BEA WebLogic Server™

BEA WebLogic Express™

WebLogic JDBC
プログラミング ガイド

BEA WebLogic Server バージョン 6.1
マニュアルの日付：2003 年 4 月 22 日

著作権

限定的権利条項

商標または登録商標

WebLogic JDBC プログラミング 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	2003 年 4 月 22 日	BEA WebLogic Server 6.1

目次

このマニュアルの内容

1. WebLogic JDBC の概要

2. WebLogic JDBC の管理とコンフィグレーション

3. JDBC アプリケーションのパフォーマンス チューニング

4. WebLogic JDBC 機能のコンフィグレーション

5. WebLogic 多層 JDBC ドライバの使い方

6. WebLogic Server でのサードパーティ ドライバの使い方

7. dbKona の使い方

8. JDBC 接続のテストとトラブルシューティング

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 サービス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 の管理とコンフィグレーション」では、WebLogic Server での JDBC の管理方法と Administration Console 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 JDBC 機能のコンフィグレーション」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 5 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 6 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 7 章「dbKona の使い方」では、Java アプリケーションとの高レベルなデータベース接続を提供する dbKona クラスについて説明します。
- 第 8 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のメイン トピックを一度に 1 つずつ印刷できます。

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

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

関連情報

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

サポート情報

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

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

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

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

表記規則

このマニュアルでは、全体を通して以下の表記規則が使用されています。

表記法	適用
〔Ctrl〕+ 〔Tab〕	同時に押すキーを示す。
斜体	強調または本のタイトルを示す。

等幅テキスト	コード サンプル、コマンドとそのオプション、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
{ }	構文内の複数の選択肢を示す。

表記法	適用
[ ]	構文内の任意指定の項目を示す。 例: `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 について概説します。

JDBC の概要

JDBC は、SQL 文を実行するための Java API です。この API は、Java プログラミング言語で記述された一連のクラスとインタフェースから構成されます。JDBC はツール / データベース開発者向けの標準 API であり、pure-Java API を使用したデータベース アプリケーションの作成を可能にします。

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 JMS や 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 ドライバの概要

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。BEA は、JDBC API 仕様を使ったさまざまなデータベース アクセス用オプションを提供しています。これらのオプションには、Oracle、Microsoft SQL Server、および Informix データベース管理システム（DBMS）用 WebLogic jDriver などの 2 層 JDBC ドライバと、クライアント アプリケーションと DBMS との仲介役として WebLogic Server で使用される多層ドライバがあります。

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 ― Java アプリケーションとデータベース間に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、WebLogic jDriver for Informix、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。

中間層アーキテクチャを採用すると、データベース リソースを WebLogic Server で集中管理できます。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。

ドライバの一覧表

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 （非 XA）	Type 2 （ネイティブ .dll） ■ WebLogic jDriver for Oracle ■ サードパーティ ドライバ Type 4 （オール Java） ■ WebLogic jDriver for Informix および 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 DB2 Informix JDBC	WebLogic Server と DBMS 間	『WebLogic JDBC プログラミング 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 のインストールと使い方』 『WebLogic jDriver for Informix のインストールと使い方』 『WebLogic jDriver for Microsoft SQL Server のインストールと使い方』
2 層 （XA）	Type 2 （ネイティブ .dll） ■ 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」

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 ■ RMI ドライバ ■ Pool ドライバ ■ JTS	クライアントと WebLogic Server 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 プログラミング ガイド』（このマニュアル）

JDBC ドライバの説明

以下の節では、表 1-1 「JDBC ドライバ」に示した JDBC ドライバについて詳しく説明します。

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 Oracle 用 Type 2 JDBC ドライバである WebLogic jDriver for Oracle は、WebLogic Server 配布キットに同梱されています。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 のインストールと使い方』を参照してください。

WebLogic jDriver for Microsoft SQL Server

WebLogic Server 6.1 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 のインストールと使い方』を参照してください。

WebLogic jDriver for Informix

WebLogic Server 6.1 配布キットに同梱の BEA WebLogic jDriver for Informix は、Informix DBMS 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 Informix のインストールと使い方』を参照してください。

WebLogic Server JDBC 多層ドライバ

以下の節では、クライアントへのデータベース アクセスを提供する WebLogic 多層 JDBC ドライバについて説明します。これらのドライバの詳細については、『WebLogic JDBC プログラミング ガイド』の「WebLogic 多層 JDBC ドライバの使い方」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 RMI ドライバ

WebLogic RMI ドライバは、多層 Type 3 JDBC（Java Data Base Connectivity）ドライバです。このドライバは WebLogic Server で実行され、任意の 2 層 JDBC ドライバと一緒に使用することでデータベース アクセスを提供します。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 JTS ドライバ

WebLogic JTS ドライバは 多層 Type 3 JDBC ドライバで、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に使用されます。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。

サードパーティ ドライバ

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
- EJB へのアクセス（JDBC でのトランザクション呼び出しの実装）

また、WebLogic Server 多層ドライバは JDBC API、および非標準 JDBC 呼び出しを越える機能を提供するサードパーティ ドライバだけをサポートします。

Cloudscape

Cloudscape の pure-Java DBMS は、WebLogic Server 配布キットに付属しています。Cloudscape DBMS にアクセスするための JDBC ドライバも付属しています。この DBMS は、配布キットに付属のコード サンプルで頻繁に使用されます。他の DBMS を使用できない場合、この DBMS をテストおよび開発用に使用できます。ただし、この評価版を使って格納できるデータの量には制限があります。

詳細については、「WebLogic Server で Cloudscape Database を使用する」を参照してください。

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、無償で提供されています。WebLogic Server でのこのドライバの使い方の詳細については、6-1 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

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。WebLogic Server でのこのドライバの使い方の詳細については、6-1 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

多層ドライバは WebLogic Server を使用して、DBMS への迅速な接続を可能にする接続プールにアクセスします。接続プールを起動したときにはデータベース接続は既に確立されているので、データベース接続を確立するオーバーヘッドが削減されます。接続プールは、WebLogic Pool ドライバを使ってサーバサイド アプリケーション（HTTP サーブレットや EJB など）から使用するか、WebLogic RMI ドライバを使ってスタンドアロン Java クライアントから使用します。この節では、接続プールの概要について説明します。詳細については、4-1 ページの「接続プールの使い方」を参照してください。

接続プールを使って WebLogic Server から DBMS に接続するには、2 層 JDBC ドライバが必要となります。この 2 層 ドライバは、WebLogic jDriver の 1 つか、またはサードパーティ JDBC ドライバ（Sybase jConnect ドライバなど）です。これらのドライバは、WebLogic 配布キットに同梱されています。次の表に、接続プールを使用するメリットを示します。

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 コード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 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 の場所を知っている必要も、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。

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

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データベースにアクセスするには、WebLogic Pool ドライバを使用します。2 フェーズ コミット トランザクションの場合は、WebLogic Server JDBC/XA ドライバの 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、JTS ドライバを使用します。また、Java Naming and Directory Interface （JNDI）と DataSource オブジェクトを使用して接続プールにアクセスすることもできます。

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 ドライバには、Java 2 Enterprise Edition（J2EE）仕様を使って標準ベースのアプローチを提供するというメリットがあります。新しいデプロイメントでは、RMI ドライバを使うことをお勧めします。t3 クライアント サービスは、このリリースでは非推奨になっています。

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 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 サポートを有効または無効にできます。

マルチプールの概要

JDBC マルチプールでは、データベースの接続性を向上させるために、高可用性またはロード バランシング アルゴリズムを選択できます。マルチプールは「プールのプール」で、接続を提供するためのプールをリストから選択するためのコンフィグレーション可能なアルゴリズムを備えています。詳細については、4-17 ページの「マルチプールの使い方」を参照してください。

マルチプール アルゴリズムの選択

マルチプールは、以下のいずれかの方法で設定できます。

- 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。

クラスタ化された JDBC の概要

WebLogic Server では、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、クラスタで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、クラスタ内の管理対象サーバごとに存在していなければなりません。また、JDBC オブジェクトをコンフィグレーションするときは、クラスタを対象にして行います。

クラスタにおける JDBC オブジェクトの詳細については、『WebLogic Server クラスタ ユーザーズ ガイド』の「JDBC 接続」を参照してください。

DataSource の概要

DataSource オブジェクトを使用すると、JDBC クライアントは DBMS 接続を取得できるようになります。DataSource は、クライアント プログラム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、接続プールまた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、4-21 ページの「DataSource のコンフィグレーションと使用方法」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 を使用します。この API を使用すると、データ ソースへの接続を確立し、クエリを送信し、その結果を処理するのに必要なクラス オブジェクトを作成できます。

WebLogic JDBC インタフェースの定義

次の表に、WebLogic Server でよく使用される JDBC インタフェースを示します。すべての JDBC インタフェースの詳細については、`java.sql` Javadoc または `weblogic.jdbc` Javadoc を参照してください。

