTORY,
         "weblogic.jndi.WLInitialContextFactory");
  ht.put(Context.PROVIDER_URL,
         "t3://hostname:port");

  try {
    ctx = new InitialContext(ht);
    javax.sql.DataSource ds
      = (javax.sql.DataSource) ctx.lookup ("myJtsDataSource");
   java.sql.Connection conn = ds.getConnection();

// これで conn オブジェクトを使用して
// 文を作成し、結果セットを検索できる
```

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

// 終了したら文と接続オブジェクトをクローズする

   stmt.close();
   conn.close();
 }
  catch (NamingException e) {
// エラー発生
  }
  finally {
    try {ctx.close();}
  catch (Exception e) {
// エラー発生
    }
  }
```

(使用する WebLogic Server に合わせて適切な hostname と port 番号に置き換えます。)

注意： 上のコードでは、JNDI コンテキストを取得するためにいくつかの使用可能なプロシージャが使用されています。JNDI の詳細については、『WebLogic JNDI プログラマーズ ガイド』を参照してください。

コード例

WebLogic Server の samples/examples/jdbc/datasource ディレクトリに収められている DataSource コード例を参照してください。

JDBC データ ソース ファクトリ

WebLogic Server では、JDBC データ ソース リソースを、リソース ファクトリとして WebLogic Server JNDI ツリーにバインドできます。その後、EJB デプロイメント記述子のリソース ファクトリ参照を、実行中の WebLogic Server の利用可能なリソース ファクトリにマップすると、接続プールから接続を取得できます。

JDBC データ ソース ファクトリの作成、および使用については、『WebLogic エンタープライズ JavaBeans プログラマーズ ガイド』の「リソース ファクトリ」を参照してください。

3　JDBC アプリケーションのパフォーマンス チューニング

以下の節では、JDBC アプリケーションを最大限に活用する方法について説明します。

- 3-1 ページの「JDBC パフォーマンスの概要」
- 3-1 ページの「WebLogic のパフォーマンス向上機能」
- 3-3 ページの「ベスト パフォーマンスのためのアプリケーション設計」

JDBC パフォーマンスの概要

Java、JDBC、および DBMS 処理に関連する概念は、多くのプログラマにとって未知のものです。Java がさらに普及していけば、データベース アクセスとデータベース アプリケーションの実装はより簡単になります。このドキュメントでは、JDBC アプリケーションから最高のパフォーマンスを引き出すためのヒントを紹介します。

WebLogic のパフォーマンス向上機能

WebLogic には、JDBC アプリケーションのパフォーマンスを向上させるための機能がいくつか用意されています。

接続プールによるパフォーマンスの向上

DBMS への JDBC 接続を確立するには非常に時間がかかる場合があります。JDBC アプリケーションでデータベース接続のオープンとクローズを繰り返す必要がある場合、これは重大なパフォーマンスの問題となります。WebLogic 接続プールは、こうした問題を効率的に解決します。

WebLogic Server を起動すると、接続プール内の接続が開き、すべてのクライアントが使用できるようになります。クライアントが接続プールの接続をクローズすると、その接続はプールに戻され、他のクライアントが使用できる状態になります。つまり、接続そのものはクローズされません。プール接続のオープンとクローズには、ほとんど負荷がかかりません。

どのくらいの数の接続をプールに作成すればよいでしょうか。接続プールの数は、コンフィグレーションされたパラメータに従って最大数と最小数の間で増減させることができます。常に最高のパフォーマンスが得られるのは、同時ユーザと同じくらいの数の接続が接続プールに存在する場合です。

Prepared Statement とデータのキャッシング

DBMS のアクセスでは大量にリソースを消費します。プログラムで Prepared Statement を再利用する場合、または複数のアプリケーションでの共有や、各接続どうしでの存続が可能な頻繁に使用するデータにアクセスする場合は、以下のものを使用してデータをキャッシュできます。

- 接続プールの Prepared Statement キャッシュ
- 読み込み専用のエンティティ Bean
- クラスタ環境での JNDI

ベスト パフォーマンスのためのアプリケーション設計

データベース アプリケーションのパフォーマンスの良し悪しはほとんどの場合、アプリケーション言語ではなく、アプリケーションがどのように設計されているかによって決定まります。クライアントの数と場所、DBMS テーブルおよびインデックスのサイズと構造、およびクエリの数とタイプは、すべてアプリケーションのパフォーマンスに影響を与えます。

以下では、すべての DBMS に当てはまる一般的なヒントを示します。また、アプリケーションで使用する特定の DBMS のドキュメントによく目を通しておくことも重要です。

1. データをできるだけデータベースの内部で処理する

DBMS アプリケーションのパフォーマンスに関する最も深刻な問題は、生データを不必要に移動することから発生します。これは、生データをネットワーク上で移動する場合にも、単に DBMS のキャッシュに出し入れする場合にも言えることです。こうした無駄を最小限に抑えるための良い方法は、クライアントが DBMS と同じマシンで動作している場合でも、ロジックをクライアントではなくデータの格納場所、つまり DBMS に置くことです。実際のところ、一部の DBMS では、1 個の CPU を共有するファット クライアントとファット DBMS はパフォーマンスの致命的な低下をもたらします。

大部分の DBMS は、ストアド プロシージャという、データの格納場所にロジックを置くための理想的なツールを備えています。ストアド プロシージャを呼び出して 10 個の行を更新するクライアントと、同じ行を取得および変更し、UPDATE 文を送信してその変更を DBMS に保存するクライアントの間には、パフォーマンスに大きな違いがあります。

また、DBMS のドキュメントを参照して、DBMS 内のキャッシュ メモリの管理について調べる必要もあります。一部の DBMS (Sybase など) は、DBMS に割り当てられた仮想メモリを分割し、特定のオブジェクトがキャッシュの固定領域を独占的に使用できるようにする機能を備えています。この機能を使用すると、重要なテーブルまたはインデックスをディスクから一度読み出しておくことで、ディスクに再度アクセスしなくてもすべてのクライアントがそれらを使用できるようになります。

2. 組み込み DBMS セットベース処理を使用する

SQL は、セット処理言語です。DBMS は、完全にセットベース処理を行うように設計されています。データベースへの 1 行へのアクセスは、例外なくセットベースの処理より遅く、また DBMS によっては実装が不完全です。たとえば、従業員 100 名に関するデータが格納されている 4 つのテーブルがある場合、全従業員について各テーブルを一度に更新する方が、従業員 1 名ごとに各テーブルを 100 回更新するより常に高速です。

あまりに複雑すぎて 1 行ずつ処理する以外に方法がないと考えられていた処理の多くが、セットベースの処理に書き換えられ、パフォーマンスの向上を実現しています。たとえば、ある有名な給与管理アプリケーションは、巨大で低速な COBOL アプリケーションから、連続実行される 4 つのストアド プロシージャに変換されました。この結果、マルチ CPU マシンで何時間もかかった処理が、より少ないリソースで 15 分で実行できるようになりました。

3. クエリを効率化する

ユーザからよく尋ねられる質問に、「特定の結果セットで返される行数はどのくらいか」というものがあります。すべての行を取り出さずに調べる唯一の方法は、次のように *count* キーワードを使用して同じクエリを発行することです。

```
SELECT count(*) from myTable, yourTable where ...
```

これにより、関連するデータには変更がなかった場合に、オリジナルのクエリが戻すべき行数が返されます。また関連するデータに影響を与えるその他の DBMS アクティビティが起きた場合に、クエリを実行すれば、実際の行数は変わります。

ただし、これはリソースを大量に消費する処理であることに注意してください。元のクエリによっては、DBMS は行を送信するのと同じくらいの処理を行って行をカウントする必要があります。

アプリケーションのクエリは、実際にどのようなデータが必要なのかをできる限り具体的に指定する必要があります。たとえば、まず一時テーブルに抽出し、カウントだけを返し、次に限定された 2 番目のクエリを送信して一時テーブル内の行のサブセットだけを返すようにします。

クライアントが本当に必要なデータだけを抽出することが、きわめて重要です。ISAM (リレーショナル データベース以前のアーキテクチャ) から移植された一部のアプリケーションでは、実際に必要なのは最初の数行だけであっても、テーブル内のすべての行を選択するクエリが送信されます。また、最初に取得する行を得るために「sort by」句を使用するアプリケーションもあります。このようなデータベース クエリは、パフォーマンスを不必要に低下させます。

SQL を適切に使用すると、こうしたパフォーマンス上の問題を回避できます。たとえば、高額給与の社員のうち上位 3 人だけのデータが必要な場合、クエリを適切に行うには、相関サブクエリを使用します。表 3-1 に SQL 文が返す全体の表を示します。

```
select * from payroll
```

表 3-1 返された完全な結果

名前	給与
Joe	10
Mikes	20
Sam	30
Tom	40
Jan	50
Ann	60
Sue	70
Hal	80

表 3-1 返された完全な結果

名前	給与
May	80

相関サブクエリ

```
select p.name, p.salary from payroll p
where 3 >= (select count(*) from payroll pp
where pp.salary >= p.salary);
```

表 3-2 に示すように、このクエリでは、より小さい結果が返されます。

表 3-2 サブクエリの結果

名前	給与
Sue	70
Hal	80
May	80

このクエリでは、上位 3 名の高所得者の名前と給与が登録された 3 行だけが返されます。このクエリでは、給与テーブル全体をスキャンし、次に各行について内部ループで給与テーブル全体を再スキャンして、ループの外でスキャンした現在の行より高額の給与が何件あるかを調べます。この処理は複雑なように見えるかもしれませんが、DBMS はこの種の処理では SQL を効率的に使用するように設計されています。

4. トランザクションを単一バッチにする

可能な限り、一連のデータ処理を収集し、更新トランザクションを次のような単一の文で発行してください。

```
BEGIN TRANSACTION
```

```
    UPDATE TABLE1...

    INSERT INTO TABLE2

    DELETE TABLE3

COMMIT
```

この方法により、別個の文とコミットを使用するよりパフォーマンスが向上します。バッチ内で条件ロジックと一時テーブルを使用する場合でも、DBMS はさまざまな行とテーブルに必要なすべてのロックを取得し、ワン ステップで使用および解放するので、この方法は望ましいと言えます。別個の文とコミットを使用すると、クライアントと DBMS 間の転送が増加し、DBMS 内のロック時間が長くなります。こうしたロックにより、他のクライアントはそのデータにアクセスできなくなり、複数の更新がさまざまな順序でテーブルを更新できるかによって、デッドロックが発生する可能性があります。

警告： ユニーク キー制約の違反などによって上記のトランザクション中の任意の文が適切に実行されなかった場合は、条件 SQL ロジックを追加して文の失敗を検出し、トランザクションをコミットせずにロール バックする必要があります。上の例の場合、INSERT 文が失敗すると、ほとんどの DBMS は挿入の失敗を示すエラー メッセージを返しますが、2 番目と 3 番目の文の間でメッセージを取得したかのように動作して、コミットが行われてしまいます。Microsoft SQL Server には、SQL `set xact_abort on` の実行によって有効となる接続オプションがあります。このオプションを使用すると、文が失敗した場合にトランザクションが自動的にロールバックされます。

5. DBMS トランザクションがユーザ入力に依存しないようにする

アプリケーションが、`'BEGIN TRAN'` と、更新のために行またはテーブルをロックする SQL を送信する場合、ユーザのキー入力がなければトランザクションをコミットできないようにアプリケーションを設計してはなりません。ユーザがキー入力をせずに昼食に出かけてしまうと、ユーザが戻ってくるまで DBMS 全体がロックされてしまいます。

トランザクションの作成と完了にユーザ入力が必要な場合は、オプティミスティック ロックを使用します。簡単に言えば、オプティミスティック ロックではクエリと更新でタイムスタンプとトリガが使用されます。クエリは、トランザクション中にデータをロックすることなく、タイムスタンプ値を持つデータを選択し、そのデータに基づいてトランザクションを準備します。

更新トランザクションがユーザ入力によって定義されると、そのデータはタイムスタンプと共に単一の送信として送られます。これにより、そのデータが最初に取り出したデータと同じであることを確認できます。トランザクションが正常に実行されると、変更されたデータのタイムスタンプが更新されます。別のユーザからの更新トランザクションによって現在のトランザクションが処理するデータが変更された場合、タイムスタンプが変更され、現在のトランザクションは拒否されます。ほとんどの場合、関連データが変更されることはないので通常トランザクションは正常に実行されます。あるトランザクションが失敗すると、アプリケーションは更新されたデータをリフレッシュし、必要に応じてトランザクションを再作成するよう通知します。

6. 同位置更新を使用する

データ行の同位置での更新は、行の移動より非常に高速です。行の移動は、更新処理でテーブル設計の許容範囲を越えるスペースが必要な場合に行う必要があります。必要なスペースを持つ行を最初から設計しておけば、更新は早くなります。ただし、テーブルに必要なディスク空間は大きくなります。ディスク スペースのコストは低いので、パフォーマンスが向上するのであれば、使用量を少しだけ増やすことは価値ある投資だと言えるでしょう。

7. 操作データをできるだけ小さくする

アプリケーションによっては、操作データを履歴データと同じテーブルに格納するものもあります。時間の経過と共に履歴データが蓄積されていくと、すべての操作クエリでは、新しいデータを取得するために (日々の作業では) 役に立たないデータを大量に読み取らなければなりません。これを回避するには、過去のデータを別のテーブルに移動し、まれにしか発生しない履歴クエリのためにこれらのテーブルを結合します。これを行うことができない場合、最も頻繁に使用されるデータが論理的および物理的に配置されるよう、テーブルをインデックス処理およびクラスタ化します。

8. パイプラインと並行処理を使用する

DBMS は、さまざまな作業を大量に処理するときに最も能力を発揮します。DBMS の最も不適切な使い方は、1 つの大規模なシングル スレッド アプリケーション用のダム ファイル ストレージとして使用することです。容易に区別できる作業サブセットを扱う大量の並行処理をサポートするようアプリケーションとデータを設計すれば、そのアプリケーションはより高速になります。処理に複数のステップがある場合、先行ステップが完了するまで次のステップを待つのではなく、いずれかの先行ステップが処理を終えた部分のデータに対して後続ステップが処理を開始できるようにアプリケーションを設計します。これは常に可能であるとは限りませんが、このことに留意してプログラムを設計すると、パフォーマンスを大幅に向上させることができます。

4 WebLogic 多層 JDBC ドライバの使い方

新しいアプリケーションでは、DataSource オブジェクトを使ってデータベース接続を取得することをお勧めします。DataSource オブジェクトを JNDI と組み合わせると、データベースへの接続を提供する接続プールにアクセスできます。JDBC 1.x API を使用した既存のアプリケーション、またはレガシー アプリケーションの場合は、WebLogic 多層 ドライバを使用して、データベース接続を取得できます。

以下の節では、WebLogic Server で多層 JDBC ドライバを使用する方法について説明します。

- 4-1 ページの「WebLogic RMI ドライバの使い方」
- 4-7 ページの「WebLogic JTS ドライバの使い方」
- 4-10 ページの「WebLogic Pool ドライバの使い方」

WebLogic RMI ドライバの使い方

WebLogic RMI ドライバは、WebLogic Server が接続プールからデータベース接続を、DataSource や TxDataSource に渡すときに使用する多層 Type 3 JDBC ドライバです。DataSource オブジェクトにより、WebLogic RMI ドライバを介してアプリケーションのデータベース接続にアクセスできます。Administration Console または WebLogic 管理 API (DBMS へのアクセスに使用する 2 層 JDBC ドライバも含め) を使用して、データベース接続パラメータを接続プールに設定します。図 1-1 を参照してください。

RMI ドライバ クライアントは、DataSource オブジェクトをルックアップすることで、DBMS への接続を確立します。このルックアップは、Java Naming and Directory Interface (JNDI) ルックアップを使うか、またはクライアントに代わって JNDI ルックアップを実行する WebLogic Server を直接呼び出すことにより実行されます。

RMI ドライバは、WebLogic t3 ドライバ (このリリースでは非推奨) と Pool ドライバの機能に取って代わるもので、独自の t3 プロトコルではなく Java 標準の Remote Method Invocation (RMI) を使用して WebLogic Server に接続します。

RMI 実装の詳細はドライバによって自動的に処理されるため、WebLogic JDBC/RMI ドライバを使用するために RMI の知識は必要ではありません。

WebLogic RMI ドライバを使用するための WebLogic Server の設定

RMI ドライバには、DataSource オブジェクトを通してだけアクセスできます。DataSource オブジェクトは、Administration Console で作成します。アプリケーションで RMI ドライバを使用するには、まず WebLogic Server コンフィグレーションに DataSource オブジェクトを作成します。Administration Console を使用してマルチプールを作成する手順については、Administration Console オンライン ヘルプを参照してください。TxDataSource の作成手順については、Administration Console オンライン ヘルプを参照してください。

RMI ドライバを使用するサンプル クライアント コード

RMI ドライバを使用して、WebLogic Server 接続プールからデータベース接続を取得し、使用する方法を以下のコード サンプルで示します。

必要なパッケージをインポートする

RMI ドライバを使用して、データベース接続を取得 / 使用する前に、次のパッケージをインポートします。

```
javax.sql.DataSource
java.sql.*
java.util.*
javax.naming.*
```

データベース接続を取得する

WebLogic JDBC/RMI クライアントは、Administration Console で定義された DataSource から DBMS への接続を取得します。クライアントは、以下の 2 通りの方法で DataSource オブジェクトを取得できます。

- JNDI ルックアップを使用します。これが最も直接的で望ましい方法です。
- `Driver.connect()` メソッドで DataSource 名を RMI ドライバに渡します。この場合、WebLogic Server はクライアントに代わって JNDI ルックアップを実行します。

JNDI ルックアップを使用した接続の取得

JNDI を使用して WebLogic RMI ドライバにアクセスするには、DataSource オブジェクトの名前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 で定義された「`myDataSource`」という DataSource にアクセスするには、以下のようにします。

```
Context ctx = null;
  Hashtable ht = new Hashtable();
  ht.put(Context.INITIAL_CONTEXT_FACTORY,
         "weblogic.jndi.WLInitialContextFactory");
  ht.put(Context.PROVIDER_URL,
         "t3://hostname:port");

  try {
    ctx = new InitialContext(ht);
    javax.sql.DataSource ds
      = (javax.sql.DataSource) ctx.lookup ("myDataSource");
   java.sql.Connection conn = ds.getConnection();
```

```
    // これで conn オブジェクトを使用して
    //  Statement オブジェクトを作成して
// SQL 文を実行し、結果セットを処理できる

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

    // 完了したら、文オブジェクトと
    //  接続オブジェクトを忘れずにクローズすること

   stmt.close();
   conn.close();
 }
  catch (NamingException e) {
// エラー発生
  }
  finally {
    try {ctx.close();}
    catch (Exception e) {
// エラー発生
    }
  }
```

(*hostname* は WebLogic Server が稼働するマシンのホスト名、*port* は WebLogic Server がリクエストをリスンするポートの番号です。)

この例では、Hashtable オブジェクトを使って、JNDI ルックアップに必要なパラメータを渡しています。JNDI ルックアップを実行する方法は他にもあります。詳細については、『WebLogic JNDI プログラマーズ ガイド』を参照してください。

ルックアップの失敗を捕捉するために JNDI ルックアップが `try/catch` ブロックで包まれている点に注意してください。また、コンテキストが `finally` ブロックの中でクローズされている点にも注意してください

WebLogic RMI ドライバだけを使用して接続を取得する

DataSource オブジェクトをルックアップしてデータベース接続を取得する代わりに、`Driver.connect()` メソッドを使用して WebLogic Server にアクセスできます。この場合は、JDBC/RMI ドライバが JNDI ルックアップを実行します。WebLogic Server にアクセスするには、WebLogic Server の URL と、DataSource オブジェクトの名前を定義するパラメータを `Driver.connect()` メソッドに渡します。たとえば、Administration Console で定義された「myDataSource」という DataSource にアクセスするには、以下のようにします。

```
java.sql.Driver myDriver = (java.sql.Driver)
  Class.forName("weblogic.jdbc.rmi.Driver").newInstance();

String url ="jdbc:weblogic:rmi";

java.util.Properties props = new java.util.Properties();
props.put("weblogic.server.url", "t3://hostname:port");
props.put("weblogic.jdbc.datasource", "myDataSource");

java.sql.Connection conn = myDriver.connect(url, props);
```

(*hostname* は WebLogic Server が稼働するマシンのホスト名、*port* は WebLogic Server がリクエストをリスンするポートの番号です。)

また、JNDI ユーザ情報を設定するために使用する以下のプロパティも定義できます。

- `weblogic.user` - ユーザ名を指定します。
- `weblogic.credential` - `weblogic.user` のパスワードを指定します。

WebLogic RMI ドライバによる行キャッシング

行キャッシングは、アプリケーションのパフォーマンスを向上するための WebLogic Server JDBC 機能です。通常、クライアントが `ResultSet.next()` を呼び出すと、WebLogic は DBMS から単一行を取得し、これをクライアント JVM に転送します。行キャッシングが有効になっていると、`ResultSet.next()` を 1 回呼び出すだけで複数の DBMS 行が取得され、これらがクライアント メモリにキャッシュされます。行キャッシングを行うと、データ取得のための通信の回数が減ることでパフォーマンスが向上します。

注意： クライアントと WebLogic Server が同じ JVM にある場合、行キャッシングは実行されません。

行キャッシングは、データ ソース属性 [行のプリフェッチを有効化] で有効にしたり無効にしたりできます。また、`ResultSet.next()` の呼び出しごとに取得される行の数は、データ ソース属性 [Row Prefetch サイズ] で設定します。データ ソース属性は、Administration Console で設定します。行キャッシングを有効にして、DataSource または TxDataSource に行のプリフェッチ サイズ属性を設定するには、次の手順に従います。

1. Administration Console の左ペインで、[サービス｜ JDBC ｜データ ソース｜トランザクション データ ソース] を選択し、行キャッシングを有効にする DataSource、または TxDataSource を選択します。
2. Administration Console の右ペインで、[コンフィグレーション] タブを選択 (まだ選択されていない場合) します。
3. [行のプリフェッチを有効化] チェックボックスを選択します。
4. [行のプリフェッチを有効化] で、`ResultSet.next()` の呼び出しごとにキャッシングする行の数を指定します。

WebLogic RMI ドライバによる行キャッシングの重要な制限事項

RMI ドライバを使用して行キャッシングを実装する場合は、以下の制限事項があることに注意してください。

- 行キャッシングは、結果セット型が TYPE_FORWARD_ONLY および CONCUR_READ_ONLY の両方である場合にのみ実行されます。
- 結果セットのデータ型によっては、その結果セットのキャッシングが無効である場合があります。これには以下が含まれます。
  - LONGVARCHAR/LONGVARBINARY
  - NULL
  - BLOB/CLOB
  - ARRAY
  - REF
  - STRUCT
  - JAVA_OBJECT
- 行キャッシングが有効で、その結果セットに対してアクティブな場合、一部の ResultSet メソッドはサポートされません。そのほとんどは、ストリーミング データ、スクロール可能な結果セット、または行キャッシングがサポートされていないデータ型に関係しています。これには以下が含まれます。
  - `getAsciiStream()`
  - `getUnicodeStream()`

- getBinaryStream()
- getCharacterStream()
- isBeforeLast()
- isAfterLast()
- isFirst()
- isLast()
- getRow()
- getObject (Map)
- getRef()
- getBlob()/getClob()
- getArray()
- getDate()
- getTime()
- getTimestamp()

