BEA WebLogic Server

WebLogic JDBC T3 ドライバ ユーザーズ ガイド (非推奨)

BEA WebLogic Server 6.1
マニュアルの日付 : 2001 年 11 月 30 日

著作権

限定的権利条項

商標または登録商標

WebLogic JDBC T3 ドライバ ユーザーズ ガイド（非推奨）

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	2001 年 11 月 30 日	BEA WebLogic Server バージョン 6.1

目次

このマニュアルの内容

1. WebLogic JDBC T3 ドライバ（非推奨）の使い方

このマニュアルの内容

このマニュアルは、非推奨になった JDBC T3 ドライバの使い方について説明します。

- 第 1 章「WebLogic JDBC T3 ドライバ（非推奨）の使い方」

対象読者

このマニュアルは、データベース アクセスが必要なアプリケーションの構築に関心があるアプリケーション開発者を対象としています。SQL、データベースの一般的な概念、および Java プログラミングに読者が精通していることを前提として書かれています。

e-docs Web サイト

BEA 製品のドキュメントは、BEA の Web サイトで入手できます。BEA のホーム ページで [製品のドキュメント] をクリックします。

このマニュアルの印刷方法

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

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

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

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

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

- お名前、電子メール アドレス、電話番号、ファクス番号
- 会社の名前と住所
- お使いの機種とコード番号

- 製品の名前とバージョン
- 問題の状況と表示されるエラー メッセージの内容

表記規則

このマニュアルでは、全体を通して以下の表記規則が使用されています。

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

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

斜体の等幅テキスト	コード内の変数を示す。 例 : `String *CustomerName;*`
すべて大文字のテキスト	デバイス名、環境変数、および論理演算子を示す。 例 : LPT1 BEA_HOME OR

表記法	適用
{ }	構文の中で複数の選択肢を示す。

[ ]	構文の中で任意指定の項目を示す。
	例 :
	`java utils.MulticastTest -n *name* -a *address*`
	`[-p *portnumber*] [-t *timeout*] [-s *send*]`

|	構文の中で相互に排他的な選択肢を区切る。
	例 :
	`java weblogic.deploy [list|deploy|undeploy|update]`
	`password {application} {source}`

...	コマンドラインで以下のいずれかを示す。 ♦ 引数を複数回繰り返すことができる ♦ 任意指定の引数が省略されている ♦ パラメータや値などの情報を追加入力できる
. . .	コード サンプルまたは構文で項目が省略されていることを示す。

1 WebLogic JDBC T3 ドライバ（非推奨）の使い方

この節では、以下の内容で非推奨になった WebLogic JDBC T3 ドライバについて説明します。

- T3 ドライバの非推奨
- JDBC の概要
- WebLogic JDBC API
- WebLogic JDBC の実装
- WebLogic JDBC のその他の機能
- WebLogic JDBC と JDBC-ODBC ブリッジの実装
- JDBC 接続のプロパティを設定するための URL の使い方と T3 ドライバの使い方

T3 ドライバの非推奨

T3 ドライバは、WebLogic Server バージョン 6.0 より非推奨になりました。BEA では、T3 ドライバの代わりに RMI ドライバを使用することをお勧めしています。「WebLogic Server 4.5 および 5.1 アプリケーションのバージョン 6.x への移行」を参照してください。

このマニュアルに新しい内容は追加されていませんが、WebLogic Server バージョン 5.1 から、主に以下の表に示す点が変更されています。

表 1-1 非推奨になった T3 ドライバのリソース

使用する機能	従来の機能	説明
RMI ドライバ	T3 ドライバ	できるかぎり、T3 ドライバの代わりに RMI ドライバを使用する。RMI 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 RMI ドライバの使い方」を参照。
myDriver.connect()	DriverManager.getConnection()	`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る。このため、`DriverManager.getConnection()` の代わりに **`Driver.connect()`** メソッドを使用することを推奨。
Administration Console	weblogic.properties ファイル	属性を設定するには Administration Console を使用する。これにより、weblogic.properties ファイルは使用されなくなった。詳細については、『管理者ガイド』の「JDBC 接続の管理」を参照。
属性	weblogic.properties ファイル	JDBC の属性リストを表示するには、オンライン ヘルプの「JDBC 接続プール」を参照。

JDBC の概要

JavaSoft の JDBC 仕様に準拠した WebLogic の多層 JDBC 実装である WebLogic JDBC を使用すると、アプリケーションが WebLogic 内からデータベースにアクセスできるようになります。WebLogic は、WebLogic JDBC を使ったデータベース アクセスを始めとするさまざまなサービスを提供する多層サーバ製品をサポートしています。WebLogic JDBC を使って構築された Java クライアントには、クライアントサイド データベース ライブラリが必要ありません。

WebLogic JDBC には、WebLogic Server とデータベース サーバとの間で JDBC ドライバが必要です。WebLogic 2 層ドライバ、WebLogic jDriver for Oracle を使用できます。また、ODBC アクセス用の JDBC ドライバなど、その他の JDBC ドライバを使用することもできます。WebLogic Server と DBMS との間で非 WebLogic JDBC ドライバを使用する場合、非 WebLogic JDBC 2 層ドライバが 2 層環境で十分に動作することを確認してからの WebLogic JDBC のデプロイだけがサポートされます。

WebLogic JDBC のアーキテクチャ

WebLogic JDBC のアーキテクチャは、WebLogic の多層環境の一部としての WebLogic のフレームワークにおける位置付けによって定義されます。WebLogic JDBC Server は、WebLogic JDBC アプリケーションとそのアクセス先のリモート DBMS との中間に位置します。WebLogic JDBC アプリケーションは WebLogic Server のクライアントであり、WebLogic Server は DBMS のクライアントとなります。この関係には 2 つの側面があり、どちらも理解しておく必要があります。

WebLogic Server とその WebLogic JDBC クライアント間の関係。各 WebLogic JDBC クライアントは、WebLogic 内で独自のコンテキスト、つまりワークスペースを持っています。WebLogic Server とクライアントとの間のプロトコルは、多重化された双方向の非同期接続です。T3Client と WebLogic Server との接続は「リッチソケット接続」で、1 つの TCP ソケット接続よりも多くの情報を転送する接続です。内部的には、WebLogic Server は効率的なパケットベースのキュー プロトコルを使用します。T3Client と WebLogic Server との関係の詳細については、『開発者ガイド』（非推奨）の「WebLogic クライアント アプリケーションの作り方」を参照してください。

WebLogic Server と DBMS 間の関係。WebLogic Server は、T3Client に代わって、JDBC ドライバとベンダ ライブラリを使用して 1 つまたは複数のデータベースとの接続を保持します。WebLogic Server は、JDBC ドライバを経由してリモート DBMS と、その JDBC ドライバによって決まる、ベンダ固有ライブラリまたは ODBC のいずれかと通信します。WebLogic Server と DBMS 間の接続を「2 層接続」としていいます。WebLogic Server は複数のデータベースに接続して 1 クライアントからのリクエストに応答することも、1 つのデータベースに接続して複数のクライアントのリクエストに応答することもできます。

また、WebLogic JDBC クライアントとクライアントがアクセスするデータベースという基本的な関係の間に、WebLogic Server が入ることもあります。この関係は、JDBC 接続オブジェクトによって定義されます。JDBC 接続の作成および使用する方法はいくつかあります。WebLogic JDBC クライアントは、特定のプロパティを設定し、適切な JDBC ドライバを識別して、JDBC 接続を作成します。JDBC 接続オブジェクトは、WebLogic Server 上で T3Client のワークスペースにキャッシュ（保存）できます。JDBC 接続は、WebLogic Server の起動時に、1 つまたは複数のクライアントが使用可能な接続プールとしても作成できます。

WebLogic Server は JDBC アプリケーションを拡張するさまざまな機能を備えていますが、WebLogic JDBC アプリケーションはセッションの開始時に WebLogic Server に接続し、セッションの終了時に接続を解除する必要があります。WebLogic Server と DBMS 間の接続は透過的に処理されます。

多層コンフィグレーションでのサードパーティ JDBC 2.0 ドライバの使い方

注意： WebLogic Server バージョン 6.0 には JDK 1.3 が必要です。

WebLogic JDBC と WebLogic Server では、サードパーティの JDBC 2.0 ドライバを使用することもできます。その場合、WebLogic Server または WebLogic JDBC を Java 2（JDK 1.2.x）環境で実行する必要があります。現在、JDBC 2.0 ドライバを使用する場合には、Java 2 の使用に関する制限を考慮する必要があります。これらの制限については、WebLogic のプラットフォームのページで説明しています。

多層コンフィグレーションでサードパーティの JDBC 2.0 ドライバを使用する場合、ドライバが呼び出したり返したりするデータは、WebLogic 多層ドライバを経由して透過的に渡されます。このため、そのドライバで使用可能なすべての機能が WebLogic の多層コンフィグレーションで使用できるようになります。

JDBC 2.0 ドライバを多層コンフィグレーションで使用するには、コードを以下のように変更します。

1. JDBC ドライバを登録する Java コード部分を変更します。
2. JDBC ドライバの URL が入っている Java コード部分を変更します。

3. CLOB または BLOB データ型を使用する場合、行キャッシュはサポートされません。行キャッシュを無効にするには、コードで次の接続プロパティを設定します。

```
weblogic.t3.cacheRows=0
```

4. コードを再コンパイルします。

WebLogic JDBC API

API リファレンス

```
Package  java.sql
Package  java.math
Package weblogic.jdbc.common
```

WebLogic は、特定の WebLogic JDBC 拡張機能（多層環境での WebLogic jDriver JDBC 拡張の一部サポート）を実現する **JDBC の拡張**を提供します。これらの拡張機能についての API（Javadoc）ドキュメントへのリンクについては、API リファレンスを参照してください。

WebLogic JDBC オブジェクトとそのクラス

注意： WebLogic Server バージョン 6.0 では、JDBC 2.0 が実装されています。

JDBC の実装については、この開発者ガイドでは説明していません。ただし、BEA のその他のオンライン リファレンスと一緒に JavaSoft の classdocs（API リファレンス マニュアル）を提供しています。JavaSoft のサイトからは、JDBC クラスおよび API のマニュアルを自由にダウンロードできます。このマニュアルで説明する WebLogic JDBC 固有のオブジェクトおよびクラスは、WebLogic のフレームワークを使用します。

WebLogic JDBC アプリケーションにインポートするクラスは以下のとおりです。

- `java.sql.*`（ドライバ `weblogic.jdbc.t3.Driver` 用）

- weblogic.common.T3Client
- java.util.Properties

その他のクラス

weblogic.common.T3Client

weblogic.common.T3User

weblogic.common.T3Exception

weblogic.common.* パッケージには、T3Client をインスタンス化する T3Client クラスと、WebLogic JDBC が WebLogic のフレームワークで機能するために使う T3 固有のオブジェクトが入っています。T3User クラスもこのパッケージに入っています。T3User オブジェクトは、ユーザ名およびパスワード情報をセキュア WebLogic Server、つまりアクセスするために認証が必要なサーバに渡すために使われます。

weblogic.jdbc.t3.Connection

WebLogic では、T3Client が接続を使って cacheRows プロパティをリセットできるように JDBC が拡張されています。WebLogic の API リファレンスおよびマニュアルではこの拡張に関してのみ説明されており、JDBC に関するその他の情報については、Sun のサイトを参照してください。

java.util.Properties

java.util.Properties オブジェクトは、JDBC 接続オブジェクトを構築するための引数として使われます。

JDK 1.3 へのアップグレード

WebLogic リリース 3.0 の場合、WebLogic JDBC を使用するには、Java Developers Kit のバージョン 1.1 にアップグレードする必要がありました。WebLogic は現在、JDK 1.0.2 をサポートしていません。1.0.2 JVM に対する WebLogic Server の実行については、現在はサポートしていません。また、1.0.2 に対する WebLogic アプリケーションの起動またはコンパイルについても、現在はサポートしていません。

JDK のバージョン 1.0.2 から 1.1 での重要な変更点は、JDBC クラス（java.sql.*）が JDK 1.1 に含まれたことです。WebLogic が提供していた一時的 JDBC クラス セット（xjava.sql.* および weblogic.db.xjdbc.*）がなくとも、JDK 1.1 を使用できるようになりました。

アップグレード方法

1. xjava.sql.* および weblogic.db.xjdbc.* を参照するインポート文を java.sql.* および weblogic.db.jdbc.* に変更します。次に例を示します。

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

   詳細 : JDK 1.1 を使う前に、コード内のすべての参照を xjava.sql.* から java.sql.* に変更します（dbKona を使う場合は、weblogic.db.xjdbc.* への参照も weblogic.db.jdbc.* に変更する必要があります）。これはインポート文の変更だけを意味する場合があります。また、xjava.sql.* および weblogic.db.xjdbc.* クラスへの明示的な参照のコードもチェックする必要があります。

次の節には実装の例があります。

1. WebLogic JDBC ドライバ クラス名への参照を weblogic.jdbc.t3.Driver に変更し、WebLogic JDBC URL への参照を jdbc:weblogic:t3 に変更します。

   詳細 : JDK 1.1 への変更では矛盾をなくすための新しい命名規則が採用されました。新しく導入されたクラス、weblogic.jdbc.t3.Driver は weblogic.jdbc.t3client.Driver と同じですが、このクラスは、xjava.sql.* の代わりに java.sql.* を使用します。WebLogic JDBC ドライバの URL への参照を *jdbc:weblogic:t3* に変更し、WebLogic JDBC ドライバのクラス名を *weblogic.jdbc.t3.Driver* に変更する必要があります。

