Oracle® Database
JDBC 開発者ガイド
19c

F16162-09(原本部品番号:E96471-14)

2023年5月

このマニュアルでは、Oracle JDBCドライバを使用して強力なJavaデータベース・アプリケーションを開発する方法を説明します。

タイトルおよび著作権情報

Oracle Database JDBC開発者ガイド, 19c

F16162-09

Copyright © 1999, 2023, Oracle and/or its affiliates.

原本著者: Tulika Das

原本協力著者: Brian Martin、Venkatasubramaniam Iyer、Elizabeth Hanes Perry、Brian Wright、Thomas Pfaeffle

協力者: Kuassi Mensah、Douglas Surber、Paul Lo、Ed Shirk、Tong Zhou、Jean de Lavarene、Rajkumar Irudayaraj、Ashok Shivarudraiah、Angela Barone、Rosie Chen、Sunil Kunisetty、Joyce Yang、Mehul Bastawala、Luxi Chidambaran、Vidya Nayak、Srinath Krishnaswamy、Swati Rao、Pankaj Chand、Aman Manglik、Longxing Deng、Magdi Morsi、Ron Peterson、Ekkehard Rohwedder、Catherine Wong、Scott Urman、Jerry Schwarz、Steve Ding、Soulaiman Htite、Anthony Lai、Prabha Krishna、Ellen Siegal、Susan Kraft、Sheryl Maring

目次

- 表一覧
- タイトルおよび著作権情報
- はじめに
  - 対象読者
  - ドキュメントのアクセシビリティ
  - 関連ドキュメント
  - 表記規則
- 『Oracle Database JDBC開発者ガイド』の今回のリリースにおける変更点
  - Oracle Database 19cでの変更点
    - 新機能
    - 非推奨となった機能
    - その他の変更
- 第I部 概要
  - 1 JDBCの概要
    - 1.1 Oracle JDBCドライバの概要
    - 1.2 適切なドライバの選択
    - 1.3 JDBC OCIドライバとJDBC Thinドライバの機能の相違点
    - 1.4 環境およびサポート
      - 1.4.1 サポートされるJDKおよびJDBCのバージョン
      - 1.4.2 JNI環境およびJava環境
      - 1.4.3 JDBCとIDE
    - 1.5 機能リスト
  - 2 スタート・ガイド
    - 2.1 Oracle JDBCドライバのバージョン互換性
    - 2.2 JDBCクライアント・インストールの検証
      - 2.2.1 インストールされたディレクトリとファイルの確認
      - 2.2.2 環境変数のチェック
      - 2.2.3 Javaコードのコンパイルと実行の確認
      - 2.2.4 JDBCドライバのバージョンの確認
      - 2.2.5 JDBCおよびデータベース接続のテスト
    - 2.3 JDBCでの基本ステップ
      - 2.3.1 パッケージのインポート
      - 2.3.2 データベースへの接続のオープン
      - 2.3.3 Statementオブジェクトの作成
      - 2.3.4 問合せの実行と結果セット・オブジェクトの取出し
      - 2.3.5 結果セット・オブジェクトの処理
      - 2.3.6 結果セット・オブジェクトとStatementオブジェクトのクローズ
      - 2.3.7 データベースの変更
      - 2.3.8 変更のコミットについて
        - 2.3.8.1 コミット動作の変更
      - 2.3.9 接続のクローズ

            - 2.4 サンプル: 接続、問合せおよび結果処理
            - 2.5 非表示列のサポート
            - 2.6 JSONデータの検証のサポート
            - 2.7 暗黙的結果のサポート
            - 2.8 軽量接続検証のサポート
            - 2.9 データベース・ノードの優先度付け解除のサポート
            - 2.10 Traffic DirectorモードのOracle Connection Managerのサポート
                - 2.10.1 Traffic DirectorモードのOracle Connection Managerの実行モード
                - 2.10.2 Traffic DirectorモードのOracle Connection Managerの利点
                - 2.10.3 Traffic DirectorモードのOracle Connection Managerの制限事項
            - 2.11 Memoptimized Rowstoreの高速収集のサポート
            - 2.12 JDBCプログラムでのストアド・プロシージャ・コール
                - 2.12.1 PL/SQLストアド・プロシージャ
                - 2.12.2 Javaストアド・プロシージャ
            - 2.13 SQL例外の処理について
- 第II部 Oracle JDBC
    - 3 JDBC標準のサポート
        - 3.1 JDBC 2.0標準のサポート
            - 3.1.1 データ型のサポート
            - 3.1.2 標準機能のサポート
            - 3.1.3 拡張機能のサポート
            - 3.1.4 JDBC2.0 Standard Extension APIとオラクル社独自のパフォーマンス強化API
        - 3.2 JDBC 3.0標準のサポート
            - 3.2.1 トランザクション・セーブポイントの概要
                - 3.2.1.1 セーブポイントの作成について
                - 3.2.1.2 セーブポイントまでのロールバックについて
                - 3.2.1.3 セーブポイントの解放について
                - 3.2.1.4 セーブポイント・サポートのチェックについて
                - 3.2.1.5 セーブポイントに関するノート
            - 3.2.2 自動生成キーの取出し
                - 3.2.2.1 java.sql.Statement
                - 3.2.2.2 サンプル・コード
                - 3.2.2.3 自動生成キーの制約
            - 3.2.3 JDBC 3.0のLOBインタフェース・メソッド
            - 3.2.4 結果セットの保持機能
        - 3.3 JDBC 4.0標準のサポート
            - 3.3.1 ラッパー・パターンのサポート
            - 3.3.2 SQLXML型
            - 3.3.3 機能拡張された例外階層とSQLException
            - 3.3.4 ROWIDデータ型
            - 3.3.5 LOBの作成
            - 3.3.6 各国語文字セットのサポート

- 3.4 JDBC 4.1標準のサポート
    - 3.4.1 setClientInfoメソッド
    - 3.4.2 getObjectメソッド
- 3.5 JDBC 4.2標準のサポート
- 4 Oracle拡張機能
    - 4.1 Oracle拡張機能の概要
    - 4.2 Oracle拡張機能
        - 4.2.1 JDBCを使用したデータベース管理
        - 4.2.2 Oracleデータ型のサポート
        - 4.2.3 Oracleオブジェクトのサポート
        - 4.2.4 スキーマの命名サポート
        - 4.2.5 DML RETURNING
        - 4.2.6 PL/SQL連想配列
    - 4.3 Oracle JDBCパッケージ
        - 4.3.1 パッケージoracle.sql
        - 4.3.2 パッケージoracle.jdbc
    - 4.4 Oracle文字データ型のサポート
        - 4.4.1 SQL CHARデータ型
        - 4.4.2 SQL NCHARデータ型
        - 4.4.3 クラスoracle.sql.CHAR
    - 4.5 その他のOracle型拡張機能
        - 4.5.1 Oracle ROWID型
        - 4.5.2 Oracle REF CURSOR型カテゴリ
        - 4.5.3 Oracle BINARY_FLOAT型およびBINARY_DOUBLE型
        - 4.5.4 Oracle SYS.ANYTYPE型およびSYS.ANYDATA型
        - 4.5.5 oracle.jdbcパッケージ
            - 4.5.5.1 インタフェースoracle.jdbc.OracleConnection
            - 4.5.5.2 インタフェースoracle.jdbc.OracleStatement
            - 4.5.5.3 インタフェースoracle.jdbc.OraclePreparedStatement
            - 4.5.5.4 インタフェースoracle.jdbc.OracleCallableStatement
            - 4.5.5.5 インタフェースoracle.jdbc.OracleResultSet
            - 4.5.5.6 インタフェースoracle.jdbc.OracleResultSetMetaData
            - 4.5.5.7 クラスoracle.jdbc.OracleTypes
    - 4.6 DML RETURNING
        - 4.6.1 Oracle固有のAPI
        - 4.6.2 DML RETURNING文の実行について
        - 4.6.3 DML RETURNINGの例
        - 4.6.4 DML RETURNINGの制限事項
    - 4.7 PL/SQL連想配列へのアクセス
- 5 JDBC Thin固有の機能
    - 5.1 JDBC Thinクライアントの概要
    - 5.2 サポートされるその他の機能
        - 5.2.1 ネイティブXAのデフォルトでのサポート

            - 5.2.2 トランザクション・ガードのサポート
            - 5.2.3 アプリケーション・コンティニュイティのサポート
    - 6 JDBC OCIドライバ固有の機能
        - 6.1 OCI接続プーリング
        - 6.2 透過的アプリケーション・フェイルオーバー
        - 6.3 OCIネイティブXA
        - 6.4 OCI Instant Client
            - 6.4.1 Instant Clientの概要
            - 6.4.2 OCI Instant Client共有ライブラリ
            - 6.4.3 Instant Clientの利点
            - 6.4.4 JDBC OCI Instant Clientのインストール手順
            - 6.4.5 Instant Clientの使用
            - 6.4.6 Instant Client共有ライブラリのパッチについて
            - 6.4.7 データ共有ライブラリとZIPファイルの再生成
            - 6.4.8 OCI Instant Client用のデータベース接続名
            - 6.4.9 OCI Instant Clientの環境変数
        - 6.5 Instant Client Light (English)について
            - 6.5.1 Instant Client Light (English)のデータ共有ライブラリ
            - 6.5.2 グローバリゼーション設定
            - 6.5.3 操作
            - 6.5.4 Instant Client Light (English)のインストール
    - 7 サーバー側内部ドライバ
        - 7.1 サーバー側内部ドライバの概要
        - 7.2 データベースへの接続
        - 7.3 セッション・コンテキストおよびトランザクション・コンテキストについて
        - 7.4 サーバー上でのJDBCのテスト
        - 7.5 サーバーへのアプリケーションのロード
            - 7.5.1 loadjavaユーティリティの使用
            - 7.5.2 JVMコマンドラインの使用
- 第III部 接続とセキュリティ
    - 8 データソースおよびURL
        - 8.1 データソースについて
            - 8.1.1 JNDIのOracleデータソース・サポートの概要
            - 8.1.2 データソースの機能とプロパティ
            - 8.1.3 データソース・インスタンスの作成と接続
            - 8.1.4 データソース・インスタンスの作成、JNDIへの登録および接続
            - 8.1.5 サポートされている接続プロパティ
            - 8.1.6 SYSログオンのためのロールの使用方法について
            - 8.1.7 データベース・リモート・ログインの構成
            - 8.1.8 Bequeath接続とSYSログオンの使用
            - 8.1.9 Oracleパフォーマンス拡張機能のプロパティの設定
            - 8.1.10 ネットワーク・データ圧縮のサポート
        - 8.2 データベースURLとデータベース指定子

- 8.2.1 インターネット・プロトコル・バージョン6のサポート
- 8.2.2 HTTPSプロキシ構成のサポート
- 8.2.3 データベース指定子
- 8.2.4 Thin形式のサービス名の構文
- 8.2.5 Easy Connect Plusのサポート
  - 8.2.5.1 TCPSプロトコルのサポート
  - 8.2.5.2 複数のホストとポートのサポート
  - 8.2.5.3 接続文字列での接続プロパティの受渡しのサポート
- 8.2.6 接続の再試行での遅延のサポート
- 8.2.7 TNSNames別名の構文
- 8.2.8 LDAP構文
- 9 JDBCクライアント側セキュリティ機能
  - 9.1 IAMのトークンベース認証のサポート
    - 9.1.1 ファイル・システムの使用
    - 9.1.2 oracle.jdbc.accessToken接続プロパティの使用
    - 9.1.3 OracleConnectionBuilderインタフェースの使用
    - 9.1.4 OracleDataSourceクラスの使用
  - 9.2 Azure ADのトークンベース認証のサポート
    - 9.2.1 ファイル・システムの使用
    - 9.2.2 oracle.jdbc.accessToken接続プロパティの使用
    - 9.2.3 OracleConnectionBuilderインタフェースの使用
    - 9.2.4 OracleDataSourceクラスの使用
  - 9.3 Oracle Advanced Securityのサポート
    - 9.3.1 Oracle Advanced Securityの概要
    - 9.3.2 JDBC OCIドライバによるOracle Advanced Securityのサポート
    - 9.3.3 JDBC ThinドライバによるOracle Advanced Securityのサポート
  - 9.4 ログイン認証のサポート
  - 9.5 厳密認証のサポート
  - 9.6 ネットワーク暗号化と整合性のサポート
    - 9.6.1 JDBCによるネットワーク暗号化と整合性のサポートの概要
    - 9.6.2 JDBC OCIドライバによるデータ暗号化と整合性のサポート
    - 9.6.3 JDBC Thinドライバによるデータ暗号化と整合性のサポート
    - 9.6.4 Javaでの暗号化および整合性パラメータの設定
  - 9.7 TLSのサポート
    - 9.7.1 JDBCによるTLSのサポートの概要
    - 9.7.2 証明書とウォレットの管理について
    - 9.7.3 キーと証明書のコンテナについて
    - 9.7.4 JDBC ThinおよびJKSを使用したTLSバージョン1.2でのデータベース接続
    - 9.7.5 TLS接続の自動構成
      - 9.7.5.1 プロバイダの解決
      - 9.7.5.2 キーストア・タイプ(KSS)の自動解決
    - 9.7.6 デフォルトのTLSコンテキストのサポート
    - 9.7.7 キーストア・サービスのサポート

- 9.8 Kerberosのサポート
  - 9.8.1 JDBCによるKerberosのサポートの概要
  - 9.8.2 Kerberosを使用するためのWindowsの構成
  - 9.8.3 Kerberosを使用するためのOracle Databaseの構成
  - 9.8.4 Kerberosを使用するコード例
- 9.9 RADIUSのサポート
  - 9.9.1 JDBCによるRADIUSのサポートの概要
  - 9.9.2 RADIUSを使用するためのOracle Databaseの構成
  - 9.9.3 RADIUSを使用するコード例
- 9.10 セキュアな外部パスワード記憶域
- 10 プロキシ認証
  - 10.1 プロキシ認証について
  - 10.2 各種のプロキシ接続
  - 10.3 プロキシ接続の作成
  - 10.4 プロキシ・セッションのクローズ
  - 10.5 プロキシ接続のキャッシュ
  - 10.6 プロキシ接続の制限
- 第IV部 データへのアクセスと操作
  - 11 Oracleデータへのアクセスと操作
    - 11.1 データ型マッピング
      - 11.1.1 マッピングの表
      - 11.1.2 マッピングに関するノート
    - 11.2 データ変換での考慮事項
      - 11.2.1 標準型とOracle型
      - 11.2.2 SQL NULLデータの変換について
      - 11.2.3 NULLのテストについて
    - 11.3 結果セットと文拡張機能
    - 11.4 OracleのgetおよびsetメソッドとJDBCの比較
      - 11.4.1 標準getObjectメソッド
      - 11.4.2 Oracle getOracleObjectメソッド
      - 11.4.3 getObjectおよびgetOracleObject戻り型のまとめ
      - 11.4.4 その他のgetXXXメソッド
        - 11.4.4.1 getXXXメソッドの戻り型
        - 11.4.4.2 getXXXメソッドに関する特別なノート
      - 11.4.5 getObjectおよびgetXXXから戻されるオブジェクトのデータ型
      - 11.4.6 setObjectメソッドとsetOracleObjectメソッド
      - 11.4.7 その他のsetXXXメソッド
        - 11.4.7.1 入力データのバインド
        - 11.4.7.2 WHERE句にCHARデータをバインドするためのメソッドsetFixedCHAR
    - 11.5 結果セット・メタデータ拡張機能の使用方法
    - 11.6 SQL CALL文とCALL INTO文の使用について
  - 12 JDBC内のJavaストリーム

- 12.1 Javaストリームの概要
- 12.2 LONGまたはLONG RAW列のストリームについて
  - 12.2.1 LONGまたはLONG RAW列のストリームの概要
  - 12.2.2 LONG RAWデータの変換
  - 12.2.3 LONGデータの変換
  - 12.2.4 例: LONG RAWデータのストリーム
  - 12.2.5 LONGまたはLONG RAWのストリーム回避について
- 12.3 CHAR、VARCHARまたはRAW列のストリームについて
- 12.4 LOBおよび外部ファイルのストリームについて
- 12.5 データ・ストリームと複数列の関係
- 12.6 ストリームのクローズ
- 12.7 ストリームに関するノート
  - 12.7.1 ストリーム・データを使用する際の注意について
  - 12.7.2 setBytesとsetStringの制限を回避するためのストリームの使用方法について
  - 12.7.3 ストリームと行のプリフェッチの関係
- 13 Oracleオブジェクト型の操作
  - 13.1 Oracleオブジェクトのマッピングについて
  - 13.2 Oracleオブジェクト用のデフォルトSTRUCTクラスの使用方法について
    - 13.2.1 Structクラスの使用の概要
    - 13.2.2 STRUCTオブジェクトと属性の取出し
    - 13.2.3 STRUCTオブジェクトの作成について
    - 13.2.4 STRUCTオブジェクトの文へのバインド
    - 13.2.5 STRUCT自動属性バッファリング
  - 13.3 Oracleオブジェクト用のカスタム・オブジェクト・クラスの作成と使用方法について
    - 13.3.1 カスタム・オブジェクト・クラスの作成および使用の概要
    - 13.3.2 OracleDataとSQLDataの利点
    - 13.3.3 SQLDataを実装するための型マップについて
    - 13.3.4 SQLDataを実装するための型マップの作成とマッピングの定義について
      - 13.3.4.1 型マップの作成およびマッピングの定義の概要
      - 13.3.4.2 既存の型マップへのエントリの追加
      - 13.3.4.3 新しい型マップの作成
      - 13.3.4.4 型マップで指定されていないオブジェクト型のインスタンス化について
    - 13.3.5 SQLData実装によるデータの読取りおよび書込みについて
    - 13.3.6 OracleDataインタフェースについて
    - 13.3.7 OracleData実装によるデータの読取りおよび書込みについて
    - 13.3.8 その他のOracleDataの使用方法
  - 13.4 オブジェクト型の継承
    - 13.4.1 サブタイプの作成について
    - 13.4.2 サブタイプに対してカスタマイズされたクラスの実装について
      - 13.4.2.1 型継承階層のためのOracleDataの使用について
      - 13.4.2.2 型継承階層のためのSQLDataの使用について
    - 13.4.3 サブタイプ・オブジェクトの取得について
    - 13.4.4 サブタイプ・オブジェクトの作成

            - 13.4.5 サブタイプ・オブジェクトの送信
            - 13.4.6 サブタイプ・データ・フィールドへのアクセス
            - 13.4.7 継承メタデータ・メソッド
        - 13.5 オブジェクト型の記述について
            - 13.5.1 オブジェクト・メタデータの取出し機能
            - 13.5.2 オブジェクト・メタデータの取得
- 14 LOBとBFILEの操作
    - 14.1 LOBデータ型
    - 14.2 Oracle SecureFiles
    - 14.3 LOBのデータ・インタフェース
        - 14.3.1 効率的なメカニズム
        - 14.3.2 入力
        - 14.3.3 出力
        - 14.3.4 CallableSatementとIN OUTパラメータ
        - 14.3.5 サイズの制限
    - 14.4 LOBロケータ・インタフェース
    - 14.5 一時LOBの使用について
    - 14.6 OpenおよびCloseメソッドによる通常のLOBのオープンについて
    - 14.7 BFILEの操作について
- 15 Oracleオブジェクト参照の使用
    - 15.1 オブジェクト参照用Oracle拡張機能
    - 15.2 オブジェクト参照の取出しと引渡し
        - 15.2.1 結果セットからのオブジェクト参照の取出し
        - 15.2.2 オブジェクト参照のコール可能文からの取出し
        - 15.2.3 オブジェクト参照のプリペアド文への引渡し
    - 15.3 オブジェクト値に対する、オブジェクト参照を介したアクセスと更新
- 16 Oracleコレクションの操作
    - 16.1 コレクションのためのOracle拡張機能
        - 16.1.1 Oracleコレクションの概要
        - 16.1.2 コレクションのインスタンス化に関する選択
        - 16.1.3 コレクションの作成
        - 16.1.4 マルチ・レベルのコレクション型の作成
    - 16.2 コレクション機能の概要
    - 16.3 ARRAYパフォーマンス拡張要素メソッド
        - 16.3.1 Javaプリミティブ型の配列としてのoracle.sql.ARRAY要素へのアクセスについて
        - 16.3.2 ARRAY自動要素バッファリング
        - 16.3.3 ARRAY自動索引作成
    - 16.4 配列の作成と使用方法
        - 16.4.1 ARRAYオブジェクトの作成
        - 16.4.2 配列とその要素の取出し
            - 16.4.2.1 配列の取出しについて
            - 16.4.2.2 データ取出しメソッド
            - 16.4.2.3 データ取出しメソッドの比較

- 16.4.2.4 型マップに従った構造化オブジェクト配列の要素の取出し
- 16.4.2.5 配列要素のサブセットの取出し
- 16.4.2.6 oracle.sql.Datum配列への配列要素の取出し
- 16.4.2.7 マルチ・レベル・コレクション要素へのアクセスについて
- 16.4.3 配列の文オブジェクトへの引渡し
- 16.5 型マップを使用した配列要素のマップ
- 17 結果セット
- 17.1 結果セットのサポートのOracle JDBC実装概要
- 17.2 結果セットの制限事項およびダウングレード・ルール
- 17.3 更新の競合回避について
- 17.4 行フェッチ・サイズ
- 17.4.1 フェッチ・サイズの設定
- 17.4.2 フェッチ方向のプリセット
- 17.5 行の再フェッチについて
- 17.6 内部的および外部的に加えられたデータベース変更の参照について
- 17.6.1 外部変更の可視性と検出
- 17.6.2 内部変更および外部変更の可視性の概要
- 17.6.3 Scroll-Sensitive結果セットのOracle実装
- 18 JDBC RowSet
- 18.1 JDBC RowSetの概要
- 18.1.1 RowSetのプロパティ
- 18.1.2 イベントおよびイベント・リスナー
- 18.1.3 コマンド・パラメータおよびコマンド実行
- 18.1.4 RowSetの横断について
- 18.2 CachedRowSetについて
- 18.3 JdbcRowSetについて
- 18.4 WebRowSetについて
- 18.5 FilteredRowSetについて
- 18.6 JoinRowSetについて
- 19 グローバリゼーション・サポート
- 19.1 グローバリゼーション・サポートの提供について
- 19.2 NCHAR、NVARCHAR2、NCLOBおよびdefaultNCharプロパティ
- 19.3 JDK 6での各国語文字セット用の新しいメソッド
- 第V部 パフォーマンスとスケーラビリティ
- 20 文キャッシュと結果セット・キャッシュ
- 20.1 文キャッシュについて
- 20.1.1 文キャッシュの基本
- 20.1.2 暗黙的文キャッシュ
- 20.1.3 明示的文キャッシュ
- 20.2 文キャッシュの使用について
- 20.2.1 文キャッシュの有効化および無効化について
- 20.2.2 キャッシュされた文のクローズについて
- 20.2.3 暗黙的文キャッシュの使用方法について

            - 20.2.3.1 文の割当ておよび暗黙的文キャッシュで使用されるメソッド
        - 20.2.4 明示的文キャッシュの使用方法について
            - 20.2.4.1 明示的にキャッシュされた文を取り出す場合に使用するメソッド
    - 20.3 文オブジェクトの再利用について
        - 20.3.1 プールされた文の使用について
        - 20.3.2 プールされた文のクローズについて
    - 20.4 結果セット・キャッシュについて
        - 20.4.1 サーバー側結果セット・キャッシュ
        - 20.4.2 クライアント側結果セット・キャッシュ
            - 20.4.2.1 クライアント側結果セット・キャッシュの有効化
            - 20.4.2.2 クライアント側結果セット・キャッシュの利点
            - 20.4.2.3 JDBCでの使用ガイドライン
                - 20.4.2.3.1 RESULT_CACHE_MODEパラメータ
                - 20.4.2.3.2 表注釈
                - 20.4.2.3.3 SQLヒント
- 21 パフォーマンス拡張機能
    - 21.1 バッチ更新
        - 21.1.1 バッチ更新の概要
        - 21.1.2 標準バッチ更新
            - 21.1.2.1 標準バッチ処理のOracle実装の制限事項
            - 21.1.2.2 バッチに対する操作の追加について
            - 21.1.2.3 バッチの処理について
            - 21.1.2.4 配列DMLの反復ごとの行数
            - 21.1.2.5 標準バッチ処理のOracle実装による変更のコミットについて
            - 21.1.2.6 バッチのクリアについて
            - 21.1.2.7 標準バッチ処理のOracle実装の更新件数
            - 21.1.2.8 標準バッチ処理のOracle実装におけるエラー処理
            - 21.1.2.9 バッチ処理される文とバッチ処理されない文の混在について
        - 21.1.3 早期バッチ・フラッシュ
    - 21.2 その他のOracleパフォーマンス拡張機能
        - 21.2.1 LOBデータのプリフェッチについて
        - 21.2.2 Oracle行プリフェッチの制限事項
        - 21.2.3 列型の定義について
        - 21.2.4 DatabaseMetaData TABLE_REMARKSのレポートについて
- 22 OCI接続プーリング
    - 22.1 OCIドライバ接続プーリングの背景
    - 22.2 OCIドライバ接続プーリングと共有サーバーの比較
    - 22.3 OCI接続プールの定義について
        - 22.3.1 OCI接続プールの作成の概要
        - 22.3.2 oracle.jdbc.poolパッケージおよびoracle.jdbc.ociパッケージのインポート
        - 22.3.3 OCI接続プールの作成
        - 22.3.4 OCI接続プール・パラメータの設定
        - 22.3.5 OCI接続プール・ステータスのチェック

        - 22.4 OCI接続プールへの接続について
        - 22.5 OCI接続プーリングのサンプル・コード
        - 22.6 文の処理とキャッシュ
        - 22.7 JNDIおよびOCI接続プール
    - 23 データベース常駐の接続プール
        - 23.1 データベース常駐接続プーリングの概要
        - 23.2 データベース常駐接続プーリングを使用可能にする方法
            - 23.2.1 サーバー側でDRCPを使用可能にする方法
            - 23.2.2 クライアント側でDRCPを使用可能にする方法
        - 23.3 複数の接続プール間でのプールされたサーバーの共有について
        - 23.4 DRCPのタグ付け
        - 23.5 セッション状態の修正のためのPL/SQLコールバック
        - 23.6 DRCPを使用するためのAPI
    - 24 データベース・シャーディングのJDBCによるサポート
        - 24.1 JDBCユーザー用のデータベース・シャーディングの概要
        - 24.2 シャーディング・キーの作成について
        - 24.3 データベース・シャーディングのサポート用API
            - 24.3.1 OracleShardingKeyインタフェース
            - 24.3.2 OracleShardingKeyBuilderインタフェース
            - 24.3.3 OracleConnectionBuilderインタフェース
            - 24.3.4 データベース・シャーディングのサポートのための他の新規クラスおよびメソッド
        - 24.4 JDBCシャーディングの例
    - 25 Oracleアドバンスト・キューイング
        - 25.1 Oracleアドバンスト・キューイングの機能とフレームワーク
        - 25.2 データベースの変更
        - 25.3 AQ非同期イベント通知
        - 25.4 メッセージの作成について
            - 25.4.1 メッセージの作成
            - 25.4.2 AQメッセージのプロパティ
            - 25.4.3 AQメッセージのペイロード
        - 25.5 例: メッセージの作成およびペイロードの設定
        - 25.6 メッセージのエンキュー
        - 25.7 メッセージのデキュー
        - 25.8 例: エンキューとデキュー
    - 26 連続問合せ通知
        - 26.1 連続問合せ通知の概要
        - 26.2 クライアント開始の連続問合せ通知の概要
        - 26.3 登録エントリの作成
            - 26.3.1 連続問合せ通知登録オプション
        - 26.4 登録エントリへの問合せの関連付け
        - 26.5 データベース変更イベントの通知
        - 26.6 登録エントリの削除
- 第VI部 高可用性

- 27 Java用のトランザクション・ガード
  - 27.1 Java用のトランザクション・ガードの概要
  - 27.2 XAトランザクションのためのトランザクション・ガードのサポート
  - 27.3 XAによるトランザクション・ガードの使用方法
  - 27.4 Java API用のトランザクション・ガード
    - 27.4.1 論理トランザクションIDの取得
    - 27.4.2 更新済論理トランザクションIDの取得
      - 27.4.2.1 イベント・リスナーの登録
      - 27.4.2.2 イベント・リスナーの登録解除
  - 27.5 完全な例: トランザクション・ガードAPIの使用
  - 27.6 サーバー側トランザクション・ガードAPIの使用について
- 28 Java用のアプリケーション・コンティニュイティ
  - 28.1 Java用のアプリケーション・コンティニュイティに対するOracle JDBCの構成について
    - 28.1.1 アプリケーション・コンティニュイティでの具象クラスのサポート
  - 28.2 Java用のアプリケーション・コンティニュイティに対するOracle Databaseの構成について
  - 28.3 アプリケーション・コンティニュイティにおけるXAデータソースのサポート
  - 28.4 Java用のアプリケーション・コンティニュイティにおけるリクエスト境界の識別について
  - 28.5 透過的アプリケーション・コンティニュイティのサポート
  - 28.6 アプリケーション・コンティニュイティのリプレイ前の初期状態の確立
    - 28.6.1 コールバックなし
    - 28.6.2 接続ラベリング
    - 28.6.3 接続初期化コールバック
      - 28.6.3.1 初期化コールバックの作成
      - 28.6.3.2 初期化コールバックの登録
      - 28.6.3.3 初期化コールバックの削除または登録解除
    - 28.6.4 FAILOVER_RESTOREの有効化について
  - 28.7 Java用のアプリケーション・コンティニュイティにおける再接続の遅延について
    - 28.7.1 Java用のアプリケーション・コンティニュイティに関する構成例
      - 28.7.1.1 Oracle RACでのサービスの作成
      - 28.7.1.2 単一インスタンス・データベースにおけるサービスの変更
  - 28.8 Java用のアプリケーション・コンティニュイティにおける可変値の保持について
    - 28.8.1 インタフェースの付与または取消し
      - 28.8.1.1 DateおよびSYS_GUID構文
      - 28.8.1.2 Sequence構文
      - 28.8.1.3 GRANT ALL文
      - 28.8.1.4 可変値における付与のルール
  - 28.9 アプリケーション・コンティニュイティの統計
  - 28.10 Java用のアプリケーション・コンティニュイティにおけるリプレイの無効化について
    - 28.10.1 リプレイを無効にする方法
    - 28.10.2 リプレイを無効にするタイミング
      - 28.10.2.1 繰り返さない外部システムをコールするアプリケーション
      - 28.10.2.2 アプリケーションが独立したセッションを同期する場合
      - 28.10.2.3 アプリケーションが実行ロジックの中間層で時間を使用する場合

- 28.10.2.4 アプリケーションでROWIDが変更されないことが想定される場合
- 28.10.2.5 アプリケーションで悪影響が一度発生することが想定される場合
- 28.10.2.6 アプリケーションでロケーション値が変更されないことが想定される場合
- 28.10.3 診断とトレース
  - 28.10.3.1 リプレイ・トレースのコンソールへの書込み
  - 28.10.3.2 リプレイ・トレースのファイルへの書込み
- 29 Oracle JDBCによるFANイベントのサポート
  - 29.1 Oracle JDBCによるFANイベントのサポートの概要
  - 29.2 計画済メンテナンスの安全なドレイニングAPI
  - 29.3 FANイベントのサポートのためのOracle JDBCドライバのインストールおよび構成
  - 29.4 計画済メンテナンスの場合のOracle JDBCドライバによるFANのサポートの例
- 30 透過的アプリケーション・フェイルオーバー
  - 30.1 透過的アプリケーション・フェイルオーバーの概要
  - 30.2 フェイルオーバー・タイプ・イベント
  - 30.3 TAFコールバック
  - 30.4 Java TAFコールバック・インタフェース
  - 30.5 TAFと高速接続フェイルオーバーの比較
- 31 単一クライアント・アクセス名
  - 31.1 単一クライアント・アクセス名の概要
  - 31.2 SCANを使用したデータベースの構成について
  - 31.3 SCANを使用した接続ロード・バランシングの動作
  - 31.4 バージョンと下位互換性
  - 31.5 最大可用性アーキテクチャ環境でのSCANの使用
  - 31.6 Oracle Connection ManagerでのSCANの使用
- 第VII部 トランザクション管理
  - 32 分散トランザクション
    - 32.1 分散トランザクションについて
      - 32.1.1 分散トランザクションの概要
      - 32.1.2 分散トランザクションのコンポーネントおよびシナリオ
      - 32.1.3 分散トランザクションの概念
      - 32.1.4 ローカル・トランザクションとグローバル・トランザクションの切替えについて
      - 32.1.5 Oracle XAパッケージ
    - 32.2 XAコンポーネント
      - 32.2.1 XADatasourceインタフェースとOracle実装
      - 32.2.2 XAConnectionインタフェースとOracle実装
      - 32.2.3 XAResourceインタフェースとOracle実装
      - 32.2.4 OracleXAResourceメソッドの機能と入力パラメータ
      - 32.2.5 Xidインタフェースと Oracle実装
    - 32.3 エラー処理と最適化
      - 32.3.1 XAExceptionクラスとメソッド
      - 32.3.2 OracleエラーとXAエラーのマッピング
      - 32.3.3 XAエラー処理

        - 32.3.4 Oracle XA最適化
    - 32.4 分散トランザクションの実装について
        - 32.4.1 Oracle XAのインポートの概要
        - 32.4.2 OracleのXAコード・サンプル
    - 32.5 Oracle JDBCドライバのネイティブXA
        - 32.5.1 OCIネイティブXA
        - 32.5.2 ThinネイティブXA
- 第VIII部 管理性
    - 33 データベース管理
        - 33.1 データベース管理メソッドの使用
        - 33.2 startupメソッドの使用
            - 33.2.1 データベース起動オプション
        - 33.3 shutdownメソッドの使用
            - 33.3.1 データベース停止オプション
            - 33.3.2 標準的なデータベース停止プロセス
        - 33.4 完全な例
    - 34 JDBCの診断機能
        - 34.1 Oracle JDBCドライバのロギング機能について
            - 34.1.1 Oracle JDBCドライバのロギング機能の概要
            - 34.1.2 JDBCロギングの有効化と使用
                - 34.1.2.1 CLASSPATHの構成について
                - 34.1.2.2 ロギングの有効化
                - 34.1.2.3 ロギングの構成
                - 34.1.2.4 ログ出力のファイルへのリダイレクト
                - 34.1.2.5 ログ出力の使用
                - 34.1.2.6 ロギングの例
            - 34.1.3 実行時における機能固有のロギングの有効化または無効化
            - 34.1.4 機能固有のロギング用のロギング構成ファイルの使用
            - 34.1.5 パフォーマンス、スケーラビリティおよびセキュリティに関する問題点
        - 34.2 診断能力管理
    - 35 JDBC DMSメトリック
        - 35.1 JDBC DMSメトリックの概要
        - 35.2 生成されるメトリックの種類の決定について
        - 35.3 SQLTextメトリックの生成について
        - 35.4 JMXを使用したDMSメトリックへのアクセスについて
- 付録
    - A JDBCリファレンス情報
        - A.1 サポートされているSQLとJDBCデータ型のマッピング
        - A.2 サポートされているSQLおよびPL/SQLデータ型
        - A.3 PL/SQLタイプの使用について
        - A.4 埋込みJDBCエスケープ構文の使用
            - A.4.1 時刻および日付リテラル
                - A.4.1.1 日付リテラル