WebLogic JTS ドライバの使い方

JTS (Java Transaction Services) ドライバは、WebLogic Server 内で実行中のアプリケーションから接続プールや SQL トランザクションへのアクセスを提供する、サーバ サイド Java JDBC (Java Database Connectivity) ドライバです。データベースへの接続は接続プールから行われ、アプリケーションに代わってデータベース管理システム (DBMS) に接続するために WebLogic Server 内で実行される 2 層 JDBC ドライバを使用します。

トランザクションが開始されると、同じ接続プールから接続を取得する実行スレッドのすべてのデータベース操作は、そのプールの同じ接続を共有することになっています。これらの操作は、エンタープライズ JavaBean (EJB) や Java Messaging Service (JMS) のようなサービスを通じて、または標準 JDBC 呼び出しを使用して直接 SQL を送信することにより行うことができます。デフォルトでは、これらすべての操作は同じ接続を共有し、同じトランザクションに参加します。トランザクションがコミットまたはロールバックされると、接続はプールに戻されます。

Java クライアントは JTS ドライバ自身を登録しない場合もありますが、Remote Method Invocation (RMI) を介してトランザクションに参加することができます。あるクライアントの 1 つのスレッド内でトランザクションを開始し、そのクライアントにリモート RMI オブジェクトを呼び出させることができます。リモート オブジェクトによって実行されるデータベース操作は、そのクライアント上で開始されたトランザクションの一部分になります。そのリモート オブジェクトがそれを呼び出したクライアントに戻されたら、そのトランザクションをコミットまたはロールバックできます。リモート オブジェクトによって実行されるデータベース操作は、すべて同一の接続プールを使用しなければならず、同一のトランザクションの一部にならなければなりません。

JTS ドライバを使用するサンプル クライアント コード

JTS ドライバを使用するには、まず Administration Console を使用して WebLogic Server に接続プールを作成しなければなりません。詳細については、2-2 ページの「接続プールのコンフィグレーションと使い方」を参照してください。

次に、サーバサイド アプリケーションから JTS トランザクションを作成して使用する方法について説明します。ここでは、「`myConnectionPool`」という接続プールを使用します。

1. 以下のクラスをインポートします。

```
import javax.transaction.UserTransaction;
import java.sql.*;
import javax.naming.*;
import java.util.*;
import weblogic.jndi.*;
```

2. `UserTransaction` クラスを使用してトランザクションを確立します。JNDI ツリー上でこのクラスをルックアップできます。`UserTransaction` クラスは、現在の実行スレッド上のトランザクションを制御します。このクラスはトランザクション自身を表さないことに注意してください。このトランザクションの実際のコンテキストは、現在の実行スレッドに関連付けられています。

```
Context ctx = null;
Hashtable env = new Hashtable();
```

```
env.put(Context.INITIAL_CONTEXT_FACTORY,
         "weblogic.jndi.WLInitialContextFactory");

// WebLogic Server の パラメータ
// 環境に合わせて適切なホスト名、ポート番号、
// ユーザ名、およびパスワードに置き換える
env.put(Context.PROVIDER_URL, "t3://localhost:7001");
env.put(Context.SECURITY_PRINCIPAL, "Fred");
env.put(Context.SECURITY_CREDENTIALS, "secret");

ctx = new InitialContext(env);

UserTransaction tx = (UserTransaction)
  ctx.lookup("javax.transaction.UserTransaction");
```

3. 現在のスレッドのトランザクションを開始します。

```
tx.begin();
```

4. JTS ドライバをロードします。

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

5. 接続プールから接続を取得します。

```
Properties props = new Properties();
props.put("connectionPoolID", "myConnectionPool");

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

6. データベース操作を実行します。これらの操作は、EJB、JMS、および標準 JDBC 文など、データベース接続を使用するどのサービスでも実行できます。これらの操作では、JTS ドライバを使用して、手順 3 で開始したトランザクションと同じ接続プールにアクセスすることにより、このトランザクションに参加する必要があります。

   JTS ドライバを使用した追加データベース操作が、手順 5 で指定した接続プールとは違う接続プールを使用する場合、そのトランザクションをコミットまたはロールバックしようとすると例外が発生します。

7. 接続オブジェクトをクローズします。接続をクローズしても、それでトランザクションがコミットされるわけでも、その接続がプールに戻されるわけでもないことに注意してください。

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

8. 他のデータベース操作を実行します。これらの操作が同じ接続プールへの接続によって行われるのであれば、それらの操作はプールから同じ接続を使用し、このスレッド内の他のすべての操作と同じ `UserTransaction` の一部となります。

9. そのトランザクションをコミットまたはロールバックすることにより、トランザクションを完了します。JTS ドライバは、現在のスレッドに存在するすべての接続オブジェクトのすべてのトランザクションをコミットし、接続をプールに返します。

```
tx.commit();

// または

tx.rollback();
```

WebLogic Pool ドライバの使い方

WebLogic Pool ドライバを使用すると、HTTP サーブレットや EJB などのサーバサイド アプリケーションから接続プールを利用できます。Pool ドライバの使い方については、『WebLogic HTTP サーブレット プログラマーズ ガイド』の「プログラミング タスク」の「データベースへのアクセス」を参照してください。

5 WebLogic Server でのサードパーティドライバの使い方

以下の節では、サードパーティ JDBC ドライバの設定および使用方法について説明します。

- 5-1 ページの「サードパーティ JDBC ドライバの概要」
- 5-4 ページの「サードパーティの JDBC ドライバに対する環境設定」
- 5-14 ページの「サードパーティ ドライバを使用した接続の取得」
- 5-19 ページの「Oracle 拡張機能と Oracle Thin Driver の使用」
- 5-37 ページの「Oracle 仮想プライベート データベースによるプログラミング」
- 5-39 ページの「Oracle 拡張機能インタフェースとサポートされるメソッドの表」

サードパーティ JDBC ドライバの概要

WebLogic Server は、以下の機能を提供するサードパーティ JDBC ドライバと連携して機能します。

- スレッドセーフ
- 標準の JDBC 文を使用してトランザクションの実装が可能

この節では、以下のサードパーティ JDBC ドライバを WebLogic Server で設定して使用する方法について説明します。

- Oracle Thin Driver 8.1.7、9.0.1、9.2.0、または 10g (インストールされている WebLogic Server に同梱)

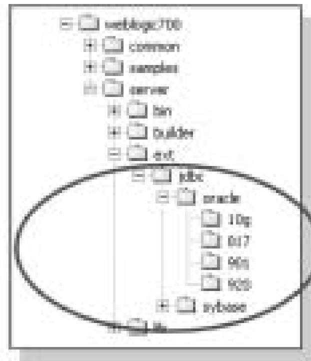
- Sybase jConnect Driver 4.5 および 5.5 (インストールされている WebLogic Server に同梱)
- IBM Informix JDBC Driver
- Microsoft SQL Server Driver for JDBC

WebLogic Server 6.1 では、Oracle Thin Driver と Sybase jConnect Driver は、`weblogic.jar` にバンドルされています。バージョン 7.x では、サードパーティの JDBC ドライバは `weblogic.jar` のバンドルから除外されます。代わりに、Oracle Thin Driver 10g (`classes12.zip`) と、Sybase jConnect Driver の 4.5 (`jConnect.jar`) および 5.5 (`jconn2.jar`) の各バージョンが `weblogic.jar` で *`WL_HOME`*`\server\lib` フォルダにインストールされます (なお *`WL_HOME`* は WebLogic Platform がインストールされるフォルダ)。`weblogic.jar` のマニフェストには、これらのファイルがリストされており、`weblogic.jar` のロード時 (サーバの起動時) にロードできます。

注意 : WebLogic Server 7.0SP5 では、Oracle Thin Driver のデフォルト バージョンは 10g ドライバ (*`WL_HOME`*`\server\lib` のバージョン) に変更されました。WebLogic Server 7.0SP2、SP3、および SP4 では、Oracle Thin Driver 9.2.0 がドライバのデフォルト バージョンでした。サービス パック 2 より前のリリースの WebLogic Server 7.0 では、Oracle Thin Driver のデフォルトのバージョンは 8.1.7 でした。

インストールされた WebLogic Server の *`WL_HOME`*`\server\ext\jdbc` フォルダ (*`WL_HOME`* は WebLogic Platform がインストールされるフォルダ) には、Oracle と Sybase の 各 JDBC ドライバのサブフォルダが入っています。図 5-1 を参照してください。

図 5-1 WebLogic Server と共にインストールされる JDBC ドライバのディレクトリ構造



`oracle` フォルダには、バージョン 10g を含む Oracle Thin Driver の各バージョンが入っています (前述のとおり、*WL_HOME*`\server\lib` フォルダにもある)。これらのファイルは、*WL_HOME*`\server\lib` フォルダにコピーすることにより、Oracle Thin Driver のバージョンを変更したり、デフォルトのバージョンに戻したりできます。詳細については、5-5 ページの「Oracle Thin Driver の変更または更新」を参照してください。

`sybase` フォルダには、Sybase jConnect Driver 4.5 と Sybase jConnect Driver 5.5 および各サポート ファイルを含むサブフォルダがあります。前述のとおり、各ドライバ (jConnect.jar と jconn2.jar) は、*WL_HOME*`\server\lib` フォルダに、ディレクトリ構造やサポート ファイルなしで、インストールされます。WebLogic Server の実行中は、*WL_HOME*`\server\lib` フォルダにあるファイルが使用されます。不良なドライバやサポート対象外のドライバの更新時に、バックアップとして *WL_HOME*`\server\ext\jdbc\`sybase フォルダにある追加コピーを利用できます。

これらのドライバのデフォルト バージョンを使用する場合は、いずれの変更も必要ありません。また、別のバージョンのドライバを使用する場合は、*WL_HOME*`\server\lib` のファイルを *WL_HOME*`\server\ext\jdbc\oracle\`*version* (*version* は使用する JDBC ドライバのバージョン) のファイル、または DBMS ベンダ (Oracle または Sybase) のファイルに置き換えます。

`weblogic.jar` のマニフェストには、*WL_HOME*\server\lib にある Oracle Thin Driver と Sybase jConnect Driver のクラス ファイルがリストされているため、`weblogic.jar` のロード時 (サーバの起動時) に、各ドライバがロードされます。したがって、CLASSPATH に JDBC ドライバを追加する必要はありません。また、WebLogic Server にはインストールされていないサードパーティの JDBC ドライバを使用する場合は、CLASSPATH にドライバ ファイルのパスを追加します。

サードパーティの JDBC ドライバに対する環境設定

WebLogic Server 7.0 には含まれていない Oracle Thin Driver または Sybase jConnect Driver 以外のサードパーティの JDBC ドライバを使用する場合は、CLASSPATH に JDBC のドライバ クラスに対するパスを追加する必要があります。サードパーティの JDBC ドライバを使用するときに Windows、および UNIX の環境に合わせた CLASSPATH を設定する方法について以下の節で説明します。

Windows でのサードパーティ JDBC ドライバの CLASSPATH

次のように、CLASSPATH に JDBC ドライバ クラスと `weblogic.jar` へのパスを指定します。

```
set CLASSPATH=DRIVER_CLASSES;WL_HOME\server\lib\weblogic.jar;
%CLASSPATH%
```

ここで、*DRIVER_CLASSES* は、JDBC ドライバ クラスへのパス、*WL_HOME* は、WebLogic Platform をインストールするディレクトリを表します。

UNIX でのサードパーティ JDBC ドライバの CLASSPATH

次のように、CLASSPATH に JDBC ドライバ クラスと weblogic.jar へのパスを追加します。

```
export CLASSPATH=DRIVER_CLASSES:WL_HOME/server/lib/weblogic.jar:
$CLASSPATH
```

ここで、*DRIVER_CLASSES* は、JDBC ドライバ クラスへのパス、*WL_HOME* は、WebLogic Platform をインストールするディレクトリを表します。

Oracle Thin Driver の変更または更新

WebLogic Server にバンドルされた Oracle Thin Driver 10g (10.1.0.2.0) は事前にコンフィグレーションされ、そのまま使用できる状態になっています。別のバージョンを使用する場合は、*WL_HOME*\server\lib\classes12.zip を該当するバージョンのファイルに置き換えてください。たとえば、Oracle Thin Driver 9.2.0 を使用する場合は、*WL_HOME*\server\ext\jdbc\oracle\920 フォルダから classes12.zip をコピーし、それを *WL_HOME*\server\lib に置いて、そのフォルダにあるバージョン 10g と置き換える必要があります。

注意： WebLogic Server 7.0SP5 では、Oracle Thin Driver のデフォルト バージョンは 10g ドライバ (*WL_HOME*\server\lib にあるバージョン) に変更されました。WebLogic Server 7.0SP2、SP3、および SP4 では、Oracle Thin Driver 9.2.0 がドライバのデフォルト バージョンでした。サービス パック 2 より前のリリースの WebLogic Server 7.0 では、Oracle Thin Driver のデフォルトのバージョンは 8.1.7 でした。

Oracle Thin Drive 9.2.0、9.0.1、または 8.1.7 を使用する場合は、次の手順に従います。

1. Windows エクスプローラ、またはコマンド シェルで、使用するドライバのバージョンのフォルダに移動します。
   - *WL_HOME*\server\ext\jdbc\oracle\920
   - *WL_HOME*\server\ext\jdbc\oracle\901

- *WL_HOME*\server\ext\jdbc\oracle\817

2. classes12.zip をコピーします。
3. Windows エクスプローラ、またはコマンド シェルで、*WL_HOME*\server\lib に移動し、既存のバージョン、classes12.zip を、コピーしたバージョンに置き換えます。

バージョン 10g (デフォルト) に戻る場合は、上記の説明に従い、フォルダ *WL_HOME*\server\ext\jdbc\oracle\10g からコピーします。

Oracle Thin Driver のバージョンを Oracle の新バージョンで更新する場合は、*WL_HOME*\server\lib の classes12.zip を Oracle が提供する新しいファイルで置き換えてください。ドライバのアップデートは、次の Oracle Web サイトからダウンロードできます。http://otn.oracle.com/software/content.html

注意 : CLASSPATH に Oracle Thin Driver の複数のバージョンを指定することはできません。さまざまなメソッドでクラッシュが発生する可能性があります。

Oracle Thin Driver 9.x および 10g でのパッケージの変更

Oracle 8.x 以前のリリースでは、Oracle Thin Driver の含まれるパッケージは oracle.jdbc.driver でした。Oracle 8.1.7 Thin Driver を使用する JDBC 接続プールをコンフィグレーションする際には、DriverName (ドライバのクラス名) を oracle.jdbc.driver.OracleDriver として指定します。Oracle 9.x と 10g では、Oracle Thin Driver の含まれるパッケージは oracle.jdbc です。Oracle 9.x または 10g Thin Driver を使用する JDBC 接続プールをコンフィグレーションする際には、DriverName (ドライバのクラス名) を oracle.jdbc.OracleDriver として指定します。oracle.jdbc.driver.OracleDriver クラスは 9.x と 10g のドライバで使用できますが、このクラスは今後、機能の拡張が行われない場合もあります。

Oracle Thin Driver の詳細については、Oracle のマニュアルを参照してください。

注意 : パッケージの変更は、XA バージョンのドライバには関係ありません。Oracle Thin Driver の XA バージョンの場合、JDBC の接続プールで DriverName (ドライバのクラス名) としては oracle.jdbc.xa.client.OracleXADataSource を使用します。

nls_charset12.zip による文字セットのサポート

Oracle Thin ドライバには、Oracle のオブジェクト型またはコレクション型の一部として取得または挿入されない、すべての CHAR および NCHAR データ型の Oracle 文字セットに対するグローバリゼーション サポートが含まれています。

ただし、Oracle オブジェクトまたはコレクションの CHAR および VARCHAR データの部分に関しては、Oracle Thin ドライバには以下の文字セットのグローバリゼーション サポートのみ含まれています。

- US7ASCII
- WE8DEC
- ISO-LATIN-1
- UTF-8

Oracle のオブジェクト型またはコレクションの中の CHAR および NCHAR データで他の文字セットを使用する場合は、`CLASSPATH` に `nls_charset.zip` を含める必要があります。このファイルが `CLASSPATH` にない場合、次の例外が発生します。

```
java.sql.SQLException: Non supported character set:
oracle-character-set-178
```

`nls_charset12.zip` ファイルは WebLogic Server と共に *`WL_HOME`*`\server\ext\jdbc\oracle\920` および *`WL_HOME`*`\server\ext\jdbc\oracle\10g` フォルダにインストールされます (*`WL_HOME`* は WebLogic Server のインストール先フォルダです)。`CLASSPATH` の設定手順については、5-4 ページの「サードパーティの JDBC ドライバに対する環境設定」を参照してください。

Sybase jConnect Driver の更新

WebLogic Server にバンドルされた Sybase jConnect Driver 4.5 および 5.5 は事前にコンフィグレーションされ、そのまま使用できる状態になっています。別のバージョンを使用する場合は、*`WL_HOME`*`\server\lib\jConnect.jar` または `jconn2.jar` を DBMS ベンダが提供する別バージョンのファイルに置き換えてください。

WebLogic Server でインストールされたバージョンに戻す場合は、*WL_HOME*\server\lib フォルダに次のファイルをコピーします。

- *WL_HOME*\server\ext\jdbc\sybase\jConnect.jar
- *WL_HOME*\server\ext\jdbc\sybase\jConnect-5_5\classes\jconn2.jar

IBM Infomix JDBC Driver のインストールと使い方

WebLogic Server と Infomix データベースを使用する場合は、IBM Informix JDBC Driver を使用することをお勧めします。このドライバは、次の IBM Web サイトからダウンロードできます (http://www-3.ibm.com/software/data/informix/tools/jdbc/)。IBM Informix JDBC Driver は無償で提供されていますが、サポートの対象にはなりません。製品をダウンロードするには、IBM への登録が必要になることがあります。JDBC/EMBEDDED SQLJ セッションでドライバをダウンロードし、ダウンロードした zip ファイルに添付された install.txt ファイルの指示に従って、ドライバをインストールします。

ドライバをダウンロードしてインストールした後、以下の手順に従って、WebLogic Server でドライバを使用できるように準備します。

1. *INFORMIX_INSTALL*\lib から ifxjdbc.jar ファイルおよび ifxjdbcx.jar ファイルをコピーして、*WL_HOME*\server\lib フォルダに貼り付けます。ここで、

   INFORMIX_INSTALL は、Informix JDBC ドライバをインストールしたルートディレクトリです。

   また *WL_HOME* は、WebLogic Platform をインストールしたフォルダ (通常は c:\bea\weblogic700) です。

2. CLASSPATH に ifxjdbc.jar および ifxjdbcx.jar へのパスを追加します。次に例を示します。

   ```
   set
   CLASSPATH=%WL_HOME%\server\lib\ifxjdbc.jar;%WL_HOME%\server\lib
   \ifxjdbcx.jar;%CLASSPATH%
   ```

   また WebLogic Server の起動スクリプトで set CLASSPATH 文にドライバファイルへのパスを追加することもできます。

IBM Infomix JDBC Driver 使用時の接続プール属性

IBM Infomix JDBC Driver を使用する接続プールを作成するときは、表 5-1 および表 5-2 に示す属性を使用します。

表 5-1 Infomix JDBC Driver 使用時の XA 非対応接続プールの属性

属性	値
[URL]	`jdbc:informix-sqli:`*`dbserver_name_or_ip:port/dbname:`*`informixserver=`*`ifx_server_name`*
[ドライバ クラス名]	`com.informix.jdbc.IfxDriver`
[プロパティ]	`user=`*`username`* `url=jdbc:informix-sqli:`*`dbserver_name_or_ip:port/dbname:`*`informixserver=`*`ifx_server_name`* `portNumber=1543` `databaseName=`*`dbname`* `ifxIFXHOST=`*`ifx_server_name`* `serverName=`*`dbserver_name_or_ip`*
[パスワード]	*`password`*
[ログイン遅延時間]	`1`
[対象]	`serverName`

`config.xml` のエントリは、次のようになります。

```
<JDBCConnectionPool
    DriverName="com.informix.jdbc.IfxDriver"
    InitialCapacity="3"
    LoginDelaySeconds="1"
    MaxCapacity="10"
    Name="ifxPool"
    Password="xxxxxxx"
    Properties="informixserver=ifxserver;user=informix"
    Targets="examplesServer"
    URL="jdbc:informix-sqli:ifxserver:1543"
/>
```

表 5-2 Infomix JDBC Driver 使用時の XA 対応接続プールの属性

属性	値
[URL]	空白のまま
[ドライバ クラス名]	com.informix.jdbcx.IfxXADataSource
[プロパティ]	user=*username* url=jdbc:informix-sqli://*dbserver_name_or_ip*:*port_num*/*dbname*:informixserver=*dbserver_name_or_ip* password=*password* portNumber =*port_num*; databaseName=*dbname* serverName=*dbserver_name* ifxIFXHOST=*dbserver_name_or_ip*
[パスワード]	空白のまま
[ローカル トランザクションのサポート]	true
[対象]	*serverName*

注意 : [プロパティ] の文字列の portNumber と = の間にはスペースが入っています。

config.xml のエントリは、次のようになります。

```
<JDBCConnectionPool CapacityIncrement="2"
    DriverName="com.informix.jdbcx.IfxXADataSource"
    InitialCapacity="2" MaxCapacity="10"
    Name="informixXAPool"
    Properties="user=informix;url=jdbc:informix-sqli:
    //111.11.11.11:1543/db1:informixserver=lcsol15;
    password=informix;portNumber =1543;databaseName=db1;
    serverName=dbserver1;ifxIFXHOST=111.11.11.11"
    SupportsLocalTransaction="true" Targets="examplesServer"
    TestConnectionsOnReserve="true" TestTableName="emp"/>
```

注意： Administration Console を使用して接続プールを作成する場合、接続プールを対象のサーバに適切にデプロイするためには、その前にサーバを停止して再起動する必要があります。この問題は確認済みです。

IBM Infomix JDBC Driver のプログラミング上の注意

IBM Informix JDBC Driver を使用するときは、次の制限について注意が必要です。

- ドライバに、statement/resultset の処理が終了したことを指示するために、必ず `resultset.close()` および `statement.close()` の各メソッドを呼び出します。呼び出さない場合、プログラムではデータベース サーバのリソースがすべて解放されない場合があります。
- `IFX_USEPUT` 環境変数が `1` に設定されていない限り、TEXT または BYTE カラムのある行を挿入しようとすると、バッチ更新は失敗します。
- JDK のバージョンが 1.4 以降の場合、トランザクション中に Java プログラムで自動コミット モードが true にセットされると、IBM Informix JDBC Driver は、現在のトランザクションをコミットします。true でない場合は、自動コミットを実行する前に現在のトランザクションをロールバックします。

Microsoft SQL Server 2000 Driver for JDBC のインストールと使い方

Microsoft SQL Server 2000 Driver for JDBC は、ライセンスを持つ SQL Server 2000 ユーザならば無料でダウンロードできます。このドライバは、JDBC 2.0 オプション パッケージのサブセットをサポートする Type 4 JDBC ドライバです。Microsoft SQL Server 2000 Driver for JDBC のインストール時に、サポート ドキュメントをオプションでインストールできます。ドライバに関する包括的な情報については、そのドキュメントを参照する必要があります。また、確認済みの問題については、リリース マニフェストを参照してください。

Microsoft SQL Server Driver for JDBC の Windows システムへのインストール

Microsoft SQL Server 2000 Driver for JDBC を Windows サーバにインストールするには、次の手順に従います。

1. Microsoft MSDN Web サイトから Microsoft SQL Server 2000 Driver for JDBC (`setup.exe` ファイル) をダウンロードします。そのファイルをローカル コンピュータ上の一時ディレクトリに保存します。
2. 一時ディレクトリから `setup.exe` を実行し、画面の指示に従います。
3. `CLASSPATH` に以下のファイルへのパスを追加します。
   - *install_dir*/lib/msbase.jar
   - *install_dir*/lib/msutil.jar
   - *install_dir*/lib/mssqlserver.jar

   ここで、*install_dir* はドライバをインストールしたフォルダです。次に例を示します。

   ```
   set CLASSPATH=install_dir\lib\msbase.jar;
   install_dir\lib\msutil.jar;install_dir\lib\mssqlserver.jar;
   %CLASSPATH%
   ```

Microsoft SQL Server Driver for JDBC の UNIX システムへのインストール

Microsoft SQL Server 2000 Driver for JDBC を UNIX サーバにインストールするには、次の手順に従います。

1. Microsoft MSDN Web サイトから Microsoft SQL Server 2000 Driver for JDBC (`mssqlserver.tar` ファイル) をダウンロードします。そのファイルをローカル コンピュータ上の一時ディレクトリに保存します。
2. 一時ディレクトリに移動し、次のコマンドを使用してファイルの内容を復元します。

   ```
   tar -xvf mssqlserver.tar
   ```

3. 次のコマンドを実行して、インストール スクリプトを実行します。

   ```
   install.ksh
   ```

4. 画面の指示に従います。インストール ディレクトリの入力を要求された場合は、必ずそのディレクトリの絶対パスを入力してください。

5. CLASSPATH に以下のファイルへのパスを追加します。

   - *install_dir*/lib/msbase.jar
   - *install_dir*/lib/msutil.jar
   - *install_dir*/lib/mssqlserver.jar

   ここで、*install_dir* はドライバをインストールしたフォルダです。次に例を示します。