WebLogic JDBC の実装

ここでは、WebLogic JDBC アプリケーションを構築する手順を説明します。完全コード例で使われている簡単なアプリケーションは、WebLogic Server を介して Oracle データベースに接続して、一連のレコードの挿入、更新、および削除を実行し、ストアド プロシージャおよび関数を作成および使用します。その他の例として、特に、ストアド プロシージャおよび関数の節で、Sybase データベースで使うコードと同様のコードを順を追って説明しています。

最初の 5 つの手順は、アプリケーションに現れる順番で説明されており、順に番号が付けられています。

- 手順 1. パッケージのインポート
- 手順 2. T3Client の作成
  - 明示的または埋め込み T3Client の使い方
- 手順 3. 接続用プロパティの設定
  - 2 層接続用に設定するプロパティ
  - 多層接続用に設定するプロパティ
  - URL を使用した WebLogic JDBC プロパティの設定
  - 埋め込み T3Client の設定
- 手順 4. DBMS との接続
  - 名前付きキャッシュ JDBC 接続の使い方
  - スタートアップ接続プールの作成
  - 動的接続プールの作成
  - 接続プールの管理
- レコードの挿入、更新、および削除
- ストアド プロシージャおよび関数の作成と使用
- 最後の手順 T3Client の接続解除
- コードのまとめ

- WebLogic JDBC のその他の機能
  - Oracle リソースの待機
  - 拡張 SQL
  - Oracle 配列フェッチ
  - マルチバイト文字セットのサポート

WebLogic JDBC で Oracle データベースを使うための完全コード例は、このマニュアルの最後に記載されています。順を追った説明で使用しているコード例の多くも含まれています。

WebLogic Server と DBMS との接続に WebLogic jDriver 2 層ドライバを使用している場合、使用する各 2 層ドライバの『開発者ガイド』も参照してください。

手順 1. パッケージのインポート

以下のものを WebLogic JDBC アプリケーションにインポートします。

- `java.sql.*`
- `weblogic.common.*`
- `java.util.Properties`

Properties オブジェクトを作成して DBMS にアクセスするためのパラメータを設定できるようにするには、`java.util.Properties` をインポートします。`weblogic.common.*` パッケージには、WebLogic フレームワーク内で機能するすべてのアプリケーションによって共有されるクラスが入っています。これらのクラスの詳細については、WebLogic JDBC とクラスを参照してください。

メソッドの最初に接続オブジェクトを宣言します。これは `try` ブロックと finally ブロックで使用されます。

また、埋め込み T3Client を**使用しない場合**は、`try` ブロックの前に T3Client オブジェクトを宣言します。リリース 2.3.2 で追加された埋め込み T3Client 機能は、`java.util.Properties` オブジェクトに別のプロパティを追加することで設定されます。埋め込み T3Client は自動的に構築および接続され、他の操作のために明示的な T3Client オブジェクトを必要としない任意の WebLogic JDBC クラスで使用できます。

手順 2. T3Client の作成

明示的または埋め込み T3Client の使い方

注意： 現在、ワークスペースは、WebLogic Server の一部ではありません。

一般に、各 WebLogic JDBC アプリケーションは、T3Client オブジェクトの作成から始まります（リリース 2.3.2 以降で使用可能な埋め込み T3Client 機能を使用すると、T3Client が自動的に作成される場合があります）。T3Client は、WebLogic Server 内でこのクライアント用のクライアント コンテキストとなり、クライアントとそのリクエストを一意に識別します。また、T3Client は、WebLogic Server 内で独自のワークスペースを所有し、そこで継続的な WebLogic Server セッションに合わせて T3Client 自身を再構成することができます。WebLogic JDBC アプリケーションに対しては、T3Client は JDBC 接続のコンストラクタに渡されるプロパティの 1 つでもあります。

T3Client クラス、weblogic.common.T3Client には、WebLogic JDBC で使用するコンストラクタおよびいくつかのメソッドが含まれています。新規の T3Client を作成することも、すでに作成されたワークスペースを再構成して T3client を前の状態に戻すこともできます。クライアントのワークスペースには、一連のキャッシュ JDBC 接続が含まれています。これらの接続は、一群の WebLogic JDBC クライアントが使用できるように WebLogic Server に格納されます。

すべての T3Client コンストラクタには、少なくとも 1 つの引数、WebLogic Server の URL および（オプションでポートが 80 以外の場合）WebLogic Server が T3Client 接続リクエストをリスンする TCP ポートが必要です。URL は次のフォーマットで表現されます。

```
accessProtocol://WebLogicServerURL:port
```

accessProtocol は、「T3Client アプリケーションの作成」に記載されているプロトコルのいずれかです。**t3**（高性能、多重送信、非同期、そして双方向の接続による標準 T3Client アクセス）、**t3s**（SSL で認証や暗号化を行う T3Client アクセス）、**http**（トランスファイアウォール用の HTTP トンネリングによる T3Client アクセス）などです。

WebLogicServerURL は、該当するマシンで適切な任意の方法で決めます。*port* は、WebLogic Server が T3Client のログイン リクエストをリスンしているポートです。

T3Client を構築する例を示します。

```
T3Client t3 = null;
  try {
  t3 = new T3Client("t3://bigbox:7001");
```

各 T3Client は、WebLogic Server 内にワークスペースを持ち、その中にコンテキストを格納し、後で再利用します。T3Client の作成時にワークスペースに名前を付けることもできます。名前を付けておくと、ワークスペースの再利用が簡単になります。または、T3Client の接続後にデフォルトのワークスペースを取得することもできます。ワークスペースは統合ビジネス アプリケーションを作成するための便利なモデルです。ワークスペースの詳細については、「WebLogic Workspaces の使い方」を参照してください。

WebLogic JDBC クライアント用として、名前の付けたワークスペース領域には、データベース（connectionID）へのキャッシュ JDBC 接続へのアクセスだけでなく、ハード、ソフト、およびアイドルによる接続解除用に設定された値も含まれています。また、T3Client は、String キーで識別される任意のオブジェクトをワークスペースに格納しておいて、後で取り出すこともできます。

再利用可能なワークスペースを作成する方法の 1 つは、ワークスペースに名前を付けることです。そのためには、T3Client を構築するときに、引数として String 名をコンストラクタに提供します。次に例を示します。

```
t3 = new T3Client("t3://bigbox:7001", "mike");
```

再利用可能なワークスペースを作成する別の方法は、T3Client のデフォルトのワークスペースの String ID を保存しておくことです。再利用するときはそれを使用できます。ワークスペースの ID は、T3Client が WebLogic Server に接続した後に使用可能になります。次に例を示します。

```
t3 = new T3Client("t3://bigbox:7001");
t3.connect();
String wsid = t3.services.workspace().getWorkspace().getID();
```

以下のように、ワークスペース ID（または T3Client の名前）を使用して、新しい T3Client を作成できます。

```
t3.disconnect();
t3 = null;

// 「wsid」を使用して再接続する
System.out.println("Reconnecting client " + wsid);
t3 = new T3Client("t3://bigbox:7001", wsid);
t3.connect();
```

どのワークスペース ID とも一致しない文字列、または現在 WebLogic Server にない T3Client 名が提供された場合、新しいワークスペースを作成するものとみなされ、提供した文字列で新しい名前が付けられます。

WebLogic Server での作業を終えると、finally ブロックで `disconnect()` メソッドを呼び出して、クライアントのリソースを破棄する必要があります（下の最後の手順参照）。

T3Client オブジェクトを取得した後は、T3Client の接続解除のタイムアウトを設定できます。WebLogic は、ハード接続解除（WebLogic Server とクライアントとのソケットを解除）およびソフト解除（T3Client が `disconnect()` メソッドを呼び出すことによって接続解除を要求）のタイムアウトをサポートします。次のように、これらのタイムアウトを設定（分単位）して、WebLogic Server での T3Client オブジェクトのクリーンアップを遅らせることができます。

```
t3.setSoftDisconnectTimeoutMins(5);
```

ハード接続解除もソフト接続解除も、デフォルトでは WebLogic Server の T3Client リソースを即時クリーンアップします。これらの値を `T3Client.DISCONNECT_TIMEOUT_NEVER` に設定して、タイムアウトを無制限にすることもできます。

接続解除タイムアウトを設定すると、WebLogic Server が特定の種類の接続解除の後、T3Client　のリソース（特にワークスペース）をクリーンアップ（破棄）するまでの時間を指定できます。T3Client のソフト接続解除のタイムアウトを `DISCONNECT_TIMEOUT_NEVER` に設定すると、基本的に、T3Client は WebLogic Server が動作しているかぎり有効となります。

手順 3. 接続用プロパティの設定

WebLogic Server セッションを確立したら、WebLogic Server を介して DBMS に接続するプロセスを開始できます。接続パラメータを設定するには、JDBC Properties オブジェクトを使用します。

Properties オブジェクトには、JDBC 接続に必要なすべての情報が含まれています。接続のセットは T3Client のワークスペースにキャッシュできます。T3Client は、`getConnection()` メソッドを実行するときに、キャッシュ接続のいずれかを connection ID を使って要求できます。Connection ID は、Property を設定するだけで作成できます。指定した connection ID があれば、それが使用されます。そうでない場合は、新しい connection ID が作成され、それと一緒に JDBC 接続のすべての Property 情報が保存されます。キャッシュ接続を再利用する際に必要なものは connection ID だけです。connection ID があれば、Properties オブジェクトは無視されます。

Properties には、WebLogic JDBC クライアントがデータベースおよび WebLogic Server にアクセスする方法についての詳細が含まれています。Properties オブジェクトが構築されると、プロパティ名とその String 値という 2 つの引数を指定することによって、`put()` メソッドで任意の数のプロパティを設定できます。この例で設定しているプロパティは、WebLogic JDBC ドライバで使用されます。使用する JDBC ドライバに必要などのようなプロパティでも含めることができます。

ここでは、2 つの Properties オブジェクトを使用します。1 つ目は *dbprops* で、WebLogic Server と DBMS との間の接続のパラメータを設定します。これは 2 層接続です。

2 つ目の Properties オブジェクトは *t3props* で、WebLogic JDBC クライアントと DBMS との間に WebLogic Server が入る接続のパラメータを設定します。これを多層接続と呼びます。WebLogic Server-DBMS Properties オブジェクト（*dbprops*）は、それ自体が WebLogic JDBCClient-WebLogic Server-DBMS Properties オブジェクト（*t3props*）になり、JDBC 接続コンストラクタの引数として使用されます。

WebLogic Server と DBMS の接続（2 層接続）、および WebLogic JDBCClient と DBMS の間に WebLogic Server が入る接続（多層接続）のためのパラメータを設定するには、`java.util.Properties` オブジェクトを使用します。Properties オブジェクトは、`getConnection()` メソッドの引数として使用できます。

わかりやすくするために、Properties を、2 層接続用と多層接続用の 2 つのリストに分けてあります。

2 層接続用に設定するプロパティ

user
: DBMS にアクセスするためのユーザ名

password
: DBMS にアクセスするためのパスワード

server
: DBMS の名前 *.server* プロパティは、多層プロパティ *weblogic.t3.driverURL* で、URL の一部としてドライバの URL の後に追加して設定することでもできます。たとえば、V2 エイリアス「DEMO」

を使用する Oracle DBMS の場合、「weblogic:jdbc:oracle:DEMO」となります。

db または *database*

データベースの名前。JDBC ドライバによっては必須です。

オプションで設定できる 2 層接続用のプロパティもあります。以下のプロパティはその 1 つです。

weblogic.oci.cacheRows

（Oracle のみ）Oracle 配列フェッチのサポート。`ResultSet.next()` を最初に呼び出したとき、単一行を取り出すのではなく、行の配列を取得し、それをメモリに格納します。それ以降、`next()` を呼び出すと、メモリに格納された行が読み出されます。この操作はメモリ内の行がなくなるまで続き、行がなくなると、`next()` への呼び出しはデータベースに戻ります。

weblogic.t3.cacheRows という別のプロパティもあります。これは、WebLogic Server でレコードをバッファリングするための多層キャッシュのサイズを調整するものです。これらのプロパティには関連がありませんが、同時に使用されることもあります。

多層接続用に設定するプロパティ

weblogic.t3 または *weblogic.t3.serverURL*

weblogic.t3 を T3Client オブジェクトに設定します。T3Client は WebLogic Server 内に独自のコンテキストを持っており、それはこのオブジェクトによって定義および保守されます。プログラムで明示的な T3Client オブジェクトを必要とする場合、このプロパティを設定します。