- A.4.1.2 時刻リテラル
- A.4.1.3 タイムスタンプ・リテラル
- A.4.2 スカラー関数
- A.4.3 LIKEエスケープ文字
- A.4.4 MATCH_RECOGNIZE句
- A.4.5 外部結合
- A.4.6 ファンクション・コール構文
- A.4.7 JDBCエスケープ構文からOracle SQL構文変換例
- A.5 Oracle JDBCのノートおよび制限事項
- A.5.1 CursorName
- A.5.2 JDBC外部結合エスケープ
- A.5.3 IEEE 754浮動小数点との互換性
- A.5.4 DatabaseMetaDataコールへのCatalog引数
- A.5.5 SQLWarningクラス
- A.5.6 DDL文の実行
- A.5.7 名前付きパラメータのバインド
- B Oracle RAC高速アプリケーション通知
- B.1 Oracle RAC高速アプリケーション通知の概要
- B.2 Oracle RAC高速アプリケーション通知のインストールおよび構成
- B.3 Oracle RAC高速アプリケーション通知の使用
- B.4 接続プールの実装
- C JDBCコーディングのヒント
- C.1 JDBCとマルチスレッド
- C.2 JDBCプログラムのパフォーマンスの最適化
- C.2.1 自動コミット・モードの無効化
- C.2.2 標準フェッチ・サイズとOracle行プリフェッチ
- C.2.3 セッション・データ・ユニット・サイズの設定について
- C.2.3.1 データベース・サーバーのSDUサイズの設定について
- C.2.3.2 JDBC OCIクライアントのSDUサイズの設定について
- C.2.3.3 JDBC ThinクライアントのSDUサイズの設定について
- C.2.4 JDBCバッチ更新機能
- C.2.5 文キャッシュ
- C.2.6 組込みSQL型とJava型間のマッピング
- C.3 JDBCのトランザクション分離レベルとアクセス・モード
- D JDBCエラー・メッセージ
- D.1 JDBCエラー・メッセージの一般構造
- D.2 一般JDBCメッセージ
- D.2.1 ORA番号でソートしたJDBCメッセージ
- D.2.2 五十音順にソートしたJDBCメッセージ
- D.3 ネイティブXAメッセージ
- D.3.1 ORA番号でソートしたネイティブXAメッセージ
- D.3.2 五十音順にソートしたネイティブXAメッセージ
- D.4 TTCメッセージ

- D.4.1 ORA番号でソートしたTTCメッセージ
- D.4.2 五十音順にソートしたTTCメッセージ
- E トラブルシューティング
  - E.1 一般的な問題
    - E.1.1 OUTまたはIN/OUT変数として定義されたCHAR列に対するメモリー消費
    - E.1.2 メモリー・リークおよびカーソルの不足
    - E.1.3 1プロセスで17以上のOCI接続のオープン数について
    - E.1.4 Statement.cancelの使用
    - E.1.5 ファイアウォールとJDBCの使用
    - E.1.6 多数回にわたるサーバーからの突然の切断
    - E.1.7 ネットワーク・アダプタで接続を確立できない
      - E.1.7.1 MTSサーバーで構成したOracleインスタンスが共有サーバーを使用
      - E.1.7.2 IPv4とIPv6の両方をサポートするNICカードを保有するJDBC Thinドライバ
      - E.1.7.3 サンプル・アプリケーション
  - E.2 基本的なデバッグ処理
    - E.2.1 ネットワーク・イベントをトラップするためのOracle Netトレース
      - E.2.1.1 クライアント側でのトレース
        - E.2.1.1.1 TRACE_LEVEL_CLIENT
        - E.2.1.1.2 TRACE_DIRECTORY_CLIENT
        - E.2.1.1.3 TRACE_FILE_CLIENT
        - E.2.1.1.4 TRACE_UNIQUE_CLIENT
      - E.2.1.2 サーバー側でのトレース
        - E.2.1.2.1 TRACE_LEVEL_SERVER
        - E.2.1.2.2 TRACE_DIRECTORY_SERVER
        - E.2.1.2.3 TRACE_FILE_SERVER
    - E.2.2 サード・パーティのデバッグ・ツール
- 索引

表一覧

- 1-1 JDBC OCIドライバとJDBC Thinドライバの機能の相違点
- 1-2 機能リスト
- 2-1 JDBCドライバ用のimport文
- 2-2 自動コミット・モードがオンの場合に実行された操作に対するエラー・メッセージ
- 3-1 JDBC 3.0機能の主な領域
- 3-2 等価のBLOBメソッド
- 3-3 等価のCLOBメソッド
- 4-1 oracle.jdbcパッケージの主なインタフェースおよびクラス
- 6-1 OCI Instant Client共有ライブラリ
- 6-2 Instant ClientおよびInstant Client Light(English)のデータ共有ライブラリ
- 8-1 標準データソース・プロパティ
- 8-2 Oracle拡張データソース・プロパティ
- 8-3 サポートされるデータベース指定子
- 9-1 暗号化または整合性のためのクライアント/サーバー間ネゴシエーション表
- 9-2 暗号化と整合性のためのOCIドライバ・クライアント・パラメータ
- 9-3 暗号化と整合性のためのThinドライバ・クライアント・パラメータ
- 11-1 SQL型とJava型間のデフォルト・マッピング
- 11-2 getObjectおよびgetOracleObjectの戻り型
- 12-1 LONGおよびLONG RAWデータの変換
- 17-1 Oracle JDBCでの内部変更および外部変更の可視性
- 18-1 JDBCのRowSetとCachedRowSetの比較
- 20-1 文キャッシュで使用されるメソッドの比較
- 20-2 文の割当ておよび暗黙的文キャッシュで使用されるメソッド
- 20-3 明示的にキャッシュされた文を取り出す場合に使用するメソッド
- 21-1 有効な列型
- 26-1 連続問合せ通知登録オプション
- 31-1 OracleクライアントおよびOracle DatabaseのSCANに関するバージョン互換性
- 32-1 接続モードの推移
- 32-2 OracleとXAのエラー・マッピング
- 33-1 サポートされているデータベース起動オプション
- 33-2 サポートされているデータベース停止オプション
- A-1 有効なSQLデータ型-Javaクラス・マッピング
- A-2 SQLデータ型に対するサポート
- A-3 ANSI-92 SQLデータ型に対するサポート
- A-4 SQLユーザー定義型に対するサポート
- A-5 PL/SQLデータ型に対するサポート
- C-1 SQLデータ型とSQLデータ型を表すJavaクラスのマッピング
- D-1 ORA番号でソートしたJDBCメッセージ
- D-2 五十音順にソートしたJDBCメッセージ
- D-3 ORA番号でソートしたネイティブXAメッセージ
- D-4 五十音順にソートしたネイティブXAメッセージ

- D-5　　　ORA番号でソートしたTTCメッセージ
- D-6　　　五十音順にソートしたTTCメッセージ

はじめに

この章では、『Oracle Database JDBC開発者ガイド』の概要、対象読者、構成および表記規則について説明します。関連するOracleドキュメントのリストも提供します。

対象読者

Oracle Database JDBC開発者ガイドは、Java Database Connectivity (JDBC)に基づくアプリケーションの開発者を対象としています。このマニュアルは、JDBCプログラミングに関心のあるすべての読者を対象にしていますが、次の事項に関するいくらかの予備知識があることを前提にしています。

- Java
- Oracle PL/SQL
- Oracle Database

ドキュメントのアクセシビリティ

OracleのアクセシビリティについてのOracle詳細情報は、Oracle Accessibility ProgramのWebサイト(http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc)を参照してください。

Oracleサポートへのアクセス

サポートを購入したオラクル社のお客様は、My Oracle Supportを介して電子的なサポートにアクセスできます。詳細情報は(http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info)か、聴覚に障害のあるお客様は(http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs)を参照してください。

関連ドキュメント

次のマニュアルは、Oracle Java Platformグループから利用できます。

- Oracle Database Java開発者ガイド

  このマニュアルでは、Javaの基本概念を導入し、サーバー側の構成と機能に関する一般的な情報を提供します。特定の製品(たとえばJDBC)よりむしろOracle Javaプラットフォーム全体に関係している情報はこのマニュアルに記載されています。このマニュアルは、以前は独立したマニュアルで説明されていた、Javaストアド・プロシージャについても説明しています。

- Oracle Database SQLJ開発者ガイド

  このマニュアルでは、Javaコードに直接静的SQL操作を埋め込むためのSQLJの使用方法や、SQLJ言語の構文およびSQLJトランスレータのオプションと機能について説明しています。標準SQLJの機能とOracle固有SQLJ機能の両方について説明しています。

次のドキュメントは、Oracle Server Technologiesグループから利用できます。

- Oracle Database開発ガイド

- Oracle Database PL/SQLパッケージ・プロシージャおよびタイプ・リファレンス
- Oracle Database PL/SQL言語リファレンス
- Oracle Database SQL言語リファレンス

リリース・ノート、インストール関連ドキュメント、ホワイト・ペーパーまたはその他の関連ドキュメントは、OTN (Oracle Technology Network)から、無償でダウンロードできます。

すでにOTNのユーザー名およびパスワードを取得している場合は、次の場所でOTN Webサイトのドキュメントのセクションに直接接続できます。

http://www.oracle.com/technetwork/documentation/index.html

次のリソースを使用できます。

- 最新の仕様などがある、JDBCに関するWebサイト。

  http://www.oracle.com/technetwork/java/javase/jdbc/index.htm

表記規則

このセクションでは、このマニュアルの本文およびコード例で使用される表記規則について説明します。この項の内容は次のとおりです。

- 本文の表記規則
- コード例の表記規則
- Windowsオペレーティング・システム環境での表記規則

本文の表記規則

本文では、特定の項目が一目でわかるように、次の表記規則を使用します。その規則と使用例を示します。

規則	意味	例
太字	太字は、本文中で定義されている用語または用語集に記載されている用語(あるいはその両方)を示します。	この句を指定すると、索引構成表が作成されます。
イタリック	イタリックは、ドキュメントのタイトルまたは強調を示します。	Oracle Database 概要 リカバリ・カタログおよびターゲット・データベースは、同一のディスクには存在しないことを確認してください。
固定幅フォントの大文字	固定幅フォントの大文字は、システム指定の要素を示します。このような要素には、パラメータ、権限、データ型、RMAN キーワード、SQL キーワード、SQL*Plus またはユーティリティ・コマンド、パッケージおよびメソッドがあります。また、シ	NUMBER 列に対してのみ、この句を指定できます。 BACKUP コマンドを使用して、データベースのバックアップを作成できます。 USER_TABLES データ・ディクショナリ・ビュー内の

規則	意味	例
	ステム指定の列名、データベース・オブジェクト、データベース構造、ユーザー名およびロールも含まれます。	TABLE_NAME 列を問い合せます。 DBMS_STATS.GENERATE_STATS プロシージャを使用します。
固定幅フォントの小文字	固定幅フォントの小文字は、実行可能ファイル、ファイル名、ディレクトリ名およびユーザーが指定する要素のサンプルを示します。このような要素には、コンピュータ名およびデータベース名、ネット・サービス名および接続識別子があります。また、ユーザーが指定するデータベース・オブジェクトとデータベース構造、列名、パッケージとクラス、ユーザー名とロール、プログラム・ユニットおよびパラメータ値も含まれます。 ノート: プログラム要素には、大文字と小文字を組み合せて使用するものもあります。これらの要素は、記載されているとおりに入力してください。	sqlplus と入力して、SQL*Plus をオープンします。 パスワードは、orapwd ファイルで指定します。 /disk1/oracle/dbs ディレクトリ内のデータ・ファイルおよび制御ファイルのバックアップを作成します。 hr.departments 表には、department_id、department_name および location_id 列があります。 QUERY_REWRITE_ENABLED 初期化パラメータを true に設定します。 oe ユーザーとして接続します。 JRepUtil クラスが次のメソッドを実装します。
固定幅フォントの小文字のイタリック	固定幅フォントの小文字のイタリックは、プレースホルダまたは変数を示します。	parallel_clause を指定できます。 old_release.SQL を実行します。ここで、old_release とはアップグレード前にインストールしたリリースを示します。

コード例の表記規則

コード例では、Javaの文、SQL文およびコマンドライン文を示しています。例は次のように固定幅フォントで表示され、通常のテキストと区別されます。

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

次の表に、コード例で使用される表記規則とその使用例を示します。

規則	意味	例
[ ]	大カッコは、カッコ内の項目を任意に選択することを表します。大カッコは、入力しないでください。	DECIMAL (digits [, precision])

規則	意味	例
{ }	中カッコは、カッコ内の項目のうち、1 つが必須であることを表します。中カッコは、入力しないでください。	{ENABLE | DISABLE}
|	縦線は、大カッコまたは中カッコ内の複数の選択項目の区切りに使用します。項目のうちの 1 つを入力します。縦線は、入力しないでください。	{ENABLE | DISABLE} [COMPRESS | NOCOMPRESS]
...	水平の省略記号は、次のいずれかを示します。 ● 例に直接関連しないコードの一部が省略されている。 ● コードの一部を繰り返すことができる。	CREATE TABLE ... AS subquery; SELECT col1, col2, ... , coln FROM employees;
. . .	垂直の省略記号は、例に直接関連しない複数の行が省略されていることを示します。	SQL> SELECT NAME FROM V$DATAFILE; NAME ------------------------------------ /fsl/dbs/tbs_01.dbf /fs1/dbs/tbs_02.dbf . . . /fsl/dbs/tbs_09.dbf 9 rows selected.
その他の記号	大カッコ、中カッコ、縦線および省略記号以外の記号は、記載されているとおりに入力する必要があります。	acctbal NUMBER(11,2); acct CONSTANT NUMBER(4) := 3;
Italics	イタリック体は、特定の値を指定する必要があるプレースホルダや変数を示します。	CONNECT SYSTEM/system_password DB_NAME = database_name
UPPERCASE	大文字は、システム指定の要素を示します。これらの要素は、ユーザー定義の要素と区別するために大文字で示されます。大カッコ内にないかぎり、表示されているとおりの順序および綴りで入力します。ただし、大/小文字が区別されないため、小文字でも入力できます。	SELECT last_name, employee_id FROM employees; SELECT * FROM USER_TABLES; DROP TABLE hr.employees;
lowercase	小文字は、ユーザー指定のプログラム要素を示します。たとえば、表名、列名	SELECT last_name, employee_id FROM employees; sqlplus HR/hr CREATE USER mjones IDENTIFIED BY

規則	意味	例
	またはファイル名などです。 ノート: プログラム要素には、大文字と小文字を組み合せて使用するものもあります。これらの要素は、記載されているとおりに入力してください。	ty3MU9;

Windowsオペレーティング・システム環境での表記規則

次の表に、Windowsオペレーティング・システム環境での表記規則とその使用例を示します。

規則	意味	例
「スタート」>を選択	プログラムを起動する方法を示します。	Database Configuration Assistant を起動するには、「スタート」→「すべてのプログラム」→「Oracle - HOME_NAME」→「Configuration and Migration Tools」→「Database Configuration Assistant」の順に選択します。
ファイル名およびディレクトリ名	ファイル名およびディレクトリ名は大/小文字が区別されません。特殊文字の左山カッコ(<)、右山カッコ(>)、コロン(:)、二重引用符(")、スラッシュ(/)、縦線(|)およびハイフン(-)は使用できません。特殊文字の円記号(¥)は、引用符で囲まれている場合でも、要素のセパレータとして処理されます。Windows では、ファイル名が¥¥で始まる場合、汎用命名規則が使用されていると解釈されます。	`c:¥winnt"¥"system32 is the same as C:¥WINNT¥SYSTEM32`
`C:¥>`	現在のハード・ディスク・ドライブの Windows コマンド・プロンプトを表します。コマンド・プロンプトのエスケープ文字はカレット(^)です。プロンプトは作業中のサブディレクトリを表します。このマニュアルでは、コマンド・プロンプトと呼びます。	`C:¥oracle¥oradata>`
特殊文字	Windows コマンド・プロンプトで二重引用符(")のエスケープ文字として円記号(¥)が必要な場合があります。丸カッコおよび一重引用符(')にはエスケープ文字は必要ありません。エスケープ文字および特殊文字の詳細は、Windows オペレーティング・システムのドキュメ	`C:¥>exp HR/hr TABLES=employees QUERY=¥"WHERE job_id='SALESMAN' and salary<1600¥"` `C:¥>imp SYSTEM/password FROM USER=HR TABLES=(employees, dept)`

規則	意味	例
	ントを参照してください。	
HOME_NAME	Oracle ホームの名前を表します。ホーム名には、英数字で 16 文字まで使用できます。ホーム名に使用可能な特殊文字は、アンダースコアのみです。	C:¥> net start OracleHOME_NAMETNSListener
ORACLE_HOME および ORACLE_BASE	Oracle8i リリース 8.1.3 より前のリリースでは、Oracle コンポーネントをインストールすると、すべてのサブディレクトリが最上位の ORACLE_HOME の直下に置かれました。ディレクトリの名前は、デフォルトでは次のいずれかです。 ● C:¥orant(Windows NT の場合) ● C:¥orawin98(Windows 98 の場合) このリリースは、Optimal Flexible Architecture (OFA)のガイドラインに準拠しています。ORACLE_HOME ディレクトリ下に配置されないサブディレクトリもあります。最上位のディレクトリは ORACLE_BASE と呼ばれ、デフォルトでは C:¥oracle¥です。他の Oracle ソフトウェアがインストールされていないコンピュータに最新リリースの Oracle をインストールした場合、Oracle ホーム・ディレクトリはデフォルトで C:¥oracle¥orann に設定されます。nn は最新リリースの番号です。Oracle ホーム・ディレクトリは、ORACLE_BASE の直下に配置されます。 このマニュアルに示すディレクトリ・パスの例は、すべて OFA の表記規則に準拠しています。 OFA の準拠の詳細、および非 OFA 準拠ディレクトリ下での Oracle 製品のインストールの詳細は、『Oracle Database プラットフォーム・ガイド for Microsoft Windows』を参照してください。	ORACLE_BASE¥ORACLE_HOME¥rdbms¥admin ディレクトリに移動します。

『Oracle Database JDBC開発者ガイド』の今回のリリースにおける変更点

内容は次のとおりです。

- Changes in Oracle Database 19c

Changes in Oracle Database 19c

Oracle Database 19c向けのOracle Database JDBC開発者ガイドの変更点は次のとおりです。

新機能

このリリースの新機能は次のとおりです。

- アプリケーション・コンティニュイティの拡張

  透過的アプリケーション・コンティニュイティのサポートを参照してください

- Easy Connect Plusのサポート

  「Easy Connect Plusのサポート」を参照してください

- トークンベース認証のサポート

  「JDBCクライアント側セキュリティ機能」を参照してください

非推奨となった機能

次の機能は非推奨であり、将来のリリースではサポートされなくなる可能性があります。

- 連想配列の拡張サポートの一環として、`OraclePreparedStatement`および`OracleCallableStatement`クラスから次のAPIが非推奨になりました。
  - `setPlsqlIndexTable`
  - `setPlsqlIndexTableAtName`
  - `registerIndexTableOutParameter`
  - `getOraclePlsqlIndexTable`
  - `getPlsqlIndexTable`
- `oracle.sql`パッケージの具象クラス

  `oracle.sql`パッケージの具象クラスは非推奨です。これらのクラスのかわりに、新しいJDBCインタフェースを使用します。

  これらのインタフェースの詳細は、MoSノート1364193.1を参照してください。

- `oracle.jdbc.rowset`パッケージは非推奨になりました。この機能にかわる標準のJDBC RowSetパッケージを使用することをお薦めします。

関連項目:

http://docs.oracle.com/javase/9/docs/api/javax/sql/rowset/package-summary.html

- defineColumnTypeメソッド

  defineColumnTypeメソッドのほとんどの変数は非推奨です。次の項目の変数がサポートされています。

  - LOBからLONGへの変換
  - LOBプリフェッチ・サイズの構成

  詳細は、JDBC Javadocを参照してください。

- CONNECTION_PROPERTY_STREAM_CHUNK_SIZEプロパティ

  詳細は、JDBC Javadocを参照してください。

- Oracleバッチ更新

  Oracleバッチ更新は、Oracle Database 12c リリース1 (12.1)で非推奨となりました。Oracle Database 12c リリース2 (12.2)以降、Oracleバッチ更新はオペレーション・コードなし(no-op)になりました。つまり、Oracle Database 19cのJDBCドライバを使用してアプリケーションでOracleバッチ更新を実装すると、指定したバッチ・サイズが設定されず、バッチ・サイズが1になるということです。バッチがこの設定の場合、アプリケーションは一度に1行ずつを処理します。Oracle Database 19cのJDBCドライバを使用する場合は、標準のJDBCバッチを使用することをお薦めします。

  詳細は、「標準バッチ更新」の項を参照してください。

- EndToEndMetrics関連のAPI

  EndToEndMetrics関連のAPIは、Oracle Database 12cリリース2 (12.2)以降では非推奨です。

  詳細は、「JDBC DMSメトリック」を参照してください。

その他の変更

ネイティブ暗号化のセキュリティ更新

Oracleでは、Oracle Databaseリリース11.2以降における、ネイティブ・ネットワーク暗号化環境に影響を与える必要なセキュリティ機能強化に対応するために、ダウンロード可能なパッチを提供しています。このパッチは、My Oracle Supportノート2118136.2で入手できます。

関連項目:

詳細は、『Oracle Databaseセキュリティ・ガイド』を参照してください

第I部 概要

この部では、Java Database Connectivity(JDBC)の概念を紹介し、JDBCのOracle実装の概要を説明します。JDBCドライバに関して、Oracleクライアントのインストールと構成の基本情報を提供します。また、JDBCアプリケーションの作成および実行の基本ステップについても説明します。

第I部は、次の章で構成されています。

- JDBCの概要
- スタート・ガイド

1 JDBCの概要

Java Database Connectivity(JDBC)は、Javaからリレーショナル・データベースに接続するためのインタフェースを提供するJava標準です。JDBC標準は定義済で、標準のjava.sqlインタフェースを介して実装されます。このため、各プロバイダは独自のJDBCドライバで標準を実装および拡張できます。JDBCは、X/Open SQLコール・レベル・インタフェース(CLI)に基づいています。JDBC 4.0では、SQL 2003標準を使用してコンパイルします。

この章では、JDBCのOracle実装の概要を説明します。内容は次のとおりです。

- Oracle JDBCドライバの概要
- 適切なドライバの選択
- JDBC OCIドライバとJDBC Thinドライバの機能の相違点
- 環境およびサポート
- 機能リスト

1.1 Oracle JDBCドライバの概要

Oracleドライバは、標準JDBC Application Program Interface(API)をサポートする他、Oracle固有のデータ型をサポートし、パフォーマンスを向上させる拡張機能を備えています。

Oracle JDBCドライバを次に示します。

- Thinドライバ

  JDBC Thinドライバは、Pure JavaのType IVドライバで、アプリケーションで使用できます。プラットフォームに依存せず、クライアント側に追加のOracleソフトウェアは必要ありません。JDBC Thinドライバは、Oracle Databaseにアクセスするために、Oracle Net Servicesを使用してサーバーと通信します。

  JDBC Thinドライバを使用すると、Javaソケット上にOracle Net Servicesを実装することで、データベースに直接接続できます。このドライバでは、TCP/IPプロトコルがサポートされており、データベース・サーバーのTCP/IPソケットにTNSリスナーが必要です。

ノート:

特定のドライバによってのみサポートされる機能がないかぎり、Thin ドライバを使用することをお薦めします。

- Oracle Call Interface(OCI)ドライバ

  クライアント側で使用されるドライバで、Oracleクライアントのインストールが必要です。アプリケーションでのみ使用できます。

  JDBC OCIドライバは、Javaアプリケーション用のType IIドライバです。これには、プラットフォーム固有のOCIライブラリが必要です。IPC(プロセス間通信)、名前付きパイプ、TCP/IPおよびInternetwork Packet Exchange/Sequenced Packet Exchange(IPX/SPX)を含む、インストールされたすべてのOracle Netアダプ

タをサポートします。

JDBC OCIドライバはJavaとCを組み合せて記述されており、Cエントリ・ポイントをコールするネイティブ・メソッドを使用してJDBCの起動をOCIへのコールに変換します。これらのコールでは、Oracle Net Servicesを使用してデータベースと通信します。

JDBC OCIドライバは、インストールされたクライアント・コンピュータにあるOCIライブラリ、Cエントリ・ポイント、Oracle Net、コア・ライブラリおよびその他の必要なファイルを使用します。

OCIは、第3世代言語のネイティブ・プロシージャやファンクション・コールを使用して、Oracle Databaseにアクセスし、SQL文処理のすべての段階を制御するアプリケーションを作成できるようにするAPIです。

- サーバー側Thinドライバ

  機能的にはクライアント側Thinドライバと同じです。ただし、データベース・サーバーで実行され、同じサーバーまたは任意の層のリモート・サーバーで別セッションにアクセスする必要があるコード用です。

  JDBCサーバー側Thinドライバは、クライアント側で実行されるJDBC Thinドライバと同じ機能を提供します。ただし、JDBCサーバー側ThinドライバはOracle Databaseの内部で実行され、リモート・データベース、または同じデータベースの別セッションにアクセスし、データベース内のJavaと連携します。

  このドライバは、次の場合に役立ちます。

  - 中間層として機能を果すOracle Databaseインスタンスからリモートのデータベース・サーバーにアクセスする場合
  - Javaストアド・プロシージャなどのOracle Databaseセッションの内部から別のOracle Databaseセッションにアクセスする場合

  JDBC Thinドライバをクライアント・アプリケーションから使用する場合とサーバーの内部から使用する場合で、コード上の違いはありません。

- サーバー側内部ドライバ

  データベース・サーバーで実行され、同じセッションにアクセスするコード用です。つまり、このコードは、単一のOracleセッションから実行され、データにアクセスします。

  JDBCサーバー側内部ドライバは、Javaストアド・プロシージャなど、Oracle Databaseの内部で実行され、同じデータベースにアクセスする、あらゆるJavaコードをサポートします。このドライバにより、Oracle Java仮想マシン(Oracle JVM)はSQLエンジンと直接通信し、データベース内のJavaと連携します。

  JDBCサーバー側の内部ドライバ、Oracle JVM、データベースおよびSQLエンジンは、すべて同じアドレス空間の中で実行されます。このため、ネットワーク・ラウンドトリップの問題は関係ありません。プログラムは、関数呼出しを使用してSQLエンジンにアクセスします。

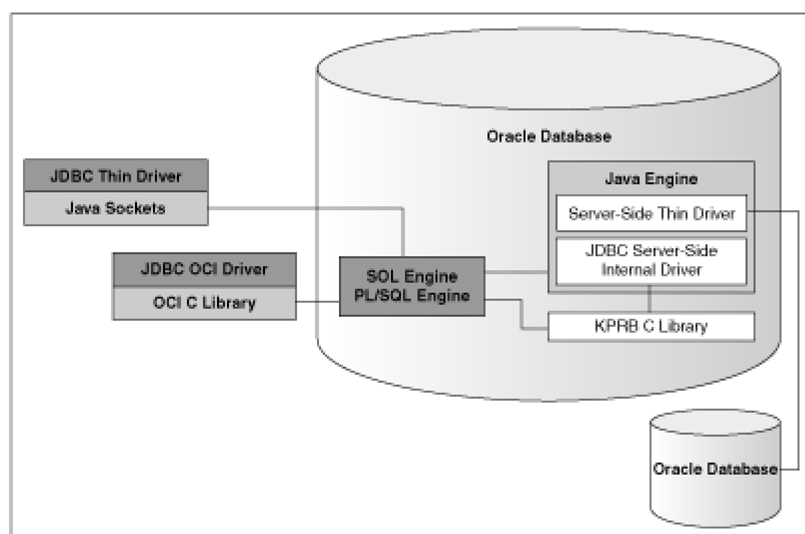
ノート:

サーバー側内部ドライバでは、`Statement` クラスの `cancel` および `setQueryTimeout` メソッドはサポートされません。

JDBCサーバー側内部ドライバは、クライアント側ドライバとの一貫性を保ち、同一の機能と拡張機能をサポートします。

次の表は、Oracle JDBCドライバとOracle Databaseのアーキテクチャを示しています。

図1-1 Oracle JDBCドライバとOracle Databaseのアーキテクチャ



関連トピック

- JDBC Thin固有の機能
- JDBC OCIドライバ固有の機能
- サーバー側内部ドライバ

1.2 適切なドライバの選択

アプリケーションまたはアプレット用のJDBCドライバを選択するときには、次の点を考慮します。

- 一般的に、TCP/IP以外のネットワークのサポートなどのOCI固有の機能が必要でないかぎり、JDBC Thinドライバを使用します。
- 最大限の移植性とパフォーマンスを実現するには、JDBC Thinドライバを使用します。JDBC Thinドライバを使用すると、アプリケーションからOracle Databaseに接続できます。
- Secure Sockets Layer (SSL)/Transport Layer Security (TLS)を介してLightweight Directory Access Protocol (LDAP)を使用する場合は、JDBC Thinドライバを使用します。
- Oracleクライアントの環境で使用するクライアント・アプリケーションを作成し、TCP/IP以外のネットワークのサポートなど、OCIドライバ固有の機能が必要な場合は、JDBC OCIドライバを使用します。
- データベース・サーバー内で動作し、リモート・データベースまたは同じデータベース・インスタンス内の別のセッションにアクセスする必要があるコードの場合は、JDBCサーバー側Thinドライバを使用します。
- コードがデータベース・サーバー内で動作し、セッション内でローカルにデータにアクセスする必要がある場合は、JDBCサーバー側内部ドライバを使用して、該当するサーバーにアクセスします。

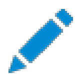
1.3 JDBC OCIドライバとJDBC Thinドライバの機能の相違点

表1-1は、Oracle Databaseリリース18cのJDBC OCIドライバまたはJDBC Thinドライバ固有の機能の一覧です。

表1-1 JDBC OCIドライバとJDBC Thinドライバの機能の相違点

JDBC OCIドライバ	JDBC Thinドライバ

JDBC OCIドライバ	JDBC Thinドライバ
OCI 接続プーリング	なし
なし	ネイティブ XA のデフォルト・サポート
透過的アプリケーション・フェイルオーバー(TAF)	なし
なし	アプリケーション・コンティニュイティ
なし	トランザクション・ガード
なし	配列 DML の反復ごとの行数のサポート
なし	Oracle Advanced Security における SHA-2 のサポート
oraaccess.xml 構成ファイル設定	なし
なし	Oracle アドバンスト・キューイング
なし	連続問合せ通知
なし	07L_MR クライアント機能のサポート
なし	ローカル・トランザクションからグローバル・トランザクションへの昇格のサポート



ノート:

- OCI の最適化フェッチ機能は、JDBC OCI ドライバの内部機能であり、JDBC Thin ドライバには適用されません。
- JDBC OCI ドライバ機能の一部は、OCI ライブラリから継承されたもので、Thin JDBC ドライバでは使用できません。

1.4 環境およびサポート

この項では、次の項目について簡単に説明します。

- サポートされるJDKおよびJDBCのバージョン
- JNI環境およびJava環境

- JDBCとIDE

1.4.1 サポートされるJDKおよびJDBCのバージョン

Oracle Database 19cでは、すべてのJDBCドライバはJDK 10と互換性があります。JDK 10のサポートは、ojdbc10.jarファイルを介して提供されます。

関連トピック

- Oracle JDBCドライバのバージョン互換性
- Oracle JDBCドライバのバージョン互換性

1.4.2 JNI環境およびJava環境

JDBC OCIドライバは、OCI Cライブラリのコールに、標準Java Native Interface (JNI)を使用しています。Java仮想マシン(JVM)、特にMicrosoft社やIBM社のJVMでJDBC OCIドライバを使用できます。

1.4.3 JDBCとIDE

Oracle JDeveloper Suiteは、開発者に対し、インターネット用のコンポーネント・ベースのデータベース・アプリケーションを作成、デバッグおよびデプロイするための、単一の統合された製品群を提供します。Oracle JDeveloper環境には、JDBC Thinドライバおよびシステム固有のOCIドライバなど、JDBCの統合サポートが含まれています。Oracle JDeveloperのデータベース・コンポーネントは、JDBCドライバを使用してクライアントとサーバー上で稼働するアプリケーション間の接続を管理します。

1.5 機能リスト

次の表は、各機能と、3つのOracle JDBCドライバ(サーバー側内部ドライバ、JDBC OCIおよびJDBC Thinドライバ)でその機能のサポートが開始されたバージョンのリストです。

表1-2 機能リスト

機能	サーバー側内部	JDBC OCI	JDBC Thin
JDK 1.0		7.2.2	7.2.2
JDBC 1.0.2		7.2.2	7.2.2
JDK 1.1.1		8.0.6	8.0.6
JDBC 1.22(新機能なし、マイナーな改訂のみ)		8.0.6	8.0.6
defineColumnType 脚注 1		8.0.6	8.0.6
行のプリフェッチ		8.0.6	8.0.6

機能	サーバー側内部	JDBC OCI	JDBC Thin
Java Native Interface		8.1.6	
JDK 1.2	9.0.1	8.1.6	8.1.6
JDBC 2.0 SQL3 型(BLOB、CLOB、Struct、Array、REF)	8.1.5	8.1.5	8.1.5
ネイティブ LOB		8.1.6	9.2.0
連想配列脚注 2	10.2.0	8.1.6	10.1.0
JDBC 2.0 スクロール可能な結果セット	8.1.6	8.1.6	8.1.6
JDBC 2.0 更新可能な結果セット	8.1.6	8.1.6	8.1.6
JDBC 2.0 標準バッチ処理	8.1.6	8.1.6	8.1.6
JDBC 2.0 接続プーリング	なし	8.1.6	8.1.6
JDBC 2.0 XA	8.1.6	8.1.6	8.1.6
サーバー側 Thin ドライバ	8.1.6	なし	なし
JDBC 2.0 RowSet		9.0.1	9.0.1
暗黙的文キャッシュ	8.1.7	8.1.7	8.1.7
明示的文キャッシュ	8.1.7	8.1.7	8.1.7
一時 LOB	9.0.1	9.0.1	9.0.1
オブジェクト型の継承	9.0.1	9.0.1	9.0.1
マルチ・レベル・コレクション	9.0.1	9.0.1	9.0.1
oracle.jdbc インタフェース	9.0.1	9.0.1	9.0.1
ネイティブ XA		9.0.1	10.1.0

機能	サーバー側内部	JDBC OCI	JDBC Thin
OCI 接続プーリング	なし	9.0.1	なし
TAF	なし	9.0.1	なし
NLS サポート	9.0.1	9.0.1	9.0.1
JDK 1.3	9.2.0	9.2.0	9.2.0
JDK 1.4	10.1.0	9.2.0	9.2.0
JDBC 3.0 セーブポイント	9.2.0	9.2.0	9.2.0
新規文キャッシュ API	9.2.0	9.2.0	9.2.0
ConnectionCacheImpl 接続キャッシュ	なし	8.1.7	8.1.7
暗黙的接続キャッシュ	なし	10.1.0	10.1.0
高速接続フェイルオーバー		10.1.0.3	10.1.0.3
接続ラップ		9.2.0	9.2.0
DMS		9.2.0	9.2.0
URL のサービス名		9.2.0	10.2.0
JDBC 3.0 接続プーリング・プロパティ	なし	10.1.0	10.1.0
JDBC 3.0 更新可能な BLOB、CLOB、REF	10.1.0	10.1.0	10.1.0
JDBC 3.0 複数の結果セットのオープン	10.1.0	10.1.0	10.1.0
JDBC 3.0 パラメータ・メタデータ	10.1.0	10.1.0	10.1.0
JDBC 3.0 名前によるストアド・プロシージャ・パラメータの設定/取得	10.1.0	10.1.0	10.1.0
JDBC 3.0 文プーリング	10.1.0	10.1.0	10.1.0

