

# Oracle9i Lite

Java 開発者ガイド

リリース 5.0

2001 年 7 月

部品番号 : J04362-01

**ORACLE®**

---

Oracle9i Lite Java 開発者ガイド, リリース 5.0

部品番号 : J04362-01

原本名 : Oracle9i Lite Developer's Guide for Java, Release 5.0

原本部品番号 : A90123-01

Copyright © 2000, 2001 Oracle Corporation. All rights reserved.

Printed in Japan.

制限付権利の説明

プログラム (ソフトウェアおよびドキュメントを含む) の使用、複製または開示は、オラクル社との契約に記載された制約条件に従うものとします。著作権、特許権およびその他の知的財産権に関する法律により保護されています。

当プログラムのリバース・エンジニアリング等は禁止されています。

このドキュメントの情報は、予告なしに変更されることがあります。オラクル社は本ドキュメントの無謬性を保証しません。

\* オラクル社とは、Oracle Corporation (米国オラクル) または日本オラクル株式会社 (日本オラクル) を指します。

危険な用途への使用について

オラクル社製品は、原子力、航空産業、大量輸送、医療あるいはその他の危険が伴うアプリケーションを用途として開発されておりません。オラクル社製品を上述のようなアプリケーションに使用することについての安全確保は、顧客各位の責任と費用により行ってください。万一かかる用途での使用によりクレームや損害が発生いたしましても、日本オラクル株式会社と開発元である Oracle Corporation (米国オラクル) およびその関連会社は一切責任を負いかねます。当プログラムを米国防総省の米国政府機関に提供する際には、『Restricted Rights』と共に提供してください。この場合次の Notice が適用されます。

Restricted Rights Notice

Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication, and disclosure of the Programs, including documentation, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication, and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-19, Commercial Computer Software - Restricted Rights (June, 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

このドキュメントに記載されているその他の会社名および製品名は、あくまでその製品および会社を識別する目的のみ使用されており、それぞれの所有者の商標または登録商標です。

---

---

# 目次

はじめに .....	v
このマニュアルについて .....	vi
表記法規約 .....	vi
<b>1 Oracle Lite の Java サポート</b>	
Oracle Lite の Java 機能 .....	1-2
ネイティブな JDBC サポート .....	1-2
Java ストアド・プロシージャとトリガー .....	1-3
SQLJ .....	1-3
Java データ型 .....	1-4
Java ツール .....	1-4
loadjava .....	1-5
Oracle Lite が提供する Java 開発環境 .....	1-5
環境の設定 .....	1-5
JDK 1.2 の場合の変数の設定 .....	1-6
<b>2 Java ストアド・プロシージャとトリガー</b>	
Oracle Lite の新機能 .....	2-2
Java ストアド・プロシージャとトリガーの概要 .....	2-2
ストアド・プロシージャの使用法 .....	2-3
モデル 1: Load and Publish ストアド・プロシージャ開発モデルの使用 .....	2-4
クラスのロード .....	2-4
loadjava .....	2-5
CREATE JAVA の使用法 .....	2-8
SQL へのストアド・プロシージャのパブリッシュ .....	2-9

パブリッシュされたストアド・プロシージャのコール .....	2-10
パブリッシュされたストアド・プロシージャの削除 .....	2-12
dropjava の使用 .....	2-12
SQL コマンドの使用 .....	2-13
例 .....	2-13
モデル 2: アタッチ済みストアド・プロシージャ開発モデルの使用 .....	2-15
表レベルのストアド・プロシージャ .....	2-15
行レベルのストアド・プロシージャ .....	2-16
表への Java クラスのアタッチ .....	2-16
アタッチ済みストアド・プロシージャのコール .....	2-17
アタッチ済みストアド・プロシージャの削除 .....	2-18
例 .....	2-18
Java ストアド・プロシージャの ODBC からのコール .....	2-20
<b>Java データ型</b> .....	2-21
パラメータの宣言 .....	2-22
ストアド・プロシージャを使用して複数行を返す方法 .....	2-22
ODBC を使用して複数行を返す方法 .....	2-23
例 .....	2-23
<b>トリガーの使用</b> .....	2-24
文レベルのトリガーと行レベルのトリガーの比較 .....	2-24
トリガーの作成 .....	2-24
トリガーの使用可能と使用禁止 .....	2-25
トリガーの削除 .....	2-25
トリガーの例 .....	2-26
トリガーの引数 .....	2-28
トリガーの引数の例 .....	2-28

### 3 Java Database Connectivity (JDBC)

Oracle Lite の新機能 .....	3-2
JDBC .....	3-2
JDBC 環境の設定 .....	3-2
Oracle Lite JDBC ドライバのロード .....	3-2
Oracle Lite データベースへの接続 .....	3-3
JDBC からの Java ストアド・プロシージャの実行 .....	3-4
ExecuteQuery メソッドの使用 .....	3-4
Callable Statement の使用 .....	3-5

Oracle Lite での拡張 .....	3-6
データ型の拡張 .....	3-6
データ・アクセスの拡張 .....	3-7
BLOB の読取りを行うサンプル・プログラム .....	3-9
CLOB への書込みを行うサンプル・プログラム .....	3-10
JDBC と Java アクセス・クラス間の相互運用性 .....	3-10
制限事項 .....	3-11

## 4 Java 埋込み SQL (SQLJ)

SQLJ の概要 .....	4-2
SQLJ の操作 .....	4-2
SQLJ の使用方法 .....	4-3
SQLJ の構成 .....	4-3
Oracle Lite への接続 .....	4-3

## A ストアド・プロシージャのチュートリアル

ストアド・プロシージャとトリガーの作成 .....	A-2
SQL*Plus の起動 .....	A-2
表の作成 .....	A-2
Java クラスの作成 .....	A-3
Java クラスのロード .....	A-5
ストアド・プロシージャのパブリッシュ .....	A-5
データベースへの値の移入 .....	A-5
プロシージャの実行 .....	A-5
電子メール・アドレスの検証 .....	A-6
トリガーの作成 .....	A-6
トリガーのテスト .....	A-6
電子メール・アドレスの検証 .....	A-6
コミットまたはロールバック .....	A-7

## B JDBC アプリケーションのチュートリアル

環境の検査 .....	B-2
アプリケーションへの JDBC ドライバのロード .....	B-2
Oracle Lite データベースへの接続 .....	B-2
問合せの発行と結果の参照 .....	B-5

トランザクションのコミット .....	B-7
完全な例 .....	B-7
この例のコンパイルと実行 .....	B-8

## C JDBC 2.0 の新機能

インタフェース接続 .....	C-2
メソッド .....	C-2
インタフェース文 .....	C-3
インタフェース ResultSet .....	C-4
フィールド .....	C-4
メソッド .....	C-5
FALSE を返すメソッド .....	C-7
インタフェース DatabaseMetaData .....	C-8
メソッド .....	C-8
FALSE を返すメソッド .....	C-9
インタフェース ResultSetMetaData .....	C-10
メソッド .....	C-10
インタフェース PreparedStatement .....	C-10
メソッド .....	C-10

## D サンプル・プログラム

Java サンプル・プログラムの概要 .....	D-2
JDBC のサンプル .....	D-2
PL/SQL から Java サンプルへの変換 .....	D-2
Java ストアド・プロシージャのサンプル .....	D-3
サンプルの実行 .....	D-4
JDBC のサンプルの実行 .....	D-4
PL/SQL 変換のサンプルの実行 .....	D-5
Java ストアド・プロシージャのサンプルの実行 .....	D-5

## 索引

---

# はじめに

ここでは、『Oracle9i Lite Java 開発者ガイド』を紹介します。説明する内容は、次のとおりです。

- [このマニュアルについて](#)
- [表記法規約](#)

## このマニュアルについて

『Oracle9i Lite Java 開発者ガイド』は、Oracle Lite の Java 機能を解説したマニュアルです。このマニュアルを使用するには、Java プログラミング言語の知識が必要となります。

『Oracle9i Lite Java 開発者ガイド』の構成は、次のとおりです。

第 1 章 「Oracle Lite の Java サポート」	Oracle Lite の Java 機能を紹介し、開発環境の設定方法について説明します。
第 2 章 「Java ストアド・プロシージャとトリガー」	Oracle Lite での Java ストアド・プロシージャとトリガーの開発方法を説明します。
第 3 章 「Java Database Connectivity (JDBC)」	Oracle Lite での JDBC サポートを説明します。
第 4 章 「Java 埋込み SQL (SQLJ)」	Java 埋込み SQL の標準である SQLJ の概要を説明します。
付録 A 「ストアド・プロシージャのチュートリアル」	ストアド・プロシージャの作成と実行に関するチュートリアルを提供します。
付録 B 「JDBC アプリケーションのチュートリアル」	Oracle Lite データベースにアクセスする JDBC プログラムの作成に関するチュートリアルを提供します。
付録 D 「サンプル・プログラム」	Oracle Lite に同梱の Java のサンプル・プログラムについて説明します。

---

---

**注意：** このマニュアルにおいて、「Oracle8i」および「Oracle8i Server」という記述は、Oracle8i のリリース 8.1.5 以上を意味します。

---

---

## 表記法規約

このマニュアルでの表記法規約は次のとおりです。

イタリック	イタリックは変数を表します。変数には適切な値を代入します。
...	コードの一部で使用される省略記号は、その説明内容に関係のないコードを省略していることを意味します。
[ ]	SQL 文では、オプション項目は大カッコで囲まれます。たとえば、[a   b] は、a と b がどちらもオプションの引数であることを表します。カッコは入力しないでください。
{ }	中カッコで囲まれている項目は、いずれか 1 つを選択します。たとえば、{a   b} は、a か b の（両方ではなく）どちらかを選択する必要があります。



- | 垂直バーは、2つのオプションがある場合に「または」を意味します。入力するオプションは1つのみです。垂直バー自体は入力しないでください。
- 英大文字 テキスト内の英大文字は、SELECT や UPDATE などの SQL キーワードを表します。



---

---

# Oracle Lite の Java サポート

この章では、Oracle Lite でサポートされている Java インタフェースとツールを紹介します。  
説明する内容は、次のとおりです。

- [Oracle Lite の Java 機能](#)
- [Java ツール](#)
- [Oracle Lite が提供する Java 開発環境](#)

## Oracle Lite の Java 機能

豊富な Java サポートを備え、コンパクトで、しかも Oracle8i との相互運用が可能な Oracle Lite は、Mobile アプリケーションとデータの配置と管理には理想的なプラットフォームです。Oracle Lite の Java 機能には、ソフトウェア開発およびデータ管理方針を幅広くサポートするライブラリ、ツールおよびインタフェースが含まれています。

Java 機能には、次のものが含まれます。

- ネイティブな JDBC ドライバ。
- Java スタアド・プロシージャおよびトリガーのサポート。
- Java アクセス・クラス (JAC)。コア・データベースへのオブジェクト指向インタフェースを提供するクラス・ライブラリです。
- SQLJ サポート。

Oracle Lite は、Java コンポーネントの開発および管理用のツールも提供します。Oracle Lite のシームレスな永続 Java ポストプロセッサは、一時 Java クラスを Oracle Lite のステートフル永続クラスに変換します。Oracle Lite Designer は、Java クラスとオブジェクトを管理するビジュアル・インタフェースで、データベースのリレーショナル・ビューとオブジェクト指向ビューを提供します。loadjava コマンドライン・ユーティリティは、Java クラスを Oracle Lite データベースにロードする作業を自動化します。

Oracle Lite が提供する Java 機能の多くは、Oracle8i と互換性があります。この互換性により、企業のデータとアプリケーションを軽量のモバイル・プラットフォーム (PC、ラップトップ、携帯端末、埋込みデバイスなど) にすばやく簡単に配置できます。

## ネイティブな JDBC サポート

JDBC (Java Database Connectivity) は、Java プログラムで SQL を使用するための業界標準の API です。JDBC は Microsoft 社の Open Database Connectivity (ODBC) を基に作成されています。JDBC 実装のほとんどでは、DBMS へのアクセスを提供するために、JDBC コールが JDBC-ODBC ブリッジを経由する必要があります。しかし、Oracle Lite では、ネイティブなタイプ 2 JDBC ドライバを提供しています。Oracle Lite が提供するネイティブなドライバにより、JDBC アプリケーションは Oracle Lite オブジェクト・カーネルと直接対話できます。

Oracle Lite JDBC ドライバは、JDBC 1.22 仕様をサポートします。さらに、JDBC 2.0 標準の特定の機能 (ラージ・オブジェクト・データ型やデータ・アクセス関数など) もサポートします。

Oracle Lite の JDBC 実装は、SQL92 Intermediate Level 標準に準拠し、次の拡張が含まれています。

- Java スタアド・プロシージャおよびトリガー
- Oracle データ型 (VARCHAR2 や BLOB など)

- CONNECT BY および START WITH 句による再帰問合せ
- Oracle 組込みファンクション

## Java ストアド・プロシージャとトリガー

Java ストアド・プロシージャおよびトリガーは、直接 Oracle Lite データベースに格納される Java プログラムです。データベースにアクセスするアプリケーションは、Java ベースでない場合でも、ストアド・プロシージャを実行できます。トリガーは、行の更新や削除などの特定のデータベース・イベントが発生したときに実行されるストアド・プロシージャです。ストアド・プロシージャとトリガーの使用により、データベースに関連するロジックは、データベースにアクセスする各アプリケーションに格納するのではなく、一箇所に集中して格納できます。

アプリケーションが Java ストアド・プロシージャをコールすると、Oracle Lite のオブジェクト・カーネルは最初に Java Virtual Machine (JVM) のインスタンスを作成します (まだ作成されていない場合)。次に、オブジェクト・カーネルはメソッドを含むクラスをデータベースから取り出し、これを JVM にロードし、このメソッドを実行します。

## SQLJ

SQLJ は、静的 SQL 文を Java プログラムに埋め込むための標準です。JDBC と比較すると、SQLJ には次のような利点があります。

- ソース・プログラムが簡潔になり、サイズも小さくなります。
- SQL 構文チェック、データベース・スキーマ・チェック、タイプ・チェックにより、プログラマによるエラーの確率が低くなります。
- 実行速度が速くなります。

ほとんどの場合、SQLJ 文は JDBC 文と同じソース内にあります。SQLJ は静的 SQL 文に、JDBC は動的 SQL 文に対して使用します。

SQLJ を使用するには、Oracle SQLJ トランスレータ、ランタイム・コンポーネントおよび付属のドキュメントを、次の Web サイトからダウンロードします。

<http://www.oracle.com/java/sqlj/index.html>

## Java データ型

Oracle Lite は、次の表に示すように Java と Oracle との間でデータ型を変換します。

Java データ型	SQL データ型
byte[], byte[][]、Byte[]	BINARY、RAW、VARBINARY
boolean、Boolean	BIT
String、String[]	CHAR、VARCHAR、VARCHAR2
short、short[]、short[][]、Short、Short[]	SMALLINT
int、int[]、int[][]、Integer、Integer[]	INT
float、float[]、float[][]、Float、Float[]	REAL
double、double[]、double[][]、Double、Double[]	DOUBLE、NUMBER（精度の指定なし）
BigDecimal、BigDecimal[]	NUMBER(n)
java.sql.Date、java.sql.Date[]	DATE
java.sql.Time、java.sql.Time[]	TIME
java.sql.Timestamp、java.sql.Timestamp[]	TIMESTAMP
java.sql.Connection	データベースへのデフォルトの JDBC 接続
oracle.lite.jac.POLJACConnection	データベースへのデフォルトの JAC 接続
oracle.lite.jac.POLPersistantCapable	オブジェクトの JAC インスタンス

## Java ツール

Oracle Lite は、Java 開発を管理する次のツールを提供します。

ツール	説明
loadjava	Java クラスを Oracle Lite にロードします。
dropjava	Java クラスを Oracle Lite から削除します。

## loadjava

loadjava ユーティリティは、Java クラスとリソースを Oracle Lite データベースにロードする作業を自動化します。loadjava を使用すると、クラスおよびリソースを個別に、または ZIP あるいは JAR アーカイブとしてロードできます。

クラスをロードした後、SQL 文からアクセスするクラスのメソッドのコール仕様を作成します。値を返すストアド・プロシージャのコール仕様を作成するには、SQL の CREATE FUNCTION 文を使用します。ストアド・プロシージャが値を返す必要がない場合は、CREATE PROCEDURE 文を使用します。

クラスのアンロード用として、Oracle Lite では dropjava を提供していますが、これは loadjava と似たように機能します。

## Oracle Lite が提供する Java 開発環境

Oracle Lite における Java 開発は、次のツールにより促進されます。

- Oracle Developer 2.1 は、Oracle Lite でのユーザー定義 Java ストアド・プロシージャをサポートしています。
- Oracle JDeveloper は、Oracle Lite での Java ストアド・プロシージャをサポートし、Java ストアド・プロシージャの開発と配置を支援するように設計された機能を装備しています。

## 環境の設定

この項では、Oracle Lite アプリケーションを作成する開発環境の設定方法を説明します。Java アプリケーションを開発するには、サン・マイクロシステムズ社の Java Development Kit (JDK) バージョン 1.1.8 (以降) が必要です。

Oracle Lite のインストール後、Oracle Lite が JDK とともに稼働できるように、環境変数 PATH および CLASSPATH を設定します。PATH および CLASSPATH の設定は、次の項で説明されているように、使用する JDK のバージョンにより異なります。

Oracle Lite をインストールする前に環境変数に CLASSPATH ユーザー変数が含まれており、このユーザー変数に CLASSPATH システム変数が含まれていない (CLASSPATH=...;%CLASSPATH% として指定されていない) 場合、CLASSPATH ユーザー変数を変更して Oracle\_Home¥Mobile¥CLASSES ディレクトリ内の OLITE40.JAR ファイルを組み込んでください。