埋め込み T3Client を使用する場合、このプロパティは**設定しません**。代わりに、プロパティ *weblogic.t3.serverURL* を設定します。これは、埋め込みクライアントが接続する WebLogic Server を指定します。

weblogic.t3.dbprops

2 層接続用の properties オブジェクト自身が多層接続用のプロパティになります。

weblogic.t3.driverClassName

WebLogic Server と DBMS との間の JDBC ドライバのクラス名。どの JDBC ドライバのクラス名でもかまいません。このプロパティによる文字列セットは、WebLogic Server で `Class.forName().newInstance()` メソッドの引数として使用されます。この例で使用するドライバは、WebLogic jDriver for Oracle ネイティブ JDBC ドライバです。ドライバのクラス名は大文字 / 小文字を区別して識別されます。すべてのクラス名と同様、このクラス名は**ドット表記**です。

weblogic.t3.driverURL

WebLogic Server と DBMS との間の 2 層 JDBC ドライバの URL（*server* プロパティとして設定されていない場合は、オプションで、データベース サーバ名）。この URL は、weblogic.t3.driverClassName プロパティにクラス名が指定されているドライバのものです（詳細については、「URL を用いた JDBC 接続プロパティの設定」を参照してください）。このプロバイダによって設定される文字列は、WebLogic Server で `DriverManager.getConnection()` メソッドの最初の引数として使用されます。この例では、WebLogic jDriver for Oracle を使って Oracle データベース「DEMO」にアクセスしています。サーバ名を提供しなかった場合、システムは環境変数（Oracle の場合は ORACLE_SID）を探します。URL は大文字 / 小文字を区別して識別しません。URL はコロンで区切ります。

WebLogic 2 層ネイティブ ドライバの URL は以下のとおりです。

- jdbc:weblogic:informix4
- jdbc:weblogic:oracle
- jdbc:weblogic.mssqlserver4

weblogic.t3.connectionID（省略可能）

名前を付けたキャッシュ JDBC 接続。T3Client が WebLogic Server に接続しているかぎり使用できる Hashtable of Connection オブジェクトを T3Client のワークスペースの中に作成できます。これらのキャッシュ接続を使用すると、T3Client が何度でも JDBC 接続を確立できます。キャッシュ connection ID には、最初に JDBC 接続を確立したときに使用したプロパティがすべて含まれています。そのため、`getConnection()` メソッドの引数として connection ID を指定すると、java.util.Properties オブジェクトで指定した他のプロパティは、connection ID が WebLogic Server に存在しない場合にだけ使用されます。

weblogic.t3.cacheRows（省略可能）

キャッシュする行の数。オプションであるこのプロパティは、T3Client とデータベースとの間の 1 回の応答でクライアントにキャッシュする行の数を設定します。行をキャッシュすると、クライアントの性能が向上します。このプロパティを使用すると、アプリケーションのキャッシュを微調整できます。

クライアントが `ResultSet.next()` を最初に呼び出したとき、WebLogic は、DBMS から行のバッチを取得し、それらを 1 回の応答でクライアント JVM に転送します。それ以降、`ResultSet.next()` を呼び出すと、WebLogic を呼び出すことなく、クライアントのメモリにある行が取り出されます。クライアントのメモリにキャッシュされている行がなくなくと、`ResultSet.next()` に対する次の呼び出しは WebLogic に渡され、WebLogic は再び行のバッチを取得してクライアントのキャッシュに送ります。

プロパティ weblogic.t3.cacheRows を使用すると、バッチに取得する行数を設定できます。このプロパティのデフォルト値は最初 10 行でしたが、リリース 2.5 で 25 行に増加されました。サイズを増減するには、このプロパティを指定します。キャッシュをオフにするには、プロパティの値を 0 に設定します。0 に設定すると、`next()` または `getXXX()` メソッドを呼び出すたびに、T3Client とデータベースとの間で必ず 1 回応答が発生します。

アプリケーションと DBMS の組み合わせによっては、WebLogic が、必ず、レコードをクライアントに直接渡さなければならない場合もあります。たとえば、アプリケーションがカーソルを処理しているとき、取得前の行データは、クライアントのカーソル位置と DBMS ドライバのカーソル位置の解釈に誤差を生じさせます。こうしたアプリケーションの場合、weblogic.t3.cacheRows プロパティをゼロ（「0」）に設定すると、必要な動作を実現できます。

クエリの結果を ResultSet に取り出す前に、接続オブジェクトでこのプロパティをリセットすることによって、cacheRows プロパティをクエリのたびに調整できます。weblogic.jdbc.t3.Connection.cacheRows() にある WebLogic の JDBC 拡張機能を使って、現在の値を取得し、新しい値に設定します。結果は、リセットされるまで現在の *cacheRow* 設定に応じてキャッシュされます。

WebLogic jDriver for Oracle を使用する場合、2 層プロパティ weblogic.oci.cacheRows を介して、Oracle の配列フェッチ機能にアクセ

スできます（この機能の詳細については、WebLogic jDriver for Oracle 用の開発者ガイドを参照してください）。このプロパティは *weblogic.t3.cacheRows* に依存しませんが、待ち時間とデータベースの負荷を相殺するために、両プロパティを一緒に使用してもかまいません。

weblogic.t3.blobChunkSize（省略可能、WebLogic jDriver for Oracle で使用）
WebLogic と WebLogic クライアントの間の BLOB ストリーム用のバッファ サイズを定義します。このプロパティは、2 層 Oracle ドライバ プロパティと連動して使用されます。

weblogic.t3.name（省略可能）
name プロパティでは接続の名前を設定できます。接続名は、Console の ManagedObject 表示で示されます。

weblogic.t3.description（省略可能）
description プロパティでは接続の簡単な説明を設定できます。この説明は、Console の ManagedObject 表示で示されます。

たとえば、顧客情報にアクセスするための接続にこれら 2 つのプロパティを追加するには、次のようにします。

```
Properties t3props = new Properties();
t3props.put("weblogic.t3.name",     "CustInfo");
t3props.put("weblogic.t3.description",
            "customer info connection");
```

この情報を設定すると、WebLogic Console の表示が有益で使いやすいものとなります。

上記以外に、もう 1 つプロパティがあります。このプロパティは接続プールから JDBC 接続にアクセスするプロパティなので、他のプロパティよりも使用頻度は高くありません。このプロパティを設定した場合、接続のその他の属性は接続プールが作成されたときに設定されるので、設定する必要がありません。このプロパティは、通常、他のプロパティがないときに使用します。

weblogic.t3.connectionPoolID（*weblogic.t3* プロパティとだけ使用）
特定の T3User のアクセス用に作成された JDBC 接続のプールを識別します。詳細およびコード例については、「接続プールの使い方」を参照してください。プールから接続を使用する場合は、WebLogic Server 起動時に（`weblogic.properties` ファイルのエントリによって）接続プールが作成されている必要があります。

パラメータは、**2 つの**接続に対して設定する必要があることを理解しておいてください。

- WebLogic Server と DBMS との間の 2 層接続
- WebLogic JDBC クライアント、WebLogic Server、および DBMS 間の多層接続

この例では、まず、Oracle 用 WebLogic ネイティブ JDBC ドライバ、WebLogic jDriver for Oracle を使って、データベース サーバ（「DEMO」）上の Oracle データベース（「mydb」）に接続する 2 層接続プロパティを設定します。

```
Properties dbprops = new Properties();
dbprops.put("user",                    "sa");
dbprops.put("password",                "");
dbprops.put("server",                  "DEMO");
dbprops.put("database",                "mydb");
```

次に、多層プロパティを設定します。多層プロパティのうち 1 つは、2 層 Properties オブジェクトです。多層プロバイダの一部として connection ID を指定しますが、この connection ID がすでに存在している接続（および Properties セット）を指定した場合、その他のプロパティは、connection ID とともに格納されているので無視されます。

```
Properties t3props = new Properties();
t3props.put("weblogic.t3", t3);
t3props.put("weblogic.t3.dbprops", dbprops);
t3props.put("weblogic.t3.driverClassName",
            "weblogic.jdbc.oci.Driver");
t3props.put("weblogic.t3.driverURL",
            "jdbc:weblogic:oracle");
t3props.put("weblogic.t3.connectionID",
            dbconnid);
t3props.put("weblogic.t3.cacheRows", "100");
t3props.put("weblogic.t3.name", "CustInfo");
t3props.put("weblogic.t3.description",
            "customer info connection");
```

ドライバのクラス名と URL のフォーマットは違います。クラス名はドットで区切り、URL はコロンで区切ります。

URL を使用した WebLogic JDBC プロパティの設定

開発環境によっては、多層データベースにプロパティを設定するとき、`java.util.Properties` オブジェクトの使用が制限される場合があります。たとえば、Powersoft の PowerJ では、Properties オブジェクトを DBMS のユーザ名とパスワードの設定にしか使用できません。WebLogic では、WebLogic JDBC

接続に必要なすべての情報を提供するために URL スキームを発展させています。URL を使用した WebLogic JDBC 接続プロパティの設定の詳細については、JDBC 接続のプロパティを設定するための URL の使い方と T3 ドライバの使い方を参照してください。

埋め込み T3Client の設定

WebLogic JDBC プログラム内で、他の目的で明示的な T3Client オブジェクトを必要としない場合は、埋め込み T3Client を使用できます。埋め込み T3Client は、作成、接続、および接続解除が自動的に行われます。埋め込み T3Client を使用するために必要な作業は、`java.util.Properties` オブジェクトに、WebLogic Server の URL を提供する別のプロパティを設定するだけです。

以下は、埋め込み T3Client を使用した簡単な例です。この例では、JDK 1.1 で使用する `weblogic.jdbc.t3.Driver` を使用します。これには、`java.sql` JDBC クラスがすべて含まれています。この例では、T3Client オブジェクトを宣言または構築しません。

```
Class.forName("weblogic.jdbc.t3.Driver").newInstance();
// DBMS へ接続するためのプロパティを設定する
Properties dbprops = new Properties();
dbprops.put("user",                  "scott");
dbprops.put("password",              "tiger");
dbprops.put("server",                "DEMO20");

Properties t3props = new Properties();
t3props.put("weblogic.t3.dbprops",          dbprops);
//WebLogic の URL を設定して埋め込み T3Client を作成する
t3props.put("weblogic.t3.serverURL","t3://localhost:7001");
t3props.put("weblogic.t3.driverClassName",
            "weblogic.jdbc.oci.Driver");
t3props.put("weblogic.t3.driverURL",
   "jdbc:weblogic:oracle");
t3props.put("weblogic.t3.cacheRows","10");

Connection conn =
  DriverManager.getConnection("jdbc:weblogic:t3", t3props);
Statement stmt = conn.createStatement();
stmt.execute("select * from empdemo");
ResultSet rs = stmt.getResultSet();

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

```
  }

  ResultSetMetaData rsmd = rs.getMetaData();

  stmt.close();
  conn.close();
}
```

また、プロパティ *weblogic.t3.connectionPoolID* を設定すると、埋め込み T3Client を、WebLogic JDBC 接続プールから JDBC 接続を使用するよう設定することができます。接続プールから JDBC 接続を使用する場合、*weblogic.t3.serverURL* プロパティとともに、プールから接続を要求するためのプロパティだけが必要です。プールの詳細については、このマニュアルの「接続プールの使い方」を参照してください。

T3Client と WebLogic のセキュリティ対策として、T3User にユーザ名とパスワードを設定する必要がある場合も埋め込み T3Client を使用できます。次に例を示します。この例は、`weblogic.properties` ファイルに、T3User「development」が、このプールからの接続を「reserve」する許可とともに追加されている接続プール「eng」へのアクセスを設定します。

```
weblogic.password.development=3Y(sf40!VmoN
weblogic.allow.reserve.weblogic.jdbc.connectionPool./
    eng=development
```

次の例は、この接続プールからの接続を埋め込み T3Client で使用する方法を示しています。

```
Properties t3props = new Properties();
t3props.put("weblogic.t3.serverURL",
        "t3://localhost:7001");
t3props.put("weblogic.t3.connectionPoolID", "eng");
t3props.put("weblogic.t3.user",             "development");
t3props.put("weblogic.t3.password",         "3Y(sf40!VmoN");
```

データベース サーバの場所など、接続のその他のプロパティはすべて、起動時に接続プールを作成する weblogic.properties ファイルのコンフィグレーション エントリによって設定されます。配布キットに付属するプロパティ ファイルには、接続プールのエントリ（コメントアウトされています）の例があります。

手順 4. DBMS との接続

WebLogic JDBC クライアントは直接データベースに接続せず、T3Client の代わりにデータベースにアクセスする WebLogic Server に接続します。2 層接続（WebLogic Server と DBMS 間）および多層接続（WebLogic JDBC クライアント、WebLogic Server、および DBMS 間）の両方に JDBC ドライバを提供する必要があります。2 層接続のクラス名および URL は、すでに説明したように Properties オブジェクトで設定します。

WebLogic JDBC クライアント、WebLogic Server、および DBMS 間の接続の場合、多層接続を管理する WebLogic JDBC ドライバのクラス名および URL を提供します。

WebLogic JDBC ドライバのクラス名は、`Class.forName().newInstance()` メソッドをクラス名 **weblogic.jdbc.t3.Driver** で呼び出します。`Class.forName().newInstance()` プロパティを呼び出すと、ドライバのクラスがロードされ登録されます。

このドライバの URL **jdbc:weblogic:t3** を `DriverManager.getConnection()` メソッドの引数として提供します。

次の例では、`Class.forName().newInstance()` メソッドを使って WebLogic JDBC ドライバを識別およびロードします。次に、WebLogic JDBC ドライバの URL および Properties オブジェクトで JDBC 接続を作成します。