機能	サーバー側内部	JDBC OCI	JDBC Thin
名前による文パラメータの設定	10.1.0	10.1.0	10.1.0
エンドツーエンドのトレース		10.1.0	10.1.0
Web RowSet	11.1	10.1.0	10.1.0
プロキシ認証		10.2.0	10.1.0
JDBC 3.0 自動生成キー		10.2.0	10.2.0
JDBC 3.0 保持可能カーソル	10.2.0	10.2.0	10.2.0
JDBC 3.0 ローカル・トランザクションとグローバル・トランザクションの切替え	9.2.0	9.2.0	9.2.0
実行時接続ロード・バランシング	なし	10.2.0	10.2.0
LOB 用の setXXX および getXXX の拡張		10.2.0	10.2.0
XA 接続キャッシュ	なし	10.2.0	10.2.0
DML RETURNING		10.2.0	10.2.0
JSR 114 RowSets		10.2.0	10.2.0
SSL/TLS 暗号化		9.2.0	10.2.0
SSL/TLS 認証		9.2.0	11.1
JDK 5.0	11.1	11.1	11.1
JDK 6		11.1	11.1
JDBC 4.0		11.1	11.1
AES 暗号化			11.1
SHA1 ハッシュ			11.1

機能	サーバー側内部	JDBC OCI	JDBC Thin
RADIUS 認証		10.2.0	11.1
Kerberos 認証			11.1
ANYDATA および ANYTYPE 型		11.1	11.1
ネイティブ AQ			11.1
問合せ変更通知			11.1
データベースの起動と停止	なし	11.1	11.1
データ型のファクトリ・メソッド	11.1	11.1	11.1
バッファ・キャッシュ	11.1	11.1	11.1
セキュア・ファイル	11.1	11.1	11.1
診断機能	11.1	11.1	11.1
クライアント結果キャッシュ		11.1.0	18.1
サーバー結果キャッシュ	11.1	11.1.0	11.1.0
ユニバーサル接続プール		11.1.0.7.0	11.1.0.7.0
TimeZone のパッチ		11.2	11.2
セキュアな Lob のサポート		11.2	11.2
LOB プリフェッチのサポート		11.2	11.2
ネットワーク接続プール			11.2
列セキュリティのサポート			11.2
XML 型のキューのサポート(AQ)			11.2

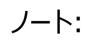
機能	サーバー側内部	JDBC OCI	JDBC Thin
通知グループ(AQとDCN)			11.2
SimpleFAN		11.2	11.2
アプリケーション・コンティニュイティ			12.1
トランザクション・ガード			12.1
SQL 文変換			12.1
データベース常駐接続プーリング		12.1	12.1
最新 JDBC 標準のサポート		12.1	12.1
Oracle Advanced Security における SHA-2 のサポート			12.1
非表示列のサポート		12.1	12.1
パラメータとして PL/SQL パッケージ・タイプのサポート		12.1	12.1
データベースの動作の監視のサポート		12.1	12.1
様々なデータ型の長さの制限拡張のサポート		12.1	12.1
暗黙的結果のサポート		12.1	12.1
配列 DML の反復ごとの行数のサポート			12.1
oraaccess.xml 構成ファイル設定		12.1	

脚注1

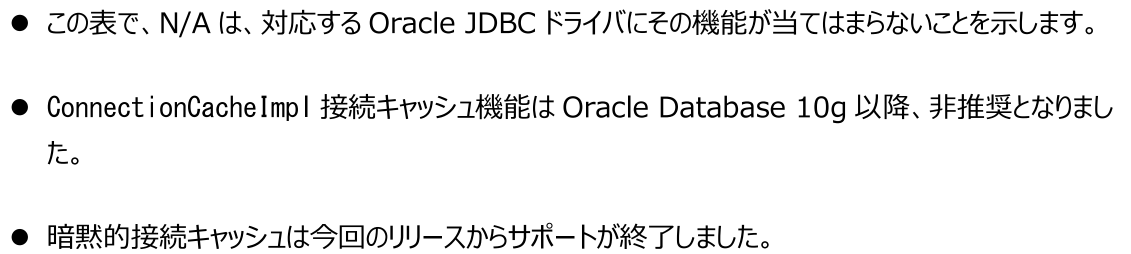
Oracle Database 12cリリース1 (12.1)以降、このメソッドのバリアントのほとんどは非推奨です。LOBからLONGへの変換を実行し、LOBプリフェッチ・サイズを構成できるのは、現在のバージョンのみです。

脚注2

連想配列は、以前は索引付き表と呼ばれていました。



ノート:



- この表で、N/A は、対応する Oracle JDBC ドライバにその機能が当てはまらないことを示します。
- `ConnectionCacheImpl` 接続キャッシュ機能は Oracle Database 10g 以降、非推奨となりました。
- 暗黙的接続キャッシュは今回のリリースからサポートが終了しました。

2 スタート･ガイド

この章では、Oracle Java Database Connectivity(JDBC)ドライバ･バージョン、データベース･バージョンおよびJava Development Kit(JDK)バージョン間の互換性について説明します。

また、クライアント･インストールと構成のテスト、および簡単なアプリケーションの実行方法の基本を説明します。この章の構成は、次のとおりです。

- Oracle JDBCドライバのバージョン互換性
- JDBCクライアント･インストールの検証
- JDBCでの基本ステップ
- サンプル：接続、問合せおよび結果処理
- 非表示列のサポート
- JSONデータの検証のサポート
- 暗黙的結果のサポート
- 軽量接続検証のサポート
- データベース･ノードの優先度付け解除のサポート
- Traffic DirectorモードのOracle Connection Managerのサポート
- Memoptimized Rowstoreの高速収集のサポート
- JDBCプログラムでのストアド･プロシージャ･コール
- SQL例外の処理について

2.1 Oracle JDBCドライバのバージョン互換性

この項では、一般的なJDBCバージョンの互換性について説明します。

次の表に、JDBCとOracle Databaseの相互運用性マトリックスまたは動作保証マトリックスを示します。

JDBCドライバのバージョン	Database 19.x	Database 18.3	Database 12.2および12.1	Database 11.2.0.4
JDBC 19.x	あり	あり	あり	あり
JDBC 18.3	あり	あり	あり	あり
JDBC 12.2 および 12.1	あり	あり	あり	あり
JDBC 11.2.0.4	あり	あり	あり	あり

Oracle JDBCドライバは常に、新しいリリースごとに最新のJDKバージョンに準拠しています。一部のバージョンでは、JDBCドライバは複数のJDKバージョンをサポートしています。次の表に、リリース固有のJDBC JARファイルと、様々なOracle Databaseバージョンでサポートされているバージョンを示します。

Oracle Databaseのバージョン	リリース固有のJDBC JARファイルとサポートされているJDKバージョン
19.x	ojdbc10.jar と JDK 10、JDK 11 ojdbc8.jar と JDK 8、JDK 9、JDK 11
18.3	ojdbc8.jar と JDK 8、JDK 9、JDK 10、JDK 11
12.2 または 12cR2	ojdbc8.jar と JDK 8
12.1 または 12cR1	ojdbc7.jar と JDK 7、JDK 8 ojdbc6.jar と JDK 6
11.2 または 11gR2	ojdbc6.jar と JDK 6、JDK 7、JDK 8 ojdbc5.jar と JDK 5

関連トピック

- Oracle Universal Connection Pool開発者ガイド
- http://www.oracle.com/technetwork/database/enterprise-edition/jdbc-faq-090281.html

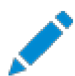
2.2 JDBCクライアント・インストールの検証

JDBCクライアント・インストールを検証するには、次の操作をすべて実行する必要があります。

- インストールされたディレクトリとファイルの確認
- 環境変数のチェック
- Javaコードのコンパイルと実行の確認
- JDBCドライバのバージョンの確認
- JDBCおよびデータベース接続のテスト

この項では、選択したドライバのインストールが完了しているものとして、JDBCドライバのOracleクライアントのインストールを検証するステップを説明します。Oracle JDBCドライバのインストールは、プラットフォームによって異なります。プラットフォーム固有の情報が記載されたマニュアルの、ドライバのインストール手順に従う必要があります。

JDBC Thinドライバを使用する場合は、クライアント・コンピュータにその他のインストールは必要ありません。JDBC Oracle Call Interface(OCI)ドライバを使用する場合は、Oracleクライアント・ソフトウェアもインストールする必要があります。これには、Oracle NetおよびOCIライブラリが含まれます。



ノート:

JDBC Thin ドライバでは、データベースがインストールされているコンピュータ上に TCP/IP リスナーが必要で

す。

2.2.1 インストールされたディレクトリとファイルの確認

Oracle Java製品をインストールすると、この項に記載されているディレクトリなどが作成されます。

- ORACLE_HOME/jdbc
- ORACLE_HOME /jlib

次のディレクトリおよびファイルがORACLE_HOME/jdbcディレクトリに作成および格納されているかどうかを確認してください。

- doc

  このディレクトリには、Oracle JDBC Application Program Interface(API)のドキュメントであるjavadoc.zipファイルが格納されています。

- lib

  libディレクトリには、次のような必須Javaクラスが格納されています。

  - ojdbc10.jarおよびojdbc10_g.jar

    JDK 8、JDK 9およびJDK 11で使用するためのJDBCドライバ・クラスが含まれています

  - ojdbc8.jarおよびojdbc8_g.jar

    JDK 8で使用するためのJDBCドライバ・クラスが含まれています

- orai18n.jarおよびorai18n-mapping.jar

  グローバリゼーション用およびマルチバイト文字セット・サポート用のクラスが含まれています。

- Readme.txt

  このファイルには、ドライバに関する最新情報およびリリース固有の情報が含まれています。これらの情報は、製品の他のドキュメントには含まれていない場合があります。

次のディレクトリがORACLE_HOME /jlibディレクトリに作成され、ファイルが格納されているかどうか確認してください。

- jta.jarおよびjndi.jar

  これらのファイルには、Java Transaction API(JTA)とJava Naming and Directory Interface(JNDI)のためのクラスが含まれています。これらのファイルが必要になるのは、分散トランザクション管理のためのJTA機能、またはネーミング・サービスのためのJNDI機能を使用する場合のみです。

- ons.jar

  このJARファイルには、Oracle RAC高速アプリケーション通知のクラスが含まれています。これは、高速接続フェイルオーバー、実行時ロード・バランシング、Webセッション・アフィニティおよびトランザクション・アフィニティなどのユニバーサル接続プール(UCP)機能にも必要です。

関連トピック

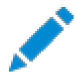
- Oracle RAC高速アプリケーション通知
- jta.jar
- jndi.jar
- Oracle Universal Connection Pool開発者ガイド

2.2.2 環境変数のチェック

この項では、Solaris、LinuxおよびMicrosoft Windowsプラットフォーム上にJDBC OCIドライバとJDBC Thinドライバのために設定する必要がある環境変数について説明します。

JDBC OCIまたはThinドライバ用のCLASSPATH環境変数を設定する必要があります。CLASSPATH環境変数には次のものを含めます。

```
ORACLE_HOME/jdbc/lib/ojdbc10.jar
ORACLE_HOME/jdbc/lib/ojdbc8.jar
ORACLE_HOME/jlib/orai18n.jar
```



ノート:

JTA 機能および JNDI 機能を使用する場合は、jta.jar と jndi.jar も CLASSPATH 環境変数に指定する必要があります。

JDBC OCIドライバ

JDBC OCIドライバを使用するには、ライブラリ・パス環境変数に次の値を設定する必要もあります。

- SolarisまたはLinuxの場合は、LD_LIBRARY_PATH環境変数を次のように設定します。

  ```
  ORACLE_HOME/lib
  ```

  このディレクトリには、libocijdbc19.so共有オブジェクト・ライブラリが格納されます。

- Microsoft Windowsでは、次のようにPATH環境変数を設定します。

  ```
  ORACLE_HOME¥bin
  ```

  このディレクトリには、ocijdbc19.dll動的リンク・ライブラリが格納されます。

ライブラリ・パス環境変数にJDBC OCI Instant Clientデータ共有ライブラリを指定すると、すべてのJDBC OCIデモ用プログラムをInstant Clientモードで実行できます。

JDBC Thinドライバ

JDBC Thinドライバを使用するために、他の環境変数を指定する必要はありません。ただし、JDBCサーバー側Thinドライバを使用するには、許可を設定する必要があります。

サーバー側Thinドライバの許可の設定

JDBCサーバー側Thinドライバは、データベースへの接続用のソケットをオープンします。Oracle DatabaseではJavaセキュリティ・モデルを施行しているため、SocketPermissionオブジェクトについてチェックが実行されます。

JDBCサーバー側Thinドライバを使用するには、接続するユーザーに適切な許可を付与する必要があります。次の例は、ユーザーHRに許可を付与する方法を示しています。

```
CREATE ROLE jdbcthin;
CALL dbms_java.grant_permission('JDBCTHIN', 'java.net.SocketPermission', '*', 'connect');
GRANT jdbcthin TO HR;
```

grant_permissionコールのJDBCTHINは大文字で指定する必要があることに注意してください。アスタリスク(*)はパターンです。特定のコンピュータまたはポートのみに接続する許可を付与してユーザーへの許可を制限できます。

関連トピック

- JDBC OCIドライバ固有の機能
- Oracle Database Java開発者ガイド

2.2.3 Javaコードのコンパイルと実行の確認

Javaがクライアント・システムで正しく設定されたことを確認するには、ORACLE_HOME/jdbc/demoの下のsamplesディレクトリに移動します。次に、コマンドラインで次のコマンドを順に入力して、JavaコンパイラおよびJavaインタプリタがエラーなく実行されていることを確認します。

```
javac
java
```

これらの各コマンドは、オプションとパラメータのリストを表示して終了します。可能であれば、jdbc/demo/samples/generic/SelectExampleなどの簡単なテスト・プログラムを使用して、コンパイルと実行の確認をしてください。

2.2.4 JDBCドライバのバージョンの確認

JDBCドライバのバージョンを確認するには、次のサンプル・コードのように、OracleDatabaseMetaDataクラスのgetDriverVersionメソッドをコールします。

```
import java.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.OracleDataSource;
class JDBCVersion
{
  public static void main (String args[]) throws SQLException
  {
    OracleDataSource ods = new OracleDataSource();
    ods.setURL("jdbc:oracle:thin:HR/hr@<host>:<port>:<service>");
    Connection conn = ods.getConnection();
    // Create Oracle DatabaseMetaData object
    DatabaseMetaData meta = conn.getMetaData();
    // gets driver info:
    System.out.println("JDBC driver version is " + meta.getDriverVersion());
  }
}
```

次のコマンドを実行すると、JDBCドライバのバージョンを確認できます。

- java -jar ojdbc8.jar
- java -jar ojdbc10.jar

2.2.5 JDBCおよびデータベース接続のテスト

samplesディレクトリには、特定のOracle JDBCドライバ用のサンプル・プログラムが格納されています。その1つである

JdbcCheckup.javaは、JDBCおよびデータベース接続のテスト用です。このプログラムには、ユーザー名、パスワードおよび接続するデータベース名の入力が必要です。このプログラムは、データベースに接続し、「Hello World」という文字列の問合せを行い、それを画面に出力します。

samplesディレクトリでJdbcCheckup.javaプログラムをコンパイルして実行します。問合せの結果の画面出力でエラーが発生しなければ、JavaおよびJDBCは正しくインストールされています。

JdbcCheckup.javaは簡単なプログラムですが、次の操作を実行することにより、いくつかの重要な機能を示します。

- JDBCクラスを含む、必要なJavaクラスのインポート
- DataSourceインスタンスの作成
- データベースへの接続
- 単純な問合せの実行
- 問合せ結果の画面への出力

JDBC OCIドライバを使用するJdbcCheckup.javaプログラムは次のとおりです。

```
/*
 * This sample can be used to check the JDBC installation.
 * Just run it and provide the connect information. It will select
 * "Hello World" from the database.
 */
// You need to import the java.sql and JDBC packages to use JDBC
import java.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.OracleDataSource;
// We import java.io to be able to read from the command line
import java.io.*;
class JdbcCheckup
{
  public static void main(String args[]) throws SQLException, IOException
  {
    // Prompt the user for connect information
    System.out.println("Please enter information to test connection to
                          the database");
    String user;
    String password;
    String database;
    user = readEntry("user: ");
    int slash_index = user.indexOf('/');
    if (slash_index != -1)
    {
      password = user.substring(slash_index + 1);
      user = user.substring(0, slash_index);
    }
    else
      password = readEntry("password: ");
    database = readEntry("database(a TNSNAME entry): ");
    System.out.print("Connecting to the database...");
    System.out.flush();
    System.out.println("Connecting...");
    // Open an OracleDataSource and get a connection
    OracleDataSource ods = new OracleDataSource();
```

```
    ods.setURL("jdbc:oracle:oci:@" + database);
    ods.setUser(user);
    ods.setPassword(password);
    Connection conn = ods.getConnection();
    System.out.println("connected.");
    // Create a statement
    Statement stmt = conn.createStatement();
    // Do the SQL "Hello World" thing
    ResultSet rset = stmt.executeQuery("select 'Hello World' from dual");
    while (rset.next())
      System.out.println(rset.getString(1));
    // close the result set, the statement and the connection
    rset.close();
    stmt.close();
    conn.close();
    System.out.println("Your JDBC installation is correct.");
  }
  // Utility function to read a line from standard input
  static String readEntry(String prompt)
  {
    try
    {
      StringBuffer buffer = new StringBuffer();
      System.out.print(prompt);
      System.out.flush();
      int c = System.in.read();
      while (c != '¥n' && c != -1)
      {
        buffer.append((char)c);
        c = System.in.read();
      }
      return buffer.toString().trim();
    }
    catch(IOException e)
    {
      return "";
    }
  }
}
```

2.3 JDBCでの基本ステップ

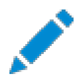
JDBCクライアント・インストールの検証が完了すると、JDBCアプリケーションの作成を開始できます。Oracle JDBCドライバを使用する場合は、プログラムに特定のドライバ固有情報を含める必要があります。この項では、ドライバ固有情報の追加場所と追加方法について、チュートリアルの形式で説明します。チュートリアルは、クライアントからデータベースへの接続および問合せを行うコードを作成する方法を、ステップを追って説明します。

次のタスクを実行するコードを記述する必要があります。

1. パッケージのインポート
2. データベースへの接続のオープン
3. 文オブジェクトの作成
4. 問合せの実行と結果セット・オブジェクトの取出し
5. 結果セット・オブジェクトの処理

6. 結果セット・オブジェクトとStatementオブジェクトのクローズ
7. データベースの変更
8. 変更のコミットについて
9. 接続のクローズ

ノート:



最初の 3 つのタスク用の Oracle ドライバ固有情報を指定して、プログラムが JDBC Application Program Interface(API)を使用してデータベースにアクセスできるようにします。その他のタスクについては、Java アプリケーションの場合と同様に、標準 JDBC Java コードを使用できます。

2.3.1 パッケージのインポート

使用するOracle JDBCドライバの種類にかかわらず、次の構文を使用して、表2-1に示されているimport文をプログラムの最初に記述する必要があります。

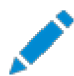
```
import <package_name>;
```

表2-1 JDBCドライバ用のimport文

import文	提供パッケージ
import java.sql.*;	標準の JDBC パッケージ。
import java.math.*;	BigDecimal クラスおよび BigInteger クラス。アプリケーションでこれらのクラスを使用しない場合は、このパッケージを省略できます。
import oracle.jdbc.*; import oracle.jdbc.pool.*; import oracle.sql.*;	JDBC に対する Oracle の拡張機能。これはオプションです。 OracleDataSource。 Oracle 型拡張機能。これはオプションです。

オプションとなっているOracleパッケージは、Oracle JDBCドライバの拡張機能へのアクセスを提供しますが、この項の例には必要ありません。

ノート:



ワイルド・カードのアスタリスク(*)は使用せずに、アプリケーションに必要なクラスのみをインポートすることをお薦めします。このガイドでは簡単にするためにアスタリスク(*)を使用していますが、クラスおよびインタフェースのインポート方法としてはお薦めしません。

2.3.2 データベースへの接続のオープン

最初に、OracleDataSourceインスタンスを作成する必要があります。次に、OracleDataSource.getConnectionメソッドを使用してデータベースへの接続をオープンします。取り出した接続のプロパティは、OracleDataSourceインスタンスから導出されたものです。URL接続プロパティを設定した場合は、TNSEntryName、DatabaseName、ServiceName、ServerName、PortNumber、Network Protocolおよびドライバのタイプを含む他のプロパティはすべて無視されます。

データベースURL、ユーザー名およびパスワードの指定

次のコードは、データソースのURL、ユーザー名およびパスワードを設定します。

```
OracleDataSource ods = new OracleDataSource();
ods.setURL(url);
ods.setUser(user);
ods.setPassword(password);
```

次の例では、JDBC Thinドライバを使用して、パスワードがhrのユーザーHRを、サービス名がorclのデータベースに、ホストmyhostのポート5221経由で接続します。

```
OracleDataSource ods = new OracleDataSource();
String url = "jdbc:oracle:thin:@//myhost:5221/orcl";
ods.setURL(url);
ods.setUser("HR");
ods.setPassword("hr");
Connection conn = ods.getConnection();
```

ノート:

引数で指定されたユーザー名とパスワードは、URL で指定されたユーザー名とパスワードをオーバーライドします。

ユーザー名とパスワードを含むデータベースURLの指定

次の例では、JDBC Oracle Call Interface(OCI)ドライバを使用して、パスワードがhrのユーザーHRを、Transparent Network Substrate(TNS)エントリがmyTNSEntryであるデータベース・ホストに接続します。この場合、URLは、ユーザー名とパスワードを含む、唯一の入力パラメータです。

```
String url = "jdbc:oracle:oci:HR/hr@myTNSEntry");
ods.setURL(url);
Connection conn = ods.getConnection();
```

Thinドライバを使用して接続する場合は、ポート番号を指定する必要があります。たとえば、ポート5221上にTCP/IPリスナーを持つホストmyhost上のデータベースに接続し、サービス識別子がorclである場合は、次のようなコードを記述します。

```
String URL = "jdbc:oracle:thin:HR/hr@//myhost:5221/orcl");
ods.setURL(URL);
Connection conn = ods.getConnection();
```

関連トピック

- [データソースおよびURL]
- [データソースおよびURL]

2.3.3 文オブジェクトの作成

データベースに接続し、そのプロセスでConnectionオブジェクトを作成したら、次のステップは、Statementオブジェクトを作成します。JDBC Connectionオブジェクトのcreateインスタンスメソッドは、JDBC Statement型のオブジェクトを返します。前の項の、Connectionオブジェクトconnが作成された例の続きとして、Statementオブジェクトを作成する方法の例を示します。

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

2.3.4 問合せの実行と結果セット・オブジェクトの取出し

データベースへの問合せを行う場合、StatementオブジェクトのexecuteQueryメソッドを使用します。このメソッドは、入力としてSQL文を受け取り、JDBC ResultSetオブジェクトを戻します。

ノート:

- Statement オブジェクトを実行するために使用されるメソッドは、実行されている SQL 文の種類により異なります。Statement オブジェクトが ResultSet オブジェクトを戻す SQL 問合せを表している場合は、executeQuery メソッドを使用します。SQL が更新件数を戻す DDL 文または DML 文の場合は、executeUpdate メソッドを使用します。SQL 文の種類がわからない場合は、execute メソッドを使用します。
- 標準 JDBC ドライバの場合、実行されている SQL 文字列が ResultSet オブジェクトを戻さないと、executeQuery メソッドで SQLException 例外がスローされます。Oracle JDBC ドライバの場合、実行されている SQL 文字列が ResultSet オブジェクトを戻さなくても、executeQuery メソッドで SQLException 例外はスローされません。

この例を続けるには、Statementオブジェクトstmtを作成したら、次のステップは、問合せを実行し、EMPLOYEESという従業員の表のfirst_name列の内容が含まれたResultSetオブジェクトを戻すことです。

```
ResultSet rset = stmt.executeQuery ("SELECT first_name FROM employees");
```

2.3.5 結果セット・オブジェクトの処理

問合せの実行後は、ResultSetオブジェクトのnext()メソッドを使用して結果を反復します。このメソッドは、結果セットを行ごとに進み、結果セットの最後に達するとそれを検出します。

結果セット内を反復しながらデータを引き出すには、ResultSetオブジェクトの適切なgetXXXメソッドを使用します。このXXXには、Javaのデータ型が対応します。

たとえば、次のコードは、以前の項のResultSetオブジェクトrset内を反復して、各従業員名の取出しおよび出力を行います。

```
while (rset.next())
   System.out.println (rset.getString(1));
```

next()メソッドは、結果セットの最後に達するとfalseを戻します。従業員名は、Java String値として実体化されます。

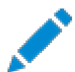
2.3.6 結果セット・オブジェクトとStatementオブジェクトのクローズ

ResultSetとStatementオブジェクトの使用後に、明示的にこれらをクローズする必要があります。これは、Oracle JDBCドライバの使用時に作成した、すべてのResultSetおよびStatementオブジェクトに適用されます。ドライバには、ファイナライザ・メソッドがありません。クリーン・アップ・ルーチンは、ResultSetおよびStatementクラスのcloseメソッドで実行されます。明示的にResultSetおよびStatementオブジェクトをクローズしないと、深刻なメモリー・リークが発生する場合があります。また、データベースのカーソルが不足します。結果セットと文の両方をクローズすると、データベース内の対応するカーソルが解放されます。結果セットのみをクローズすると、カーソルは解放されません。

たとえば、ResultSetオブジェクトがrsetで、Statementオブジェクトがstmtの場合は、次のコードの行を使用して結果セットと文をクローズできます。

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

指定したConnectionオブジェクトが作成するStatementオブジェクトをクローズする場合、接続自体はオープンしたままになります。



ノート:

一般に、close 文は finally 句に書き込みます。

2.3.7 データベースの変更

DML操作

INSERT操作またはUPDATE操作などのDML(データ操作言語)操作を実行するには、StatementオブジェクトまたはPreparedStatementオブジェクトのいずれかを作成します。PreparedStatementオブジェクトによって、様々な入力パラメータのセットで文を実行できます。JDBC ConnectionオブジェクトのprepareStatementメソッドを使用すると、様々なバインド・パラメータを取り、文定義でJDBC PreparedStatementオブジェクトを戻す文を定義できます。

データベースに送信するプリペアド文にデータをバインドするには、PreparedStatementでsetXXXメソッドを使用します。

次の例では、プリペアド文を使用して、EMPLOYEES表に2行を追加するINSERT操作を実行する方法を示します。

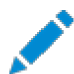
```
    // Prepare to insert new names in the EMPLOYEES table
PreparedStatement pstmt = null;
try{
    pstmt = conn.prepareStatement ("insert into EMPLOYEES (EMPLOYEE_ID, FIRST_NAME) values (?, ?)");
    // Add LESLIE as employee number 1500
    pstmt.setInt (1, 1500);          // The first ? is for EMPLOYEE_ID
    pstmt.setString (2, "LESLIE");   // The second ? is for FIRST_NAME
    // Do the insertion
    pstmt.execute();
    // Add MARSHA as employee number 507
    pstmt.setInt (1, 507);           // The first ? is for EMPLOYEE_ID
```

```
    pstmt.setString (2, "MARSHA");   // The second ? is for FIRST_NAME
    // Do the insertion
    pstmt.execute();
}
finally{
                if(pstmt!=null)
    // Close the statement
    pstmt.close();
}
```

DDL操作

データ定義言語(DDL)操作を実行するには、Statementオブジェクトを作成する必要があります。次の例では、データベースに表を作成する方法を示します。

```
//create table EMPLOYEES with columns EMPLOYEE_ID and FIRST_NAME
String query;
Statement stmt=null;
try{
    query="create table EMPLOYEES " +
          "(EMPLOYEE_ID int, " +
          "FIRST_NAME varchar(50))";
    stmt = conn.createStatement();
    stmt.executeUpdate(query);
    }
finally{
     //close the Statement object
     stmt.close();
    }
```



ノート:

PreparedStatement オブジェクトを使用して DDL 操作を実行することもできます。ただし、PreparedStatement オブジェクトの便利なところはパラメータを設定できることですが、DDL 操作にはパラメータがないため、このオブジェクトを使用しないでください。

また、データベース制限のため、PreparedStatement オブジェクトを DDL 操作に使用した場合、動作するのは初回実行時のみです。そのため、DDL 操作には Statement オブジェクトのみを使用してください。

次の例では、再実行の前にDDL文を準備する方法を示しています。

```
//
Statement stmt = null;
PreparedStatement pstmt = null;
try{
    pstmt = conn.prepareStatement ("insert into EMPLOYEES (EMPLOYEE_ID, FIRST_NAME) values (?, ?)");
    stmt = conn.createStatement("truncate table EMPLOYEES");

    // Add LESLIE as employee number 1500
    pstmt.setInt (1, 1500);          // The first ? is for EMPLOYEE_ID
    pstmt.setString (2, "LESLIE");   // The second ? is for FIRST_NAME
    pstmt.execute();
    stmt.executeUpdate();
```

```
    // Add MARSHA as employee number 507
    pstmt.setInt (1, 507);          // The first ? is for EMPLOYEE_ID
    pstmt.setString (2, "MARSHA");  // The second ? is for FIRST_NAME
    pstmt.execute();
    stmt.executeUpdate();
    }
finally{
if(pstmt!=null)

    // Close the statement
     pstmt.close();
}
```

関連トピック

- setObjectメソッドとsetOracleObjectメソッド
- その他のsetXXXメソッド

2.3.8 変更のコミットについて

デフォルトでは、データ操作言語(DML)操作は実行時に自動的にコミットされます。これは自動コミット・モードともいいます。自動コミット・モードがオンで、接続オブジェクトのcommitまたはrollbackメソッドを使用して、COMMITまたはROLLBACK操作を実行する場合、次のエラー・メッセージを受信します。

表2-2 自動コミット・モードがオンの場合に実行された操作に対するエラー・メッセージ

操作	エラー・メッセージ
COMMIT	自動コミットがオンの状態でコミットできませんでした。
ROLLBACK	自動コミットがオンの状態でロールバックできませんでした。

COMMITまたはROLLBACK操作を実行し、前の表に示したエラー・メッセージを受信したときにSQLExceptionが発生した場合、接続の自動コミット・ステータスを確認します。これは、自動コミット値がtrueに設定されている接続でこれらの操作を実行した場合、例外が発生するためです。

このような例外は、次の場合のいずれかで発生します。

- 自動コミット・ステータスがtrueに設定されていて、commitまたはrollbackメソッドがコールされた場合
- 自動コミットのデフォルトのステータスが変更されないで、commitまたはrollbackメソッドがコールされた場合
- COMMIT_ON_ACCEPT_CHANGESプロパティの値がtrueで、acceptChangesメソッドが行セットでコールされた後にcommitまたはrollbackメソッドがコールされた場合

ただし、Connectionオブジェクトでの次のメソッドのコールによって、自動コミット・モードを無効にできます。

```
conn.setAutoCommit(false);
```

自動コミット・モードを無効にした場合は、Connectionオブジェクトで適切なメソッドをコールして、変更を手動でコミットするか、ロールバックする必要があります。

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

または

```
conn.rollback();
```

COMMIT操作またはROLLBACK操作は、直前のCOMMITまたはROLLBACK以降に実行されたすべてのDML文に影響を与えます。

ノート:

- 自動コミット・モードを無効にして、直前の変更を明示的にコミットまたはロールバックせずに接続をクローズした場合は、暗黙的な COMMIT 操作が実行されます。
- すべてのデータ定義言語(DDL)操作には、常に暗黙的な COMMIT が含まれます。自動コミット・モードを無効にした場合、この暗黙的な COMMIT は、まだ明示的にコミットまたはロールバックされていない保留 DML 文もコミットします。

関連トピック

- 自動コミット・モードの無効化

2.3.8.1 コミット動作の変更

トランザクションによりデータベースを更新すると、この更新に対応するREDOエントリが生成されます。Oracle Databaseでは、トランザクションの完了まで、このREDOを一時的にメモリーに保存します。トランザクションをコミットすると、ログ・ライター(LGWR)プロセスによりコミットのREDOエントリが、そのトランザクションにおけるすべての変更について累積したREDOエントリとともに、ディスクに書き込まれます。デフォルトでは、Oracle Databaseは、コールがクライアントに戻る前にREDOをディスクに書き込みます。アプリケーションはREDOエントリがディスクに保存されるまで待つ必要があるため、この動作によりコミットに待機時間が生じます。

アプリケーションに非常に高いトランザクション・スループットが必要で、コミット待機時間を短縮するためにコミットの永続性を犠牲にしてもよいという場合は、アプリケーションの必要に応じて、デフォルトのCOMMIT操作の動作を変更できます。COMMIT操作の動作は、次のオプションにより変更できます。

- WAIT
- NOWAIT
- WRITEBATCH
- WRITEIMMED

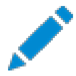
これらのオプションを指定すると、コミット段階の2つの異なる面を制御できます。

- COMMITコールが、サーバーによって処理されるまで待機するかどうか。これには、WAITオプションまたはNOWAITオプションを使用します。
- ログ・ライターがコールをバッチ処理するかどうか。これには、WRITEIMMEDオプションまたはWRITEBATCHオプションを使用します。

異なるオプションを組み合せることもできます。たとえば、COMMITコールがサーバーによる処理を待たずに戻り、ログ・ライターがコ

ミットをバッチ処理するようにするには、NOWAITオプションとWRITEBATCHオプションを一緒に使用します。たとえば:

```
((OracleConnection)conn).commit(
    EnumSet.of(
      OracleConnection.CommitOption.WRITEBATCH,
      OracleConnection.CommitOption.NOWAIT));
```



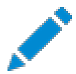
ノート:

WAIT オプションと NOWAIT オプションは、逆の意味を持つため、一緒に使用できません。一緒に使用すると、JDBC ドライバは例外をスローします。同じことが、WRITEIMMED オプションと WRITEBATCH オプションにも当てはまります。

2.3.9 接続のクローズ

必要な操作をすべて実行し、接続が不要になった後に、データベースへの接続をクローズする必要があります。次のように、Connectionオブジェクトのcloseメソッドを使用してクローズできます。

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



ノート:

一般に、close 文は finally 句に書き込みます。

2.4 サンプル: 接続、問合せおよび結果処理

次の例は、前の項で説明した処理を例証するものです。Oracle JDBC Thinドライバを使用してデータソースを作成し、データベースに接続し、Statementオブジェクトを作成して問合せを実行し、結果セットを処理します。

Statementオブジェクトの作成、問合せの実行、ResultSetの戻りと処理、および文と接続のクローズを行うコードは、標準JDBC APIを使用していることに注意してください。

```
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.sql.SQLException;
import oracle.jdbc.pool.OracleDataSource;
class JdbcTest
{
   public static void main (String args []) throws SQLException
   {
OracleDataSource ods = null;
Connection conn = null;
Statement stmt = null;
ResultSet rset = null;
      // Create DataSource and connect to the local database
      ods = new OracleDataSource();
      ods.setURL("jdbc:oracle:thin:@//localhost:5221/orcl");
      ods.setUser("HR");
      ods.setPassword("hr");
```

```
      conn = ods.getConnection();
try
{
      // Query the employee names
      stmt = conn.createStatement ();
      rset = stmt.executeQuery ("SELECT first_name FROM employees");
      // Print the name out
      while (rset.next ())
         System.out.println (rset.getString (1));
    }
      //Close the result set, statement, and the connection
finally{
      if(rset!=null) rset.close();
      if(stmt!=null) stmt.close();
      if(conn!=null) conn.close();
}
   }
}
```

OCIドライバ用のコードを利用する場合は、OracleDataSource.setURLメソッドのコールを次の文で置き換えます。

```
ods.setURL("jdbc:oracle:oci:@MyHostString");
```

MyHostStringには、TNSNAMES.ORAファイル内のエントリを指定します。

2.5 非表示列のサポート

今回のリリース以降、Oracle Databaseでは非表示の列をサポートしています。この機能を使用すると、非表示モードで表に列を追加し、後で表示するようにできます。JDBCでは、非表示列の情報を取得するAPIを提供しています。列が非表示かどうかについての情報を取得するには、次のようにoracle.jdbc.OracleResultSetMetaDataインタフェースで利用可能なisColumnInvisibleメソッドを使用できます。