---

---

**注意：** CLASSPATH に対する変更を反映するには、すべてのコマンド・プロンプト・ウィンドウを閉じてから再度オープンする必要があります。

---

---

## JDK 1.2 の場合の変数の設定

JDK 1.2 を使用する場合は、JDK 1.2 Java コンパイラ (**javac.exe**) が含まれたディレクトリを、他の Java コンパイラを含むディレクトリよりも前に PATH 変数内に指定する必要があります。

Classic Java VM の共有ライブラリ (**jvm.dll**) を含むディレクトリを PATH に追加します。**jvm.dll** は、`JDK_Home\jre\bin\classic` ディレクトリにあります。

**oljdk11.jar** を CLASSPATH から削除します。これにより、Oracle Lite で使用されるシステム・クラスのバージョンが、JDK で使用されるバージョンと同じになります。例を次に示します。

```
set PATH=c:\JDK_Home\bin;c:\JDK_Home\jre\bin\classic
set CLASSPATH=c:\JDK_Home\jre\lib\rt.jar;c:\Oracle_Home\Mobile\classes\olite40.jar
```

Classic Java VM を使用するかわりに、HotSpot Java VM を使用することもできます。HotSpot は、サン・マイクロシステムズ社により提供されている JDK アドオン・モジュールです。これは JDK 1.2 のパフォーマンスを強化します。また、Java クラスおよびオブジェクトの管理用のビジュアル・ツールである Oracle Lite Designer のパフォーマンスを改善します。HotSpot はサン・マイクロシステムズ社の Web サイトから入手できます。

HotSpot をインストールした後、次のように PATH を設定します。

```
set PATH=c:\jdk1.2\bin;c:\jdk1.2\jre\bin\hotspot;%PATH%
```

システムが正常に設定されたかどうかを調べるには、`Oracle_Home\Mobile\Sdk\Lite\Examples\Java` ディレクトリにある Java のサンプルを実行します。



---

# Java ストアド・プロシージャとトリガー

この章では、Oracle Lite リレーショナル・モデル内で Java ストアド・プロシージャとトリガーを使用する方法を説明します。説明する内容は、次のとおりです。

- [Oracle Lite の新機能](#)
- [Java ストアド・プロシージャとトリガーの概要](#)
- [ストアド・プロシージャの使用方法](#)
- [Java データ型](#)
- [トリガーの使用方法](#)

## Oracle Lite の新機能

この項では、今回のリリースで拡張された Oracle Lite の Java ストアド・プロシージャとトリガーのサポートについて説明します。

Oracle Lite では、Oracle8i のストアド・プロシージャ開発モデルをサポートするようになりました。このモデル (Load and Publish 開発モデル) では、クラスを表にアタッチするかわりに、loadjava コマンドライン・ユーティリティまたは SQL の CREATE JAVA コマンドを使用して、Java クラスを Oracle Lite データベースにロードします。クラスをデータベースにロードした後、コール仕様を使用して、SQL からコールするクラスのメソッドをパブリッシュします。コール仕様を作成するには、CREATE FUNCTION または CREATE PROCEDURE コマンドを使用します。詳細は、「[モデル 1: Load and Publish ストアド・プロシージャ開発モデルの使用](#)」を参照してください。

Oracle Lite では、ストアド・プロシージャに作成にあたり従来のモデルもサポートしています。従来のモデルでは、Java クラスを表にアタッチします。クラス内の静的メソッドがその表の表レベルのストアド・プロシージャになり、非静的 (インスタンス) メソッドが、行レベルのストアド・プロシージャになります。

Oracle Lite には、今回、Java クラスをデータベースにロードする作業を自動化する loadjava ユーティリティが含まれています。loadjava を使用すると、Java クラス、ソースおよびリソースを個別にまたはアーカイブとしてロードできます。詳細は、「[loadjava](#)」を参照してください。

## Java ストアド・プロシージャとトリガーの概要

Java ストアド・プロシージャは、Oracle Lite データベースに格納される Java メソッドです。このプロシージャは、データベースにアクセスするアプリケーションにより起動できます。トリガーは、行の更新、挿入または削除などの特定のイベントが発生したときに実行 (起動) されるストアド・プロシージャです。列を更新した場合も、トリガーを起動できます。

トリガーは、文レベルまたは行レベルで操作できます。文レベルのトリガーは、複数の行が影響されても、トリガー文につき 1 回起動されます。行レベルのトリガーは、トリガー文により影響される行につき 1 回起動されます。Java ストアド・プロシージャは、1 つの値、1 行または複数行を返しますが、トリガーは値を返しません。

ストアド・プロシージャを作成するには、まず Oracle Lite に格納するクラスを作成します。Java IDE を使用してプロシージャを作成するか、既存のプロシージャが要件に合うものであれば、それを再利用できます。

クラスを作成するときは、SQL DML 文からの Java ストアド・プロシージャのコールに関する次の制限を考慮してください。

- INSERT 文、UPDATE 文または DELETE 文からコールされたメソッドは、その文により変更されたデータベース表に対して問合せまたは変更できません。
- SELECT 文、INSERT 文、UPDATE 文または DELETE 文からコールされたメソッドは、COMMIT などの SQL トランザクション制御文を実行できません。また、CREATE など

の DDL 文には自動コミットが付随するため、ストアド・プロシージャでこのような DDL 文を実行できません。

ストアド・プロシージャ内に、前述の制限に違反する SQL 文があると、実行時に（その文が解析されるときに）エラーが生成されます

1 つの Oracle Lite アプリケーションには Java Virtual Machine (JVM) が 1 つしかロードされないため、クラスには配置環境において一意の名前を付ける必要があります。アプリケーションが複数のデータベースからメソッドをコールする場合、それぞれのデータベースの Java クラスは同じ JVM にロードされます。Java クラス名にデータベース名を接頭辞として付けると、複数のデータベースにわたって Java クラス名を一意にできます。

Java ストアド・プロシージャが `java.sql.Connection` 型、`POLPersistentCapable` 型または `POLConnection` 型の引数を取る場合、Oracle Lite はカレント・トランザクションまたはカレント行から適切な値を最初の引数としてメソッドに渡します。このメソッドをコールするアプリケーション側でこのパラメータに値を指定する必要はありません。

## ストアド・プロシージャの使用方法

Oracle Lite は、ストアド・プロシージャを作成する開発モデルをいくつかサポートします。Load and Publish モデルでは、Java クラスを Oracle Lite にロードし、次に、SQL からコールするクラスの静的メソッド用にコール仕様を作成します。このモデルは、Oracle8i でもサポートされています。このため、Oracle8i の企業向けアプリケーションの実装で培ったスキルとリソースを利用できます。

このモデルでは、次の手順を使用します。

1. 格納するメソッドを含む Java クラスを開発します。
2. `loadjava` ユーティリティまたは SQL の `CREATE JAVA` コマンドを使用して、クラスを Oracle Lite データベースにロードします。
3. SQL からアクセスするメソッドを、コール仕様を作成してパブリッシュします。メソッドのパブリッシュにより、SQL 名がメソッドに関連付けられます。SQL アプリケーションではこの名前を使用してメソッドをコールします。

Oracle Lite に格納してあるプロシージャをすべてパブリッシュする必要はありません。SQL からコールするプロシージャのみをパブリッシュします。他のストアド・プロシージャからコールされるだけのストアド・プロシージャも多数あり、これらをパブリッシュする必要はありません。このモデルを使用してストアド・プロシージャを開発する方法の詳細は、「[モデル 1: Load and Publish ストアド・プロシージャ開発モデルの使用](#)」を参照してください。Load and Publish モデルでサポートされているのは静的メソッドのみです。

次のモデルでは、クラスを表にアタッチし、クラス内のメソッドを名前によりコールします。これは、ストアド・プロシージャを開発する従来の Oracle Lite モデルです。このモデルを使用すると、クラス・レベルの（静的）メソッドとオブジェクト・レベルの（非静的）メソッドの両方を格納できます。

このモデルでは、次の手順に従います。

1. 格納するメソッドを含む Java クラスを開発します。
2. SQL の ALTER TABLE コマンドを使用して、クラスを表にアタッチします。

クラスをアタッチした後は、クラス内のメソッドを SQL から直接コールできます。メソッドは次の構文で指定します。

```
table_name.method_name
```

Java クラスを表にアタッチする方法は、「[モデル 2: アタッチ済みストアド・プロシージャ開発モデルの使用](#)」を参照してください。

Oracle Lite は、ストアド・プロシージャを削除するためのツールと SQL コマンドを提供しています。Oracle Lite ではクラス間の依存性を追跡しないため、データベースからプロシージャを削除するときは注意が必要です。削除するストアド・プロシージャが、他のストアド・プロシージャで参照されていないことを確認する必要があります。クラスを削除すると、直接的あるいは間接的にそのクラスに依存するクラスが無効になります。

## モデル 1: Load and Publish ストアド・プロシージャ開発モデルの使用

この項では、Load and Publish 開発モデルを使用してストアド・プロシージャを作成する方法を説明します。ストアド・プロシージャを作成するには、まずクラスを作成します。クラスがエラーなしでコンパイルされ実行されることを確認します。次に、クラスを Oracle Lite データベースにロードします。最後に、SQL からコールするメソッドをパブリッシュします。Oracle Lite では、main メソッドにマップされるメソッドはパブリッシュできません。一方、Oracle8i では、main メソッドをパブリッシュするコール仕様を作成できます。

---

**注意：** Load and Publish 開発モデルでサポートされているのは Java 静的メソッドのみです。静的メソッドと非静的（インスタンス）メソッドの両方を格納するには、「[モデル 2: アタッチ済みストアド・プロシージャ開発モデルの使用](#)」で説明されているように、データベース表にクラスをアタッチする必要があります。

---

### クラスのロード

Java クラスを Oracle Lite データベースにロードするには、次のいずれかを使用できます。

- loadjava
- SQL の CREATE JAVA コマンド

loadjava コマンドライン・ユーティリティは、Java クラスを Oracle Lite および Oracle8i データベースにロードする作業を自動化します。Java クラスを手動でロードするには、SQL の CREATE JAVA コマンドを使用します。

**loadjava** loadjava によりファイルからスキーマ・オブジェクトが作成され、そのオブジェクトがスキーマにロードされます。スキーマ・オブジェクトは、Java ソース、クラスおよびリソースから作成できます。リソースとしては、イメージ、SQLJ リソース、またはその他プロシージャがデータとしてアクセスする必要があるものであれば、何でもかまいません。ファイルは、個別に、または ZIP や JAR アーカイブ・ファイルとして loadjava に渡すことができます。

Oracle Lite ではクラスの依存性を追跡しません。ストアド・プロシージャが必要とするクラスとリソースが、すべてデータベースにロードされているか CLASSPATH に指定されていることを確認してください。

### 構文

loadjava の構文は、次のとおりです。

```
loadjava {-user | -u} username/password[@database]
        [-option_name -option_name ...] filename filename ...
```

### 引数

次に、loadjava の引数について詳しく説明します。

#### user

user 引数には、ユーザー名、パスワードおよびデータベース接続文字列を次の書式で指定します。

```
<user>/<password>[@<database>]
```

たとえば、次のとおりです。

```
scott/tiger@c:¥ORANT¥OLDB40¥Polite.odb
```

### オプション

Oracle Lite では、次のオプションをサポートします。

オプション	説明
-force   -f	ファイルのロードを施行します。すでにデータベースに同じ名前のスキーマ・オブジェクトが存在する場合でも施行されます。
-verbose   -v	実行中に詳細なステータス・メッセージを表示するように、loadjava に指示します。

複数のオプションを指定する場合は、オプション間をスペースで区切る必要があります。たとえば、次のとおりです。

```
-force -verbose
```

Oracle8i は、『Oracle8i Java ストアド・プロシージャ開発者ガイド』で説明されているように、その他のオプションもサポートします。その他のオプションが Oracle Lite で使用された場合、認識はされますがサポートはされません。このようなオプションを使用しても、エラーにはなりません。

Oracle8i でサポートされているオプションを表示するには、次の構文で loadjava のヘルプ情報を参照してください。

```
loadjava {-help | -h}
```

### filename

コマンドラインには、クラス、ソース、JAR、ZIP およびリソースを任意の順序でいくつでも指定できます。複数のファイル名を指定する場合は、カンマではなくスペースで区切る必要があります。ソースが渡されると、loadjava は Java コンパイラを起動して、ファイルを表にロードする前にコンパイルします。JAR ファイルまたは ZIP ファイルが渡されると、loadjava は JAR または ZIP 内の各ファイルを処理します。JAR または ZIP アーカイブの場合はスキーマ・オブジェクトは作成されません。loadjava は、JAR または ZIP アーカイブ内にある JAR または ZIP アーカイブは処理しません。

ファイルをロードする最善の方法は、JAR または ZIP 内にファイルを入れ、そのアーカイブをロードすることです。アーカイブをロードすることにより、リソースのスキーマ・オブジェクトの複雑な名前体系に煩わされません。JDK で有効な JAR または ZIP がある場合は、この JAR または ZIP を loadjava でロードできるため、リソースのスキーマ・オブジェクトの複雑なネーミングに煩わされなくてすみます。

loadjava では、データベースにファイルをロードするときに、ファイル用に作成するスキーマ・オブジェクトに名前を付ける必要があります。スキーマ・オブジェクトの名前は、ファイル名とは少し違います。また、異なるスキーマ・オブジェクトには異なるネーミング規則があります。クラスは名前で識別できるため、loadjava は自動的にファイル名をスキーマ名にマップできます。同様に、JAR および ZIP にはその中にあるファイルの名前が含まれています。

しかし、リソースは名前では識別できないため、コマンドラインに入力されるリテラル名（または JAR や ZIP アーカイブ内のリテラル名）から Java リソースのスキーマ・オブジェクト名が導出されます。クラスの実行中にリソースのスキーマ・オブジェクトが使用されるため、コマンドラインには正しいリソース名を指定することが重要です。

リソースを個別にロードする最善の方法は、loadjava をパッケージ・ツリーの最上位のディレクトリから実行し、そのディレクトリを基準にした相対的なリソース名を指定することです。リソースのロードをパッケージ・ツリーの最上位から行わない場合は、リソースの命名に関する次の情報を考慮してください。

リソースをロードするとき、loadjava は、コマンドラインに入力されたファイル名からリソースのスキーマ・オブジェクトの名前を導出します。次のような相対パス名と絶対パス名をコマンドラインに入力したとします。

```
cd %scott%javastuff
loadjava options alpha%beta%x.properties
loadjava options %scott%javastuff%alpha%beta%x.properties
```

同一ファイルを相対パス名と絶対パス名で指定しましたが、loadjava は 2 つのスキーマ・オブジェクトを作成します。

- alpha%beta%x.properties
- %scott%javastuff%alpha%beta%x.properties

loadjava は、入力されたファイル名からリソースのスキーマ・オブジェクトの名前を生成します。

---

**注意：** Oracle Lite では、スラッシュで始まるスキーマ・オブジェクト名を使用できます。Oracle8i では、スラッシュで始まるスキーマ・オブジェクトの前に「ROOT」が追加されます。

---

クラスからのリソースの参照は、相対的（たとえば、b.properties）または絶対的（たとえば、%a%b.properties）に指定できます。loadjava とクラス・ローダーでスキーマ・オブジェクトに必ず同じ名前が使用されるようにするには、クラスから java.lang.Object.getResource メソッドまたは java.lang.Class.getResourceAsStream メソッドに渡されるリソースの名前を loadjava に渡します。

コマンドラインに正しい名前を入力できるように、クラスがリソースの指定に使用している相対名あるいは絶対名を覚えておいてディレクトリを変更するという作業のかわりに、次のように、リソースを JAR ファイルにロードできます。

```
cd %scott%javastuff
jar -cf alphasources.jar alpha%*.properties
loadjava options alphasources.jar
```

または、もっと簡単にクラスとリソースの両方を JAR に入れると、次の起動方法が同じになります。

```
loadjava options alpha.jar
loadjava options %scott%javastuff%alpha.jar
```

### 例

次の指定により、クラスとリソースが Oracle Lite データベースにロードされます。この指定では force オプションが使用されているため、指定された名前のオブジェクトがすでにデータベースにある場合は、loadjava により置き換えられます。

```
c:¥> loadjava -u scott/tiger@c:¥ORANT¥OLDB40¥Polite.odbc -f Agent.class ¥
images.dat
```

### CREATE JAVA の使用法

Java クラスを手動でロードするには、次の構文を使用します。

```
CREATE [OR REPLACE] [AND RESOLVE] [NOFORCE]
  JAVA {CLASS [SCHEMA <schema_name>] |
  RESOURCE NAMED [<schema_name>.<primary_name>]
  [<invoker_rights_clause>]
  RESOLVER <resolver_spec>]
  USING BFILE ('<dir_path>', '<class_name>')
```

CREATE JAVA のパラメータは、次のとおりです。

- OR REPLACE 句を指定すると、同じ名前のファンクションまたはプロシージャがデータベースにすでに存在する場合、再作成されます。
- Oracle8i と互換性を保つために、Oracle Lite では <resolver\_spec> 句は認識されませんが処理は行いません。Oracle8i とは異なり、Oracle Lite ではクラスの依存性は解決されません。クラスを手動でロードする場合は、必ず依存クラスをすべてロードするようにしてください。
- Oracle Lite では <invoker\_rights\_clause> が認識されますが、処理は行いません。
- CREATE JAVA SOURCE 文は Oracle8i ではサポートされていますが、Oracle Lite ではサポートされません。

Java 文を埋め込む際は、SQL\*Plus 文の最後にセミコロン文字 (;) を使用しないでください。行の最後の文字をセミコロンにする必要がある場合は、/\*\*/ の文字を使用して、空のコメント行を追加する必要があります。このコンテキストでは、通常のコメント記号 (//) が機能しません。行の最後に /\*\*/ を置くことで、SQL\*Plus によりセミコロンが SQL 文の末尾として解釈されることを防ぎます。