```
// WebLogic JDBC のクラス名
Class.forName("weblogic.jdbc.t3.Driver").newInstance();
Connection conn =
  DriverManager.getConnection("jdbc:weblogic:t3",
                              t3props);
```

接続を確立する（つまり、JDBC 接続オブジェクトを構築する）と、WebLogic JDBC メソッドを JDBC の他の実装と同じように使用できます。

キャッシュ接続と接続プール

WebLogic は、何度も使用できる JDBC 接続を T3Clients に提供します。DBMS にログインするのは、負荷も時間もかかります。再使用できる接続があれば、DBMS に接続するための負荷は、接続を作成するときの 1 回で済みます。WebLogic Server は、何度も使用できるように、接続を開いたままにします。

再使用できる接続は 2 種類あります。名前を付けたキャッシュ JDBC 接続と接続プールです。

キャッシュ JDBC 接続は作成され、使用されてから、後で使用できるように T3Client のワークスペースに格納されます。JDBC 接続をキャッシュしておくと、接続を何度も作成するための負荷を発生させずに済みます。ただし、キャッシュ接続は、長期的にはデータベース リソースを停滞させることにもなります。JDBC 接続をキャッシュすると、その全体の状態も格納されます。

JDBC 接続のプールは、WebLogic Server が起動したときに、JDBC 接続が要求される前に作成されるか、T3Client. によって動的に作成されます。`weblogic.Admin` クラスの create_pool コマンドを使用して、動的に作成することもできます。

T3Client が WebLogic Server に接続する場合、接続をプールから取得でき、終了時に戻すことができます。接続プールを作成しておくことは、少ないリソース（データベース接続など）を複数のクライアントに割り当てる便利な方法です。すべての接続が割り当てられた場合は、接続の最大数までプールを増加できます。また、名前を付けたプールにユーザ グループを割り当てることもできます。接続プールは一度作成すると何度でも再使用できるので、WebLogic Server および DBMS が接続を作成するときの負荷を抑えることができます。

接続プールとキャッシュ接続は違います。キャッシュ JDBC 接続は特定の T3Client によって作成され、JDBC 接続に関する詳細は、その T3Client のワークスペースに格納されます。キャッシュ接続が有効なのは、T3Client の有効期間だけです。WebLogic Server がその T3Client のリソースをクリーン アップすると、キャッシュ接続は破棄されます。

一方、接続プールは、どの T3Client も使用できます。接続の有効期間は、T3Client の有効期間とは関係ありません。T3Client 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 T3Client 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。

キャッシュ JDBC 接続および接続プールの作成および使用方法について以下で詳しく説明します。これらは、特に「クライアントサイド」接続および接続プールです。別の WebLogic JDBC 拡張機能を利用すると、HTTP サーブレットや、T3Client を使用しないその他のアプリケーションに使用するためのサーバ側接続プールを作成できます。サーバサイド接続プールの詳細については、開発者ガイドの『WebLogic HTTP サーブレット プログラマーズ ガイド』を参照してください。

名前付きキャッシュ JDBC 接続の使い方

JDBC 接続を作成し、名前を付けて、WebLogic Server 上で T3Client のワークスペースにキャッシュしておくと、この JDBC 接続は再使用できます。T3Client のキャッシュ JDBC 接続は、T3Client が WebLogic Server 上にある限り存続します。このキャッシュ JDBC 接続の有効期間を無限にするには、T3Client のソフト接続解除タイムアウトを DISCONNECT_TIMEOUT_NEVER に設定します。キャッシュ JDBC 接続の有効期間を無限にすると、T3Client は、WebLogic Server 上に、WebLogic Server の有効期間の限り存続するワークスペースを持つことになります。ソフト接続解除タイムアウトを never に設定すると、`T3Client.disconnect()` メソッドを呼び出して T3Client を接続解除しても、WebLogic Server は T3Client のリソースの返還を要求しません。

このクラスに、再使用できる、名前を付けた JDBC 接続を作成します。名前を付けた JDBC 接続は、`java.util.Property`*connectionID* で識別できます。

このクラスには 2 つのメソッドがあります。JDBC 接続を作成および再使用するための static `getConnection()` メソッドと `main()` です。まず、`getConnection()` メソッドのコードについて説明します。

このメソッドは JDBC 接続オブジェクトを返します。WebLogic Server にすでに存在するプロパティ「connectionID」を提供した場合、ログイン アクセスなどのその他のプロパティはすべて無視され、データベースへの接続を再使用するのに必要なすべての情報を含んだキャッシュ JDBC 接続が使用されます。

```
static Connection getConnection(T3Client t3, String dbconnid)
  throws Exception
{

  // connectionID を指定すると、WebLogic Server にその connectionID が
  // 存在しない場合にのみ、
  // 他のプロパティが使用される。

  // connectionID が存在する場合、他の値は無視される
  Properties dbprops = new Properties();

  // 2 層プロパティを設定する
  dbprops.put("user",                          "scott");
  dbprops.put("password",                      "tiger");
  dbprops.put("server",                        "DEMO");

  Properties t3props = new Properties();
  // connectionID を含む多層プロパティを設定する
  t3props.put("weblogic.t3",                   t3);
  t3props.put("weblogic.t3.dbprops",           dbprops);
```

```
    t3props.put("weblogic.t3.driverClassName",
                "weblogic.jdbc.oci.Driver");
    t3props.put("weblogic.t3.driverURL",
                "jdbc:weblogic:oracle");
    // dbconnid がキャッシュされている場合、それまでのプロパティは
    // すべて無視される。接続はすぐに得られる
    t3props.put("weblogic.t3.connectionID",   dbconnid);

    Class.forName("weblogic.jdbc.t3.Driver").newInstance();
    return  DriverManager.getConnection("jdbc:weblogic:t3",
                                        t3props);
  }
```

次のコード例では、以下の手順を実行します。

1. T3Client を作成します。
2. WebLogic Server に接続します。
3. このクライアント用のソフト接続解除タイムアウトを無限にします。
4. 後で T3Client がこのワークスペースに戻れるように、ワークスペースの ID を保存します。T3Client オブジェクトを新しく作成する場合は、引数としてクライアントに名前（「mike」など）を提供することもできます。

```
  T3Client t3 = new T3Client("t3://localhost:7001");
  t3.connect();
t3.setSoftDisconnectTimeoutMins(T3Client.DISCONNECT_TIMEOUT_NEVER
);
  String wsid = t3.services.workspace().getWorkspace().getID();
```

次に、`getConnection()` メソッドを「myconn」という名前を指定して呼び出すことによって、Oracle データベースへ接続するための名前を付けた JDBC 接続を作成します。

```
  System.out.println("Logging into database and " +
                     "saving session as myconn");
  Connection conn = getConnection(t3, "myconn");
```

ここで、T3Client を接続解除して、null に設定します。ソフト接続解除タイムアウトが never なので、WebLogic Server は、このクライアントのワークスペースおよび名前を付けた JDBC 接続を含めたリソースを保持します。

```
  t3.disconnect();
  t3 = null;
```

次に、最初のクライアントのワークスペース ID を持つ新しい T3Client を作成することによって、WebLogic Server に再接続します。これは、T3Client を最初のクライアントで作成したワークスペースに再リンクします。そのワークスペースには、「myconn」という名前でキャッシュした JDBC 接続が格納されています。

これで、`getConnection()` メソッドを「myconn」という名前を指定して呼び出すことによって、Oracle データベースへの接続を再開できます。接続に必要なすべてのパラメータはすでに存在しています。任意のデータベース処理を行って JDBC 接続オブジェクトを閉じます。

```
t3 = new T3Client("t3://localhost:7001", wsid);
t3.connect();

for (int j = 0; j < 5; j++) {
  System.out.println("Reestablishing database connection");
  conn = getConnection(t3, "myconn");

  System.out.println("Performing query");
  QueryDataSet qds = new QueryDataSet(conn, "select * from emp");

  qds.fetchRecords();
  System.out.println("Record count = " + qds.size());
  qds.close();
}
 conn.close();
```

T3Client のリソースを使用した作業が終了したら、T3Client のソフト接続解除タイムアウトを zero にすることで、WebLogic Server がそれらのリソースの返還を要求できるようにすることができます。これにより、T3Client が `disconnect()` メソッドを呼び出すと、直ちにクリーンアップが実行されます。次に例を示します。

```
t3.setSoftDisconnectTimeoutMins(0);
  t3.disconnect();
```

接続プールの使い方

接続のもう 1 つの方法は、JDBC 接続のプールを作成するものです。T3User はプールから接続を要求できます。接続プールを定義するには、`weblogic.properties` ファイルで定義するか、「スタートアップ」接続を呼び出すか、または T3Client アプリケーション内から、実行中の WebLogic Server に「動的に」接続プールを作成します。

JDBC 接続のプールを作成しておくと、T3Client はすでに開いている接続にすぐにアクセスすることができます。プールの接続は、グループのメンバーによって共有されるので、各 DBMS ユーザが新しい接続を開く負荷を発生させずに済みます。

スタートアップ接続プールの作成

スタートアップ接続プールは、weblogic.properties ファイルで宣言されます。WebLogic Server は、WebLogic の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。

スタートアップ接続プールは、weblogic.properties ファイルの以下のパターンの後のエントリで定義します。配布キットに付属のプロパティ ファイルの「JDBC Connection Pool Management:」という見出しの下に例を挙げて説明されています。

```
weblogic.jdbc.connectionPool.VirtualName=\
         url=JDBC driver URL,\
         driver=full package name for JDBC driver,\
         loginDelaySecs=seconds between each login attempt,\
        initialCapacity=initial number of connections in the pool,\
         maxCapacity=max number of connections in the pool,\
        capacityIncrement=number of connections to add at a time,\
         allowShrinking=true to allow shrinking,\
         shrinkPeriodMins=interval before shrinking,\
         testTable=name of table for connection test,\
         refreshTestMinutes=interval for connection test,\
         testConnsOnReserve=true to test connection at reserve,\
 testConnsOnRelease=true to test connection at release,\
         props=DBMS connection properties

weblogic.allow.reserve.weblogic.jdbc.connectionPool.name=\
         T3Users who can use this pool
weblogic.allow.reset.weblogic.jdbc.connectionPool.name=\
         T3Users who can reset this pool
weblogic.allow.shrink.weblogic.jdbc.connectionPool.name=\
         T3Users who can shrink this pool
```

ユーザが提供する情報は、赤で示されています。省略できない情報は必須と記されています。必須の値が提供されなかった場合、WebLogic Server は起動時に例外を送出します。

このプロパティの引数の簡単な説明を以下に示します。

name

（必須）接続プールの名前。このプールから JDBC 接続にアクセスする場合、T3Client クラスを記述するときにこの名前を使用します。

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。

driver

（必須）WebLogic Server と DBMS との間の接続に使用する JDBC 2 層ドライバの絶対パス名。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。

loginDelaySecs

（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。

initialCapacity

（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 capacityIncrement に設定されている値になります。

maxCapacity

（必須）プールの最大サイズ。

capacityIncrement

（必須）プールの容量を増加するサイズ。*initialCapacity* および *capacityIncrement* は、Java Vector のように動作し、初期割り当て（「capacity」）があり、プールの *maxCapacity* まで、必要に応じて capacityIncrement ずつ増加されます。

allowShrinking

（省略可能、3.1 から導入）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、*shrinkPeriodMins* 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。allowShrinking は下位互換のため、デフォルトで false に設定されます。

shrinkPeriodMins

（省略可能、3.1 から導入）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。このプロパティは、

allowShrinking が true のとき使用します。shrink period のデフォルト値は 15 分で、最小値は 1 分です。

testTable

（refreshTestMinutes、testConnsOnReserve、または testConnsOnRelease を設定する場合にのみ必須。4.0 から導入）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` *testTable* がテストに使用されます。*testTable* 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、*testTable* 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。

refreshTestMinutes

（省略可能、4.0 から導入）このプロパティは、*testTable* 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。

自動リフレッシュを有効にするには、*refreshTestMinutes* 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 *refreshTestMinutes* 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、*testTable* を既存のデータベース テーブルの名前に設定します。自動リフレッシュ機能を有効にするには、refreshTestMinutes と testTable の両方のプロパティを設定しなければなりません。

testConnsOnReserve

（省略可能、4.0.1 から導入）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。この機能を使用するには、testTable パラメータを設定する必要があります。

testConnsOnRelease

（省略可能、4.0.1 から）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、testTable パラメータを設定する必要があります。

props

（必須）**ユーザ名、パスワード、サーバ**などの、データベースに接続するためのプロパティ。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。

allow

この属性は、3.0 で評価されませんでした。すでに説明したように、「reserve」および「reset」パーミッションを使って接続プールへのアクセスを設定します。

次の例は、WebLogic 配布キットに付属の `weblogic.properties` ファイルから取ったもので、3 つの T3User（Guest、Joe、Jill）がアクセスできる「eng」という名前の接続プールを作成します。この例は最小 4、最大 10 の Oracle データベースへの JDBC 接続を、ユーザ名「SCOTT」、パスワード「tiger」、サーバ名「DEMO」で割り当てます。WebLogic Server は、飽和状態のネットワークでロードされず、DBMS からログインを拒否されることがないよう、接続を試みるごとに 1 秒間スリープ状態になります。接続プールは、プール内の接続が 15 分以上使用されなければ、接続数を 4 つに戻します。10 分ごとに未使用の接続がテストされ、古くなってしまったものはリフレッシュされます。