例

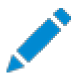
```
...
Connection conn = DriverManager.getConnection(jdbcURL, user, password);
Statement stmt = conn.createStatement ();
stmt.executeQuery ("create table hiddenColsTable (a varchar(20), b int invisible)");
stmt.executeUpdate("insert into hiddenColsTable (a,b ) values('somedata',1)");
stmt.executeUpdate("insert into hiddenColsTable (a,b) values('newdata',2)");

System.out.println ("Invisible columns information");
try
{
      ResultSet rset = stmt.executeQuery("SELECT a, b FROM hiddenColsTable");
      OracleResultSetMetaData rsmd = (OracleResultSetMetaData)rset.getMetaData();
      while (rset.next())
      {
        System.out.println("column1 value:" + rset.getString(1));
        System.out.println("Visibility:" + rsmd.isColumnInvisible(1));
        System.out.println("column2 value:" + rset.getInt(2));
        System.out.println("Visibility:" + rsmd.isColumnInvisible(2));
      }
}
catch (Exception ex)
```

```
{
        System.out.println("Exception :" + ex);
        ex.printStackTrace();
}
```

また、oracle.jdbc.OracleDatabaseMetaDataクラスで利用可能なgetColumnsメソッドを使用して、非表示列の情報を取得することもできます。

例

```
...
Connection conn = DriverManager.getConnection(jdbcURL, user, password);
Statement stmt = conn.createStatement ();
stmt.executeQuery ("create table hiddenColsTable (a varchar(20), b int invisible)");
stmt.executeUpdate("insert into hiddenColsTable (a,b ) values('somedata',1)");
stmt.executeUpdate("insert into hiddenColsTable (a,b) values('newdata',2)");

 System.out.println ("getColumns for table with invisible columns");
 try
 {
      DatabaseMetaData dbmd = conn.getMetaData();
      ResultSet rs = dbmd.getColumns(null, "HR", "hiddenColsTable", null);
      OracleResultSetMetaData rsmd = (OracleResultSetMetaData)rs.getMetaData();
      int colCount = rsmd.getColumnCount();
      System.out.println("colCount: " + colCount);
      String[] columnNames = new String [colCount];

      for (int i = 0; i < colCount; ++i)
      {
        columnNames[i] = rsmd.getColumnName (i + 1);
      }

      while (rs.next())
      {
        for (int i = 0; i < colCount; ++i)
           System.out.println(columnNames[i] +":" +rs.getString (columnNames[i]));
      }
 }
 catch (Exception ex)
 {
      System.out.println("Exception: " + ex);
      ex.printStackTrace();
 }
```



ノート:

サーバー側の内部ドライバである kprb は、非表示列の情報のフェッチをサポートしていません。

2.6 JSONデータの検証のサポート

Oracle Databaseリリース18c以降、JDBCドライバは、ResultSetで返される列がJSON列であるかどうかを確認できます。列がJSONであるかどうかについての情報を取得するには、次のようにoracle.jdbc.OracleResultSetMetaDataインタフェー

スで利用可能なisColumnJSONメソッドを使用できます。

例2-1 例

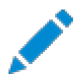
```
...
public void test(Connection conn)
    throws Exception{
  try {
    show ("tkpjb26776242 - start");
    createTable(conn);

    String sql = "SELECT col1, col2, col3, col4, col5, col6, col7, col8 FROM tkpjb26776242_tab";
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(sql);
    ResultSetMetaData rsmd = rs.getMetaData();
    OracleResultSetMetaData orsmd   = (OracleResultSetMetaData)rsmd;

    int colCnt = orsmd.getColumnCount();
    show("Table has " + colCnt + " columns.");
    for (int i = 1; i <= colCnt; i++) {
      String columnName = orsmd.getColumnName(i);
      String typeName = orsmd.getColumnTypeName(i);
      boolean invisible = orsmd.isColumnInvisible(i);
      boolean json = orsmd.isColumnJSON(i);
      show(columnName + "  " + typeName + (invisible?"   INVISIBLE":"") + (json?"   JSON":""));
    }
    rs.close();
    stmt.close();

    show ("tkpjb26776242 - end");
  }
  finally {
    dropTable(conn);
  }
}


private void createTable(Connection conn) throws Exception{
  String sql = " create table tkpjb26776242_tab ( "
             + "  col1 clob, "
             + " col2 clob , "
             + " col3 clob INVISIBLE, "
             + " col4 clob INVISIBLE, "
             + " col5 varchar2(200), "
             + " col6 varchar2(200), "
             + " col7 varchar2(200) INVISIBLE, "
             + " col8 varchar2(200) INVISIBLE, "
             + " check (col2 IS JSON), "
             + " check (col4 IS JSON), "
             + " check (col6 IS JSON), "
             + " check (col8 IS JSON))";

  Util.doSQL(conn, sql);
}

private void dropTable(Connection conn) throws Exception{
  String sql = " drop table tkpjb26776242_tab";
```

```
    Util.trySQL(conn, sql);
  }
    ...
```

2.7 暗黙的結果のサポート

このリリースより、Oracle Databaseで、ストアド・プロシージャで実行されたSQL文の結果がクライアント・アプリケーションに暗黙的に返されることがサポートされ、REF CURSORを明示的に使用する必要がなくなりました。次のメソッドを使用して、PL/SQLプロシージャまたはブロックで戻された暗黙的な結果を取得し処理できます。

メソッド	説明
getMoreResults	結果セットで使用できる結果があるかどうかを確認します。
getMoreResults(int)	オーバーロードされたメソッドなどの結果セットで使用できる結果があるかどうかを確認します。このメソッドは int パラメータを受け入れます。値は次のいずれかです。 ● KEEP_CURRENT_RESULT ● CLOSE_ALL_RESULTS ● CLOSE_CURRENT_RESULT
getResultSet	実行された PL/SQL 文から暗黙的結果を繰り返し取得します。



ノート:

- サーバー側の内部ドライバである kprb は、暗黙的結果の情報のフェッチをサポートしていません。
- SELECT 問合せのみを暗黙的に戻すことができます。
- アプリケーションは順次、各結果セットを取得しますが、順序に依存しない結果セットから行をフェッチできます。

次のように、プロシージャでfooをコールするとします。

```
 create procedure foo as
  c1 sys_refcursor;
  c2 sys_refcursor;
begin
  open c1 for select * from hr.employees;
  dbms_sql.return_result(c1); --return to client
  -- open 1 more cursor
  open c2 for select * from hr.departments;
  dbms_sql.return_result (c2); --return to client
end;
```

次のコードの抜粋に、getMoreResultsメソッドを使用してPL/SQLプロシージャによって戻された暗黙的結果を取得する方法を示します。

例1

```
String sql = "begin foo; end;";
...
Connection conn = DriverManager.getConnection(jdbcURL, user, password);
 try {
        Statement stmt = conn.createStatement ();
        stmt.executeQuery (sql);

        while (stmt.getMoreResults())
        {
              ResultSet rs = stmt.getResultSet();
              System.out.println("ResultSet");
              while (rs.next())
             {
                   /* get results */
             }
         }
      }
```

次のように、別のプロシージャでfooをコールするとします。

```
create or replace procedure foo asc1 sys_refcursor; c2 sys_refcursor; c3 sys_refcursor; begin   open
c1 for 'select * from hr.employees';
dbms_sql.return_result (c1);-- cursor 2open c2 for 'select * from hr.departments';
dbms_sql.return_result (c2);-- cursor 3open c3 for 'select first_name from hr.employees';
dbms_sql.return_result (c3); end;
```

次のコードの抜粋に、getMoreResults(int)メソッドを使用してPL/SQLプロシージャによって戻された暗黙的結果を取得する方法を示します。

例2

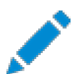
```
String sql = "begin foo; end;";
...
Connection conn = DriverManager.getConnection(jdbcURL, user, password);

try {
        Statement stmt = conn.createStatement ();
        stmt.executeQuery (sql);
        ResultSet rs = null;

        boolean retval = stmt.getMoreResults(Statement.KEEP_CURRENT_RESULT))
        if (retval)
        {
            rs = stmt.getResultSet();
            System.out.println("ResultSet");
            while (rs.next())
            {
                /* get results */
            }
```

```
        }

        /* closes open results */
        retval = stmt.getMoreResults(Statement.CLOSE_ALL_RESULTS);

        if (retval)
        {
            System.out.println("More ResultSet available");
            rs = stmt.getResultSet();
            System.out.println("ResultSet");
            while (rs.next())
            {
                /* get results */
            }
        }

        /* close current result set */
        retval = stmt.getMoreResults(Statement.CLOSE_CURRENT_RESULT);

        if(retval)
        {
            System.out.println("More ResultSet available");
            rs = stmt.getResultSet();
            while (rs.next())
            {
                /* get Results */

            }
        }
    }
```

2.8 軽量接続検証のサポート

Oracle Databaseリリース18c以降、JDBC Thinドライバは軽量接続検証をサポートしています。軽量接続検証を使用すると、JDBCアプリケーションは、データベースへのラウンドトリップを必要としない、長さ0のNSデータ・パケットを送信することで、接続の有効性を検証できます。Oracle Databaseの以前のリリースでは、接続の有効性をテストするために isValid(timeout)メソッドをコールすると、Oracle JDBCドライバではピンポン・プロトコルが使用されます。これは、データベースへの完全なラウンドトリップを行うため、負荷の高い操作になります。Oracle Databaseリリース18cでは、isValid(timeout)メソッドは空のパケットをデータベースに送信し、その応答を待機しません。そのため、接続検証が高速になり、アプリケーションのパフォーマンスが改善されます。

軽量接続検証はデフォルトで無効になっています。この機能を有効にするには、oracle.jdbc.defaultConnectionValidation接続プロパティの値をSOCKETに設定する必要があります。このプロパティが設定されている場合、isValid(timeout)メソッドをコールすると、JDBCドライバによって軽量接続検証が実行されます。



ノート:

- 軽量接続検証は、基礎となるソケット状態のみをチェックします。isValid(timeout)メソッドが true を返した場合、つまり、接続が有効であるとみなされた場合、この検証では、サーバーが到達不能(デッド・ソケット)ではないことのみが保証されます。サーバー・プロセスに関するステータス(実行されているかどうかなど)は

提供されません。ただし、デフォルトでは、つまり、軽量接続検証が有効でない場合、isValid(timeout)メソッドは、クライアントとサーバー間のネットワークが損なわれていないかどうかを確認します。

- この機能は、JDBC Thin ドライバのみでサポートされます。

軽量接続検証の新しいAPI

- oracle.jdbc.defaultConnectionValidation

  この接続プロパティは、接続検証のレベルを指定します。このプロパティに指定できる値は、NONE、LOCAL、SOCKET、NETWORK、SERVERおよびCOMPLETEです。これらの値では大文字と小文字が区別され、これらの値以外の値を設定すると、例外がスローされます。デフォルト値はNETWORKです。

- public boolean isValid(ConnectionValidation validation_level, int timeout) throws SQLException

  既存のisValid(timeout)メソッドの新しいバリエーションでは、検証のレベル(validation_level)とtimeoutという2つのパラメータを指定できます。最初のパラメータは、接続検証のレベルを指定します。

例2-2 軽量接続検証の例

次のコード・スニペットに、軽量接続メカニズムを実装する方法を示します。

```
...
OracleDataSource ods = new OracleDataSource();
ods.setURL(url);
ods.setUser(user);
ods.setPassword(password);
Connection conn = ods.getConnection();
try{
      boolean isValid = ((OracleConnection)conn).
                                    isValid(ConnectionValidation.SOCKET,timeout);
      System.out.println("Connection isValid = "+isValid);
}
catch (Exception ex)
{
     System.out.println("Exception :" + ex);
     ex.printStackTrace();
}
...    ...
```

2.9 データベース・ノードの優先度付け解除のサポート

Oracle Database 12cリリース2 (12.2.0.1)以降、JDBCドライバではデータベース・ノードの優先度付け解除がサポートされています。ノードが失敗すると、JDBCではこのノードの優先度付けを次の10分間解除します。これはデフォルトの失効時間です。たとえば、3つのノード(A、B、C)があり、ノードAが停止した場合、接続が割り当てられるのは、まずノードBおよびCからで、次にノードAとなります。デフォルトの失効時間の後、ノードAの優先度付けは解除されなくなります。つまり、接続は使用可能な3つのノードから割り当てられます。また、デフォルトの失効時間にノードAへの接続試行が成功すると、ノードAは優先度付け解除済ノードとみなされなくなります。ユーザーは、oracle.net.DOWN_HOSTS_TIMEOUTシステム・プロパティを使用して、優先度付け解除のデフォルトの失効時間を指定できます。

たとえば、次のURLのscan_listener0は、IPアドレスを取得すると、ip1、ip2およびip3のIPアドレスが構成されています。

ip1の優先度付けが解除されている場合、IPアドレスの試行の順序はip2、ip3、ip1となります。すべてのIPアドレスを使用できない場合、node_1およびnode_2を試行した後、最後にホスト全体が試行されます。

```
(DESCRIPTION_LIST=
    (DESCRIPTION=
        (ADDRESS_LIST=
            (ADDRESS=(PROTOCOL=tcp)(HOST=scan_listener0)(PORT=1521))
            (ADDRESS=(PROTOCOL=tcp)(HOST=node_1)(PORT=1528))
            (ADDRESS=(PROTOCOL=sdp)(HOST=node_2)(PORT=1527))
        )
        (ADDRESS_LIST=
            (ADDRESS=(PROTOCOL=tcp)(HOST=node_3)(PORT=1528))
        )
        (CONNECT_DATA=(SERVICE_NAME=cdb3))
    )
    (DESCRIPTION=
        (ADDRESS=(PROTOCOL=tcp)(HOST=node_0)(PORT=1528))
        (CONNECT_DATA=(SERVICE_NAME=cdb3))
    )
)
```

2.10 Traffic DirectorモードのOracle Connection Managerのサポート

Oracle Databaseリリース18cのJDBCドライバは、Traffic DirectorモードのOracle Connection Managerをサポートします。これは、データベース・クライアントとデータベース・インスタンスの間に配置されたプロキシです。JDBCクライアントは、Traffic DirectorモードのOracle Connection Managerに接続し、そこからターゲットOracle Databaseに接続できます。クライアントから送信されるTwo-Task Common (TTC)メッセージは、Traffic DirectorモードのOracle Connection Managerによって捕捉されます。ここで、受信TTCメッセージを解析し、リクエストを適切な宛先データベースに引き継ぎます。レスポンスを受信すると、Traffic DirectorモードのOracle Connection Managerは、TTCレスポンスを介して宛先データベースからクライアントに結果を返します。

次の図は、Traffic DirectorモードのOracle Connection Managerのアーキテクチャを示しています。

図2-1 Traffic DirectorモードのOracle Connection Managerのアーキテクチャ

関連項目:

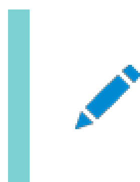
- Traffic DirectorモードのOracle Connection Managerを設定するためにcman.oraファイルを構成する方法の詳細は、Oracle Database Net Services管理者ガイドを参照してください
- Traffic DirectorモードのOracle Connection Managerのパラメータの詳細は、Oracle Database Net Servicesリファレンスを参照してください

2.10.1 Traffic DirectorモードのOracle Connection Managerの実行モード

Traffic DirectorモードのOracle Connection Managerは次のいずれかのモードで実行できます。

- プールされた接続モード

  プールされた接続モードは、プロキシ常駐接続プーリングと呼ばれる新機能を使用します。これは、データベース常駐接続プーリングのプロキシ対応モードです。プロキシ常駐接続プーリングは、少数のデータベース接続に対して大量のクライアント接続を多重化するため、データベース上の接続負荷を軽減します。Oracle Database 12cリリース1 (12.1)以降のJDBCドライバを使用するアプリケーションは、この接続モードを使用できます。



ノート:

この機能は、DRCP 対応の接続プールを使用しているクライアントで使用することをお薦めします。

- プールされていない接続モードまたは専用接続モード

  プールされていない接続モードまたは専用接続モードは、Oracle Database 11gリリース2 (11.2.0.4)以降のJDBCドライバを使用するアプリケーションで使用できます。ただし、接続の多重化などの一部の機能は、このモードでは使用できません。

関連トピック

- データベース常駐接続プーリングの概要

関連項目:

- データベース管理者ガイド
- Universal Connection Pool開発者ガイド

2.10.2 Traffic DirectorモードのOracle Connection Managerの利点

Traffic DirectorモードのOracle Connection Managerには、次の利点があります。

- 透過的パフォーマンスの強化および接続の多重化。これには、次のようなことが含まれます。
  - 文キャッシュ、行のプリフェッチ、および結果セット・キャッシュがすべての操作モードで自動的に有効になります。
  - プロキシ常駐接続プール(PRCP)を使用したデータベース・セッションの多重化(プールされたモードのみ)。PRCPは、データベース常駐接続プーリング(DRCP)のプロキシ・モードです。アプリケーションは、Traffic DirectorモードのOracle Connection Managerとデータベースとの間における透過的な接続時ロード・バランシングおよび実行時ロード・バランシングを利用できます。
  - Traffic DirectorモードのOracle Connection Managerに複数のインスタンスがある場合、クライアント側の接続時ロード・バランシングにより、またはロード・バランサ(BIG-IP、NGINXなど)により、アプリケーションのスケーラビリティが向上します。
- アプリケーション停止時間ゼロ
  - 計画済のデータベース・メンテナンスまたはプラガブル・データベース(PDB)の再配置
    - プールされたモード

      Traffic DirectorモードのOracle Connection Managerは、計画済停止のOracle Notification Service (ONS)イベントに応答し、作業をリダイレクトします。要求が完了すると、Traffic DirectorモードのOracle Connection Managerのプールから接続がドレインされます。サービスの再配置は、Oracle Database 11gリリース2 (11.2.0.4)以降でサポートされています。

      PDBの再配置の場合、PDBが再配置されると、ONSが構成されていなくても、Traffic DirectorモードのOracle Connection Managerはインバンド通知に応答します(Oracle Databaseリリース18c以降のサーバーの場合のみ)。

    - プールされていないモードまたは専用モード

      クライアントからのリクエスト境界情報がない場合、Traffic DirectorモードのOracle Connection Managerは、多くのアプリケーションの計画済停止をサポートしています(リクエストまたはトランザクションの境界を越えて単純なセッション状態やカーソル状態のみを保持する必要がある場合)。次の機能がサポートされます。

      - トランザクション境界でサービスまたはPDBを停止するか、Oracle Databaseリリース18cの継続的なアプリケーション可用性を利用してリクエスト境界でサービスを停止します
      - Traffic DirectorモードのOracle Connection Managerは、透過的アプリケーション・フェイルオーバー(TAF)のフェイルオーバーのリストアを利用して、再接続して単純な状態をリストアします。
  - 大部分が読取りのワークロードでの計画外データベース停止
- 単一障害点をなくすためのTraffic DirectorモードのOracle Connection Managerの高可用性。これは次のよ

うなことによってサポートされます。

  - 接続文字列でロード・バランサまたはクライアント側ロード・バランシング/フェイルオーバーを使用する、Traffic DirectorモードのOracle Connection Managerの複数のインスタンス
  - Traffic DirectorモードのOracle Connection Managerのインスタンスのローリング・アップグレード
  - 計画済停止に際して、クライアントからTraffic DirectorモードのOracle Connection Managerへの既存の接続を正常に終了
  - Oracle Databaseリリース18c以降のクライアントへのインバンド通知
  - 古いクライアントの場合、現在のリクエストのレスポンスとともに通知を送信

- セキュリティと分離のため、Traffic DirectorモードのOracle Connection Managerには次のような備えがあります。
  - Transmission Control Protocol/Transmission Control Protocol Secure (TCP/TCPS)およびプロトコル変換をサポートするデータベース・プロキシ
  - IPアドレス、サービス名およびSecure Socket Layer/Transport Layer Security (SSL/TLS)ウォレットに基づくファイアウォール
  - マルチテナント環境でのテナントの分離
  - サービス拒否攻撃とファジング攻撃からの保護
  - オンプレミスのOracle DatabaseとOracle Cloudの間のデータベース・トラフィックのセキュア・トンネリング

2.10.3 Traffic DirectorモードのOracle Connection Managerの制限事項

次の機能は、Traffic DirectorモードのOracle Connection Managerではサポートされていません。

- 分散トランザクション
- アドバンスト・キューイング(AQ)
- データベースの起動または停止のコール
- データベース・シャーディング
- XMLコーディング
- SQL翻訳
- プロキシ認証およびTLS外部認証(LDAPで使用される識別名(DN)など)
- オブジェクトREF
- セッションの切替え
- スクロール可能カーソル
- 反復ごとのDML行数
- 暗黙的結果
- 連続問合せ通知(CQN)
- クライアント結果キャッシュ
- データベース常駐接続プーリング(DRCP)でのセッション状態の修正のためのPL/SQLコールバック

- データベース常駐接続プーリング(DRCP)での複数のタグ付け
- アプリケーション・コンティニュイティ
- SYSDBA、SYSOPERなどの認証
- Real Application Security
- PL/SQL索引付き表バインドなどのデータ型
- 一括コピー(ODP.Netのみ)
- 自己チューニング(ODP.Netのみ)
- ASO暗号化およびサポートされているアルゴリズム(ASOのみ)

2.11 Memoptimized Rowstoreの高速収集のサポート

Memoptimizeされた行ストアを使用すると、モノのインターネット(IoT)などのアプリケーションで高パフォーマンス・データ・ストリーミングが可能になります。

Memoptimized Rowstoreには、次の機能があります。

- データベースへの高頻度の単一行データの挿入処理が最適化される高速収集。
- 高頻度の問合せのためにデータベースからのデータの高速取得を可能にする高速参照。

関連項目:

Memoptimizeされた行ストアを使用した高パフォーマンス・データ・ストリーミングの有効化

この機能を使用するには、次のような表を作成します。

```
CREATE TABLE customers (
id NUMBER(20,0),
name VARCHAR2(90 BYTE),
region VARCHAR2(10 BYTE)
)
segment creation immediate
memoptimize for write
;
```

次に、JavaまたはJDBCアプリケーションからヒントを使用してINSERT文を起動します。

```
INSERT /*+ MEMOPTIMIZE_WRITE */ INTO CUSTOMERS VALUES (2, 'DOS', 'NORTH');
```

2.12 JDBCプログラムでのストアド・プロシージャ・コール

この項では、Oracle JDBCドライバが次の種類のストアド・プロシージャをサポートする方法について説明します。

- PL/SQLストアド・プロシージャ
- Javaストアド・プロシージャ

2.12.1 PL/SQLストアド・プロシージャ

JDBCでは、JDBCエスケープ構文またはPL/SQLブロック構文を使用して、PL/SQLプロシージャ/ファンクションおよび匿名ブロックの起動をサポートしています。次のPL/SQLコールは、すべてのOracle JDBCドライバで動作します。

```
// JDBC escape syntax
CallableStatement cs1 = conn.prepareCall
                        ( "{call proc (?,?)}" ) ; // stored proc
CallableStatement cs2 = conn.prepareCall
                        ( "{? = call func (?,?)}" ) ; // stored func
// PL/SQL block syntax
CallableStatement cs3 = conn.prepareCall
                        ( "begin proc (?,?); end;" ) ; // stored proc
CallableStatement cs4 = conn.prepareCall
                        ( "begin ? := func(?,?); end;" ) ; // stored func
```

Oracle構文の例として、ストアド・ファンクションを作成するPL/SQLコードの一部を示します。このPL/SQLファンクションでは文字列を取得して接尾辞を連結します。

```
create or replace function foo (val1 char)
return char as
begin
   return val1 || 'suffix';
end;
```

JDBCプログラム内のファンクションの起動は、次のようになります。

```
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:oci:@<hoststring>");
ods.setUser("HR");
ods.setPassword("hr");
Connection conn = ods.getConnection();
CallableStatement cs = conn.prepareCall ("begin ? := foo(?); end;");
cs.registerOutParameter(1,Types.CHAR);
cs.setString(2, "aa");
cs.execute();
String result = cs.getString(1);
```

2.12.2 Javaストアド・プロシージャ

JDBCを使用して、SQLインタフェース経由でJavaストアド・プロシージャをコールできます。Javaストアド・プロシージャをコールするための構文は、正しく公開されていると仮定すると、PL/SQLストアド・プロシージャをコールするための構文と同じです。正しく公開されているとは、つまり、Oracleデータ・ディクショナリに公開するためのコール仕様を記述していることを意味します。アプリケーションでJavaネイティブ・インタフェースを使用してJavaストアド・プロシージャをコールすると、static Javaメソッドを直接起動できます。

2.13 SQL例外の処理について

エラー状況を処理するために、Oracle JDBCドライバはSQL例外をスローし、java.sql.SQLExceptionクラスまたはそのサブクラスのインスタンスを作成します。エラーはJDBCドライバまたはデータベース自体のいずれかで発生する可能性があります。表示されるメッセージには、エラーおよびエラーを発行したメソッドの説明が含まれます。ランタイム情報が追加されることもあります。

JDBC 3.0では、SQLExceptionという単一の例外しかサポートされていません。ただし、エラーはカテゴリが多いので、区別すると便利です。そのため、JDBC 4.0では様々なカテゴリのエラーを特定するためにSQLException例外の一連のサブクラスが導入されています。

基本例外処理には、エラー・メッセージの取出し、エラー・コードの取出し、SQL状態の取出しおよびスタック・トレースの出力が含まれます。SQLExceptionクラスには、使用可能な場合にこのようなすべての情報を取り出す機能があります。

エラー情報の取出し

SQLExceptionクラスの次のメソッドで、基本エラー情報を取り出すことができます。

- getMessageクラスには、使用可能な場合にこのようなすべての情報を取り出す機能があります。
- getErrorCodeクラスには、使用可能な場合にこのようなすべての情報を取り出す機能があります。
- getSQLStateクラスには、使用可能な場合にこのようなすべての情報を取り出す機能があります。

次の例はgetMessage()メソッド・コールからの情報を出力します。

```
catch(SQLException e)
{
   System.out.println("exception: " + e.getMessage());
}
```

これはJDBCドライバで発生したエラーに対して、次のような情報を出力します。

```
exception: Invalid column type
```

ノート:

Oracle でサポートされている言語と文字セットでエラー・メッセージ・テキストを使用できます。

スタック・トレースの出力

SQLExceptionクラスには、スタック・トレースを出力するためのprintStackTrace()メソッドがあります。このメソッドは、Throwableオブジェクトのスタック・トレースを標準エラー・ストリームに出力します。出力のために、java.io.PrintStreamオブジェクトまたはjava.io.PrintWriterオブジェクトも指定できます。

次のコード・フラグメントは、SQLの例外を捕捉してスタック・トレースを表示する方法を示しています。

```
try { <some code> }
catch(SQLException e) { e.printStackTrace (); }
```

JDBCドライバでエラーを処理する方法を説明するために、次のコードでは不適切な列索引を使用していると仮定します。

```
// Iterate through the result and print the employee names
// of the code

try {
  while (rset.next ())
      System.out.println (rset.getString (5));  // incorrect column index
}
```

```
catch(SQLException e) { e.printStackTrace (); }
```

列索引が不適切であると仮定すると、このプログラムの実行によって次のエラー・テキストが生成されます。

```
java.sql.SQLException: Invalid column index
at oracle.jdbc.OracleDriver.OracleResultSetImpl.getDate(OracleResultSetImpl.java:1556)
at Employee.main(Employee.java:41)
```

関連トピック

- JDBCエラー・メッセージ
- Oracle Databaseエラー・メッセージ・リファレンス

第II部 Oracle JDBC

この部では、Oracle Database 12cでサポートされている様々なバージョンのJava Database Connectivity(JDBC)について説明します。JDBC Thinドライバ、JDBC Oracle Call Interface(OCI)ドライバおよびサーバー側内部ドライバに固有の機能についても説明します。

第II部は、次の章で構成されます。

- JDBC標準のサポート
- Oracleの拡張機能
- JDBC Thin固有の機能
- JDBC OCIドライバ固有の機能
- サーバー側内部ドライバ

3 JDBC標準のサポート

Oracle Java Database Connectivity(JDBC)ドライバは、様々なバージョンのJDBC標準機能をサポートしています。Oracle Database 12cリリース2 (12.2.0.1)では、Oracle JDBCドライバが強化され、JDBC 4.1標準がサポートされました。これらの機能は、oracle.jdbcパッケージとoracle.sqlパッケージによって提供されます。これらのパッケージでは、Java Development Kit(JDK)リリース8がサポートされています。この章では、Oracle JDBCドライバでのJDBC標準のサポートについて説明します。内容は次のとおりです。

- JDBC 2.0標準のサポート
- JDBC 3.0標準のサポート
- JDBC 4.0標準のサポート
- JDBC 4.1標準のサポート
- JDBC 4.2標準のサポート

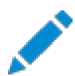
3.1 JDBC 2.0標準のサポート

今回のリリースのOracle JDBCドライバでは、JDK 1.2以降のバージョンを介してJDBC 2.0機能がサポートされています。考慮すべき点が3つあります。

- オブジェクト、配列およびラージ・オブジェクト(LOB)などのデータ型のサポート。これはjava.sqlパッケージを介して処理されます。
- 結果セットの拡張およびバッチ更新などの標準機能のサポート。これは、JDK 1.2.x以降でConnection、ResultSetおよびPreparedStatementなどの標準オブジェクトを介して処理されます。
- JDBC 2.0 Optional Packageの機能など、Standard Extension Application Program Interface(API)とも呼ばれる拡張機能に対するサポート。たとえばデータソース、接続プーリングおよび分散トランザクションなどがあげられます。

この項の内容は次のとおりです。

- データ型のサポート
- 標準機能のサポート
- 拡張機能のサポート
- JDBC2.0 Standard Extension APIとオラクル社独自のパフォーマンス強化API



ノート:

5.0 より前の JDK のバージョンは、もうサポートされていません。oracle.jdbc2 パッケージは削除されました。

3.1.1 データ型のサポート

Oracle JDBCは、標準java.sqlパッケージ内のインタフェースの実装による標準JDBC 2.0機能を含む、JDK 6およびJDK

7を完全にサポートしています。これらのインタフェースは、oracle.sqlパッケージとoracle.jdbcパッケージ内のクラスによって、適宜実装されます。

3.1.2 標準機能のサポート

ojdbc6.jarのJDBCクラスを使用するJDK 6.0環境では、スクロール可能結果セット、更新可能結果セット、バッチ更新などのJDBC 2.0機能は、標準のJDBC 2.0インタフェースによって指定されたメソッドを介してサポートされます。

3.1.3 拡張機能のサポート

データソース、接続プーリングおよび分散トランザクションなどのJDBC 2.0 Optional Packageの各機能は、JDK 1.2.x以上の環境でサポートされます。

標準javax.sqlパッケージとそのインタフェースを実装するクラスは、Oracle Databaseでパッケージ化されたJavaアーカイブ(JAR)ファイルに格納されています。

3.1.4 JDBC2.0 Standard Extension APIとオラクル社独自のパフォーマンス強化API

フェッチ・サイズまたは行のプリフェッチは、以前はOracle拡張機能でしか使用できませんでしたが、現在はJDBC 2.0で使用できます。標準モデルまたはOracleモデルを使用するオプションがあります。可能なかぎりJDBC標準モデルの使用をお薦めします。ただし、この機能のために、1つのアプリケーション内で標準モデルとOracleモデルを同時に使用しないでください。

関連トピック

- 行フェッチ・サイズ

3.2 JDBC 3.0標準のサポート

Oracle Database 12cリリース1のJDBCドライバでは、JDK 1.4以降のバージョンを介して標準JDBC 3.0機能がサポートされています。次の表では、今回のリリースのOracle JDBCドライバでサポートしているJDBC 3.0機能と、各機能に関する詳細の参照先を示しています。

表3-1 JDBC 3.0機能の主な領域

機能	コメントと参照
トランザクションのセーブポイント	詳細は、「トランザクション・セーブポイントの概要」を参照してください。
文キャッシュ	接続プールによるプリペアド文の再利用「文キャッシュと結果セット・キャッシュ」を参照してください。
ローカル・トランザクションとグローバル・トランザクションの切	「ローカル・トランザクションとグローバル・トランザクションの切替えについて」を参照してください。

機能	コメントと参照
替え	
LOB の変更	「JDBC 3.0 の LOB インタフェース・メソッド」 JDBC 3.0 の LOB インタフェース・メソッドを参照してください。
名前付き SQL パラメータ	「Interface oracle.jdbc.OracleCallableStatement」および「Interface oracle.jdbc.OraclePreparedStatement」 Interface oracle.jdbc.OraclePreparedStatement を参照してください。
RowSet	「JDBC RowSet」を参照してください
自動生成キーの取出し	「自動生成キーの取出し」 自動生成キーの取出しを参照してください
結果セットの保持機能	「結果セットの保持機能」 結果セットの保持機能を参照してください

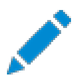
この項では、Oracle JDBCドライバによってサポートされている次のJDBC 3.0機能について説明します。

- トランザクション・セーブポイントの概要
- 自動生成キーの取出し
- JDBC 3.0のLOBインタフェース・メソッド
- 結果セットの保持機能

3.2.1 トランザクション・セーブポイントの概要

JDBC 3.0の仕様はセーブポイントをサポートしており、トランザクション内でより細かい境界設定をすることができます。アプリケーションはトランザクション内にセーブポイントを設定して、セーブポイントより後に行われたすべての作業をロールバックすることができます。セーブポイントを使用することで、トランザクションの原子性が緩和されます。セーブポイントを持つトランザクションは、トランザクションのコンテキスト外部では1つの単位のように見えるという点で原子的ですが、トランザクション内で動作しているコードは部分的な状態を維持することができます。

ノート:



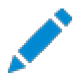
セーブポイントがサポートされるのは、ローカル・トランザクションのみです。グローバル・トランザクション内でセーブポイントを指定すると、SQLException 例外が発生します。

3.2.1.1 セーブポイントの作成について

セーブポイントを作成するには、java.sql.Savepointインスタンスを戻すConnection.setSavepointを使用します。

セーブポイントには名前がある場合とない場合があります。セーブポイントの名前を指定するには、setSavepointメソッドに文

字列を指定します。名前が未指定の場合は、整数のIDが割り当てられます。名前の取出しにはgetSavepointNameメソッドを使用します。IDの取出しにはgetSavepointIdメソッドを使用します。



ノート:

名前付けされていないセーブポイントから名前を取り出そうとしたり、名前付けされているセーブポイントから ID を取り出そうとしたりすると、SQLException 例外がスローされます。

3.2.1.2 セーブポイントまでのロールバックについて

セーブポイントまでロールバックするには、Connection.rollback(Savepoint svpt)メソッドを使用します。解放されているセーブポイントまでロールバックしようとすると、SQLException例外がスローされます。

3.2.1.3 セーブポイントの解放について

セーブポイントを削除するには、Connection.releaseSavepoint(Savepoint svpt)メソッドを使用します。

3.2.1.4 セーブポイント・サポートのチェックについて

使用中のデータベースでセーブポイントがサポートされているかどうかを調べるには、oracle.jdbc.OracleDatabaseMetaData.supportsSavepointsメソッドをコールします。セーブポイントが使用可能な場合はtrue、そうでない場合はfalseが戻されます。

3.2.1.5 セーブポイントに関するノート

セーブポイントを使用する場合は、次の点に注意する必要があります。

- セーブポイントの解放後、ロールバック操作でそのセーブポイントを参照しようとすると、SQLException例外がスローされます。
- トランザクションがコミットまたはロールバックされると、そのトランザクション内の作成済セーブポイントはすべて自動的に解放され、無効になります。
- あるセーブポイントまでトランザクションをロールバックすると、そのセーブポイントより後に作成されたセーブポイントはすべて自動的に解放され、無効になります。