```
export CLASSPATH=install_dir/lib/msbase.jar:
install_dir/lib/msutil.jar:install_dir/lib/mssqlserver.jar:
$CLASSPATH
```

Microsoft SQL Server Driver for JDBC 使用時の接続プール属性

Microsoft SQL Server Driver for JDBC を使用する接続プールを作成するときは、次の属性を使用します。

- ドライバ名：com.microsoft.jdbc.sqlserver.SQLServerDriver
- URL： jdbc:microsoft:sqlserver://server_name:1433
- プロパティ：

  user=<myuserid>

  databaseName=<dbname>

  selectMethod=cursor

- パスワード：*mypassword*

config.xml のエントリは、次のようになります。

```
<JDBCConnectionPool
    Name="mssqlDriverTestPool"
    DriverName="com.microsoft.jdbc.sqlserver.SQLServerDriver"
    URL="jdbc:microsoft:sqlserver://lcdbnt4:1433"
    Properties="databasename=lcdbnt4;user=sa;
    selectMethod=cursor"
    Password="{3DES}vlsUYhxlJ/I="
    InitialCapacity="4"
    CapacityIncrement="2"
```

```
        MaxCapacity="10"
        Targets="examplesServer"
    />
```

注意 : 接続をトランザクション モードで使用するには、接続プロパティのリストに selectMethod=cursor を追加する必要があります。このように設定することで、アプリケーションで特定の接続から同時に複数の文を開くことが可能となります。これは、プールされた接続において必要となります。

selectMethod=cursor を設定しない場合は、同時に開いた文ごとに、接続の内部的なクローンが別々の DBMS ユーザとして作成されます。この場合、トランザクションを同時にコミットできなくなるため、デッドロックが発生するおそれがあります。

サードパーティ ドライバを使用した接続の取得

Oracle Thin Driver や Sybase jConnect Driver などのサードパーティ Type 4 ドライバを使用してデータベース接続を取得する方法について以下の節で説明します。接続を確立するには、接続プール、データ ソース、および JNDI ルックアップを使用することをお勧めします。

サードパーティ ドライバでの接続プールの使い方

まず、Administration Console を使用して接続プールとデータ ソースを作成し、次に JNDI ルックアップを使用して接続を確立します。

接続プールと DataSource の作成

JDBC 接続プールと JDBC DataSource の作成手順については、2-2 ページの「接続プールのコンフィグレーションと使い方」および 2-32 ページの「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。

JNDI を使用した接続の取得

JNDI を使用してサードパーティドライバにアクセスするには、まずサーバの URL を指定して JNDI ツリーから Context オブジェクトを取得し、次にそのコンテキスト オブジェクトと DataSource 名を使用してルックアップを実行します。

たとえば、Administration Console で定義された「`myDataSource`」という DataSource にアクセスするには、以下のようにします。

```
Context ctx = null;
  Hashtable ht = new Hashtable();
  ht.put(Context.INITIAL_CONTEXT_FACTORY,
         "weblogic.jndi.WLInitialContextFactory");
  ht.put(Context.PROVIDER_URL,
         "t3://hostname:port");

  try {
    ctx = new InitialContext(ht);
    javax.sql.DataSource ds
      = (javax.sql.DataSource) ctx.lookup ("myDataSource");
   java.sql.Connection conn = ds.getConnection();

   // これで conn オブジェクトを使用して
   // Statement オブジェクトを作成して
// SQL 文を実行し、結果セットを処理できる

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

   // 完了したら、文オブジェクトと
   // 接続オブジェクトを忘れずにクローズすること

   stmt.close();
   conn.close();
 }
  catch (NamingException e) {
// エラー発生
  }
  finally {
    try {ctx.close();}
  catch (Exception e) {
// エラー発生
    }
  }
```

(*`hostname`* は WebLogic Server が稼働するマシンのホスト名、`port` は WebLogic Server がリクエストをリスンするポートの番号です。)

この例では、*Hashtable* オブジェクトを使って、JNDI ルックアップに必要なパラメータを渡しています。JNDI ルックアップを実行する方法は他にもあります。詳細については、『WebLogic JNDI プログラマーズ ガイド』を参照してください。

ルックアップの失敗を捕捉するために JNDI ルックアップが `try/catch` ブロックで包まれている点に注意してください。また、コンテキストが `finally` ブロックの中でクローズされている点にも注意してください。

接続プールからの物理的な接続の取得

接続プールから接続を取得すると、WebLogic Server によって物理的な接続ではなく論理的な接続が提供されます。これは、接続を接続プールを使用して管理できるようにするためです。これにより、接続プール機能を有効にし、アプリケーションに提供する接続の品質を維持することが可能となります。しかし、物理的な接続を使用したい場合もあります。たとえば、DBMS ベンダの接続クラスを必要とするベンダ固有のメソッドに接続を渡す必要がある場合などです。WebLogic Server では、`weblogic.jdbc.extensions.WLConnection` インタフェースに `getVendorConnection()` メソッドが含まれており、論理的な接続から、その基底となる物理的な接続を取得することができます。詳細については、WebLogic Javadoc を参照してください。

注意 : 物理的な接続は、接続プールから取得した論理的な接続の代りとして使用しないようにしてください。詳細については、5-19 ページの「物理的な接続を使用する際の制限事項」を参照してください。

物理的なデータベース接続は、ベンダ固有の必要性がある場合にのみ使用します。ほとんどの JDBC 呼び出しは、論理的な接続に対してコーディングしておく必要があります。

接続の使用が完了したら、論理的な接続をクローズします。物理的な接続をコード内でクローズしないようにしてください。

物理的なデータベース接続をアプリケーション コードにエクスポーズした場合、その接続に対してアクセス権限を持たないユーザが接続にアクセスできてしまうおそれがあります。そのため WebLogic Server では、論理的な接続がクローズされると、これを接続プールに戻します。その基底となる物理的な接続は破棄さ

れ、プール内の論理的な接続に対する物理的な接続が新たに開かれます。この方式は安全である反面、処理には時間がかかります。接続プールへの要求が発生するごとに、新しいデータベース接続が作成されるようにすることは可能です。

物理的な接続を取得するサンプル コード

物理的なデータベース接続を取得するには、最初に 5-15 ページの「JNDI を使用した接続の取得」の説明に従って接続プールから接続を取得してから、次のいずれかを行います。

- 接続を WLConnection としてキャストし、`getVendorConnection()` を呼び出す
- 物理的な接続を必要とするメソッド内で、物理的な接続を暗黙的に渡す (`getVendorConnection()` メソッドを使用)

次に例を示します。

```
// このクラスと、必要となるすべてのベンダ パッケージをイン
// ポートする
import weblogic.jdbc.extensions.WLConnection
.
.
.
myJdbcMethod()
{

  // 接続プールからの接続は、クラス メソッドやインスタンス メソッド
  // ではなく、常にメソッド レベルの変数とする
  Connection conn = null;

   try {
     ctx = new InitialContext(ht);
     // JNDI ツリー上でデータ ソースをルックアップし、
     // 接続を要求する
     javax.sql.DataSource ds
        = (javax.sql.DataSource) ctx.lookup ("myDataSource");

     // プールされた接続は常に try ブロックで取得する。取得した
     // 接続は完全に使用し、必要に応じて finally ブロックで
     // クローズする
   conn  = ds.getConnection();

     // これで conn オブジェクトの WLConnection インタフェース
     // へのキャストが可能となり、基底となる物理的な接続を取得できる
```

```
      java.sql.Connection vendorConn =
        ((WLConnection)conn).getVendorConnection();
      // vendorConn はクローズしない

      // vendorConn オブジェクトをベンダ インタフェースにキャストするこ
      // とも可能。次に例を示す
     // oracle.jdbc.OracleConnection vendorConn = (OracleConnection)
      // ((WLConnection)conn).getVendorConnection()

      // ベンダ固有のメソッドで物理的な接続が必要に
      // なる場合、物理的な接続を取得または保持するのではなく、
      // 必要に応じて暗黙的に渡すほうがよい
      // 次に例を示す

//vendor.special.methodNeedingConnection(((WLConnection)conn)).ge
tVendorConnection());

      // ベンダ固有の呼び出しの使用が完了したら、
      // 接続への参照を即座に破棄する
      // 参照は保持したりクローズしたりしない
      // 汎用 JDBC にはベンダ接続を使用しない
      // 標準の JDBC には、論理的な ( プールされた ) 接続を使用する
      vendorConn = null;

      ... do all the JDBC needed for the whole method...

      // 論理的な ( プールされた ) 接続をクローズして
      // 接続プールに戻し、参照を破棄する
    conn.close();
       conn = null;
   }

   catch (Exception e)
   {
     // 例外を処理する
   }
   finally
   {
     // 念のため、論理的な ( プールされた ) 接続がクローズされているか
     // どうか確認する
      // finally ブロックの冒頭では、必ず論理的な ( プールされた ) 接続
     // をクローズする

    if (conn != null) try {conn.close();} catch (Exception ignore){}
   }
}
```

物理的な接続を使用する際の制限事項

物理的な接続は、接続プールから取得した論理的な接続の代りとして使用しないようにしてください。ただし、STRUCT を作成するためなど、どうしても物理的な接続を使用しなければならない場合は、以下のデメリットと制限事項を考慮に入れてください。

- 物理的な接続は、サーバサイド コードでしか使用できない。
- 物理的な接続を使用すると、WebLogic Server が提供するすべての接続管理機能 (エラー処理、文のキャッシングなど) が使用できなくなる。
- 物理的な接続は、それを必要とするベンダ固有のメソッドまたはクラスにのみ使用する。汎用 JDBC (文やトランザクション呼び出しの作成など) には物理的な接続を使用しないでください。
- 接続が再利用されない。接続をクローズすると、物理的な接続がクローズされます。物理的な接続として渡された接続は、接続プールが作成する新しい接続によって置換されます。物理的な接続を使用すると接続が再利用されないため、以下の理由でパフォーマンスが低下します。
  - 物理的な接続は接続プール内の新しいデータベース接続によって置換されるが、この処理にはアプリケーション サーバとデータベース サーバ両方のリソースが必要となる
  - 元の接続の Statement キャッシュ がクローズされ、新しい接続用に新しいキャッシュが開かれるため、Statement キャッシュを使用することによるパフォーマンスの向上が無効になる

Oracle 拡張機能と Oracle Thin Driver の使用

Oracle 拡張機能では、Oracle データベースのデータを操作するための独自の方法が追加されます。これにより、標準 JDBC インタフェースの機能が拡張されます。BEA では、Oracle Thin Driver や Oracle 拡張機能をサポートする他のドライバに対応できるように、拡張機能をサポートしています。

- `OracleStatement`

- `OracleResultSet`
- `OraclePreparedStatement`
- `OracleCallableStatement`
- `OracleArray`
- `OracleStruct`
- `OracleRef`
- `OracleBlob`
- `OracleClob`

以降の節では、Oracle 拡張機能のサンプル コードと、サポートされるメソッドの表を示します。詳細については、Oracle のマニュアルを参照してください。

Oracle JDBC 拡張機能の使用時の制限

JDBC インタフェースに Oracle 拡張機能を使用するときは、次の制限があります。

- Oracle 拡張機能は、サーバと同じ JVM のみを使用したサーバサイド アプリケーションで、ARRAY、REF、および STRUCT の各オブジェクトに適用できます。クライアント アプリケーションの ARRAY、REF、STRUCT に Oracle 拡張機能は使用できません。
- アプリケーションで ARRAY、REF、STRUCT は作成できません。既存の ARRAY、REF、STRUCT オブジェクトは、データベースからのみ検索できます。アプリケーションでこれらのオブジェクトを作成するには、標準外の Oracle 記述子オブジェクトを使用します。ただし、WebLogic Server ではサポートされていません。

Oracle 拡張機能から JDBC インタフェースにアクセスするサンプル コード

以下のコード例は、WebLogic Oracle 拡張機能から標準 JDBC インタフェースにアクセスする方法を示しています。最初の例では、`OracleConnection` および `OracleStatement` 拡張機能を使用します。この例の構文は、WebLogic Server でサポートされるメソッドを使用する場合、`OracleResultSet`、`OraclePreparedStatement`、および `OracleCallableStatement` インタフェースで使用できます。サポートされるメソッドについては、5-39 ページの「Oracle 拡張機能インタフェースとサポートされるメソッドの表」を参照してください。

その他の Oracle 拡張機能メソッドを使用した例については、以下の節を参照してください。

- 5-23 ページの「ARRAY によるプログラミング」
- 5-25 ページの「STRUCT によるプログラミング」
- 5-30 ページの「REF によるプログラミング」
- 5-36 ページの「BLOB と CLOB によるプログラミング」

WebLogic Server を使用して、サーバの例をインストールするオプションを選択した場合は、JDBC 例を参照してください。この例は通常、*WL_HOME*`\samples\server\src\examples\jdbc` (*WL_HOME* は WebLogic Platform をインストールしたフォルダ) にあります。

Oracle 拡張機能へアクセスするパッケージをインポートする

この例で使用する Oracle インタフェースをインポートします。OracleConnection および OracleStatement インタフェースは、`oracle.jdbc.OracleConnection` および `oracle.jdbc.OracleStatement` に相当し、WebLogic Server でサポートされるメソッドを使用する場合は、これらの Oracle インタフェースと同様に使用できます。

```
import java.sql.*;
import java.util.*;
import javax.naming.Context;
import javax.naming.InitialContext;
```

```
import javax.sql.DataSource;
import weblogic.jdbc.vendor.oracle.*;
```

接続を確立する

JNDI、DataSource、および接続プール オブジェクトを使用して、データベース接続を確立します。詳細については、5-15 ページの「JNDI を使用した接続の取得」を参照してください。

```
// 接続プールの有効な DataSource オブジェクトを取得する
// ここでは、その詳細を getDataSource() が
// 処理すると仮定する
javax.sql.DataSource ds = getDataSource(args);

// DataSource から java.sql.Connection オブジェクトを取得する
   java.sql.Connection conn = ds.getConnection();
```

デフォルトの行プリフェッチ値を取得する

次のコードでは、Oracle Thin Driver で使用できる Oracle の行プリフェッチ メソッドの使い方を示します。

```
// OracleConnection にキャストして、この接続の
// デフォルトの行プリフェッチ値を取得する

int default_prefetch =
          ((OracleConnection)conn).getDefaultRowPrefetch();

System.out.println("Default row prefetch
    is " + default_prefetch);

java.sql.Statement stmt = conn.createStatement();

// OracleStatement にキャストして、この文の
// 行プリフェッチ値を設定する
// このプリフェッチ値は、WebLogic Server とデータベースの
      // 間の接続に適用されることに注意
      ((OracleStatement)stmt).setRowPrefetch(20);

      // 通常の SQL クエリを実行して、その結果を処理 ...
      String query = "select empno,ename from emp";
      java.sql.ResultSet rs = stmt.executeQuery(query);

      while(rs.next()) {
         java.math.BigDecimal empno = rs.getBigDecimal(1);
         String ename = rs.getString(2);
         System.out.println(empno + "\t" + ename);
      }
```

```
        rs.close();
        stmt.close();

        conn.close();
        conn = null;
    }
```

ARRAY によるプログラミング

WebLogic Server のサーバサイド アプリケーションでは、結果セット、または Java 配列として callable statement を使用することにより、Oracle Collection (SQL ARRAY) を実体化できます。

WebLogic Server アプリケーションで ARRAY を使用するには、次の手順に従います。

1. 必要なクラスをインポートします (5-21 ページの「Oracle 拡張機能へアクセスするパッケージをインポートする」を参照)。
2. 接続を取得 (5-22 ページの「接続を確立する」を参照) して、接続のための文を作成します。
3. 結果セット、または callable statement を使用して ARRAY を取得します。
4. `java.sql.Array` または `weblogic.jdbc.vendor.oracle.OracleArray` のいずれかとして、ARRAY を使用します。
5. 標準 Java メソッド (`java.sql.Array` として使用)、または Oracle 拡張機能メソッド (`weblogic.jdbc.vendor.oracle.OracleArray` としてキャスト) を使用して、データを操作します。

以下の節では、これらのアクションの詳細について説明します。

注意： ARRAY はサーバサイド アプリケーションでのみ使用できます。クライアント アプリケーションでは ARRAY は使用できません。

ARRAY を取得する

callable statement、または結果セットに getArray() メソッドを使用して、Java 配列を取得できます。この配列は、java.sql.array として使用することにより標準 java.sql.array メソッドを利用することも、また weblogic.jdbc.vendor.oracle.OracleArray としてキャストすることにより、配列の Oracle 拡張機能メソッドとして利用することもできます。

以下の例では、ARRAY を含む結果セットから java.sql.array を取得する方法を示します。この例では、クエリにより、オブジェクト カラム (学生の成績を示す ARRAY) を含む結果セットが返されます。

```
try {

  conn = getConnection(url);
  stmt = conn.createStatement();
  String sql = "select * from students";
// 結果セットの取得
  rs = stmt.executeQuery(sql);

  while(rs.next()) {
    BigDecimal id = rs.getBigDecimal("student_id");
    String name   = rs.getString("name");
   log("ArraysDAO.getStudents() -- Id = "+id.toString()+", Student
= "+name);
// 結果セットからの配列の取得
    Array scoreArray  = rs.getArray("test_scores");
    String[] scores = (String[])scoreArray.getArray();
    for (int i = 0; i < scores.length; i++) {
      log("   Test"+(i+1)+" = "+scores[i]);
    }
  }
```

データベースで ARRAY を更新する

データベースにおける ARRAY を更新するには、次の手順に従います。

1. 更新する配列がデータベースにない場合、PL/SQL を使用してデータベースに配列を作成します。
2. 結果セット、または callable statement を使用して ARRAY を取得します。
3. Java アプリケーション内の配列を java.sql.Array または weblogic.jdbc.vendor.oracle.OracleArray として扱います。

4. prepared statement または callable statement に `setArray()` メソッドを使用して、データベース内の配列を更新します。次に例を示します。次に例を示します。

```
String sqlUpdate = "UPDATE SCOTT."+ tableName + " SET col1 = ?";
   conn  = ds.getConnection();
   pstmt = conn.prepareStatement(sqlUpdate);
   pstmt.setArray(1, array);
   pstmt.executeUpdate();
```

Oracle Array 拡張機能メソッドを使用する

ARRAY に Oracle 拡張機能メソッドを使用するにはまず、`weblogic.jdbc.vendor.oracle.OracleArray` として配列をキャストする必要があります。この後、ARRAY の Oracle 拡張機能メソッドを呼び出すことができます。次に例を示します。

```
oracle.sql.Datum[] oracleArray = null;
oracleArray =
((weblogic.jdbc.vendor.oracle.OracleArray)scoreArray).getOracleArray();
String sqltype = null
sqltype = oracleArray.getSQLTypeName()
```

STRUCT によるプログラミング

WebLogic Server アプリケーションでは、Oracle データベースからオブジェクトにアクセスしたり、オブジェクトを操作したりできます。Oracle データベースからオブジェクトを検索すると、カスタム Java オブジェクト、または STRUCT (`java.sql.struct` あるいは `weblogic.jdbc.vendor.oracle.OracleStruct`) のいずれかとして、オブジェクトをキャストできます。STRUCT は、アプリケーション中のカスタム クラスを置き換える構造化データを表す型制限の緩いデータ型です。JDBC API における STRUCT インタフェースには、STRUCT 中の属性値を操作するためのさまざまなメソッドが組み込まれています。Oracle では、いくつかの追加メソッドを使用して、STRUCT インタフェースを拡張しています。WebLogic Server では、すべての標準メソッドと大部分の Oracle 拡張機能が実装されています。

注意： STRUCT を使用する場合、次の制限があります。

- STRUCT は、Oracle 専用としてサポートされるデータ型です。アプリケーション中で STRUCT を使用するには、Oracle Thin Driver を使用して (通常は接続プールを介して) データベースとやり取りします。WebLogic jDriver for Oracle では、STRUCT データ型はサポートされません。
- STRUCT はサーバサイド アプリケーションでのみ使用できます。クライアント アプリケーションでは STRUCT は使用できません。

WebLogic Server アプリケーションで STRUCT を使用するには、次の手順に従います。

1. 必要なクラスをインポートします (5-21 ページの「Oracle 拡張機能へアクセスするパッケージをインポートする」を参照) 。
2. 接続を取得します (5-22 ページの「接続を確立する」を参照)。
3. `getObject` を使用して STRUCT を取得します。
4. STRUCT を `java.sql.Struct` または `weblogic.jdbc.vendor.oracle.OracleStruct` の STRUCT としてキャストします。
5. 標準メソッド、または Oracle 拡張機能メソッドを使用して、データを操作します。

以下の節では、手順 3 ～ 5 について詳しく説明します。

STRUCT を取得する

データベース オブジェクトを STRUCT として取得するには、まずクエリを使用して結果セットを作成し、次に `getObject` メソッドを使用して、結果セットから STRUCT を取得します。次に STRUCT を `java.sql.Struct` としてキャストすることにより、標準 Java メソッドを使用できるようになります。次に例を示します。