```
weblogic.jdbc.connectionPool.eng=\
       url=jdbc:weblogic:oracle,\
       driver=weblogic.jdbc.oci.Driver,\
       loginDelaySecs=1,\
       initialCapacity=4,\
       maxCapacity=10,\
       capacityIncrement=2,\
       allowShrinking=true,\
       shrinkPeriodMins=15,\
       refreshTestMinutes=10,\
       testTable=dual,\
       props=user=SCOTT;password=tiger;server=DEMO

weblogic.allow.reserve.weblogic.jdbc.connectionPool.eng=\
       guest,joe,jill
weblogic.allow.reset.weblogic.jdbc.connectionPool.eng=\
       joe,jill
weblogic.allow.shrink.weblogic.jdbc.connectionPool.eng=\
       joe,jill
```

ユーザ名のパスワードが null の場合、接続プール登録のパスワードに空の文字列を入力しないで、空白のままにしておいてください。次の例は、WebLogic 管理者ガイドの properties から取ったものです。

```
weblogic.jdbc.connectionPool.eng=\
       url=jdbc:weblogic:oracle,\
       driver=weblogic.jdbc.oci.Driver,\
       loginDelaySecs=1,\
       initialCapacity=4,\
       capacityIncrement=2,\
       maxCapacity=10,\
       props=user=sa;password=;server=demo
weblogic.allow.reserve.weblogic.jdbc.connectionPool.eng=guest,joe
,jill
```

動的接続プールの作成

WebLogic リリース 4.0 からの JNDI ベースの API では、T3Client アプリケーション内で接続プールを作成することができます。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。

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。

`weblogic.properties` ファイルのプロパティ、`weblogic.allow.admin.weblogic.jdbc.connectionPoolcreate` は、動的接続プールを作成できるユーザを指定します。このプロパティが設定されていない場合、「system」ユーザだけが動的接続プールを作成できます。

たとえば、次のプロパティは、ユーザ「joe」および「jane」に動的接続プールの作成を許可します。

```
weblogic.allow.admin.weblogic.jdbc.connectionPoolcreate=joe,jane
```

動的接続プールに ACL を作成するには、`weblogic.allow.reserve.`*ACLname* エントリおよび `weblogic.allow.admin.`*ACLname* エントリを `weblogic.properties` ファイルに追加します。たとえば、以下の 2 つのプロパティは、「dynapool」という ACL を定義して、「everyone」グループのすべてのユーザに接続プールの使用を許可し、ユーザ「joe」および「jane」に接続プールの管理を許可します。

```
weblogic.allow.admin.dynapool=joe,jane
weblogic.allow.reserve.dynapool=everyone
```

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。

T3 アプリケーションの中で動的接続プールを作成するには、まず WebLogic JNDI プロバイダへの初期 JNDI コンテキストを取得し、次に「weblogic.jdbc.common.JdbcServices」をルックアップします。次の例で、その方法を示します。

```
Hashtable env = new Hashtable();

env.put(java.naming.factory.initial,
                "weblogic.jndi.WLInitialContextFactory");
// WebLogic Server の URL
env.put(java.naming.provider.url, "t3://localhost:7001");
env.put(java.naming.security.credentials,
         new T3User("joe", "joez_secret_wrdz"));

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");
```

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 オブジェクトに渡します。Properties オブジェクトには、`weblogic.properties` ファイルで接続プールを作成するために使用したプロパティと同じプロパティが含まれています。ただし、「aclName」プロパティだけは、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 `weblogic.properties` ファイルの「dynapool」ACL が使用されることが指定されています。

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

 try {
```

```
  // 接続プールのプロパティを設定する。
  // プロパティは、weblogic.properties ファイルで
  // 起動接続プールの定義に使用されるものと同じ
  Properties poolProps = new Properties();

  poolProps.put("poolName",         "eng2");
  poolProps.put("url",              "jdbc:weblogic:oracle");
  poolProps.put("driver",           "weblogic.jdbc.oci.Driver");
  poolProps.put("initialCapacity", "1");
  poolProps.put("maxCapacity",      "10");
  poolProps.put("props",            "user=SCOTT;
                                    password=tiger;server=DEMO");
  poolProps.put("aclName",          "dynapool");  // 使用する ACL

  // 同じ名前の既存のプールが存在する場合、作成は失敗する
  jdbc.createPool(poolProps);
}
catch (Exception e) {
   system.out.Println("Error creating connection pool eng2.");
}
finally { // JNDI コンテキストをクローズする
       ctx.close();
}
```

接続プールからの接続の取得

接続プールから接続を使用する手順は、JDBC 接続を開く場合とほぼ同じです。JDBC ドライバ クラスは weblogic.jdbc.t3Client.Driver で、接続 URL は「jdbc:weblogic:t3」です。使用する接続プールを指定するには、java.util.Properties オブジェクトを作成し、`weblogic.t3.connectionPoolID` というプロパティを接続プールの名前に設定します。

接続プールから接続を取得する。クライアント アプリケーションで接続プールから接続を使用するには、java.util.Properties オブジェクトを作成し、`weblogic.t3.connectionPoolID` というプロパティを WebLogic Server の `weblogic.properties` ファイルで作成した接続プールの名前に設定します。

次の簡単な例では、T3Client を作成し、Properties オブジェクトを設定してから、接続プール「eng」から接続を開きます。この接続プールの `weblogic.properties` ファイル エントリはすでに説明したとおりです。

```
T3Client t3 = new T3Client("t3://bigbox:7001");
  t3.connect();

  // 設定が必要なプロパティは
  // T3Client と connectionPoolID の 2 つだけであることに注意する
```

```
    Properties t3props = new Properties();
    t3props.put("weblogic.t3", t3);
    t3props.put("weblogic.t3.connectionPoolID", "eng");

    Class.forName("weblogic.jdbc.t3.Driver").newInstance();
    Connection conn =
      DriverManager.getConnection("jdbc:weblogic:t3",
                                  t3props);
    // 任意のデータベース処理を行う
    QueryDataSet qds = new QueryDataSet(conn, "select * from emp");
    qds.fetchRecords();
    System.out.println("Record count = " + qds.size());
    qds.close();
    // 接続を解放する
    conn.close();
    // クライアントの接続を解除する
    t3.disconnect();
  }
```

JDBC 接続はこのクライアントによって解放され、クライアントが接続を解除する前にプールに戻されます。JDBC 接続が接続プールに戻されると、未処理の JDBC トランザクションはロールバックされてクローズされます。

プール接続が使用できるようになるまで待機する。プール内のすべての接続が使用されている場合、クライアントは、デフォルトでは、接続が使用可能になるまで待機します。この動作は、以下の 2 つの方法で変更できます。

- 待機を無効にします。どの接続も使用できない場合、`DriverManager.getConnection()` は、すぐに例外処理します。
- 接続を待つ秒数を指定します。指定秒内で接続が使用できるようにならなかった場合、`DriverManager.getConnection()` が例外処理します。

待機を無効にするには、`DriverManager.getConnection()` に渡す Properties オブジェクトの `weblogic.t3.waitForConnection` プロパティを「false」に設定します。

```
Properties t3props = new Properties();
t3props.put("weblogic.t3", t3);
t3props.put("weblogic.t3.connectionPoolID", "eng");
t3props.put("weblogic.t3.waitForConnection", "false");

Class.forName("weblogic.jdbc.t3.Driver").newInstance();
Connection conn =
   DriverManager.getConnection("jdbc:weblogic:t3",
                               t3props);
```

待機を無効にすると、DriverManager.getConnection() の呼び出しは、使用できる接続がないとすぐに例外を送出します。

使用できる接続を待つ時間を指定するには、weblogic.t3.waitSecondsForConnection プロパティで任意の時間を設定します。次の例では、15 秒待ちます。

```
Properties t3props = new Properties();
t3props.put("weblogic.t3", t3);
t3props.put("weblogic.t3.connectionPoolID", "eng");
t3props.put("weblogic.t3.waitSecondsForConnection", "15");

Class.forName("weblogic.jdbc.t3.Driver").newInstance();
Connection conn =
   DriverManager.getConnection("jdbc:weblogic:t3",
                               t3props);
```

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.shrinking

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

weblogic.jdbc.t3.Connection

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。

1 つの接続だけをリフレッシュするには、weblogic.jdbc.t3.Connection クラスの refresh() メソッドを使用します。このメソッドを呼び出すと、接続時の文および Resultset はすべて失われるとともに、接続を開くときに比較的高い負荷がアプリケーションにかかります。したがって、接続が不良になっていたというエラーを受け取った場合には、その接続だけをリフレッシュすることをお勧めします。

JDBC 接続を (weblogic.jdbc.t3.Connection) として明示的にキャストする必要があります。その他には、Connection クラスの refresh() メソッドを、接続に対して reset() メソッドを使用するのと同じように使用する方法があります。まず、接続自体が有効であるかぎり成功する保証があるアクションを、接続を使用して実行します。例外を取得して、refresh() メソッドを呼び出します。

次に例を示します。JDBC への WebLogic 拡張機能を利用するために、最後の行で JDBC 接続を weblogic.jdbc.t3.Connection としてキャストしています。

```
T3Client t3 = new T3Client("t3://localhost:7001");
t3.connect();

Properties t3props = new Properties();
t3props.put("weblogic.t3",                    t3);
t3props.put("weblogic.t3.connectionPoolID", "eng");

Class.forName("weblogic.jdbc.t3c.Driver").newInstance();
Connection conn =
  DriverManager.getConnection("jdbc:weblogic:t3client",
                              t3props);

try {
  Statement stmt = conn.createStatement();

  // この SQL は Oracle DMBS への適切な接続については
  // 成功が保証されている
  ResultSet rs = stmt.executeQuery("select 1 from dual");
```

```
    if (rs != null) {
      while (rs.next()) {;}
    }

    rs.close();
    stmt.close();
    conn.close();
  }

  catch(SQLException e) {

    // JDBC Connection を weblogic.jdbc.t3.Connection
    // にキャストし、それに対して
    // refresh() メソッドを呼び出す
    ((weblogic.jdbc.t3.Connection)conn).refresh();
  }
```

1 つのプール接続のリフレッシュ

weblogic.jdbc.common.JdbcServicesDef

プール内の 1 またはそれ以上の接続が古くなった場合、プール内の 1 つの接続だけをリフレッシュすることも、接続プール全体をリセットすることもできます。たとえば、WebLogic Server がアクティブに接続プールをサポートしているときに、DBMS がダウンした場合などです。WebLogic Server 4.0 からの接続プール自動リフレッシュ機能を使っても、接続を周期的にテストして、リフレッシュすることができます。

プールをリセットするのが適切なのは、特定の場合に限定されます。この機能は、ユーザ プログラムのルーチンとして使用できません。通常、接続プールのリセットが必要になるのは、DBMS がダウンして、プール内の接続が実行不可能になった場合です。DBMS がアップして使用可能になったことを確認する前にプールをリセットしようとすると、例外が送出されます。プールのリセットは、管理特権を持ったユーザが実行する特別な操作です。

接続プールをリセットするには、いくつかの方法があります。

- 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。Admin コマンドの詳細については、WebLogic 『管理者ガイド』の「WebLogic Server の運用とメンテナンス」を参照してください。

- 接続プールの実行可能性を周期的にチェックして、必要であれば自動的にリフレッシュするには、WebLogic Events クラス ActionRefreshPool を使用します。ActionRefreshPool は、WebLogic の公開 API の一部です。このクラスを実行するにも、WebLogic Server の管理特権が必要です。このクラスをスタートアップ クラスとして登録することもできます。**接続プールの接続の実行可能性を判断するには、このメソッドを使用するのが最も簡単な方法です。**
- 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。サンプル コードを使用して、`reset()` メソッドの使用方法を説明します。

以下は、`reset()` メソッドを使用したプールのリセットの例です。

1. 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。
2. SQLException を取得します。
3. catch ブロックの中で `reset()` メソッドを呼び出します。

次の簡単なコード例は、Oracle DBMS への 5 つの接続からなるプールである、デモ JDBC 接続プール「eng」をテストします。

```
String poolID = "eng";

T3Client t3 = new T3Client("t3://localhost:7001");
t3.connect();

Properties t3props = new Properties();
t3props.put("weblogic.t3",                    t3);
t3props.put("weblogic.t3.connectionPoolID", poolID);

Class.forName("weblogic.jdbc.t3.Driver").newInstance();
```

```
Connection conn =
  DriverManager.getConnection("jdbc:weblogic:t3client",
                              t3props);

try {

  // 接続が使用可能でない場合、その接続を通じて
  // 何らかの処理を試みると、SQLException が発生する
  Statement stmt = conn.createStatement();

  // この SQL は Oracle DMBS への適切な接続については
  // 成功が保証されている
  ResultSet rs = stmt.executeQuery("select 1 from dual");

  // 任意のデータベース処理を行う
  if (rs != null) {
    while (rs.next()) {;}
  }

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

  // 接続を解放してプールに返す
  conn.close();
}

// try ブロックが失敗する場合はプールをリセットする。このメソッドを
// 呼び出すプログラムでは管理者特権が必要であることに注意する。
// データベースが再び使用可能になったことを確認するまで
// プールのリセットを試みないように注意する

catch(SQLException e) {
  t3.services.jdbc().resetPool(poolID);
}
```