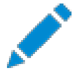
3.2.2 自動生成キーの取出し

多くのデータベース・システムでは、行が挿入される際に一意のキー・フィールドが自動的に生成されます。Oracle Databaseでは、順序とトリガーにより、この機能が提供されます。JDBC 3.0には、自動生成キーの取出し機能が導入され、生成された値を取り出せるようになりました。JDBC 3.0では、自動生成キーの取出し機能をサポートするために、次のインタフェースが拡張されました。

- java.sql.DatabaseMetaData
- java.sql.Connection
- java.sql.Statement

これらのインタフェースには、自動生成キーの取出しをサポートするメソッドが用意されています。ただし、この機能はINSERT文が

処理される場合にのみサポートされます。その他のデータ操作言語(DML)文の処理では、自動生成キーの取出しは実施されません。



ノート:

Oracle のサーバー側内部ドライバでは、自動生成キーの取出し機能をサポートしていません。

3.2.2.1 java.sql.Statement

キー列を明示的に指定しないと、Oracle JDBCドライバでは取り出す列を特定できません。Oracle JDBCドライバで自動生成キーを含む列を特定できるのは、列名または列索引の配列が使用されている場合です。ただし、int型のStatement.RETURN_GENERATED_KEYSフラグが使用されている場合、Oracle JDBCドライバではこれらの列を特定できません。int型フラグを使用して自動生成キーが戻るように指定した場合は、ROWID擬似列がキーとして戻ります。このROWIDは、ResultSetオブジェクトからフェッチして、他の列を取り出すために使用できます。

3.2.2.2 サンプル・コード

次のコードは、自動生成キーの取出しを示しています。

```
/** SQL statements for creating an ORDERS table and a sequence for generating the
  * ORDER_ID.
  *
  * CREATE TABLE ORDERS (ORDER_ID NUMBER, CUSTOMER_ID NUMBER, ISBN NUMBER,
  * DESCRIPTION NCHAR(5))
  *
  * CREATE SEQUENCE SEQ01 INCREMENT BY 1 START WITH 1000
  */
...
String cols[] = {"ORDER_ID", "DESCRIPTION"};
// Create a PreparedStatement for inserting a row into the ORDERS table.
OraclePreparedStatement pstmt = (OraclePreparedStatement)
conn.prepareStatement("INSERT INTO ORDERS (ORDER_ID, CUSTOMER_ID, ISBN,  DESCRIPTION) VALUES
(SEQ01.NEXTVAL, 101,
 966431502, ?)", cols);
char c[] = {'a', '¥u5185', 'b'};
String s = new String(c);

pstmt.setNString(1, s);
pstmt.executeUpdate();
ResultSet rset = pstmt.getGeneratedKeys();
...
```

前の例では、シーケンス(SEQ01)がORDER_ID列の値を生成するために作成されます。値は、1000から始まり、シーケンスが処理されて次の値が生成されるたびに1ずつ増加します。OraclePreparedStatementオブジェクトは、ORDERS表に行を挿入するために作成されています。

3.2.2.3 自動生成キーの制約

自動生成キーはDML RETURNING句を使用して実装されます。したがって、getGeneratedKeysメソッドから戻されるResultSetオブジェクトには位置によってのみアクセスする必要があります。ResultSetオブジェクトの列としてはバインド変数名

を使用しないでください。

3.2.3 JDBC 3.0のLOBインタフェース・メソッド

次の表では、Oracle独自のメソッドとJDBC 3.0標準メソッドとの間の変換を示しています。

表3-2 等価のBLOBメソッド

Oracle独自のメソッド	JDBC 3.0標準メソッド
putBytes(long pos, byte [ ] bytes)	setBytes(long pos, byte[] bytes)
putBytes(long pos, byte [ ] bytes, int length)	setBytes(long pos, byte[] bytes, int offset, int len)
getBinaryOutputStream(long pos)	setBinaryStream(long pos)
trim (long len)	truncate(long len)

表3-3 等価のCLOBメソッド

Oracle独自のメソッド	JDBC 3.0標準メソッド
putString(long pos, String str)	setString(long pos, String str)
該当なし	setString(long pos, String str, int offset, int len)
getAsciiOutputStream(long pos)	setAsciiStream(long pos)
getCharacterOutputStream(long pos)	setCharacterStream(long pos)
trim (long len)	truncate(long len)

3.2.4 結果セットの保持機能

結果セットの保持機能は、JDBC 3.0から導入されました。この機能は、コミット操作が実行されるとき、ResultSetオブジェクトがオープンされているべきか、クローズされているべきかをアプリケーションが判断できるようにします。コミット操作は暗黙的でも明示的でもかまいません。

Oracle DatabaseでサポートしているのはHOLD_CURSORS_OVER_COMMITのみです。したがって、これが、Oracle JDBCドライバのデフォルト値です。保持機能を変更しようとすると必ず、SQLFeatureNotSupportedException例外がスローされます。

3.3 JDBC 4.0標準のサポート

Oracle Databaseリリース18cのJDBCドライバでは、JDBC 4.0標準がサポートされています。

ノート:

JDBC 4.0 仕様では、java.sql.Array オブジェクトを作成する java.sql.Connection.createArrayOf ファクトリ・メソッドが定義されています。createArrayOf メソッドは配列要素型の名前を引数の 1 つに取ります。ここで、配列のタイプは匿名です。Oracle Database では、匿名の配列型でなく、名前付きの配列型のみがサポートされます。したがって、現在のリリースの Oracle JDBC ドライバは createArrayOf メソッドをサポートしていません。配列型を作成するには、Oracle 固有の createARRAY メソッドを使用する必要があります。

関連項目:

- createArrayOfメソッドの詳細は、「ARRAYオブジェクトの作成」を参照してください。
- このドキュメントはこれらの新機能の概要のみ示しているため、これらの機能の詳細は、次のページを参照してください

  http://docs.oracle.com/javase/6/docs/

Oracle Databaseリリース18cのJDBCドライバでは、次のような機能を利用できます。

- ラッパー・パターンのサポート
- SQLXML型
- 機能拡張された例外階層とSQLException
- ROWIDデータ型
- LOBの作成
- 各国語文字セットのサポート

3.3.1 ラッパー・パターンのサポート

ラッパー・パターンは、Javaアプリケーションで使用される一般的なコーディング・パターンで、データソース固有の従来のJDBC APIを超える機能を提供します。実際のリソースを表すプロキシ・クラス・インスタンスとしてラップされているリソースにアクセスする場合に、これらの拡張機能を使用する必要があることがあります。JDBC4.0では、リソース代理への直接アクセスを許可するために、プロキシによって表されるこれらのラップされたリソースにアクセスするための標準メカニズムを記述するWrapperインタフェースが導入されています。

Wrapperインタフェースでは、次の2つのメソッドが提供されています。

- public boolean isWrapperFor(Class<?> iface) throws SQLException;
- public <T> T unwrap(Class<T> iface) throws SQLException;

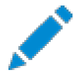
SQLデータを表すものを除く他のJDBC4.0インタフェースはすべてこのインタフェースを実装しています。これには、Connection、Statementとそのサブタイプ、ResultSetおよびメタデータ・インタフェースが含まれます。

関連項目:

http://docs.oracle.com/javase/7/docs/api/java/sql/Wrapper.html

3.3.2 SQLXML型

JDBC 4.0標準で最も重要な更新内容の1つはSQL 2003標準で定義されたXMLデータ型のサポートです。現在、JDBCは、SQL/XMLデータベース・データ型をサポートするためのマッピング・インタフェースを提供しています(java.sql.SQLXML)。この新しいJDBCインタフェースは、XMLの、Javaネイティブのバインディングを定義し、それにより、データベースXMLデータの処理をより簡単、より効率的にしています。



ノート:

- また、SQLXML 型のデータを使用するには、classpath 環境変数に xdb6.jar ファイルとxmlparserv2.jar ファイルを指定する必要があります(classpath にすでに指定されている場合を除く)。
- SQLXML は、CachedRowset オブジェクトではサポートされません。

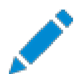
java.sql.ConnectionインタフェースでcreateSQLXMLメソッドをコールすることによって、XMLのインスタンスを作成することができます。このメソッドは、空のXMLオブジェクトを戻します。

次に示すように、PreparedStatement、CallableStatementおよびResultSetインタフェースは拡張され、適切なgetterメソッドとsetterメソッドが用意されました。

- PreparedStatement: メソッドsetSQLXMLが追加されました。
- CallableStatement: メソッドgetSQLXMLおよびsetSQLXMLが追加されました。
- ResultSet: メソッドgetSQLXMLが追加されました。

ノート:

Oracle Database 10g と、Oracle Database 11g の初期バージョンでは、Oracle JDBC ドライバは、Oracle の独自規格の拡張機能を介して Oracle SQL XML 型(XMLType)をサポートしていました。これは、JDBC 標準に準拠していませんでした。



11.2.0.2 の Oracle JDBC ドライバでは新しい接続プロパティ(oracle.jdbc.getObjectReturnsXMLType)が導入され、JDBC 標準に準拠しています。このプロパティに false を設定した場合、getObject メソッドは java.sql.SQLXML のデータ型のインスタンスを戻し、oracle.xdb.XMLType を使用する既存の Oracle 独自規格の SQL XMLType サポートに依存している場合、このプロパティの値を true に戻すことができます。

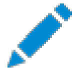
ただし、getObjectReturnsXMLType プロパティの設定は、現在のバージョンの Oracle JDBC ドライバでは必要ありません。

例

例3-1 SQLXMLデータへのアクセス

次の例は、StringからXMLのインスタンスを作成し、データベースにXMLデータを書き込み、データベースからXMLデータを取得する方法を示しています。

```
import java.sql.*;
import java.util.Properties;
import oracle.jdbc.pool.OracleDataSource;

public class SQLXMLTest
 {

  public static void main(String[] args)
  {

  Connection conn = null;
  Statement stmt = null;
  ResultSet rs = null;
  PreparedStatement ps = null;

  String xml = "<?xml version=¥"1.0¥"?>¥n" +
    "<oldjoke>¥n" +
    "<burns>Say <quote>goodnight</quote>, Gracie.</burns>¥n" +
    "<allen><quote>Goodnight, Gracie.</quote></allen>¥n" +
    "<applause/>¥n" +
    "</oldjoke>";

  try
  {

     OracleDataSource ods = new OracleDataSource();
     ods.setURL("jdbc:oracle:thin:@//localhost:5221/orcl");
     ods.setUser("HR");
     ods.setPassword("hr");
     conn = ods.getConnection();

     ps = conn.prepareStatement("insert into x values (?, ?)");
     ps.setString(1, "string to string");
     SQLXML x = conn.createSQLXML();
     x.setString(xml);
     ps.setSQLXML(2, x);
     ps.execute();
     stmt = conn.createStatement();
     rs = stmt.executeQuery("select * from x");
     while (rs.next())
     {
       x = rs.getSQLXML(2);
       System.out.println(rs.getString(1) + "¥n" + rs.getSQLXML(2).getString());
       x.free();
     }

     rs.close();
     ps.close();
  }

  catch (SQLException e){e.printStackTrace ();}

  }
}
```



ノート:

空の XML を指定して setter メソッドをコールすると、SQLException がスローされます。getter メソッドが空の XML を戻すことはありません。

3.3.3 機能拡張された例外階層とSQLException

JDBC 3.0では、SQLExceptionという単一の例外しかサポートされていません。ただし、エラーはカテゴリが多いので、区別すると便利です。この機能では、様々なカテゴリのエラーを特定するために、SQLExceptionクラスのサブクラスがサポートされています。永続エラーと一時エラーの切分けが、主要な切分けになります。永続エラーは、システムの正しい動作の結果であり、常に発生します。一時エラーは、タイムアウトなど、システムの一部で障害が発生した結果であり、再発しない場合もあります。

JDBC 4.0では、一時エラー、永続エラーおよびそれらエラーの様々なカテゴリを表現するために、例外が追加されています。

また、SQLExceptionクラスとそのサブクラスは、J2SE関連の例外機能をサポートするために機能拡張されています。

3.3.4 ROWIDデータ型

JDBC 4.0では、SQLのROWID値を表現するために、java.sql.RowIdデータ型がサポートされています。ResultSetインタフェースおよびCallableStatementインタフェースで定義されているgetterメソッドを使用すると、RowId値を取り出すことができます。開発者は、パラメータ化されたPreparedStatementの中でRowId値を使用して、RowIdオブジェクトのパラメータを設定したり、更新可能な結果セットの中で使用して、列を特定のRowId値に更新したりできます。

RowIdオブジェクトは、指定された行が削除されるまで有効です。RowIdオブジェクトは、次の場合にも有効なことがあります。

- 作成元トランザクションの期間中。
- 作成元セッションの期間中。
- 永続的に有効である期間(未定義)。

DatabaseMetaData.getRowIdLifetimeメソッドをコールすると、RowIDオブジェクトの存続期間を確認できます。

3.3.5 LOBの作成

JDBC 4.0では、BLOB、CLOBおよびNCLOBオブジェクトの作成をサポートするために、Connectionインタフェースが機能拡張されています。このインタフェースでサポートされているcreateBlob、createClobおよびcreateNClobメソッドを使用すると、Blob、ClobおよびNClobオブジェクトを作成できます。

作成されたLOB(Large OBject)にデータは含まれません。java.sql.Blob、java.sql.Clobおよびjava.sql.NClobインタフェースで使用可能なAPIをコールして、これらのオブジェクトにデータを追加または取得できます。これらのオブジェクトからは、内容の全体または一部を取り出すことができます。次のコードでは、BLOBオブジェクトのオフセット200の位置から100バイトのデータを取り出す方法を示しています。

```
...
Connection con = DriverManager.getConnection(url, props);
Blob aBlob = con.createBlob();
```

```
// Add data to the BLOB object.
aBlob.setBytes(...);
...
// Retrieve part of the data from the BLOB object.
InputStream is = aBlob.getBinaryStream(200, 100);
...
```

setBlob、setClobおよびsetNClobメソッドを使用して、PreparedStatementオブジェクトにLOBを入力パラメータとして渡すこともできます。updateBlob、updateClobおよびupdateNClobメソッドを使用すると、更新可能結果セット内の列値を更新できます。

これらのLOBは一時LOBで、一時LOBを使用する必要がある場合に使用します。データベース内の記憶域を永続的にするには、これらのLOBを表に書き込む必要があります。

関連項目:

「一時LOBの使用について」

一時LOBは、作成元のトランザクションの期間中は少なくとも有効です。ただし、トランザクションの実行期間が長いと、メモリーの使用が保証されない場合があります。次のようにLOBのfreeメソッドをコールすると、LOBを解放できます。

```
...
Clob aClob = con.createClob();
int numWritten = aClob.setString(1, val);
aClob.free();
...
```

3.3.6 各国語文字セットのサポート

JDBC 4.0では、各国語文字セットの型にアクセスするためのNCHAR、NVARCHAR、LONGNVARCHARおよびNCLOBというJDBC型が導入されています。これらの型は、各国語文字セットを使用して値がエンコードされる点を除き、CHAR、VARCHAR、LONGVARCHARおよびCLOB型に類似しています。

3.4 JDBC 4.1標準のサポート

Oracle Database 12cリリース1のJDBCドライバでは、JDK 7を介してJDBC 4.1標準がサポートされています。この項では、JDBC 4.1仕様の次の重要なメソッドについて説明します。

- setClientInfoメソッド
- getObjectメソッド

3.4.1 setClientInfoメソッド

データベース・リソースの消費の監視については、setClientInfoメソッドを使用して、指定した時点にデータベースを使用する様々なアプリケーション・タスクを識別できます。setClientInfoメソッドは、様々なアプリケーション情報を提供するプロパティの値を設定します。このメソッドは、<namespace>.<keyname>の形式のキーを受け入れます。たとえば、次のコードに示すように

setClientInfoメソッドを使用してACTIONキー、MODULEキーおよびCLIENTIDキー(V$SESSIONビューおよび多くのパフォーマンス・ビューにあり、トレース・ファイルでレポート可能)を使用できます。

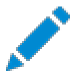
```
// "conn" is an instance of java.sql.Connection:
conn.setClientInfo("OCSID.CLIENTID", "Alice_HR_Payroll");
conn.setClientInfo("OCSID.MODULE", "APP_HR_PAYROLL");
conn.setClientInfo("OCSID.ACTION", "PAYROLL_REPORT");
```

setClientInfoメソッドでは、Java権限oracle.jdbc.clientInfoをチェックします。このセキュリティ・チェックが失敗した場合、SecurityExceptionがスローされます。これは、<namespace>.*の形式の権限の名前パターンをサポートしています。setClientInfoメソッドはすべてのペアの設定またはクリアを行います。したがって、権限の名前にアスタリスク(*)を設定する必要があります。

JDBCドライバでは、<namespace>.<keyname>の組合せをサポートしています。setClientInfoメソッドでは、他のネームスペースではOCSIDネームスペースをサポートしています。ただし、OCSIDネームスペースと他のネームスペースの使用方法には違いがあります。OCSIDネームスペースの場合、setClientInfoメソッドでは次のキーのみをサポートしています。

- ACTION
- CLIENTID
- ECID
- MODULE
- SEQUENCE_NUMBER
- DBOP

また、他のネームスペースに関連付けられた情報は単一のプロトコルを使用してネットワークで通信されますが、OCSIDネームスペースに関連付けられた情報は別のプロトコルを使用して通信されます。OCSIDネームスペースに使用されるプロトコルは、エンドツーエンドのメトリック値を送信するために、OCI Cライブラリと10gのJDBC Thinドライバおよび以降のThinドライバでも使用されます。



ノート:

- setClientInfo メソッドは setEndToEndMetrics および setClientIdentifier メソッドとの下位互換性があり、DMS を使用してクライアント・タグを設定できます。
- setEndToEndMetrics メソッドは、Oracle Database 12c リリース 1 (12.1)では非推奨になりました。

データベース操作の監視について

多くのJavaアプリケーションにはデータベース接続機能がありませんが、その機能のかわりにデータベース・アクティビティを追跡する必要があります。このようなアプリケーションのために、Oracle Database 12cリリース1 (12.1)では、DBOPタグが導入されました。これは、アプリケーションがデータベースに明示的にアクセスできない場合に、アプリケーションのスレッドに関連付けることができます。DBOPタグは、DMS APIを起動してスレッドと関連付けられます。データベースへのアクティブな接続は不要です。スレッドが次のデータベース・コールを送信したとき、DMSはデータベース・コールとともに接続を介してこれらのタグを伝播します。追加のラウンドトリップの必要はありません。このように、アプリケーションは、アプリケーション・レイヤーのコードを分析しながら、そのアク

ティビティをデータベースの動作に関連付けることができます。DBOPタグは次の項目で構成されています。

- データベースの動作の名前
- 実行ID
- 動作属性

setClientInfoメソッドでは、DBOPタグをサポートしています。setClientInfoメソッドは、データベース動作を監視するために、このタグの値を設定します。JDBCアプリケーションがデータベースに接続し、データベース・ラウンドトリップが行われると、データベース・アクティビティを追跡できます。たとえば、次のようにDBOPタグの値にfooを設定できます。

```
...
Connection conn = DriverManager.getConnection(myUrl, myUsername, myPassword);
conn.setClientInfo("E2E_CONTEXT.DBOP", "foo");
Statement stmt = conn.createStatement();
stmt.execute("select 1 from dual"); // DBOP tag is set after this
...
```

3.4.2 getObjectメソッド

getObjectメソッドは、渡されたパラメータに基づいて、オブジェクトを取得します。Oracle Database 12cリリース2 (12.2.0.1)では、次の2つの新規getObjectメソッドをサポートしています。

メソッド1

```
<T> T getObject(intparameterIndex,
                java.lang.Class<T>type)
            throws SQLException
```

メソッド2

```
<T> T getObject(java.lang.StringparameterName,
                java.lang.Class<T>type)
            throws SQLException
```

これらのメソッドでは、JDBC仕様に示された変換と表A-1に示された追加の変換をサポートしています。Oracle Database 12cリリース2 (12.2.0.1)のドライバでは、一部の追加クラスへの変換もサポートしています。これは、次の条件のいずれかが満たされている場合、1つ以上の静的なvalueOfメソッドを実装しています。

- JDBC仕様または表A-1の他の変換が指定されていない。
- type引数で、valueOfという名前の単一のパブリックで静的な引数メソッドを1つ以上定義している。
- 1つ以上のvalueOfメソッドがJDBC仕様または表A-1のためにサポートされている型の値である引数を取る。

今回のリリースのJDBCドライバは、JDBC仕様または表A-1で指定された型に値を変換してから、変換した値を引数として、対応するvalueOfメソッドをコールします。適切なvalueOfメソッドが複数ある場合、JDBCドライバはvalueOfメソッドを1つ選択します。この方法は指定されていません。

例

```
ResultSet rs = . . . ;
Character c = rs.getObject(1, java.lang.Character.class);
```

Characterクラスは、次のvalueOfメソッドを定義します。

```
public static Character valueOf(char c);
```

表A-1では、NUMBERをcharに変換できることが示されています。したがって、ResultSetの最初の列がNUMBERである場合、getObjectメソッドはそのNUMBER値をcharに変換し、そのchar値をvalueOf(char)メソッドに渡し、結果のCharacterオブジェクトを戻します。

3.5 JDBC 4.2標準のサポート

Oracle Database 12cリリース2 (12.2.0.1)のJDBCドライバでは、JDK 8を介してJDBC 4.2標準がサポートされています。この項では、今回のリリースに追加された重要なメソッドの一部について説明します。

%Large%メソッド

今回のリリースのOracle JDBCドライバでは、long値に対応する、JDBC 4.2標準で導入された次のメソッドがサポートされています。

- executeLargeBatch()
- executeLargeUpdate(String sql)
- executeLargeUpdate(String sql, int autoGeneratedKeys)
- executeLargeUpdate(String sql, int[] columnIndexes)
- executeLargeUpdate(String sql, String[] columnNames)
- getLargeMaxRows()
- getLargeUpdateCount()
- setLargeMaxRows(long max)

これらの新しいメソッドは、java.sql.Statementインタフェースの一部として使用できます。%Large%メソッドは、int値のかわりにlong値を使用すること以外は、対応するnon-largeメソッドと同様です。たとえば、executeUpdateメソッドはint値として更新された行数を戻しますが、executeLargeUpdateメソッドはlong値として更新された行数を戻します。行数がInteger.MAX_VALUEの値より大きい場合、アプリケーションではexecuteLargeUpdateメソッドを使用する必要があります。

次のコードでは、executeLargeUpdate(String sql)メソッドを使用する方法を示します。

```
...
Statement stmt = conn.createStatement();
stmt.executeQuery("create table BloggersData (FIRST_NAME varchar(100), ID int)");
long updateCount = stmt.executeLargeUpdate("insert into BloggersData (FIRST_NAME,ID)
values('John',1)");
...
```

SQLTypeメソッド

今回のリリースのOracle JDBCドライバでは、SQLTypeパラメータを使用する、JDBC 4.2標準で導入された次のメソッドがサポートされています。

- setObject

  setObjectメソッドでは、指定したオブジェクトに指定したパラメータの値を設定します。このメソッドは、

setObject(int parameterIndex, Object x, SQLType targetSqlType, int scaleOrLength)に似ていますが、scaleに0を仮定している点が異なります。このメソッドのデフォルト実装はSQLFeatureNotSupportedExceptionをスローします。

```
void setObject(int parameterIndex, java.lang.Object x, SQLType targetSqlType) throws
SQLException
```

ここで、

parameterIndexは指定されたパラメータの索引で、最初のパラメータは1、2番目のパラメータは2のようになります

xは、入力パラメータ値を含むオブジェクトです

targetSqlTypeは、データベースに送られるSQL型です

- updateObject

  updateObjectメソッドでは、パラメータとして列索引を使用し、指定した列をオブジェクト値で更新します。

- registerOutParameter

  registerOutParameterメソッドでは、JDBC型をSQLTypeとするように指定したパラメータを登録します。

次のコードでは、setObjectメソッドを使用する方法を示します。

```
...
int empId = 100;
connection.prepareStatement("SELECT FIRST_NAME, LAST_NAME FROM EMPLOYEES WHERE EMPNO = ?");
preparedStatement.setObject(1, Integer.valueOf(empId), OracleType.NUMBER);
...
```

関連トピック

- https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/jdbc_42.html

4 Oracle拡張機能

Oracleには、Java Database Connectivity(JDBC)標準の実装を拡張するJavaクラスとインタフェースが用意されており、開発者は、Oracleデータ型にアクセスして操作し、Oracleパフォーマンス拡張機能を使用できます。この章では、JDBC標準実装を拡張するためにOracleで提供されるクラスとインタフェースの概要を示します。また、この拡張機能がサポートする主な機能についても説明します。

この章の構成は、次のとおりです。

- Oracle拡張機能の概要
- Oracle拡張機能
- Oracle JDBCパッケージ
- Oracle文字データ型のサポート
- その他のOracle型拡張機能
- DML RETURNING
- PL/SQL連想配列へのアクセス

関連トピック

- パフォーマンス拡張機能

4.1 Oracle拡張機能の概要

標準機能に加えて、Oracle JDBCドライバにはOracle固有型拡張機能およびパフォーマンス拡張機能があります。これらの拡張機能は、次のJavaパッケージに用意されています。

- `oracle.sql`

  Oracle形式でSQLデータを示すクラスを提供します。

- `oracle.jdbc`

  Oracle型形式でのデータベース・アクセスおよび更新をサポートするためのインタフェースを提供します。

関連トピック

- Oracle JDBCパッケージ

4.2 Oracle拡張機能

JDBCへのOracle拡張機能には、Oracle Databaseを操作する能力を高めるいくつかの機能があります。次のようなものがあります:

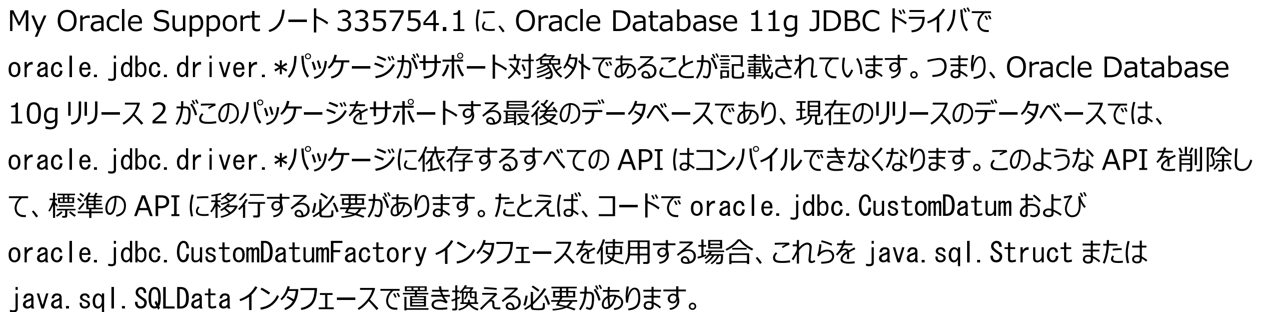
- JDBCを使用したデータベース管理
- Oracleデータ型のサポート
- Oracleオブジェクトのサポート

- スキーマの命名サポート
- DML RETURNING
- PL/SQL連想配列へのアクセスについて

4.2.1 JDBCを使用したデータベース管理

Oracle Database 11gリリース1以降、2つのJDBCメソッド、startupおよびshutdownがoracle.jdbc.OracleConnectionインタフェースに追加されており、これによりOracle Databaseインスタンスの起動と停止を実行できます。

ノート:



My Oracle Support ノート 335754.1 に、Oracle Database 11g JDBC ドライバで oracle.jdbc.driver.*パッケージがサポート対象外であることが記載されています。つまり、Oracle Database 10g リリース 2 がこのパッケージをサポートする最後のデータベースであり、現在のリリースのデータベースでは、oracle.jdbc.driver.*パッケージに依存するすべての API はコンパイルできなくなります。このような API を削除して、標準の API に移行する必要があります。たとえば、コードで oracle.jdbc.CustomDatum および oracle.jdbc.CustomDatumFactory インタフェースを使用する場合、これらを java.sql.Struct または java.sql.SQLData インタフェースで置き換える必要があります。

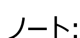
関連トピック

- データベース管理

4.2.2 Oracleデータ型のサポート

Oracle JDBC拡張機能の1つは、oracle.sqlパッケージの型のサポートです。このパッケージには、Oracle形式でのデータの正確な表現であるクラスが含まれています。プログラムでoracle.sql型を使用する際には、次の重要な点に注意してください。

- 数値型データの場合、標準Java型への変換では、データ変換プロセスの制限のために完全な精度が維持されません。データ損失の問題を最小限に抑えるには、BigDecimal型を使用してください。
- 特定のデータ型では、標準Java型への変換がシステムの設定によって決まる可能性があり、プログラムが予想どおりに実行されない場合があります。これは、oracle.sql型から標準Java型へのデータ変換中に発生することが知られている制限です。
- プログラムの機能が、1つの表からのデータの読取りと、もう1つの表への同じデータの書込みに限定されている場合、数値および日付データについては、標準Java型と比べてoracle.sql型の方がわずかに速くなります。しかし、比較や印刷などの単純なデータ操作が1つでも含まれている場合は、標準Java型の方が速くなります。
- oracle.sql.CHARはOracle形式に従うデータの正確な表現ではありません。oracle.sql.CHARはjava.lang.Stringから構成されます。java.lang.Stringの方が常に高速で、同じ文字セットを表すため、いくつかのサポートされていない文字セットを除いて、oracle.sql.CHARを使用する利点がありません。



ノート:

標準の Java 型を使用して、既存の oracle.sql 型のデータを標準の Java 型に変換することを強くお薦めします。内部的には、Oracle JDBC ドライバは Java 標準型のパフォーマンスを最適化するように動作します。oracle.sql 型は、下位互換性のためにのみサポートされており、使用は推奨されません。

関連トピック

- パッケージoracle.sql
- Oracle文字データ型のサポート
- その他のOracle型拡張機能

4.2.3 Oracleオブジェクトのサポート

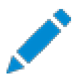
Oracle JDBCはデータベース内の構造化オブジェクトの使用をサポートしています。ここで、オブジェクトのデータ型はネスト属性を持つユーザー定義の型です。たとえば、ユーザー・アプリケーションがEmployeeオブジェクト型を定義し、各Employeeオブジェクトが、firstname属性(文字列)、lastname属性(文字列)およびemployeenumber属性(整数値)を持つ例が考えられます。

Oracle JDBCは、Oracleオブジェクト・データ型をサポートします。JavaアプリケーションでOracleオブジェクト・データ型を使用する場合、次の点を考慮する必要があります。

- Oracleオブジェクト・データ型とJavaクラス間のマップ方法
- Oracleオブジェクトの属性を対応するJavaオブジェクトに格納する方法
- SQLとJava形式間で属性データを変換する方法
- データへのアクセス方法

Oracleオブジェクトは、弱い型指定のjava.sql.Struct、または強い型指定のカスタマイズされたクラスのいずれかにマップできます。これらの強い型はカスタムJavaクラスとして参照され、標準java.sql.SQLDataインタフェースまたはOracle拡張機能のoracle.jdbc.OracleDataインタフェースのいずれかを実装している必要があります。各インタフェースは、SQLとJavaとの間でデータ変換を行うメソッドを指定します。

ノート:



Oracle Database 12c リリース 1 (12.1)以降、OracleData インタフェースは ORAData インタフェースに置き換わりました。

Oracleオブジェクトに対応するカスタムJavaクラスを作成する場合は、Oracle JVM Webサービス・コールアウト・ユーティリティを使用することをお薦めします。

関連トピック

- Oracleオブジェクト型の操作
- Oracle Database Java開発者ガイド

4.2.4 スキーマの命名サポート

Oracleオブジェクト・データ型クラスには、完全修飾スキーマ名の受入れおよび復帰を行う機能があります。完全修飾スキーマ

名は、次の構文になります。

```
{[schema_name].}[sql_type_name]
```

ここで、schema_nameはスキーマの名前、sql_type_nameはオブジェクトのSQL型名で、schema_nameとsql_type_nameはピリオド(.)で区切られています。

JDBCでオブジェクト型を指定するには、その完全修飾名を使用します。型名が現行ネームスペース内、つまり現行スキーマ内にある場合は、スキーマ名を入力する必要はありません。スキーマは、次の規則に従って命名されます。

- スキーマ名と型名は、どちらも引用符で囲んでも囲まなくてもかまいません。ただし、CORPORATE.EMPLOYEEのように、SQL型名にピリオドが含まれている場合は、型名を引用符で囲む必要があります。
- JDBCドライバは、オブジェクト名内の引用符で囲まれていない最初のピリオドを検索して、ピリオドの前の文字列をスキーマ名として使用し、ピリオドの後の文字列を型名として使用します。ピリオドが見つからない場合、JDBCドライバは現行スキーマをデフォルトとします。つまり、オブジェクト型名が現行スキーマに属している場合は、完全修飾名を指定するかわりに、スキーマを指定せずに型名のみを指定できます。これが、型名にピリオドが含まれない場合に型名を引用符で囲む理由です。

  たとえば、ユーザーHRがperson.addressという型を作成し、自分のセッション内でそれを使用するとします。HRは、スキーマ名を省略して、person.addressで型をJDBCドライバに渡します。この場合、person.addressを引用符で囲まないとピリオドが検出され、JDBCドライバはpersonをスキーマ名、addressを型名として誤って解釈してしまいます。
- JDBCは、オブジェクト型名の文字列をデータベースにそのまま渡します。つまり、JDBCドライバは、オブジェクト型名が引用符で囲まれていてもその大/小文字を変更しません。

  たとえば、HR.PersonTypeがJDBCドライバにオブジェクト型名として渡されると、JDBCドライバはその文字列をそのままデータベースに渡します。別の例として、型名の文字列内に空白文字が含まれている場合、JDBCドライバは空白文字を削除しません。

4.2.5 DML RETURNING

Oracle Databaseでは、データ操作言語(DML)文にRETURNING句を使用できます。これによって、2つのSQL文を1文に結合できます。DML RETURNINGは、Oracle JDBC Oracle Call Interface(OCI)ドライバとOracle JDBC Thinドライバの両方でサポートされます。

関連項目:

「DML RETURNING」

4.2.6 PL/SQL連想配列

Oracle JDBCドライバを使用すると、JDBCアプリケーションで、連想配列パラメータを使用してPL/SQLをコールできます。Oracle JDBCドライバは、スカラー・データ型のPL/SQL連想配列をサポートします。

関連項目:

「PL/SQL連想配列へのアクセス」

4.3 Oracle JDBCパッケージ

この項では、Oracle JDBC拡張機能をサポートする、次のJavaパッケージについて説明します。

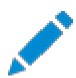
- パッケージoracle.sql
- パッケージoracle.jdbc

4.3.1 パッケージoracle.sql

oracle.sqlパッケージは、SQL形式でデータへの直接アクセスをサポートしています。このパッケージは、主に、SQLデータ型とそのサポート・クラスへのJavaマッピングを提供するクラスで構成されています。実質的に、このクラスはSQLデータのJavaコンテナとして機能します。

oracle.sql.*データ型の各クラスは、すべてのデータ型に共通する機能をカプセル化したスーパークラスのoracle.sql.Datumを拡張します。その中には、JDBC 2.0準拠のデータ型に対応したクラスもあります。これらのクラスは、oracle.sql.Datumクラスを拡張すると同時に、java.sqlパッケージの標準JDBC 2.0インタフェースを実装します。

LONGおよびLONG RAWのSQL型と、REF CURSOR型カテゴリには、oracle.sql.*クラスは含まれません。これらの型では、標準JDBC機能を使用してください。たとえば、LONGまたはLONG RAWデータを、標準JDBC結果セットおよびコール可能文メソッドgetBinaryStreamおよびgetCharacterStreamを使用して入力ストリームとして取り出します。REF CURSOR型にはgetCursorメソッドを使用します。