### 例

次に CREATE JAVA 文の例を示します。この例では、Employee というクラスがデータベースにロードされます。

```
CREATE JAVA CLASS USING BFILE ('c:¥myprojects¥java',
  'Employee.class');
```



## SQL へのストアド・プロシージャのパブリッシュ

loadjava または CREATE JAVA を使用してクラスを Oracle Lite データベースにロードした後は、クラス内の静的メソッドで SQL からコールするメソッドをパブリッシュします。メソッドをパブリッシュするには、そのためのコール仕様を作成します。コール仕様は、Java メソッドの名前、パラメータ・タイプおよび戻り型を、対応する SQL 要素にマップします。

ストアド・プロシージャをすべてパブリッシュする必要はありません。アプリケーションのエントリ・ポイントとして機能するもののみをパブリッシュします。通常の実装では、ストアド・プロシージャのほとんどは SQL ユーザーではなく他のストアド・プロシージャからのみコールされます。

コール仕様を作成するには、SQL コマンドの CREATE FUNCTION または CREATE PROCEDURE を使用します。値を返すメソッドには CREATE FUNCTION を、値を返さないメソッドには CREATE PROCEDURE を使用します。CREATE FUNCTION 文と CREATE PROCEDURE 文の構文は次のとおりです。

```
CREATE [OR REPLACE]
  { PROCEDURE [<schema_name>.<proc_name> [[<sql_parms>]] |
    FUNCTION [<schema_name>.<func_name> [[<sql_parms>]]
  RETURN <sql_type> }
<invoker_rights_clause>
{ IS | AS } LANGUAGE JAVA NAME
'<java_fullname> ([<java_parms>])
[return <java_type_fullname>]';
/
```

この文のキーワードとパラメータは次のとおりです。

- <sql\_parms> は次の書式です。
 

```
<arg_name> [IN | OUT | IN OUT]
          <datatype>
```
- <java\_parms> は、Java データ型の完全修飾名です。
- Oracle8i との互換性を保つために、Oracle Lite では <invoker\_rights\_clause> 句は認識されますが処理は行いません。
- <java\_fullname> は、静的 Java メソッドの完全修飾名です。

---

**注意：** SQL\*Plus では、CREATE PROCEDURE 文と CREATE FUNCTION 文の後ろにスラッシュ (/) を指定する必要があります。

---

たとえば、次のクラスがデータベースにロードされているとします。

```
import java.sql.*;
import java.io.*;

public class GenericDrop {
    public static void dropIt (Connection conn, String object_type,
        String object_name) throws SQLException {
        // Build SQL statement
        String sql = "DROP " + object_type + " " + object_name;
        try {
            Statement stmt = conn.createStatement();
            stmt.executeUpdate(sql);
            stmt.close();
        } catch (SQLException e) {
            System.err.println(e.getMessage());
        } // dropIt
    } // GenericDrop
}
```

クラス GenericDrop には、dropIt という名前のメソッドが1つあり、このメソッドはどのような種類のスキーマ・オブジェクトでも削除します。たとえば、引数に「table」と「emp」を dropIt に渡すと、データベース表 EMP をスキーマから削除します。

次のコール仕様により、このメソッドが SQL にパブリッシュされます。

```
CREATE OR REPLACE PROCEDURE drop_it (
    obj_type VARCHAR2,
    obj_name VARCHAR2)
AS LANGUAGE JAVA
NAME 'GenericDrop.dropIt(java.sql.Connection,
    java.lang.String, java.lang.String)';
/
```

Java データ型パラメータには完全修飾名を指定する必要があるので、注意してください。

### パブリッシュされたストアド・プロシージャのコール

ストアド・プロシージャを SQL にパブリッシュした後、SQL DML 文を使用してこのプロシージャをコールします。たとえば、次のクラスがデータベース内に格納されているとします。

```
public class Formatter {
    public static String formatEmp (String empName, String jobTitle) {
        empName = empName.substring(0,1).toUpperCase() +
            empName.substring(1).toLowerCase();
        jobTitle = jobTitle.toLowerCase();
        if (jobTitle.equals("analyst"))
            return (new String(empName + " is an exempt analyst"));
    }
}
```

```

        else
            return (new String(empName + " is a non-exempt " + jobTitle));
    }
}

```

クラス `Formatter` には `formatEmp` というメソッドが1つあり、このメソッドは、従業員の名前と職務状態を含む書式化された文字列を返します。`Formatter` 用のコール仕様を次のように作成します。

```

CREATE OR REPLACE FUNCTION format_emp (ename VARCHAR2, job VARCHAR2)
RETURN VARCHAR2
AS LANGUAGE JAVA
NAME 'Formatter.formatEmp (java.lang.String, java.lang.String)
return java.lang.String';
/

```

このコール仕様により、メソッド `formatEmp` が `format_emp` としてパブリッシュされます。これを次のようにコールします。

```

SELECT FORMAT_EMP(ENAME, JOB) AS "Employees" FROM EMP
WHERE JOB NOT IN ('MANAGER', 'PRESIDENT') ORDER BY ENAME;

```

この文により、次の出力が生成されます。

```

Employees
-----
Adams is a non-exempt clerk
Allen is a non-exempt salesman
Ford is an exempt analyst
James is a non-exempt clerk
Martin is a non-exempt salesman
Miller is a non-exempt clerk
Scott is an exempt analyst
Smith is a non-exempt clerk
Turner is a non-exempt salesman
Ward is a non-exempt salesman

```

---

**注意：** Oracle Lite では、Oracle8i SQL の CALL 文を使用してストアド・プロシージャを起動することはできません。

C および C++ アプリケーションからストアド・プロシージャをコールする方法は、「[Java ストアド・プロシージャの ODBC からのコール](#)」を参照してください。

---

## パブリッシュされたストアド・プロシージャの削除

Oracle Lite からクラスを削除するには、次のいずれかを使用します。

- dropjava ユーティリティ
- SQL の DROP JAVA 文

コール仕様を削除するには、DROP FUNCTION または DROP PROCEDURE を使用します。

**dropjava の使用** dropjava は、Oracle Lite および Oracle8i データベースから Java クラスを削除する作業を自動化するコマンドライン・ユーティリティです。dropjava は、ファイル名をスキーマ・オブジェクト名に変換し、そのスキーマ・オブジェクトを削除します。dropjava の起動には次の構文を使用します。

```
dropjava {-user | -u} username/password[@database]
        [-option] filename filename ...
```

### 引数

次に、dropjava の引数について説明します。

#### user

user 引数には、ユーザー名、パスワードおよびデータベース接続文字列を次の書式で指定します。

```
<user>/<password>[@<database>]
```

たとえば、次のとおりです。

```
scott/tiger@c:¥ORANT¥OLDB40¥Polite.odb
```

### オプション

verbose オプション (-verbose | -v) を指定すると、実行中に詳しいステータス・メッセージを生成するように dropjava に指示できます。

Oracle8i は、『Oracle8i Java ストアド・プロシージャ開発者ガイド』で説明されているように、dropjava のその他のオプションもサポートします。その他のオプションが Oracle Lite で使用された場合、認識はされますがサポートはされません。このようなオプションを使用しても、エラーにはなりません。

サポートされるオプションと認識されるオプションの一覧を表示するには、コマンド・プロンプトで次のように入力します。

```
dropjava -help
```

## filename

filename 引数には、Java クラス、ソース、JAR、ZIP およびリソースを任意の順序でいくつでも指定できます。JAR および ZIP ファイルは圧縮を解凍する必要があります。dropjava では、ほとんどのファイル名を loadjava と同じ方法で解釈します。

- クラスを指定した場合、ファイル内でクラス名を検索し、それに対応するスキーマ・オブジェクトを削除します。
- ソースを指定した場合、ファイル内の最初のクラス名を検索し、それに対応するスキーマ・オブジェクトを削除します。
- JAR および ZIP ファイルを指定した場合、アーカイブされているファイルの名前がコマンドラインで入力されたかのように、ファイル名を処理します。

ファイル名に **.java**、**.class**、**.jar** または **.zip** 以外の拡張子がある場合、または拡張子がない場合は、ファイル名がスキーマ・オブジェクト名であると解釈され、その名前のソース、クラス、またはリソースのスキーマ・オブジェクトがすべて削除されます。スキーマ・オブジェクトのどの名前とも一致しないファイル名が検出されると、エラー・メッセージが表示され、残りのファイル名が処理されます。

**SQL コマンドの使用法** Oracle Lite データベースから Java クラスを手動で削除するには、DROP JAVA 文を使用します。DROP JAVA の構文は次のとおりです。

```
DROP JAVA { CLASS | RESOURCE } [<schema-name> .]<object_name>
```

コール仕様を削除するには、DROP FUNCTION 文または DROP PROCEDURE 文を使用します。

```
DROP { FUNCTION | PROCEDURE } [<schema-name>.<object_name>
```

スキーマ名が指定された場合、Oracle Lite では認識されますがサポートはされません。

## 例

次の例では、Load and Publish モデルを使用して Java ストアド・プロシージャを作成します。

この例では、Java メソッド paySalary を Oracle Lite データベースに格納します。paySalary は、従業員の手取り給与を計算します。Oracle Lite はカレントの Connection オブジェクトを paySalary に自動的に渡します。

この例には、次の手順が含まれています。

- **手順 1: Java クラスの作成**
- **手順 2: データベースへの Java クラスのロード**
- **手順 3: ファンクションのパブリッシュ**
- **手順 4: ファンクションの実行**

Java ストアド・プロシージャの例は、他にも `Oracle_Home¥Mobile¥Sdk¥Examples¥Java` ディレクトリにあります。

### 手順 1: Java クラスの作成

**Employee.java** ファイルに Java クラス `Employee` を作成します。Employee クラスは `paySalary` メソッドを実装します。

```
import java.sql.*;
public class Employee {
    public static String paySalary(float sal, float fica, float sttax,
        float ss_pct, float espp_pct) {
        float deduct_pct;
        float net_sal;
        // compute take-home salary
        deduct_pct = fica + sttax + ss_pct + espp_pct;
        net_sal = sal * deduct_pct;
        String returnstmt = "Net salary is " + net_sal;
        return returnstmt;
    } // paySalary
}
```

---

---

**注意：** 最初の PUBLIC クラス文の前に、コメント内でキーワードの「public class」を使用しないでください。

---

---

### 手順 2: データベースへの Java クラスのロード

SQL\*Plus から、次のように CREATE JAVA を使用して Java クラスをロードします。

```
CREATE JAVA CLASS USING BFILE ('c:¥myprojects¥doc',
'Employee.class');
```

このコマンドは、`c:¥myprojects¥doc` にある Java クラスを Oracle Lite データベースにロードします。

### 手順 3: ファンクションのパブリッシュ

`paySalary` メソッド用のコール仕様を作成します。次のコール仕様は、Java メソッド `paySalary` をファンクション `pay_salary` としてパブリッシュします。

```
CREATE FUNCTION pay_salary (
sal float, fica float, sttax float, ss_pct float, espp_pct float)
RETURN VARCHAR2
AS LANGUAGE JAVA NAME
'Employee.paySalary(float, float, float, float, float)
return java.lang.String';
/
```

#### 手順 4: ファンクションの実行

SQL\*Plus 内で `pay_salary` を実行するには、次のようにします。

```
SELECT pay_salary(6000.00, 0.2, 0.0565, 0.0606, 0.1)
FROM DUAL;
```

ODBC 内で `pay_salary` を実行するには、次のようにします。

```
SQLExecDirect (hstm,
    "SELECT pay_salary (6000.00,0.2,0.0565,0.0606,0.1)
    FROM DUAL);
```

`pay_salary` への引数が定数のため、FROM 句ではダミー表 `DUAL` を指定します。この SELECT 文により、次の出力が生成されます。

```
Net salary is 2502.6
```

## モデル 2: アタッチ済みストアド・プロシージャ開発モデルの使用

この項では、クラスを表にアタッチしてストアド・プロシージャを作成する方法を説明します。この方法は Oracle Lite 固有のもので、ここで説明されている方法で Oracle8i の表にクラスをアタッチすることはできません。「[モデル 1: Load and Publish ストアド・プロシージャ開発モデルの使用](#)」に説明されているストアド・プロシージャ開発用の Load and Publish モデルがサポートするのは、クラスの（静的）メソッドのみですが、クラスを表にアタッチすると、Java クラスとインスタンスのメソッドを格納しコールできます。

アタッチ済みストアド・プロシージャを作成するには、アタッチするクラスを開発します。クラスがエラーなしでコンパイルされ実行されることを確認します。次に、クラスを Oracle Lite の表にアタッチします。クラスがアタッチされると、クラス内のメソッドがその表の表レベルおよび行レベルのストアド・プロシージャになります。

### 表レベルのストアド・プロシージャ

表レベルのストアド・プロシージャは、Java クラスの静的メソッドです。このため、メソッドの実行時に Oracle Lite がメソッドの属するクラスをインスタンス化することはありません。コール文の中では、表レベルのストアド・プロシージャは `table_name.method_name` として参照されます。

文レベルのトリガーと BEFORE INSERT および AFTER DELETE 行レベル・トリガー（「[文レベルのトリガーと行レベルのトリガーの比較](#)」を参照）は、表レベルのストアド・プロシージャであることが必要です。

## 行レベルのストアド・プロシージャ

行レベルのストアド・プロシージャは、Java クラスの非静的メソッドです。行レベルのストアド・プロシージャを実行するために、Oracle Lite はそのプロシージャが属するクラスをインスタンス化します。クラス・コンストラクタへの引数によって、コンストラクタがクラス・インスタンスの作成にパラメータとして使用する列値が決定されます。コール文の中では、行レベルのストアド・プロシージャは *method\_name* (表修飾子なし) として参照されます。行レベル・トリガーは、行レベル・ストアド・プロシージャを間接的に実行できます。

## 表への Java クラスのアタッチ

表に Java クラスをアタッチするには、SQL コマンドの ALTER TABLE を使用します。ALTER TABLE コマンドの構文は、次のとおりです。

```
ALTER TABLE [schema.]table
  ATTACH JAVA {CLASS|SOURCE} "cls_or_src_name "
  IN {DATABASE|'cls_or_src_path '}
  [WITH CONSTRUCTOR ARGS (col_name_list )]
```

アタッチできるのは、ソースまたはクラスです。ソースは、システム・パスにある Java コンパイラによりコンパイルされます。

*cls\_or\_src\_name* は、クラスまたはソースの完全修飾名を示します。Oracle.Lite.Customer のように、パッケージ名の後にクラス名が続きます。クラスまたはソース・ファイル名にはファイル拡張子を含めないでください。名前の大 / 小文字は区別されます。小文字を使用する場合は、名前を二重引用符 ("" ) で囲みます。

*cls\_or\_src\_name* で指定されるパッケージ内にソースまたはクラスがあることを確認します (サンプル・クラス Customer のソースには、行「package Oracle.Lite;」が含まれています)。クラスは、データベースの同一パッケージ内に格納されます。パッケージが存在していない場合は Oracle Lite により作成されます。

Java クラスをデータベースの別の表にすでにアタッチしてある場合は、IN DATABASE 句を使用できます。クラスがまだアタッチされていない場合は、クラスまたはソースのディレクトリ位置を *cls\_or\_src\_path* に指定します。

行レベルのストアド・プロシージャを実行する前に、Oracle Lite がその行に Java オブジェクトを作成します (Java オブジェクトが存在しない場合)。ALTER TABLE 文に WITH CONSTRUCTOR 句が含まれている場合は、*col\_name\_list* に含まれている列のデータ型に最適のクラス・コンストラクタを使用して、Oracle Lite がこのオブジェクトを作成します。ALTER TABLE 文に WITH CONSTRUCTOR 句が含まれていない場合は、Oracle Lite はデフォルトのコンストラクタを使用します。

表の中に定義されているメソッドに関する情報を取り出すには、ODBC 関数の SQLProcedures と SQLProcedureColumns を使用できます。



ALTER TABLE 文については、次のことに注意してください。

- ALTER TABLE 文に引数を指定した場合、Oracle Lite は引数リストで渡される値を使用してクラスのインスタンスを作成します。その後で Java アプリケーションがこれらのパラメータの値を変更しても、新しい値はデータベースに自動的にポストされません。新しい値でデータベースを更新するには、アプリケーションで JDBC コールまたは JAC コールを使用する必要があります。
- ALTER TABLE 文でコンストラクタが指定されていない場合に行レベル・トリガーが起動されると、Oracle Lite がデフォルト・コンストラクタを使用してオブジェクトを作成します。

## アタッチ済みストアド・プロシージャのコール

ALTER TABLE 文を使用してプロシージャを表にアタッチした後は、SELECT 文を使用してこのプロシージャをコールできます。表レベルのストアド・プロシージャは *table\_name.method\_name* として参照し、行レベルのプロシージャは *method\_name* として参照します。

たとえば、表レベルのストアド・プロシージャを実行するには、次のように指定します。

```
SELECT table_name.proc_name [arg_list]
FROM {DUAL| [schema.]table WHERE condition};
```

*proc\_name* は、表レベルのストアド・プロシージャの名前です。*arg\_list* の各引数は、定数かまたは表内の列への参照です。*arg\_list* のすべての引数が定数の場合は、FROM 句でダミー表 DUAL を参照する必要があります。

行レベルのストアド・プロシージャは、次のように実行します。

```
SELECT [schema.]proc_name [arg_list]
FROM [schema.]table
WHERE condition;
```

プロシージャを *table\_name.method\_name* の書式でコールし、その名前の表またはメソッドが存在しない場合、Oracle Lite では、*table\_name* 名がスキーマ名を意味し、*method\_name* がプロシージャ名を意味すると解釈されます。*method\_name* のみを参照する場合、Oracle Lite では参照されたメソッドが行レベル・プロシージャであると解釈されます。ただし、そのようなプロシージャが定義されていない場合、Oracle Lite では *method\_name* が現在のスキーマ内のプロシージャを意味すると解釈されます。