WebLogic レルム内での接続プール用 ACL のセットアップ

`weblogic.jdbc.connectionPool`

`weblogic.jdbc.connectionPool.`*poolID*

WebLogic は、WebLogic レルムでセットアップされた ACL を通して、JDBC 接続プールのような内部リソースへのアクセスを制御します。WebLogic レルムの ACL のエントリは、`weblogic.properties` ファイルのプロパティとしてリストされています。

接続プールの JDBC 接続に、パーミッション「reserve」、「reset」、および「shrink」を設定するには、プロパティ ファイルにプロパティを入力します。ACL「weblogic.jdbc.connectionPool」にパーミッションを設定すると、すべての接続プールへのアクセスが制限されます。「weblogic.jdbc.connectionPool.poolID」という ACL にエントリを追加することによって、他のユーザのパーミッションを追加します。これは、接続プール *poolID* へのアクセスを制御します。特別なユーザ *system* は、他のパーミッションがどのように設定されているかにかかわらず、常に、各 ACL のパーミッション「reserve」、「reset」、および「shrink」を持っています。

例：

```
weblogic.allow.reserve.weblogic.jdbc.connectionPool.eng=margaret,
joe,maryweblogic.allow.reset.weblogic.jdbc.connectionPool./
eng=sysMonitorweblogic.allow.shrink.weblogic.jdbc./
connectionPool.eng=sysMonitor
```

下位互換のため、旧スタイルのプロパティ構文を使用して「reserve」パーミッションを与えることもできます。それには、プロパティ weblogic.jdbc.connectionPool.*poolID*=allow= に userlist を設定します。WebLogic では、旧スタイルのプロパティをいつまでサポートするかわからないため、新しい機能を反映するには、プロパティ ファイルをできるだけ早くアップグレードすることをお勧めします。

レコードの挿入、更新、および削除

データベースに対してクエリを実行するには、JDBC 文、または JDBC 接続のコンテキストで作成された JDBC 文のサブクラスのいずれかを使用します。クエリの結果は、JDBC ResultSet に格納されます。データベースの行にアクセスするには、ResultSet クラスの `next()` メソッドおよび `getXXX()` メソッドを使用します。

次の例では、まず、Employee テーブルに 10 のレコードを挿入します。PreparedStatement を JDBC PreparedStatement「?」構文とともに使用します。レコードへの更新は、PreparedStatement クラスから `setInt()` メソッドおよび `setString()` メソッドを使用します。

```
String inssql = "insert into emp (empno, empname) values (?, ?)";

 PreparedStatement pstmt = conn.prepareStatement(inssql);
 for (int j = 0; j < 10; j++) {
   pstmt.setInt(1, i);
   pstmt.setString(2, "John Smith");
```

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

値を挿入したら、データベース クエリを実行し、ResultsSet.next() メソッドで結果を調べることによって、挿入された内容を確認できます。ResultSet および Statement オブジェクトは、それらがインスタンス化されたときとは逆の順序でクローズします。

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.execute("select empno, empname from emp");

while (rs.next())  {
  System.out.println("Value = " + rs.getString("empno"));
  System.out.println("Value = " + rs.getString("empname"));
}
rs.close();
stmt.close();
```

次に、別の PreparedStatement を使用して、挿入したレコードを更新します。

```
String updsql = "update emp set empname = ? where empno = ?";
PreparedStatement pstmt1 = conn.prepareStatement(updsql);

for (int j = 0; j < 10; j++) {
  pstmt1.setInt(2, j);
  pstmt1.setString(1, "Person" + j);
  pstmt1.executeUpdate();
}
pstmt1.close();
```

次の例では、挿入したレコードを削除します。ここでも PreparedStatement と setInt() メソッドを使用して、削除するレコードを選択します。

```
String delsql = "delete from emp where empno = ?";
PreparedStatement pstmt2 = conn.prepareStatement(delsql);

for (int j = 0; j < 10; j++) {
  pstmt2.setInt(1, j);
  pstmt2.executeUpdate();
}
pstmt2.close();
```

各 Statement および PreparedStatement オブジェクトは、使い終わったら閉じてください。

ストアド プロシージャおよび関数の作り方と使い方

WebLogic JDBC を使用すると、ストアド プロシージャおよび関数の作成、使用、および削除を行えます。パラメータを設定するには、CallableStatement オブジェクト（PreparedStatement のサブクラス）を JDBC PreparedStatement「?」構文を使用します。

次の例では、いくつかのストアド プロシージャおよび関数をデータベースから削除します。この処理には JDBC 文を使用し、終わった後は閉じます。

```
Statement stmt = conn.createStatement();
try {stmt.execute("drop procedure proc_squareInt");
  }  catch (SQLException e) {;}
try {stmt.execute("drop procedure func_squareInt");
  }  catch (SQLException e) {;}
try {stmt.execute("drop procedure proc_getResults");
  } catch (SQLException e) {;}

stmt.close();
```

ストアド プロシージャまたは関数を作成するには、JDBC Statement クラスを使用し、それらを実行するには、JDBC CallableStatement クラスを使用します。ストアド プロシージャの入力パラメータは、JDBC の `IN` パラメータにマップされており、`setInt()` などの `CallableStatement.setXXX()` メソッドと JDBC PreparedStatement「?」構文で使われます。ストアド プロシージャの出力パラメータは、JDBC の `OUT` パラメータにマップされており、`CallableStatement.registerOutParameter()` メソッドと JDBC PreparedStatement「?」構文で使われます。`IN` と `OUT` の両方のパラメータを使って、`setXXX()` と `registerOutParameter()` の呼び出しが両方とも同じパラメータ番号で行われるようにしてもかまいません。Sybase DBMS の場合、JDBC キーワード `CALL` が、SQL Server キーワード `EXECUTE` の代わりに使用されます。詳細については、Sybase のマニュアルを参照してください。

次に、JDBC 文を使用して、整数を 2 乗する Sybase ストアド プロシージャを作成します。ストアド プロシージャを作成したら、CallableStatement を使用して実行します。出力パラメータを設定するには、`registerOutParameter()` メソッドを使用します。

```
Statement stmt1 = conn.createStatement();
stmt1.execute("create procedure proc_squareInt " +
              "(@field1 int, @field2 int output) as " +
"begin select @field2 = @field1 * @field1 end");
stmt1.close();

CallableStatement cstmt1 =
```

```
    conn.prepareCall("{call proc_squareInt(?, ?)}");

  cstmt1.registerOutParameter(2, Types.INTEGER);
  for (int i = 0; i < 10; i++) {
    cstmt1.setInt(1, i);
    cstmt1.execute();
    System.out.println(i + " " + cstmt1.getInt(2));
  }
  cstmt1.close();
```

次のコード例は、整数の 2 乗を返す Sybase ストアド関数を作成する方法を示します。CallableStatement でストアド関数を実行し、`registerOutParameter()` で戻り値を登録します。

```
  Statement stmt2 = conn.createStatement();
 stmt2.execute("create procedure func_squareInt (@field2 int) as
" +
                "begin return @field1 * @field1 end");
  stmt2.close();

  CallableStatement cstmt2 =
    conn.prepareCall("{? = call func_squareInt(?)}");

  cstmt2.registerOutParameter(1, Types.INTEGER);
  for (int i = 0; i < 10; i++) {
    cstmt2.setInt(2, i);
    cstmt2.execute();
    System.out.println(i + " " + cstmt2.getInt(1));
  }
  cstmt2.close();
```

次のコード例は、SQL クエリの結果を返す Sybase ストアド プロシージャを作成する方法を示します。CallableStatement でストアド プロシージャを実行し、ResultSet に結果を取得します。

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

```
  Statement stmt3 = conn.createStatement();
  stmt3.executeUpdate("create procedure proc_getResults as " +
                      "begin select name from sysusers \n" +
      "select gid from sysusers end");
  stmt3.close();

  CallableStatement cstmt3 =
    conn.prepareCall("{call proc_getResults()}");

  boolean hasResultSet = cstmt3.execute();
  while (hasResultSet) {
    ResultSet rs = cstmt3.getResultSet();
    while (rs.next())
      System.out.println("Value: " + rs.getString(1));
```

```
      rs.close();
      hasResultSet = cstmt3.getMoreResults();
    }
    cstmt3.close();
```

次のコード例は、Statement および CallableStatement オブジェクトを使用して Oracle ストアド プロシージャを作成し、呼び出す方法を示しています。このプロセスは、Sybase プロシージャとほぼ同じです。

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

    CallableStatement cstmt1 =
      conn.prepareCall("BEGIN proc_squareInt(?, ?); END;");

    cstmt1.registerOutParameter(2, Types.INTEGER);
    for (int k = 0; k < 100; k++) {
      cstmt1.setInt(1, k);
      cstmt1.execute();
      System.out.println(k + " "
                         + cstmt1.getInt(1)
 + " " + cstmt1.getInt(2));
    }
    cstmt1.close();
```

最後のサンプル コードでは、Statement および CallableStatement オブジェクトを使用して Oracle ストアド関数を作成し、使用する方法を示しています。

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

      // ストアド関数を使用する
    CallableStatement cstmt2 =
      conn.prepareCall("BEGIN ? := func_squareInt(?); END;");

    cstmt2.registerOutParameter(1, Types.INTEGER);
    for (int k = 0; k < 100; k++) {
      cstmt2.setInt(2, k);
      cstmt2.execute();
      System.out.println(k + " "
                         + cstmt2.getInt(1)
 + " " + cstmt2.getInt(2));
```

```
    }
    cstmt2.close();
```

最後の手順 . 接続のクローズと T3Client の接続解除

他のすべての JDBC オブジェクトと同様に、接続も close() する必要があります。データベースへのログインが失敗した場合でも同じです。閉じておかないと、ログインの最大数を超えてしまう可能性があります。また、WebLogic JDBCClient を WebLogic Server から disconnect() する必要もあります。

finally ブロックの中の try ブロックで close() メソッドおよび disconnect() メソッドを呼び出して、適切な例外を取得します。finally ブロック内でこれらのメソッドを呼び出すと、メインの try ブロックが例外を送出して完了しない場合でも、メソッドは確実に実行されます。次に例を示します。

```
finally {
  if (conn != null)
    try { conn.close();   } catch (SQLException sqe)  {}
  if (t3   != null)
    try { t3.disconnect();} catch (Exception e)       {}
}
```

クライアントが disconnect() メソッドを使って接続解除すると、WebLogic Server は EOF 例外を含んだダンプ スタックを行います。disconnect() で Netscape サーバ ログに現れる EOF 例外は無視してください。

コードのまとめ

```
package examples.jdbc;

import java.sql.*;
import weblogic.db.jdbc.*;
import weblogic.common.*;

import java.util.Properties;

public class t3client1 {

  public static void main(String argv[]) {

    T3Client t3 = null;
    Connection conn = null;
    try {
      t3 = new T3Client("t3://bigbox:7001");
      t3.connect();

      Properties dbprops = new Properties();
```

```
      dbprops.put("user",                  "scott");
      dbprops.put("password",              "tiger");
      dbprops.put("server",              getParameter("server"));

      Properties t3props = new Properties();
      t3props.put("weblogic.t3",           t3);
      t3props.put("weblogic.t3.dbprops",   dbprops);

      // ドライバのクラス名と URL のフォーマットが
      // 異なることに注意する。クラス名はドットで区切り、
      // URL はコロンで区切る
      t3props.put("weblogic.t3.driverClassName",
                  "weblogic.jdbc.oci.Driver");
      t3props.put("weblogic.t3.driverURL",
                  "jdbc:weblogic:oracle");
      t3props.put("weblogic.t3.cacheRows",
                  getParameter("cacheRows"));

      Class.forName("weblogic.jdbc.t3.Driver").newInstance();
      conn = DriverManager.getConnection("jdbc:weblogic:t3client",
                                         t3props);

      // 一連のレコードを挿入する
      String inssql = "insert into emp (empno, empname) values (?,
?)";
      PreparedStatement pstmt = conn.prepareStatement(inssql);
      for (int j = 0; j < 10; j++)
        pstmt.setInt(1, i);
pstmt.setString(2, "John Smith");
pstmt.execute();
      }
      pstmt.close();

      Statement stmt = conn.createStatement();
      ResultSet rs = stmt.executeQuery("select empno, " +
                                       "empname from emp");

      while (rs.next())  {
        System.out.println("Value = " + rs.getString("empno"));
        System.out.println("Value = " + rs.getString("empname"));
      }

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

      // 一連のレコードを更新する
      String updsql = "update emp set empname = ? where empno = ?";



      PreparedStatement pstmt1 = conn.prepareStatement(updsql);

      for (j = 0; j < 10; j++) {
        pstmt1.setInt(2, j);
        pstmt1.setString(1, "Person" + j);
        pstmt1.executeUpdate();
```

```
    }
    pstmt1.close();

    // 一連のレコードを削除する
    String delsql = "delete from emp where empno = ?";
    PreparedStatement pstmt2 = conn.prepareStatement(delsql);

    for (int j = 0; j < 10; j++) {
      pstmt2.setInt(1, j);
      pstmt2.executeUpdate();
    }
    pstmt2.close();