ノート:

可能な場合は JDBC 標準型または Java 型の使用をお薦めします。パッケージ oracle.sql.*の方は主に下位互換性、または Oracle の少数の特定機能(OPAQUE、OracleData、TIMESTAMPTZ など)のサポートのために用意されています。

oracle.sql.*データ型の一般的なサポート

各Oracleデータ型クラスは、特に次の機能をサポートします。

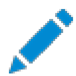
- SQLデータ用のJavaバイト配列であるデータ記憶域。
- SQLデータをバイト配列で戻すgetBytes()メソッド。
- JDBC仕様の定義に従って、データを対応するJavaクラスのオブジェクトに変換するtoJdbc()メソッド。

  JDBCドライバは、BFILEなどの、JDBC仕様の一部ではないOracle固有のデータ型を変換しません。ドライバは、対応するoracle.sql.*形式でオブジェクトを戻します。
- SQLデータをJava型に変換するのに適切なxxxValueメソッド。たとえば、stringValue、intValue、booleanValue、dateValueおよびbigDecimalValueがあげられます。

- 補足的な変換メソッド。データをストリームとして取り出すラージ・オブジェクト(LOB)クラスのメソッド、およびオブジェクト参照を使用してオブジェクト・データの取出しや設定を行うREFクラスのメソッドなど、データ型の機能に適したgetXXXおよびsetXXX。

クラスoracle.sql.STRUCTの概要

oracle.sql.STRUCTクラスは、java.sql.StructインタフェースのOracle実装です。このクラスは値クラスです。構成後にクラスの内容を変更しないでください。このクラスは、すべてのoracle.sql.*データ型のクラスと同様に、oracle.sql.Datumクラスのサブクラスです。

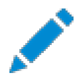


ノート:

Oracle Database 12c リリース 1 (12.1)以降、oracle.sql.STRUCT クラスは非推奨となり、oracle.jdbc.OracleStruct インタフェースに置き換えられています。このインタフェースは oracle.jdbc パッケージに属します。標準互換性には(可能であれば)java.sql パッケージの使用可能なメソッドを使用し、Oracle 固有の拡張機能には oracle.jdbc パッケージの使用可能なメソッドを使用することを強くお薦めします。oracle.jdbc.OracleStruct インタフェースの詳細は、MoS ノート 1364193.1 を参照してください。

クラスoracle.sql.REFの概要

oracle.sql.REFクラスは、Oracleオブジェクト参照をサポートする一般クラスです。このクラスは、すべてのoracle.sql.*データ型のクラスと同様に、oracle.sql.Datumクラスのサブクラスです。



ノート:

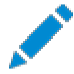
Oracle Database 12c リリース 1 (12.1)以降、oracle.sql.REF クラスは非推奨となり、oracle.jdbc.OracleRef インタフェースに置き換えられています。このインタフェースは oracle.jdbc パッケージに属します。標準互換性には(可能であれば)java.sql パッケージの使用可能なメソッドを使用し、Oracle 固有の拡張機能には oracle.jdbc パッケージの使用可能なメソッドを使用することを強くお薦めします。oracle.jdbc.OracleRef インタフェースの詳細は、MoS ノート 1364193.1 を参照してください。

REFクラスには、オブジェクト参照を取り出して渡すためのメソッドがあります。ただし、オブジェクト参照の選択によって取り出されるのは、オブジェクトへのポインタのみです。これによってオブジェクト自体がインスタンス化されることはありません。ただし、REFクラスにはオブジェクト・データを取り出して渡すためのメソッドも含まれています。JDBCアプリケーションではREFオブジェクトを作成できません。データベースから取り出すことができるのは、既存のREFオブジェクトのみです。

oracle.sql.REFを使用するのではなく、JDBC標準型、java.sql.RefおよびJDBC標準メソッドを使用する必要があります。コードを移植可能にする場合、Oracle JDBCドライバのみがoracle.sql.REF型のインスタンスを使用するので、標準型を使用する必要があります。

クラスoracle.sql.BLOB、oracle.sql.CLOB、oracle.sql.BFILEの概要

バイナリ・ラージ・オブジェクト(BLOB)、キャラクタ・ラージ・オブジェクト(CLOB)およびバイナリ・ファイル(BFILE)は、大きすぎてデータベース表に直接格納できないデータ項目に使用します。一方、データベース表はデータの実際の場所を指すロケータを格納します。



ノート:

- Oracle Database 12c リリース 1 (12.1)以降、oracle.sql.BLOB クラスおよび Oracle.sql.CLOB クラスは非推奨となり、それぞれ oracle.jdbc.OracleBlob インタフェースおよび oracle.jdbc.OracleClob インタフェースに置き換えられています。これらのインタフェースは oracle.jdbc パッケージに属します。標準互換性には(可能であれば) java.sql パッケージの使用可能なメソッドを使用し、Oracle 固有の拡張機能には oracle.jdbc パッケージの使用可能なメソッドを使用することを強くお薦めします。oracle.jdbc.OracleBlob インタフェースおよび oracle.jdbc.OracleClob インタフェースの詳細は、MoS ノート 1364193.1 を参照してください。
- oracle.sql.BFILE は Oracle 独自の拡張機能であり、これに相当する JDBC 標準機能はありません。

oracle.sqlパッケージは、これらのデータ型を次のいくつかの方法でサポートします。

- BLOBは、ラージ非構造化バイナリ・データ項目を指し、oracle.sql.BLOBクラスによりサポートされます。
- CLOBは、ラージ文字データ項目を指し、oracle.sql.CLOBクラスによりサポートされます。
- BFILEは、外部ファイル(オペレーティング・システム・ファイル)の内容を指し、oracle.sql.BFILEクラスによりサポートされます。BFILEは読取り専用です。

BLOB、CLOBまたはBFILEの各ロケータは、標準的なSELECT文を使用してデータベースから選択できます。ただし、受け取るのはデータではなく、ロケータのみです。データの取出しには、追加ステップが必要です。

クラスoracle.sql.DATE、oracle.sql.NUMBERおよびoracle.sql.RAWの概要

これらのクラスは、プリミティブSQLデータ型を、Oracleネイティブの表現で保持します。ほとんどの場合、ドライバはこれらの型を内部的に使用しません。なるべく標準JDBC型を使用してください。

JavaのDouble値およびFloat NaN値には、等価のOracleのNUMBER表現がありません。たとえば、OracleのBINARY_FLOATデータ型およびBINARY_DOUBLEデータ型の場合、負のゼロは正のゼロに強制変換され、NaNはすべて正規のものに強制変換されます。そのため、oracle.sql.NUMBERクラスを使用して、Double.NaN値またはFloat.NaN値がOracleのNUMBERに変換されると、必ずNullPointerExceptionがスローされます。たとえば、次のコードにより、NullPointerExceptionが発生します。

```
oracle.sql.NUMBER n = new oracle.sql.NUMBER(Double.NaN);
System.out.println(n.doubleValue());  // throws NullPointerException
```

クラスoracle.sql.TIMESTAMP、oracle.sql.TIMESTAMPTZおよびoracle.sql.TIMESTAMPLTZの概要

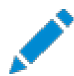
JDBCドライバでは、次のような日付/時刻データ型をサポートしています。

- TIMESTAMP (TIMESTAMP)
- TIMESTAMP WITH TIME ZONE(TIMESTAMPTZ)
- TIMESTAMP WITH LOCAL TIME ZONE(TIMESTAMPLTZ)

JDBCドライバでは、DATEと日付/時刻データ型の間での変換が可能です。たとえば、DATE値としてTIMESTAMP WITH TIME ZONE列にアクセスできます。

JDBCドライバは、業界で最も一般的なタイムゾーン名と、JDKで定義されたタイムゾーン名をサポートしています。タイムゾーン

は、java.util.TimeZoneクラスを使用して指定します。



ノート:

- タイムゾーン・オブジェクトの作成には TimeZone.getTimeZone を使用しないでください。これは、Oracle のタイムゾーン・データ型では、JDK よりも多くのタイムゾーン名をサポートしているためです。
- 結果セットの TIMESTAMPLTZ 列の後に LONG 列が続く場合、LONG 列を読み取ろうとするとエラーになります。

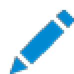
次のコードは、JDKに定義されていないタイムゾーン名であるUS_PACIFICにTimeZoneオブジェクトとCalendarオブジェクトを作成する方法を示します。

```
TimeZone tz = TimeZone.getDefault();
tz.setID("US_PACIFIC");
GregorianCalendar gcal = new GregorianCalendar(tz);
```

次のJavaクラスは、SQL日付/時刻データ型を表します。

- oracle.sql.TIMESTAMP
- oracle.sql.TIMESTAMPTZ
- oracle.sql.TIMESTAMPLTZ

TIMESTAMP WITH LOCAL TIME ZONEデータにアクセスする前に、OracleConnection.setSessionTimeZone(String regionName)メソッドをコールして、セッション・タイム・ゾーンを設定します。このメソッドがコールされると、JDBCドライバは接続のセッション・タイム・ゾーンを設定、保存して、JDBCを介してアクセスされるTIMESTAMP WITH LOCAL TIME ZONEデータをセッション・タイム・ゾーンを使用して調整できるようにします。



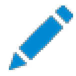
ノート:

TIMESTAMP WITH TIME ZONE 型と TIMESTAMP WITH LOCAL TIME ZONE 型は、標準の java.sql.Timestamp 型として表せます。TIMESTAMP WITH TIME ZONE 型と TIMESTAMP WITH LOCAL TIME ZONE 型の java.sql.Timestamp に対するバイト表現は簡単です。これは、TIMESTAMP WITH TIME ZONE データ型と TIMESTAMP WITH LOCAL TIME ZONE データ型の内部形式は GMT で、java.sql.Timestamp 型オブジェクトは内部では、エポックからのミリ秒数であるミリ秒時間値を使用するためです。ただし、これらのデータ型の String 表現には、サーバーから動的に取得され、クライアント側にキャッシュされているタイム・ゾーン情報が必要です。

JDBC ドライバの旧バージョンでは、タイム・ゾーンのキャッシュは様々な接続で共有されていました。このため、様々なタイム・ゾーンでの非互換性が原因で問題が発生することがよくありました。Oracle Database 11 リリース 2 の JDBC ドライバから、タイム・ゾーン・キャッシュは、データベースが提供するタイム・ゾーンのバージョンを基礎とします。この新設計のキャッシュは、タイム・ゾーンのバージョン非互換性に関連するどのような問題も回避します。

クラスoracle.sql.OPAQUEの概要

oracle.sql.OPAQUEクラスは、OPAQUE型の名前と特性および任意の属性を提供します。OPAQUE型では、インスタンスの連続バイトにのみアクセスできます。



ノート:

Oracle Database 12c リリース 1 (12.1)以降、oracle.sql.OPAQUE クラスは非推奨となり、oracle.jdbc.OracleOpaque インタフェースに置き換えられています。このインタフェースは oracle.jdbc パッケージに属します。標準互換性には(可能であれば)java.sql パッケージの使用可能なメソッドを使用し、Oracle 固有の拡張機能には oracle.jdbc パッケージの使用可能なメソッドを使用することをお薦めします。oracle.jdbc.OracleOpaque インタフェースの詳細は、MoS ノート 1364193.1 を参照してください。

関連トピック

- Oracle Database SQL言語リファレンス
- JDBC Java APIリファレンス
- LOBとBFILEの操作

4.3.2 パッケージoracle.jdbc

oracle.jdbcパッケージのインタフェースは、java.sqlのインタフェースに対するOracle拡張機能を定義します。これらの拡張機能は、Oracle SQL形式データおよびOracle固有の他の機能(Oracleパフォーマンス拡張機能など)にアクセスできます。

関連項目:

「oracle.jdbcパッケージ」

4.4 Oracle文字データ型のサポート

Oracle文字データ型には、SQL CHARおよびNCHARデータ型が含まれます。次の各項では、oracle.sql.*クラスを使用してこれらのデータ型にアクセスする方法について説明します。

- SQL CHARデータ型
- SQL NCHARデータ型
- クラスoracle.sql.CHAR

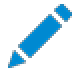
4.4.1 SQL CHARデータ型

SQL CHARデータ型には、CHAR、VARCHAR2およびCLOBデータ型が含まれます。これらのデータ型を使用すると、文字データをデータベース文字セットのコード体系で格納できます。データベースの文字セットは、データベースの作成時に決まります。

4.4.2 SQL NCHARデータ型

SQL NCHARデータ型は、グローバリゼーション・サポート用に作成されたものです。SQL NCHARデータ型には、NCHAR、NVARCHAR2およびNCLOBデータ型が含まれます。これらのデータ型を使用すると、Unicodeデータをデータベース NCHAR文字セットのコード体系で格納できます。NCHAR文字セットは、データベースの作成時に決まり、変更されることはありません。

ノート:



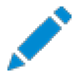
UnicodeStream クラスが非推奨となって CharacterStream クラスが導入されたため、NCHAR データ型でのアクセスでは setUnicodeStream メソッドと getUnicodeStream メソッドはサポートされません。ストリーム・アクセスを使用する場合、setCharacterStream メソッドと getCharacterStream メソッドを使用します。

SQL NCHARデータ型の使用方法は、SQL CHARデータ型の場合に類似しています。JDBCでは、対応するSQL CHARデータ型で使用されるのと同じクラスおよびメソッドを使用して、SQL NCHARデータ型にアクセスします。したがって、oracle.sqlパッケージには、SQL NCHARデータ型用の対応する別のクラスが定義されていません。同様に、SQL NCHARデータ型用のoracle.jdbc.OracleTypesクラスにも、対応する別の定数は定義されていません。

関連項目:

「NCHAR、NVARCHAR2、NCLOBおよびdefaultNCharプロパティ」

ノート:



setFormOfUse メソッドは、registerOutParameter メソッドをコールする前にコールして、予測できない結果を回避する必要があります。

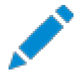
次のコードは、SQL NCHARデータにアクセスする方法を示します。

```
//
// Table TEST has the following columns:
// - NUMBER
// - NVARCHAR2
// - NCHAR
//
oracle.jdbc.OraclePreparedStatement pstmt =
  (oracle.jdbc.OraclePreparedStatement)
conn.prepareStatement("insert into TEST values(?, ?, ?)");
//
// oracle.jdbc.OraclePreparedStatement.FORM_NCHAR should be used for all NCHAR,
// NVARCHAR2 and NCLOB data types.
//
pstmt.setInt(1, 1);                    // NUMBER column
pstmt.setNString(2, myUnicodeString1);  // NVARCHAR2 column
pstmt.setNString(3, myUnicodeString2);  // NCHAR column
pstmt.execute();
```

4.4.3 クラスoracle.sql.CHAR

oracle.sql.CHARクラスは、文字データを処理および変換するためにOracle JDBCによって使用されます。このクラスは、文字データを変換するためのグローバリゼーション・サポート機能を提供します。このクラスには、グローバリゼーション・サポート文字セットおよび文字データという2つの主要な属性があります。グローバリゼーション・サポート文字セットは、文字データのエンコーディン

グを定義します。これは、CHARオブジェクトの作成時に必ず渡されるパラメータです。グローバリゼーション・サポート文字セット情報がない状態では、CHARオブジェクト内のデータ・バイトは無意味です。oracle.sql.CHARクラスは、SQL CHARおよびSQL NCHARの両方のデータ型に使用されます。



ノート:

10g リリース 1 より前のバージョンの Oracle JDBC ドライバでは、oracle.SQL.CHAR を使用することでパフォーマンス上の利点がありました。Oracle Database 10g 以上では、このような利点はありません。実際には、java.lang.String を使用することで最適なパフォーマンスを実現できます。すべての Oracle JDBC ドライバが、Java UCS2 文字セット内のすべての文字データを処理します。oracle.sql.CHAR を使用すると、データベース文字セットと UCS2 文字セットとの間の変換が発生します。

oracle.sql.CHARクラスに残された唯一の使用方法は、文字データをOracleグローバリゼーション・サポート文字セットでエンコードされたRAWバイトの形式で処理する場合です。Oracle Databaseから取り出された文字データはすべてjava.lang.Stringクラスを使用してアクセスしてください。別のソースからのバイト・データを処理する場合は、oracle.sql.CHARを使用してバイトをjava.lang.Stringに変換できます。

oracle.sql.CHARを変換するには、データ・バイトと、そのデータ・バイトのエンコードに使用するグローバリゼーション・サポート文字セットを示すoracle.sql.CharacterSetインスタンスを用意する必要があります。

Oracleオブジェクト属性であるCHARオブジェクトは、データベース文字セットで戻されます。

CHARオブジェクトは、必要に応じてJDBCドライバで自動的に作成されるため、まれに作成する必要がある場合でも、JDBCアプリケーション・コードでCHARオブジェクトを直接作成する必要はほとんどありません。

CHARオブジェクトを作成する場合は、CharacterSetクラスのインスタンスによって、CHARオブジェクトに文字セット情報を提供する必要があります。このクラスの各インスタンスは、Oracleがサポートしているグローバリゼーション・サポート文字セットの1つを表します。CharacterSetインスタンスは、文字セットのメソッドおよび属性をカプセル化し、主に他の文字セットとの相互変換機能を提供します。

oracle.sql.CHARオブジェクトの作成

CHARオブジェクトを構築する際、次の一般的なステップに従ってください。

1. CharacterSetオブジェクトは、static CharacterSet.makeメソッドをコールして作成します。

   このメソッドは、文字セット・インスタンスのファクトリです。makeメソッドは、Oracleがサポートする文字セットIDに対応する整数を入力として取ります。たとえば:

   ```
   int oracleId = CharacterSet.JA16SJIS_CHARSET; // this is character set ID,
                                                 // 832
   ...
   CharacterSet mycharset = CharacterSet.make(oracleId);
   ```

   Oracleがサポートする各文字セットは、事前定義済の一意なOracle IDを保持します。

2. CHARオブジェクトを作成します。

文字列または文字列を表すバイトを、文字セットに基づくバイトの解釈方法を指定するCharacterSetオブジェクトとともに、コンストラクタに渡します。たとえば:

```
String mystring = "teststring";
...
CHAR mychar = new CHAR(teststring, mycharset);
```

CHARには、CharacterSetオブジェクトとともに、String、byte配列またはオブジェクトを入力として使用できる複数のコンストラクタがあります。Stringの場合、文字列は、CharacterSetオブジェクトが示す文字セットに変換されてから、CHARオブジェクトに格納されます。

ノート:

- CharacterSet オブジェクトを NULL 値にすることはできません。
- CharacterSet クラスは抽象クラスであるため、コンストラクタを持ちません。インスタンスを作成するには、make メソッドを使用するのが唯一の方法です。
- サーバーは、特別な値である CharacterSet.DEFAULT_CHARSET をデータベース文字セットとして認識します。クライアントの場合、この値は無意味です。
- ユーザーが CharacterSet クラスを拡張することを、オラクル社は意図しておらず、お薦めもしません。

oracle.sql.CHAR変換メソッド

CHARクラスは、文字データを文字列に変換するための次のようなメソッドを提供します。

- getString

  このメソッドは、CHARオブジェクトによって表された文字のシーケンスを文字列に変換し、Java Stringオブジェクトを戻します。無効なOracleIDを入力した場合、文字セットは認識されず、getStringメソッドはSQLException例外をスローします。

- toString

  このメソッドはgetStringメソッドと同じです。ただし、無効なOracleIDを入力すると、文字セットが認識されず、toStringメソッドによりCHARデータの16進表現が戻されます。SQLException例外はスローされません。

- getStringWithReplacement

  このメソッドはgetStringメソッドと同一ですが、CHARオブジェクトの文字セットにUnicode表現のない文字が、デフォルトの置換文字で置き換えられる点が異なります。デフォルトの置換文字は文字セットによって異なりますが、多くの場合は疑問符(?)です。

データベース・サーバーとクライアント、またはクライアント上で動作するアプリケーションでは、異なる文字セットを使用できます。CHARクラスのメソッドを使用してデータをサーバーとクライアント間で転送する場合、JDBCドライバはデータをサーバーの文字セットからクライアントの文字セットに(またはその逆に)変換する必要があります。データを変換する際、ドライバはグローバリゼーション・サポートを使用します。

関連項目:

グローバリゼーション・サポート

4.5 その他のOracle型拡張機能

Oracle JDBCドライバは、Oracle固有のBFILEデータ型、ROWIDデータ型およびREF CURSOR型をサポートしますが、これらは標準JDBC仕様に含まれていません。このセクションでは、ROWIDおよびREF CURSOR型の拡張について説明します。ROWIDはJava文字列としてサポートされ、REF CURSOR型はJDBC結果セットとしてサポートされます。

この項の内容は次のとおりです。

- Oracle ROWID型
- OracleのREF CURSOR型のカテゴリ
- Oracle BINARY_FLOAT型およびBINARY_DOUBLE型
- Oracle SYS.ANYTYPE型およびSYS.ANYDATA型
- oracle.jdbcパッケージ

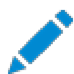
4.5.1 Oracle ROWID型

ROWIDは、Oracle Database表の行ごとに一意の識別タグです。ROWIDは、各行のIDを含む仮想列とみなすこともできます。

oracle.sql.ROWIDクラスは、ROWID SQLデータ型のコンテナとして提供されます。

ROWIDは、java.sql.ResultSetインタフェースで指定されるgetCursorNameメソッドおよびjava.sql.Statementインタフェースで指定されるsetCursorNameメソッドに類似した機能を提供します。

問合せにROWID擬似列を指定した場合は、結果セットgetStringメソッドを使用してROWIDを取得できます。また、setStringメソッドを使用すると、ROWIDにPreparedStatementパラメータをバインドできます。これによって、次の例に示すように、埋込み更新を実行できます。

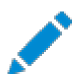


ノート:

oracle.sql.ROWID クラスを使用するのは、J2SE 5.0 を使用する場合のみとしてください。JSE 6 の場合は、標準の java.sql.RowId インタフェースを使用してください。

例

次の例では、ROWIDデータに対するアクセスおよび操作の方法を示します。



ノート:

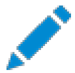
次の例は、JSE 6 でのみ動作します。

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

// Query the employee names with "FOR UPDATE" to lock the rows. // Select the ROWID to identify the rows to be updated. ResultSet rset = stmt.executeQuery ("SELECT first_name, rowid FROM employees FOR UPDATE"); // Prepare a statement to update the first_name column at a given ROWID PreparedStatement pstmt = conn.prepareStatement ("UPDATE employees SET first_name = ? WHERE rowid = ?"); // Loop through the results of the query while (rset.next ()) { String ename = rset.getString (1); RowId rowid = rset.getROWID(2); // Get the ROWID as a String pstmt.setString (1, ename.toLowerCase ()); pstmt.setROWID (2, rowid); // Pass ROWID to the update statement pstmt.executeUpdate (); // Do the update }

4.5.2 OracleのREF CURSOR型のカテゴリ

カーソル変数は、問合せ作業領域の(内容でなく)メモリーの場所を保持します。カーソル変数を宣言すると、ポインタが作成されます。SQLでは、ポインタはデータ型REF xを取ります。ここで、REFはREFERENCEの省略形で、xは参照されるエンティティを表します。したがって、REF CURSORはカーソル変数への参照を表します。多くの作業領域を参照する多くのカーソル変数が存在する可能性があるため、REF CURSORは、異なる多くの型のカーソル変数を識別するカテゴリまたはデータ型指定子と考えることができます。Oracle Databaseリリース18c以降、JDBCドライバはINバインド変数としてREF CURSORをサポートしています。



ノート:

REF CURSOR インスタンスはスクロールできません。

カーソル変数を作成するには、次のステップを実行します。

1. REF CURSORカテゴリに属している型を指定します。たとえば:

   ```
   DECLARE TYPE DeptCursorTyp IS REF CURSOR
   ```

2. 次に、DeptCursorTyp型であることを宣言して、カーソル変数を作成します。

   ```
   dept_cv DeptCursorTyp  -- declare cursor variable
   ...
   ```

   REF CURSORは、特定のデータ型というよりも、データ型のカテゴリです。

ストアド・プロシージャは、REF CURSORカテゴリのカーソル変数を受け入れるか、または戻します。この出力は、データベース・カーソルまたはJDBC結果セットと等価です。REF CURSORには、基本的に問合せの結果が格納されます。

JDBCでは、REF CURSORは次のようにアクセスできます。

1. ストアド・プロシージャをコールするには、JDBCコール可能文またはプリペアド文を使用します。
2. ストアド・プロシージャは、REF CURSORを受け入れるか、または戻します。
3. Javaアプリケーションは、コール可能文またはプリペアド文をOracleコール可能文またはOracleプリペアド文にキャストします。

4. JavaアプリケーションはOraclePreparedStatementインタフェースのsetCursorメソッドまたはOracleCallableStatementインタフェースのgetCursorメソッドを使用して、REF CURSORをJDBC ResultSetオブジェクトとしてインスタンス化します。

5. 結果セットは要求どおりに処理されます。

> ノート:
>
> - REF CURSORと関連付けられたカーソルは、そのREF CURSORを作成したStatementオブジェクトがクローズされるたびにクローズされます。
> - 過去のリリースとは異なり、REF CURSORのインスタンス化が行われた結果セット・オブジェクトがクローズされても、REF CURSORに関連付けられたカーソルはクローズされません。

例

この例では、REF CURSORデータにアクセスする方法を示します。

```
    ...
    // Prepare a PL/SQL call
    CallableStatement cstmt =
      conn.prepareCall ("DECLARE  rc sys_refcursor; curid NUMBER;BEGIN open rc FOR SELECT empno FROM
emp order by empno; ? := rc;  END;");
    cstmt.registerOutParameter (1, OracleTypes.CURSOR);
    cstmt.execute ();
    ResultSet rset = (ResultSet)cstmt.getObject (1);
    if (rset.next ()) {
      show (rset.getString ("empno"));
    }

    CallableStatement cstmt2 =
      conn.prepareCall ("DECLARE  rc sys_refcursor; v1 NUMBER; BEGIN rc := ?; fetch rc INTO v1; ? :=
v1; END;");
    ((OracleCallableStatement)call2).setCursor(1, rset);
    cstmt2.registerOutParameter (2, OracleTypes.INTEGER);

    cstmt2.execute();

    int empno = cstmt2.getInt(2);

    show("Fetch in PL/SQL empno=" + empno);

    // Dump the cursor
    while (rset.next ())
      show (rset.getString ("empno"));
    // Close all the resources
    rset.close();
    cstmt.close();
    cstmt2.close();
  ...
```

前述の例は、次のとおりです。

- cstmt1とcstmt2という2つのCallableStatementオブジェクトがConnectionクラスのprepareCallメソッドを使用して

作成されます。

- cstmt2コール可能文では入力パラメータとしてREF CURSORを使用します。
- コール可能文によって、REF CURSORを戻すPL/SQLプロシージャが実装されます。
- コール可能文の出力パラメータは、通常どおり登録してその型を定義する必要があります。REF CURSORには、型コードOracleTypes.CURSORを使用します。
- コール可能文が実行され、REF CURSORを返すか、またはREF CURSORを入力バインドとして送信します。

4.5.3 Oracle BINARY_FLOAT型およびBINARY_DOUBLE型

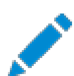
Oracle BINARY_FLOAT型およびBINARY_DOUBLE型は、IEEE 574のfloat型とdouble型のデータを格納するために使用します。これらの型は、負とゼロおよびNaNを除いて、Javaのfloatスカラー型とdoubleスカラー型に相当します。

関連項目:

『Oracle Database SQL言語リファレンス』

BINARY_DOUBLE列を問合せに含めると、データはデータベースからバイナリ形式で取り出されます。getDoubleメソッドも、データをバイナリ形式で戻します。一方、NUMBERデータ型の列の場合は、数値ビットが戻り、Javaのdoubleデータ型に変換されます。

ノート:

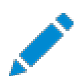


SQL FLOAT 型、DOUBLE PRECISION 型および REAL データ型の Oracle 表現は、Oracle NUMBER 表現を使用します。BINARY_FLOAT 型と BINARY_DOUBLE データ型は、独自の型とみなされます。

PreparedStatementインタフェースのJDBC標準setDouble(int, double)メソッドのコールは、Javaのdouble引数をOracleのNUMBERスタイル・ビットに変換してからデータベースに送信します。一方、oracle.jdbc.OraclePreparedStatementインタフェースのsetBinaryDouble(int, double)メソッドは、データを内部バイナリ・ビットに変換してからデータベースに送信します。

使用するデータ形式が、PreparedStatementインタフェースのターゲット・パラメータの型と一致していることを確認してください。この確認によって、データの正確性が保持され、CPUの使用も最小限に抑えられます。NUMBERパラメータにsetBinaryDoubleを使用すると、バイナリ・ビットはサーバーに送信され、NUMBER形式に変換されます。データの正確性は保持されますが、サーバーのCPU負荷は増加します。BINARY_DOUBLEパラメータにsetDoubleを使用すると、データは、クライアントでNUMBERビットに変換されてからサーバーに送信され、そこでバイナリ形式に再変換されます。この方法では、クライアントとサーバーの両方でCPU負荷が増大し、データの精度も低下する可能性があります。

SetFloatAndDoubleUseBinary接続プロパティがtrueに設定されていると、JDBC標準API、setFloat(int, float)、setDouble(int, double)、およびすべての変数では、NUBMERビットではなく、内部バイナリ・ビットを送信するようになります。



ノート:

ここでは、主にBINARY_DOUBLEについて説明しましたが、BINARY_FLOATについても同じ内容が適用されます。

4.5.4 Oracle SYS.ANYTYPE型およびSYS.ANYDATA型

Oracle Database 12cリリース1 (12.1)では、SYS.ANYTYPEおよびSYS.ANYDATAというOracle型にアクセスするためのJavaインタフェースが提供されています。

関連項目:

これらのOracle型の詳細は、『Oracle Database PL/SQLパッケージおよびタイプ・リファレンス』を参照してください

SYS.ANYTYPE型のインスタンスには、オブジェクト型およびコレクション型も含め、SQL型の型記述(永続か一時、名前付きか名前なし)が含まれます。oracle.sql.TypeDescriptorクラスを使用すると、SYS.ANYTYPE型にアクセスできます。ANYTYPEインスタンスは、PL/SQLのプロシージャまたはSQLのSELECT文(SYS.ANYTYPEを列型として使用)を使用して取り出すことができます。ANYTYPEインスタンスをデータベースから取り出すには、getObjectメソッドを使用します。このメソッドは、TypeDescriptorのインスタンスを戻します。

次の任意のANYTYPEインスタンスを取り出すことができます。

- 一時オブジェクト型
- 一時事前定義済型
- 永続オブジェクト型
- 永続事前定義済型

例4-1 SYS.ANYTYPE型へのアクセス

次のコードは、ANYTYPEのインスタンスをデータベースから取り出す方法を示しています。

```
...
ResultSet rs = stmt.executeQuery("select anytype_column from my_table");
TypeDescriptor td = (TypeDescriptor)rs.getObject(1);
short typeCode = td.getInternalTypeCode();
if(typeCode == TypeDescriptor.TYPECODE_OBJECT)
{
  // check if it's a transient type
  if(td.isTransientType())
  {
    AttributeDescriptor[] attributes = ((StructDescriptor)td).getAttributesDescriptor();
    for(int i=0; i<attributes.length; i++)
      System.out.println(attributes[i].getAttributeName());
  }
  else
  {    System.out.println(td.getTypeName());  }}
...
```

例4-2 一時オブジェクト型のPL/SQLによる作成とJDBCによる取出し

次のサンプル・コードは、JDBCによる一時オブジェクト型の取出し方法を示しています。

```
...
OracleCallableStatement cstmt = (OracleCallableStatement)conn.prepareCall
  ("BEGIN ? := transient_obj_type (); END;");
cstmt.registerOutParameter(1,OracleTypes.OPAQUE,"SYS.ANYTYPE");
cstmt.execute();
TypeDescriptor obj = (TypeDescriptor)cstmt.getObject(1);
if(!obj.isTransient())
  System.out.println("This must be a JDBC bug");
cstmt.close();
return obj;
...
```

例4-3 ANYTYPEをINパラメータとして受け取るPL/SQLストアド・プロシージャのコール

次のコードは、ANYTYPEをINパラメータとして受け取るPL/SQLストアド・プロシージャのコール方法を示しています。

```
...
CallableStatement cstmt = conn.prepareCall("BEGIN ? := dumpanytype(?); END;");
cstmt.registerOutParameter(1,OracleTypes.VARCHAR);
// obj is the instance of TypeDescriptor that you have retrieved
cstmt.setObject(2,obj);
cstmt.execute();
String str = (String)cstmt.getObject(1);
...
```

oracle.sql.ANYDATAクラスを使用すると、データベースからSYS.ANYDATAインスタンスにアクセスできます。このクラスのインスタンスは、oracle.sql.Datumの任意の有効なインスタンスから取得できます。convertDatumファクトリ・メソッドは、Datumのインスタンスを受け取り、ANYDATAのインスタンスを戻します。このファクトリ・メソッドの構文は、次のとおりです。

```
public static ANYDATA convertDatum(Datum datum) throws SQLException
```

oracle.sql.ANYDATAのインスタンスを作成するためのサンプル・コードを、次に示します。

```
// struct is a valid instance of oracle.sql.STRUCT that either comes from the
// database or has been constructed in Java.
ANYDATA myAnyData = ANYDATA.convertDatum(struct);
```

例4-4 ANYDATAのインスタンスに対するデータベースからのアクセス

```
...
// anydata_table has been created as:
// CREATE TABLE anydata_tab (data SYS.ANYDATA)
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("select data from my_anydata_tab");
while(rs.next())
{
  ANYDATA anydata = (ANYDATA)rs.getObject(1);
  if(!anydata.isNull())
  {
    TypeDescriptor td = anydata.getTypeDescriptor();
    if(td.getTypeCode() == OracleType.TYPECODE_OBJECT)
      STRUCT struct = (STRUCT)anydata.accessDatum();
  }
}
...
```

例4-5 データベース表へのANYDATAとしてのオブジェクトの挿入

表およびオブジェクト型が次のように定義されているものとします。

```
CREATE TABLE anydata_tab ( id NUMBER, data SYS.ANYDATA)
CREATE OR REPLACE TYPE employee AS OBJECT ( employee_id NUMBER, first_name VARCHAR2(10) )
```

次の方法で、SQLオブジェクト型EMPLOYEEのインスタンスを作成し、anydata_tableに挿入することができます。

```
...
PreparedStatement pstmt = conn.prepareStatement("insert into anydata_table values (?,?)");
Struct myEmployeeStr = conn.createStruct("EMPLOYEE", new Object[]{1120, "Papageno"});
ANYDATA anyda = ANYDATA.convertDatum(myEmployeeStr);
pstmt.setInt(1,123);
pstmt.setObject(2,anyda);
pstmt.executeUpdate();
...
```

例4-6 データベース表からのANYDATA列の選択

```
...
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("select data from anydata_table");
while(rs.next())
{
  ANYDATA obj = (ANYDATA)rs.getObject(1);
  TypeDescriptor td = obj.getTypeDescriptor();
}
rs.close();
stmt.close();
...
```

4.5.5 oracle.jdbcパッケージ

oracle.jdbcパッケージのインタフェースは、java.sqlのインタフェースに対するOracle拡張機能を定義します。これらの拡張機能によって、この章に記載されているSQL形式のデータへのアクセスが可能になります。また、Oracleパフォーマンス拡張要素など、Oracle固有の他の機能へのアクセスも可能になります。