JDBC インタフェース	説明
Driver	ドライバとデータベース間の接続を設定し、ドライバに関する情報またはデータベースへの接続に関する情報を提供する。各ドライバ クラスが実装しなければならないインタフェース。
DataSource	特定の DBMS または他のデータ ソースを表す。データ ソースへの接続を確立するために使用される。
Statement	単純な SQL 文をパラメータなしでデータソースに送信する。
PreparedStatement	Statement から継承される。コンパイル済みの SQL 文を（IN パラメータを指定するか、または指定せずに）実行するために使用される。
CallableStatement	Statement から継承される。データベース ストアド プロシージャへの呼び出しを実行するために使用される。OUT パラメータを扱うメソッドを追加する。
ResultSet	SQL クエリの実行結果が格納される。クエリの条件を満たす列が格納される。
ResultSetMetaData	ResultSet オブジェクト内のカラムのタイプとプロパティに関する情報を提供する。
DataBaseMetaData	データベース全般に関する情報を提供する。単一の値または結果セットを返す。
Clob	Character Large Object をカラム値としてデータベース テーブルの行に格納する組み込みタイプ。
Blob	Binary Large Object をカラム値としてデータベース テーブルの行に格納する組み込みタイプ。

WebLogic jDriver for Oracle を使用するときのこれらのインタフェースの詳細については、『WebLogic jDriver for Oracle のインストールと使い方』を参照してください。

JDBC 2.0

WebLogic Server は、JDBC 2.0 をサポートしている JDK 1.3 を使用します。

制限

次の制限に注意してください。

- RMI ドライバを WebLogic jDriver for Oracle またはサードパーティの 2 層ドライバと組み合わせて使用している場合、`callableStatement` または `preparedStatement` SQL 文ではバッチ更新（`addBatch()`）は使用できません。

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

2 WebLogic JDBC の管理とコンフィグレーション

この章では、BEA WebLogic Server に関連する JDBC 管理タスクの概要について説明します。

- 2-2 ページの「JDBC のコンフィグレーション」
- 2-3 ページの「JDBC 接続のモニタ」

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。
- 4-1 ページの「WebLogic JDBC 機能のコンフィグレーション」。JDBC API を使用して接続をコンフィグレーションする方法について説明します。

JDBC のコンフィグレーション

Administration Console は、JDBC などの WebLogic Server の機能を有効化、コンフィグレーション、およびモニタするためのインタフェースを備えています。Administration Console を起動するには、『管理者ガイド』の「WebLogic Server とクラスタのコンフィグレーション」に示された手順を参照してください。属性では、以下のような JDBC 環境を定義します。

- 接続プール
- マルチプール
- DataSource

接続プールのコンフィグレーション

Administration Console を使用して接続プールをコンフィグレーションします。このコンフィグレーションには、属性と接続パラメータの定義、プールの複製、接続プールのサーバまたはドメインへの割り当てなどが含まれます。接続プールの詳細については、4-1 ページの「接続プールの使い方」を参照してください。データベース接続のコンフィグレーションについては、『管理者ガイド』の「JDBC 接続の管理」を参照してください。

マルチプールのコンフィグレーション

Administration Console でマルチプールを定義（名前を指定）したら、特定のマルチプールに入れる定義済みの接続プールを指定します。特定の接続プール内のすべての接続は同じです。つまり、それらは同じユーザ、パスワード、および接続プロパティを持つ 1 つのデータベースに関連付けられます。しかし、マルチプールを使用する場合、マルチプール内の接続プールには異なる DBMS を関連付けることができます。また、ロード バランシングまたは高可用性のいずれかのアルゴリズム動作を選択することによって、検索方法を指定できます。

マルチプールの使い方の詳細については、4-17 ページの「マルチプールの使い方」を参照してください。データベース接続のコンフィグレーションについては、『管理者ガイド』の「JDBC 接続の管理」を参照してください。

DataSource および TxDataSource のコンフィグレーション

接続プールやマルチプールと同じように、Administration Console で DataSource オブジェクトを作成します。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 (TxDataSource) または無効 (DataSource) にできます。DataSource 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。ローカルおよび分散トランザクションの DataSource オブジェクトについては、『管理者ガイド』の「JDBC 接続の管理」および 4-1 ページの「WebLogic JDBC 機能のコンフィグレーション」を参照してください。

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、DataSource、および TxDataSource）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、『管理者ガイド』の「WebLogic 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。

MBeans 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。

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。

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。

データのキャッシュ

DBMS のアクセスでは大量にリソースを消費します。プログラムが頻繁にアクセスするデータがアプリケーション間で共有されるか、または接続間で持続する場合、次の機能を使用してデータをキャッシュできます。

- 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

DBMS アプリケーションのパフォーマンスの向上または低下は、アプリケーション言語ではなくアプリケーションの設計方法によってほぼ決まります。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。

セットベース方式を理解すると、非常に役に立ちます。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。これは妥当な質問ですが、答えは簡単ではありません。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、元のクエリが返した行の数が返されます。該当するデータを変更する他の DBMS の動作があれば、クエリが実行されたときの実際のカウントは変わる場合があります。

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。

アプリケーションは、実際に必要なデータに限定したクエリを使用する必要があります。その方法としては、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 文によって返されたテーブル全体を示します。

```
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
May	80

次に、相関サブクエリを示します。

```
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便利な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いけません。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（一部の DBMS ではテーブルの設定によってこれらが自動的に生成されます）。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。

このテクニックの詳細については、使用している DBMS のドキュメントを参照してください。

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。この場合、テーブルに必要なスペースは大きくなりますが、処理は高速になります。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 JDBC 機能のコンフィグレーション

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。

- 4-1 ページの「接続プールの使い方」
- 4-17 ページの「マルチプールの使い方」
- 4-21 ページの「DataSource のコンフィグレーションと使用方法」

接続プールの使い方

接続プールとは、接続プールが登録されるとき（通常は WebLogic Server の起動時）に作成される、データベースへの同一 JDBC 接続のグループに名前を付けたものです。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、数多くの性能上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ごと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 ヘルプで定義されています。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。4-8 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。

接続プールのフェイルオーバに関する要件

WebLogic Server は、アプリケーションによる使用中に障害が発生した接続をフェイルオーバすることはできません。接続を使用している間の障害に対しては、トランザクションを実行し直し、このような障害を処理するためのコードを用意する必要があります。

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 で属性を設定します。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。

次に、接続プールの属性および説明のリストを示します。詳細については、『管理者ガイド』の「JDBC 接続の管理」と『Administration Console オンライン ヘルプ』を参照してください。

接続プールの属性

[名前]

（必須）接続プールの名前。この名前を使用して、このプールから JDBC 接続プールにアクセスします。

[URL]

（必須）WebLogic Server と DBMS との間の接続に使用する JDBC 2 層ドライバの URL。WebLogic jDriver のいずれか、または 2 層接続環境でテスト済みの別の JDBC ドライバでもかまいません。URL については、使用する JDBC ドライバのマニュアルを参照してください。

[ドライバ クラス名]

（必須）WebLogic Server と DBMS との間の接続に使用する JDBC 2 層ドライバの完全なパッケージ名。絶対パス名については、使用する JDBC ドライバのマニュアルを参照してください。

このプロパティは、使用する 2 層 JDBC ドライバによって定義および処理されます。DBMS への接続に必要なプロパティについては、使用する JDBC ドライバのマニュアルを参照してください。

[プロパティ]

（必須）ユーザ名、サーバおよび XA 接続のためのオープン文字列など、データベースに接続するためのプロパティ。データベース パスワードについては、**[パスワード]** プロパティを使用します。オープン文字列内のパスワードについては、**[文字列のオープン パスワード]** 属性を使用します。

このプロパティは、使用する 2 層 JDBC ドライバによって定義および処理されます。DBMS への接続に必要なプロパティについては、使用する JDBC ドライバのマニュアルを参照してください。

[パスワード]

（省略可能）物理的なデータベース接続作成時に、2 層 JDBC ドライバに渡されるデータベース パスワード。この値は、**[プロパティ]** 属性で（名前 / 値のペアとして）定義されているデータベース パスワードがあっても、それを上書きします。この値は、config.xml ファイルに暗号化された形で格納されます。

[文字列のオープン パスワード]

（省略可能）XA 物理データベース接続の作成のためのオープン文字列内で使われるパスワード。この値は、**[プロパティ]** 属性で定義されているオープン文字列内のパスワードを上書きします。この値は、config.xml ファイルに暗号化された形で格納されます。

[ログイン遅延時間]

（省略可能）データベースへの接続を開くための試行の間隔（秒）。データベースによっては、複数の接続リクエストが短い間隔で繰り返されると処理できないものもあります。このプロパティを使用すると、データベース サーバの処理が追いつくように、少しの間隔をあけることができます。

[初期容量]

（省略可能）プールの初期サイズ。この値が設定されていない場合、デフォルト値は **[増加容量]** に設定されている値になります。

[最大容量]

（必須）プールの最大サイズ。

[増加容量]

プールの容量を増加するサイズ。**[初期容量]** および **[増加容量]** は、Java Vector のように動作し、初期割り当て（「capacity」）があり、プールの **[最大容量]** まで、必要に応じて capacityIncrement ずつ増加されます。デフォルトは 1 です。

[縮小可]

（省略可能）接続プールが需要に見合うよう増加された後、初期のサイズに戻すかどうかを設定します。このプロパティが `true` の場合、**[縮小間隔]** を設定します。設定しない場合は、デフォルトで 15 分になります。**[縮小可]** は下位互換のため、デフォルトで `false` に設定されます。

[縮小間隔]

（省略可能）接続プールが需要に見合うよう増加された後、初期のサイズに戻すまでの分数。縮小間隔のデフォルト値は 15 分で、最小値は 1 分です。

注意 : [縮小可] が false に設定されているときにこの属性の値を設定すると、WebLogic Server は false 設定を無視し、[縮小間隔] の値に従って縮小を許可します。