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

    // ストアド プロシージャを使用する
    CallableStatement cstmt1 =
      conn.prepareCall("BEGIN proc_squareInt(?, ?); END;");

    cstmt1.registerOutParameter(2, Types.INTEGER);
    for (int k = 0; k < 100; k++) {
      cstmt1.setInt(1, k);
      cstmt1.execute();
      System.out.println(k + " "
                 + cstmt1.getInt(1)
+ " " + cstmt1.getInt(2));
    }
    cstmt1.close();

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


 "END func_squareInt;");
     stmt2.close();

    // ストアド関数を使用する
    CallableStatement cstmt2 =
      conn.prepareCall("BEGIN ? := func_squareInt(?); END;");

    cstmt2.registerOutParameter(1, Types.INTEGER);
    for (int k = 0; k < 100; k++) {
      cstmt2.setInt(2, k);
      cstmt2.execute();
```

```
        System.out.println(k + " "
                        + cstmt2.getInt(1)
    + " " + cstmt2.getInt(2));
        }
        cstmt2.close();
      }
      finally {
        if (conn != null)
            try {conn.close();} catch (SQLException sqe) {}
        if (t3   != null)
            try {t3.disconnect();} catch (Exception e) {}
      }
    }
}
```

WebLogic JDBC のその他の機能

WebLogic は、WebLogic jDriver および WebLogic JDBC にいくつかの機能を提供し、それぞれのデータベースが持つ特長を活かします。2 層ドライバにサポートされているすべての機能、および多層使用に固有のその他の機能は、WebLogic JDBC の多層環境で使用することができます。

Oracle リソースの待機

weblogic.jdbc.t3.Connection

リリース 2.5 から、WebLogic は、Oracle の `oopt()` C 関数をサポートしています。これは、リソースが使用可能になるまでクライアントが待機できるようにする機能です。Oracle C 関数は、要求されたリソースが使用可能でない場合のオプション（ロックを待機するかどうかなど）を設定します。この関数については、*The OCI Functions for C* のセクション 4-97 に記載されています。

開発者は、クライアントが DBMS リソースを待機するか、または直ちに例外を受け取るかを指定できます。以下は、`examples\jdbc\oracle\waiton.java` からの例です。

```
t3 = new T3Client("t3://bigbox:7001");
t3.connect();

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

```
  dbprops.put("server",   "bigbox");

  Properties t3props = new Properties();
  t3props.put("weblogic.t3",                t3);
  t3props.put("weblogic.t3.dbprops",        dbprops);
  t3props.put("weblogic.t3.driverClassName",
              "weblogic.jdbc.oci.Driver");
  t3props.put("weblogic.t3.driverURL",
              "jdbc:weblogic:oracle");
  t3props.put("weblogic.t3.cacheRows",
              getParameter("cacheRows"));

  Class.forName("weblogic.jdbc.t3.Driver").newInstance();

  // この拡張機能を利用するために
  // Connection を weblogic.jdbc.oci.Connection としてキャストする必要
がある
  Connection conn =
    (weblogic.jdbc.oci.Connection)
       DriverManager.getConnection("jdbc:weblogic:t3", t3props);

  // オブジェクトの作成後、直ちに
  // waitOnResources メソッドを呼び出す
  conn.waitOnResources(true);
```

このメソッドを使用すると、断続的にロックされる内部リソースを待つ間に、エラー リターン コードが返されることがあります。

この機能を利用するには、接続オブジェクトを `weblogic.jdbc.oci.Connection` としてキャストしてから、`waitOnResources()` メソッドを呼び出します。

拡張 SQL

JavaSoft の JDBC 仕様には、SQL 拡張または SQL Escape Syntax と呼ばれる機能が含まれています。すべての WebLogic jDriver JDBC ドライバは、拡張 SQL をサポートしています。詳細については、使用するドライバ用の開発者ガイドを参照してください。リストについては、WebLogic JDBC オプションを参照してください。

Oracle 配列フェッチ

WebLogic jDriver for Oracle は、2 層 JDBC Connection プロパティ `weblogic.oci.cacheRows` を設定することによって、Oracle 配列フェッチをサポートします。WebLogic jDriver for Oracle を使用する場合、この機能は WebLogic JDBC でもサポートされています。

WebLogic JDBC でこの機能を利用するには、2 層プロパティ `weblogic.oci.cacheRows` を設定します。

マルチバイト文字セットのサポート

WebLogic jDriver for Oracle は、インターナショナライゼーション サポート（AL24UTFFSS/UTF-8）も提供しており、WebLogic Server で使用している場合、WebLogic JDBC に拡張できます。この機能の詳細については、WebLogic jDriver for Oracle 用の開発者ガイドに記載しています。

WebLogic JDBC と Oracle NUMBER カラム

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

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

ただし、`getObject()` メソッドは、いくつかの問題を引き起こします。WebLogic jDriver for Oracle は、NUMBER カラムのどのような値も精度を変更することなく表す Java オブジェクトを返すことを保証します。つまり、値 1 は `Integer` として返されますが、123434567890.123456789 のような値は `BigDecimal` でのみ返されます。

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

修飾された NUMBER カラム、つまり、特定の精度が定義されているカラムでは、より信頼性の高い応答を得られます Oracle のメタデータはこれらのパラメータをドライバに提供するため、WebLogic jDriver for Oracle はテーブルの値にかかわりなく、常に特定の精度とスケールに合わせて適切な Java オブジェクトを返します。

多層ドライバの場合には、この問題がより複雑になります。デフォルトでは、WebLogic は、多層 JDBC アプリケーションが要求する前に、DBMS データのコンフィグレーション可能な数の ResultSet 行を取得します。これにより、クライアントがはっきりとわかるほどの性能の向上が実現されます。しかし、WebLogic では、クライアントがデータをどの形式で要求するかが事前にわからないので、行を包括的にプリフェッチして、クライアント アプリケーションに送信されたデータが、ResultSet のいずれか 1 つのカラムと同じ型になるようにします。プリフェッチを使用する場合、number 型に `getObject()` メソッドを使用することはできません。WebLogic Server で使用されている WebLogic jDriver for Oracle が、指定されたカラムに別の Java 型を返す可能性があるからです。その結果、WebLogic は、すべての greater-than-integer 数値のデータを String 形式でプリフェッチして、精度が失われないようにします。

この処理は、データが `getInt()`、`getFloat()`、`getBigDecimal()` などで要求された場合の、WebLogic JDBC クライアント アプリケーションでのデータ取得には影響しません。WebLogic JDBC は、String を適切な型に変換するからです。ただし、`getObject()` メソッドを呼び出した場合、WebLogic JDBC は、プリフェッチされたとおりに String を返します。

多層クライアント アプリケーションでの `getObject()` の動作を、2 層 WebLogic jDriver for Oracle アプリケーションでの動作と同じようにするには（つまり、プリフェッチせずに、同じカラム内でデータ型が混在する可能性がある状態で）、接続プロパティ weblogic.t3.cacheRows を zero（0）に設定して、WebLogic での行キャッシングをオフにします。

WebLogic JDBC と JDBC-ODBC ブリッジの実装

注意： ここでは、2 層ドライバとして JDBC-ODBC ブリッジを使用して、WebLogic JDBC でベンダのデータベースに接続する方法を、例を用いて示します。

注意： JDBC-ODBC ブリッジを使用して、エンタープライズ JavaBean を持つ Microsoft Access データベースにアクセスすることはサポートされていません。

- 手順 1. パッケージのインポート
- 手順 2. T3Client の作成
- 手順 3. 接続
  - データへのアクセス
  - 例外処理
- 最後の手順 . 接続の解除とオブジェクトのクローズ
- コードのまとめ

手順 1. パッケージのインポート

WebLogic JDBC で JDBC-ODBC ブリッジを使用するには、以下の、他の WebLogic JDBC クラスを使用するときと同じ import パッケージが必要です。

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

手順 2. T3Client の作成

WebLogic JDBC クライアントのコンストラクタは 1 つの引数を取ります。T3Client リクエストをリスンするポートを含む WebLogic Server の URL です。次の例は、try ブロックでのプログラムの初期作業です。

```
try {
  t3 = new T3Client("t3://bigbox:7001");
  t3.connect();
```

接続が失敗した場合でも、finally ブロックで t3Client に `disconnect()` メソッドを呼び出す必要があります。

手順 3. 接続

接続パラメータを設定するには、java.util.Properties オブジェクトを使用します。WebLogic Server と DBMS との接続（2 層接続）に 1 つの Properties セットを使用し、WebLogic JDBC クライアント、WebLogic Server、および DBMS 間の接続（多層接続）に別の Properties セットを使用します。2 層 Properties オブジェクト自体は、多層 Property の 1 つとして設定されます。多層 Properties オブジェクトは、Connection コンストラクタの引数として使用されます（Properties の設定に関する詳細）。

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

Properties t3props = new Properties();
t3props.put("weblogic.t3",              t3);
t3props.put("weblogic.t3.dbprops",      dbprops);
t3props.put("weblogic.t3.driverClassName",
            "sun.jdbc.odbc.JdbcOdbcDriver");
t3props.put("weblogic.t3.driverURL",
            "jdbc:odbc:Oracle_on_SS2");
t3props.put("weblogic.t3.cacheRows",    "10");

Class.forName("weblogic.jdbc.t3.Driver").newInstance();
conn = DriverManager.getConnection("jdbc:weblogic:t3client",
                                   t3props);

checkForWarning(conn.getWarnings());
```

この例の最後の行では、接続が確立された後にベンダ固有の警告をチェックしています。以下は、`checkForWarning` メソッドのコードです。このメソッドは、SQLWarning オブジェクトを引数として取り、SQLState、警告メッセージ、およびデータベース ベンダのエラー コードについての情報を表示します。1 つの SQLWarning オブジェクトに複数の警告がある場合もあります。

```
private static boolean checkForWarning (SQLWarning warn)
     throws SQLException
{
  boolean rc = false;
  if (warn != null) {
    System.out.println ("\n *** Warning ***\n");
```

```
      rc = true;
      while (warn != null) {
System.out.println ("SQLState: " +
    warn.getSQLState ());
System.out.println ("Message:  " +
    warn.getMessage ());
System.out.println ("Vendor:   " +
    warn.getErrorCode ());
System.out.println ("");
warn = warn.getNextWarning ();
      }
    }
    return rc;
  }
```

データへのアクセス

データベース メタデータを取り出すには、getMetaData メソッド（Connection クラス）を使用します。次の例では、取り出したメタデータに基づいたデータベースについての詳細を表示しています。

```
DatabaseMetaData dma = conn.getMetaData();

System.out.println("Connected to " + dma.getURL());
System.out.println("Driver       " + dma.getDriverName());
System.out.println("Version      " + dma.getDriverVersion());
System.out.println("");
```

Statement オブジェクトを使用して、簡単な SQL select オブジェクトを構築して実行し、ResultSet にデータを取り出します。Statement オブジェクトおよび ResultSet オブジェクトは、使い終わったらクローズします。

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM emp");
dispResultSet (rs);

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

クエリの結果は、ResultSet を引数とする private メソッドで表示します。dispResultSet() メソッドは、ResultSetMetaData を使ってテーブルのカラム見出しを出力し、ResultSet のコンテンツを行の多い順に表示します。

```
private static void dispResultSet (ResultSet rs)
     throws SQLException
{
  int i;

  ResultSetMetaData rsmd = rs.getMetaData ();
  int numCols = rsmd.getColumnCount();
```

```
    for (i=1; i < = numCols; i++) {
      if (i > 1) System.out.print(",");
      System.out.print(rsmd.getColumnLabel(i));
    }
    System.out.println("");

    boolean more = rs.next ();
    while (more) {
      for (i=1; i<=numCols; i++) {
if (i > 1) System.out.print(",");
System.out.print(rs.getString(i));
      }
      System.out.println("");
      more = rs.next ();
    }
  }
```

例外処理

SQLException は、最も面白いものです。以下に、Connection をインスタンス化してから SQLWarning をチェックしたときの SQL エラーと同じ情報を表示します。1 つの SQLException に複数の SQL エラーが含まれていることもあります。

```
 catch (SQLException ex) {
   System.out.println ("\n*** SQLException caught ***\n");
   ex.printStackTrace();

   while (ex != null) {
      System.out.println ("SQLState: " +
    ex.getSQLState ());
      System.out.println ("Message:  " +
    ex.getMessage ());
      System.out.println ("Vendor:   " +
    ex.getErrorCode ());
      ex = ex.getNextException ();
      System.out.println ("");
   }
 }
```

その他のすべての例外に関しては、単に、デバッグ目的でスタック トレースを出力します。

```
 catch (java.lang.Exception ex) {
   ex.printStackTrace ();
 }
```

最後の手順 . 接続解除とオブジェクトのクローズ

適切にクリーンアップを完了するため、必ず、finally ブロックで接続を閉じて、T3Client の接続を解除します。

```
finally {
  if (conn != null)
    try {conn.close();}    catch (Exception e) {;}
  if (t3   != null)
    try {t3.disconnect();} catch (Exception e) {;}
}
```

コードのまとめ

以下は、例で説明したコードのまとめです。

```
package examples.jdbc.odbc;