表4-1に、oracle.jdbcパッケージに含まれる接続に使用する主なインタフェースとクラス、文および結果セットのリストを示します。

表4-1 oracle.jdbcパッケージの主なインタフェースおよびクラス

名前	インタフェースまたはクラス	主要機能
OracleDriver	クラス	java.sql.Driver を実装します。
OracleConnection	インタフェース	Oracle Database インスタンスを起動または停止するためのメソッド、Oracle Statement オブジェクトを戻すメソッド、および現行接続で実行される任意の文に対応した Oracle パフォーマンス拡張機能を設定するメソッド

名前	インタフェースまたはクラス	主要機能
		を提供します。 java.sql.Connection を実装します。
OracleStatement	インタフェース	文ごとに Oracle パフォーマンス拡張機能を設定するメソッドを提供します。 OraclePreparedStatement および OracleCallableStatement のスーパータイプです。 java.sql.Statement を実装します。
OraclePreparedStatement	インタフェース	oracle.sql.*型をプリペアド文にバインドする setXXX メソッドを提供します。 SELECT 文を実行しないでプリペアド文からメタデータを取得する getMetaData メソッドを提供します。 java.sql.PreparedStatement を実装します。 OracleStatement を機能拡張します。 OracleCallableStatement のスーパータイプです。
OracleCallableStatement	インタフェース	データを oracle.sql 形式で取り出す getXXX メソッド、および oracle.sql.*型をコール可能文にバインドする setXXX メソッドを提供します。 java.sql.CallableStatement を実装します。 OraclePreparedStatement を機能拡張します。
OracleResultSet	インタフェース	oracle.sql 形式でデータを取り出す getXXX メソッドを提供します。 java.sql.ResultSet を実装します。
OracleResultSetMetaData	インタフェース	列名やデータ型などの Oracle 結果セットのメタデータ情報を取得するメソッドを提供します。 java.sql.ResultSetMetaData を実装します。

名前	インタフェースまたはクラス	主要機能
OracleDatabaseMetaData	クラス	データベース製品名/バージョン、表情報、デフォルト・トランザクション分離レベルなどのデータベースに関するメタデータ情報を取得するメソッドを提供します。 java.sql.DatabaseMetaData を実装します。
OracleTypes	クラス	SQL 型を識別する整数定数を定義します。 標準型の場合、同じ値を標準 java.sql.Types クラスとして使用します。さらに、Oracle 拡張型に対応した定数も追加します。
OracleArray	インタフェース	配列全体を取り出し、配列要素のサブセットを取り出し、配列要素の SQL ベース型名を取り出すための機能があります。
OracleStruct	インタフェース	
OracleClob	インタフェース	
OracleBlob	インタフェース	
OracleRef	インタフェース	
OracleOpaque	インタフェース	

この項の内容は次のとおりです。

- インタフェースoracle.jdbc.OracleConnection
- インタフェースoracle.jdbc.OracleStatement
- インタフェースoracle.jdbc.OraclePreparedStatement
- インタフェースoracle.jdbc.OracleCallableStatement
- インタフェースoracle.jdbc.OracleResultSet
- インタフェースoracle.jdbc.OracleResultSetMetaData
- クラスoracle.jdbc.OracleTypes

4.5.5.1 インタフェースoracle.jdbc.OracleConnection

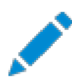
このインタフェースは、標準JDBC接続機能を拡張して、Oracle Statementオブジェクトの作成と取得、Oracleパフォーマン

ス拡張機能用のフラグやオプションの設定、Oracleオブジェクト用型マップのサポート、およびクライアント識別子のサポートを実施します。

Oracle Database 11gリリース1では、Oracle Databaseインスタンスの起動と停止を実行できる新しいメソッドがこのインタフェースに追加されました。また、OracleConnectionインタフェースでは、可視性および透明性を向上させるために、すべての接続プロパティを定数として定義します。

このインタフェースは、DATEおよびNUMBERのようなoracle.sqlデータ値を構築するために、ファクトリ・メソッドも定義します。ファクトリ・メソッドの使用時には次の点に注意してください。

- oracle.sql型のインスタンスを構築するすべてのコードで、Oracle拡張ファクトリ・メソッドを使用する必要があります。たとえば、ARRAY、BFILE、DATE、INTERVALDS、NUMBER、STRUCT、TIME、TIMESTAMPなどです。
- 標準型のインスタンスを構築するすべてのコードで、JDBC4.0標準ファクトリ・メソッドを使用する必要があります。たとえば、CLOBBLOBNCLOBなどです。
- CHAR、JAVA_STRUCT、ArrayDescriptorおよびStructDescriptorには、ファクトリ・メソッドがありません。これらのタイプは内部ドライバ専用です。



ノート:

Oracle Database 11g リリース 1 より前では、引数として ARRAY および STRUCT クラス・コンストラクタに渡すために、ArrayDescriptors および StructDescriptors を作成する必要がありました。新しい ARRAY および Struct ファクトリ・メソッドは記述子の引数がありません。ドライバはまだ内部で記述子を使用していますが、作成する必要はありません。

クライアント識別子

クライアント識別子を接続プーリング環境で使用すると、データベース・セッションを現在使用している軽量ユーザーを識別できます。クライアント識別子を使用すると、異なるデータベース・セッション間でグローバル・アクセス・アプリケーション・コンテキストを共有することもできます。データベース・セッションで設定されたクライアント識別子は、データベース監査がオンにされたときに監査されます。

関連項目:

詳細は、Oracle Database JDBC Java APIリファレンスを参照してください

4.5.5.2 インタフェースoracle.jdbc.OracleStatement

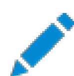
このインタフェースは標準のJDBC文機能を拡張するもので、OraclePreparedStatementクラスとOracleCallableStatementクラスのスーパーインタフェースです。拡張された機能には、Oracleパフォーマンス拡張機能用の設定フラグとオプションの文単位でのサポートが含まれます。OracleConnectionインタフェースでは、それらを接続単位で設定していました。

4.5.5.3 インタフェースoracle.jdbc.OraclePreparedStatement

このインタフェースは、OracleStatementインタフェースを拡張し、標準JDBCのプリペアド文の機能を拡張します。また、oracle.jdbc.OraclePreparedStatementインタフェースはOracleCallableStatementインタフェースによって拡張されます。拡張機能は次の機能から構成されます。

- oracle.sql.*型とオブジェクトをプリペアド文にバインドするためのsetXXXメソッド
- SELECT文を実行しないで、プリペアド文からメタデータを取得するためのgetMetaDataメソッド
- 文単位でOracleパフォーマンス拡張機能をサポートするためのメソッド

ノート:

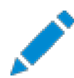


:NEW または:OLD 列を参照するトリガーの作成に PreparedStatement インタフェースを使用しないでください。かわりに、Statement を使用してください。PreparedStatement を使用すると、実行が失敗し、メッセージ「java.sql.SQLException: IN または OUT パラメータがありません - 索引:: 1」が発行されます。

4.5.5.4 インタフェースoracle.jdbc.OracleCallableStatement

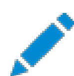
このインタフェースは、OracleStatementインタフェースを拡張するOraclePreparedStatementインタフェースを機能拡張し、標準JDBCのコール可能文機能を組み込みます。

ノート:



:NEW または:OLD 列を参照するトリガーの作成に CallableStatement インタフェースを使用しないでください。かわりに Statement を使用してください。CallableStatement を使用すると、実行が失敗し、メッセージ「java.sql.SQLException: IN または OUT パラメータがありません - 索引:: 1」が発行されます。

ノート:



- setXXX(String,...)と registerOutParameter(String,...)メソッドは、すべてのバインドがプロシージャまたは関数パラメータのみの場合にかぎり使用できます。文はそれ以外のバインドを含むことができません。パラメータ・バインドは疑問符で示す必要があります(:XX ではなく、?)。
- setXXX(int,...)または setXXXAtName(String,...)メソッドを使用している場合、出力パラメータは、名前付きパラメータを指定するための registerOutParameter(String,...)ではなく、registerOutParameter(int,...)を使用してバインドされます。

4.5.5.5 インタフェースoracle.jdbc.OracleResultSet

このインタフェースは、標準JDBC結果セットの機能を拡張し、getXXXメソッドを実装してデータをoracle.sql.*オブジェクト内に取得します。

4.5.5.6 インタフェースoracle.jdbc.OracleResultSetMetaData

このインタフェースは、標準JDBC結果セットのメタデータ機能を拡張して、Oracle結果セット・オブジェクトの情報を取得します。

関連項目:

「結果セット・メタデータ拡張機能の使用方法」

4.5.5.7 クラスoracle.jdbc.OracleTypes

OracleTypesクラスは、JDBCがSQL型を識別するのに使用する定数を定義します。このクラス内の各変数は、整数の定数値を持ちます。oracle.jdbc.OracleTypesクラスには、標準Java java.sql.Typesクラスの型コード定義の複製と、次のOracle拡張機能用の補足的な型コードが含まれます。

- OracleTypes.BFILE
- OracleTypes.ROWID
- OracleTypes.CURSOR(REF CURSOR型の場合)
- OracleTypes.CHAR_BYTES(同じ列のsetNullとsetCHARメソッドをコールする場合)

java.sql.Typesの場合と同様に、変数名はすべて大文字にする必要があります。

JDBCは、OracleTypesクラスの要素によって識別されたSQL型を、出力パラメータの登録、およびPreparedStatementクラスのsetNullメソッドという2つの分野で使用します。

OracleTypesと出力パラメータの登録

java.sql.Typesまたはoracle.jdbc.OracleTypesの型コードは、java.sql.CallableStatementインタフェースおよびoracle.jdbc.OracleCallableStatementインタフェースのregisterOutParameterメソッドで出力パラメータのSQL型を識別します。

次に、CallableStatementインタフェースおよびOracleCallableStatementインタフェースでregisterOutputParameterメソッドが使用可能な形式を示します。

```
cs.registerOutParameter(int index, int sqlType);
cs.registerOutParameter(int index, int sqlType, String sql_name);
cs.registerOutParameter(int index, int sqlType, int scale);
```

これらのシグネチャでは、indexにはパラメータ索引が、sqlTypeにはSQLデータ型の型コードが入ります。sql_nameにはデータ型に与えられた名前が入りますが、sqlTypeがSTRUCT、REFまたはARRAY型コードの場合にはユーザー定義型の名前が入ります。さらに、sqlTypeがNUMERICまたはDECIMAL型コードの場合には、scaleに、小数点の右側にある桁の数が入ります。

次の例は、CallableStatementインタフェースを使用して、charoutという名前付きプロシージャをコールしています。CHARデータ型が戻されます。registerOutParameterメソッドにおけるOracleTypes.CHAR型のコードの使用に注意してください。

```
CallableStatement cs = conn.prepareCall ("BEGIN charout (?); END;");
cs.registerOutParameter (1, OracleTypes.CHAR);
cs.execute ();
System.out.println ("Out argument is: " + cs.getString (1));
```

次の例は、CallableStatementインタフェースを使用して、structoutをコールしています。STRUCTデータ型が戻されます。registerOutParameterの形式では、型コードにTypes.STRUCTまたはOracleTypes.STRUCT、SQL名にEMPLOYEEを指定する必要があります。

この例では、EMPLOYEE型でいかなる型マップも宣言されていないことを前提としているため、STRUCTデータ型内に取得されます。oracle.sql.STRUCTオブジェクトとしてEMPLOYEEの値を取り出すために、Statemenオブジェクトcsは、OracleCallableStatementにキャストされ、Oracle拡張機能getSTRUCTメソッドが実行されます。

```
CallableStatement cs = conn.prepareCall ("BEGIN structout (?); END;");
cs.registerOutParameter (1, OracleTypes.STRUCT, "EMPLOYEE");
cs.execute ();
// get the value into a STRUCT because it
// is assumed that no type map has been defined
STRUCT emp = ((OracleCallableStatement)cs).getSTRUCT (1);
```

OracleTypesおよびsetNullメソッド

TypesおよびOracleTypesでの型コードは、データ項目のSQL型を特定します。setNullメソッドがそれをNULLに設定します。setNullメソッドは、java.sql.PreparedStatementインタフェースとoracle.jdbc.OraclePreparedStatementインタフェースにあります。

次に、PreparedStatementおよびOraclePreparedStatementオブジェクトでsetNullメソッドが使用可能な形式を示します。

```
ps.setNull(int index, int sqlType);
ps.setNull(int index, int sqlType, String sql_name);
```

これらのシグネチャでは、indexがパラメータ索引を表します。sqlTypeはSQLデータ型のための型コードです。sql_nameは、sqlTypeがSTRUCT、REFまたはARRAY型コードである場合に、ユーザー定義のデータ型に対して指定される名前です。無効なsqlTypeを入力すると、ParameterTypeConflict例外がスローされます。

次の例では、プリペアド文を使用してNULL値をデータベースに挿入します。NULLに設定する数値オブジェクトの識別にはOracleTypes.NUMERICを使用することに注意してください。ただし、Types.NUMERICも使用されることがあります。

```
PreparedStatement pstmt =
    conn.prepareStatement ("INSERT INTO num_table VALUES (?)");
pstmt.setNull (1, OracleTypes.NUMERIC);
pstmt.execute ();
```

この例では、プリペアド文を使用して、EMPLOYEE型のNULL STRUCTオブジェクトをデータベースに挿入します。

```
PreparedStatement pstmt = conn.prepareStatement
                               ("INSERT INTO employees VALUES (?)");
pstmt.setNull (1, OracleTypes.STRUCT, "EMPLOYEE");
pstmt.execute ();
```

同じ列のsetCHARメソッドもコールする場合、OracleTypes.CHAR_BYTES型をsetNullメソッドで使用することもできます。たとえば:

```
 ps.setCHAR(n, aCHAR);
  ps.addBatch();
  ps.setNull(n, OracleTypes.CHAR_BYTES);
```

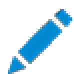
```
ps.addBatch();
```

この例では、OracleTypes.CHAR_BYTES型以外のどの型を使用しても、データベースとの余計なラウンドトリップが発生します。また、setNullメソッドを使用しないでコードを記述することもできます。たとえば、次の例のようにコードを記述することもできます。

```
ps.setCHAR(n, null);
```

4.6 DML RETURNING

DML RETURNING機能は、自動生成キーの取出しに比べて、より豊富な機能を提供します。この機能は、自動生成キーの取出しに使用できますが、アプリケーションで使用する可能性がある他の列や値の取出しにも使用できます。



ノート:

- サーバー側内部ドライバは、DML RETURNING および自動生成キーの取出しをサポートしていません。
- 同一文中で DML RETURNING と自動生成キーの取得の両方を使用することはできません。

次の各項では、DML RETURNING機能のサポートについて説明します。

- Oracle固有のAPI
- DML RETURNING文の実行について
- DML RETURNINGの例
- DML RETURNINGの制限事項

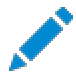
関連項目:

「自動生成キーの取出し」

4.6.1 Oracle固有のAPI

OraclePreparedStatementインタフェースは、DML RETURNINGをサポートするOracle固有のApplication Program Interface(API)に伴って機能拡張されました。DML RETURNINGによって戻されるパラメータおよび取り出されるデータを登録するために、registerReturnParameterメソッドおよびgetReturnResultSetメソッドがoracle.jdbc.OraclePreparedStatementインタフェースに追加されています。

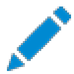
DML RETURNING機能の戻りパラメータは、registerReturnParameterメソッドを使用して登録します。エラーが発生した場合、メソッドはSQLExceptionインスタンスをスローします。戻りパラメータの索引を指定する正の整数を渡す必要があります。また、戻りパラメータの型も指定する必要があります。戻りパラメータの最大バイト数または最大文字数も指定できます。このメソッドは、charまたはRAW型でのみ使用できます。SQL構造型の完全修飾名も指定できます。



ノート:

戻りパラメータの最大サイズが不明な場合は、デフォルトの最大サイズを選択するregisterReturnParameter(int paramIndex, int externalType)を使用してください。戻りパラメータの最大サイズが判明している場合は、registerReturnParameter(int paramIndex, int externalType, int maxSize)を使用することでメモリー消費を抑制できます。

getReturnResultSetメソッドは、DML RETURNING機能で戻されたデータをフェッチし、ResultSetオブジェクトとして戻します。エラーが検出された場合は、SQLException例外が発生します。



ノート:

DML RETURNING 機能のための Oracle 固有の API は、Java Development Kit (JDK) 6.0 の場合は ojdbc6.jar、JDK 7 の場合は ojdbc7.jar に含まれています。

4.6.2 DML RETURNING文の実行について

DML RETURNING文を実行する前に、JDBCアプリケーションが、registerReturnParameterメソッドを1つ以上コールする必要があります。これらのメソッドは、戻りパラメータに関する型やサイズなどの情報をJDBCドライバに提供します。DML RETURNING文は、executeUpdateまたはexecuteのいずれかの標準JDBC APIを使用して処理されます。戻されたパラメータは、oracle.jdbc.OraclePreparedStatementインタフェースのgetReturnResultSetメソッドを使用して、ResultSetオブジェクトとしてフェッチできます。

ResultSetオブジェクトの値を読み取るには、基礎となるStatementオブジェクトをオープンする必要があります。基礎となるStatementオブジェクトがクローズされている場合は、戻されたResultSetオブジェクトもクローズ状態となります。この状態は、SQL問合せ文を処理することで取り出されたResultSetオブジェクトの状態と一致します。

DML RETURNING文を実行する場合、getReturnResultSetメソッドで戻されるResultSetオブジェクトの同時実行性はCONCUR_READ_ONLYであり、そのResultSetオブジェクト型はTYPE_FORWARD_ONLYまたはTYPE_SCROLL_INSENSITIVEである必要があります。

4.6.3 DML RETURNINGの例

ここでは、DML RETURNING機能に関する2つのコード例を示します。

次のコードは、DML RETURNING機能の使用例です。この例では、name列の最大サイズを100文字と仮定します。name列の最大サイズが判明しているため、registerReturnParameter(int paramIndex, int externalType, int maxSize)メソッドを使用します。

```
...
OraclePreparedStatement pstmt = (OraclePreparedStatement)conn.prepareStatement(
       "delete from tab1 where age < ? returning name into ?");
pstmt.setInt(1,18);
/** register returned parameter
  * in this case the maximum size of name is 100 chars
```

```
  */
pstmt.registerReturnParameter(2, OracleTypes.VARCHAR, 100);
// process the DML returning statement
count = pstmt.executeUpdate();
if (count>0)
{
  ResultSet rset = pstmt.getReturnResultSet(); //rest is not null and not empty
  while(rset.next())
  {
    String name = rset.getString(1);
    ...
  }
}
...
```

次のコードも、DML RETURNING機能の使用例です。ただし、この例では、戻りパラメータの最大サイズが不明です。したがって、registerReturnParameter(int paramIndex, int externalType)メソッドを使用します。

```
...
OraclePreparedStatement pstmt = (OraclePreparedStatement)conn.prepareStatement(
  "insert into lobtab values (100, empty_clob()) returning col1, col2 into ?, ?");
// register return parameters
pstmt.registerReturnParameter(1, OracleTypes.INTEGER);
pstmt.registerReturnParameter(2, OracleTypes.CLOB);
// process the DML returning SQL statement
pstmt.executeUpdate();
ResultSet rset = pstmt.getReturnResultSet();
int r;
CLOB clob;
if (rset.next())
{
  r = rset.getInt(1);
  System.out.println(r);
  clob = (CLOB)rset.getClob(2);
  ...
}
...
```

4.6.4 DML RETURNINGの制限事項

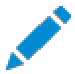
DML RETURNING機能を使用する場合は、次の点に注意してください。

- getReturnResultSetメソッドを複数回起動した場合は、その戻り値が不正確になります。そのため、これに関連するアクションには依存しないようにします。
- DML RETURNING文の実行で戻るResultSetオブジェクトは、ResultSetMetaData型をサポートしません。したがって、アプリケーションでは、DML RETURNING文を実行する前に、戻りパラメータの情報を把握している必要があります。
- DML RETURNINGでは、ストリームはサポートされません。
- DML RETURNING機能をバッチ更新と組み合せることはできません。
- 単一のSQL DML文の中で自動生成キー機能とDML RETURNING機能を両方とも使用することはできません。たとえば、次のような使用方法は許可されません。

```
...
PreparedStatement pstmt = conn.prepareStatement('insert into orders (?, ?, ?) returning
order_id into ?");
pstmt.setInt(1, seq01.NEXTVAL);
pstmt.setInt(2, 100);
pstmt.setInt(3, 966431502);
pstmt.registerReturnParam(4, OracleTypes.INTEGER);
pstmt.executeUpdate;
ResultSet rset = pstmt.getGeneratedKeys;
...
```

4.7 PL/SQL連想配列へのアクセス

Oracle JDBCドライバを使用すると、JDBCアプリケーションで、連想配列パラメータを使用してPL/SQLをコールできます。PL/SQLでは、連想配列はキーと値ペアのセットです。キーはPLS_INTEGERまたは文字列の場合があります。キーは任意の値であり、稠密である必要はありません。クライアント・アプリケーションから、PLS_INTEGERまたはBINARY_INTEGERキーのみを使用できます。



ノート:

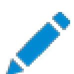
PLS_INTEGER と BINARY_INTEGER は同一のデータ・タイプです。

Oracle JDBCドライバの以前のリリースでは、スカラー・データ型のPL/SQL連想配列のみがサポートされていました。また、サポートは、配列のキーと値のペアの値のみに制限されていました。Oracle Databaseリリース18cでは、連想配列のキー(索引)と値の両方へのアクセスがサポートされ、オブジェクト型の連想配列もサポートされます。新しい機能を実現するには、次のメソッドを使用します。

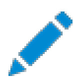
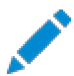
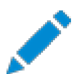
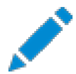
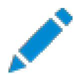
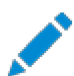
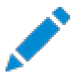
- ```
  Array createOracleArray(String arrayTypeName,
                          Object elements)
                          throws SQLException
  ```
- ```
  ARRAY createARRAY(String typeName,
                    Object elements)
                    throws SQLException
  ```

  前述のメソッドの両方で、2番目のパラメータは、連想配列のキーと値のペアを保持するjava.util.Map<Integer, ?>にすることも、または値の配列だけにすることもできます。これが値の配列である場合、JDBCドライバによって索引が0,1,2などにデフォルト設定されます。java.util.Map<Integer, ?>である場合、JDBCドライバによってキーはデフォルト設定されません。マップで指定されたままになり、疎と負の場合があります。
- ```
  Map<?,?> oracle.jdbc.OracleArray.getJavaMap();
  ```

  このメソッドは、連想配列のデータ型にMap<?,?>を戻し、ネストした表およびVARRAYにnullを戻します。

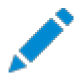
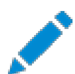
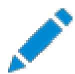
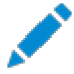
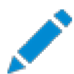
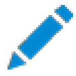
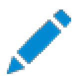
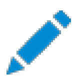
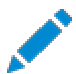
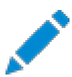
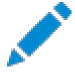
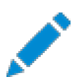
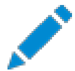


ノート:

- 連想配列は、以前は索引付き表と呼ばれていました。
  ```

- 文字列のデータ型を使用する場合、サイズは PL/SQL のサイズ(32767 文字)に制限されます。サーバー側内部ドライバの場合、制限は小さくなります。

関連項目:

- *Oracle Database JDBC Java API*リファレンス
- Oracle Database PL/SQL言語リファレンス

  連想配列の詳細は、これらを参照してください

5 JDBC Thin固有の機能

この章では、Java Database Connectivity (JDBC) Thinクライアントの概要、およびJDBC Thinドライバのみでサポートされている次の機能について説明します。

- JDBC Thinクライアントの概要
- サポートされる追加機能

5.1 JDBC Thinクライアントの概要

JDBCシン・クライアントは、Pure Javaのタイプ IVドライバです。このドライバは軽量で、インストールも簡単です。このドライバは、JDBC Oracle Call Interface(OCI)ドライバのパフォーマンスに匹敵する高いパフォーマンスを提供します。JDBC Thinドライバは、すべてJavaで記述されており、プラットフォームに依存しません。また、クライアント側に追加のOracleソフトウェアは必要ありません。

JDBC Thinドライバは、Oracle Databaseからデータにアクセスするために Oracleによって開発されたプロトコル、TTCを使用しているサーバーと通信します。これは、アプリケーション・サーバーに対して使用できます。このドライバは、Javaソケットの上にOracle NetとTTCを実装するTCP/IPの実装を提供することにより、データベースへの直接接続を可能とします。このプロトコルはどちらも、対応するサーバー上のプロトコルの軽量版実装です。Oracle Netプロトコルは TCP/IP経由でのみ実行されます。

JDBC Thinドライバは、クライアント側とサーバー側の両方で使用できます。クライアント側では、ドライバは、クライアント上または3層構成の中間層で実行されているJavaアプリケーションで使用できます。サーバー側では、このドライバが、リモートのOracle Databaseインスタンスまたは同じデータベースの別のセッションへのアクセスに使用されます。

5.2 サポートされるその他の機能

JDBC Thinドライバは、標準JDBC機能のすべてをサポートします。JDBC Thinドライバは、次の追加機能に対するサポートも提供しています。

- ネイティブXAのデフォルトでのサポート
- トランザクション・ガードのサポート
- アプリケーション・コンティニュイティのサポート

5.2.1 ネイティブXAのデフォルトでのサポート

JDBC Thinドライバには、JDBC OCIドライバと同様にネイティブXAに対するサポートが用意されています。ただし、JDBC Thinドライバはデフォルトでネイティブ XAをサポートします。この点は、ネイティブXAのサポートをデフォルトでは使用できないJDBC OCIドライバと異なります。

関連項目:

「Oracle JDBCドライバのネイティブXA」

5.2.2 トランザクション・ガードのサポート

トランザクション・ガード機能は、計画済および計画外の停止と重複発行の際に、実行を最大1回とするための汎用インフラストラクチャを提供します。トランザクション・ガード機能を(アプリケーション・コンティニュイティ機能とともに)使用すると、透過的なセッション・リカバリと、処理中のトランザクションの初めからSQL文(問合せおよびDML)のリプレイが可能になります。

関連項目:

Java用のトランザクション・ガード

5.2.3 アプリケーション・コンティニュイティのサポート

アプリケーション・コンティニュイティは、汎用目的のアプリケーションから独立したインフラストラクチャを提供します。これにより、計画済または計画外の停止発生後にアプリケーションからの作業のリカバリが可能になります。これには、次の利点があります。

- エンドユーザーから停止をマスク
- ユーザー環境、処理中のトランザクションおよび失われた結果のリカバリ
- アプリケーションがリカバリするための単一の容易で確実なメソッド
- 停止の有無にかかわらず、アプリケーションによるターゲットとの応答時間が明確

関連項目:

Java用のアプリケーション・コンティニュイティ

6 JDBC OCIドライバ固有の機能

この章では、Java Database Connectivity(JDBC)Oracle Call Interface(OCI)ドライバ固有の機能について説明します。OCI Instant Clientについても説明します。この章の構成は、次のとおりです。

- OCI接続プーリング
- 透過的アプリケーション・フェイルオーバー
- OCIネイティブXA
- OCI Instant Client
- Instant Client Light (English)について

6.1 OCI接続プーリング

OCI接続プーリング機能は、オラクル社設計の拡張機能です。JDBC OCIドライバで提供される接続プーリングによって、アプリケーションは少数の物理接続を使用して、複数の論理接続を保持できます。この論理接続でのコールは、指定された時間に使用可能な物理接続にルーティングされます。

関連項目:

OCI接続プーリング

6.2 透過的アプリケーション・フェイルオーバー

JDBC OCIドライバの透過的アプリケーション・フェイルオーバー機能を使用すると、接続先のデータベース・インスタンスがダウンした場合でも、データベースに自動的に再接続できます。別のノードで作成されても、新しいデータベース接続は元の接続とまったく同じです。

関連項目:

透過的アプリケーション・フェイルオーバー

6.3 OCIネイティブXA

JDBC OCIには、ネイティブXAと呼ばれる機能も用意されています。この機能により、ネイティブAPIを使用してXAコマンドを送信できます。非ネイティブAPIと比較して、ネイティブAPIを使用するほうが、高いパフォーマンスを達成できます。

関連トピック

- OCIネイティブXA

6.4 OCI Instant Client

この項の内容は次のとおりです。

- Instant Clientの概要
- OCI Instant Client共有ライブラリ
- Instant Clientの利点
- JDBC OCI Instant Clientのインストール手順
- Instant Clientの使用
- Instant Client共有ライブラリのパッチについて
- データ共有ライブラリとZIPファイルの再生成
- OCI Instant Client用のデータベース接続名
- OCI Instant Clientの環境変数

6.4.1 Instant Clientの概要

Instant Clientは、Oracleホームの必要性がなくなり、OCI、Oracle C++ Call Interface(OCCI)、Open Database Connectivity(ODBC)およびJDBC-OCIベースのカスタマ・アプリケーションのデプロイが非常に簡単になるようにパッケージ化されています。Instant Clientを使用するJDBC OCIアプリケーションに必要な記憶域の容量は、クライアント側に完全にインストールして実行するアプリケーションに比べて少なくなります。Instant Clientの共有ライブラリに使用されるディスク領域は、クライアント側に完全にインストールした場合の約1/4です。

6.4.2 OCI Instant Client共有ライブラリ

JDBC OCIアプリケーションのデプロイに必要なOracleのクライアント側ファイルが必要です。この表に記載されているライブラリ名はOracle Databaseリリース19cに対応しています。ライブラリ名の数字の部分は、リリースに応じて将来変更されます。

表6-1 OCI Instant Client共有ライブラリ

LinuxシステムとUNIXシステム	LinuxシステムとUNIXシステムの場合の説明	Microsoft Windows	Microsoft Windowsでの説明
libclntsh.so.19.1 libclntshcore.so.19.1<a id="fn_1" name="fn_1" href="#fn_1" onclick='footdisplay(1, " Beginning with Oracle Database 12c Release 1, the libclntshcore.so.<rel	クライアント・コード・ライブラリ	oci.dll	アプリケーションがリンクする転送機能

LinuxシステムとUNIXシステム	LinuxシステムとUNIXシステムの場合の説明	Microsoft Windows	Microsoft Windowsでの説明
ease number> library is separated from the libclntsh.so.脚注 1			
libociei.so 脚注 2	OCI Instant Client データの共有ライブラリ	oraociei19.dll	データとコード
libnnz19.so	セキュリティ・ライブラリ	orannzsbb19.dll	セキュリティ・ライブラリ
libocijdbc19.so	OCI Instant Client JDBC ライブラリ	ocijdbc19.dll	OCI Instant Client JDBC ライブラリ
すべての JDBC Java アーカイブ(JAR)ファイル	関連項目:「環境変数の確認」	すべての JDBC JAR ファイル	関連項目:「環境変数の確認」

脚注1

Oracle Database 12cリリース1以降、libclntshcore.so.<release number>ライブラリは、libclntsh.so.<release number>ライブラリおよびデータ共有ライブラリとは別個になっています。

脚注2

libclntsh.so.19.1ライブラリ、libclntshcore.so.19.1ライブラリおよびlibociei.soライブラリは、インスタント・クライアント・モードで動作するために同じディレクトリ内に存在する必要があります。

ノート:



ネイティブ XA 機能を提供するには、JDBC XA クラス・ライブラリをコピーする必要があります。UNIX システムでは、このライブラリ libheteroxa19.so は ORACLE_HOME/jdbc/lib ディレクトリに格納されています。Microsoft Windows では、このライブラリ heteroxa19.dll は ORACLE_HOME¥bin ディレクトリに格納されています。

6.4.3 Instant Clientの利点

Instant Clientには次のような利点があります。

- インストール時にコピーされるファイルの数がそれほど多くありません。
- Oracleのクライアント側で必要なファイルの数と合計ディスク領域が大幅に減少します。
- Instant Clientでデプロイされたアプリケーションでは、機能またはパフォーマンスに損失がありません。
- 独立したソフトウェア・ベンダーは、アプリケーションを簡単にパッケージ化できます。

6.4.4 JDBC OCI Instant Clientのインストール手順

Instant Clientライブラリは、Oracle Universal InstallerでInstant Clientオプションを選択するとインストールできます。Oracle Technology NetworkのWebサイトからダウンロードすることもできます。インストールの手順は、次のとおりです。

1. Instant Client共有ライブラリおよびOracle JDBCクラス・ライブラリをダウンロードし、instantclientなどのディレクトリにインストールします。
2. ライブラリ・パスの環境変数を、ステップ1のディレクトリに設定します。たとえば、UNIXシステムでは、LD_LIBRARY_PATH環境変数をinstantclientに設定します。Microsoft Windowsでは、PATH環境変数をinstantclientディレクトリに設定します。
3. CLASSPATH環境変数に、JDBCクラス・ライブラリの完全パス名を追加します。

これらのステップを完了した後、JDBC OCIアプリケーションを実行する準備が整いました。

Instant Clientを使用する場合、JDBC OCIアプリケーションはOCIおよびJDBC共有ライブラリにライブラリ・パス環境変数を介してアクセスできます。この場合、ORACLE_HOMEへの依存性はなく、ORACLE_HOMEに用意されている他のコードおよびデータファイルは、tnsnames.oraファイルを除いて、JDBC OCIには不要です。

Instant Clientは、Oracle Universal InstallerでInstant Clientオプションを選択するとインストールできます。Instant Clientファイルは、必ず空のディレクトリにインストールしてください。Instant Clientを使用するには、OTNのインストールと同じように、LD_LIBRARY_PATH環境変数をInstant Clientディレクトリに設定する必要があります。

管理者オプションを選択して完全なクライアント・インストールを実行した場合は、Instant Client共有ライブラリもインストールされます。完全なクライアント・インストールでは、Instant Client共有ライブラリおよびJDBCクラスは次の場所に格納されます。

LinuxシステムまたはUNIXシステムの場合:

- libociei.soライブラリは、$ORACLE_HOME/instantclientにあります。
- libclnstsh.so.18.1、libocijdbc18.soおよびlibnnz18.soは、$ORACLE_HOME/libに格納されます。
- JDBCクラス・ライブラリは、$ORACLE_HOME/jdbc/libに格納されます。

Microsoft Windowsの場合:

- oraociei18.dllライブラリは、ORACLE_HOME¥instantclientにあります。
- oci.dll、ocijdbc18.dllおよびorannzsbb18.dllは、ORACLE_HOME¥binに格納されます。
- JDBCクラス・ライブラリはORACLE_HOME¥jdbc¥libに格納されます。

これらのファイルを別のディレクトリにコピーし、そのディレクトリにライブラリ・パスを設定し、JDBCクラス・ライブラリのパス名をCLASSPATH環境変数に追加すると、Instant Clientを使用するJDBC OCIアプリケーションを実行できます。

ノート:



- ネイティブ XA 機能を提供するには、JDBC XA クラス・ライブラリをコピーする必要があります。UNIX では、このライブラリ libheteroxa18.so は ORACLE_HOME/jdbc/lib に格納されます。Windows では、このライブラリ heteroxa18.dll は ORACLE_HOME¥bin に格納されます。

- すべてのライブラリは、同じ ORACLE_HOME からコピーし、同じディレクトリに格納する必要があります。

- Sparc64 などのハイブリッド・プラットフォームでは、JDBC OCI ドライバで Instant Client ライブラリを使用する必要がある場合、libociei.so ライブラリを ORACLE_HOME/instantclient32 ディレクトリからコピーする必要があります。JDBC OCI Instant Client に必要なその他すべての Sparc64 ライブラリは、ORACLE_HOME/lib32 ディレクトリからコピーしてください。

- ライブラリ・パス環境変数で、Oracle ライブラリのセットを 1 つのみ指定する必要があります。つまり、複数のディレクトリに Instant Client ライブラリが含まれている場合、ライブラリ・パス環境変数で、該当するディレクトリを 1 つのみ指定します。

- コンピュータ上で Oracle ホームを使用している場合、ライブラリ・パス環境変数での指定順序に関係なく、ライブラリ・パス環境変数で ORACLE_HOME/lib と Instant Client ディレクトリを同時に使用しないでください。つまり、ライブラリ・パス環境変数で、ORACLE_HOME/lib ディレクトリ(Instant Client 以外で動作している場合)または Instant Client ディレクトリ(Instant Client で動作している場合)のいずれか一方のみを指定します。

- Instant Client は、Oracle Technology Network(OTN)からダウンロードすることをお薦めします。

  https://www.oracle.com/technetwork/database/database-technologies/instant-client/overview/index.html

6.4.5 Instant Clientの使用

Instant Clientはデプロイメント用の機能であるため、本番アプリケーションの実行に使用します。デプロイメントでは、デモ用のプログラムなどにアクセスできるようにするために、完全インストールが必要です。一般的に、Instant Clientがクライアント側動作専用でないかぎり、Instant Clientを使用するアプリケーションではすべてのJDBC OCI機能を使用できます。したがって、サーバー側の外部プロシージャは、Instant Clientを使用できません。

6.4.6 Instant Client共有ライブラリのパッチについて

Instant Clientはデプロイメント用の機能であるため、JDBC OCIアプリケーションの実行に必要なファイルの数とサイズの減少に重点が置かれています。したがって、Instant Client共有ライブラリのパッチに必要なすべてのファイルがInstant Clientのデプロイ時に揃っているわけではありません。Instant Client共有ライブラリのパッチには、ORACLE_HOMEベースの完全クライアント・インストールが必要です。opatchユーティリティを使用して、Instant Client共有ライブラリにパッチを適用します。



ノート:

Microsoft Windows 上では、共有ライブラリにパッチを当てることはできません。

ORACLE_HOME環境でパッチを適用した後で、表6-1に示されているファイルを、インスタント・クライアント・ディレクトリにコピーします。

個々のファイルをコピーするかわりに、OCI、OCCI、JDBCおよびSQL*PlusのInstant Client ZIPファイルを生成できます。

次に、ZIPファイルをターゲット・コンピュータにコピーして解凍できます。

opatchユーティリティは、libclnstsh.so.18.1にORACLE_HOMEインストールのパッチ情報を格納します。この情報を取り出すには、次のコマンドを使用します。