[テスト テーブル名]

（**[更新間隔]**、**[リザーブされたときに接続をテスト]**、または **[リリースされたときに接続をテスト]** を設定する場合にのみ、必須）接続プール内の接続の実行可能性をテストするために使用するデータベース テーブルの名前。クエリ「`select count(*) from` **テスト テーブル名**」がテストに使用されます。**テスト テーブル名**は、必ず実在し、その接続のデータベース ユーザがアクセスできるものでなければなりません。ほと

んどのデータベース サーバはこの SQL を最適化して、テーブル スキャンを回避します。それでも、[**テスト テーブル名**] を、行が少ない（またはまったくない）テーブルの名前に設定することは有益です。

[更新間隔]

（省略可能）このプロパティは、[**テスト テーブル名**] プロパティと連動して、プールの接続の自動リフレッシュを有効にします。接続プールの各未使用接続は、指定された間隔で簡単なクエリを実行することによってテストされます。テストが失敗すると、接続のリソースは破棄され、それに代わって新しい接続が作成されます。デフォルトは 1 です。
自動リフレッシュを有効にするには、[**更新間隔**] を接続テストの周期の分数に設定します。最小値は 1 です。無効な [**更新間隔**] 値を設定した場合、値はデフォルトで 5 分になります。既存のデータベース テーブルをテストに使用するには、[**テスト テーブル名**] を既存のデータベース テーブルの名前に設定します（必須）。

[リザーブされたときに接続をテスト]

（省略可能）`true` に設定すると、WebLogic Server は、プールから削除した後に接続のテストを実行してから、クライアントに渡します。テストを実行すると、クライアントのリクエストに応じてプールから接続を提供するまでに若干時間が余分にかかりますが、クライアントは必ず有効な接続を受け取ることができます（DBMS が使用可能またはアクセス可能であることが前提）。この機能を使用するには、[テスト テーブル名] パラメータを設定する必要があります。

高可用性アルゴリズムと共にマルチプールで接続プールを使用するときは、この属性を `true` に設定し、リスト中の次の接続プールにいつフェイルオーバするか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。4-19 ページの「マルチプールのフェイルオーバに関する制限と要件」を参照してください。

[リリースされたときに接続をテスト]

（省略可能）`true` に設定すると、WebLogic Server は、接続プールに戻す前に接続をテストします。プール内のすべての接続がすでに使用されており、クライアントが接続を待機している場合、クライアントの待機時間は接続がテストされている間だけ若干長くなります。この機能を使用するには、[テスト テーブル名] パラメータを設定する必要があります。

パーミッション

Administration Console で、動的接続プールの作成に対するパーミッションを設定します。動的接続プールを作成するときに、ACL がその接続プールに関連付けられます。ACL と接続プールは同じ名前である必要はなく、複数の接続プールが 1 つの ACL を使用することができます。ACL を指定しない場合は「system」ユーザがプールのデフォルトの管理ユーザとなり、またどのユーザもプールから提供される接続を使用できます。

接続プールに対して ACL を定義する場合、アクセスは ACL の定義内容に厳密に制限されます。たとえば、`fileRealm.properties` ファイルで接続プールの ACL を定義するまでは、ドメイン内のすべての接続プールに誰もが無制限にアクセスできます。一方、ファイルに次の行を追加すると、アクセスは非常に厳しく制限されます。

```
acl.reset.weblogic.jdbc.connectionPool=Administrators
```

この行では、すべての接続プールを対象として Administrators にリセット権限を付与し、*それ以外のすべてのユーザによるその他すべてのアクションを禁止します*。ACL を追加することにより、接続プールに対してファイル レルム保護が有効になります。WebLogic Server は `fileRealm.properties` に定義された ACL を適用し、ファイル内で明示的に権限を付与されたアクセスだけを許可します。ACL 追加の目的が、接続プールだけを対象としてリセットを制限することであった場合、その他のアクションを実行するための権限をすべてのユーザ、あるいは特定のロールまたはユーザに付与しなければなりません。次に例を示します。

```
acl.reserve.weblogic.jdbc.connectionPool=everyone
acl.shrink.weblogic.jdbc.connectionPool=everyone
acl.admin.weblogic.jdbc.connectionPool=everyone
```

表 4-1 は、接続プールのセキュリティを保護するために `fileRealm.properties` で使用できる ACL の一覧です。

表 4-1 JDBC のファイル レルム ACL

使用する ACL	制限するアクション
`reserve.weblogic.jdbc.connectionPool[.poolname]`	接続プール内の接続の予約

表 4-1 JDBC のファイル レルム ACL

使用する ACL	制限するアクション
`reset.weblogic.jdbc.connectionPool[.poolname]`	シャットダウンして割り当て済みのすべての接続を再確立することによる、接続プール内のすべての接続のリセット
`shrink.weblogic.jdbc.connectionPool[.poolname]`	元のサイズ（接続数）への接続プールの縮小
`admin.weblogic.jdbc.connectionPool[.poolname]`	接続プールの有効化、無効化、シャットダウン
`admin.weblogic.jdbc.connectionPoolcreate`	接続プールの作成

ACL の変更方法については、『管理者ガイド』の「セキュリティの管理」にある「ACL の定義」を参照してください。

接続プールについての制限事項

接続プールを使うとき、DBMS 固有の SQL コードを実行して、データベース接続のプロパティを、WebLogic Server や JDBC ドライバが認識できない値に変更することができます。このような接続を接続プールに返しても、接続の特性が有効な状態に戻らないことがあります。たとえば、Sybase DBMS では、`set rowcount 3 select * from y` のような文を実行すると、その接続は最大 3 行しか返さなくなります。接続プールに返された接続をクライアントが再使用すると、選択したテーブルが 500 行であっても、クライアントは依然として 3 行しか取得できません。ほとんどの場合、同じ結果を得ることができて、WebLogic Server または JDBC ドライバが接続をリセットするような、標準の (DBMS 固有ではない) SQL コードが存在します。先の例では、`set rowcount` の代わりに `setMaxRows()` を使用できます。

DBMS 固有の SQL コードを使って接続を変更する場合は、接続プールに返す前に、接続を許容できる状態に戻す必要があります。

接続プールの動的作成

JNDI ベースの API を使用すると、Java アプリケーションの内部から接続プールを作成できます。この API を使用すると、すでに実行されている WebLogic Server に接続プールを作成できます。接続プールへ動的にアクセスするには、JTS または Pool ドライバが必要です。

動的プールは一時的に無効にできます。無効にすると、プール内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。無効にしたプールを有効にした場合、各接続の状態はプールを無効にしたときと同じなので、クライアントは中断されたところからデータベース操作を継続できます。

プロパティ

接続プールの特定のプロパティを定義するには、キーの綴りと大文字 / 小文字を正確に指定していることを確認してください。プールを作成するときに使用する `java.utils.Properties` オブジェクトで、以下の表のように、これらのタイプ（キー）とその値をペアにします。

表 4-2 接続プールのプロパティ

プロパティの種類	説明	サンプルのプロパティ値
`poolName`	必須。プールのユニーク名。	`myPool`
`aclName`	必須。サーバの config ディレクトリにある `fileRealm.properties` 内の異なるアクセス リストを識別する。ペアになる名前は dynaPool でなければならない。	`dynaPool`
`props`	データベース接続プロパティ。通常、この形式は「データベース ログイン名 ; サーバ ネットワーク ID」。	`user=scott; server=ora817`

表 4-2 接続プールのプロパティ（続き）

プロパティの種類	説明	サンプルのプロパティ値
password	省略可能。物理的なデータベース接続作成時に、2 層 JDBC ドライバに渡されるデータベース パスワード。この値は、props で（名前 / 値のペアとして）定義されているデータベース パスワードを上書きする。 この値は、config.xml ファイルに暗号化された形で格納される。	tiger
xapassword	省略可能。XA 物理データベース接続の作成時にオープン文字列内で使用されるパスワード。props で定義されているオープン文字列内のパスワードを上書きする。 この値は、config.xml ファイルに暗号化された形で格納される。	secret
initialCapacity	プール内の接続の初期数。このプロパティが定義済みで、0 より大きい正の数である場合、WebLogic Server は起動時にこの数の接続を作成する。デフォルトは 0。maxCapacity より大きい値は指定できない。	1
maxCapacity	プールで許可される接続の最大数。デフォルトは 1。定義する場合、maxCapacity は =>1 でなければならない。	10
capacityIncrement	一度に追加できる接続の数。デフォルトは 0。	1

表 4-2 接続プールのプロパティ（続き）

プロパティの種類	説明	サンプルのプロパティ値
allowShrinking	接続が使用中でないことが検出されたときに、プールを縮小できるかどうかを指定する。デフォルトは true。	True
shrinkPeriodMins	縮小の間隔。allowShrinking = True の場合、デフォルトは 15 分。	5
driver	必須。JDBC ドライバの名前。ローカル（非 XA）ドライバのみ参加できる。	weblogic.jdbc.oci.Driver
url	必須。JDBC ドライバの URL。	jdbc:weblogic:oracle
testConnectionsOnReserve	予約される接続をテストすること示す。デフォルトは False。	true
testConnectionsOnRelease	解放されるときに接続をテストすることを示す。デフォルトは False。	true
testTableName	接続をテストするときに使用されるデータベース名。テストを正常に行うには、指定されている必要がある。testConnectionsOnReserve、testConnectionsOnRelease、または refreshPeriod が定義されている場合は必須。	myTestTable
refreshPeriod	接続をテストする間隔。	1
loginDelaySecs	ログインを試行する間隔の秒数。デフォルトは 0。	1