---

---

**注意：** Oracle Lite では、Oracle8i SQL の CALL 文を使用してストアド・プロシージャを起動できません。

Callable Statement を使用して、ODBC または JDBC アプリケーションからプロシージャを実行できます。詳細は、[第3章「Java Database Connectivity \(JDBC\)」](#)を参照してください。また、「[Java ストアド・プロシージャの ODBC からのコール](#)」も参照してください。

---

---

## アタッチ済みストアド・プロシージャの削除

ストアド・プロシージャを削除するには、ALTER TABLE コマンドを使用します。ALTER TABLE コマンドの構文は、次のとおりです。

```
ALTER TABLE [schema.] table
  DETACH [AND DELETE] JAVA CLASS "class_name"
```

---

---

**注意：** クラス名に小文字が含まれている場合は、クラス名を二重引用符 (" ") で囲む必要があります。

---

---

Java クラスをデタッチしてもデータベースからは削除されません。データベースから Java クラスを削除するには、DETACH AND DELETE 文を使用します。

Java クラスをストアド・プロシージャまたはトリガーとして起動した後に、その Java クラスをデータベースから削除した場合、そのクラスはアプリケーションにアタッチされている JVM の中に残ります。クラスを JVM からアンロードするには、(必要な場合) データベースの変更をコミットし、データベースに接続しているアプリケーションをすべてクローズします。Java クラスを置き換えるには、データベースへの接続をすべてクローズしてから Java クラスをリロードする必要があります。

## 例

次の例は、Oracle Lite に Java ストアド・プロシージャを作成する方法を示します。この例では、Java メソッド `paySalary` を表 EMP にアタッチします。`paySalary` は、従業員の手取り給与を計算します。Oracle Lite はカレントの Connection オブジェクトを最初の引数として `paySalary` に渡します。

この例には、次の手順が含まれています。

- 手順 1: 表を作成する
- 手順 2: Java クラスを作成する

- 手順 3: Java クラスを表へアタッチする
- 手順 4: メソッドを実行する

### 手順 1: 表を作成する

次の SQL コマンドを使用して表を作成します。

```
CREATE TABLE EMP(Coll char(10));
```

### 手順 2: Java クラスを作成する

**Employee.java** ファイルに Java クラス Employee を作成します。Employee クラスは paySalary メソッドを実装します。

```
import java.sql.*;
public class Employee {
    public static String paySalary(float sal, float fica, float sttax,
                                   float ss_pct, float espp_pct) {
        float deduct_pct;
        float net_sal;
        // compute take-home salary
        deduct_pct = fica + sttax + ss_pct + espp_pct;
        net_sal = sal * deduct_pct;
        String returnstmt = "Net salary is " + net_sal;
        return returnstmt;
    } // paySalary
}
```

### 手順 3: Java クラスを表へアタッチする

SQL\*Plus から、次のように ALTER TABLE コマンドを使用して Java クラスをアタッチします。

```
ALTER TABLE EMP ATTACH JAVA SOURCE "Employee" IN 'C:¥tmp';
```

このコマンドは、ディレクトリ C:¥tmp にある Employee クラスの Java ソースを EMP 表にアタッチします。

### 手順 4: メソッドを実行する

paySalary メソッドを SQL\*Plus で実行するには、次の文を入力します。

```
SELECT EMP."paySalary"(6000.00,0.2,0.0565,0.0606,0.1)
FROM DUAL;
```

ODBC から paySalary を実行するには、SQLExecDirect を起動します。

```
SQLExecDirect(hstm,
"SELECT EMP.¥"paySalary¥"(6000.00,0.2,0.0565,0.0606,0.1)
FROM DUAL);
```

この文により、次の結果が生成されます。

```
Net salary is 2502.6
```

## Java ストアド・プロシージャの ODBC からのコール

Java ストアド・プロシージャを起動するには、ODBC 関数 `SQLExecute` と `SQLExecuteDirect` を使用できます。`SQLExecute` は、`SQLPrepare` を使用して作成された SQL 文へのハンドルを引数として取ります。`SQLExecute` と `SQLPrepare` の使用例は、[「ODBC を使用して複数行を返す方法」](#) を参照してください。

C または C++ のマルチスレッド・アプリケーションから Java ストアド・プロシージャを起動するには、アプリケーションの main 関数から `loadjvm40.dll` をロードする必要があります。これにより、C または C++ アプリケーションがストアド・プロシージャを直接または間接的に起動する複数のスレッドを作成するときに、JVM のガベージ・コレクションで発生する問題が解決されます。スレッドは、終了するまで JVM からデタッチされないため、JVM のメモリーが足りなくなります。Oracle Lite では、そのスレッドを作成したのが JVM かユーザー・アプリケーションかを判断できないため、スレッドのデタッチは行われません。

main では、次のように、すべてのアクションの前にライブラリをロードする必要があります。

```
int main (int argc, char** argv)
{
    LoadLibrary("loadjvm40.dll");
    ...
}
```

このライブラリは、JVM をアプリケーションのメイン・スレッドにロードします。このライブラリは、スレッドがプロセスからデタッチされた場合は、スレッドを JVM からデタッチしようとします。`loadjvm40.dll` は、スレッドが JVM にアタッチされていない場合でも正しく動作します。

## Java データ型

Oracle Lite は、標準 SQL の規則に基づいて Java と SQL の間でデータ型の変換を実行します。たとえば、文字列を引数として取るストアド・プロシージャに整数を渡すと、Oracle Lite はその整数を文字列に変換します。行レベル・トリガーの引数の詳細は、「[トリガーの引数](#)」を参照してください。Java と SQL 間のデータ型の対応付け一覧は、[第 1 章「Oracle Lite の Java サポート」](#)の「[Java データ型](#)」を参照してください。

---

---

**注意：** Oracle8i では、DATE 列は TIMESTAMP として作成されます。トリガー・メソッドもそのように指定する必要があります。

---

---

Java では、メソッドの有効範囲外にある引数の値をメソッドで変更できません。ただし、Oracle Lite では、IN、OUT および IN/OUT パラメータをサポートします。

Java データ型の多くは不変であるか、NULL 値をサポートしません。このようなデータ型に NULL 値を渡して IN/OUT パラメータを使用する場合は、ストアド・プロシージャでその型の配列を使用するか、NULL をサポートするかわりのデータ型を使用します。次の表は、Java の整数データ型に IN/OUT パラメータを使用できるか、NULL 値を設定できるかを示しています。

Java の引数	IN/OUT	NULL
int	いいえ	いいえ
int []	はい	いいえ
Integer	いいえ	はい
Integer []	はい	はい
int [] []	はい	はい

Date などの可変 Java データ型を使用すると、NULL または IN/OUT パラメータを渡すことができますが、ストアド・プロシージャから引数の NULL 状態を変更する必要がある場合は、Date 配列を使用してください。

---

---

**注意：** NULL ではあり得ない Java 引数に NULL を渡すと、エラーが発生します。

---

---

## パラメータの宣言

Java メソッドの戻り値は、プロシージャの OUT パラメータに設定できます。プリミティブ型または不変参照型は、IN パラメータに設定できます。可変参照型または配列型は、IN/OUT パラメータに設定できます。次の表は、対応する Oracle Lite パラメータを IN/OUT パラメータにするために使用する Java データ型を示します。

IN/OUT パラメータの型	使用するデータ型
NUMBER	Number []
Binary	byte [] または byte [] []
String	string []

ストアド・プロシージャが `java.sql.Connection`、`POLPersistentCapable` または `POLConnection` 引数を取る場合、Oracle Lite はカレント・トランザクションまたはカレント行の値を使用して、引数を自動的に指定します。この引数は、プロシージャに渡される最初の引数です。

## ストアド・プロシージャを使用して複数行を返す方法

ストアド・プロシージャを使用して複数行を返すことができます。ただし、複数行を返すストアド・プロシージャを起動できるのは、JDBC アプリケーションまたは ODBC アプリケーションからのみです。ストアド・プロシージャで複数行を返すには、対応する Java メソッドが `java.sql.ResultSet` オブジェクトを返す必要があります。SELECT 文字列を実行することにより、Java メソッドは `ResultSet` オブジェクトを取得して返します。`ResultSet` の列名は SELECT 文に指定します。結果列を、表で使用されている名前ではなく別の名前と呼ぶ必要がある場合は、SELECT 文で結果列の別名を使用する必要があります。たとえば、次のとおりです。

```
SELECT emp.name Name, dept.Name Dept
   FROM emp, dept
  WHERE emp.dept# = dept.dept#;
```

複数行を返すストアド・プロシージャの戻り型は `java.sql.ResultSet` である必要があるため、結果の列名または型の取得に、そのストアド・プロシージャのシグネチャは使用できません。この結果、ストアド・プロシージャの結果の列名または型を追跡する別の表を設計する必要があります。たとえば、前述の SELECT 文を Java メソッドに埋め込んだ場合、このメソッドの戻り型は `char Name` や `char Dept` ではなく、`java.sql.ResultType` です。

---

---

**注意：** 複数行を返す Java ストアド・プロシージャの作成に使用できるのは、「[モデル 2: アタッチ済みストアド・プロシージャ開発モデルの使用](#)」で説明されているアタッチ済みストアド・プロシージャ開発モデルのみです。

---

---

## ODBC を使用して複数行を返す方法

複数行を返すストアド・プロシージャを ODBC アプリケーションで実行するには、次の CALL 文を使用します。P はストアド・プロシージャの名前で、 $a_1$  から  $a_n$  はストアド・プロシージャへの引数です。

```
{CALL P( $a_1, \dots, a_n$ )}
```

文を実行する前に値をバインドする必要がある引数には、マーカー (?) を使用します。文が実行されるとプロシージャが実行され、結果セット上のカーソルが文ハンドル内に格納されます。文ハンドルを使用して順次フェッチを行い、プロシージャから行を返します。

CALL 文を実行した後は、SQLNumResultCols を使用してそれぞれの結果行の列数を見つけ出します。列の名前とデータ型を返すには、SQLDescribeCol 関数を使用します。

## 例

次の例は、ODBC を使用して、複数行を返すストアド・プロシージャを実行する方法を示します。この例では、SQLNumResultCols 関数も SQLDescribeCol 関数も使用されていません。ここでは、ストアド・プロシージャが作成済みで、PROC として SQL にパブリッシュされていると想定されています。PROC は引数として整数を取ります。

```
rc = SQLPrepare(StmtHdl, "{call PROC(?)}", SQL_NTS);
CHECK_STMT_ERR(StmtHdl, rc, "SQLPrepare");

rc = SQLBindParameter(StmtHdl, 1, SQL_PARAM_INPUT_OUTPUT,
    SQL_C_LONG,SQL_INTEGER, 0, 0, &InOutNum, 0, NULL);
CHECK_STMT_ERR(StmtHdl, rc, "SQLBindParameter");

rc = SQLExecute(StmtHdl);
CHECK_STMT_ERR(StmtHdl, rc, "SQLExecute");

/* you can use SQLNumResultCols and SQLDescribeCol here */

rc = SQLBindCol(StmtHdl, 1, SQL_C_CHAR, c1, 20, &pcbValue1);
CHECK_STMT_ERR(StmtHdl, rc, "SQLBindCol");

rc = SQLBindCol(StmtHdl, 2, SQL_C_CHAR, c2, 20, &pcbValue2);
CHECK_STMT_ERR(StmtHdl, rc, "SQLBindCol");
```

```
while ((rc = SQLFetch(StmtHdl)) != SQL_NO_DATA_FOUND) {
    CHECK_STMT_ERR(StmtHdl, rc, "SQLFetch");
    printf("%s, %s\n", c1, c2);
}
```

## トリガーの使用法

トリガーとは、特定のイベントが発生したときに実行（起動）されるストアド・プロシージャのことです。トリガーは、列が更新されたとき、または行が追加あるいは削除されたときに起動されます。トリガーはイベントの前または後で起動できます。

トリガーは、一般的には、データベースのビジネス・ルールを施行するために使用します。たとえば、トリガーは入力値を検証し、無効な挿入を拒否できます。同様に、トリガーを使用すると、特定の行を削除する前に、その行に依存するすべての表を一貫性のある状態にすることができます。

## 文レベルのトリガーと行レベルのトリガーの比較

トリガーには、行レベルと文レベルの2種類があります。行レベル・トリガーは、データベースへの変更により影響される各行に対して1回起動されます。文レベル・トリガーは、変更により複数行が影響される場合でも、1回のみ起動されます。

行オブジェクトのインスタンスを生成してプロシージャをコールできないため、BEFORE INSERT トリガーと AFTER DELETE トリガーが起動できるのは、表レベルのストアド・プロシージャのみです。AFTER INSERT、BEFORE DELETE および UPDATE の各トリガーでは、表レベルまたは行レベルのストアド・プロシージャを起動できます。

## トリガーの作成

トリガーを作成するには、CREATE TRIGGER コマンドを使用します。CREATE TRIGGER 文の構文は次のとおりです。

```
CREATE [OR REPLACE] TRIGGER trigger_name {BEFORE | AFTER} [{INSERT | DELETE |
UPDATE [OF column_list]} [OR ]] ON table_reference
[FOR EACH ROW] procedure_ref
(arg_list)
```

CREATE TRIGGER の構文では、次の点に注意してください。

- OR 句は、複数のトリガー・イベントの指定に使用します。
- FOR EACH ROW は、行レベル・トリガーの作成に使用します。表レベル・トリガーの場合は、この句は指定しません。
- procedure\_ref は、実行するストアド・プロシージャの識別に使用します。



1つの表に対して、同じ種類のトリガーを複数作成できます。ただし、各トリガーにはスキーマ内で一意の名前を指定する必要があります。

次の例では、プロシージャを `PROCESS_NEW_HIRE` として格納しパブリッシュしてあるものとします。トリガー `AIEMP` は、`EMP` 表に行が1つ挿入されるたびに起動されます。

```
CREATE TRIGGER AIEMP AFTER INSERT ON EMP FOR EACH ROW
PROCESS_NEW_HIRE(ENO);
```

表の複数の列に対して同じストアド・プロシージャを使用する `UPDATE` トリガーは、文中で列のサブセットが変更されたときに1回のみ起動されます。たとえば、次の文は表 `T` に `BEFORE UPDATE` トリガーを作成します。表 `T` には、`C1`、`C2`、`C3` という列があります。

```
CREATE TRIGGER T_TRIGGER BEFORE UPDATE OF C1,C2,C3 ON T
FOR EACH ROW trigg(old.C1,new.C1,old.C2,new.C2,
old.C3,new.C3);
```

この `UPDATE` 文は、`T_TRIGGER` を1回のみ起動します。

```
UPDATE T SET C1 = 10, C2 = 10 WHERE ...
```

## トリガーの使用可能と使用禁止

トリガーは、作成すると自動的に使用可能になります。トリガーを使用禁止にするには、`ALTER TABLE` 文か `ALTER TRIGGER` 文を使用します。

トリガーを個別に使用可能または使用禁止にするには、`ALTER TRIGGER` 文を使用します。`ALTER TRIGGER` 文の構文は次のとおりです。

```
ALTER TRIGGER <trigger_name> {ENABLE | DISABLE}
```

表にアタッチされているすべてのトリガーを使用可能または使用禁止にするには、`ALTER TABLE` 文を使用します。

```
ALTER TABLE <table_name> {ENABLE | DISABLE} ALL TRIGGERS
```

## トリガーの削除

トリガーを削除するには、`DROP TRIGGER` 文を使用します。`DROP TRIGGER` 文の構文は次のとおりです。

```
DROP TRIGGER [schema.]trigger
```

## トリガーの例

次の例ではトリガーを作成します。「[モデル 2: アタッチ済みストアド・プロシージャ開発モデルの使用](#)」で説明されている開発モデルに従って作成します。Load and Publish モデルを使用したトリガー作成の例は、「[トリガーの引数の例](#)」を参照してください。この例では、まず、表と Java クラスを作成します。次に、クラスを表にアタッチします。最後に、トリガーを作成して起動します。

SalaryTrigger クラスには check\_sal\_raise メソッドが含まれています。このメソッドは、従業員が 10% を超える昇給を受ける場合はメッセージを出力します。EMP 表内の給与を更新する前に、トリガーがメソッドをコールします。

check\_sal\_raise はメッセージを標準出力に書き込むため、この例にある SQL コマンドは DOS 版の SQL\*Plus を使用して発行します。DOS コマンド・プロンプトで、次のように入力して SQL\*Plus を起動します。

```
plus80 username/password@connect_string
```

connect\_string は ODBC:data\_source\_name です。たとえば、デフォルト・データベースにユーザー SYSTEM として接続するには、DOS プロンプトで次のように入力します。

```
plus80 system/pw@odbc:polite
```

SQL\*Plus コマンドラインで、次のように EMP 表を作成して値を挿入します。

```
CREATE TABLE EMP(E# int, name char(10), salary real,
  Constraint E#_PK primary key (E#));

INSERT INTO EMP VALUES (123,'Smith',60000);
INSERT INTO EMP VALUES (234,'Jones',50000);
```

次のクラスを **SalaryTrigger.java** に入れます。

```
class SalaryTrigger {
    private int eno;
    public SalaryTrigger(int enum) {
        eno = enum;
    }
    public void check_sal_raise(float old_sal,
        float new_sal)
    {
        if (((new_sal - old_sal)/old_sal) > .10)
        {
            // raise too high do something here
            System.out.println("Raise too high for employee " + eno);
        }
    }
}
```