```
conn = ds.getConnection();
stmt = conn.createStatement();
rs   = stmt.executeQuery("select * from people");
struct = (java.sql.Struct)(rs.getObject(2));
Object[] attrs = ((java.sql.Struct)struct).getAttributes();
```

WebLogic Server では、STRUCT に対応するすべての JDBC API メソッドがサポートされます。

- `getAttributes()`
- `getAttributes(java.util.Dictionary map)`
- `getSQLTypeName()`

Oracle では、標準メソッド以外に、Oracle 拡張機能メソッドもサポートしています。したがって、STRUCT を `weblogic.jdbc.vendor.oracle.OracleStruct` としてキャストすると、標準メソッドと拡張機能メソッドの両方が使用できるようになります。

OracleStruct 拡張機能メソッドを使用する

STRUCT に Oracle 拡張機能メソッドを使用する場合、`java.sql.Struct` (またはオリジナルの `getObject` 結果) を `weblogic.jdbc.vendor.oracle.OracleStruct` としてキャストする必要があります。次に例を示します。

```
java.sql.Struct struct =
(weblogic.jdbc.vendor.oracle.OracleStruct)(rs.getObject(2));
```

WebLogic Server では次の Oracle 拡張機能がサポートされます。

- `getDescriptor()`
- `getOracleAttributes()`
- `getAutoBuffering()`
- `setAutoBuffering(boolean)`

STRUCT 属性を取得する

STRUCT で個々の属性に対する値を取得するには、`getAttributes()` および `getAttributes(java.util.Dictionary map)` の標準 JDBC API メソッド、または `getOracleAttributes()` の Oracle 拡張機能メソッドを使用できます。

標準メソッドを使用するには、まず結果セットを作成し、この結果セットから STRUCT を取得し、次に `getAttributes()` メソッドを使用します。このメソッドにより、順序の付いた属性の配列が返されます。アプリケーションのオブジェ

クト (Java 言語タイプなど) に STRUCT (データベースのオブジェクト) の属性を割り当てることができます。この後、属性を個別に操作できるようになります。次に例を示します。

```
conn = ds.getConnection();

stmt = conn.createStatement();

rs   = stmt.executeQuery("select * from people");

// 第 3 カラムはオブジェクト データ型を使用
// getObject() により、値の配列にオブジェクトを割り当てる

struct = (java.sql.Struct)(rs.getObject(2));

Object[] attrs = ((java.sql.Struct)struct).getAttributes();

String address = attrs[1];
```

上の例で、`people` テーブルの第 3 カラムではオブジェクト データ型を使用しています。この例は、値の配列を含む Java オブジェクト `getObject` メソッドの結果を割り当て、必要に応じて配列中の個別の値を使用する方法を示します。

また、`getAttributes(java.util.Dictionary map)` メソッドを使用しても、STRUCT から属性を取得できます。このメソッドを使用する場合は、Java 言語データ型に Oracle オブジェクトのデータ型をマッピングします。次に例を示します。

```
java.util.Hashtable map = new java.util.Hashtable();

map.put("NUMBER", Class.forName("java.lang.Integer"));

map.put("VARCHAR", Class.forName("java.lang.String"));

Object[] attrs = ((java.sql.Struct)struct).getAttributes(map);

String address = attrs[1];
```

また、Oracle 拡張機能メソッドの `getOracleAttributes()` を使用して、STRUCT の属性を取得することもできます。まず、STRUCT を `weblogic.jdbc.vendor.oracle.OracleStruct` としてキャストします。このメソッドにより、`oracle.sql.Datum` オブジェクトのデータ配列が返されます。次に例を示します。

```
oracle.sql.Datum[] attrs =
 ((weblogic.jdbc.vendor.oracle.OracleStruct)struct).getOracleAttributes();

      oracle.sql.STRUCT address = (oracle.sql.STRUCT) attrs[1];

      Object address_attrs[] = address.getAttributes();
```

上の例では、STRUCT がネスト構造になっています。つまり、ここで返されるデータ配列には、別の STRUCT が入れ子の構造で組み込まれています。

STRUCT によってデータベース オブジェクトを更新する

STRUCT を使用してデータベースのオブジェクトを更新するには、prepared statement にある `setObject` メソッドを使用します。次に例を示します。

```
conn = ds.getConnection();

stmt = conn.createStatement();

ps = conn.prepareStatement ("UPDATE SCHEMA.people SET EMPLNAME = ?,
 EMPID = ? where EMPID = 101");

ps.setString (1, "Smith");

ps.setObject (2, struct);

ps.executeUpdate();
```

WebLogic Server では、`setObject` メソッドの 3 つのバージョンがすべてサポートされます。

データベース オブジェクトを作成する

STRUCT は通常、Java アプリケーション中で、データベース オブジェクトにマッピングするカスタム Java クラスに代わるデータベース オブジェクトを実体化する場合に使用します。WebLogic Server アプリケーションでは、データベースに転送する STRUCT は作成できません。ただし、アプリケーション上から検索や操作が実行できるようなデータベース オブジェクトを作成する文は使用できます。次に例を示します。

```
conn = ds.getConnection();

stmt = conn.createStatement();

cmd = "create type ob as object (ob1 int, ob2 int)"

stmt.execute(cmd);

cmd = "create table t1 of type ob";

stmt.execute(cmd);

cmd = "insert into t1 values (5, 5)"

stmt.execute(cmd);
```

注意 : アプリケーションで STRUCT は作成できません。データベースから既存のオブジェクトを検索して、これを STRUCT としてキャストすることはできます。アプリケーションで STRUCT オブジェクトを作成するには、標準外の Oracle STRUCT 記述子オブジェクトを使用します。ただし、WebLogic Server ではサポートされません。

STRUCT 属性を自動バッファリングする

STRUCT を使用した WebLogic Server アプリケーションのパフォーマンスを改善するために、自動バッファリング機能と `setAutoBuffering(boolean)` メソッドを切り換えることができます。自動バッファリングを `true` に設定すると、`weblogic.jdbc.vendor.oracle.OracleStruct` により、STRUCT オブジェクトにあるすべての属性のローカル コピーが変換済みのフォーム (すなわち SQL から Java 言語オブジェクトに実体化した形式) で保持されます。アプリケーションが、STRUCT に再びアクセスした時点で、データを再度変換する必要はありません。

注意 : 変換した属性をバッファリングすると、アプリケーションで使用するメモリ量が過度に増大することがあります。自動バッファリングの有効 / 無効を切り換えるときは、可能メモリ使用量についても考慮してください。

以下の例は、自動バッファリングをアクティブにする方法を示します。

```
((weblogic.jdbc.vendor.oracle.OracleStruct)struct).setAutoBuffering(true);
```

また、`getAutoBuffering()` メソッドを使用して、自動バッファリング モードを設定することもできます。

REF によるプログラミング

REF は、行オブジェクトに対する論理ポインタです。REF を検索すると、実際には別のテーブルにある値を指すポインタが返されます。REF のターゲットは、オブジェクト テーブルの行でなければなりません。REF を使用して、これが参照するオブジェクトを検証したり、更新したりできます。また REF を変更することにより、同じオブジェクト タイプの別のオブジェクトを指示したり、null 値を割り当てたりすることができます。

注意： REF を使用する場合、次の制限があります。

- REF は、Oracle 専用としてサポートされるデータ型です。アプリケーションで REF を使用するには、Oracle Thin Driver を使用して (通常は接続プールを介して) データベースとやり取りします。WebLogic jDriver for Oracle では、REF データ型はサポートされません。
- REF はサーバサイド アプリケーションでのみ使用できます。

WebLogic Server アプリケーションで REF を使用するには、次の手順に従います。

1. 必要なクラスをインポートします (5-21 ページの「Oracle 拡張機能へアクセスするパッケージをインポートする」を参照) 。
2. データベース接続を取得します (5-22 ページの「接続を確立する」を参照)。
3. 結果セット、または callable statement を使用して REF を取得します。
4. 結果を STRUCT として、または Java オブジェクトとしてキャストします。これにより、STRUCT メソッド、または Java オブジェクトのメソッドを使用して、データを操作できるようになります。

また、データベースで REF を作成したり、更新したりできます。

手順 3 と 4 について以下の節で詳しく説明します。

REF を取得する

アプリケーションで REF を取得するには、まずクエリを使用して結果セットを作成し、次に `getRef` メソッドを使用して、結果セットから REF を取得します。次に REF を `java.sql.Ref` としてキャストすることにより、ビルトイン Java メソッドを使用できます。次に例を示します。

```
conn = ds.getConnection();
stmt = conn.createStatement();
rs   = stmt.executeQuery("SELECT ref (s) FROM t1 s where s.ob1=5");
rs.next();
// java.sql.Ref としてキャストし、REF を取得
```

```
ref = (java.sql.Ref) rs.getRef(1);
```

なお、上の例の WHERE 句では、ドット表記法を使用して、参照するオブジェクトの属性を指定しています。

REF を java.sql.Ref としてキャストした後は、getBaseTypeName の Java API メソッドが使用できます (これは REF に対応した唯一の JDBC 2.0 標準メソッド)。

REF を取得すると、実際には別のオブジェクト テーブルにある値を指すポインタが返されます。REF 値の取得や操作を行うには、Oracle 拡張機能を使用します。この拡張機能は、sql.java.Ref を weblogic.jdbc.vendor.oracle.OracleRef としてキャストする場合に限って使用できます。

OracleRef 拡張機能メソッドを使用する

REF に Oracle 拡張機能メソッドを使用するには、REF を Oracle REF としてキャストします。たとえば、次のとおりです。

```
oracle.sql.StructDescriptor desc =
((weblogic.jdbc.vendor.oracle.OracleRef)ref).getDescriptor();
```

WebLogic Server では次の Oracle 拡張機能がサポートされます。

- getDescriptor()
- getSTRUCT()
- getValue()
- getValue(dictionary)
- setValue(object)

値を取得する

Oracle では、2 つのバージョンの getValue() メソッドが提供されています。パラメータの指定が不要なメソッドと、戻り値の型をマッピングするハッシュテーブルを要求するメソッドの 2 種類です。いずれかの getValue() メソッドを使用して、REF の属性値を取得すると、STRUCT または Java オブジェクトのいずれかの形式で結果が返されます。

パラメータなしの getValue() メソッドを使用する方法を以下の例で示します。この例では、REF を oracle.sql.STRUCT としてキャストします。getAttributes() メソッドの説明で示したとおり、STRUCT メソッドを使用して、値を操作できます。

```
oracle.sql.STRUCT student1 =
(oracle.sql.STRUCT)((weblogic.jdbc.vendor.oracle.OracleRef)ref).getValue ();

Object attributes[] = student1.getAttributes();
```

また、getValue(dictionary) を使用して、REF に対する値を取得できます。また REF の属性ごとにデータ型を Java 言語データ型にマッピングするためのハッシュ テーブルが必要になります。次に例を示します。

```
java.util.Hashtable map = new java.util.Hashtable();

map.put("VARCHAR", Class.forName("java.lang.String"));

map.put("NUMBER", Class.forName("java.lang.Integer"));

oracle.sql.STRUCT result = (oracle.sql.STRUCT)
 ((weblogic.jdbc.vendor.oracle.OracleRef)ref).getValue (map);
```

REF 値を更新する

REF を更新する場合、次のいずれかの操作を実行します。

- setValue(object) メソッドを使用して、基盤となるテーブルの値を変更する
- prepared statement または callable statement を使用して REF が指示する位置を変更する
- REF の値を null に設定する

setValue(object) を使用して REF 値を更新する場合は、まず REF の新しい値を使用してオブジェクトを作成した後、setValue メソッドのパラメータとしてオブジェクトを渡します。次に例を示します。

```
STUDENT s1 = new STUDENT();

s1.setName("Terry Green");

s1.setAge(20);

((weblogic.jdbc.vendor.oracle.OracleRef)ref).setValue(s1);
```

REF の値を setValue(object) メソッドで更新すると、実際には REF が指示するテーブルの値が更新されます。

prepared statement を使用して REF が指示する位置を更新するには、次の 3 つの基本手順に従います。

1. 新しい位置を指示する REF を取得します。この REF を使用して、別の REF の値を置き換えます。
2. SQL コマンドの文字列を作成して、既存の REF の位置を、別の REF の値で置き換えます。
3. prepared statement を作成、および実行します。

次に例を示します。

```
try {
conn = ds.getConnection();
stmt = conn.createStatement();
// REF の取得
rs   = stmt.executeQuery("SELECT ref (s) FROM t1 s where s.ob1=5");
rs.next();
ref = (java.sql.Ref) rs.getRef(1); // REF を java.sql.Ref としてキャストする
}
// prepared statement の作成と実行
String sqlUpdate = "update t3 s2 set col = ? where s2.col.ob1=20";
pstmt = conn.prepareStatement(sqlUpdate);
pstmt.setRef(1, ref);
pstmt.executeUpdate();
```

callable statement を使用して、REF が指示する位置を更新する場合は、まずストアド プロシージャを作成し、いずれかの IN パラメータを設定して、OUT パラメータを登録した後、文を実行します。ストアド プロシージャでは、実際の位置を指示する REF 値が更新されます。次に例を示します。

```
conn = ds.getConnection();
stmt = conn.createStatement();
rs   = stmt.executeQuery("SELECT ref (s) FROM t1 s where s.ob1=5");
```

```
rs.next();
ref1 = (java.sql.Ref) rs.getRef(1);
// ストアド プロシージャの作成
sql = "{call SP1 (?, ?)}";
cstmt = conn.prepareCall(sql);
// IN パラメータと OUT パラメータの登録
cstmt.setRef(1, ref1);
cstmt.registerOutParameter(2, getRefType(), "USER.OB");
// 実行
cstmt.execute();
```

データベースで REF を作成する

JDBC アプリケーションで REF オブジェクトは作成できません。単に、データベースから既存の REF オブジェクトが検索されるだけです。ただし、文または prepared statement を使用して、データベースに REF を作成できます。次に例を示します。

```
conn = ds.getConnection();
stmt = conn.createStatement();
cmd = "create type ob as object (ob1 int, ob2 int)"
stmt.execute(cmd);
cmd = "create table t1 of type ob";
stmt.execute(cmd);
cmd = "insert into t1 values (5, 5)"
stmt.execute(cmd);
cmd = "create table t2 (col ref ob)";
stmt.execute(cmd);
cmd = "insert into t2 select ref(p) from t1 where p.ob1=5";
stmt.execute(cmd);
```

上の例では、オブジェクト タイプ (ob)、このオブジェクト タイプのテーブル (t1)、ob オブジェクトのインスタンスを指示する REF カラムを含むテーブル (t2) を作成して、REF を REF カラムに挿入します。REF は、t1 の行を指します (最初のカラムの値は 5)。

BLOB と CLOB によるプログラミング

この節では、OracleBlob インタフェースへのアクセス方法を示すサンプル コードについて説明します。WebLogic Server でサポートされるメソッドを使用している場合、この例で示す構文は、OracleBlob インタフェースで使用できます。5-39 ページの「Oracle 拡張機能インタフェースとサポートされるメソッドの表」を参照してください。

注意： BLOB および CLOB (「LOB」と呼ばれる) を使用する場合、トランザクションの境界を考慮する必要があります。たとえば、すべての読み取り / 書き込みをトランザクション内の特定の LOB に転送します。詳細については、Oracle Web サイトにある Oracle のマニュアルの「LOB Locators and Transaction Boundaries」を参照してください。

DBMS から BLOB ロケータを選択するクエリを実行する

BLOB ロケータまたはハンドルは、Oracle Thin Driver Blob への参照です。

```
String selectBlob = "select blobCol from myTable where blobKey =
666"
```

WebLogic Server java.sql オブジェクトを宣言する

次のコードでは、Connection が既に確立されていることを前提としています。

```
ResultSet rs = null;
Statement myStatement = null;
java.sql.Blob myRegularBlob = null;
java.io.OutputStream os = null;
```

SQL 例外ブロックを開始する

この try/catch ブロックでは、BLOB ロケータを取得して、Oracle Blob 拡張機能にアクセスします。

```
try {

     // BLOB ロケータを取得 ..
```

```
      myStatement = myConnect.createStatement();
      rs = myStatement.executeQuery(selectBlob);
      while (rs.next()) {
        myRegularBlob = rs.getBlob("blobCol");

}

      // 記述用の基底の Oracle 拡張機能にアクセスする
      // OracleThinBlob インタフェースをキャストして、
      // Oracle メソッドにアクセスする

      os = ((OracleThinBlob)myRegularBlob).getBinaryOutputStream();
      .....
      ........

      } catch (SQLException sqe) {
        System.out.println("ERROR(general SQE): " +
          sqe.getMessage());
      }
```

Oracle.ThinBlob インタフェースをキャストしたら、BEA がサポートするメソッドにアクセスできます。

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。

Oracle 仮想プライベート データベースによるプログラミング

WebLogic Server 7.0 SP3 では、Oracle 仮想プライベート データベース (Virtual Private Database : VPD) がサポートされています。VPD を使用することで、アプリケーション定義のファイングレイン アクセス コントロールをサーバで実施し、Oracle 9i データベース サーバ内のアプリケーション コンテキストのセキュリティを確保できます。

WebLogic Server アプリケーションで VPD を使用するには、以下の手順に従います。

1. Oracle Thin ドライバまたは Oracle OCI ドライバを使用する WebLogic Server コンフィグレーション内に JDBC 接続プールを作成します。詳細については、2-1 ページの「WebLogic JDBC のコンフィグレーションと管理」、または『管理者ガイド』の「Administration Console を使用した JDBC 接続のコンフィグレーション」を参照してください。

   注意 : XA 対応の JDBC ドライバを使用している場合は、`KeepXAConnTillTxComplete=true` を設定する必要があります。詳細については、『管理者ガイド』の「追加の XA 接続プール プロパティ」を参照してください。

   WebLogic jDriver for Oracle は `ClientIdentifier` を伝播できないため、VPD で使用するドライバとしては適していません。

2. 接続プールを指す WebLogic Server コンフィグレーション内にデータ ソースを作成します。

3. アプリケーションで次のコードを実行します。

```
import weblogic.jdbc.vendor.oracle.OracleConnection;

// WLS JDBC 接続プールから接続を取得する
Connection conn = ds.getConnection();

// Oracle 拡張機能にキャストして CLIENT_IDENTIFIER を設定する
// ( データベース サーバ サイドの USERENV ネーミング コンテキスト
//   からアクセス可能になる )
((OracleConnection)conn).setClientIdentifier(clientId);

/* アプリケーション固有の処理を実行する */

// WLS JDBC 接続プールに戻る前に接続をクリーンアップする
((OracleConnection)conn).clearClientIdentifier(clientId);