動的接続プールのサンプル コード

次のサンプル コードでは、プログラムで接続プールを割り当てる方法を示します。

注意： 以下のサンプル コードはクラスタ環境では使用できません。そこで、『Administration Console オンライン ヘルプ』で説明しているとおりに Administration Console で接続プールとデータ ソースを作成し、それらの接続プールとデータ ソースのターゲットをクラスタにすれば、その問題を回避することができます。

パッケージをインポートする

以下のパッケージをインポートします。

```
import java.util.Properties
import weblogic.common.*;
import weblogic.jdbc.common.JdbcServices;
import weblogic.jdbc.common.Pool;
```

JNDI を使用して JdbcServices オブジェクトを取得する

オブジェクト参照を使用すると、動的プールの作成に必要なすべてのメソッドにアクセスできます。最初に、初期 JNDI コンテキストを WebLogic JNDI プロバイダへ取得します。次に、「`weblogic.jdbc.common.JdbcServices`」をルックアップします。

```
Hashtable env = new Hashtable();

env.put(Context.INITIAL_CONTEXT_FACTORY,
        "weblogic.jndi.WLInitialContextFactory");
// WebLogic Server の URL
env.put(Context.PROVIDER_URL, "t3://localhost:7001");
env.put(Context.SECURITY_PRINCIPAL, "Fred");
env.put(Context.SECURITY_CREDENTIALS, "secret");

Context ctx = new InitialContext(env);

// weblogic.jdbc.JdbcServices をルックアップ
weblogic.jdbc.common.JdbcServices jdbc =
 (weblogic.jdbc.common.JdbcServices)
   ctx.lookup("weblogic.jdbc.JdbcServices");
```

プロパティを設定する

プールの属性を定義する java.utils.properties オブジェクトを設定します。4-8 ページの「プロパティ」の表 4-2「接続プールのプロパティ」を参照してください。

`weblogic.jdbc.JdbcServices` をロードしたら、`weblogic.jdbc.common.JdbcServices.createPool()` メソッドに、プールを記述する Properties オブジェクトを渡します。Properties オブジェクトには、Administration Console で接続プールを作成するために使うのと同じプロパティが含まれています。ただし、「`aclName`」プロパティは、動的接続プールに固有のものです。

以下の例では、Oracle データベース DEMO 用に「eng2」という接続プールを作成しています。これらの接続は、「tiger」というパスワードを持つユーザ「SCOTT」としてデータベースにログインします。プールが作成されると、データベース接続が 1 つ開きます。このプールには、接続を最大 10 個作成することができます。「aclName」プロパティには、この接続プールでは「dynapool」が使用されることが指定されています。

```
String thePoolName = "eng2";
Properties poolProps = null;
Pool myPool = null;
weblogic.jdbc.common.Pool pool = null

poolProps = new Properties();

 // 接続プールのプロパティを設定する
poolProps.put("poolName", thePoolName);
poolProps.put("url",              "jdbc:weblogic:oracle");
poolProps.put("driver",           "weblogic.jdbc.oci.Driver");
poolProps.put("props", "user=scott;password=tiger;server=demo");
poolProps.put("password", "tiger");
poolProps.put("initialCapacity", "1");
poolProps.put("maxCapacity",     "10");
poolProps.put("capacityIncrement", "1");
poolProps.put("aclName", "weblogic.jdbc.connectionPool.dynapool");
poolProps.put("allowShrinking", "true");
poolProps.put("shrinkPeriodMins", "5");
poolProps.put("refreshPeriod", "10");
poolProps.put("testConnectionsOnReserve", "true");
poolProps.put("testConnectionsOnRelease", "false");
poolProps.put("testTableName", "dual");
poolProps.put("loginDelaySecs", "1");
```

接続プールを作成する

新しく定義された Properties オブジェクトを、JNDI から事前に取得した JdbcServices オブジェクトに渡すことにより、プールを作成します。新しいプールの名前が既存のプールと同じ場合など、プールの作成に問題がある場合は、例外が送出されます。

```
// プールを作成する
        try {
            myJdbc.createPool(poolProps);
        } catch (Exception e) {
            System.out.println(thePoolName
                      + " can't be created ..");
            System.exit(666);
        }
```

プール ハンドルを取得する

新しく作成したプールからプール ハンドルを取得します。プール ハンドルを使用して、アプリケーションの処理中にプールを操作します。

```
        weblogic.jdbc.common.Pool myPool = null;

        // プールを取得。プールで何らかの処理を行う ...
        try {
           theNewPool = myJdbc.getPool(thePoolName);
       } catch (Exception e) {
          System.out.println("Cannot retrieve pool: "
                        +  thePoolName);
           System.exit(666);
        }
```

接続プールの管理

`weblogic.jdbc.common.Pool` インタフェースと `weblogic.jdbc.common.JdbcServices` インタフェースは、接続プールを管理し、それらに関する情報を取得するためのメソッドを提供します。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。

プールに関する情報の取得

`weblogic.jdbc.common.JdbcServices.poolExists()`

`weblogic.jdbc.common.Pool.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、動的接続プールがすで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

`weblogic.jdbc.common.Pool.disableDroppingUsers()`

`weblogic.jdbc.common.Pool.disableFreezingUsers()`

`weblogic.jdbc.common.pool.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

weblogic.jdbc.common.Pool.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 上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

weblogic.jdbc.common.Pool.shutdownSoft()

weblogic.jdbc.common.Pool.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

weblogic.jdbc.common.Pool.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。

weblogic.jdbc.common.Pool.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 を使用します。
- weblogic.Admin コマンドを（管理特権を持ったユーザとして）使用して、管理者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。ほかにも、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・アプリケーションで、JdbcServicesDef インターフェイスに属する reset() メソッドを使用します。

  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。

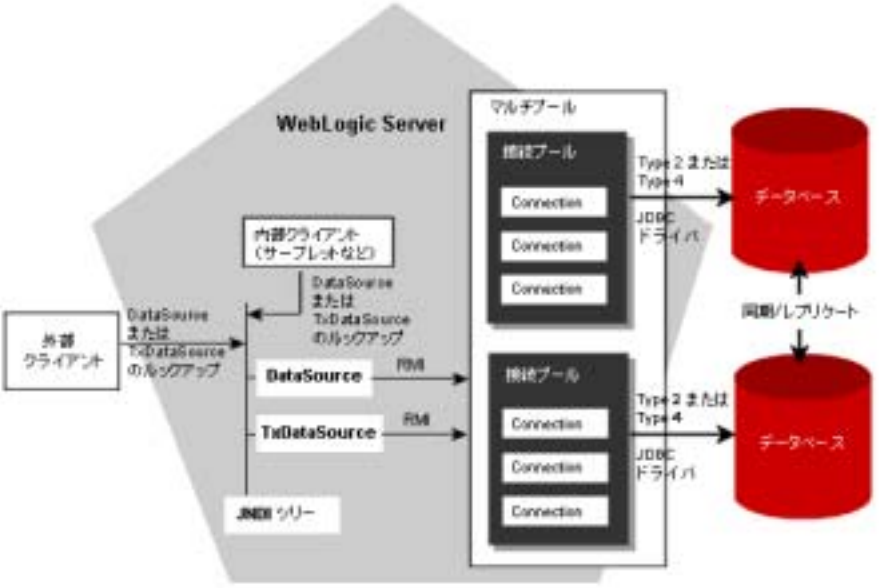
c. catch ブロックの中で `reset()` メソッドを呼び出します。

マルチプールの使い方

マルチプールは「プールのプール」です。マルチプールには、アプリケーションに接続を返す接続プールを決定するためのコンフィグレーション可能なアルゴリズムとして、高可用性と接続プール ロードバランシングが含まれています。

マルチプールと接続プールは、次のような点が異なります。つまり、特定の「接続プール」に含まれる接続はすべて、同じデータベース、同じユーザ、同じ接続属性を使って、まったく等しく作成されます。一方、「マルチプール」内の接続プールは、異なるユーザまたは DBMS と関連付けることができます。

図 4-1 マルチプールのアーキテクチャ



マルチプールは、複数のデータベースからの接続、またはユーザが異なる複数の接続を返すことができますが、WebLogic Server には異なるデータベースの内容を統合または処理するための手段が用意されていないことに注意してください。アプリケーションまたは DBMS 環境で同期またはデータ統合の処理を行って、基になっているどの接続プールから接続を受け取ったときでも、アプリケーションが透過的に正しく動作するようにしなければなりません。

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。最初のプールがデータベースへの接続を失うと、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は `TestConnsOnReserve=true` を設定し、いつリスト内の次の接続プールにフェイルオーバするか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。

ある接続プール内のすべての接続が使用されている場合、高可用性アルゴリズムを使用するマルチプールは、リストの次のプールから接続を提供しません。これは、接続プールの容量を設定できるようにするためのもので、マルチプール本来の動作です。マルチプールのフェイルオーバは、データベース接続が失われた場合にだけ行われます。このような状況を防ぐには、接続プールの接続の最大数を増やす必要があります。