SalaryTrigger クラス・コンストラクタは整数を引数にとり、この整数を属性 eno（従業員番号）に割り当てます。表 EMP の各行（つまり各従業員ごと）に SalaryTrigger が作成されます。

check\_sal\_raise メソッドは非静的メソッドです。実行するには、そのクラスのオブジェクトがこのメソッドをコールする必要があります。EMP の行の salary 列が更新されるたびに、その行に対応する SalaryTrigger コンストラクタへの引数として従業員番号 (E#) を使用して、このコンストラクタのインスタンスが（存在しない場合は）作成されます。次に、トリガーが check\_sal\_raise メソッドをコールします。

Java クラスを作成した後は、次のように、クラスを表にアタッチします。

```
ALTER TABLE EMP ATTACH JAVA SOURCE "SalaryTrigger" IN '.'
WITH CONSTRUCTOR ARGS (E#);
```

この文は、Oracle Lite に対して、現在のディレクトリにある Java ソース **SalaryTrigger.java** をコンパイルし、結果のクラスを EMP 表にアタッチするように指示します。また、この文は、Oracle Lite がクラスのインスタンスを生成するときに、E# 列にある値を引数として取るコンストラクタを使用するように指定します。

クラスを表にアタッチした後、次のように、トリガーを作成します。

```
CREATE TRIGGER CHECK_RAISE BEFORE UPDATE OF SALARY ON EMP FOR EACH ROW
"check_sal_raise"(old.salary, new.salary);
/
```

この文は、check\_raise というトリガーを作成します。このトリガーは、EMP 表のいずれかの行の salary 列が更新される前に、check\_sal\_raise メソッドをコールします。Oracle Lite は、salary 列の古い値と新しい値をメソッドの引数として渡します。

この例では、行レベル・トリガーが行レベル・プロシージャ（非静的メソッド）を起動します。行レベル・トリガーは、表レベル・プロシージャ（静的メソッド）を起動することもできます。ただし、文レベル・トリガーは文全体に対して 1 回起動され、文が複数行に影響する可能性があるため、文レベル・トリガーは表レベル・プロシージャを 1 回しか起動できません。

次のコマンドは、salary 列を更新しトリガーを起動します。

```
UPDATE EMP SET SALARY = SALARY + 6100 WHERE E# = 123;
```

これにより、次の出力が生成されます。

```
Raise too high for employee 123
```

## トリガーの引数

「[モデル 2: アタッチ済みストアド・プロシージャ開発モデルの使用](#)」で説明されているように、アタッチ済みストアド・プロシージャを使用する場合は、行レベル・トリガーは Java から SQL への型変換をサポートしません。このため、トリガー引数の Java データ型は、トリガー列の対応する SQL データ型（「[Java データ型](#)」を参照）に一致している必要があります。ただし、Load and Publish モデルを使用する場合は、Oracle Lite がデータ型のキャストをサポートします。

次の表に、それぞれの列のタイプで有効なトリガー引数を示します。

トリガーの引数	新規列のアクセス	古い列のアクセス
insert	はい	いいえ
delete	いいえ	はい
update	はい	はい

---

---

**注意：** `java.sql.Connection` オブジェクトを引数として持つトリガーは、リレーショナル・モデルを使用するアプリケーションでのみ使用できます。

---

---

## トリガーの引数の例

次の例は、IN/OUT パラメータを使用するトリガーの作成方法を示します。

1. まず、Java クラス `EMPTrigg` を作成します。

```
import java.sql.*;

public class EMPTrigg {
    public static final String goodGuy = "Oleg";

    public static void NameUpdate(String oldName, String[] newName)
    {
        if (oldName.equals(goodGuy))
            newName[0] = oldName;
    }

    public static void SalaryUpdate(String name, int oldSalary,
        int newSalary[])
```

```

    {
        if (name.equals(goodGuy))
            newSalary[0] = Math.max(oldSalary, newSalary[0])*10;
    }

    public static void AfterDelete(Connection conn,
        String name, int salary) {
        if (name.equals(goodGuy))
            try {
                Statement stmt = conn.createStatement();
                stmt.executeUpdate(
                    "insert into employee values('" + name + "', " +
                    salary + ")");
                stmt.close();
            } catch(SQLException e) {}
    }
}

```

2. 新規の表 EMPLOYEE を作成し、これに値を挿入します。

```

CREATE TABLE EMPLOYEE(NAME VARCHAR(32), SALARY INT);
INSERT INTO EMPLOYEE VALUES('Alice', 100);
INSERT INTO EMPLOYEE VALUES('Bob', 100);
INSERT INTO EMPLOYEE VALUES('Oleg', 100);

```

3. 次に、クラスを Oracle Lite にロードします。

```

CREATE JAVA CLASS USING BFILE ('c:\myprojects', 'EMPTrigg.class');

```

4. CREATE PROCEDURE 文を使用して、コールする EMPTrigg メソッドをパブリッシュします。

```

CREATE PROCEDURE NAME_UPDATE(
    OLD_NAME IN VARCHAR2, NEW_NAME IN OUT VARCHAR2)
AS LANGUAGE JAVA NAME
'EMPTrigger.NameUpdate(java.lang.String, java.lang.String[])';
/

CREATE PROCEDURE SALARY_UPDATE(
    ENAME VARCHAR2, OLD_SALARY INT, NEW_SALARY IN OUT INT)
AS LANGUAGE JAVA NAME
'EMPTrigger.SalaryUpdate(java.lang.String, int, int[])';
/

CREATE PROCEDURE AFTER_DELETE(
    ENAME VARCHAR2, SALARY INT)
AS LANGUAGE JAVA NAME
'EMPTrigger.AfterDelete(java.sql.Connection,

```

```
        java.lang.String, int)';  
    /
```

5. ここで、各プロシージャにトリガーを作成します。

```
CREATE TRIGGER NU BEFORE UPDATE OF NAME ON EMPLOYEE FOR EACH ROW  
    NAME_UPDATE(old.name, new.name);  
  
CREATE TRIGGER SU BEFORE UPDATE OF SALARY ON EMPLOYEE FOR EACH ROW  
    SALARY_UPDATE(name, old.salary, new.salary);  
  
CREATE TRIGGER AD AFTER DELETE ON EMPLOYEE FOR EACH ROW  
    AFTER_DELETE(name, salary);
```

6. 次のコマンドを入力し、トリガーを起動して結果を表示します。

```
SELECT * FROM EMPLOYEE;  
UPDATE EMPLOYEE SET SALARY=0 WHERE NAME = 'Oleg';  
SELECT * FROM EMPLOYEE;  
  
DELETE FROM EMPLOYEE WHERE NAME = 'Oleg';  
SELECT * FROM EMPLOYEE;  
  
UPDATE EMPLOYEE SET NAME='TEMP' WHERE NAME = 'Oleg';  
DELETE FROM EMPLOYEE WHERE NAME = 'TEMP';  
  
SELECT * FROM EMPLOYEE;
```

---

---

# Java Database Connectivity (JDBC)

この章では、Oracle Lite の Java Database Connectivity (JDBC) に対するサポートを説明します。説明する内容は、次のとおりです。

- [Oracle Lite の新機能](#)
- [JDBC](#)
- [JDBC 環境の設定](#)
- [JDBC からの Java スタアド・プロシージャの実行](#)
- [Oracle Lite での拡張](#)
- [JDBC と Java アクセス・クラス間の相互運用性](#)
- [制限事項](#)

---

---

**注意：** Oracle Lite JDBC ドライバの使用例は、[付録 B 「JDBC アプリケーションのチュートリアル」](#) を参照してください。

---

---

## Oracle Lite の新機能

この項では、Oracle Lite の今回のリリースで強化された JDBC サポートについて説明します。

JDBC ドライバでは、バイナリ・ラージ・オブジェクト (Binary Large Object: BLOB) およびキャラクタ・ラージ・オブジェクト (Character Large Object: CLOB) データ型を含むラージ・オブジェクト (Large Object: LOB) をサポートするようになりました。このサポートは、JDBC 2.0 仕様の機能に準拠し、Oracle8i の JDBC 実装と互換性があります。

Oracle Lite には、LOB にアクセスするための新機能も含まれています。詳細は、「[Oracle Lite での拡張](#)」を参照してください。

## JDBC

JDBC とは、Java プログラムからリレーショナル・データベースにアクセスするためのアプリケーション・プログラミング・インタフェースです。Oracle Lite では、Java アプリケーションが Oracle Lite のオブジェクト・リレーショナル・データベース・エンジンと直接通信できるネイティブな JDBC ドライバを提供しています。Oracle Lite の JDBC 実装は、JDBC 1.22 に準拠しています。さらに、Oracle Lite では JDBC 2.0 で指定されている特定の拡張機能を提供しています。Oracle Lite の拡張は、Oracle8i の JDBC 実装と互換性があります。JDBC の詳細は、サン・マイクロシステムズ社の Web サイトを参照してください。

## JDBC 環境の設定

Oracle Lite データベースにアクセスするには、Java プログラムで最初に Oracle Lite JDBC ドライバをロードしてから、データベースへの接続を確立する必要があります。

## Oracle Lite JDBC ドライバのロード

Oracle Lite JDBC ドライバをロードするには、Oracle Lite JDBC ドライバ・クラスの完全修飾名を `Class.forName` メソッドに渡します。

```
Class.forName("oracle.lite.poljdbc.POLJDBCdriver");
```

`Class.forName` メソッドは、指定された Oracle Lite JDBC ドライバ・クラスを検索し、ロードし、リンクします。POLJDBCdriver クラスのインスタンスを明示的に作成する必要はありません。POLJDBCdriver クラスのインスタンスは、`Class.forName` をコールしたときに JDBC ドライバ・マネージャにより自動的に作成され登録されます。



---

---

**注意：** Oracle Lite JDBC ドライバと JDK 1.2 JDBC ドライバでは特定のデータ型（CLOB や BLOB など）に対して同じ名前を使用するため、`java.sql.*` と `oracle.sql.*` の両方をインポートするプログラムからこのようなデータ型をコールすると、コンパイラ・エラーが返される可能性があります。このようなエラーの回避方法の詳細は、「[制限事項](#)」を参照してください。

---

---

## Oracle Lite データベースへの接続

Oracle Lite データベースの JDBC 接続文字列は次のとおりです。

```
jdbc:polite:data_sourcename
```

`data_sourcename` は、`odbc.ini` に定義されているデータ・ソースの名前です。JDBC ドライバは、`odbc.ini` 内の `data_sourcename` により識別されるデータベースの物理的な位置を参照します。

次の文では JDBC 接続文字列を使用して、データ・ソース名が POLITE の Oracle Lite データベースに接続します。

```
Connection conn = DriverManager.getConnection("jdbc:polite:POLITE",  
    "system", "passwd");
```

ユーザー `system` として接続する場合は、パスワードには任意の文字を組み合わせで指定できます。ただし、パスワードは空白にはできません。データベースが暗号化されている場合は、正しいパスワードを指定する必要があります。

Oracle Lite データベース内のデータにアクセスし操作するには、`getConnection` で返される `Connection` オブジェクトを使用します。

JDBC ドライバとの接続方法の詳細は、[付録 B 「JDBC アプリケーションのチュートリアル」](#) を参照してください。

## JDBC からの Java ストアド・プロシージャの実行

Java ストアド・プロシージャを作成した後は、次のようにして、JDBC アプリケーションからプロシージャを実行できます。

- ストアド・プロシージャを実行する SQL の SELECT 文字列を `Statement.executeQuery` メソッドに渡します。
- JDBC `CallableStatement` を使用します。

---

---

**注意：** ストアド・プロシージャの作成方法の詳細は、第 2 章「[Java ストアド・プロシージャとトリガー](#)」を参照してください。

---

---

`executeQuery` メソッドは、表レベルおよび行レベルのストアド・プロシージャを実行します。`CallableStatement` は、現時点では表レベルのストアド・プロシージャの実行しかサポートしていません。

### ExecuteQuery メソッドの使用方法

`executeQuery` メソッドを使用してストアド・プロシージャをコールするには、まず、`Statement` オブジェクトを作成します。このオブジェクトには、カレントの `Connect` オブジェクトの `createStatement` メソッドにより返された値を割り当てます。次に `Statement.executeQuery` メソッドをコールします。このメソッドには、Java ストアド・プロシージャを起動する SQL の SELECT 文字列を渡します。

たとえば、表 `INVENTORY` 上の行レベルのプロシージャ `SHIP` を、変数 `q` に格納されている引数値を使用して実行するとします。変数 `p` には、ストアド・プロシージャを実行する製品 (行) の製品 ID が含まれています。

```
int res = 0;
Statement s = conn.createStatement();
ResultSet r = s.executeQuery("SELECT SHIP(" + q + ") " +
    "FROM INVENTORY WHERE PID = " + p);
if(r.next()) res = r.getInt(1);
r.close();
s.close();
return res;
```

パラメータを様々に変えてプロシージャを繰り返し実行する場合は、`Statement` のかわりに `PreparedStatement` を使用します。`PreparedStatement` 内の SQL 文はプリコンパイルされているため、`PreparedStatement` は効率良く実行できます。さらに、`PreparedStatement` は、文の中で疑問符 (?) を使用して IN パラメータを設定できます。ただし、`PreparedStatement` が `LONG` や `LONG RAW` などの `LONG` 型パラメータを使

用する場合は、`setAsciiStream`、`setUnicodeStream`または`setBinaryStream`メソッドを使用して、パラメータをバインドする必要があります。

前述の例で、プロシージャ SHIP でデータベースを更新する場合、前述の問合せを発行するトランザクションの分離レベルが `READ COMMITTED` であれば、`SELECT` 文に `FOR UPDATE` 句を追加する必要があります。次のように指定します。

```
"SELECT SHIP(" + q + ")" +
  FROM INVENTORY WHERE PID = " +
  p + "FOR UPDATE");
```

## Callable Statement の使用方法

Callable Statement を使用してストアド・プロシージャを実行するには、次のように、CallableStatement オブジェクトを作成してそのパラメータを登録します。

```
CallableStatement cstmt = conn.prepareCall(
    "{?=call tablename.methodname() }");
cstmt.registerOutParameter(1, ...);
cstmt.executeUpdate();
cstmt.get..(1);
cstmt.close();
```

JDBC の Callable Statement の使用には、次の制限が適用されます。

- JDBC の Callable Statement で実行できるのは、表レベルのストアド・プロシージャのみです。
- IN パラメータと OUT パラメータの両方がサポートされるが、OUT パラメータとして使用する Java データ型に制限が適用されます。詳細は、[第 2 章「Java ストアド・プロシージャとトリガー」](#)を参照してください。
- プロシージャ名は Java メソッド名に対応し、大 / 小文字が区別されます。
- PreparedStatement の場合と同じように、Callable Statement が LONG 型 (LONG、LONG VARBINARY、LONG VARCHAR、LONG VARCHAR2、LONG RAW など) を使用する場合は、`setAsciiStream`、`setUnicodeStream`または`setBinaryStream`メソッドを使用してパラメータをバインドする必要があります。

---

---

**注意：** ResultSet オブジェクトや Statement オブジェクトなどの JDBC オブジェクトが不要になった場合、オブジェクトをクローズしてシステム・リソースを解放する必要があります。

---

---

## Oracle Lite での拡張

Oracle Lite JDBC ドライバは JDBC 1.22 をサポートし、JDBC 2.0 で定義されている特定の機能拡張も提供します。この拡張には、BLOB や CLOB データ型とデータ・アクセス関数に対するサポートが含まれています。Oracle Lite での JDBC 拡張は、Oracle8i の JDBC 実装と互換性があります。ただし、Oracle Lite では Oracle8i の JDBC データ型拡張、配列、構造体、REF はサポートしていません。

この項では、Oracle Lite のデータ型とデータ・アクセスに関する拡張を説明します。関数の構文やコール・パラメータの詳細は、次のサイトにあるサン・マイクロシステムズ社の『Java 2 仕様』を参照してください。

<http://www.javasoft.com/products/jdk/1.2/docs/api/index.html>

## データ型の拡張

BLOB と CLOB は、データベース表に直接格納するには大きすぎるデータを格納するために使用されます。データベース表には、データではなく、実際のデータの場所を指すロケータが格納されます。BLOB には、非構造化バイナリ・データが大量に含まれ、CLOB には固定幅文字データ（1 文字につき一定のバイト数が必要な文字）が大量に含まれます。

BLOB または CLOB のロケータは、標準の SELECT 文を使用してデータベースから選択できます。SELECT 文を使用して BLOB または CLOB ロケータを選択すると、LOB のロケータのみが取得されます。データ自体は取得されません。ただし、一度ロケータを取得すると、アクセス関数を使用して LOB のデータの読取りと書込みが可能になります。

Oracle Lite の BLOB クラスには、次のメソッドが含まれています。

機能	説明
length	BLOB の長さをバイト単位で返します。
getBinaryOutputStream	BLOB データを返します。
getBinaryStream	BLOB インスタンスをバイト・ストリームとして返します。
getBytes	指定された位置から、BLOB データをバッファに読み取ります。
getConnection	カレントの接続を返します。
isConvertibleTo	BLOB を特定のクラスに変換できるかどうかを判断します。
putBytes	BLOB データ内の指定された位置にバイトを書き込みます。
makeJdbcArray	JDBC 配列で表された BLOB を返します。
toJdbc	BLOB を JDBC クラスに変換します。

Oracle Lite の CLOB クラスには、次のメソッドが含まれています。

機能	説明
length	CLOB の長さをバイト単位で返します。
getSubString	CLOB データ内の指定された位置からサブ文字列を取り出します。
getCharacterStream	CLOB データを Unicode ストリームとして返します。
getAsciiStream	CLOB インスタンスを ASCII ストリームとして返します。
getChars	CLOB データ内の指定された位置からの文字を文字配列に取り出します。
getCharacterOutputStream	Unicode ストリームから CLOB データを書き込みます。
getAsciiOutputStream	ASCII ストリームから CLOB データを書き込みます。
getConnection	カレントの接続を返します。
putChars	文字配列からの文字を、CLOB データ内の指定された位置に書き込みます。
putString	CLOB データ内の指定された位置にデータを書き込みます。
toJdbc	CLOB を JDBC クラスに変換します。
isConvertibleTo	CLOB を特定のクラスに変換できるかどうかを判断します。
makeJdbcArray	JDBC 配列で表された CLOB を返します。