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

class simpleselect {

  public static void main (String args[]) {

    String url   = "jdbc:odbc:Oracle_on_SS2";
    String query = "SELECT * FROM emp";

    T3Client   t3   = null;
    Connection conn = null;
    try {

      t3 = new T3Client("t3://bigbox:7001");
      t3.connect();

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

      Properties t3props = new Properties();
      t3props.put("weblogic.t3",                t3);
      t3props.put("weblogic.t3.dbprops",        dbprops);
      t3props.put("weblogic.t3.driverClassName",
                  "sun.jdbc.odbc.JdbcOdbcDriver");
      t3props.put("weblogic.t3.driverURL",      url);
      t3props.put("weblogic.t3.cacheRows",      "10");

      Class.forName("weblogic.jdbc.t3.Driver").newInstance();
      conn = DriverManager.getConnection
                  ("jdbc:weblogic:t3client",t3props);
```

```
        checkForWarning(conn.getWarnings());

        DatabaseMetaData dma = conn.getMetaData();

        System.out.println("Connected to " + dma.getURL());
        System.out.println("Driver       " + dma.getDriverName());
       System.out.println("Version      " + dma.getDriverVersion());

        System.out.println("");

        // ドライバに SQL 文を送信するために
        // Statement オブジェクトを作成する
        Statement stmt = conn.createStatement();

        // クエリを送信して、ResultSet オブジェクトを作成する
        ResultSet rs = stmt.executeQuery(query);

        // 結果セットからすべての列と行を表示する
        dispResultSet (rs);

        // 結果セットと文を閉じる
        rs.close();
        stmt.close();

      }
      catch (SQLException ex) {

        // SQLExceptions を捕捉して、具体的なエラー情報を表示する
        System.out.println ("\n*** SQLException caught ***\n");
        ex.printStackTrace();

        while (ex != null) {
System.out.println ("SQLState: " +  ex.getSQLState ());
System.out.println ("Message:  " +  ex.getMessage ());
System.out.println ("Vendor:   " +  ex.getErrorCode ());
ex = ex.getNextException ();
System.out.println ("");
        }
      }


      catch (java.lang.Exception ex) {
        // 他の例外の場合は、単に StackTrace を出力する
        ex.printStackTrace();
      }
      finally {
        if (conn != null)
          try {conn.close();}    catch (SQLException sqe) {}
        if (t3   != null)
          try {t3.disconnect();} catch (Exception e) {}
      }
    }

    private static boolean checkForWarning (SQLWarning warn)
         throws SQLException
    {
```

```
    boolean rc = false;

    // SQLWarning オブジェクトを取得して、
    // warning messages.  Note that there could be
    // 複数の警告が一つにつながっている場合があることに注意する

    if (warn != null) {
      System.out.println ("\n *** Warning ***\n");
      rc = true;
      while (warn != null) {
System.out.println ("SQLState: " + warn.getSQLState ());
System.out.println ("Message:  " + warn.getMessage ());
System.out.println ("Vendor:   " + warn.getErrorCode ());
System.out.println ("");
warn = warn.getNextWarning ();
      }
    }
    return rc;
  }

  private static void dispResultSet (ResultSet rs)
       throws SQLException
  {
    int i;
    ResultSetMetaData rsmd = rs.getMetaData();
    int numCols = rsmd.getColumnCount();

    for (i=1; i<=numCols; i++) {
      if (i > 1) System.out.print(",");
      System.out.print(rsmd.getColumnLabel(i));
    }
    System.out.println("");

    boolean more = rs.next();


    while (more) {
      for (i=1; i<=numCols; i++) {
if (i > 1) System.out.print(",");
System.out.print(rs.getString(i));
      }
      System.out.println("");
      more = rs.next();
    }
  }
}
```

JDBC 接続のプロパティを設定するための URL の使い方と T3 ドライバの使い方

URL の機能

URL（uniform resource locator）は、ネットワーク上で特定のリソースを探し出すためのツールです。JDBC 仕様では、JDBC の運用に際して URL を広く使用するよう推奨されています。BEA では、すべての製品で一貫して、また現在の URL ガイドライン（www.w3.org を参照）でも、推奨されている内容に従っています。

したがって、WebLogic 製品では、WebLogic JDBC 接続の際に、URL を使用して必要なパラメータを指定できます。

WebLogic URL の構成

Properties オブジェクトと URL による接続の指定

WebLogic JDBC ドライバは、クライアントと対象データベースとの間で JDBC 接続を開くためのプロパティを `java.utils.Properties` オブジェクトを使用して設定します。また、JDBC 仕様で説明されているように、URL を使用して JDBC ドライバを指定することもできます。Properties オブジェクトと URL は、`DriverManager.getConnection()` メソッドまたは `Driver.connect()` メソッドの引数となります。

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。そのため、このマニュアルのコード例で説明したように、`Driver.connect` メソッドを使用することをお勧めします。

このモデルの通常の処理は以下のとおりです。

ユーザ名やパスワードなどの情報を指定するための `java.util.Properties` オブジェクトを作成します。

ロードして java.sql.Driver オブジェクトにキャストする JDBC ドライバのクラス名を指定して、Class.forName().newInstance() メソッドを呼び出します。

Driver.connect() メソッドを使用して接続を作成します。このメソッドには、引数として JDBC ドライバの URL とプロパティのセットを指定します。

Properties オブジェクトを作成した後、Class.forName().newInstance() メソッドを呼び出し、接続オブジェクトを作成するという処理を行うコードを以下に示します。

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

2 層接続用のユーザ名、パスワード、およびサーバ名を格納する Properties オブジェクトは、多層 WebLogic JDBC 接続用のパラメータを設定するための別の Properties オブジェクトの一部となります。

```
Properties t3props = new Properties();
t3props.put("weblogic.t3",                t3);
t3props.put("weblogic.t3.dbprops",        dbprops);
t3props.put("weblogic.t3.driverClassName",
"weblogic.jdbc.oci.Driver");
t3props.put("weblogic.t3.driverURL",
"jdbc:weblogic:oracle");
```

2 層接続と多層接続のどちらの情報も、Driver.connect() メソッドを呼び出して JDBC 接続をインスタンス化するときにドライバに渡されます。このメソッドには、URL（文字列）と java.util.Properties オブジェクトという 2 つの引数を指定します。

```
Driver myDriver = (Driver)
Class.forName("weblogic.jdbc.t3.Driver").newInstance();
Connection conn = Driver.connect("jdbc:weblogic:t3", t3props);
```

URL を使用した WebLogic JDBC 接続の指定

Java 開発環境の中には、Sybase の PowerJ のように、Properties オブジェクトをユーザ名とデータベースを指定するためだけに使用するものもあります。こうした開発環境に対応するには、`DriverManager.getConnection()` メソッドの最初の引数として使用する URL にサーバ名、`weblogic.t3` プロパティを含めて、WebLogic JDBC 接続を行うために必要なパラメータをすべて指定する必要があります。

WebLogic では、PowerJ のような環境を対象として、RFC 1630 スタイルのクエリ文字列に基づいて長い URL を構築することで、WebLogic JDBC 接続に必要な情報をすべて指定できるようにしています。

WebLogic URL の構文は以下のとおりです。太字で示してあるのはプロパティの名前で、そのプロパティに指定する値の例が続きます。

```
jdbc:weblogic:t3 (with JDK 1.1)
```

URL の先頭部分で、WebLogic JDBC ドライバの URL を指定します（専門用語では、WebLogic JDBC ドライバの URL は「URL のスキーム」と呼ばれます）。

現在、WebLogic は JDK 1.0.2 をサポートしていないという点に注意してください。

WebLogic JDBC ドライバの URL には、疑問符（?）を続け、さらにプロパティ名と値のペアを続けます。ペアが複数の場合は、間にアンパサンド（&）を入れます（例では、区切り文字の色を変えて示しています）。または、「?」の後の引数は、後の例とは異なる任意の順序で指定できます。

```
weblogic.t3.serverURL=t3://localhost:7001
```

WebLogic Server の URL です。

```
weblogic.t3.driverURL=jdbc:weblogic:oracle:DEMO
```

2 層接続用 JDBC ドライバの URL です。この構文には、「server」プロパティとして設定する情報を入れる必要があります。たとえば、`jdbc:weblogic:oracle:DEMO` は、Oracle データベース用の WebLogic JDBC ドライバがホスト「DEMO」に存在するということを示します。

```
weblogic.t3.driverClassName=weblogic.jdbc.oci.Driver
```

2 層接続用 JDBC ドライバのクラス名です。クラス名は、ドット表記の完全パッケージ名でなければなりません。

`weblogic.t3.cacheRows=10`

WebLogic Server にキャッシュする行の数です。

`weblogic.t3.connectionPoolID=eng`

接続プールの ID です。この ID は、`weblogic.properties` ファイルに登録しなければなりません。接続プールの JDBC 接続を使用する場合、URL には接続プールの ID 以外の情報は不要です。詳細については、『WebLogic JDBC プログラミング ガイド』で接続プールの説明を参照してください。

上記以外にも、省略可能なプロパティがあります。これらのプロパティの詳細については、『開発者ガイド』の「接続用プロパティの設定」を参照してください。

次の例は、「DEMO」という Oracle データベースに接続するための URL を指定します。接続には、ホスト「toyboat.toybox.com」のポート 7001 使用して動作する WebLogic Server を使用し、cacheRows プロパティは 25 に設定します。この例では、データベースに接続するための「username」および「password」プロパティが Properties オブジェクトで設定されたことを前提にしています。赤で示した文字「?」および「&」は、URL では特殊な意味を持ちます。見やすくするために、この例では URL を別の行に示してありますが、実際には 1 行の長い文字列になります。

```
jdbc:weblogic:t3?
weblogic.t3.serverURL=t3://toyboat.bigbox.com:7001&
weblogic.t3.driverClassName=weblogic.jdbc.oci.Driver&
weblogic.t3.driverURL=jdbc:weblogic:oracle:DEMO&
weblogic.t3.cacheRows=25
```

URL の簡略化

URL では、ある部分に指定された情報を基に、その他の部分に指定すべき情報を推測することができます。この性質を利用すると、URL を簡略化できます。

たとえば、URL の先頭部分（WebLogic JDBC ドライバの URL）にデータベースベンダや DBMS ホストを入れておくと、次に示すように、`weblogic.t3.driverURL` を指定する必要がなくなります。

```
jdbc:weblogic:t3:oracle:DEMO
```

次の URL を上記のような形にできます。

```
jdbc:weblogic:t3?weblogic.t3.driverURL=jdbc:weblogic:oracle:DEMO
```

また、WebLogic jDriver のいずれかのドライバを使用する場合は、ドライバのクラス名を省略することもできます。URL の先頭部分（上記の例を参照）、またはプロパティ `weblogic.t3.driverURL` から推測できるからです。以下の表は、URL に指定されたベンダ名に、どの WebLogic jDriver ドライバが対応するかを示します。

URL	対応するドライバ
`jdbc:weblogic:oracle`	`weblogic.jdbc.oci.Driver`
`jdbc:weblogic:mssqlserver4`	`weblogic.jdbc.mssqlserver4.Driver`
`jdbc:weblogic:informix4`	`weblogic.jdbc.informix4.Driver`

ここで説明したことを応用して、先の例に使った URL を簡略すると、以下のようになります。

```
jdbc:weblogic:t3:oracle:DEMO?
weblogic.t3.serverURL=t3://toyboat.bigbox.com:7001&
weblogic.t3.cacheRows=25
```

URL に記述する特殊文字

URL に指定する文字は、US-ASCII 文字セットに属するグラフィック表示可能なものに限ります。具体的には、A ～ Z の英文字（大文字と小文字）、0 ～ 9 の整数の他、$-_.+!*'()" などの特殊文字が含まれます。しかし、その他の文字は、適切な 2 桁の 16 進数（0123456789ABCDEF）のコードに置き換える必要があります（先頭には「%」を付ける）。

特殊文字の中には、URL で予約文字になっているものもあります（; / ? : @ = & など）。これらの文字も、使用する場合は、16 進数のコードに変換しなければなりません。

また、以下に示す文字は、WebLogic URL では予約文字となっています。これらの文字も同様に、使用する場合は、16 進数のコードに変換しなければなりません。

%26 (&)

%3D (=)

%3F (?)

%2F (/)

%3A (:)

IDE（ウィザード）の使い方

URL は、Sybase の PowerJ などの IDE（統合開発環境。ウィザードともいう）にも使用できます。クラス名や URL を常に完全な形で指定しなければ正しく動作しない IDE を使用する場合は、WebLogic の JDBC ドライバのクラス名を以下のように入力します。

```
JDBC Driver:
 weblogic.jdbc.t3.Driver
```

赤で示した文字「?」および「&」は、URL では特殊な意味を持ちます。見やすくするために、この例では URL を別の行に示してありますが、実際には 1 行の長い文字列になります。次の例は、PowerJ データベース ウィザード向けの URL です。

```
Data Source URL:
 jdbc:weblogic:t3?
 weblogic.t3.serverURL=t3://toyboat.bigbox.com:7001&
 weblogic.t3.driverClassName=weblogic.jdbc.oci.Driver&
 weblogic.t3.driverURL=jdbc:weblogic:oracle:DEMO&
 weblogic.t3.cacheRows=25
```