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。プールは順序付けされずに追加され、ラウンドロビン方式でアクセスされます。接続を切り替えると、最後にアクセスされたプールの直後の接続プールが選択されます。

マルチプールのフェイルオーバに関する制限と要件

WebLogic Server ではマルチプール用に高可用性アルゴリズムが提供されており、接続プールで障害が発生した場合（データベース管理システムがクラッシュした場合など）でも、システムは動作を続けることができます。

接続プールは、`TestConnsOnReserve` の機能を利用して、データベース接続が失われたことを認識します。接続のテストが自動的に行われるのは、アプリケーションが接続を予約してからです。接続プールに対する `TestConnsOnReserve=true` の設定は、マルチプール内で行う必要があります。この機能を有効にすると、WebLogic Server はアプリケーションに返す前に接続を個別にテストします。これは、高可用性アルゴリズムにおいて必要不可欠の動作です。高可用性アルゴリズムでは、マルチプールは、予約時に行う接続テストの結果を使って、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

ングを決定します。テストが不合格になると、接続プールは接続の再作成を試みます。この試みが失敗した場合、マルチプールは次の接続プールにフェイルオーバします。

予約された後の接続で障害が発生する可能性があります。このような場合は、アプリケーションで障害に対処する必要があります。WebLogic Server は、アプリケーションの使用中に障害が発生した接続をフェイルオーバすることはできません。使用中の接続の障害に対しては、トランザクションを再度実行すると共に、障害を処理するためのコードを用意する必要があります。

接続待ち時間を設定するためのガイドライン

接続待ち時間の設定は、接続試行のプロパティです。プール接続の待ち時間の設定に慣れている場合、接続待ち時間のプロパティを特定の接続試行で行われるすべての接続に適用します。

マルチプールには任意の接続プールを追加できます。しかし、リソースの最適化は、接続プールをコンフィグレーションするときに接続待ち時間をどのように設定したかに応じて行います。

メッセージとエラー条件

ユーザは、接続を提供する接続プールに関する情報を要求する場合があります。

例外

例外は、以下の状態のときに JDBC ログに記録されます。

- 起動時に、接続プールがマルチプールに追加されたとき
- ロード バランシングまたは高可用性のいずれかで、マルチプール内で新しい接続プールに切り替わったとき

容量の問題

高可用性のシナリオでは、リスト内の最初のプールが使用中であっても、リスト内の次のプールから接続が取得されるわけではありません。

DataSource のコンフィグレーションと使用方法

接続プールやマルチプールの場合と同様に、DataSource オブジェクトは、Administration Console で、あるいは WebLogic 管理 API を使って作成することができます。トランザクション サービスに対応する DataSource オブジェクトを定義することも、対応しない DataSource オブジェクトを定義することもできます。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することができます。各 DataSource では、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 を定義することができます。これによって、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るようになります。

WebLogic Server では、以下の 2 種類の DataSource オブジェクトをサポートしています。

- DataSource (ローカル トランザクション専用)
- TxDataSource (分散トランザクション用)

アプリケーションが以下の条件のいずれかを満たす場合には、TxDataSource を WebLogic Server で使用すべきです。

- JTA (Java Transaction API) を使用する場合
- WebLogic Server の EJB コンテナを使用してトランザクションを管理する場合
- 単一のトランザクションで複数のデータベース更新を実行する場合

TxDataSource を使用すべき場合とそのコンフィグレーション方法の詳細については、「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ようにしたい場合には (この方法を推奨)、アプリケーションを実行する前に、Administration Console で DataSource を定義しておかなければなりません。DataSource の作成方法については、『Administration Console オンライン ヘルプ』を参照してください。また、TxDataSource の作成方法については、『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ためのパッケージのインポート

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
    }
  }
```

（WebLogic Server は適切な hostname と port 番号に置き換えられます。）

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。

5 WebLogic 多層 JDBC ドライバの使い方

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。

WebLogic 多層ドライバの概要

多層ドライバには次のようにアクセスできます。

- 新しいアプリケーションの場合。新しいアプリケーションでは、DataSource オブジェクトの使用をお勧めします。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。各データ ソースには、独自の DataSource オブジェクトが必要です。DataSource オブジェクトは、接続プールまた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、4-1 ページの「WebLogic JDBC 機能のコンフィグレーション」を参照してください。
- 既存のアプリケーションの場合。JDBC 1.x API を使用する既存のアプリケーションの場合は、以降の節を参照してください。

WebLogic RMI ドライバの使い方

WebLogic RMI ドライバは WebLogic Server 内で動作する多層 Type 3 JDBC ドライバで、以下のドライバと一緒に使用されます。

- ローカル トランザクションにデータベース アクセスを提供する 2 層 JDBC ドライバ（WebLogic jDriver ファミリなど）
- 分散トランザクションのための 2 層 JDBC XA ドライバ（WebLogic jDriver for Oracle/XA など）Oracle Thin ドライバ 8.1.7 の使い方の詳細については、6-14 ページの「Oracle Thin Driver の拡張機能」を参照してください。

BEA WebLogic RMI ドライバは、WebLogic Server と連携して動作します。DBMS 接続は、WebLogic Server、*DataSource* オブジェクト、および WebLogic Server 内で動作する接続プールを使って行われます。

DataSource オブジェクトは、RMI ドライバ接続へのアクセスを提供します。接続パラメータは、Administration Console で設定します。次に、DBMS への 2 層 JDBC によるアクセス用に、この接続プールのコンフィグレーションを行います。

RMI ドライバ クライアントは、この 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（前回のリリースから非推奨）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際の制限事項

次の事項に注意してください。

- RMI ドライバを WebLogic jDriver for Oracle ドライバまたは準拠するサードパーティ 2 層ドライバと組み合わせて使用している場合、`callableStatement` または `preparedStatement` SQL 文ではバッチ更新（`addBatch()`）は使用できません。

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。

WebLogic Server を使用するためのクライアントの設定

以下のコード例では、接続を取得して使用するための方法を説明します。

以下のパッケージをインポートする

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

クライアント接続を取得する

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ほかにもあります。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た接続の取得

Driver.connect() メソッドを使用して WebLogic Server にアクセスすることもできます。その場合には、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。

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。

XA 非対応の JDBC ドライバを備えたトランザクション データ ソースに対して [2 フェーズ コミットを有効化] (`enableTwoPhaseCommit = true`) を選択すると、WebLogic Server では内部で JTS ドライバを使用して、XA 非対応のリソースが 2 フェーズ コミット (2PC) をエミュレートしてグローバル トランザクションに参加できるようにします。XA 非対応のリソースをグローバル トランザクションに参加させることができる仕組みと JTS ドライバの使い方の詳細については、『WebLogic Server 管理者ガイド』の「分散トランザクション用の XA 非対応 JDBC ドライバのコンフィグレーション」を参照してください。

JTS ドライバを使用した実装

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、『管理者ガイド』の「JDBC 接続の管理」の「接続プール」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。このクラスは、Java Naming and Directory Service（JNDI）でルックアップされます。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、データベース接続を使用する任意のサービスによって行うことができます。こうしたサービスには、EJB、JMS、標準 JDBC 文が含まれます。これらの操作が手順 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ために JTS ドライバを使用する場合、それらはそのトランザクションに参加することになります。

   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。

6 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。

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
- EJB へのアクセス（JDBC でのトランザクション呼び出しの実装）

この節では、以下のサードパーティ 2 層 Type 4 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
- Sybase jConnect ドライバ

Sybase jConnect Driver（バージョン 4.2/5.2 および 5.5）と Oracle Thin Driver（バージョン 9.2.0）は WebLogic Server 配布キットに同梱されています。`weblogic.jar` ファイルには、Oracle Thin Driver クラスと Sybase jConnect クラスが収めれています。これらの Oracle ドライバと Sybase ドライバの詳細は、それぞれの Web サイトで入手できます。

注意： WebLogic Server 6.1 サービス パック 4 では、weblogic.jar に収められている Oracle Thin Driver のバージョンは 9.2.0 です。WebLogic Server 6.1 の以前のリリースには、Oracle Thin Driver のバージョン 8.1.7 が付属していました。

制限

次の事項に注意してください。

- RMI ドライバを 2 層ドライバと組み合わせて使用している場合、`callableStatement` または `preparedStatement` SQL 文ではバッチ更新（`addBatch()`）は使用できません。

サードパーティ ドライバ用の環境の設定

`weblogic.jar` に含まれる Oracle Thin Driver または Sybase jConnect Driver 以外のサードパーティ JDBC ドライバを使用する場合、JDBC ドライバ クラスのパスを `CLASSPATH` に追加する必要があります。以降のトピックでは、サードパーティ JDBC ドライバの使用時に Windows および UNIX で `CLASSPATH` を設定する方法について説明します。

Windows でのサードパーティ ドライバの CLASSPATH

次のように、JDBC ドライバ クラスおよび `weblogic.jar` へのパスを `CLASSPATH` に追加します。

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

`DRIVER_CLASSES` は JDBC ドライバ クラスのパス、*`WL_HOME`* は WebLogic Server がインストールされているディレクトリです。

UNIX でのサードパーティ ドライバの CLASSPATH

次のようにして、JDBC ドライバ クラスおよび weblogic.jar へのパスを CLASSPATH に追加します。

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

DRIVER_CLASSES は JDBC ドライバ クラスのパス、*WL_HOME* は WebLogic Server がインストールされているディレクトリです。

Oracle Thin Driver の更新

WebLogic Server に同梱されている Oracle Thin Driver を更新するには、CLASSPATH で、weblogic.jar のパスの前に新しいドライバ クラスのパスを追加する必要があります。次に例を示します。