// 接続をクローズする
   conn.close();
```

Oracle 拡張機能インタフェースとサポートされるメソッドの表

Oracle インタフェースを以下の表に示します。また、標準 JDBC (`java.sql.*`) インタフェースを拡張するために Oracle Thin Driver (またはこれらのメソッドをサポートするその他のドライバ) で使用するメソッドでサポートされているものも示します。

注意： 通常、Oracle Thin Driver の新しいバージョンがリリースされるときには、一部の拡張機能メソッドがドライバから削除されます。WebLogic Server では、ドライバに含まれなくなったメソッドをサポートできません。Oracle Thin Driver 9.2.0 (および WebLogic Server 7.0 サービス パック 2) では、以下のメソッドが削除されています。

- `OracleStatement.getAutoRollback()`
- `OracleStatement.getWaitOption()`
- `OracleConnection.isCompatibleTo816()`

表 5-3 OracleConnection インタフェース

拡張	メソッド シグネチャ
OracleConnection 拡張 java.sql.Connection	boolean **getAutoClose()** throws java.sql.SQLException;
	void **setAutoClose(boolean on)** throws java.sql.SQLException;
	String **getDatabaseProductVersion()** throws java.sql.SQLException;
	String **getProtocolType()** throws java.sql.SQLException;
	String **getURL()** throws java.sql.SQLException;
	String **getUserName(**) throws java.sql.SQLException;
	boolean **getBigEndian()** throws java.sql.SQLException;
	boolean **getDefaultAutoRefetch()** throws java.sql.SQLException;
	boolean **getIncludeSynonyms()** throws java.sql.SQLException;
	boolean **getRemarksReporting()** throws java.sql.SQLException;
	boolean **getReportRemarks()** throws java.sql.SQLException;
	boolean **getRestrictGetTables()** throws java.sql.SQLException;
	boolean **getUsingXAFlag()** throws java.sql.SQLException;
	boolean **getXAErrorFlag()** throws java.sql.SQLException;

表 5-3 OracleConnection インタフェース

拡張	メソッド シグネチャ
OracleConnection 拡張 java.sql.Connection (続き)	byte[] **getFDO(boolean b)** throws java.sql.SQLException;
	int **getDefaultExecuteBatch()** throws java.sql.SQLException;
	int **getDefaultRowPrefetch()** throws java.sql.SQLException;
	int **getStmtCacheSize()** throws java.sql.SQLException;
	java.util.Properties **getDBAccessProperties()** throws java.sql.SQLException;
	short **getDbCsId()** throws java.sql.SQLException;
	short **getJdbcCsId()** throws java.sql.SQLException;
	short **getStructAttrCsId()** throws java.sql.SQLException;
	short **getVersionNumber()** throws java.sql.SQLException;
	void **archive(int i, int j, String s)** throws java.sql.SQLException;
	void **close_statements()** throws java.sql.SQLException;
	void **initUserName()** throws java.sql.SQLException;
	void **logicalClose()** throws java.sql.SQLException;
	void **needLine()** throws java.sql.SQLException;
	void **printState()** throws java.sql.SQLException;
	void **registerSQLType(String s, String t)** throws java.sql.SQLException;
	void **releaseLine()** throws java.sql.SQLException;

表 5-3 OracleConnection インタフェース

拡張	メソッド シグネチャ

OracleConnection 拡張 java.sql.Connection (続き)	void **removeAllDescriptor()** throws java.sql.SQLException;
	void **removeDescriptor(String s)** throws java.sql.SQLException;
	void **setDefaultAutoRefetch(boolean b)** throws java.sql.SQLException;
	void **setDefaultExecuteBatch(int i)** throws java.sql.SQLException;
	void **setDefaultRowPrefetch(int i)** throws java.sql.SQLException;
	void **setFDO(byte[] b)** throws java.sql.SQLException;
	void **setIncludeSynonyms(boolean b)** throws java.sql.SQLException;
	void **setPhysicalStatus(boolean b)** throws java.sql.SQLException;
	void **setRemarksReporting(boolean b)** throws java.sql.SQLException;
	void **setRestrictGetTables(boolean b)** throws java.sql.SQLException;
	void **setStmtCacheSize(int i)** throws java.sql.SQLException;
	void **setStmtCacheSize(int i, boolean b)** throws java.sql.SQLException;
	void **setUsingXAFlag(boolean b)** throws java.sql.SQLException;
	void **setXAErrorFlag(boolean b)** throws java.sql.SQLException;
	void **shutdown(int i)** throws java.sql.SQLException;
	void **startup(String s, int i)** throws java.sql.SQLException;

表 5-4 OracleStatement インタフェース

拡張	メソッド シグネチャ
OracleStatement 拡張 java.sql.statement	String **getOriginalSql()** throws java.sql.SQLException;
	String **getRevisedSql()** throws java.sql.SQLException; (Oracle 8.1.7 で非推奨、Oracle 9i で破棄)
	boolean **getAutoRefetch()** throws java.sql.SQLException;
	boolean **is_value_null(boolean b, int i)** throws java.sql.SQLException;
	byte ge**tSqlKind()** throws java.sql.SQLException;
	int **creationState()** throws java.sql.SQLException;
	int **getRowPrefetch()** throws java.sql.SQLException;
	int **sendBatch()** throws java.sql.SQLException;
	void **clearDefines()** throws java.sql.SQLException;
	void **defineColumnType(int i, int j)** throws java.sql.SQLException;
	void **defineColumnType(int i, int j, String s)** throws java.sql.SQLException;

表 5-4 OracleStatement インタフェース

拡張	メソッド シグネチャ
OracleStatement 拡張 java.sql.statement (続き)	void **defineColumnType(int i, int j, int k)** throws java.sql.SQLException;
	void **describe()** throws java.sql.SQLException;
	void **notify_close_rset(**) throws java.sql.SQLException;
	void **setAutoRefetch(boolean b)** throws java.sql.SQLException;
	void **setAutoRollback(int i)** throws java.sql.SQLException; (非推奨)
	void **setRowPrefetch(int i)** throws java.sql.SQLException;
	void **setWaitOption(int i)** throws java.sql.SQLException; (非推奨)

表 5-5 OracleResultSet インタフェース

拡張	メソッド シグネチャ

`OracleResultSet` 拡張 `java.sql.ResultSet`	`boolean` **`getAutoRefetch()`** `throws java.sql.SQLException;`
	
	`int` **`getFirstUserColumnIndex()`** `throws java.sql.SQLException;`
	
	`void` **`closeStatementOnClose()`** `throws java.sql.SQLException;`
	
	`void` **`setAutoRefetch(boolean b)`** `throws java.sql.SQLException;`
	
	`java.sql.ResultSet` **`getCursor(int n)`** `throws java.sql.SQLException;`
	
	`java.sql.ResultSet` **`getCURSOR(String s)`** `throws java.sql.SQLException;`

表 5-6 OracleCallableStatement インタフェース

拡張	メソッド シグネチャ

`OracleCallableStatement` 拡張 `java.sql.CallableStatement`	`void` **`clearParameters()`** `throws java.sql.SQLException;`
	
	`void` **`registerIndexTableOutParameter(int i, int j, int k, int l)`** `throws java.sql.SQLException;`
	
	`void` **`registerOutParameter (int i, int j, int k, int l)`** `throws java.sql.SQLException;`
	
	`java.sql.ResultSet` **`getCursor(int i)`** `throws java.sql.SQLException;`
	
	`java.io.InputStream` **`getAsciiStream(int i`**`)` `throws java.sql.SQLException;`
	
	`java.io.InputStream` **`getBinaryStream(int i)`** `throws java.sql.SQLException;`
	
	`java.io.InputStream` **`getUnicodeStream(int i)`** `throws java.sql.SQLException;`

.

表 5-7 OraclePreparedStatement インタフェース

拡張	メソッド シグネチャ
OraclePreparedStatement 拡張 OracleStatement および java.sql.PreparedStatement	int **getExecuteBatch()** throws java.sql.SQLException; void **defineParameterType(int i, int j, int k)** throws java.sql.SQLException; void **setDisableStmtCaching(boolean b)** throws java.sql.SQLException; void **setExecuteBatch(int i)** throws java.sql.SQLException; void **setFixedCHAR(int i, String s)** throws java.sql.SQLException; void **setInternalBytes(int i, byte[] b, int j)** throws java.sql.SQLException;

表 5-8 OracleArray インタフェース

拡張	メソッド シグネチャ
OracleArray 拡張 java.sql.Array	public **ArrayDescriptor getDescriptor()** throws java.sql.SQLException; public **Datum[] getOracleArray()** throws SQLException; public **Datum[] getOracleArray(long l, int i)** throws SQLException; public **String getSQLTypeName()** throws java.sql.SQLException; public **int length()** throws java.sql.SQLException; public **double[] getDoubleArray()** throws java.sql.SQLException; public **double[] getDoubleArray(long l, int i)** throws java.sql.SQLException; public **float[] getFloatArray()** throws java.sql.SQLException; public **float[] getFloatArray(long l, int i)** throws java.sql.SQLException; public **int[] getIntArray()** throws java.sql.SQLException; public **int[] getIntArray(long l, int i)** throws java.sql.SQLException; public **long[] getLongArray()** throws java.sql.SQLException; public **long[] getLongArray(long l, int i)** throws java.sql.SQLException;

表 5-8 OracleArray インタフェース

拡張	メソッド シグネチャ
OracleArray 拡張 java.sql.Array (続き)	public **short[] getShortArray()** throws java.sql.SQLException; public **short[] getShortArray(long l, int i)** throws java.sql.SQLException; public **void setAutoBuffering(boolean flag)** throws java.sql.SQLException; public **void setAutoIndexing(boolean flag)** throws java.sql.SQLException; public **boolean getAutoBuffering()** throws java.sql.SQLException; public **boolean getAutoIndexing()** throws java.sql.SQLException; public **void setAutoIndexing(boolean flag, int i)** throws java.sql.SQLException;

表 5-9 OracleStruct インタフェース

拡張	メソッド シグネチャ

OracleStruct 拡張 java.sql.Struct	public **Object[] getAttributes()** throws java.sql.SQLException;
	public **Object[] getAttributes(java.util.Dictionary map)** throws java.sql.SQLException;
	public **Datum[] getOracleAttributes()** throws java.sql.SQLException;
	public **oracle.sql.StructDescriptor getDescriptor()** throws java.sql.SQLException;
	public **String getSQLTypeName()** throws java.sql.SQLException;
	public void **setAutoBuffering(boolean flag)** throws java.sql.SQLException;
	public **boolean getAutoBuffering()** throws java.sql.SQLException;

表 5-10 OracleRef インタフェース

拡張	メソッド シグネチャ

OracleRef 拡張 java.sql.Ref	public **String getBaseTypeName()** throws SQLException;
	public **oracle.sql.StructDescriptor getDescriptor()** throws SQLException;
	public **oracle.sql.STRUCT getSTRUCT()** throws SQLException;
	public **Object getValue()** throws SQLException;
	public **Object getValue(Map map)** throws SQLException;
	public void **setValue(Object obj)** throws SQLException;

表 5-11 OracleThinBlob インタフェース

拡張	メソッド シグネチャ
OracleThinBlob 拡張 java.sql.Blob	int **getBufferSize()** throws java.sql.Exception int **getChunkSize(**)throws java.sql.Exception int **putBytes(long, int, byte[])** throws java.sql.Exception int **getBinaryOutputStream()** throws java.sql.Exception

表 5-12 OracleThinClob インタフェース

拡張	メソッド シグネチャ

OracleThinClob 拡張 java.sql.Clob	public OutputStream **getAsciiOutputStream()** throws java.sql.Exception;
	public Writer **getCharacterOutputStream()** throws java.sql.Exception;
	public int **getBufferSize()** throws java.sql.Exception;
	public int **getChunkSize()** throws java.sql.Exception;
	public char[] **getChars(long l, int i)** throws java.sql.Exception;
	public int **putChars(long start, char myChars[])** throws java.sql.Exception;
	public int **putString(long l, String s)** throws java.sql.Exception;

6 dbKona (非推奨) の使い方

dbKona クラスには、Java アプリケーションやアプレットからデータベースにアクセスできる高度なデータベース接続オブジェクトのセットが用意されています。dbKona は JDBC API の最上位に配置され、WebLogic JDBC ドライバ、またはその他の JDBC 対応のドライバと連動します。

以下の節では、dbKona クラス について説明します。

- 6-1 ページの「dbKona の概要」
- 6-4 ページの「dbKona API」
- 6-18 ページの「エンティティの関係」
- 6-19 ページの「dbKona の実装」

dbKona の概要

dbKona クラスには、データ管理に関する低レベルの詳細を扱う JDBC よりも高レベルな抽象化概念が備えられています。dbKona クラスから提供されるオブジェクトを使用することで、プログラマは、ベンダに依存しない、高レベルな方法でデータベース データを表示および変更できるようになります。dbKona オブジェクトを使用する Java アプリケーションでは、データベースに対してデータの検索、挿入、変更、削除などを行うにあたって、DBMS のテーブル構造やフィールド タイプに関するベンダ固有の知識は不要です。

多層コンフィグレーションでの dbKona

また、dbKona は、WebLogic Server および多層ドライバで構成される多層 JDBC 実装でも使用できます。このコンフィグレーションでは、クライアント サイド ライブラリは不要です。多層コンフィグレーションでは、WebLogic JDBC は、WebLogic 多層フレームワークへのアクセス メソッドとして機能します。WebLogic では、WebLogic jDriver for Oracle などの単一の JDBC ドライバを使用して、WebLogic Server から DBMS への通信を行います。

dbKona は、多層環境でデータベース アクセス プログラムを作成する場合によく使用されます。dbKona オブジェクトを使用すれば、ベンダにまったく依存しないデータベース アプリケーションを作成できるからです。dbKona および WebLogic の多層フレームワークは、ユーザに意識させずに、複数の異種データベースからデータを取り出すようなアプリケーションに特に適しています。

WebLogic と WebLogic JDBC サーバの詳細については、『WebLogic JDBC プログラマーズ ガイド』を参照してください。

dbKona と JDBC ドライバの相互作用

dbKona は、DBMS への接続とその維持を JDBC ドライバに依存しています。dbKona を使用するには、JDBC ドライバをインストールする必要があります。

- WebLogic jDriver for Oracle のネイティブ JDBC ドライバを使用している場合は、『WebLogic jDriver for Oracle のコンフィグレーションと使い方』にある説明に従って、使用しているオペレーティング システムに適した、WebLogic 提供の .dll、.sl、または .so ファイルをインストールします。
- WebLogic JDBC ドライバ以外の JDBC ドライバを使用している場合は、その JDBC ドライバのマニュアルを参照します。

JavaSoft の JDBC は、BEA が jDriver JDBC ドライバを作成するために実装した一連のインタフェースです。BEA の JDBC ドライバは、Oracle および Microsoft SQL Server 用のデータベース固有のドライバの JDBC 実装です。dbKona でデータベース固有のドライバを使用すると、パフォーマンスが向上するだけでなく、プログラマは各データベースのすべての機能にアクセスできます。

dbKona の基礎部分ではデータベース トランザクション用に JDBC が使用されていますが、dbKona を使用することによって、データベースへのより高レベルで便利なアクセスが可能になります。

dbKona と WebLogic Event の相互作用

dbKona パッケージには、ローカルまたは DBMS 内でデータが更新されるときに、WebLogic Event を使用してイベントを (WebLogic Server 内で) 送受信する「eventful」クラスが含まれています。

dbKona アーキテクチャ

dbKona では、データベースに存在するデータの記述および操作に、高レベルな抽象化概念が使用されます。dbKona のクラスは、データを検索および変更するオブジェクトの作成と管理を行います。特定のベンダのデータ保存方法や処理方法に関する知識がなくても、アプリケーションでは一貫性のある方法で dbKona オブジェクトを使用できます。

dbKona アーキテクチャの中心的な概念は、`DataSet` です。`DataSet` には、クエリの結果が含まれます。`DataSet` を使用すると、クライアントサイドでクエリ結果を管理できます。プログラマはレコードを 1 つずつ処理するのではなく、クエリ結果全体を制御できます。

`DataSet` には `Record` オブジェクトが含まれています。さらに各 `Record` オブジェクトには 1 つまたは複数の `Value` オブジェクトが含まれています。`Record` は、データベースの行に相当し、`Value` はデータベースのセルに相当します。`Value` オブジェクトは、DBMS に格納される場合の自身の内部データ型を「知っています」。しかし、プログラマはベンダ固有の内部データ型を気にせずに、一貫性のある方法で `Value` オブジェクトを使用できます。

`DataSet` クラス (およびそのサブクラス `TableDataSet` と `QueryDataSet`) のメソッドを使用すると、高レベルで柔軟な方法でクエリ結果を自在に操作できます。`TableDataSet` の変更内容は、DBMS に保存できます。この場合、dbKona

では変更したレコードについての情報が保持され、選択的に保存されます。これにより、ネットワークトラフィックおよび DBMS のオーバーヘッドが減少します。

また dbKona では、プログラマがベンダ固有の SQL を気にする必要のない、`SelectStmt` や `KeyDef` などのオブジェクトも使用できます。これらのクラスのメソッドを使用すると、dbKona によって適切な SQL が作成されます。ベンダ固有の SQL についての知識が不要な上、構文エラーも減少します。その一方で、dbKona では、プログラマは必要に応じて SQL を DBMS に渡すことができます。

dbKona API

以下の節では、dbKona API について説明します。

dbKona API リファレンス

`weblogic.db.jdbc` パッケージ

`weblogic.db.jdbc.oracle` パッケージ (Oracle 向け拡張)