## データ・アクセスの拡張

Oracle Lite では、CLOB および BLOB データ型の値を設定および返すアクセス関数が提供されています。さらに、ラージ・オブジェクトに対するストリーム形式のアクセスを可能にする関数が、ストリーム・クラスにより提供されています。

OraclePreparedStatement クラス、OracleCallableStatement クラスおよび OracleResultSet クラスには、LOB に対するアクセス関数が含まれています。

OraclePreparedStatement クラスには、次のデータ・アクセス関数が含まれています。

機能	説明
setBLOB	BLOB データを設定します。
setCLOB	CLOB データを設定します。

OracleCallableStatement クラスには、次のデータ・アクセス関数が含まれています。

機能	説明
getBLOB	BLOB データを指すロケータを返します。
getCLOB	CLOB データを指すロケータを返します。

OracleResultSet クラスには、次のデータ・アクセス関数が含まれています。

機能	説明
getBLOB	BLOB データを指すロケータを返します。
getCLOB	CLOB データを指すロケータを返します。

POLLobInputStream、POLLobOutputStream、POLClobReader および POLClobWriter は、ストリーム形式のアクセス・クラスです。

POLLobInputStream クラスには、次のデータ・アクセス関数が含まれています。

機能	説明
read	LOB から配列に読み取ります。

POLLobOutputStream クラスには、次のデータ・アクセス関数が含まれています。

機能	説明
write	出力ストリームから LOB に書き込みます。

POLClobReader クラスは、クラス `java.io.reader` を拡張します。次のデータ・アクセス関数が含まれています。

機能	説明
read	CLOB 内の文字を配列の一部に読み取ります。
ready	ストリームが読み取れる状態になっているかどうかを示します。

機能	説明
close	ストリームを閉じます。
markSupported	ストリームで mark 操作がサポートされているかどうかを示します。
mark	ストリーム内のカレント位置にマークを設定します。この後で reset 関数をコールすると、ストリームの位置がマークされた位置に変更されます。
reset	ストリーム内のカレント位置を、マークされた位置にリセットします。ストリームがマークされていない場合、この関数はそのストリームに適した（たとえば、開始位置に戻すなどの）方法でストリームをリセットしようとします。
skip	ストリーム内の文字をスキップします。

POLClobWriter クラスは、クラス `java.io.writer` を拡張します。次のデータ・アクセス関数が含まれています。

機能	説明
write	文字配列を出力ストリームに書き込みます。
flush	バッファ内のすべての文字を、目的の宛先に書き込みます。
close	ストリームをフラッシュして閉じます。

## BLOB の読取りを行うサンプル・プログラム

次のサンプルでは、`getBinaryStream` メソッドを使用して BLOB データをバイト・ストリームに読み取ります。次に、バイト・ストリームをバイト配列に読み取り、読み取られたバイト数を返します。

```
// Read BLOB data from BLOB locator.
InputStream byte_stream = my_blob.getBinaryStream();
byte [] byte_array = new byte [10];
int bytes_read = byte_stream.read(byte_array);
...
```

## CLOB への書き込みを行うサンプル・プログラム

次のサンプルでは、文字配列にデータを読み取り、次に `getCharacterOutputStream` メソッドを使用して文字配列を CLOB に書き込みます。

```
java.io.Writer writer;
// read data into a character array
char[] data = {'0','1','2','3','4','5','6','7','8','9'};

// write the array of character data to a CLOB
writer = ((CLOB)my_clob).getCharacterOutputStream();
writer.write(data);
writer.flush();
writer.close();
...
```

## JDBC と Java アクセス・クラス間の相互運用性

JDBC インタフェースはデータ記憶域のリレーショナル・モデルを使用するアプリケーションで使用され、Java アクセス・クラス (JAC) はデータのオブジェクト指向モデルを使用するアプリケーションで使用できます。どちらのインタフェースも同一のデータベースにデータを格納できます。一方のインタフェースで作成したデータは他方のインタフェースからアクセスできます。ただし、これには後述のような制限があります。

JDBC アプリケーションによって作成された表は、JAC インタフェースからはすべてクラスとして参照でき、JAC アプリケーションはその表の行をオブジェクトとして選択できます。表のプロシージャもすべて、対応するクラス内のメソッドとして JAC インタフェースから参照できます。

JAC を使用して作成したクラスは、次の条件がすべて満たされている場合、JDBC インタフェースからアクセスできます。

- JDBC アプリケーションで、JAC を介して作成されたクラスや属性名を二重引用符 ("") で囲んで参照している場合。これらの名前が大 / 小文字によって区別されるためです。
- JDBC で、JAC を介して作成されたクラスのドットをスラッシュに置き換えて参照している場合。たとえば、クラス `oracle.personnel.Person` からすべての `Person` オブジェクトの名前属性を JDBC で検索するには、次のように指定します。

```
SELECT "Name"
FROM "Oracle/personnel/Person" ;
```

属性名とクラス名は二重引用符で囲まれ、クラス名のドットはすべてスラッシュに置き換えられています。

- クラスが最上位のクラスの場合。つまり、他のユーザー定義クラスのサブクラスではありません。



- クラスの属性のすべてが、JDBC のサポートするデータ型の場合。たとえば、JDBC では整数や文字列などのリテラル・データ型がサポートされますが、他のクラスを参照する属性タイプはサポートされません。サポートされている属性にアクセスするには、JAC クラスの SQL ビューを作成します。このビューは、JDBC でサポートされているデータ型の属性をクラスから選択します（クラス名は表名と同じです）。
- クラスのインスタンスが、グループを指定しないで作成されている場合。つまり、クラスのインスタンスがクラスのデフォルト・グループにおいて作成されているということです。Oracle Lite では、最上位クラスのそれぞれに対して、クラス名と同じ名前のデフォルト・グループを作成します。JAC を使用して作成したメソッドやトリガーは、すべて JDBC インタフェースからアクセスできます。

JDBC でサポートしていない属性の型は、次のとおりです。

- すべての配列型（文字配列型とバイト配列型を除く）
- すべての OID 型

## 制限事項

書込み中にデータの切捨てが発生すると、SQL データ切捨て例外が発生します。読取り中にデータの切捨てが発生した場合は、SQL データ切捨て警告が発生します。しかし、データ切捨てオブジェクトには意味のある情報は含まれていません。

Oracle Lite の JDBC クラスと JDBC 2.0 クラスでは、特定のデータ型に同じ名前を使用しています（たとえば、`oracle.sql.Blob` と `java.sql.Blob`）。プログラムで `oracle.sql.*` と `java.sql.*` を両方ともインポートする場合、完全修飾名を指定せずに重複しているクラスにアクセスしようとする、コンパイル・エラーが発生する可能性があります。この問題を回避するには、次の方法のいずれかを使用します。

1. BLOB、CLOB およびデータ・クラスに完全修飾名を使用します。
2. クラスを明示的にインポートします（たとえば、`import oracle.sql.Blob`）。
3. JDK 1.1 コンパイラで CLASSPATH に `oljdk11.jar` を指定しないでコンパイルします。

クラスには常に完全修飾のクラス名が含まれているため、データ型の名前が重複しても実行時に競合は発生しません。



---

---

## Java 埋込み SQL (SQLJ)

SQLJ は、SQL 文を Java プログラムに埋め込むための規格です。SQLJ は、静的 SQL 文を必要とする Oracle Lite Java アプリケーションの中で使用できます。

この項では、Oracle Lite の開発における Oracle SQLJ ツールの使用方法の概要を説明します。説明する内容は、次のとおりです。

- [SQLJ の概要](#)
- [SQLJ の操作](#)
- [SQLJ の使用方法](#)

## SQLJ の概要

SQLJ を使用すると、Java プログラムに静的 SQL 文を埋め込むことができます。静的 SQL 文は事前に定義された文です。つまり、ユーザーがプログラムを実行する際に変化しない文のことを指します。ただし、SQL 文に渡されるデータ値はホスト変数を使用して変更できます。現実には、アプリケーションには静的 SQL 文と動的 SQL 文の両方が必要になる場合がほとんどのため、SQLJ 文は JDBC 文と同じソース内に混在させて使用します。SQLJ は静的 SQL 文に、JDBC は静的 SQL 文と動的 SQL 文に使用します。

Oracle Lite の SQLJ サポートは Oracle8i と互換性があります。Oracle8i 用に作成された SQLJ プログラムは、あまり変更しなくても Oracle Lite で機能します。

## SQLJ の操作

SQLJ 文は、`#sql` タグにより識別されます。SQLJ トランスレータは、SQLJ 文を SQLJ ランタイム・ライブラリへのコールに置き換え、クラスを生成します。トランスレータが SQLJ 文を変換するとき、構文がチェックされ、Java と SQL 間のデータ交換の互換性を確認するためにタイプ・チェックが実行されます。また、トランスレータはスキーマもチェックして、SQL 構文とデータベース・スキーマとの間で矛盾がないようにします。

次のサンプル・コードは、JDBC と SQLJ で同じ内容の SELECT 文を示しています。SQLJ 文では、入力および出力引数に、ホスト・プログラムで宣言された変数を使用します。JDBC には、各引数をバインドする文と、それぞれの結果を取り出す文が別々に必要です。

```
// SQLJ
float w; java.sql.Date x; int y; String z;
...
#sql { SELECT C1, C2 INTO :w, :x FROM TAB WHERE C3 = :y AND C4 = :z };

// JDBC
float w; java.sql.Date x; int y; String z;
...
PreparedStatement s = connection.prepareStatement(
    "SELECT C1, C2 FROM TAB WHERE C3 = ? AND C4 = ?");
s.setInt(1, y);
s.setString(2, z);
ResultSet r = s.executeQuery();
r.next();
w = r.getFloat(1);
x = r.getDate(2);
r.close();
s.close();
```

コンパイル済みの SQLJ プログラムは、JVM、SQLJ ランタイム・ライブラリおよび JDBC ドライバが利用できれば、どのような環境でも実行できます。

## SQLJ の使用方法

SQLJ を使用するには、SQLJ トランスレータ、ランタイム・ライブラリおよび付属のマニュアルを、次の Oracle Web サイトからダウンロードします。

<http://www.oracle.com/java/sqlj/index.html>

配布パッケージには、SQLJ の詳しいドキュメントが含まれています。次の項では、Oracle Lite とともに SQLJ を使用方法について説明します。

## SQLJ の構成

SQLJ をインストールし構成するには、配布パッケージに含まれている手順に従います。Oracle Lite とともに使用するには、SQLJ パラメータを次のように構成します。

1. **connect.properties** ファイルで、SQLJ URL を次のように設定します。

```
jdbc:polite:dsn_name
```

2. **sqlj.properties** ファイルで、SQLJ URL を次のように設定します。

```
jdbc:polite:dsn_name
```

3. SQLJ ドライバを次のように設定します。

```
oracle:lite:pol.jdbc.POLJDBCdriver
```

SQL スクリプトを実行して表をインストールした後、`conn.commit()` を使用して、変更をデータベースに明示的にコミットします (`conn` は `Connection` オブジェクト)。

## Oracle Lite への接続

SQLJ アプリケーションでは、Oracle Lite データベースに接続する前に、まず Oracle Lite JDBC ドライバをロードする必要があります。JDBC ドライバをロードするには、アプリケーションで JDBC ドライバ・クラスの完全修飾名を引数として `Class.forName` メソッドに渡す必要があります。

```
class.forName("oracle:lite:poljdbc.POLJDBCdriver")
```



---

# ストアド・プロシージャのチュートリアル

この付録では、Java ストアド・プロシージャとトリガーの作成方法を示します。説明する内容は、次のとおりです。

- ストアド・プロシージャとトリガーの作成
- トリガーの作成
- コミットまたはロールバック

## ストアド・プロシージャとトリガーの作成

このチュートリアルでは、Java クラス `EMAIL` を作成し、Oracle Lite にロードし、このクラスのメソッドを SQL にパブリッシュし、このメソッドのトリガーを作成します。`EMAIL` クラスはソース `EMAIL.java` にあります。このファイルは、Java のサンプル・ディレクトリ `Oracle_Home¥Mobile¥Sdk¥Examples¥Java` にあります。

`EMAIL` には `assignEMailAddress` というメソッドがあります。このメソッドは、従業員のファースト・ネームの最初の 1 文字とラスト・ネームの最大 7 文字に基づいて、その従業員の電子メール・アドレスを生成します。アドレスがすでに割り当てられている場合、このメソッドは、ファースト・ネームとラスト・ネームの文字を組み合わせで一意的な電子メール・アドレスを見つけようとします。

クラスを作成した後、SQL\*Plus を使用してそのクラスを Oracle Lite にロードします。この例では、SQL の `CREATE JAVA` 文を使用します。かわりに、`loadjava` ユーティリティを使用してクラスを Oracle Lite にロードすることもできます。クラスをロードした後、`assignEMailAddress` メソッドを SQL にパブリッシュします。

最後に、`T_EMP` (従業員情報が含まれている表) に行が挿入されるたびに `assignEMailAddress` メソッドを起動するトリガーを作成します。

`assignEMailAddress` は、引数として、JDBC Connection オブジェクト、従業員の ID 番号、ファースト・ネーム、ミドル・イニシャル、ラスト・ネームを取ります。JDBC Connection オブジェクトは、Oracle Lite により提供されます。`assignEMailAddress` メソッドを実行するときに Connection オブジェクトの値を指定する必要はありません。このメソッドでは、JDBC Connection オブジェクトを使用して、生成された電子メール・アドレスが一意的であることを確認します。

## SQL\*Plus の起動

SQL\*Plus を起動し、デフォルトの Oracle Lite データベースに接続します。このチュートリアルの Java アプリケーションでは標準出力に出力するため、SQL\*Plus は DOS 版を使用します。DOS プロンプトから次を入力します。

```
plus80 system/mgr@odbc:polite
```

SQL プロンプトが表示されます。

## 表の作成

表を作成するには、次を入力します。

```
CREATE TABLE T_EMP(ENO INT PRIMARY KEY,  
    FNAME VARCHAR(20),  
    MI CHAR,  
    LNAME VARCHAR(20),  
    EMAIL VARCHAR(8));
```



## Java クラスの作成

c:\tmp の **EMAIL.java** ファイルに Java クラス **EMAIL** を作成しコンパイルします。**EMAIL.java** は `assignEMailAddress` メソッドを実装します。このファイルの内容は、次のとおりです。このファイルは、`Oracle_Home¥Mobile¥Sdk¥EXAMPLES¥JAVA` ディレクトリからコピーできます。

```
import java.sql.*;

public class EMAIL {
    public static void assignEMailAddress(Connection conn,
        int eno, String fname,String lname)
        throws Exception
    {
        Statement stmt = null;
        ResultSet retset = null;
        String emailAddr;
        int i,j,fnLen, lnLen, rowCount;

        /* create a statement */
        try {
            stmt = conn.createStatement();
        }
        catch (SQLException e)
        {
            System.out.println("conn.createStatement failed: " +
                e.getMessage() + "\n");
            System.exit(0);
        }
        /* check fname and lname */
        fnLen = fname.length();
        if(fnLen > 8) fnLen = 8;
        if (fnLen == 0)
            throw new Exception("First name is required");
        lnLen = lname.length();
        if(lnLen > 8) lnLen = 8;
        if (lnLen == 0)
            throw new Exception("Last name is required");
        for (i=1; i <= fnLen; i++)
        {
            /* generate an e-mail address */
            j = (8-i) > lnLen? lnLen:8-i;
            emailAddr =
                new String(fname.substring(0,i).toLowerCase()+
                    lname.substring(0,j).toLowerCase());
            /* check if this e-mail address is unique */
            try {
```

```

        reset = stmt.executeQuery(
            "SELECT * FROM T_EMP WHERE email = '" +
            emailAddr+"'");
        if(!reset.next()) {
            /* e-mail address is unique;
            * so update the email column */
            reset.close();
            rowCount = stmt.executeUpdate(
                "UPDATE T_EMP SET EMAIL = '"
                + emailAddr + "' WHERE ENO = "
                + eno);
            if(rowCount == 0)
                throw new Exception("Employee "+fname+ " " +
                    lname + " does not exist");
            else return;
        }
    }
    catch (SQLException e) {
        while(e != null) {
            System.out.println(e.getMessage());
            e = e.getNextException();
        }
    }
}
/* Can't find a unique name */
emailAddr = new String(fname.substring(0,1).toLowerCase() +
    lname.substring(0,1).toLowerCase() + eno);
rowCount = stmt.executeUpdate(
    "UPDATE T_EMP SET EMAIL = '"
    + emailAddr + "' WHERE ENO = "
    + eno);
if(rowCount == 0)
    throw new Exception("Employee "+fname+ " " +
        lname + " does not exist");
else return;
}
}

```

## Java クラスのロード

EMAIL クラスを Oracle Lite にロードするには、次を入力します。

```
CREATE JAVA CLASS USING BFILE
  ('c:¥tmp', 'EMAIL.class');
```

クラスをロードした後、クラスに変更を加える場合は、次を行う必要があります。

1. dropjava または DROP JAVA CLASS を使用して、データベースからクラスを削除します。
2. 作業をコミットします。
3. SQL\*Plus を終了します。
4. SQL\*Plus を再起動します。