```
set CLASSPATH=%ORACLE_HOME%\jdbc\lib\classes12.zip;
%WL_HOME%\lib\weblogic.jar;%CLASSPATH%
```

（Windows の場合）

または

```
export CLASSPATH=$ORACLE_HOME/jdbc/lib/classes12.zip:
$WL_HOME/lib/weblogic.jar:$CLASSPATH
```

（UNIX の場合）

weblogic.jar に収められている Oracle Thin Driver バージョン 9.2.0 に更新するには、または新しいバージョンのドライバを使用するには、次の手順に従います。

Oracle Thin Driver は、Oracle DBMS ソフトウェアに含まれます。ドライバのアップデートは、http://otn.oracle.com/software/content.html の Oracle Web サイトからダウンロードできます。

Sybase jConnect Driver の更新

WebLogic Server に同梱されている Sybase jConnect Driver を更新するには、CLASSPATH で、weblogic.jar のパスの前に jConnect ドライバのパスを追加する必要があります。次に例を示します。

```
set CLASSPATH=%SYBASE_HOME%\jConnect-5_5\classes\jconn2.jar;
%WL_HOME%\lib\weblogic.jar;%CLASSPATH%
```

（Windows の場合）

または

```
export CLASSPATH=$SYBASE_HOME/jConnect-5_5/classes/jconn2.jar:
$WL_HOME/lib/weblogic.jar:$CLASSPATH
```

（UNIX の場合）

Sybase jConnect Driver（jConnect.jar）は Sybase DBMS ソフトウェアに付属します。Sybase Web サイトからドライバの更新をダウンロードすることもできます。

IBM Informix JDBC ドライバのインストールと使い方

Informix データベースと共に Weblogic Server を使用する場合は、IBM の Web サイト http://www.informix.com/evaluate/ から IBM Informix JDBC ドライバを入手して使用することをお勧めします。IBM Informix JDBC ドライバは自由に使用できますが、サポートを受けることはできません。製品をダウンロードするには、IBM への登録が必要な場合があります。JDBC/EMBEDDED SQLJ のセクションからドライバをダウンロードし、ダウンロードした zip ファイルに収められている 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は、以下の手順に従って、WebLogic Server でドライバを使用するための準備をします。

1. ifxjdbc.jar ファイルと ifxjdbcx.jar ファイルを *INFORMIX_INSTALL*\lib からコピーし、*WL_HOME*\server\lib フォルダに格納します。

   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 Server をインストールしたフォルダで、通常は c:\bea\wlserver6.1 です。

2. ifxjdbc.jar と ifxjdbcx.jar に対するパスを CLASSPATH に追加します。次はその例です。

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

WebLogic Server の起動スクリプトの set CLASSPATH 文に、ドライバ ファイルへのパスを追加してもかまいません。

IBM Informix JDBC ドライバを使用するときの接続プール属性

IBM Informix JDBC ドライバを使用する接続プールを作成するときは、表 6-1 および 表 6-2 で示す属性を使用します。

表 6-1 Informix JDBC ドライバを使用する XA 以外の接続プールの属性

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*
ログイン遅延秒数	1

表 6-1 Informix JDBC ドライバを使用する XA 以外の接続プールの属性

属性	値
ターゲット	serverName

config.xml ファイルのエントリの例を次に示します。

```
<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 6-2 Informix JDBC ドライバを使用する XA 接続プールの属性

属性	値
URL	*leave blank*
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	*leave blank*
ローカル トランザクションのサポート	true

表 6-2 Informix JDBC ドライバを使用する XA 接続プールの属性

属性	値
ターゲット	*serverName*

注意： プロパティの文字列では、portNumber と = の間にはスペースがあります。

config.xml ファイルのエントリの例を次に示します。

```
<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 を使って接続プールを作成する場合は、サーバを停止して再起動しないと、接続プールがターゲット サーバに正しくデプロイされない場合があります。これは確認されている問題です。

IBM Informix JDBC ドライバを使用するプログラミングでの注意事項

IBM Informix JDBC ドライバを使用するときは、以下の制限事項について考慮する必要があります。

- 必ず resultset.close() メソッドおよび statement.close() メソッドを呼び出して、Statement や ResultSet の使用が終了したことをドライバに示す必要があります。これを行わないと、データベース サーバ上のリソースをすべて解放することができない場合があります。
- IFX_USEPUT 環境変数を 1 に設定していない場合、TEXT カラムまたは BYTE カラムを含む行を挿入しようとすると、バッチ更新が失敗します。
- トランザクションの間に Java プログラムが自動コミット モードを true に設定すると、JDK がバージョン 1.4 以降の場合には IBM Informix JDBC ドラ

イバは現在のトランザクションをコミットし、それ以外の場合には自動コミットを有効に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。

サードパーティ ドライバを使用した接続の取得

以下の節では、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接続を取得するための方法を 2 つ説明します。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。また、Java クライアントとデータベース間の単純な接続を直接取得するという方法もあります。

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

Administration Console を使用して以下の作業を行う方法については、『管理者ガイド』の「JDBC 接続の管理」を参照してください。

- JDBC 接続プールの作成
- JDBC DataSource の作成

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ほかにもあります。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接続の取得

接続プールから接続を取得すると、WebLogic Server が接続プールで接続を管理できるよう、物理接続ではなく論理接続が提供されます。これは、アプリケーションが接続プールの機能を利用できるようにし、アプリケーションに提供される接続の品質を維持するために必要なことです。しかし、ベンダの接続クラスを必要とする DBMS 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のように、状況によっては、物理接続を使用したいことがあります。WebLogic Server に含まれる `weblogic.jdbc.extensions.WLConnection` インタフェースの `getVendorConnection()` メソッドを使うと、論理接続から基になっている物理接続を取得できます。このインタフェースについては、『WebLogic クラスの Javadoc』を参照してください。

注意： 接続プールの論理接続ではなく物理接続を使うことは極力避けるよう強くお勧めします。6-12 ページの「物理接続の使用に対する制限事項」を参照してください。

物理データベース接続の使用は、ベンダ固有の処理のために必要な場合に限るべきです。コードでは、大半の JDBC 呼び出しは引き続き論理接続に対して行うようにしなければなりません。

接続を使い終わったら、論理接続を閉じなければなりません。コード内で、物理接続を閉じてはいけません。

物理データベース接続がアプリケーション コードから参照できるようになっていると、接続プールは、その接続を次に使用するユーザによる排他的アクセスを保証できなくなります。そのため、論理接続が閉じられると、WebLogic Server 側では、その論理接続を接続プールに戻しますが、その基になっている物理接続は破棄し、プール内の論理接続用に新しい物理接続を開きます。この方式は安全ですが、反面、低速でもあります。接続プールへのリクエストのたびに新しいデータベース接続が作成される可能性があるからです。

物理接続取得のサンプル コード

物理データベース接続を取得するには、最初に 6-8 ページの「JNDI を使用した接続の取得」で説明されているように接続プールから接続を取得したあと、以下のいずれかを実行します。

- 接続を `WLConnection` にキャストして `getVendorConnection()` を呼び出す。

- 物理接続が必要なメソッドの中で物理接続を暗黙的に渡す (getVendorConnection() メソッドを使用)。

以下に例を示します。

```
// この追加クラスおよび必要なすべてのベンダ パッケージを
// インポートする。
import weblogic.jdbc.extensions.WLConnection
.
.
.
myJdbcMethod()
{
  // 接続プールから取得される接続は、クラス変数やインスタンス変数ではなく、
  // 常にメソッド レベルの変数でなければならない。
  Connection conn = null;

   try {
     ctx = new InitialContext(ht);
     // JNDI ツリーでデータ 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");

     // プール接続は、常に try ブロック内で取得する。
     // 接続の使用はそのブロックの中ですべて済ませ、必要なら、
     // finally ブロック内で接続を閉じる。
     conn = ds.getConnection();

     // conn オブジェクトを WLConnection インタフェースにキャストし、
     // 基になっている物理接続を取得する。

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

     // 次のように、vendorConn オブジェクトをベンダのインタフェースに
     // キャストすることもできる。
     // oracle.jdbc.OracleConnection vendorConn =
     //  ((WLConnection)conn)).getVendorConnection()

     // 物理接続を必要とするベンダ固有のメソッドを扱う場合には、
     // 物理接続を取得したり保持するのではなく、必要に応じて
     // ただ物理接続を暗黙的に渡すようにするのが最もよい。
     // たとえば、以下のようにする。