```
java.lang.Object クラス
  weblogic.db.jdbc.Column クラス
     (weblogic.common.internal.Serializable の実装 )
  weblogic.db.jdbc.DataSet クラス
     (weblogic.common.internal.Serializable の実装 )
    weblogic.db.jdbc.QueryDataSet クラス
    weblogic.db.jdbc.TableDataSet クラス
           weblogic.db.jdbc.EventfulTableDataSet クラス
       (weblogic.event.actions.ActionDef の実装 )
    weblogic.db.jdbc.Enums クラス
    weblogic.db.jdbc.KeyDef クラス
    weblogic.db.jdbc.Record クラス
       weblogic.db.jdbc.EventfulRecord クラス
     (weblogic.common.internal.Serializable の実装 )
    weblogic.db.jdbc.Schema クラス
     (weblogic.common.internal.Serializable の実装 )
    weblogic.db.jdbc.SelectStmt クラス
    weblogic.db.jdbc.oracle.Sequence クラス
    java.lang.Throwable クラス
       java.lang.Exception クラス
```

weblogic.db.jdbc.DataSetException クラス

weblogic.db.jdbc.Value クラス

dbKona オブジェクトとそれらのクラス

dbKona のオブジェクトは、以下の 3 つのカテゴリに分けられます。

- データ コンテナ オブジェクトには、データベースから取り出されたデータやデータベースにバインドされたデータが保持されます。または、データを保持する他のオブジェクトが含まれます。データ コンテナ オブジェクトは、常に一連のデータ記述オブジェクトおよび一連のセッション オブジェクトに関連付けられます。TableDataSet オブジェクトや、Record オブジェクトは、データ コンテナ オブジェクトの例です。
- データ記述オブジェクトには、データ オブジェクトに関するメタデータが保持されます。メタデータとは、データ構造やデータ型、リモート DBMS からデータを取り出すためのパラメータなどを記述したものです。すべてのデータ オブジェクトまたはそのコンテナは、一連のデータ記述オブジェクトに関連付けられます。Schema オブジェクトや、SelectStmt オブジェクトは、このデータ記述オブジェクトの例です。
- その他のオブジェクトは、エラーに関する情報を格納したり、定数シンボルを提供したりします。

オブジェクトのこのような大きなカテゴリは、アプリケーションのビルドにおいて相互に依存し合っています。通常は、どのデータ オブジェクトにも、一連の記述オブジェクトが関連付けられています。

dbKona のデータ コンテナ オブジェクト

データ コンテナとして機能する基本的なオブジェクトには 3 種類あります。DataSet、Record、および Value の各オブジェクトです。DataSet (またはそのサブクラスである QueryDataSet あるいは TableDataSet) オブジェクトには Record オブジェクトが含まれ、Record オブジェクトには Value オブジェクトが含まれます。DataSet のサブクラス EventfulTableDataSet は非推奨になりました。

DataSet

dbKona パッケージでは DataSet の概念を使用して、DBMS サーバから取り出されたレコードをキャッシュできます。この概念は、SQL におけるテーブルとほぼ同じです。DataSet クラスには 2 つのサブクラス、QueryDataSet と TableDataSet があります。

WebLogic Server を使用する多層モデルでは、DataSet を WebLogic Server に保存 (キャッシュ) できます。

- DataSet は、クエリまたはストアド プロシージャの結果を保持する QueryDataSet または TableDataSet として作成されます。
- DataSet の検索パラメータは、SQL 文、または dbKona の SQL 文用の抽象オブジェクトである SelectStmt オブジェクトによって定義されます。
- DataSet には、Value オブジェクトを含む Record オブジェクトが含まれます。Record には、インデックス位置 (0 が起点) を指定してアクセスします。
- DataSet は、Schema によって記述され、Schema にバインドされます。Schema には、DataSet に表示される各データベース カラムの名前、データ型、サイズ、順番などの属性情報が格納されます。Schema 内のカラム名には、インデックス位置 (1 が起点) を指定してアクセスします。

DataSet クラス (weblogic.db.jdbc.DataSet を参照) は、QueryDataSet および TableDataSet の抽象的な親クラスです。

QueryDataSet

QueryDataSet を使用すると、SQL クエリの結果を、インデックス位置 (0 が起点) を指定してアクセスする Record のコレクションとして使用できます。TableDataSet とは異なり、QueryDataSet に対して変更および追加した内容はデータベースに保存できません。

QueryDataSet と TableDataSet には、機能的な違いが 2 つあります。1 番目の相違点は、TableDataSet の変更内容はデータベースに保存できるという点です。QueryDataSet の Record も変更できますが、その変更内容は保存できません。2 番目の相違点は、QueryDataSet には、複数のテーブルからのデータを取り出せるという点です。

- QueryDataSet は、java.sql.Connection のコンテキスト内で、または java.sql.ResultSet を使用して作成されます。つまり、Connection オブジェクトを引数として QueryDataSet コンストラクタに渡します。QueryDataSet のデータ検索は、SQL クエリや SelectStmt オブジェクトによって指定されます。
- QueryDataSet には、Record オブジェクト (0 が起点のインデックスを指定してアクセスする) が含まれます。Record オブジェクトには Value オブジェクト (1 が起点のインデックスを指定してアクセスする) が含まれます。
- QueryDataSet は、Schema によって記述されます。Schema には、QueryDataSet の属性に関する情報が格納されます。属性には、QueryDataSet に表示される各データベース カラムの名前、データ型、サイズ、順番などがあります。

QueryDataSet クラス (weblogic.db.jdbc.QueryDataSet を参照) には、QueryDataSet を作成、保存、および検索するためのメソッドがあります。QueryDataSet には、結合用の SQL など、任意の SQL を指定できます。そのスーパークラスである DataSet には、レコード キャッシュの詳細を管理するためのメソッドが含まれています。

TableDataSet

TableDataSet と QueryDataSet の機能的な違いは、TableDataSet の変更内容はデータベースに保存できるという点です。TableDataSet を使用すると、Record の値の更新、新しい Record の追加、および Record への削除のマーク付けができます。TableDataSet 全体を保存する場合は TableDataSet クラスの save() メソッドを使用し、1 つのレコードを保存する場合は Record クラスの save() メソッドを使用して、最終的に変更内容をデータベースに保存できます。さらに、TableDataSet に取り出されるデータは、定義上、単一のデータベース テーブルからのデータです。複数のデータベース テーブルを結合して TableDataSet にデータを取り出すことはできません。

更新情報または削除情報をデータベースに保存するには、KeyDef オブジェクトを使用して TableDataSet を作成する必要があります。KeyDef オブジェクトは、UPDATE 文または DELETE 文に WHERE 句を作成するためのユニークなキーを指定します。挿入の操作には WHERE 句は必要ないので、挿入だけを行う場合は、

KeyDef オブジェクトは不要です。KeyDef のキーには、DBMS によって入力または変更されるカラムが含まれないようにしてください。dbKona では、正しい WHERE 句を作成するためにキー カラムの値を把握しておく必要があるからです。

また、SQL 文の末尾を構成する任意の文字列で TableDataSet を限定することもできます。Oracle データベースで dbKona を使用している場合、たとえば「for UPDATE」などの文字列で TableDataSet を限定すると、クエリによって検索されるレコードをロックできます。

TableDataSet は、KeyDef を使用して作成できます。KeyDef は dbKona のオブジェクトであり、DBMS に更新情報および削除情報を保存するためのユニークなキーを設定する場合に使用されます。Oracle データベースを使用している場合は、TableDataSet の KeyDef をテーブルごとにユニークなキーである「ROWID」に設定できます。その後、「ROWID」を含む一連の属性を使用して、TableDataSet を作成します。

- TableDataSet は、java.sql.Connection のコンテキスト内で作成されます。つまり、Connection オブジェクトを引数として TableDataSet コンストラクタに渡します。そのデータ検索は、DBMS テーブルの名前によって指定されます。更新情報および削除情報を保存する場合は、TableDataSet の作成時に KeyDef オブジェクトを指定する必要があります。TableDataSet を作成した後で、where() メソッドおよび order() メソッドを使用してクエリを修正し、WHERE 句および ORDER BY 句を設定することもできます。
- TableDataSet には、関連付けられているデフォルトの SelectStmt オブジェクトがあります。このオブジェクトは、サンプルを使用したクエリ機能を利用する場合に使用されます。
- QueryDataSet には、Record オブジェクト (0 が起点のインデックスを指定してアクセスする) が含まれます。Record オブジェクトには Value オブジェクト (1 が起点のインデックスを指定してアクセスする) が含まれます。
- TableDataSet の属性は、schema によって記述されます。schema には、TableDataSet に表示されるデータベース カラムの名前、データ型、サイズ、順番などの TableDataSet の属性情報が格納されます。
- TableDataSet は、WebLogic Server サーバにキャッシュできます。
- setRefreshOnSave() メソッドは、保存中に挿入または更新されたレコードもすぐに DBMS から更新されるように、TableDataSet を設定します。

TableDataSet に DBMS によって変更されたカラム (Microsoft SQL Server の IDENTITY カラムや挿入または更新がきっかけとなって変更されたカラムなど) がある場合は、このフラグを設定します。

- Refresh() メソッドは、データベースに保存された TableDataSet 内のレコード、つまり TableDataSet で変更したレコードを更新します。レコードの変更内容は失われ、レコードには更新済みのマークが付きます。削除のマークが付けられたレコードは、更新されません。TableDataSet に追加されたレコードの場合は、更新元の DBMS の行が存在しないことを示す例外が生成されます。

- saveWithoutStatusUpdate() メソッドは、TableDataSet 内のレコードの保存状態を更新せずに DBMS に TableDataSet レコードを保存します。トランザクション内で TableDataSet レコードを保存する場合には、このメソッドを使用します。トランザクションがロールバックされても、TableDataSet 内のレコードはデータベースと一致しており、トランザクションを再試行できます。トランザクションのコミット後、updateStatus() を呼び出して TableDataSet 内のレコードの保存状態を更新します。一度、saveWithoutStatusUpdate() を使用してレコードを保存すると、そのレコードに対して updateStatus() を呼び出すまでレコードは変更できません。

- TableDataSet.setOptimisticLockingCol() メソッドを使用すると、TableDataSet の 1 つのカラムをオプティミスティック ロックのカラムとして指定できます。このカラムをアプリケーションで使用すると、データベースから読み込んでから他のユーザがその行を変更したかどうかを検出できます。dbKona では、行が変更されるたびに DBMS によってカラムが更新されるようになっているので、TableDataSet の値によってこのカラムが更新されることはありません。dbKona では、レコードまたは TableDataSet を保存するときに UPDATE 文の WHERE 句でこのカラムが使用されます。別のユーザがそのレコードを変更した場合は、dbKona による更新は失敗します。この場合、Record.refresh() を使用してそのレコードの新しい値を取り出し、レコードに変更を加えてから、再度保存を試みることができます。

TableDataSet クラス (weblogic.db.jdbc.TableDataSet を参照) には、次のメソッドがあります。

- TableDataSet を作成するためのメソッド
- WHERE 句および ORDER BY 句を設定するためのメソッド
- KeyDef を取得するためのメソッド
- 関連付けられた JDBC ResultSet を取得するためのメソッド
- SelectStmt を取得するためのメソッド
- 関連付けられた DBMS テーブル名を取得するためのメソッド
- 変更内容をデータベースに保存するためのメソッド
- DBMS からレコードを更新するためのメソッド
- 関連するその他情報を取得するためのメソッド

そのスーパークラスである DataSet には、レコード キャッシュを管理するためのメソッドが含まれています。

EventfulTableDataSet (非推奨)

WebLogic Server 内部で使用するための EventfulTableDataSet は、データがローカルまたは DBMS で更新されたときに、イベントを送信および受信する TableDataSet です。EventfulTableDataSet は、WebLogic Event のすべての Action クラスによって実装されるインタフェースである weblogic.event.actions.ActionDef を実装しています。EventfulTableDataSet の action() メソッドは、DBMS を更新し、同じ DBMS テーブルに関する他のすべての EventfulTableDataSet にその変更を通知します (WebLogic Event (非推奨) に関する詳細については、ホワイトペーパーおよび WebLogic Events の開発者ガイドを参照してください)。

EventfulTableDataSet の EventfulRecord が変更されると、WebLogic Server に ParamSet を持つ EventMessage が送信されます。ParamSet には、変更されたデータと行が含まれます。このとき、そのトピックは、WEBLOGIC.[tablename] になります。ここで tablename には EventfulTableDataSet に関連付けられたテーブルの名前が入ります。EventfulTableDataSet は、受信し、評価されたイベントに基づいて動作し、変更されたレコードの独自のコピーを更新します。

EventfulTableDataSet は、java.sql.Connection オブジェクトのコンテキスト内で作成されます (Connection オブジェクトを引数としてコンストラクタに渡す)。 また、t3 Client オブジェクト、挿入、更新、削除に使用される KeyDef オブジェクト、および DBMS のテーブル名も指定する必要があります。

- TableDataSet と同様、EventfulTableDataSet には、関連付けられているデフォルトの SelectStmt オブジェクトがあります。このオブジェクトは、サンプルを使用したクエリ機能を利用する場合に使用されます。
- EventfulTableDataSet には、EventfulRecord オブジェクト (0 が起点のインデックスを指定してアクセスする) が含まれます。Record オブジェクトのように、EventfulRecord オブジェクトには、Value オブジェクト (1 が起点のインデックスを指定してアクセスする) が含まれます。
- EventfulTableDataSet の属性は、Schema によって TableDataSet と同じ方法で記述されます。

たとえば、EventfulTableDataSet は、数多くのテーブル ビューを自動的に更新する、倉庫の在庫システムなどで使用されます。ここではその動作について説明します。各倉庫の従業員のクライアント アプリケーションが、「stock」テーブルから EventfulTableDataSet を作成し、そのレコードを Java アプリケーションに表示します。別の仕事をしている従業員は別の表示を見ていますが、すべてのクライアント アプリケーションでは、「stock」テーブルの同じ EventfulTableDataSet が使用されています。TableDataSet が「イベントフル」であるため、データ セット内の各レコードは自動的にそれ自身に対する関心を登録済みです。WebLogic のトピック ツリーには、すべてのレコードへの関心が登録されています。そこには、クライアントごとの、TableDataSet の各レコードに対する関心の登録があります。

ユーザがレコードを変更すると、DBMS は新しいレコードにより更新されます。同時に、EventMessage (変更された Record 自身が埋め込まれています) が自動的に WebLogic Server に送信されます。「Stock」テーブルの EventfulTableDataSet を使用している各クライアントは、変更された Record が埋め込まれたイベント通知を受信します。各クライアントの EventfulTableDataSet は、変更されたレコードを受け入れて GUI を更新します。

Record

Record オブジェクトは、DataSet の一部として作成されます。Record は、DataSet のコンテキストとその Schema、または現在アクティブになっている Database セッションで認識される SQL テーブルの Schema に基づいて手動で作成することもできます。

TableDataSet 内の Record は、Record クラスの save() メソッドを使用すれば個別に、または TableDataSet クラスの save() メソッドを使用すれば一括してデータベースに保存できます。

- Record は、DataSet が作成され、そのクエリが実行されたときに作成されます。また、Record は、(DataSet の fetchRecords() メソッドが呼び出されて、その Schema が取得された後で) DataSet の addRecord() メソッドまたは Record コンストラクタを使用して既存の DataSet に追加することもできます。
- Record には、Value のコレクションが含まれています。Record には、0 が起点のインデックス位置を指定してアクセスします。Record 内の Value には、1 が起点のインデックス位置を指定してアクセスします。
- Record は、その親の DataSet の Schema によって記述されます。Record に関連付けられた Schema には、Record 内の各フィールドの名前、データ型、サイズ、および順番などに関する情報が格納されます。

Record クラス (weblogic.db.jdbc.Record を参照) には、次のメソッドがあります。

- Record オブジェクトを作成するためのメソッド
- 親の DataSet および Schema を確認するためのメソッド
- Record 内のカラム数を確認するためのメソッド
- ステータスが保存なのか更新なのかを確認するためのメソッド
- データベースへの Record の保存または更新に使用する SQL 文字列を確認するためのメソッド
- Value の取得と設定を行うためのメソッド
- 各カラムの値をフォーマット文字列として返すためのメソッド

Value

Value オブジェクトには、親の DataSet の Schema によって定義される内部データ型があります。Value オブジェクトには、有効な割り当てであればその内部データ型以外のデータ型の値を割り当てることができます。 また、Value オブジェクトには、有効なリクエストであればその内部データ型以外のデータ型の値を返すこともできます。

Value オブジェクトでは、アプリケーションでベンダ固有のデータ型を操作しなくてもいいようになっています。Value オブジェクトはそのデータ型を「知っています」が、すべての Value オブジェクトはその内部データ型に関係なく同じメソッドを使用して Java アプリケーション内で操作できます。

- Value オブジェクトは、Record オブジェクトの作成時に作成されます。
- Value オブジェクトの内部データ型は、次のいずれかになります。
  - Boolean
  - Byte
  - Byte[]
  - Date
  - Double-precision
  - Floating-point
  - Integer
  - Long
  - Numeric
  - Short
  - String
  - Time
  - Timestamp
  - NULL

  これらのデータ型は、java.sql.Types に表示されている JDBC のタイプに対応しています。
- Value オブジェクトは、親の DataSet に関連付けられた Schema によって記述されます。

Value クラス (weblogic.db.jdbc.Value を参照) には、Value オブジェクトのデータおよびデータ型を取得および設定するためのメソッドがあります。

dbKona のデータ記述オブジェクト

データ記述オブジェクトには、メタデータが含まれます。メタデータとは、データ構造、DBMS へのデータの格納方法や DBMS からのデータの取り出し方法、データの更新方法などに関する情報のことです。dbKona では、JDBC インタフェースの実装として提供される次のデータ記述オブジェクトを使用します。

- Schema
- Column
- KeyDef
- SelectStmt

Schema

DataSet をインスタンス化すると、それを記述する Schema が暗黙に作成されます。そしてその Record を取り出すと、その DataSet Schema が更新されます。

- Schema は、DataSet がインスタンス化されるときに自動的に作成されます。
- DataSet の属性 (つまり、QueryDataSet と TableDataSet、およびそれらに関連付けられた Record の属性) は、テーブルの属性のように Schema によって定義されます。
- Schema 属性は、Column オブジェクトのコレクションとして記述されます。

Schema クラス (weblogic.db.jdbc.Schema を参照) には、次のメソッドがあります。

- Schema に関連付けられた Column を追加したり、返したりするためのメソッド
- Schema 内のカラム数を確認するためのメソッド
- Schema 内の特定のカラム名のインデックス位置 (1 が起点) を確認するためのメソッド

Column

Schema が作成されます。

Column クラス (weblogic.db.jdbc.Column を参照) には、次のメソッドがあります。

- Column を特定のデータ型に設定するためのメソッド
- Column のデータ型を確認するためのメソッド
- Column のデータベース固有のデータ型を確認するためのメソッド
- Column の名前、スケール、精度、およびストレージの長さを確認するためのメソッド
- ネイティブ DBMS カラムで NULL 値を使用できるかどうかを確認するためのメソッド
- Column が読み込み専用や検索可能になっているかどうかを確認するためのメソッド

KeyDef

特定のデータベース レコードをユニークなものとして識別し操作するための「WHERE attribute1 = value1 and attribute2 = value2」などのパターンです。KeyDef の属性は、データベース テーブルのユニークなキーに対応させる必要があります。

属性のない KeyDef オブジェクトは、KeyDef クラスで作成されます。addAttrib() メソッドを使用して、KeyDef の属性を作成してから、KeyDef を TableDataSet 用のコンストラクタで引数として使用します。KeyDef は、一度 DataSet に関連付けられると属性を追加することはできません。

Oracle データベースを使用している場合、属性「ROWID」を追加できます。「ROWID」は、各テーブルに関連付けられた本質的にユニークなキーであり、TableDataSet を使用した挿入および削除に使用されます。

KeyDef クラス (weblogic.db.jdbc.KeyDef を参照) には、次のメソッドがあります。

- 属性を追加するためのメソッド
- KeyDef オブジェクト中の属性数を確認するためのメソッド
- KeyDef オブジェクトに、特定のカラム名またはインデックス位置に対応する属性があるかどうかを確認するためのメソッド

SelectStmt

SelectStmt クラスの中に SelectStmt オブジェクトを作成することができます。その後、SelectStmt クラスのメソッドを使用して SelectStmt に句を追加し、その結果の SelectStmt オブジェクトを QueryDataSet を作成するときの引数として使用します。TableDataSet には、関連付けられたデフォルトの SelectStmt オブジェクトがあります。このオブジェクトを使用すると、TableDataSet が作成された後でデータ検索の精度を向上させることができます。

SelectStmt クラス (weblogic.db.jdbc.SelectStmt を参照) のメソッドは、SQL 文の次の句に対応しています。

- Field (およびエリアス)
- From
- Group
- Having
- Order by
- Unique
- 各要素の説明は次のとおりです。

また、サンプルを使用したクエリの句の設定および追加もサポートされています。from() メソッドでは、エリアスを含む文字列を「<i>tableName alias</i>」という形式で指定できます。field() メソッドでは、「<i>tableAlias.attribute</i>」という形式の文字列を引数として使用できます。テーブルの結合が役立つかどうかは使用法によりますが、SelectStmt オブジェクトを作成する場合には複数のテーブル名を使用できます。QueryDataSet に関連付けられた SelectStmt オブジェクトでは 1 つまたは複数

のテーブルを結合できますが、`TableDataSet` に関連付けられた SelectStmt オブジェクトではこれができません。定義上、使用できるのが 1 つのテーブルのデータに制限されているからです。

dbKona のその他オブジェクト

dbKona のその他オブジェクトには、例外、定数などがあります。

例外

- `DataSetException`
- `LicenseException`
- `java.sql.SQLException`

通常、`DataSetException` は、`DataSet` にストアド プロシージャによるエラーなどの問題がある場合や、内部 IO エラーがある場合などに発生します。

SQL 文の作成または DBMS サーバでの SQL 文の実行に問題がある場合は、`java.sql.SqlException` が送出されます。

定数

`Enums` クラスには、以下の項目用の定数が含まれます。

- トリガ状態
- ベンダ固有のデータベースのタイプ
- `INSERT`、`UPDATE`、および `DELETE` のデータベース操作

`java.sql.Types` クラスには、データ型用の定数が含まれています。

エンティティの関係

継承関係

以下に、dbKona クラス間の重要な継承関係を示します。1 つのクラスがサブクラス化されています。

DataSet

DataSet は、QueryDataSet および TableDataSet の抽象的な基本クラスです。

その他の dbKona オブジェクトは DbObject から派生します。

DataSetException や LicenseException などのほとんどの dbKona Exceptions は、java.lang.Exception および weblogic.db.jdbc.DataSetException のサブクラスです。LicenseException は RuntimeException のサブクラスです。

所有関係

各 dbKona オブジェクトには、その構造をさらに詳しく定義する、関連付けられたその他のオブジェクトがある場合もあります。

DataSet

DataSet には、Record オブジェクトがあり、各 Record オブジェクトには Value オブジェクトがあります。また、DataSet には、その構造を定義する Schema があり、これは 1 つまたは複数の Column で作成されています。さらに、DataSet には、データ検索用のパラメータを設定する SelectStmt がある場合もあります。

TableDataSet

TableDataSet には、キーによって更新および削除を行うための KeyDef があります。

Schema

Schema には、その構造を定義する Column があります。

dbKona の実装

以降の節では、リモート DBMS からデータを取り出して表示する単純な Java アプリケーションのビルド手順の概要を、一連のサンプルを使用して説明します。

dbKona を使用した DBMS へのアクセス

以下の手順では、dbKona を使用して DBMS にアクセスする方法について説明します。

手順 1. パッケージのインポート

dbKona を使用するアプリケーションは java.sql および weblogic.db.jdbc (WebLogic dbKona パッケージ) に加えて、使用する他の Java クラスにもアクセスする必要があります。以下の例では、ログイン プロセスで使用する java.util の Properties クラス、および weblogic.html パッケージもインポートします。

```
import java.sql.*;
import weblogic.db.jdbc.*;
import weblogic.html.*;
import java.Properties;
```

JDBC ドライバ用のパッケージは、インポートしないでください。JDBC ドライバは、接続段階で確立されます。バージョン 2.0 以降では、weblogic.db.common、weblogic.db.server、weblogic.db.t3client はいずれもインポートしないでください。

手順 2. 接続確立用のプロパティの設定

次のコード例は、Properties オブジェクトを作成するためのメソッドのサンプルです。このメソッドは、Oracle DBMS との接続を確立するために使用されます。各プロパティは、文字列をダブルクォテーション ("") で囲んで設定します。

```
public class tutor {

  public static void main(String argv[])
    throws DataSetException, java.sql.SQLException,
    java.io.IOException, ClassNotFoundException
   {
      Properties props = new java.util.Properties();
      props.put("user",               "scott");
      props.put("password",           "tiger");
      props.put("server",             "DEMO");
    ( 以降に続く )
```

Properties オブジェクトは、Connection を作成するための引数として使用されます。JDBC Connection オブジェクトは、その他のデータベース操作でも重要なコンテキストになります。

手順 3. DBMS との接続の確立

Connection オブジェクトは、Class.forName() メソッドで JDBC ドライバ クラスをロードし、次に java.sql.myDriver.connect() コンストラクタを呼び出すことにより作成されます。このコンストラクタは、使用する JDBC ドライバの URL と java.util.Properties オブジェクトの 2 つの引数を取ります。

Properties オブジェクトの作成方法については、手順 2 の props を参照してください。

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

Connection *conn* は、DBMS に関連するその他のアクション (たとえば、クエリ結果を保持する DataSet の作成など) のための引数となります。DBMS への接続の詳細については、使用しているドライバの開発者ガイドを参照してください。

Connection、DataSet (使用している場合は JDBC ResultSet)、および Statement は、それらの操作を終了するときに close() メソッドで閉じる必要があります。サンプルでは、この方法に従って、それらが明示的に閉じられています。

注意： java.sql.Connection のデフォルト モードでは、autoCommit が true に設定されています。Oracle の場合は、上記のサンプルのように autoCommit を false に設定するとパフォーマンスが向上します。

注意： DriverManager.getConnection() は同期メソッドなので、特定の状況では、アプリケーションがハングする原因となります。このため、DriverManager.getConnection() の代わりに **Driver.connect()** メソッドを使用することをお勧めします。

クエリの準備、およびデータの検索と表示

以下の手順では、クエリを準備し、データを検索および表示する方法について説明します。

手順 1. データ検索用のパラメータの設定

dbKona には、データ検索を行う場合に SQL 文を作成したり、その範囲を設定したりするためのパラメータを設定する方法がいくつかあります。ここでは、JDBC ResultSet の結果を使用し、DataSet を作成するという、dbKona と JDBC ドライバの基本的な相互作用について説明します。このサンプルでは、SQL 文を実行するのに Statement オブジェクトを使用しています。Statement オブジェクトは、JDBC Connection クラスのメソッドによって作成されます。また、ResultSet は、Statement オブジェクトを実行することによって作成されます。

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

Statement オブジェクトを使用して実行したクエリの結果を使用して、QueryDataSet をインスタンス化できます。この QueryDataSet は、JDBC ResultSet を使用して作成されます。

```
Statement stmt = conn.createStatement();
stmt.execute("SELECT * from empdemo");
ResultSet rs = stmt.getResultSet();
QueryDataset ds = new QueryDataSet(rs);
```

JDBC Statement の実行結果を使用することが、DataSet を作成する唯一の方法になります。この方法には、SQL に関する知識が必要であり、かつ、クエリの結果をあまり細かく指定することはできません (基本的には、JDBC の next() メソッドを使用すれば、レコード操作を繰り返すことができます)。dbKona を使用すると、レコードを検索するのに SQL の知識はあまり必要になりません。つまり、dbKona のメソッドを使用してクエリを設定することができ、一度レコードを保持する DataSet を作成すればレコードの操作をより細かく指定できます。

手順 2. クエリ結果用の DataSet の生成

SQL 文を作成する必要はありませんが、dbKona では SQL 文の特定の部分を設定するメソッドを使用する必要があります。DataSet (TableDataSet または QueryDataSet) をクエリの結果用に作成します。

たとえば、dbKona で最も単純なデータ検索は、TableDataSet に対するものです。TableDataSet の作成に必要なのは、Connection オブジェクトと検索する DBMS テーブル名だけです。Employee テーブル (エリアスは「empdemo」) を検索するサンプルを次に示します。

```
TableDataSet tds = new TableDataSet(conn, "empdemo");
```

TableDataSet は、DBMS テーブルの属性 (カラム) のサブセットを使用して作成できます。非常に大きなテーブルから数個のカラムだけを取り出す場合には、それらのカラムを指定する方がテーブル全体を検索するより効率的です。そのためには、コンストラクタの引数としてテーブル属性のリストを渡します。次に例を示します。

```
  TableDataSet tds = new TableDataSet(conn, "empdemo", "empno,
dept");
```

DBMS に変更内容を保存する場合や、1 つまたは複数のテーブルの結合を行ってデータを取り出すつもりがない場合は、TableDataSet を使用し、それ以外の場合は、QueryDataSet を使用します。次のサンプルでは、2 つの引数 (Connection オブジェクトと SQL 文の文字列) を取る QueryDataSet コンストラクタを使用しています。

```
  QueryDataSet qds = new QueryDataSet(conn, "select * from
empdemo");
```

実際には、DataSet クラスの fetchRecords() メソッドを呼び出すまではデータの受け取りは開始されません。DataSet を作成した後は、データ パラメータに引き続き修正を加えることができます。たとえば、where() メソッドを使用して、TableDataSet に取り出すレコードの選択精度を向上させることができます。where() メソッドは、dbKona が作成する SQL 文に WHERE 句を追加します。次のサンプルでは、WHERE 句を作成する where() メソッドを使用して、Employee テーブルからレコードを 1 つだけ取り出しています。

```
  TableDataSet tds = new TableDataSet(conn, "empdemo");
  tds.where("empno = 8000");
```

手順 3. 結果の取り出し

データ パラメータを設定したら、次の例のように DataSet クラスの fetchRecords() メソッドを呼び出します。

```
  TableDataset tds = new TableDataSet(conn, "empdemo", "empno,
  dept");
  tds.where("empno = 8000");
  tds.fetchRecords();
```

fetchRecords() メソッドは、特定の数のレコードを取り出す引数や、特定のレコードで始まるレコードを取り出す引数を取ることができます。次のサンプルでは、最初の 20 レコードのみを取り出し、残りは clearRecords() を使用して破棄しています。

```
  TableDataSet tds = new TableDataSet(conn, "empdemo", "empno,
  dept");
  tds.where("empno > 8000");
  tds.fetchRecords(20)
     .clearRecords();
```

非常に大きなクエリ結果を処理する場合は、一度に取り出すレコード数を少なくしてまずそれを処理し、DataSet を消去してから次の取り出しに進んだ方が良い場合もあります。次の取り出しまでの間に TableDataSet を消去するには、DataSet クラスから clearRecords() メソッドを使用します。次にそのサンプルを示します。

```
  TableDataSet tds = new TableDataSet(conn, "empdemo", "empno,
  dept");
  tds.where("empno > 2000");
  while (!tds.allRecordsRetrieved()) {
```

```
    tds.fetchRecords(100);
    // 100 個のレコードを処理する . .
    tds.clearRecords();
  }
```

また、リリース 2.5.3 で新規追加されたメソッドを使用して `DataSet` を再利用することもできます。その `DataSet.releaseRecords()` メソッドは、`DataSet` を閉じてすべての Record を解放しますが破棄は行いません。その `DataSet` を再利用して、新しいレコードを生成できますが、アプリケーションによって保持されている最初に使用した DataSet からのレコードは読み込み可能のままです。

手順 4. TableDataSet の Schema の検査

`TableDataSet` に関する `Schema` 情報を検査する簡単なサンプルを以下に示します。`Schema` クラスの `toString()` メソッドは、`TableDataSet` *tds* 用にクエリされるテーブル内のカラムの名前、タイプ、長さ、精度、スケール、NULL 許容の各属性を含む、改行で区切られたリストを表示します。

```
Schema sch = tds.schema();
System.out.println(sch.toString());
```

Statement オブジェクトを使用してクエリを作成した場合は、クエリが終了して、その結果を取り出した後で Statement オブジェクトを閉じる必要があります。

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

手順 5. htmlKona を使用したデータの検査

次のサンプルでは、htmlKona `UnorderedList` を使用してデータを検査する方法を示します。このサンプルでは、`DataSet.getRecord()` と `Record.getValue()` を使用して、for ループで各レコードを検査します。手順 2. で作成した QueryDataSet に取り出したレコードから収入額が最高である従業員の名前、ID、および給料を検索します。

```
// ( データベース セッション オブジェクトと QueryDataSet qds の作成 )
  UnorderedList ul = new UnorderedList();

  String name     = "";
  String id       = "";
  String salstr   = "";
  int sal         = 0;
  for (int i = 0; i < qds.size(); i++) {
  // レコードを取得する
```

```
      Record rec = qds.getRecord(i);
      int tmp = rec.getValue("Emp Salary").asInt();
    // htmlKona ListElement に給与額を追加する
      ul.addElement(new ListItem("$" + tmp));
    // この給与額とこれまでに見つかった給与最高額と比較する
      if (tmp > sal) {
      // この給与額が新しい最高額の場合には、その従業員の情報を取得する
        sal    = tmp;
        name   = rec.getValue("Emp Name").asString();
        id     = rec.getValue("Emp ID").asString();
        salstr = rec.getValue("Emp Salary").asString();
    }
```

手順 6. htmlKona を使用した結果の表示

htmlKona を使用すると、上記のサンプルで作成したような動的データを簡単に表示できます。次のサンプルは、クエリの結果を表示するページを動的に作成する方法を示しています。