これで Java Virtual Machine (JVM) からクラスがアンロードされます。

## ストアド・プロシージャのパブリッシュ

ストアド・プロシージャのコール仕様を作成すると、そのストアド・プロシージャは SQL からコールできるようになります。assignEmailAddress は値を返さないため、次のように CREATE PROCEDURE コマンドを使用します。

```
CREATE OR REPLACE PROCEDURE
  ASSIGN_EMAIL(E_NO INT, F_NAME VARCHAR2, L_NAME VARCHAR2)
  AS LANGUAGE JAVA NAME 'EMAIL.assignEmailAddress(java.sql.Connection,
int, java.lang.String,
  java.lang.String)';
/
```

SQL\*Plus を使用する場合は、CREATE PROCEDURE 文に続いてスラッシュを入力する必要があります。

## データベースへの値の移入

T\_EMP に行を挿入します。

```
INSERT INTO T_EMP VALUES (100, 'John', 'E', 'Smith', null);
```

## プロシージャの実行

プロシージャを実行するには、次のように入力します。

```
SELECT ASSIGN_EMAIL(100, 'John', 'Smith')
  FROM dual
```

## 電子メール・アドレスの検証

ASSIGN\_EMAIL プロシージャの結果を表示するには、次のように入力します。

```
SELECT * FROM T_EMP;
```

このコマンドにより、次の出力が生成されます。

ENO	FNAME	M LNAME	EMAIL
-----	-----	-----	-----
100	John	E Smith	jsmith

## トリガーの作成

T\_EMP に行が挿入されるたびに ASSIGN\_EMAIL が実行されるようにするには、T\_EMP に AFTER INSERT トリガーを作成します。このトリガーは次のように作成します。

```
CREATE TRIGGER EMP_TRIGG AFTER INSERT ON T_EMP FOR EACH ROW
  ASSIGN_EMAIL(eno, fname, lname);
/
```

SQL\*Plus では、CREATE TRIGGER 文の後にピリオドとスラッシュを1つずつ別々の行に入力する必要があります。

T\_EMP に1行挿入されるたびに、EMP\_TRIGG というトリガーが起動されます。このプロシージャの実際の引数は、列 eno、fname および lname の値です。

connection 引数を指定する必要はありません。

## トリガーのテスト

T\_EMP に1行挿入して、トリガーをテストします。

```
INSERT INTO T_EMP VALUES (200, 'James', 'A', 'Smith', null);
```

## 電子メール・アドレスの検証

SELECT 文でトリガーの起動を確認します。

```
SELECT * FROM T_EMP;
```

ENO	FNAME	M LNAME	EMAIL
-----	-----	-----	-----
100	John	E Smith	jsmith
200	James	A Smith	jasmith

## コミットまたはロールバック

最後に、変更をコミットして作業を保存するか、ロールバックして変更を取り消します。



---

---

## JDBC アプリケーションのチュートリアル

このチュートリアルでは、Oracle Lite で JDBC アプリケーションを作成する方法を説明します。説明する内容は、次のとおりです。

- 環境の検査
- アプリケーションへの JDBC ドライバのロード
- Oracle Lite データベースへの接続
- 問合せの発行と結果の参照
- トランザクションのコミット
- 完全な例
- この例のコンパイルと実行

## 環境の検査

開始する前に、**olite40.jar** が CLASSPATH 内にあることを確認します。環境変数 CLASSPATH には、Oracle Lite のインストール中に **olite40.jar** が設定されます。設定されていない場合は、この変数を追加します。JDK をインストールしていない場合はインストールします。

## アプリケーションへの JDBC ドライバのロード

JDBC アプリケーションでは、最初に Oracle Lite JDBC ドライバをロードする必要があります。Oracle Lite JDBC ドライバをアプリケーションにロードするには、次のように `forName` 関数を使用します。

```
Class.forName("oracle.lite.poljdbc.POLJDBCdriver");
```

## Oracle Lite データベースへの接続

Oracle Lite データベースに接続するには、次の 2 つの方法があります。

- データベースが JDBC アプリケーションと同じコンピュータにある場合は、ネイティブ・ドライバ接続 URL 構文を使用してデータベースに接続します。
- データベースがサーバー上にあり、クライアント・コンピュータから接続する場合は、タイプ 2 ドライバ接続 URL 構文を使用して接続します。
- どちらのドライバも **olite40.jar** のコンポーネントです。これら 2 つのドライバは、JDBC1.0 規格に準拠しています。

### ネイティブ・ドライバ接続 URL 構文

```
jdbc:polite[:uid / pwd]:dsn [;key=value]*
```

次の引数は、DSN 句の一部とすることも、個別のキー値のペアとすることもできます。ドライバに追加情報を提供するキー値のペアは、出現する場合と出現しない場合があります。DSN で指定できる情報はすべて、キー値のペアとして指定できます。情報をキー値のペアとして指定すると、DSN で指定した情報は必ず上書きされます。

URL の解析とキー値のペアのオプションは、次のとおりです。

引数	説明
jdbc	プロトコルを JDBC として識別する。
polite	サブプロトコルを polite として識別する。



引数	説明
<code>uid/pwd</code>	Oracle Lite データベースのオプションのユーザー ID およびパスワード。この引数を指定すると、データ・ソース名 (DSN) およびキー値のペアでのユーザー ID およびパスワードの指定が上書きされる。
<code>dsn</code>	<b>odbc.ini</b> ファイル内のデータソース名 (DSN) のエントリを識別する。このエントリには、サーバーへの接続を完了するために必要な情報がすべて含まれている。
<code>DataDirectory=</code>	<b>.odb</b> ファイルが常駐しているディレクトリ。
<code>Database=</code>	作成時に指定したデータベース名。
<code>IsolationLevel=</code>	トランザクション分離レベル。READ COMMITTED、REPEATABLE READ、SERIALIZABLE または SINGLE USER。分離の詳細は、各プラットフォーム用の開発者ガイドを参照してください。
<code>Autocommit=</code>	コミット動作。ON または OFF。
<code>CursorType=</code>	カーソル動作。DYNAMIC、FORWARD ONLY、KEYSET DRIVE または STATIC。カーソル・タイプの詳細は、各プラットフォーム用の開発者ガイドを参照してください。
<code>UID=</code>	ユーザー名。
<code>PWD=</code>	パスワード。
<code>key=value</code>	

## 例

Oracle Lite JDBC ドライバをロードした後は、JDBC アプリケーションから Oracle Lite データベースに接続できます。たとえば、次のとおりです。

```
String ConnectMe=("jdbc:polite:SCOTT/tiger:polite;DataDirectory=c:\Oracle_
Home;Database=polite;IsolationLevel=SINGLE;USER;Autocommit=ON;CursorType=DYNAMIC")

try {
    Connection conn = DriverManager.getConnection(ConnectMe)
}
catch (SQLException e)
{
    ...
}
```

**タイプ2 ドライバ接続 URL 構文**

```
jdbc:polite[:uid / pwd]@[host]:[port]:dsn [;key=value]*
```

ネイティブ・ドライバ接続と同様、タイプ2 ドライバ接続では、次の引数を、DSN 句の一部として使用することも、ドライバに追加情報を提供するオプションのキー値のペアとして使用することもできます。情報をキー値のペアとして指定すると、DSN で指定した情報は必ず上書きされます。

引数	説明
jdbc	プロトコルを JDBC として識別する。
polite	サブプロトコルを polite として識別する。
UID	ユーザー名。
PWD	パスワード。
host	Oracle Lite データベースのホストとなり、ODBC アダプタ・サービス <b>olsv2040.exe</b> が実行されるマシンの名前。このホスト名はオプション。未指定の場合のデフォルト値は、JDBC アプリケーションが実行されるローカル・マシンの名前。
port	ODBC アダプタ・サービスによってリスニングが行われるポート番号。このポート番号はオプションで、未指定の場合のデフォルト値はポート「100」。
dsn	<b>odbc.ini</b> ファイル内のデータソース名 (DSN) のエントリを識別する。このエントリには、サーバーへの接続を完了するために必要な情報がすべて含まれている。
DataDirectory=	.odb ファイルが常駐しているディレクトリ。
Database=	作成時に指定したデータベース名。
IsolationLevel=	トランザクション分離レベル。READ COMMITTED、REPEATABLE READ、SERIALIZABLE または SINGLE USER。分離の詳細は、各プラットフォーム用の開発者ガイドを参照してください。
Autocommit=	コミット動作。ON または OFF。
CursorType=	カーソル動作。DYNAMIC、FORWARD ONLY、KEYSET DRIVE または STATIC。カーソル・タイプの詳細は、各プラットフォーム用の開発者ガイドを参照してください。
UID=	ユーザー名。
PWD=	パスワード。

## 例

次に、このタイプの接続の例を示します。

```
try {
    Connection conn = DriverManager.getConnection(
        "jdbc:polite
        :polite@URL_Name ;DataDirectory=c:¥Oracle_Home
        ;Database=polite
        ;IsolationLevel=SINGLE USER
        ;Autocommit=ON
        ;CursorType=DYNAMIC", "UID", "PWD")
}
catch (SQLException e)
{
    ...
}
```

getConnection メソッドの最初の引数は JDBC URL です。

2 番目の引数は、接続先のカタログ名またはユーザー名です。すべての Oracle Lite データベースには、system というカタログ名が付いています。

3 番目の引数はパスワードです。データベースが暗号化されている場合は、パスワードを指定する必要があります。

getConnection メソッドは try-catch ブロックで囲んで、接続試行中に SQL 例外が発生しないようにします。catch ブロックに例外処理文を挿入できます。

## 問合せの発行と結果の参照

問合せを発行して結果を参照する前に、表を作成し、その表に新規行を移入する必要があります。

### 表の作成

JDBC を使用してデータベース内に表を作成するには、まず、Statement オブジェクトを作成します。

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

次に、表を作成する SQL 文とともにこの文を実行します。

```
stmt.executeUpdate("create table person (name char(20), age integer,
    address char(100))");
```

### 表への新規行の移入

表に新規行を挿入するには、INSERT 文を使用して PreparedStatement オブジェクトを作成します。

```
PreparedStatement pstmt = conn.prepareStatement("insert into person  
values(?,?,?)");
```

疑問符 (?) のそれぞれが、値を入れるプレースホルダです。このプレースホルダには、文を実行する前に値をバインドしておく必要があります。

```
pstmt.setString(1, "Kevin Smith");  
pstmt.setInt(2, 30);  
pstmt.setString(3, "123 Never Land Dr, SF, CA 11111");  
pstmt.executeUpdate();
```

### 表の問合せと結果の参照

表を問い合わせるには、SQL 問合せを指定した文を実行します。

```
ResultSet rset = stmt.executeQuery("select * from person");
```

結果を参照するには、次のように指定します。

```
while (rset.next())  
{  
    System.out.println("Name : " + rset.getString(1)+"\n");  
    System.out.println("Age: " +rset.getInt(2)+"\n");  
    System.out.println("Address: "+rset.getString(3)+"\n\n");  
}
```

結果セットが不要な場合はクローズします。

```
rset.close();
```

次のように文をクローズします。

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

## トランザクションのコミット

トランザクションをコミットするには、commit メソッドを使用します。

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

データベース接続が不要な場合は、次のように指定してアプリケーションから切断できます。

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

## 完全な例

この例は、このチュートリアル全体を示したものです。

```
import java.sql.*;
public class jdbcexample {

    public static void main(String args[]) {
        // to load the driver
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            // connect to the database
            Connection conn = DriverManager.getConnection("jdbc:polite:polite",
                "system", "");
            // create a table
            Statement stmt = conn.createStatement();
            stmt.executeUpdate("create table person (name char(20), age integer, "+
                " address char(100))");
            // insert new rows into the table
            PreparedStatement pstmt = conn.prepareStatement(
                "insert into person values(?,?,?)");
            pstmt.setString(1, "Kevin Smith");
            pstmt.setInt(2, 30);
            pstmt.setString(3, "123 Never Land Dr, SF, CA 11111");
            pstmt.executeUpdate();
            // query the table
            ResultSet rset = stmt.executeQuery("select * from person");
            // browse the result
            while (rset.next())
            {
                System.out.println("Name : " + rset.getString(1)+"\n");
                System.out.println("Age: " +rset.getInt(2)+"\n");
                System.out.println("Address: "+rset.getString(3)+"\n\n");
            }
            rset.close();
            stmt.close();
        }
    }
}
```

```
        // commit the transaction
        conn.commit();
        // disconnect from database
        conn.close();
    }
    catch (SQLException e)
    // exception handling
    {
        System.out.println(e);
    }
    catch (ClassNotFoundException ee)
    {
        System.out.println(ee);
    }
}
}
```

## この例のコンパイルと実行

この例をコンパイルするには、次のように入力します。

```
javac jdbcexample.java
```

この例を実行するには、次のように入力します。

```
java jdbcexample
```

# C

---

---

## JDBC 2.0 の新機能

## インタフェース接続

### メソッド

#### Statement

`createStatement(int resultSetType, int resultSetConcurrency)`

指定された型と並行性の `ResultSet` オブジェクトを生成する `Statement` オブジェクトを作成します。

#### Map

`getTypeMap()`

この接続に対応付けられた型マップ・オブジェクトを取得します。

#### CallableStatement

`prepareCall(String sql, int resultSetType, int resultSetConcurrency)`

指定された型と並行性の `ResultSet` オブジェクトを生成する `CallableStatement` オブジェクトを作成します。

#### PreparedStatement

`prepareStatement(String sql, int resultSetType, int resultSetConcurrency)`

指定された型と並行性の `ResultSet` オブジェクトを生成する `PreparedStatement` オブジェクトを作成します。

#### void

`setTypeMap(Map map)`

指定された型を、この接続に対する型マップとしてインストールします。



## インタフェース文

### Connection

getConnection()

この Statement オブジェクトを生成した Connection オブジェクトを返します。

### int

getFetchDirection()

この Statement オブジェクトから生成される結果セットについて、データベース表から行をフェッチするデフォルトの方向を取り出します。現在サポートされているのは、FETCH\_FORWARD のみです。

### int

getFetchSize()

この Statement オブジェクトから生成される結果セットについて、デフォルトのフェッチ・サイズである結果セット行数を取り出します。現在サポートされているフェッチ・サイズは 1 のみです。

### int

getResultSetConcurrency()

結果セットの並行性を取り出します。現在サポートされているのは、CONCUR\_READ\_ONLY のみです。

### int

getResultSetType()

結果セットの型を判断します。現在サポートされているのは、TYPE\_FORWARD\_ONLY と TYPE\_SCROLL\_INSENSITIVE のみです。

### void

setFetchDirection(int direction)

結果セット行の処理方向についてのヒントをドライバに提供します。

### void

setFetchSize(int rows)

より多くの行が必要なときにデータベースからフェッチする行数についてのヒントを JDBC ドライバに提供します。

## インタフェース ResultSet

### フィールド

**static int**

CONCUR\_READ\_ONLY

更新できない ResultSet オブジェクトの並行性モード。

**static int**

CONCUR\_UPDATABLE

更新できる ResultSet オブジェクトの並行性モード。現在サポートされていません。

**static int**

FETCH\_FORWARD

結果セット内の行が、順方向（先頭から最後）で処理されます。

**static int**

FETCH\_REVERSE

結果セット内の行が、逆方向（最後から先頭）で処理されます。現在サポートされていません。

**static int**

FETCH\_UNKNOWN

結果セット内の行の処理順序は不明です。

**static int**

TYPE\_FORWARD\_ONLY

カーソルが前方にのみ移動する ResultSet オブジェクトの型。

**static int**

TYPE\_SCROLL\_INSENSITIVE

スクロール可能だが、一般に他からの変更は反映されない ResultSet オブジェクトの型。

**static int**

TYPE\_SCROLL\_SENSITIVE

スクロール可能で、一般に他からの変更が反映される `ResultSet` オブジェクトの型。現在サポートされていません。

## メソッド

**boolean**

absolute(int row)

結果セット内の指定された行番号へカーソルを移動します。

**void**

afterLast()

結果セット内の最後尾（最後の行の直後）の行の直後へカーソルを移動します。

**void**

beforeFirst()

結果セット内の先頭（最初の行の直前）へカーソルを移動します。

**boolean**

first()

結果セット内の最初の行へカーソルを移動します。

**Array**

getArray(String colName)

この `ResultSet` オブジェクトのカレント行の SQL ARRAY 値を取得します。

**BigDecimal**

getBigDecimal(int columnIndex)

カレント行の列の値を、完全な精度の `java.math.BigDecimal` オブジェクトとして取得します。

### **BigDecimal**

`getBigDecimal(String columnName)`

カレント行の列の値を、完全な精度の `java.math.BigDecimal` オブジェクトとして取得します。

### **int**

`getConcurrency()`

この結果セットの並行性モードを返します。

### **Date**

`getDate(int columnIndex, Calendar cal)`

カレント行の列の値を、`java.sql.Date` オブジェクトとして取得します。

### **int**

`getFetchDirection()`

この結果セットのフェッチ方向を返します。

### **int**

`getFetchSize()`

この結果セットのフェッチ・サイズを返します。

### **int**

`getRow()`

カレント行の番号を返します。

### **Statement**

`getStatement()`

この `ResultSet` オブジェクトを生成した `Statement` オブジェクトを返します。

### **int**

`getType()`

この結果セットの型を返します。

### **boolean**

`isAfterLast()`

**boolean**

isBeforeFirst()

**boolean**

isFirst()

**boolean**

isLast()

**boolean**

last()

結果セット内の最後の行へカーソルを移動します。

**boolean**

previous()

結果セット内で前の行へカーソルを移動します。

**void**

refreshRow()

カレント行を、データベース内の最新の値でリフレッシュします。現在は何もしません。

**boolean**

relative(int rows)

カーソルを、前後どちらかの方向へ相対的な行数分移動します。

**FALSE を返すメソッド**

このリリースでは削除、挿入または更新をサポートしていないため、次の3つのメソッドは常に FALSE を返します。

**boolean**

rowDeleted()

行が削除されているかどうかを示します。

**boolean**

rowInserted()