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

```
        // ベンダ固有の呼び出しが終わったらすぐに、
        // 接続への参照を破棄する。
        // 接続を保持したり閉じたりしてはいけない。
        // 汎用的な JDBC に対してはベンダ接続を使用しない。
        // 標準の JDBC には、引き続き論理接続 ( プール接続 ) を使用する。
        vendorConn = null;

        ... メソッド全体に必要な JDBC 呼び出しをすべて行う ...

        // 論理接続 ( プール接続 ) を閉じて接続プールに戻し、
        // その接続への参照を破棄する。
        conn.close();
        conn = null;
      }

      catch (Exception e)
      {
        // 例外を処理する
      }
      finally
      {
        // 安全のために、論理接続 ( プール接続 ) が閉じたかどうか
        // チェックする。
        // finally ブロックでは、常に最初のステップとして
        // 論理接続 ( プール接続 ) を閉じる。

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

物理接続の使用に対する制限事項

接続プールの論理接続の代わりに物理接続を使うことは極力避けるよう強くお勧めします。ただし、STRUCT を作成するためなど、物理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を考慮してください。

- 物理接続は、サーバサイド コードでのみ使用できます。
- 物理接続を使用すると、エラー処理や文のキャッシュなど、WebLogic Server によって提供される接続管理のメリットをすべて失います。
- 物理接続の使用は、それを必要とするベンダ固有のメソッドやクラスの場合に限るべきです。SQL 文の生成やトランザクション呼び出しなどの汎用的な JDBC 呼び出しには、物理接続を使用してはいけません。
- 接続は再利用されません。接続を閉じると、物理接続が閉じられ、接続プールが作成する新しい接続によって、物理接続として渡された接続は置き換え

られます。物理接続を使用すると、接続が再利用されないため、次のような理由によりパフォーマンスが低下します。

- 物理接続は接続プールの新しいデータベース接続で置き換えられるので、アプリケーション サーバとデータベース サーバの両方のリソースが使用されます。
- 元の接続に対する文キャッシュは閉じられ、新しい接続用に新しいキャッシュが開かれます。したがって、文キャッシュを使用することによるパフォーマンスのメリットが失われます。

直接 (非プール) JDBC 接続の取得

次の単純な例では、WebLogic Server で実行される Java コードとデータベースの間に単純な接続を直接確立する方法を示します。`driver.connect()` を使用して直接接続を設定します。DriverManager を使用して JDBC 接続を取得しないでください。DriverManager のメソッドは、マルチスレッドのアプリケーションに対して過剰に同期が取られるため、WebLogic Server がシングルスレッドになったり、ロックしたりする可能性があります。

以下の例では、サードパーティのドライバを使用して直接接続を取得する方法について説明します。

Oracle Thin Driver を使用した直接接続の取得

以下の例では、Oracle Thin Driver を使用して直接接続を設定する方法について説明します。

- ドライバをインスタンス化します。

```
// ThinDriver ドライバ
driver = (Driver)Class.forName
   ("oracle.jdbc.driver.OracleDriver").newInstance();

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

- 接続を行います。

```
// Thin ドライバ接続
con = driver.connect
   ("jdbc:oracle:thin:@myHost.mydomain.com:1521:DEMO", props);
```

Sybase jConnect Driver を使用した直接接続の取得

以下の例では、Sybase jConnect Driver を使用して直接接続を設定する方法について説明します。

- ドライバをインスタンス化します。

```
// Sybase jConnect ドライバ
driver = (Driver)Class.forName
   ("com.sybase.jdbc.SybDriver").newInstance()

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

- 接続を行います。

```
// Sybase jConnect
con = driver.connect
   ("jdbc:sybase:Tds:myDB@myhost:myport), props);
```

Oracle Thin Driver の拡張機能

BEA では、RMI、JTS、および Pool ドライバを使用する場合に、Oracle Thin Driver の以下の拡張機能をサポートしています。

Oracle 標準の拡張機能

- `OracleConnection`
- `OracleStatement`
- `OracleResultSet`
- `OraclePreparedStatement`
- `OracleCallableStatement`
- `OracleArray`

Oracle Blob または Clob

- `OracleThinBlob`
- `OracleThinClob`

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 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。以下の例では、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 の各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、6-18 ページの「Oracle インタフェースの表」を参照してください。

OracleThinBlob および OracleThinClob インタフェースへのアクセス方法の例については、6-17 ページの「Oracle Blob/Clob インタフェースにアクセスするサンプル コード」を参照してください。

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

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、接続を確立します。詳細については、6-8 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;
   }
```

Oracle Blob/Clob インタフェースにアクセスするサンプル コード

この節では、OracleThinBlob インタフェースへのアクセス方法を示すサンプル コードを提供します。WebLogic Server でサポートされるメソッドを使用する場合、この例の構文は、OracleThinBlob インタフェースで使用できます。6-18 ページの「Oracle インタフェースの表」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。

Blob および Clob 拡張機能にアクセスするパッケージをインポートする

インポートするクラスに次の WebLogic Server Oracle Blob も加えます。

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

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。

Prepared Statement を使用した CLOB 値の更新

Prepared Statement を使用して CLOB を更新する場合、新しい値が古い値より短いと、更新の際に明示的に置換されなかった文字が CLOB に残ります。たとえば、現在の値が `abcdefghij` である CLOB を Prepared Statement で `zxyw` という値に更新すると、更新後の CLOB の値は `zxywefghij` になります。Prepared Statement で更新した結果を正しい値にするには、`dbms_lob.trim` プロシージャを使って、更新後に残っている余分な文字を削除する必要があります。`dbms_lob.trim` プロシージャの詳細については、Oracle のマニュアルを参照してください。

Oracle インタフェースの表

以下の表に、Oracle インタフェースを示します。

Oracle 拡張機能およびサポートされるメソッド

以下の表に Oracle インタフェースを示します。また、`java.sql.*` インタフェースを拡張するために Oracle Thin Driver で使用するメソッドで、サポートされているものを示します。Blob/Clob インタフェースについては、6-17 ページの「Oracle Blob/Clob インタフェースにアクセスするサンプル コード」を参照してください。

表 6-3 OracleConnection インタフェース

拡張	メソッド シグネチャ

OracleConnection extends 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 6-3 OracleConnection インタフェース（続き）

拡張	メソッド シグネチャ

OracleConnection extends java.sql.Connection（続く）	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 6-3 OracleConnection インタフェース（続き）

拡張	メソッド シグネチャ

OracleConnection extends java.sql.Connection（続く）	void **removeAllDescriptor()**
	  throws java.sql.SQLException;
	// これは Sun の綴り
	void **removeDecriptor(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;

注意： サービス パック 4 では、次のメソッドが削除されました。

- isCompatibleTo816()

表 6-4 OracleStatement インタフェース

拡張	メソッド シグネチャ

OracleStatement extends java.sql.statement	String **getOriginalSql()** throws java.sql.SQLException;
	String **getRevisedSql()** throws java.sql.SQLException;
	boolean **getAutoRefetch()** throws java.sql.SQLException;
	boolean **is_value_null(boolean b, int i)** throws java.sql.SQLException;
	byte **getSqlKind()** 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;

OracleStatement extends java.sql.statement
（続く）
	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 **setRowPrefetch(int i)** throws java.sql.SQLException;

注意： サービス パック 4 では、次のメソッドが削除されました。

- getWaitOption()
- setWaitOption(int i)
- setAutoRollback(int i)
- getAutoRollback()

表 6-5 OracleResultSet インタフェース

拡張	メソッド シグネチャ

OracleResultSet extends 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;

注意： サービス パック 4 では、次のメソッドが削除されました。

- getCURSOR(String s)

表 6-6 OracleCallableStatement インタフェース

拡張	メソッド シグネチャ
OracleCallableStatement extends 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 6-7 OraclePreparedStatement インタフェース

拡張	メソッド シグネチャ
OraclePreparedStatement extends OracleStatement and 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;

Oracle Blob/Clob 拡張機能とサポートされるメソッド

次の表に、java.sql.* インタフェースの拡張機能を示します。

表 6-8 OracleThinBlob インタフェース

拡張	メソッド シグネチャ
OracleThinBlob extends 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 6-9 OracleThinClob インタフェース

拡張	メソッド シグネチャ
OracleThinClob extends 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;

7　dbKona の使い方

以下の節では、Java アプリケーションとの高レベルなデータベース接続を提供する dbKona クラスについて説明します。

dbKona の概要

dbKona クラスには、Java アプリケーションやアプレットにデータベースへのアクセスを提供する、一連の高レベルなデータベース接続オブジェクトが用意されています。dbKona は、JDBC API の最上位に位置し、WebLogic JDBC ドライバなどの JDBC 準拠ドライバと共に機能します。

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 プログラミング 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 のインストールと使い方』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、Informix、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 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）

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

    Class weblogic.db.jdbc.Value
```

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
  - QueryDataset
  - TableDataSet
  - EventfulTableDataSet （非推奨）
- Record
  - Value

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「ROWID」に設定できます。「ROWID」は、各テーブルのユニークなキーです。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 JDBC 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 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します）。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 Record 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 オブジェクトは、DataSet およびその Schema のコンテキスト内で、またはアクティブな Database セッションに知られている SQL テーブルの Schema のコンテキスト内で、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.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 Schama 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 で使用されるデータ記述オブジェクトの中には、JDBC インタフェースの実装であるオブジェクトもあります。ここでは、以下のデータ記述オブジェクトの概要とその使用法について説明します。

- 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 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、QueryDataSets と TableDataSets、およびそれらに関連付けられた Record の属性）は、Table 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 内の属性数を確認するためのメソッド
- 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 オブジェクトは、SelectStmt クラスで作成されます。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
- Where

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「tableName alias」という形式で指定できます。field() メソッドでは、「tableAlias.attribute」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
- 定数

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図

継承関係図

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 を継承します。

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。

所有関係図

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。その関係を次に示します。

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 Columns があります。

dbKona を使用した実装

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.util.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、このチュートリアルで Oracle DBMS との接続を確立するために後で使用されます。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。

Connections、DataSets（使用している場合は JDBC ResultSets）、および Statements 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() の変わりに 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() および Record.getValue() を使用して、for ループで各レコードを検査し、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.util.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.util.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 7-35 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 stmtl = conn.createStatement();
stmtl.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;");
stmtl.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` 型の出力パラメータとして登録します。そして最後にストアド プロシージャを実行しています。