```
HtmlPage hp = new HtmlPage();
hp.getHead()
  .addElement(new TitleElement("Highest Paid Employee"));
hp.getBodyElement()
  .setAttribute(BodyElement.bgColor, HtmlColor.white);
hp.getBody()
  .addElement(MarkupElement.HorizontalLine)
  .addElement(new HeadingElement("Query String: ", +2))
  .addElement(stmt.toString())
  .addElement(MarkupElement.HorizontalLine)
  .addElement("I examined the values: ")
  .addElement(ul)
  .addElement(MarkupElement.HorizontalLine)
  .addElement("Max salary of those employees examined is: ")
  .addElement(MarkupElement.Break)
  .addElement("Name: ")
  .addElement(new BoldElement(name))
  .addElement(MarkupElement.Break)
  .addElement("ID: ")
  .addElement(new BoldElement(id))
  .addElement(MarkupElement.Break)
  .addElement("Salary: ")
  .addElement(new BoldElement(salstr))
  .addElement(MarkupElement.HorizontalLine);

    hp.output();
```

手順 7. DataSet および接続のクローズ

```
  qds.close();
  tds.close();
```

DBMS への接続を閉じることも重要です。次のサンプルのように、接続を閉じるコードが、すべてのデータベース操作の最後に finally ブロック内に表示される必要があります。

```
  try {
  // 処理を行う
  }
      catch (Exception mye) {
  // 例外を検出し処理する
  }
  finally {
    try {conn.close();}
    catch (Exception e) {
      // 例外を処理する
    }
  }
```

コードのまとめ

```
import java.sql.*;
import weblogic.db.jdbc.*;
import weblogic.html.*;
import java.Properties;

public class tutor {

  public static void main(String[] argv)
       throws java.io.IOException, DataSetException,
       java.sql.SQLException, HtmlException,
       ClassNotFoundException
  {
    Connection conn = null;
    try {
      Properties props = new java.util.Properties();
      props.put("user",               "scott");
      props.put("password",           "tiger");
      props.put("server",             "DEMO");

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

    // TableDataSet オブジェクトを作成し、レコードを 10 個追加する
```

```
      TableDataSet tds = new TableDataSet(conn, "empdemo");
for (int i = 0; i < 10; i++) {
        Record rec = tds.addRecord();
      rec.setValue("empno", i)
         .setValue("ename", "person " + i)
         .setValue("esalary", 2000 + (i * 10));
    }

    // データを保存し TableDataSet を閉じる
    tds.save();
    tds.close();

    // QueryDataSet を作成し、テーブルへの追加分を取り出す
    Statement stmt = conn.createStatement();
    stmt.execute("SELECT * from empdemo");

    QueryDataSet qds = new QueryDataSet(stmt.getResultSet());
    qds.fetchRecords();

    // QueryDataSet 内のデータを使用する
    UnorderedList ul = new UnorderedList();

    String name     = "";
    String id       = "";
    String salstr   = "";
    int sal         = 0;
    for (int i = 0; i < qds.size(); i++) {
      Record rec = qds.getRecord(i);
      int tmp = rec.getValue("Emp Salary").asInt();
      ul.addElement(new ListItem("$" + tmp));
      if (tmp > sal) {
        sal    = tmp;
        name   = rec.getValue("Emp Name").asString();
        id     = rec.getValue("Emp ID").asString();
        salstr = rec.getValue("Emp Salary").asString();
      }
    }

    // htmlKona ページを使用して、取り出したデータと
    // その取り出しに使用された文を表示する
    HtmlPage hp = new HtmlPage();
      hp.getHead()
      .addElement(new TitleElement("Highest Paid Employee"));
    hp.getBodyElement()
      .setAttribute(BodyElement.bgColor, HtmlColor.white);
    hp.getBody()
      .addElement(MarkupElement.HorizontalLine)
      .addElement(new HeadingElement("Query String: ", +2))
      .addElement(stmt.toString())
      .addElement(MarkupElement.HorizontalLine)
      .addElement("I examined the values: ")
      .addElement(ul)
      .addElement(MarkupElement.HorizontalLine)
      .addElement("Max salary of those employees examined is: ")
      .addElement(MarkupElement.Break)
```

```
          .addElement("Name: ")
          .addElement(new BoldElement(name))
          .addElement(MarkupElement.Break)
          .addElement("ID: ")
          .addElement(new BoldElement(id))
          .addElement(MarkupElement.Break)
          .addElement("Salary: ")
          .addElement(new BoldElement(salstr))
          .addElement(MarkupElement.HorizontalLine);

          hp.output();

        // QueryDataSet を閉じる
        qds.close();
        }
      catch (Exception e) {
          // 例外を処理する
        }
      finally {
        // 接続を閉じる
          try {conn.close();}
          catch (Exception mye) {
          // 例外を処理する
          }
        }
      }
    }
```

各 Statement および各 DataSet を使用後に閉じていること、および finally ブロックで接続を閉じていることに注意してください。

SelectStmt オブジェクトを使用したクエリの作成

以下の手順では、SelectStmt オブジェクトを使用してクエリを作成する方法について説明します。

手順 1. SelectStmt パラメータの設定

TableDataSet を作成すると、TableDataSet は空の SelectStmt に関連付けられます。その後、その SelectStmt を変更してクエリを作成できます。次のサンプルでは、接続 *conn* は既に作成済みです。ここでは TableDataSet の SelectStmt にアクセスする方法を示します。

```
  TableDataSet tds = new TableDataSet(conn, "empdemo");
      SelectStmt sql = tds.selectStmt();
```

ここで SelectStmt オブジェクト用のパラメータを設定します。次のサンプルでは、各フィールドの最初の引数が属性名、2 番目の引数がエリアスです。このクエリは、収入が $2000 未満のすべての従業員に関する情報を取り出します。

```
sql.field("empno", "Emp ID")
       .field("ename", "Emp Name")
       .field("sal", "Emp Salary")
       .from("empdemo")
       .where("sal < 2000")
       .order("empno");
```

手順 2. QBE を使用したパラメータの修正

SelectStmt オブジェクトでも Query-by-example 機能を提供します。Query-by-example または QBE は、カラム、演算子、値という形式の句を使用して、データを取得するためのパラメータを作成します。たとえば、「empno = 8000」は、employee フィールド値 (「empno」、エリアスは「Emp ID」) が 8000 に等しい、1 つまたは複数のテーブル内のすべての行を選択できる Query-by-example の句です。

さらに、次のサンプルに示すように、SelectStmt クラスの setQbe() メソッドおよび addQbe() メソッドを使用することによって、データ選択用のパラメータを定義することもできます。これらのメソッドでは、Select 文の作成にベンダ固有の QBE 構文を使用できます。

```
sql.setQbe("ename", "MURPHY")
       .addUnquotedQbe("empno", "8000");
```

パラメータの定義が終わったら、2 番目のチュートリアルで実施したように、fetchRecords() メソッドを使用して DataSet にデータを取り込みます。

SQL 文を使用した DBMS データの変更

以下の手順では、SQL 文を使用して DBMS データを変更する方法について説明します。

手順 1. SQL 文の記述

変更するデータを取り出してその変更内容をリモート DBMS に保存する必要がある場合には、TableDataSet にそのデータを取り出さなければなりません。QueryDataSet に取り出しても変更を保存できないからです。

ほとんどの dbKona 操作同様、Properties オブジェクトおよび Driver オブジェクトを作成して Connection をインスタンス化することによって操作を開始する必要があります。

手順 1. SQL 文の記述

```
String insert = "insert into empdemo(empno, " +
                    "ename, job, deptno) values " +
                    "(8000, 'MURPHY', 'SALESMAN', 10)";
```

2 番目の SQL 文は、名前「Murphy」を「Smith」に変更し、ジョブ ステータスを「Salesman」から「Manager」に変更するものです。

```
String update = "update empdemo set ename = 'SMITH', " +
                    "job = 'MANAGER' " +
                    "where empno = 8000";
```

3 番目の SQL 文は、データベースからこのレコードを削除するものです。

```
String delete = "delete from empdemo where empno = 8000";
```

手順 2. 各 SQL 文の実行

まず、テーブルのスナップショットを TableDataSet に保存します。その後、各 TableDataSet を検査して実行結果が予想どおりであるかどうかを検証します。TableDataSet は実行されたクエリの結果によってインスタンス化されるということに注意してください。

```
Statement stmt1 = conn.createStatement();
stmt1.execute(insert);

TableDataSet ds1 = new TableDataSet(conn, "emp");
ds1.where("empno = 8000");
ds1.fetchRecords();
```

TableDataSet に関連付けられたメソッドを使用すると、SQL の WHERE 句や ORDER BY 句を指定したり、QBE 文を設定および追加したりできます。このサンプルでは、各文を実行して execute() メソッドの結果を調べた後に、TableDataSet を使用してデータベース テーブル「emp」を再クエリしています。「WHERE」句を使用して、テーブル内のレコードを従業員番号が 8000 のレコードに限定しています。

UPDATE 文および DELETE 文に対して execute() メソッドを繰り返して、さらに 2 つの TableDataSet、ds2 および ds3 に結果を格納します。

手順 3. htmlKona を使用した結果の表示

```
  ServletPage hp = new ServletPage();
  hp.getHead()
    .addElement(new TitleElement("Modifying data with SQL"));
  hp.getBody()
    .addElement(MarkupElement.HorizontalLine)
    .addElement(new TableElement(tds))
    .addElement(MarkupElement.HorizontalLine)
    .addElement(new HeadingElement("Query results afer INSERT", 2))
    .addElement(new HeadingElement("SQL: ", 3))
    .addElement(new LiteralElement(insert))
    .addElement(new HeadingElement("Result: ", 3))
    .addElement(new LiteralElement(ds1))
    .addElement(MarkupElement.HorizontalLine)
     .addElement(new HeadingElement("Query results after UPDATE",
2))
    .addElement(new HeadingElement("SQL: ", 3))
    .addElement(new LiteralElement(update))
    .addElement(new HeadingElement("Result: ", 3))
    .addElement(new LiteralElement(ds2))
    .addElement(MarkupElement.HorizontalLine)
     .addElement(new HeadingElement("Query results after DELETE",
2))
    .addElement(new HeadingElement("SQL: ", 3))
    .addElement(new LiteralElement(delete))
    .addElement(new HeadingElement("Result: ", 3))
    .addElement(new LiteralElement(ds3))
    .addElement(MarkupElement.HorizontalLine);
  hp.output();
```

コードのまとめ

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;
import java.util.*;
import weblogic.db.jdbc.*;
```

```
import weblogic.html.*;

public class InsertUpdateDelete extends HttpServlet {

  public synchronized void service(HttpServletRequest req,
                                   HttpServletResponse res)
    throws IOException
  {
    Connection conn = null;
    try {
      res.setStatus(HttpServletResponse.SC_OK);
      res.setContentType("text/html");

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

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

      // 関連付けられた SelectStmt を持つ TableDataSet を作成する
      TableDataSet tds = new TableDataSet(conn, "empdemo");
      SelectStmt sql = tds.selectStmt();
      sql.field("empno", "Emp ID")
         .field("ename", "Emp Name")
         .field("sal", "Emp Salary")
         .from("empdemo")
         .where("sal < 2000")
         .order("empno");
      sql.setQbe("ename", "MURPHY")
         .addUnquotedQbe("empno", "8000");
    tds.fetchRecords();

      String insert = "insert into empdemo(empno, " +
                      "ename, job, deptno) values " +
                      "(8000, 'MURPHY', 'SALESMAN', 10)";

      // 文を作成して、実行する
      Statement stmt1 = conn.createStatement();
      stmt1.execute(insert);
      stmt1.close();

      // 結果を検証する
      TableDataSet ds1 = new TableDataSet(conn, "empdemo");
      ds1.where("empno = 8000");
      ds1.fetchRecords();

      // 文を作成して、実行する
      String update = "update empdemo set ename = 'SMITH', " +
                      "job = 'MANAGER' " +
                      "where empno = 8000";
```

```
        Statement stmt2 = conn.createStatement();
        stmt2.execute(insert);
        stmt2.close();

        // 結果を検証する
        TableDataSet ds2 = new TableDataSet(conn, "empdemo");
        ds2.where("empno = 8000");
        ds2.fetchRecords();

        // 文を作成して、実行する
        String delete = "delete from empdemo where empno = 8000";


        Statement stmt3 = conn.createStatement();
        stmt3.execute(insert);
        stmt3.close();

        // 結果を検証する
        TableDataSet ds3 = new TableDataSet(conn, "empdemo");
        ds3.where("empno = 8000");
        ds3.fetchRecords();

        // 結果を表示するサーブレット ページを作成する
        ServletPage hp = new ServletPage();
        hp.getHead()
          .addElement(new TitleElement("Modifying data with SQL"));
        hp.getBody()
          .addElement(MarkupElement.HorizontalRule)
          .addElement(new HeadingElement("Original table", 2))
          .addElement(new TableElement(tds))
          .addElement(MarkupElement.HorizontalRule)
         .addElement(new HeadingElement("Query results afer INSERT",
2))
          .addElement(new HeadingElement("SQL: ", 3))
          .addElement(new LiteralElement(insert))
          .addElement(new HeadingElement("Result: ", 3))
          .addElement(new LiteralElement(ds1))
          .addElement(MarkupElement.HorizontalRule)
        .addElement(new HeadingElement("Query results after UPDATE",
2))
          .addElement(new HeadingElement("SQL: ", 3))
          .addElement(new LiteralElement(update))
          .addElement(new HeadingElement("Result: ", 3))
          .addElement(new LiteralElement(ds2))
          .addElement(MarkupElement.HorizontalRule)
        .addElement(new HeadingElement("Query results after DELETE",
2))
          .addElement(new HeadingElement("SQL: ", 3))
          .addElement(new LiteralElement(delete))
          .addElement(new HeadingElement("Result: ", 3))
          .addElement(new LiteralElement(ds3))
          .addElement(MarkupElement.HorizontalRule);

        hp.output();

    tds.close();
```

```
            ds1.close();
            ds2.close();
            ds3.close();
          }
        catch (Exception e) {
            // 例外を処理する


          }
          // 常に、finally ブロック内で接続を閉じる
        finally {
         conn.close();
          }
        }
      }
```

KeyDef を使用した DBMS データの変更

KeyDef オブジェクトを使用して、リモート DBMS に対するデータの削除および挿入用のキーを取得します。KeyDef は、「WHERE KeyDef attribute1 = value1 and KeyDef attribute2 = value2」というパターンに従って、更新および削除を行う場合に文と同じように機能します。

最初の手順は、DBMS への接続を確立することです。ここで示すサンプルでは、最初のチュートリアルで作成した Connection オブジェクト *conn* を使用します。また、使用するデータベース テーブルは、empno、ename、job、および deptno の各フィールドを持つ Employee テーブル (「empdemo」) です。実行するクエリは、テーブル empdemo の内容をすべて取り出します。

手順 1. KeyDef とその属性の作成

このチュートリアルで挿入および削除用に作成する KeyDef オブジェクトには、データベースの empno カラムという 1 つの属性があります。この属性を持った KeyDef を作成すると、WHERE empno = および保存する各レコードの empno に割り当てられている特定の値、というパターンに従ったキーが設定されます。

KeyDef オブジェクトは、次のサンプルに示すように、KeyDef クラス内で作成されます。

```
KeyDef key = new KeyDef().addAttrib("empno");
```

Oracle データベースを使用している場合は、属性「ROWID」を持った KeyDef を作成して、次のサンプルのようにこの Oracle キーで挿入および削除を行うことができます。

```
KeyDef key = new KeyDef().addAttrib("ROWID");
```

手順 2. KeyDef を使用した TableDataSet の作成

次の例では、クエリの結果を使用して TableDataSet を作成します。引数として Connection オブジェクト、DBMS テーブル名、および KeyDef を取る TableDataSet コンストラクタを使用します。

```
TableDataSet tds = new TableDataSet(conn, "empdemo", key);
```

KeyDef は、データに対して行うすべての変更の参照になります。TableDataSet を保存するたびに、KeyDef 属性の値 (および SQL UPDATE、INSERT、DELETE の各操作に設定された制限) に基づいて、データベースのデータを変更します。このサンプルでは、属性は従業員番号 ("empno") です。

Oracle データベースを使用し、属性 ROWID を KeyDef に追加した場合は、次のように挿入および削除用の TableDataSet を作成できます。

```
KeyDef key = new KeyDef().addAttrib("ROWID");
TableDataSet tds =
    new TableDataSet(conn, "empdemo", "ROWID, dept", key);
tds.where("empno < 100");
  tds.fetchRecords();
```

手順 3. TableDataSet へのレコードの挿入

TableDataSet のコンテキストで新しい Record オブジェクトを作成できます。新しいオブジェクトは、TableDataSet クラスの addRecord() メソッドを使用して TableDataSet に追加されます。レコードを追加すると、Record クラスの setValue() メソッドを使用して、レコードの各フィールドの値を設定できます。レコードをデータベース (KeyDef フィールド) に保存するには、新しい Record で少なくとも 1 つの値を設定する必要があります。

```
Record newrec = tds.addRecord();
newrec.setValue("empno",  8000)
      .setValue("ename",  "MURPHY")
      .setValue("job",    "SALESMAN")
```

```
        .setValue("deptno", 10);
  String insert = newrec.getSaveString();
  tds.save();
```

Record クラスの getSaveString() メソッドは、Record をデータベースに保存する場合に使用される、SQL 文字列 (SQL の UPDATE 文、DELETE 文、または INSERT 文) を返します。この文字列をオブジェクトに保存し、後でそのオブジェクトを表示させることで、挿入操作が実際どのように実行されたのかを確認できます。

手順 4. TableDataSet でのレコードの更新

setValue() メソッドを使用して Record を更新することもできます。次の例では、前の手順で作成したレコードを変更します。次の例では、前の手順で作成したレコードを変更します。

```
  newrec.setValue("ename", "SMITH")
        .setValue("job",   "MANAGER");
  String update = newrec.getSaveString();
  tds.save();
```

手順 5. TableDataSet からのレコードの削除

Record クラスの markToBeDeleted() メソッドを使用して、TableDataSet のレコードに、削除するためのマークを付けることができます (または、unmarkToBeDeleted() メソッドでマークを解除できます)。たとえば、作成したばかりのレコードを削除するには、次のように、削除するレコードにマークを付けます。

```
  newrec.markToBeDeleted();
  String delete = newrec.getSaveString();
  tds.save();
```

削除するようにマークが付けられたレコードは、save() メソッドを実行するか、または TableDataSet クラスの removeDeletedRecords() メソッドを実行するまでは TableDataSet から削除されません。

(removeDeletedRecords() メソッドにより) TableDataSet から削除されたが、まだデータベースからは削除されていないレコードは、ゾンビ状態になります。レコードがゾンビ状態かどうかは、次のように Record クラスの isAZombie() メソッドを使用して確認できます。

```
if (!newrec.isAZombie()) {
  . . .
}
```

手順 6. TableDataSet の保存の詳細

Record または TableDataset を保存すると、データベースにデータが効率的に保存されます。dbKona では選択的に変更が行われます。つまり、変更されたデータのみが保存されます。TableDataSet 内のレコードを挿入、更新、および削除しても、Record.save() メソッドまたは TableDataSet.save() メソッドが実行されるまでは TableDataSet 内のデータだけが影響を受けます。

保存前の Record 状態の確認

Record クラスのメソッドの中には、save() を実行する前に Record の状態に関する情報を返すメソッドがあります。以下にその一部を示します。

needsToBeSaved() および recordIsClean()

needsToBeSaved() メソッドを使用すると、Record を保存する必要があるかどうかを確認できます。つまり、Record が取り出されてから、または、前回保存されてから、変更されたかどうかを確認できます。recordIsClean() メソッドは、Record にある Value のいずれかを保存する必要があるかどうかを確認するために使用します。このメソッドは、スケジュールされたデータベース操作が挿入、更新、または削除のいずれであるかに関係なく、Record が変更されている状態かどうかを確認するだけです。操作のタイプ (挿入 / 更新 / 削除) にかかわらず、needsToBeSaved() メソッドを save() メソッドの後で実行すると、false が返されます。

valueIsClean(int)

Record 内の特定のインデックス位置にある Value を保存する必要があるかどうかを確認します。このメソッドは、Value のインデックス位置を引数に取ります。

toBeSavedWith...()

toBeSavedWithDelete()、toBeSavedWithInsert()、および toBeSavedWithUpdate() の各メソッドを使用すると、特定の SQL アクションと共に、Record がどのように保存されるかを確認できます。これらのメソッドのセマンティクスは、「この行が変更されている、また

は変更される場合、TableDataSet を保存するときにどのようなアクションが行われるか」という問いに対する答えと同じです。

行が DBMS への保存対象かどうかを知るには、isClean() メソッドと needsToBeSaved() メソッドを使用します。

Record または TableDataSet を変更する場合は、いずれかのクラスの save() メソッドを使用して、その変更内容をデータベースに保存します。上記の手順では、各トランザクションの後で次のように TableDataSet を保存しました。

```
tds.save();
```

手順 7. 変更内容の検証

レコードを 1 つだけ取り出す場合のサンプルを以下に示します。この方法は、1 レコードの変更内容を検証するには、効率的な方法です。このサンプルでは、query-by-example (QBE) の句を使用して TableDataSet から関心のあるレコードだけを取り出しています。

```
TableDataSet tds2 = new TableDataSet(conn, "empdemo");
tds2.where("empno = 8000")
    .fetchRecords();
```

最後の手順として、各手順の後、および各 save() メソッドの後に作成した「insert」、「update」、および「delete」の各文字列の後にクエリ結果を表示できます。結果を表示する htmlKona の使用方法については、前のチュートリアルの「コードのまとめ」を参照してください。

DataSet の操作を終了したら、次のように close() メソッドを使用して各 DataSet を閉じます。

```
tds.close();
tds2.close();
```

コードのまとめ

次に、この節で説明した概念を使用するサンプル コードを示します。

```
package tutorial.dbkona;

import weblogic.db.jdbc.*;
import java.sql.*;
```

```
import java.Properties;

public class rowid {

  public static void main(String[] argv)
    throws Exception
  {
    Driver myDriver = (Driver)
    Class.forName("weblogic.jdbc.oci.Driver").newInstance();
    conn =
      myDriver.connect("jdbc:weblogic:oracle:DEMO",
                                    "scott",
                                    "tiger");

    // ここで、レコードを 100 個挿入する
    TableDataSet ts1 = new TableDataSet(conn, "empdemo");
    for (int i = 1; i <= 100; i++) {
      Record rec = ts1.addRecord();
      rec.setValue("empid", i)
         .setValue("name",  "Person " + i)
         .setValue("dept",  i);
    }

    // 新しいレコードを保存する。dbKona は選択的に保存を行う
    // つまり、TableDataSet 内の変更されたレコードだけを保存し、
    // ネットワーク トラフィックとサーバ呼び出しを削減する
    System.out.println("Inserting " + ts1.size() + " records.");
    ts1.save();
    // 処理が完了したので DataSet を閉じる
    ts1.close();

    // 更新および削除用の KeyDef を定義する
    // ROWID は Oracle 固有のフィールドで、更新および削除用の
    // 主キーとして機能することができる
    KeyDef key = new KeyDef().addAttrib("ROWID");

    // 最初に追加した 100 個のレコードを更新する
    TableDataSet ts2 =
      new TableDataSet(conn, "empdemo", "ROWID, dept", key);
    ts2.where("empid <= 100");
    ts2.fetchRecords();

    for (int i = 1; i <= ts2.size(); i++) {
      Record rec = ts2.getRecord(i);
      rec.setValue("dept", i + rec.getValue("dept").asInt());
    }

    // 更新されたレコードを保存する
    System.out.println("Update " + ts2.size() + " records.");
    ts2.save();

    // 同じ 100 個のレコードを削除する
    ts2.reset();
    ts2.fetchRecords();
```

```
    for (int i = 0; i < ts2.size(); i++) {
      Record rec = ts2.getRecord(i);
      rec.markToBeDeleted();
    }

    // レコードをサーバから削除する
    System.out.println("Delete " + ts2.size() + " records.");
    ts2.save();

    // DataSet、ResultSet、および Statement は、
    // 操作が終わったら必ず閉じる必要がある
    ts2.close();

    // 最後に、必ず接続を閉じる
   conn.close();
  }
}
```

dbKona での JDBC PreparedStatement の使い方

dbKona では構文的に正しい SQL 文が作成されるため、ベンダ固有の SQL の記述方法について知識がそれほど必要ないという点で便利です。しかし、dbKona で JDBC の `PreparedStatement` を使用できる場合もあります。