カレント行に挿入があったかどうかを示します。

**boolean**

rowUpdated()

カレント行が更新されたかどうかを示します。

**void**

setFetchDirection(int direction)

この結果セット内の行の処理方向についてのヒントを提供します。

**void**

setFetchSize(int rows)

この結果セットに対してより多くの行が必要なときに、データベースからフェッチされる行数についてのヒントを JDBC ドライバに提供します。

## インタフェース DatabaseMetaData

### メソッド

**Connection**

getConnection()

このメタデータ・オブジェクトを生成した接続を取り出します。

**boolean**

supportsResultSetConcurrency(int type, int concurrency)

指定された結果セットの型とともに並行性の型をサポートします。

**boolean**

supportsResultSetType(int type)

指定された結果セットの型をサポートします。

## FALSE を返すメソッド

このリリースでは削除、挿入または更新をサポートしていないため、次のメソッドは常に FALSE を返します。

### boolean

`deletesAreDetected(int type)`

参照できる行の削除を、`ResultSet.rowDeleted()` をコールして検出できるかどうかを示します。

### boolean

`insertsAreDetected(int type)`

参照できる行の挿入を、`ResultSet.rowInserted()` をコールして検出できるかどうかを示します。

### boolean

`othersDeletesAreVisible(int type)`

他からの削除を参照できるかどうかを示します。

### boolean

`othersInsertsAreVisible(int type)`

他からの挿入を参照できるかどうかを示します。

### boolean

`othersUpdatesAreVisible(int type)`

他からの更新を参照できるかどうかを示します。

### boolean

`ownDeletesAreVisible(int type)`

結果セット自身による削除を参照できるかどうかを示します。

### boolean

`ownInsertsAreVisible(int type)`

結果セット自身による挿入を参照できるかどうかを示します。

**boolean**

ownUpdatesAreVisible(int type)

結果セット自身による更新を参照できるかどうかを示します。

**boolean**

updatesAreDetected(int type)

参照できる行の更新を、メソッド `ResultSet.rowUpdated()` をコールして検出できるかどうかを示します。

## インタフェース ResultSetMetaData

### メソッド

**String**

getColumnClassName(int column)

列の値を取り出すためにメソッド `ResultSet.getObject` がコールされた場合に、インスタンスが生成されている Java クラスの完全修飾名を返します。

## インタフェース PreparedStatement

### メソッド

**Result**

setMetaData(getMetaData())

`ResultSet` の列の数、型およびプロパティを取得します。

**void**

setDate(int parameterIndex, Date x, Calendar cal)

指定されたパラメータを、指定された `Calendar` オブジェクトを使用して `java.sql.Date` 値に設定します。



**void**`setTime(int parameterIndex, Time x, Calendar cal)`

指定されたパラメータを、指定された `Calendar` オブジェクトを使用して `java.sql.Time` 値に設定します。

**void**`setTimestamp(int parameterIndex, Timestamp x, Calendar cal)`

指定されたパラメータを、指定された `Calendar` オブジェクトを使用して `java.sql.Timestamp` 値に設定します。



# D

---

---

## サンプル・プログラム

この付録では、Oracle Lite に同梱された Java サンプル・プログラムの使用手順を説明します。説明する内容は、次のとおりです。

- [Java サンプル・プログラムの概要](#)
- [サンプルの実行](#)

## Java サンプル・プログラムの概要

Oracle\_Home¥Mobile¥Sdk¥Examples¥Java ディレクトリには、Java スタアド・プロシージャ、Java アクセス・クラス、Java レプリケーション・クラスおよび JDBC を Oracle Lite で使用する方法を示すサンプル・プログラムが含まれています。

Java のサンプル用ディレクトリには、次のファイルが含まれています。

1. Stoproex.sql
2. INVENTORY.java
3. JDBCEx.java
4. plsqllex.sql
5. PLSQLEX.java

次の項ではサンプルについて説明します。Java のクラス、メソッドおよびファイルの名前は、大 / 小文字が区別されます。SQL から Java プログラムを実行するときは、大 / 小文字をそのまま保持するために名前は二重引用符で囲む必要があります。

---

---

**注意：** サンプルを JDK 1.1.8 とともに実行すると、すべてのシステムで一貫性のある結果が生成されない場合があります。Symantec の JIT コンパイラを使用すると、この問題が解決される可能性があります。

---

---

## JDBC のサンプル

ファイル JDBCEx.java には、JDBC クラスを使用して PRODUCT 表の行を選択し情報を表示する Java プログラムのサンプルが含まれています。

## PL/SQL から Java サンプルへの変換

Oracle の PL/SQL 言語で作成されたスタアド・プロシージャとトリガーは Java に変換できます。PLSQLEX.java に含まれている Java プログラムのいくつかは、『Oracle Server PL/SQL ユーザーズ・ガイドおよびリファレンス』マニュアルに説明されている PL/SQL プログラムと対応しています。Plsqllex.sql には、Java スタアド・プロシージャを起動する SQL 文が含まれています。

## Java ストアド・プロシージャのサンプル

Java ストアド・プロシージャのサンプルは、Oracle Lite の表にクラスを手動でアタッチする方法を示します。このかわりに、loadjava を使用してクラスを Oracle Lite データベースにロードし、CREATE PROCEDURE 文または CREATE FUNCTION 文を使用してそのクラスの方法を SQL にパブリッシュすることもできます。このモデルでは、クラスはデータベース表にはアタッチしません。ストアド・プロシージャを開発するパブリッシュ・モデルの説明は、第 2 章「Java ストアド・プロシージャとトリガー」の「モデル 1: Load and Publish ストアド・プロシージャ開発モデルの使用」を参照してください。

ファイル **Stoproex.sql** には、サンプル・スキーマを作成する SQL 文が含まれています。このスクリプトは、Java サンプルを実行する前に SQL\*Plus を使用して実行する必要があります。サンプル・スキーマには、次の 3 つの表が含まれています。

表	説明
PRODUCT	製品に関する情報を格納します。
PRODUCT_COMPOSITION	製品の構成に関する情報を格納します。表の各行は、その製品の構成に必要な部品の数量を追跡します。
INVENTORY	倉庫にある製品の数量を格納します。

**Stoproex.sql** には、表に行を挿入する文、INVENTORY 表に Java クラスをアタッチする文および INVENTORY 表の QTY 列に AFTER UPDATE トリガーを作成する文も含まれています。

INVENTORY 表にアタッチされる Java クラスは、**INVENTORY.java** ファイルで定義されます。このクラスには、SHIP\_PRODUCT という静的メソッドが 1 つと、SHIP および CHECK\_INVENTORY という非静的（インスタンス）メソッドが 2 つ含まれています。

SHIP\_PRODUCT メソッドは引数を 3 つ取ります。Connection オブジェクト、製品 ID および顧客に出荷される製品の数です。

**Stoproex.sql** は、次の SQL 文を使用してこのメソッドをコールします。

```
SELECT inventory.ship_product(100,1) FROM DUAL FOR UPDATE;
```

次の点に注意してください。

1. 静的メソッドは、*table\_name.method\_name* として参照する必要があります。静的メソッドを実行するための FROM 句は、常に疑似表 DUAL を参照する必要があります。
2. SQL は `inventory.ship_product` を大文字に変換します。これは、**INVENTORY.java** 内のメソッド名が SHIP\_PRODUCT のためです。表「Inventory」とメソッド「shipProduct」を指定する場合は、両方の名前を "Inventory"."shipProduct" のように二重引用符で囲みます。
3. Connection オブジェクトは、メソッドへの引数として明示的には指定されません。型が `java.sql.Connection` の引数には Oracle Lite がカレントの接続を提供します。

Java メソッド SHIP では、JDBC クラスを使用して Oracle Lite データベースにアクセスします。引数として渡される接続から文を作成し、SELECT 文を実行します。SELECT 文は、INVENTORY.java にも定義されている Java 非静的メソッド SHIP を実行します。

メソッド SHIP は、出荷する製品数を更新します。SHIP が非静的メソッドであるため、Oracle Lite はこのメソッドをコールする前に INVENTORY クラスのインスタンスを作成します。ATTACH 文の WITH CONSTRUCTOR ARGS 句に指定されている列を取るコンストラクタを使用して、インスタンスを作成します。

このサンプルでは、INVENTORY 表の QTY 列に対する AFTER UPDATE トリガーが作成されているため、UPDATE のたびに CHECK\_INVENTORY メソッドが実行されます。CHECK\_INVENTORY は非静的メソッドであるため、Oracle Lite では行インスタンスを使用するか、(行インスタンスがない場合は) 新規インスタンスを作成します。

更新された在庫数が在庫のしきい値を下回った場合は、CHECK\_INVENTORY メソッドが PRODUCT\_COMPOSITION 表を使用して製品の構成部品を参照します。また、各部品の数を更新して、在庫の補充に必要な製品数に見合うように部品数を更新します。この更新は、最終的な数に到達するまで再帰的に発生します。到達した時点で、CHECK\_INVENTORY により製品の注文が入れられます。

## サンプルの実行

Java サンプルを実行するには、次の手順に従います。

1. サンプル・ディレクトリ `Oracle_Home¥Mobile¥Sdk¥Examples¥JAVA` に移動します。次に例を示します。

```
cd ¥ORACLE_HOME¥MOBILE¥SDK¥EXAMPLES¥JAVA
```

2. 「.」(カレント・ディレクトリ) が CLASSPATH に含まれていない場合は、追加します。

```
set CLASSPATH=.;%CLASSPATH%
```

3. DOS 版の SQL\*Plus を使用して SQL スクリプトを実行します。例を次に示します。

```
mSQL system/mgr@ODBC:polite @stoproex.sql
```

## JDBC のサンプルの実行

JDBC のサンプルを使用するには、`Stoproex.sql` スクリプトを実行して Oracle Lite データベースに PRODUCT 表をインストールします。

```
mSQL system/mgr@ODBC:polite @stoproex.sql
```

次のコマンドを使用してソース・ファイルをコンパイルします。

```
javac JDBCEX.java
```

コンパイルされたクラスを次のように実行します。

```
java JDBCEX
```

## PL/SQL 変換のサンプルの実行

**PLSQLEX.java** を実行するには、SQL\*Plus を起動します。

```
mSQL system/mgr@odbc:polite
```

SQL\*Plus のプロンプトで次のスクリプトを実行します。

```
@plsqllex.sql
```

Java ソース・ファイル **PLSQLEX.java** を表にアタッチします。

```
alter table temp attach java source "PLSQLEX" in '.';
```

表のメソッドをコールするには、次のように入力します。

```
select temp."sampleOne"() from dual for update;
```

結果を表示するには、次のように指定します。

```
select * from temp;
```

その他のサンプルの詳細は、**PLSQLEX.java** を参照してください。サンプルには、`sampleOne` から `sampleFour` という名前が付けられています。

## Java ストアド・プロシージャのサンプルの実行

**Stoproex.sql** スクリプトを実行し、ストアド・プロシージャのサンプルに必要な表とストアド・プロシージャをインストールします。

```
mSQL system/ mgr@ODBC:polite @stoproex.sql
select inventory.ship_product(p,q) from dual;
```

スクリプトが完了すると、在庫表の内容が表示されます。SQL\*Plus のプロンプトで次のように入力します。

```
select * from inventory;
```

PID	QTY	THRESHOLD
100	1	1
101	-6	2
102	-26	8
103	-26	8

表の中の負の数値は、部品 101、102、103 に補充が必要であることを示します。



# 索引

## A

---

ALTER TABLE 文, 2-16, 2-18, 2-25  
ALTER TRIGGER 文, 2-25

## B

---

BLOB, 3-6, 3-11  
 値の取得, 3-7  
 値の設定, 3-7

## C

---

CallableStatement クラス, 3-5  
CLOB, 3-6, 3-7, 3-11  
 値の取得, 3-7  
 値の設定, 3-7  
close メソッド, 3-9  
Connection オブジェクト、引数として渡される, 2-22  
CREATE FUNCTION 文, 2-9  
CREATE JAVA 文, 2-8, 2-14  
CREATE PROCEDURE 文, 2-9  
CREATE TRIGGER 文, 2-24  
createStatement メソッド, 3-4

## D

---

DETACH AND DELETE 文, 2-18  
DROP FUNCTION 文, 2-13  
DROP JAVA 文, 2-13  
DROP PROCEDURE 文, 2-13  
DROP TRIGGER 文, 2-25  
dropjava, 1-5, 2-12 ~ 2-13  
 オプション, 2-12

引数, 2-12  
ファイル名の指定, 2-13

## E

---

executeQuery メソッド, 3-4

## F

---

flush メソッド, 3-9  
force、loadjava オプション, 2-5

## G

---

getAsciiOutputStream, 3-7  
getAsciiStream, 3-7  
getBinaryOutputStream, 3-6  
getBinaryStream, 3-6  
getBLOB, 3-8  
getBytes, 3-6  
getCharacterOutputStream, 3-7  
getCharacterStream, 3-7  
getChars, 3-7  
getCLOB, 3-8  
getConnection, 3-6, 3-7  
getSubString, 3-7

## I

---

isConvertibleTo, 3-6, 3-7

## J

---

JAR ファイルのロード, 2-6

Java

Oracle8i Lite の機能, 1-2

管理ツール, 1-4

ストアド・プロシージャ, 1-3, 2-2

静的メソッド, 2-15

トリガー, 1-3

非静的メソッド, 2-15

JavaVM, 「JVM」を参照

Java アクセス・クラス

JDBC との相互運用性, 3-10

JDBC

JAC との相互運用性, 3-10

拡張, 1-2, 3-6 ~ 3-10

サンプル・アプリケーション, B-2

ストアド・プロシージャ, 3-4

設定, 3-2

説明, 1-2, 3-1

標準実装, 1-2

JVM, 1-3, 2-3, 2-18, 2-20, A-5

## L

---

length メソッド, 3-6, 3-7

loadjava, 1-5, 2-5 ~ 2-8

オプション, 2-5

構文, 2-5

引数, 2-5

ファイル名の指定, 2-6

例, 2-8

loadjvm40.dll, 2-20

## M

---

makeJdbcArray, 3-6, 3-7

markSupported メソッド, 3-9

mark メソッド, 3-9

## O

---

ODBC、ストアド・プロシージャのコール, 2-20, 2-23

Oracle8i Lite

Java 機能, 1-2

サンプル・アプリケーション, D-2

ツール, 1-4

OracleCallableStatement クラス, 3-8

OraclePreparedStatement クラス, 3-7

OracleResultSet クラス, 3-8

## P

---

POLClobReader クラス, 3-8

POLClobWriter クラス, 3-9

POLConnection, 2-22

POLLobInputStream クラス, 3-8

POLLobOutputStream クラス, 3-8

PreparedStatement クラス, 3-5

putBytes, 3-6

putChars, 3-7

putString, 3-7

## R

---

ready メソッド, 3-8

read メソッド, 3-8

reset メソッド, 3-9

## S

---

SELECT 文、ストアド・プロシージャのコール, 2-17

setBLOB, 3-7

setCLOB, 3-7

skip メソッド, 3-9

SQLDescribeCol, 2-23

SQLJ

設定, 4-3

説明, 1-3, 4-1

SQLNumResultCols, 2-23

## T

---

toJdbc メソッド, 3-6, 3-7

## V

---

verbose、loadjava オプション, 2-5

## W

---

write メソッド, 3-8, 3-9

## Z

---

ZIP ファイルのロード, 2-6

## あ

---

アタッチ、クラス, 2-16

## う

---

埋込み SQL, 「SQL」を参照

## か

---

環境、設定, 1-5

## き

---

行レベルのトリガー, 2-2  
文レベルのトリガーとの比較, 2-24

## く

---

クラス  
管理ツール, 1-4

## こ

---

コール仕様  
作成, 2-9, A-5  
サンプル, 2-10, 2-11, 2-14  
コール、ストアド・プロシージャ, 2-10, 2-20

## さ

---

削除、ストアド・プロシージャ, 2-12

## す

---

スキーマ・オブジェクト名, 2-6, 2-7  
ストアド・プロシージャ, 1-3  
SQL へのパブリッシュ, 2-9, A-5  
行レベル, 2-16  
コール, 2-10, 2-17, 3-5  
削除, 1-5, 2-12, 2-18  
説明, 1-3, 2-2  
表レベル, 2-15

複数行を返す, 2-22

命名, 2-3

例, 2-13, A-1

ストアド・プロシージャの削除, 1-5

スレッド、マルチスレッドからのストアド・プロシージャのコール, 2-20

## せ

---

設定, 1-5

## そ

---

相互運用性、JAC と JDBC, 3-10

## つ

---

ツール  
開発, 1-5  
クラス管理, 1-4

## て

---

データ型, 1-4, 2-21  
データベース  
JDBC による接続, 3-3

## と

---

問合せ  
JDBC 内, 3-4  
トランザクション  
分離レベル, 3-5  
トリガー, 1-3  
行レベル, 2-2  
削除, 2-25  
作成, 2-24, A-6  
説明, 1-3, 2-2, 2-24  
引数, 2-28  
文レベル, 2-2  
例, 2-26, A-1  
トリガーの使用可能, 2-25  
トリガーの使用禁止, 2-25

## は

---

パブリッシュ、ストアド・プロシージャ, 2-9, A-5  
パラメータ、ストアド・プロシージャ, 2-22, 3-5

## ひ

---

引数、トリガー, 2-28

## ふ

---

分離レベル、トランザクション, 3-5  
文レベルのトリガー, 2-2  
行レベルのトリガーとの比較, 2-24

## ま

---

マルチスレッド・プログラムからのストアド・プロ  
シージャのコール, 2-20

## め

---

命名、ストアド・プロシージャ, 2-3

## ろ

---

ロード

JAR ファイル, 2-6  
ZIP ファイル, 2-6  
クラス, 1-5, 2-4, 2-5