```
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.util.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();
    }
}
```

8 JDBC 接続のテストとトラブルシューティング

以下の節では、JDBC 接続のテスト方法を説明するとともに、トラブルシューティングのヒントを紹介します。

接続のテスト

以降の節では、接続をテストする方法について説明します。

コマンドラインからの DBMS 接続の有効性の検証

BEA では、WebLogic 2 層ドライバ、WebLogic Server、または WebLogic JDBC をインストールした後、2 層および 3 層の JDBC データベース接続をテストするために使用できるユーティリティを用意しています。

コマンドラインからの 2 層接続をテストする方法

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 NT）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 ドライバのインストール手順については、以下を参照してください。
  - 「WebLogic jDriver for Oracle のインストール」
  - 「WebLogic jDriver for Microsoft SQL Server のインストール」
  - 「WebLogic jDriver for Informix のインストール」

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

ORACLE、MSSQLSERVER4、または INFORMIX4 を使用します。

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`、Informix では `DBACCESS` 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`、`sqlplus`、または `DBACCESS` 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`、`sqlplus`、または `DBACCESS` で使用するものと同じ値と形式を使用します。MSSQLServer4 や Informix4 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。

例

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。

Informix（Type 4 ドライバ）

`DBACCESS` で使用する同じ値を利用し、Java から WebLogic jDriver for Informix 経由で Informix に接続します。引数は、次の順序で指定します。

```
$ java utils.dbping INFORMIX user pass db@server:port
```

この例では、次のように指定します。

```
$ java utils.dbping INFORMIX bill secret stores@myserver:8543
```

コマンドラインからの多層 WebLogic JDBC 接続の有効性を検証する方法

`utils.t3dbping` ユーティリティを使用すると、WebLogic Server を使用した多層データベース接続が可能なことを確認できます。`t3dbping` ユーティリティは、多層接続のテストのみに使用します。このテストは、2 層接続が正常に機能することを確認し、WebLogic を起動した後に行います。

2 層 JDBC ドライバが WebLogic jDriver の場合は、`utils.dbping` を使用して 2 層接続をテストします。他のドライバの場合は、多層接続をテストする前に、使用する 2 層 JDBC ドライバのドキュメントを参照して接続のテスト方法を確認してください。

構文

```
$ java utils.t3dbping URL user password DB driver_class driver_URL
```

引数

URL

WebLogic Server の URL。

username

DBMS の有効なユーザ名です。

password

ユーザの有効なパスワードです。

DB

データベースの名前。前述の 2 層接続のテストで使用した同じ値と形式を使用します。

driver_class

WebLogic と DBMS を接続する JDBC ドライバのクラス名です。たとえば、サーバサイドで WebLogic jDriver for Oracle を使用する場合、ドライバのクラス名は `weblogic.jdbc.oci.Driver` です。ドライバのクラス名では、ドット（.）表記を使用します。

driver_URL

WebLogic と DBMS を接続する JDBC ドライバの URL です。たとえば、サーバサイドで WebLogic jDriver for Oracle を使用する場合、ドライバの URL は `jdbc:weblogic:oracle` です。ドライバの URL はコロンで区切ります。

例

以下の例は、読みやすいように複数の行に分けられています。実際には、1 つのコマンドとして入力してください。

Oracle

次の例は、bigbox というサーバで動作している Oracle DBMS DEMO20 に対して ping を実行する方法を示しています。WebLogic も同じホストにあり、ポート 7001 でリスンしています。

```
$ java utils.t3dbping           // コマンド
     t3://bigbox:7001           // WebLogic URL
     scott tiger                // ユーザ名とパスワード
     DEMO20                     // DB
     weblogic.jdbc.oci.Driver   // ドライバのクラス
     jdbc:weblogic:oracle       // ドライバの URL
```

DB2 と AS/400 Type 4 JDBC ドライバ

次の例は、IBM AS/400 Type 4 JDBC ドライバを使用してワークステーションのコマンド シェルから AS/400 DB2 データベースに対して ping を実行する方法を示しています。

```
$ java utils.t3dbping                       // コマンド
     t3://as400box:7001                     // WebLogic URL
     scott tiger                            // ユーザ名とパスワード
     DEMO                                   // データベース
     com.ibm.as400.access.AS400JDBCDriver   // ドライバのクラス
     jdbc:as400://as400box                  // ドライバの URL
```

WebLogic jDriver for Microsoft SQL Server（Type 4 JDBC ドライバ）

次の例は、WebLogic jDriver for Microsoft SQL Server を使用して Microsoft SQL Server データベースに対して ping を実行する方法を示しています。

```
$ java utils.t3dbping       // コマンド
     t3://localhost:7001    // WebLogic URL
     sa                     // ユーザ名
     abcd                   // パスワード
     hostname               // database@hostname:port
                            // （URL の一部として指定する場合は
                            // 省略可能）
weblogic.jdbc.mssqlserver4.Driver // ドライバのクラス
jdbc:weblogic:mssqlserver4:pubs@localhost:1433
                // ドライバの URL:database@hostname:port
                //（データベース パラメータで使用されている場合は省略可）
```

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 接続のトラブルシューティング

WebLogic への接続をテストする場合は、WebLogic のログを調べてください。デフォルトでは、ログは weblogic\myserver ディレクトリの weblogic.log というファイルに記録されています。

UNIX ユーザ

native_login のロードで問題が発生した場合は、truss を使用して問題の原因を突き止めます。たとえば、tutorial.example3 を実行するには、次のように入力します。

```
$ truss -f -t open -s\!all java tutorial.example3
```

WinNT

.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。

JDBC と Oracle データベースでの SEGV

条件次第では、JDBC と Oracle データベースを使用するときにセグメンテーション違反エラー（SEGV）やハングが発生します。以下の事項に注意してください。

- クライアント ライブラリは、「BEA WebLogic Server プラットフォーム サポート」で指定されているとおりに最新版にアップグレードする必要があります。
- WebLogic jDriver for Oracle で .dll、.sl、または .so のバージョンが一致していない WebLogic クラスが使用されています。常に、WebLogic 配布キットの特定のバージョンで提供される .dll、.so、または .sl ファイルを使用しなければなりません。
- 接続プールの利用可能な接続が使い果たされています。利用が終わった接続では必ず close() メソッドが呼び出されるようにします。もっと多くの接続が必要な場合は、プールのサイズを大きくします。
- Oracle サーバと WebLogic が同じホストで動作している状況で Oracle への IPC 接続を使用する場合、クライアント ライブラリのバージョンはサーバのバージョンと一致していなければなりません。サーバとクライアントが同じホストにある場合、sqlnet ではデフォルトで IPC 接続が試行されます。IPC

接続を無効にするには、sqlnet.ora ファイルで "automatic_ipc"=off を指定します。

- ORACLE_HOME 環境変数が正確に設定されていません。ORACLE_HOME は、OCI ライブラリが必要なリソース ファイルを見つけられるように正確に設定する必要があります。

メモリ不足エラー

メモリ不足エラーのよくある原因は、ResultSet が閉じていないことです。たいていは、以下のようなエラー メッセージが表示されます。

```
Run-time exception error; current exception: xalloc
No handler for exception
```

配列フェッチを使用すると、ネイティブ レイヤではメモリが Java ではなく C に割り当てられます。そのため、Java のガベージ コレクションではメモリがすぐにはクリーンアップされません。メモリを解放する唯一の方法は、ResultSet を閉じることです（このメモリ使用量を最小限に抑えれば、パフォーマンスが向上します）。

メモリ不足エラーを防止するには、プログラム ロジックで必ず ResultSet を閉じるようにします。閉じていない ResultSet が原因でメモリ不足エラーが発生しているのかをテストするには、配列フェッチのサイズを最小限にして、割り当てられる C メモリの量を少なくします。そのためには、weblogic.oci.cacheRows プロパティ（JDBC 接続プロパティ）を小さい値に設定します。次に例を示します。

```
   Properties props = new java.util.Properties();
   props.put("user",                  "scott");
   props.put("password",              "tiger");
   props.put("server",                "DEMO" );
   props.put("weblogic.oci.cacheRows", "1"    );

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

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

これでメモリ不足エラーが解消されたら、ResultSet がコードのどこかで閉じられていないと判断できます。詳細については、「JDBC オブジェクトを閉じる」を参照してください。

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。

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

プログラムを効率的に実行するために、finally ブロックで Connection、Statement、ResultSet などの JDBC オブジェクトを必ず閉じるようにしてください。これは WebLogic の推奨であるとともに、プログラミングの優れた慣例でもあります。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) {}

    }
```

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 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 のインストール」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 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 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 weblogic_directory/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、weblogic_directory/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 weblogic_directory/lib/hpux11:$SHLIB_PATH
```

あるいは、WebLogic 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 libweblogicoci37.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=
$WL_HOME/lib/hpux11/oci901_8:ORACLE/lib32:$SHLIB_PATH
```

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。