JDBC `PreparedStatement` は、複数回使用される SQL 構文をあらかじめコンパイルする場合に使用されます。`PreparedStatement` のパラメータは、`PreparedStatement.clearParameters()` を呼び出すことで消去できます。

`PreparedStatment` オブジェクトは、JDBC `Connection` クラス (これまでのサンプルで *conn* という名前で使用されていたオブジェクト) の `preparedStatement()` メソッドを使用して作成されます。次のサンプルでは、`PreparedStatement` を作成してそれをループの中で実行しています。この文には、従業員 ID、名前、および部署という 3 つの入力 (IN) パラメータがあります。このサンプルでは、100 人の従業員をテーブルに追加します。

```
String inssql = "insert into empdemo(empid, " +
                "name, dept) values (?, ?, ?)";
PreparedStatement pstmt = conn.prepareStatement(inssql);

for (int i = 1; i <= 100; i++) {
   pstmt.setInt(1, i);
   pstmt.setString(2, "Person " + i);
   pstmt.setInt(3, i);
   pstmt.executeUpdate();
}
```

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

作業が終了したら、Statement オブジェクトまたは PreparedStatement オブジェクトを必ず閉じます。

SQL を意識せずに同じタスクを dbKona で実行することもできます。この場合、KeyDef を使用して、更新または削除するフィールドを設定します。詳細については、チュートリアルの 6-34 ページの「KeyDef を使用した DBMS データの変更」を参照してください。

dbKona でのストアド プロシージャの使い方

固有のタスク (システムまたはベンダに依存しないタスクである場合が多い) を実行できる、リモート マシンに格納されたプロシージャや関数にアクセスして、dbKona の能力を向上させることができます。ストアド プロシージャおよび関数を使用するには、dbKona の Java アプリケーションとリモート マシンの間でリクエストがどのように受け渡しされるかを理解する必要があります。ストアド プロシージャまたは関数を実行すると、入力されたパラメータの値が変更されます。また、実行が成功したか失敗したかを示す値も返されます。

dbKona アプリケーションでの最初の手順は、DBMS に接続することです。ここで示すサンプルでは、最初のチュートリアルで作成した同じ Connection オブジェクト conn を使用します。

手順 1. ストアド プロシージャの作成

DBMS の CREATE の呼び出しを実行することにより、Statement オブジェクトを使用してストアド プロシージャを作成します。次の例では、パラメータ「field1」が integer 型の入出力として宣言されます。

```
Statement stmt1 = conn.createStatement();
stmt1.execute("CREATE OR REPLACE PROCEDURE proc_squareInt " +
              "(field1 IN OUT INTEGER, " +
              "field2 OUT INTEGER) IS " +
              "BEGIN field1 := field1 * field1; " +
              "field2 := field1 * 3; " +
              "END proc_squareInt;");
stmt1.close();
```

手順 2. パラメータの設定

JDBC Connection クラスの prepareCall() メソッド

次のサンプルでは、setInt() メソッドを使用して第 1 パラメータに整数「3」を設定しています。第 2 パラメータは、java.sql.Types.INTEGER 型の OUT パラメータとして登録します。最後にストアド プロシージャを実行します。

```
CallableStatement cstmt =
  conn.prepareCall("BEGIN proc_squareInt(?, ?):END;");
cstmt.setInt(1, 3);
cstmt.registerOutParameter(2, java.sql.Types.INTEGER);
cstmt.execute();
```

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。その代わり、「:1」、「:2」などを使用します。dbKona では、SQL でどちらも使用できます。

手順 3. 結果の検査

最も単純なメソッドを使用して結果を画面に出力します。

```
System.out.println(cstmt.getInt(1));
System.out.println(cstmt.getInt(2));
cstmt.close();
```

画像およびオーディオ用バイト配列の使い方

サイズの大きいバイナリ オブジェクト ファイルを、バイト配列を使用してデータベースから取り出したりデータベースに保存したりできます。データベースでデータを管理することの多いマルチメディア アプリケーションでは、画像ファイルやサウンド ファイルのようなサイズの大きいデータを処理する必要があります。

ここでは、htmlKona の便利さも理解できます。htmlKona を使用すれば、dbKona を使って取り出したデータベース データを HTML 環境に簡単に統合できます。このチュートリアルで使用するサンプルは、htmlKona に依存しています。

手順 1. 画像データの検索と表示

次のサンプルでは、htmlKona フォームで送信され、Netscape サーバで動作しているサーバサイド Java を使用して、ユーザが表示する画像の名前を取り出しています。その画像名を使って「imagetable」という名前のデータベース テーブルの内容をクエリし、その結果得られる最初のレコードを取得します。SelectStmt オブジェクトを使用して QBE によって SQL クエリを作成しています。

画像レコードを取り出した後、HTML ページのタイプを画像タイプに設定し、それから画像データをバイト配列 (byte[]) として取り出して htmlKona ImagePage に入れます。これにより、ブラウザに画像が表示されます。

```
if (iname != null) {
  // 画像をデータベースから取り出す
  TableDataSet tds = new TableDataSet(conn, "imagetable");
  tds.selectStmt().setQbe("name", iname);
  tds.fetchRecords();

  Record rec = tds.getRecord(0);

  this.returnNormalResponse("image/" +
                            rec.getValue("type").asString());

  ImagePage hp = new ImagePage(rec.getValue("data").asBytes());
  hp.output(getOutputStream());
}
```

手順 2. データベースへの画像の挿入

dbKona を使用してデータベースに画像ファイルを挿入することもできます。次に、データベースにタイプ配列オブジェクトとして 2 つの画像を追加するコードの抜粋を示します。この処理は、各画像の Record を TableDataSet へ追加し、Record の Values を設定して、TableDataSet を保存することにより行われます。

```
TableDataSet tds = new TableDataSet(conn, "imagetable");
Record rec = tds.addRecord();
rec.setValue("name", "vars")
   .setValue("type", "gif")
   .setValue("data", "c:/html/api/images/variables.gif");

rec = tds.addRecord();
rec.setValue("name", "excepts")
```

```
    .setValue("type", "jpeg")
    .setValue("data", "c:/html/api/images/exception-index.jpg");

tds.save();
tds.close();
```

Oracle シーケンス用の dbKona の使い方

dbKona では、Oracle シーケンスの機能にアクセスするためのラッパー、つまり、`Sequence` オブジェクトが用意されています。Oracle シーケンスは、シーケンスに開始番号とインクリメント間隔 (増分) を指定することによって dbKona で作成されます。

以下の節では、Oracle シーケンス用の dbKona の使い方について説明します。

手順 1. dbKona Sequence オブジェクトの作成

JDBC Connection と Oracle サーバに既に存在するシーケンスの名前を使用して、`Sequence` オブジェクトを作成します。次に例を示します。

```
Sequence seq = new Sequence(conn, "mysequence");
```

手順 2. dbKona からの Oracle サーバのシーケンスの作成と破棄

Oracle シーケンスが存在しない場合は、dbKona から `Sequence.create()` メソッドを使用して作成できます。このメソッドは、JDBC `Connection`、作成するシーケンスの名前、インクリメント間隔、および開始点の 4 つの引数を取ります。次のサンプルでは、開始点が 1000 でインクリメント間隔が 1 の Oracle シーケンス「mysequence」を作成しています。

```
Sequence.create(conn, "mysequence", 1, 1000);
```

次のように Oracle シーケンスを dbKona から削除できます。

```
Sequence.drop(conn, "mysequence");
```

手順 3. Sequence の使い方

Sequence オブジェクトを作成したら、このオブジェクトを使用して自動的にインクリメントする int を生成できます。たとえば、レコードをテーブルに追加するたびに自動的にインクリメントするキーを設定できます。nextValue() メソッドを使用して、Sequence の 次のインクリメントである int を返します。次に例を示します。

```
  TableDataSet tds = new TableDataSet(conn, "empdemo");
    for (int i = 1; i <= 10; i++) {
        Record rec = tds.addRecord();
        rec.setValue("empno", seq.nextValue());
    }
```

currentValue() メソッドを使用して、Sequence の現在の値を確認できます。ただし、このメソッドは、nextValue() メソッドを少なくとも一度呼び出した後でなければ呼び出せません。

```
  System.out.println("Records 1000-" + seq.currentValue() + "
added.");
```

コードのまとめ

次に、この節で説明した概念の使い方を示すサンプル コードを示します。最初に、Oracle サーバから「testseq」という名前のシーケンスを削除して、その名前のシーケンスが既に 1 つ存在している場合に、同じ名前のシーケンスを作成してもエラーが出力されないようにしています。その後、サーバ上にシーケンスを作成し、その名前で dbKona Sequence オブジェクトを作成しています。

```
package tutorial.dbkona;

import weblogic.db.jdbc.*;
import weblogic.db.jdbc.oracle.*;
import java.sql.*;
import java.Properties;

public class sequences {

  public static void main(String[] argv)
    throws Exception
  {
    Connection conn = null;
Driver myDriver = (Driver)
    Class.forName("weblogic.jdbc.oci.Driver").newInstance();
    conn =
      myDriver.connect("jdbc:weblogic:oracle:DEMO",
```

```
                                            "scott",
                                            "tiger");

    // シーケンスがサーバ上に既に存在する場合には、それを削除する
    try {Sequence.drop(conn, "testseq");} catch (Exception e) {;}

    // 新しいシーケンスをサーバ上に作成する
    Sequence.create(conn, "testseq", 1, 1);

    Sequence seq = new Sequence(conn, "testseq");

    // ループでシーケンス内の次の値を出力する
    for (int i = 1; i <= 10; i++) {
      System.out.println(seq.nextValue());
    }

    System.out.println(seq.currentValue());

    // シーケンスをサーバからドロップし、
    // Sequence オブジェクトを閉じる
    Sequence.drop(conn, "testseq");
    seq.close();

    // 最後に接続を閉じる
   conn.close();
  }
}
```

7 JDBC 接続のテストとトラブルシューティング

以下で、JDBC 接続のテスト、モニタ、およびトラブルシューティングの方法について説明します。

- 7-1 ページの「JDBC 接続のモニタ」
- 7-2 ページの「コマンドラインからの DBMS 接続の有効性の検証」
- 7-4 ページの「JDBC のトラブルシューティング」
- 7-8 ページの「UNIX での共有ライブラリに関連する問題のトラブルシューティング」

JDBC 接続のモニタ

Administration Console では、各サブコンポーネント (接続プール、マルチプール、および DataSource) の接続パラメータをモニタするためのテーブルと統計を表示できます。

`JDBCConnectionPoolRuntimeMBean` を使用して、接続プールの統計にプログラムでアクセスすることもできます。『WebLogic Server パートナーズ ガイド』および WebLogic の Javadoc を参照してください。この MBean は、Administration Console に統計を取り込む API と同じものです。接続のモニタの詳細については、「WebLogic Server ドメインのモニタ」および「JDBC 接続の管理」を参照してください。

MBean の使い方については、『WebLogic JMX Service プログラマーズ ガイド』を参照してください。

コマンドラインからの DBMS 接続の有効性の検証

WebLogic Server をインストールした後、`utils.dbping` BEA ユーティリティを使用して 2 層 JDBC データベース接続をテストします。`utils.dbping` ユーティリティを使用するには、JDBC ドライバのインストールを完了する必要があります。以下の作業を必ず行ってください。

- Type2 JDBC ドライバ (WebLogic jDriver for Oracle など) の場合は、`PATH` (Windows) あるいは共有ライブラリ パスまたはロード ライブラリ パス (Unix) で DBMS 提供のクライアントと BEA 提供のネイティブ ライブラリの両方を設定します。
- すべてのドライバについて、`CLASSPATH` で JDBC ドライバのクラスを設定します。
- BEA WebLogic jDriver JDBC ドライバのコンフィグレーション手順については、以下を参照してください。
  - 「WebLogic jDriver for Oracle のコンフィグレーション」
  - 「WebLogic jDriver for Microsoft SQL Server の使い方」

`utils.dbping` ユーティリティを使用すると、Java とデータベースの間で接続が可能なことを確認できます。dbping ユーティリティは、WebLogic jDriver for Oracle などの WebLogic 2 層 JDBC ドライバを使用した 2 層接続 のテストにのみ使用します。

構文

```
$ java utils.dbping DBMS user password DB
```

引数

DBMS

使用。`ORACLE` または `MSSQLSERVER4`

user

データベース ログインに使用する有効なユーザ名です。SQL Server では isql、Oracle では sqlplus で使用するものと同じ値と形式を使用します。

password

ユーザの有効なパスワード。isql、または sqlplus で使用するものと同じ値と形式を使用します。

DB

データベースの名前。形式は、データベースとバージョンに応じて異なります。isql、または sqlplus で使用するものと同じ値と形式を使用します。MSSQLServer4 などの Type 4 ドライバの場合は、環境にアクセスできないので、サーバを見つけるには補足情報が必要です。

サンプル

Oracle

sqlplus で使用する同じ値を利用し、Java から WebLogic jDriver for Oracle 経由で Oracle に接続します。

SQLNet を使用しない (かつ ORACLE_HOME と ORACLE_SID が定義されている) 場合は、次の例に従います。

```
$ java utils.dbping ORACLE scott tiger
```

SQLNet V2 を使用する場合は、次の例に従います。

```
$ java utils.dbping ORACLE scott tiger TNS_alias
```

TNS_alias は、ローカルの tnsnames.ora ファイルで定義されているエリアスです。

Microsoft SQL Server (Type 4 ドライバ)

Java から WebLogic jDriver for Microsoft SQL Server 経由で Microsoft SQL Server に接続するには、`user` と `password` で `isql` の場合と同じ値を使用します。ただし、SQL Server を指定するには、SQL Server が動作しているコンピュータの名前と SQL Server がリスンしている TCP/IP ポートを指定します。コンピュータ名が `mars` で、リスン ポートが 1433 の SQL Server にログインするには、次のように入力します。

```
$ java utils.dbping MSSQLSERVER4 sa secret mars:1433
```

1433 は Microsoft SQL Server のデフォルト ポート番号なので、この例の「`:1433`」は省略してもかまいません。デフォルトでは、Microsoft SQL Server は TCP/IP 接続をリスンしないことがあります。DBA でリスンするようにコンフィグレーションできます。

JDBC のトラブルシューティング

以降の節では、トラブルシューティングのヒントを紹介します。

JDBC 接続

WebLogic への接続をテストする場合は、WebLogic Server のログを調べてください。デフォルトでは、ログは次のフォーマットでファイルに記録されます。

`domain``\`*`server`*`\`*`server`*`.log`

ここで *`domain`* はドメインのルート フォルダ、*`server`* はサーバの名前です。サーバ名は、フォルダ名やログ ファイル名で使用されます。

Windows

`.dll` のロードが失敗したことを示すエラー メッセージが表示された場合は、`PATH` で 32 ビット データベース関連の `.dll` を指定してください。

UNIX

.so または .sl のロードが失敗したことを示すエラー メッセージが表示された場合は、LD_LIBRARY_PATH または SHLIB_PATH で 32 ビット データベース関連のファイルを指定してください。

コードセットのサポート

WebLogic では、Oracle のコードセットがサポートされています。ただし、次のことに注意してください。

- NLS_LANG 環境変数が設定されていないか、US7ASCII または WE8ISO8859-1 に設定されている場合、ドライバは常に 8859-1 で機能します。
- NLS_LANG 環境変数がデータベースで使用するコードセット以外の値に設定されている場合、Oracle Thin Driver および WebLogic jDriver for Oracle では、クライアント コードセットを使用して、データベースへの書き込みを行います。

詳細については、「WebLogic jDriver for Oracle の使い方」の「コードセットのサポート」を参照してください。

UNIX での Oracle に関わる他の問題

使用するスレッディング モデルをチェックしてください。グリーン スレッドは、OCI で使用されるカーネル スレッドと衝突します。Oracle ドライバを使用する場合は、ネイティブ スレッドを使用することをお勧めします。ネイティブ スレッドの使用を指定するには、Java を起動するときに -native フラグを追加します。

UNIX でのスレッド関連の問題

UNIX では、グリーン スレッドとネイティブ スレッドという 2 つのスレッディング モデルを利用できます。詳細については、Sun の Web サイトで提供されている Solaris 環境用の JDK を参照してください。

使用しているスレッドの種類は、THREADS_TYPE 環境変数を調べることで確認できます。この変数が設定されていない場合は、Java の bin ディレクトリにあるシェル スクリプトを調べてください。

一部の問題は、各オペレーティング システムの JVM でのスレッドの実装に関連しています。すべての JVM で、オペレーティング システム固有のスレッドの問題が等しく適切に処理されるわけではありません。以下に、スレッド関連の問題を防止するためのヒントを紹介します。

- Oracle ドライバを使用する場合は、ネイティブ スレッドを使用します。
- HP UNIX を使用する場合は、バージョン 11.x にアップグレードする。HP UX 10.20 などの旧バージョンでは JVM との互換性に問題があります。
- HP UNIX の場合、新しい JDK ではグリーン スレッド ライブラリが SHLIB_PATH に追加されません。現在の JDK では、SHLIB_PATH で定義されたパスにない限り、共有ライブラリ (.sl) を見つけることができません。SHLIB_PATH の現在の値を確認するには、コマンドラインで次のように入力します。

  ```
  $ echo $SHLIB_PATH
  ```

  set コマンドまたは setenv コマンド (どちらを使用するかはシェルによる) を使用すると、シンボル SHLIB_PATH で定義されたパスに WebLogic の共有ライブラリを追加できます。SHLIB_PATH で定義されていない場所にある共有ライブラリを認識させるには、システム管理者に連絡する必要があります。

JDBC オブジェクトを閉じる

プログラムが効率的に実行されるように、Connection、Statement、ResultSet などの JDBC オブジェクトは必ず、finally ブロックで閉じるようにしてください。次に、一般的な例を示します。

```
try {

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

Connection conn = d.connect("jdbc:weblogic:oracle:myserver",
                                  "scott", "tiger");

    Statement stmt = conn.createStatement();
    stmt.execute("select * from emp");
    ResultSet rs = stmt.getResultSet();
    // 処理を行う

    }

    catch (Exception e) {

      // あらゆる例外を適切に処理する

    }

    finally {

      try {rs.close();}
      catch (Exception rse) {}
      try {stmt.close();}
      catch (Exception sse) {}
      try {conn.close();
      catch (Exception cse) {}

    }
```

JDBC オブジェクトの破棄

次のようなやり方も避けてください (破棄される JDBC オブジェクトが作成されます)。

```
// これは実行しない
stmt.executeQuery();
rs = stmt.getResultSet();

// 代わりにこれを実行する
rs = stmt.executeQuery();
```

この例の最初の行では、失われ、すぐにガベージ コレクションされる結果セットが作成されます。

2 番目の行の動作は、どのサービス パックの WebLogic Server を実行しているのかによって異なります。7.0SP2 より前の WebLogic Server では、オリジナル オブジェクトのクローンが返されていました (この場合もガベージ コレクションの対象になります)。7.0SP2 以降の WebLogic Server ではオリジナル オブジェクトが返され、また、オブジェクトは使用されなくなるまでガベージ コレクションされません。

UNIX での共有ライブラリに関連する問題のトラブルシューティング

ネイティブの 2 層 JDBC ドライバをインストールするとき、パフォーマンス パックを使用するように WebLogic Server をコンフィグレーションするとき、または UNIX で BEA WebLogic Server を Web サーバとして設定するときには、システムで共有ライブラリまたは共有オブジェクト (WebLogic Server ソフトウェアと一緒に配布される) をインストールします。ここでは、予期される問題について説明し、それらの問題の解決策を提案します。

オペレーティング システムのローダでは、さまざまな場所でライブラリが検索されます。ローダの動作は、UNIX の種類によって異なります。以降の節では、Solaris と HP-UX について説明します。

WebLogic jDriver for Oracle

共有ライブラリは、このマニュアルで説明されている手順に従って設定してください。実際に指定するパスは、Oracle クライアントのバージョンや Oracle サーバのバージョンなどによって異なります。詳細については、「WebLogic jDriver for Oracle のコンフィグレーション」を参照してください。

Solaris

どのダイナミック ライブラリが実行ファイルによって使用されているのかを確認するには、ldd コマンドを実行します。このコマンドの出力が、ライブラリが見つからないことを示している場合は、次のようにして、ライブラリの位置を LD_LIBRARY_PATH 環境変数に追加します (C シェルまたは Bash シェルの場合)。

```
# setenv LD_LIBRARY_PATH weblogic_directory/lib/solaris/oci817_8
```

このようにして追加すれば、ld を実行してもライブラリの紛失は報告されないはずです。

HP-UX

不適切なファイル パーミッションの設定

HP-UX システムで WebLogic Server をインストールした後、発生する可能性が最も高い共有ライブラリの問題は、不適切なファイル パーミッションの設定です。WebLogic Server をインストールした後は、chmod コマンドを使用して共有ライブラリのパーミッションを適切に設定してください。HP-UX 11.0 で適切なパーミッションを設定するには、次のように入力します。

```
% cd WL_HOME/lib/hpux11/oci817_8
% chmod 755 *.sl
```

ファイル パーミッションを設定した後に共有ライブラリをロードできない場合は、ライブラリの位置を特定することに問題があることが考えられます。その場合はまず、次のようにして、*WL_HOME*/server/lib/hpux11 が SHLIB_PATH 環境変数に設定されていることを確認してください。

```
% echo $SHLIB_PATH
```

そのディレクトリがない場合は、次のようにして追加してください。

```
# setenv SHLIB_PATH WL_HOME/server/lib/hpux11:$SHLIB_PATH
```

あるいは、WebLogic Server 配布キットにある .sl ファイルを SHLIB_PATH 変数で既に設定されているディレクトリへコピー (またはリンク) してください。

それでも問題が解決しない場合は、chatr コマンドを使用して、アプリケーションが SHLIB_PATH 環境変数のディレクトリを検索するように指定してください。+s enabled オプションを使用すると、SHLIB_PATH 変数を検索するようにアプリケーションが設定されます。次に、このコマンドの例を示します。この例は、HP-UX 11.0 の WebLogic jDriver for Oracle 共有ライブラリで実行します。

```
# cd weblogic_directory/lib/hpux11
# chatr +s enable libweblogicoci39.sl
```

このコマンドの詳細については、chatr のマニュアル ページを参照してください。

不適切な SHLIB_PATH

Oracle 9 を使用している場合、SHLIB_PATH に適切なパスが含まれていないことが原因で共有ライブラリの問題が発生する場合もあります。SHLIB_PATH には、ドライバ (oci901_8) へのパスと、ベンダ提供のライブラリ (lib32) へのパスが含まれている必要があります。たとえば、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
```

パスに Oracle 8.1.7 ライブラリを含めることはできません。含まれているとクラッシュします。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。