```
genezi -v
```

Instant Clientのデプロイ元のコンピュータにgeneziユーティリティが装備されていない場合、ORACLE_HOMEインストールが装備されたコンピュータ上のORACLE_HOME/binディレクトリからコピーする必要があります。

関連トピック

- JDBC OCI Instant Clientのインストール手順

6.4.7 データ共有ライブラリとZIPファイルの再生成

ORACLE_HOMEを管理者オプションでインストールした場合は、次のステップを実行して、OCI Instant Clientデータ共有ライブラリ(libociei.so)を再生成できます。

```
mkdir -p $ORACLE_HOME/rdbms/install/instantclient/light
cd $ORACLE_HOME/rdbms/lib
make -f ins_rdbms.mk ilibociei
```

ORACLE_HOME内の現行ファイルに基づく新しいバージョンのlibociei.soデータ共有ライブラリが、ORACLE_HOME/rdbms/install/instantclientディレクトリに配置されます。

再生成されたデータ共有ライブラリlibociei.soの場所は、元のデータ共有ライブラリlibociei.soの場所と異なることに注視してください。これはORACLE_HOME/instantclientディレクトリにあります。前述のステップで、OCI、OCCI、JDBCおよびSQL*Plus用のInstant Client ZIPファイルも生成されます。

Microsoft Windowsプラットフォームでは、データ共有ライブラリとZIPファイルを再生成することはできません。

6.4.8 OCI Instant Client用のデータベース接続名

tnsnames.oraやsqlnet.oraなどの構成ファイルの格納先としてORACLE_HOMEまたはTNS_ADMIN環境変数を使用しないOracle Netネーミング・メソッドは、すべてInstant Clientを使用します。特に、接続文字列は次のような形式で指定できます。

- 次のようなThin形式の接続文字列の場合:

  ```
  host:port:service_name
  ```

  たとえば:

  ```
  url="jdbc:oracle:oci:@example.com:5521:orcl"
  ```

- 次のようなSQL接続URL文字列の場合:

  ```
  //host:[port][/service_name]
  ```

  たとえば:

```
url="jdbc:oracle:oci:@//example.com:5521/orcl"
```

- Oracle Netのキーワード値ペア。たとえば:

```
url="jdbc:oracle:oci:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
  (HOST=localhost) (PORT=5521))
  (CONNECT_DATA=(SERVICE_NAME=orcl)))"
```

構成ファイルの格納先としてTNS_ADMINが必要なネーミング・メソッドをそのまま使用するには、TNS_ADMIN環境変数を設定します。

関連項目:

接続フォーマットの詳細は、『Oracle Database Net Services管理者ガイド』を参照してください

TNS_ADMIN環境変数を設定せずに、inst1などのTNSNAMESエントリを使用する場合は、ORACLE_HOME環境変数を設定し、構成ファイルを$ORACLE_HOME/network/admin ディレクトリに格納する必要があります。



ノート:

この場合、ORACLE_HOME 環境変数は、Oracle Net 構成ファイルの格納先としてのみ使用されます。それ以外のクライアント・コード・ライブラリのコンポーネントでは、ORACLE_HOME 環境変数の値は使用されません。

空の接続文字列は、サポートされません。ただし、空の接続文字列の使用に代わる方法として、tnsnames.oraエントリまたはOracle Netキーワード値ペアに対して、UNIXシステム上ではTWO_TASK環境変数を、Microsoft Windows上ではLOCAL変数を設定する方法があります。TWO_TASKまたはLOCALがtnsnames.oraエントリに設定されている場合、TNS_ADMINまたはORACLE_HOME設定によってtnsnames.oraファイルをロードする必要があります。

例

データベース・サーバー上のlistener.oraファイルに次の情報が格納されているとします。

```
LISTENER = (ADDRESS_LIST=(ADDRESS=(PROTOCOL=tcp)(HOST=server6)(PORT=5221)))
SID_LIST_LISTENER = (SID_LIST=
                     (SID_DESC=(SID_NAME=rdbms3)
                     (GLOBAL_DBNAME=rdbms3.server6.com)
                     (ORACLE_HOME=/home/dba/rdbms3/oracle)))
```

このサーバーには、次のいずれかの方法で接続できます。

```
url = "jdbc:oracle:oci:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
                     (HOST=server6)(PORT=5221))
                     (CONNECT_DATA=(SERVICE_NAME=rdbms3.server6.com)))"
```

または

```
url = "jdbc:oracle:oci:@//server6:5221/rdbms3.server6.com"
```

または、TWO_TASK環境変数を任意の接続文字列に設定すれば、sqlplusコマンドとともに接続文字列を指定しなくても、デー

タベース・サーバーに接続できます。たとえば、次の方法のいずれかを使用してTWO_TASK環境変数を設定します。

```
setenv TWO_TASK "(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=server6)(PORT=5221))
                 (CONNECT_DATA=(SERVICE_NAME=rdbms3.server6.com)))"
```

または

```
setenv TWO_TASK //server6:5221/rdbms3.server6.com
```

これで、次のURLを使用すると、データベース・サーバーに接続できます。

```
url = "jdbc:oracle:oci:@"
```

接続文字列は、tnsnames.oraファイルに格納できます。たとえば、tnsnames.oraファイルに次の情報が格納されているとします。

```
conn_str = (DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=server6)(PORT=5221))
                (CONNECT_DATA=(SERVICE_NAME=rdbms3.server6.com)))
```

このtnsnames.oraファイルが/home/webuser/instantclientディレクトリに配置されている場合、次のように、TNS_ADMIN環境変数(または、Microsoft WindowsではLOCAL)を設定できます。

```
setenv TNS_ADMIN /home/webuser/instantclient
```

これで、次のように接続できます。

```
url = "jdbc:oracle:oci:@conn_str"
```



ノート:

TNS_ADMIN 環境変数では、tnsnames.ora ファイルが配置されているディレクトリを指定します。ただし、TNS_ADMIN では、tnsnames.ora ファイルのフルパスを指定するのではなく、ディレクトリを指定します。

このtnsnames.oraファイルが、Oracleホームの/network/server6/home/dba/oracle/network/adminディレクトリに配置されている場合、TNS_ADMINを使用してtnsnames.oraファイルを配置するかわりに、次のようにORACLE_HOME環境変数を使用できます。

```
setenv ORACLE_HOME /network/server6/home/dba/oracle
```

これで、先に指定したように、接続文字列conn_strのいずれかを使用して接続できます。

TNS_ADMINまたはORACLE_HOMEによってtnsnames.oraを特定できる場合、TWO_TASKは次のように設定できます。

```
setenv TWO_TASK conn_str
```

次のURLで接続できます。

```
url = "jdbc:oracle:oci:@"
```

6.4.9 OCI Instant Clientの環境変数

ORACLE_HOME環境変数は、NLS、COREおよびエラー・メッセージ・ファイルの位置を決定するものではなくなりました。OCIのみのアプリケーションでは、ORACLE_HOME環境変数を設定する必要はありません。ただし、変数が設定されていても、OCIドライバの動作に影響はありません。OCIドライバは、常にデータ共有ライブラリからドライバのデータを入手します。データ共有ライブラリが利用できない場合のみ、ORACLE_HOME環境変数が使用され、フル・クライアント・インストールとみなされます。ORACLE_HOME環境変数は設定不要ですが、設定する場合は、ディレクトリを識別する有効なオペレーティング・システム・パス名に設定する必要があります。

Instant Clientの使用中、環境変数ORA_NLS10およびORA_NLSPROFILES33は無視されます。

ORA_TZFILE変数が設定されていない場合、Instant Clientでは、デフォルト設定のサイズの大きいtimezlrg_n.datファイルがデータ共有ライブラリから使用されます。データ共有ライブラリのより小さいtimezone_n.datファイルが使用される場合は、絶対パス名または相対パス名を使用せずに、ORA_TZFILE環境変数をファイルの名前に設定します。つまり、次のように設定します。

UNIXシステムの場合:

```
setenv ORA_TZFILE timezone_n.dat
```

Microsoft Windowsの場合:

```
set ORA_TZFILE timezone_n.dat
```

前述の例では、nはタイムゾーン・データファイルのバージョン番号です。

OCIドライバがデータ共有ライブラリを利用できないためにInstant Clientを使用していない場合、ORA_TZFILE変数を設定する際は、以前のOracle Databaseのリリースのときと同様に完全パス名を指定します。

TNSNAMESのエントリを使用する場合は、前述のように、TNS_ADMINディレクトリにTNSNAMES構成ファイルを格納する必要があります。TNS_ADMINを設定しない場合は、ORACLE_HOME/network/adminディレクトリにOracle Net Services構成ファイルを格納する必要があります。

6.5 Instant Client Light (English)について

Instant Clientの軽量バージョンは、Instant Client Light(English)と呼ばれます。Instant Client Lightは、短いファイル名です。Instant Client Lightは、Instant Clientを大幅に小さくしたバージョンです。クライアント・インストールのディスク空き容量要件は、約63MB少なくてすみます。これは、サイズが4MBの軽量データ共有ライブラリ(UNIXシステムの場合libociicus.so)によるもので、これはサイズが67MBのデータ共有ライブラリlibociei.soのサブセットです。

この軽量データ共有ライブラリは、少数の文字セットとエラー・メッセージを英語のみでサポートしています。そのため、Instant Client Light(English)という名前が付いています。Instant Client Lightは、英語のエラー・メッセージのみを必要とし、US7ASCIIかWE8DEC、またはUnicode文字セットの1つを使用するアプリケーションに対応して設計されています。

この項の内容は次のとおりです。

- Instant Client Light (English)のデータ共有ライブラリ

- グローバリゼーション設定
- 操作
- Instant Client Light (English)のインストール

6.5.1 Instant Client Light (English)のデータ共有ライブラリ

表6-2では、様々なプラットフォームにおけるInstant ClientとInstant Client Light(English)用のデータ共有ライブラリ名を示しています。この表では、各データ共有ライブラリのサイズもライブラリ・ファイル名に続くカッコ内に記載しています。

表6-2 Instant ClientとInstant Client Light(English)用のデータ共有ライブラリ

プラットフォーム	Instant Client	Instant Client Light(English)
Solaris	libociei.so (67 MB)	libociicus.so (4 MB)
Linux	libociei.so (67 MB)	libociicus.so (4 MB)
Microsoft Windows	oraociei19.dll (85 MB)	oraociicus19.dll (15 MB)

6.5.2 グローバリゼーション設定

NLS_LANG設定は、言語、地域および文字セットを、language_territory.charactersetとして指定します。Instant Client Lightでは、languageはAmericanのみ、territoryはサポートされる任意の値、charactersetは次のいずれかです。

- シングルバイト
  - US7ASCII
  - WE8DEC
  - WE8MSWIN1252
  - WE8ISO8859P1
- Unicode
  - UTF8
  - AL16UTF16
  - AL32UTF8

クライアントやサーバーの文字セットとしてあげられている以外の文字セットまたは各国語文字セットを指定したり、クライアントのNLS_LANGに言語を設定したりすると、次のエラーが発生します。

- ORA-12734
- ORA-12735
- ORA-12736
- ORA-12737

Instant Client Lightでは、エラー・メッセージは英語でのみ出力されます。したがって、NLS_LANG設定に有効な値は、次のよ

うになります。

```
American_territory.characterset
```

territoryにはサポートされる有効な地域を、charactersetには前述のいずれかの文字セットを指定できます。

Instant Client Lightは、OCI_UTF16モードで作成されたOCI環境ハンドルを使用して動作します。

関連項目:

NLS設定の詳細は、『Oracle Databaseグローバリゼーション・サポート・ガイド』を参照してください

6.5.3 操作

Instant Client Lightを使用するには、アプリケーションで、UNIXシステムの場合はLD_LIBARARY_PATH環境変数を、Microsoft Windowsの場合はPATH環境変数を、クライアントおよびデータ共有ライブラリが格納されている場所に設定する必要があります。OCIアプリケーションは、デフォルトでOCIデータ共有ライブラリ、つまりUNIXシステムではLD_LIBRARY_PATH環境変数のlibociei.soを、Microsoft WindowsではPATH環境変数のoraociei18.dllデータ共有ライブラリを検索して、アプリケーションがInstant Clientを使用するかどうかを判断します。このライブラリが見つからない場合、OCIは、Instant Client Lightのデータ共有ライブラリ、つまりUNIXシステムではlibociicus.so、Microsoft Windowsではlibociicus18.dllをロードしようとします。このライブラリが見つかった場合、アプリケーションはInstant Client Lightを使用します。このライブラリが見つからない場合は、Instant Client以外を使用します。

6.5.4 Instant Client Light (English)のインストール

Instant Client Lightは、次のいずれかの方法でインストールできます。

- OTNから

  次のサイトから必要なファイルをダウンロードできます。

  https://www.oracle.com/technetwork/database/database-technologies/instant-client/overview/index.html

  Instant Client Lightの場合は、Basicパッケージをダウンロードして展開するかわりに、Basic Lightパッケージをダウンロードして解凍します。解凍する前に、軽量ライブラリを解凍するディレクトリを空にしてください。

- クライアントの完全(Adminオプション)インストールから

  libociei.soまたはoraociei18.dllをORACLE_HOME/instantclientディレクトリからコピーするかわりに、libociicus.soまたはoraociic18.dllをORACLE_HOME/instantclient/lightディレクトリからコピーします。つまり、UNIXシステムの場合、LD_LIBRARY_PATH環境変数のInstant Clientディレクトリに、大規模なOCI Instant Clientのデータ共有ライブラリlibociei.soではなく、Instant Client Lightのデータ共有ライブラリlibociicus.soを格納してください。Microsoft Windowsの場合、PATH環境変数にはoraociei18.dllではなくoraociicus18.dllを指定する必要があります。

- Oracle Universal Installerから

  Oracle Universal InstallerでInstant Clientオプションを選択すると、libociei.so (Microsoft Windowsの

場合はoraociei18.dll)が、インストールのベース・ディレクトリにインストールされ、LD_LIBRARY_PATH環境変数には、このディレクトリが指定されます。このため、Instant Client Lightはデフォルトでは使用できません。Instant Client Lightのデータ共有ライブラリlibociicus.so (Microsoft Windowsの場合はoraociicus18.dll)は、ベース・ディレクトリのlightサブディレクトリにインストールされます。したがって、Instant Client Lightを使用するには、OCIデータ共有ライブラリのlibociei.so (Windowsではoraociei18.dll)を削除するか、名前を変更し、Instant Client Lightのデータ共有ライブラリを、インストールのlightサブディレクトリからベース・ディレクトリにコピーする必要があります。

たとえば、Oracle Universal InstallerによってLD_LIBRARY_PATH環境変数のmy_oraic_18_1ディレクトリにInstant Clientをインストールした場合、Instant Client Lightを使用するには、次のコマンドを実行する必要があります。

```
cd my_oraic_18_1
rm libociei.so
mv light/libociicus.so .
```



ノート:

Instant Client ファイルは、必ず空のディレクトリにコピーまたはインストールしてください。これは、インストール時に互換性のないバイナリが存在しないようにするためです。

7 サーバー側内部ドライバ

この章の構成は、次のとおりです。

- サーバー側内部ドライバの概要
- データベースへの接続
- セッション・コンテキストおよびトランザクション・コンテキストについて
- サーバー上でのJDBCのテスト
- サーバーへのアプリケーションのロード

7.1 サーバー側内部ドライバの概要

サーバー側内部ドライバは、Oracle DatabaseとOracle Java仮想マシン(Oracle JVM)とも呼ばれる埋込みJava仮想マシンに結び付けられています。このドライバは、データベースと同じプロセスの一部として動作します。また、デフォルトのセッション(Oracle JVMが起動されたセッションと同じセッション)内で動作します。各Oracle JVMセッションには、存在するデータベース・セッションに対して1つの暗黙的なネイティブ接続があります。この接続は概念的で、Javaオブジェクトではありません。これはこのセッションに固有な側面であり、JVM内でオープンまたはクローズできません。

サーバー側内部ドライバはデータベース・サーバー内で動作するよう最適化されており、このドライバを使用するとローカル・データベース上のSQLデータおよびPL/SQLサブプログラムに直接アクセスできます。JVM全体は、データベースおよびSQLエンジンと同じアドレス空間内で動作します。SQLエンジンへのアクセスはファンクション・コールです。これによって、Java Database Connectivity(JDBC)アプリケーションのパフォーマンスが向上し、SQLエンジンへのアクセスにリモートOracle Netコールを実行するよりも速くなります。

サーバー側内部ドライバは、クライアント側ドライバと同じ機能、Application Program Interface(API)およびOracle拡張機能をサポートします。これにより、アプリケーションのパーティション化が非常に簡単になります。たとえば、データ集中処理型のJavaアプリケーションがある場合は、アプリケーション固有のコールを修正しなくても、パフォーマンスを向上させるために簡単にデータベース・サーバーに移動できます。

7.2 データベースへの接続

前の項で説明したように、サーバー側内部ドライバはデフォルトのセッション内で動作します。したがって、すでに接続された状態になっています。デフォルト接続にアクセスするには、次の2つのメソッドを使用できます。

- 次のいずれかの形式をURL文字列として、`OracleDataSource.getConnection`メソッドを使用します。
  - `jdbc:oracle:kprb`
  - `jdbc:default:connection`
  - `jdbc:oracle:kprb:`
  - `jdbc:default:connection:`
- `OracleDriver`クラスのOracle固有の`defaultConnection`メソッドを使用します。

通常はdefaultConnectionの使用をお薦めします。



ノート:

サーバー側内部ドライバと接続するためにOracleDriverクラスを登録する必要はなくなりました。

OracleDriverクラスのdefaultConnectionメソッドによる接続

oracle.jdbc.OracleDriverクラスのdefaultConnectionメソッドは、Oracle拡張機能で、常に同じ接続オブジェクトを戻します。発生した接続オブジェクトを別の変数名に割り当て、このメソッドを複数回コールしたとしても、1つの接続オブジェクトのみが再利用されます。

defaultConnectionコールに接続文字列を含める必要はありません。たとえば:

```
import java.sql.*;
import oracle.jdbc.*;

class JDBCConnection
{
  public static Connection connect() throws SQLException
  {
    Connection conn = null;
    try {
      // connect with the server-side internal driver

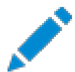
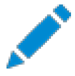
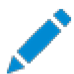
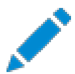
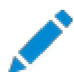
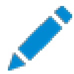
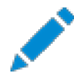
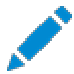
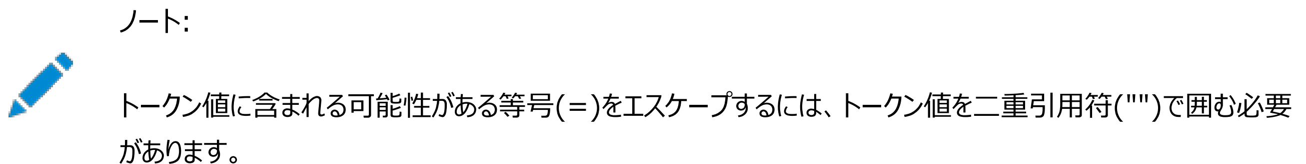
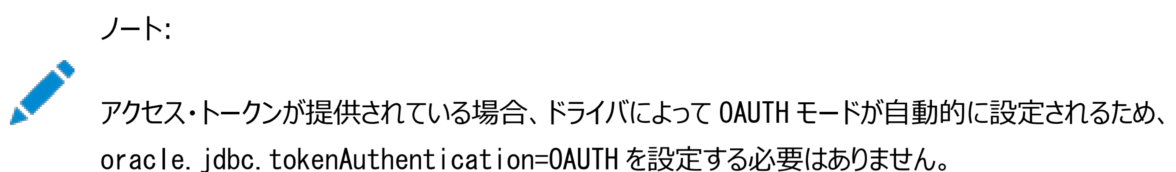
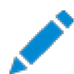
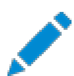
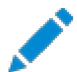
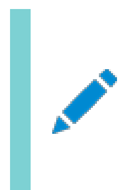
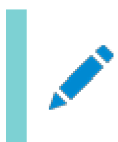
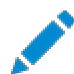
         conn = ora.defaultConnection();
      }

    } catch (SQLException e) {...}
    return conn;
  }
}
```

この例にはconn.closeコールがないことに注意してください。JDBCコードがターゲット・サーバー内で実行されている場合、接続は暗黙的なデータ・チャネルで、クライアントからの場合のように明示的な接続インスタンスではありません。クローズしないでください。

OracleDriverは、デフォルトの接続インスタンスを格納するための静的な値です。接続が存在しクローズされていない場合に、メソッドOracleDriver.defaultConnectionはこのデフォルトの接続インスタンスを戻します。それ以外の場合、新しいオープン・インスタンスを作成し、静的な値に格納してコール元に戻します。

通常、OracleDriver.defaultConnectionメソッドを使用します。このメソッドは、高速でリソースをあまり使用しません。Javaストアド・プロシージャは注意深く記述する必要があります。たとえば、各コールを終了する前に文をクローズします。

通常、デフォルトの接続インスタンスはクローズしないでください。このインスタンスは複数の場所に格納される可能性がある単一インスタンスであり、クローズするとそれぞれの場所が使用できなくなります。クローズすると、それ以降にOracleDriver.defaultConnectionメソッドをコールしたときに、新しいオープン・インスタンスが作成されます。

OracleDataSource.getConnectionメソッドによる接続

ターゲット・サーバー内で実行中のコードから内部サーバー接続に接続するには、次のいずれかのURLとともに、OracleDataSource.getConnectionメソッドを使用できます。

```
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:kprb");
Connection conn = ods.getConnection();
```

または

```
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:default:connection");
Connection conn = ods.getConnection();
```

URL内で指定したユーザー名またはパスワードは、デフォルトのサーバー接続への接続では無視されます。

OracleDataSource.getConnectionメソッドをコールするたびに、このメソッドは新しいJava Connectionオブジェクトを戻します。オブジェクト・マップまたは型マップを操作している場合は、OracleDataSource.getConnectionをコールするたびに、このメソッドが新しい接続オブジェクトを戻すということに重要な意味があります。型マップは、特定のConnectionオブジェクトおよびそのオブジェクトの一部である状態に関連付けられます。プログラムの一部として複数の型マップを使用する場合は、getConnectionをコールして、各型マップに対して新しいConnectionオブジェクトを作成できます。



ノート:

OracleDataSource.getConnection メソッドはコールするたびに新規オブジェクトを戻しますが、毎回、新しいデータベース接続を作成するわけではありません。同じ暗黙的なネイティブ接続を使用し、同じセッション状態(特にローカル・トランザクション)を共有します。

7.3 セッション・コンテキストおよびトランザクション・コンテキストについて

サーバー側ドライバは、デフォルト・セッションおよびデフォルト・トランザクションのコンテキストで動作します。デフォルト・セッションとは、JVMが起動されたセッションです。サーバー上では、事実上データベースにすでに接続されています。これは、デフォルト・セッションがないクライアント側とは異なります。クライアント側では、明示的にデータベースに接続する必要があります。

サーバーでは、自動コミット・モードは無効になっています。接続オブジェクトで適切なメソッドを使用して、明示的にトランザクションのCOMMITおよびROLLBACK操作を管理する必要があります。

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

または

```
conn.rollback();
```



ノート:

ベスト・プラクティスとして、サーバー内ではトランザクションをコミットまたはロールバックしないことをお薦めします。

7.4 サーバー上でのJDBCのテスト

クライアント上で実行できるJDBCプログラムはほとんどすべてサーバー上でも実行できます。samplesディレクトリ内のすべてのプログラムは、少し修正するのみでサーバー上で実行できます。通常、修正は接続文に関するもののみです。

データベースへの接続を取得するための次のようなコード・フラグメントについて考えてみます。

```
ods.setUrl(
"jdbc:oracle:oci:@(DESCRIPTION=
  (ADDRESS=(PROTOCOL=TCP)(HOST=cluster_alias)
    (PORT=5221))
    (CONNECT_DATA=(SERVICE_NAME=orcl)))");
ods.setUser("HR");
ods.setPassword("hr");
Connection conn = ods.getConnection();
```

サーバー側内部ドライバで使用できるように、このコード・フラグメントを変更できます。サーバー側内部ドライバでは、ユーザー、パスワードまたはデータベースの情報は不要です。接続文は次のように記述します。

```
ods.setUrl(
"jdbc:oracle:kprb:@");
Connection conn = ods.getConnection();
```

ただし、接続を取得するには、次のようにOracleDriver.defaultConnectionメソッドをコールするのが最も便利な方法です。

```
Connection conn = OracleDriver.defaultConnection();
```

7.5 サーバーへのアプリケーションのロード

サーバーにアプリケーションをロードする場合は、クライアントでコンパイル済の.classファイルをロードするか、.javaソース・ファイルをロードして、サーバーで自動的にコンパイルできます。

7.5.1 loadjavaユーティリティの使用

loadjavaユーティリティを使用してファイルをロードします。コマンドラインでソース・ファイル名を指定するか、Javaアーカイブ(JAR)ファイルにそのファイルを格納して、コマンドラインでそのJARファイル名を指定します。

実際のユーティリティを実行するloadjavaスクリプトは、Oracleホームのbinディレクトリにあります。このディレクトリは、Oracleをインストールすると自動的に作成されます。

ノート:

loadjava ユーティリティは圧縮ファイルをサポートしています。

サーバーへのクラス・ファイルのロード

アプリケーションに3つのクラス・ファイルFoo1.class、Foo2.classおよびFoo3.classがある場合を考えてみます。各クラスは、サーバーで独自のクラス・スキーマ・オブジェクトに書き込まれます。

次のように、JDBC Oracle Call Interface(OCI)のデフォルト・ドライバを使用して、クラス・ファイルをロードできます。

- クラス・ファイル名を個別に指定します。

```
loadjava -user HR Foo1.class Foo2.class Foo3.class
Password: password
```

- ワイルドカードを使用してクラス・ファイル名を指定します。

```
loadjava -user HR Foo*.class
Password: password
```

- クラス・ファイルが格納されているJARファイルを指定します。

```
loadjava -user HR Foo.jar
Password: password
```

次のように、JDBC Thinドライバを使用してファイルをロードできます。

```
loadjava -thin -user HR@localhost:5221:orcl Foo.jar
Password: password
```



ノート:

Oracle Database 12c リリース 1 (12.1)以降、JDK 6 および JDK 7 がサポートされています。ただし、ある時点でアクティブな JVM は 1 つのみです。

クラスは、サーバーのアクティブなランタイム・バージョンよりも新しいバージョンの JDK を使用してコンパイルされないようにしてください。

サーバーへのソース・ファイルのロード

.javaソース・ファイルをロードする際にloadjava -resolveオプションを有効にすると、サーバー側コンパイラはロード時にアプリケーションをコンパイルします。その結果、オリジナル・ソース・コードのソース・スキーマ・オブジェクトとコンパイル済出力の1つ以上のクラス・スキーマ・オブジェクトの両方が生成されます。

-resolveを指定しない場合は、ソースはコンパイルされずに、ソース・スキーマ・オブジェクトにロードされます。ただし、この場合は、ソースで定義されたクラスを最初に使用しようとしたときに、ソースが暗黙的にコンパイルされます。

たとえば、デフォルトのJDBC OCIドライバを使用して、Foo.javaをロードしてコンパイルするには、次のようにloadjavaを実行します。

```
loadjava -user HR -resolve Foo.java
Password: password
```

あるいは、次のコマンドを入力し、JDBC Thinドライバを使用してロードします。

```
loadjava -thin -user HR@localhost:5221:orcl -resolve Foo.java
Password: password
```

いずれの方法でも、ソース・スキーマ・オブジェクトに加えて、適切なクラス・スキーマ・オブジェクトが作成されます。



ノート:

通常はできるかぎりクライアントでソースをコンパイルし、ソース・ファイルではなく、.class ファイルをサーバーにロードすることをお薦めします。

関連項目:

『Oracle Database Java開発者ガイド』

7.5.2 JVMコマンドラインの使用

JVMコマンドライン・オプションを使用してファイルをロードすることもできます。Oracle JVMへのコマンドライン・インタフェースは、JDKまたはJREのシェル・コマンドの使用方法と似ています。次のことが可能です。

- 標準の-classpath構文を使用して、ロードするクラスの検索場所を示します。
- 標準の-D構文を使用して、システム・プロパティを設定します。

このインタフェースは、文字列(VARCHAR2)引数を取り、この引数をコマンドライン入力として解析するPL/SQLファンクションです。形式が適切であれば、Oracle JVMで指定のJavaメソッドを実行します。これを実行するために、PL/SQLのパッケージDBMS_JAVAには次のファンクションが用意されています。

- runjava

  runjavaファンクションを使用するには、次のようにします。

  ```
  FUNCTION runjava(cmdline VARCHAR2) RETURN VARCHAR2;
  ```

- runjava_in_current_session

  runjava_in_current_sessionファンクションを使用するには、次のようにします。

  ```
  FUNCTION runjava_in_current_session(cmdline VARCHAR2) RETURN VARCHAR2;
  ```



ノート:

Oracle Database 11g リリース 1 以降には、Oracle JVM 環境用の Just-In-Time(JIT)コンパイラがあります。Oracle JVM の JIT コンパイラでは、以前のネイティブ・コンパイラと比較して高度な技術を使用し、動的に生成されたコードをコンパイルするため、実行速度が上がります。以前のネイティブ・コンパイラと異なり、JIT コンパイラには C コンパイラが必要ありません。プラグインのサポートがなくても有効です。

第III部 接続とセキュリティ

この部では、データソースおよびURLを使用したデータベースへの接続について説明します。Oracle Java Database Connectivity (JDBC) Oracle Call Interface (OCI)ドライバとThinドライバでサポートされているセキュリティ機能、JDBC ThinドライバでのTLS (Transport Layer Security)サポート、プロキシ接続を介した中間層認証についても説明します。

第III部は、次の章で構成されています。

- データソースおよびURL
- JDBCクライアント側セキュリティ機能
- プロキシ認証

8 データソースおよびURL

この章では、Java Database Connectivity(JDBC)データソースを使用したアプリケーションのデータベースへの接続と、データベースを記述するURLについて説明します。この章の構成は、次のとおりです。

- データソースについて
- データベースURLとデータベース指定子

8.1 データソースについて

データソースは、使用するデータベースまたはその他のリソースを指定するための標準汎用オブジェクトです。データソースの概念は、JDBC 2.0 Extension Application Program Interface(API)で導入されました。データソースは、便宜性と移植性のために、Java Naming and Directory Interface(JNDI)の実体にバインドできるため、データベースには論理名でアクセスできます。

データソース機能は、以前のJDBC `DriverManager`機能の完全な代替機能を提供します。いずれの機能も同じアプリケーションで使用できますが、使用しているアプリケーションはデータソースに移行することをお薦めします。

この項の内容は次のとおりです。

- JNDIのOracleデータソース・サポートの概要
- データソースの機能とプロパティ
- データソース・インスタンスの作成と接続
- データソース・インスタンスの作成、JNDIへの登録および接続
- サポートされている接続プロパティ
- SYSログオンのためのロールの使用方法について
- データベース・リモート・ログインの構成
- Bequeath接続とSYSログオンの使用
- Oracleパフォーマンス拡張機能のプロパティの設定
- ネットワーク・データ圧縮のサポート

8.1.1 JNDIのOracleデータソース・サポートの概要

JNDI標準は、リモート・サービスおよびリソースを検出してアクセスする方法をアプリケーションに提供します。これらのサービスは、エンタープライズ・サービスにも可能です。ただし、JDBCアプリケーションの場合は、これらのサービスにデータベース接続およびサービスが含まれます。

JNDIを使用すると、アプリケーションはアプリケーション・コードからベンダー固有の構文を削除し、論理名を使用してこれらのサービスにアクセスできます。JNDIには、目的のサービスの特定のソースと論理名を関連付ける機能があります。

すべてのOracle JDBCデータソースは、JNDIによる参照が可能です。開発者が必ずしもこの機能を使用する必要はありませ

んが、JNDI論理名を使用してデータベースにアクセスすると、コードの移植性が高まります。



ノート:

JNDI 機能の使用には、CLASSPATH 環境変数に jndi.jar ファイルが含まれている必要があります。このファイルは、Java 製品とともにインストール・メディアに含まれています。CLASSPATH 環境変数に、別個に追加する必要があります。

8.1.2 データソースの機能とプロパティ

JNDIを使用するデータソース機能では、ベンダー固有のJDBCドライバ・クラス名を登録する必要はなく、URLとその他のプロパティの論理名を使用できます。このため、データベース接続をオープンするためのコードは、他の環境に移植できます。

DataSourceインタフェースとOracle実装

JDBCデータソースは、標準javax.sql.DataSourceインタフェースを実装するクラスのインスタンスです。

```
public interface DataSource
{
   Connection getConnection() throws SQLException;
   Connection getConnection(String username, String password)
      throws SQLException;
   ...
}
```

Oracleでは、このインタフェースをoracle.jdbc.poolパッケージのOracleDataSourceクラスに実装しています。オーバーロードされたgetConnectionメソッドは、データベースへの接続を戻します。

他の値を使用するには、適切なsetterメソッドを使用してプロパティを設定します。代替のユーザー名とパスワードを設定するには、入力としてこれらのパラメータを取るgetConnectionメソッドを使用することもできます。この設定は、プロパティ設定よりも優先されます。



ノート:

OracleDataSource クラスとすべてのサブクラスは、java.io.Serializable インタフェースとjavax.naming.Referenceable インタフェースを実装します。

DataSourceのプロパティ

DataSourceインタフェースを実装するすべてのクラスと同様に、OracleDataSourceクラスは、接続するデータベースの指定に使用できる一連のプロパティを提供します。これらのプロパティはJavaBeansのデザインパターンに従います。

次の表に、OracleDataSource標準プロパティおよびOracle拡張機能を示します。



ノート:

Oracle は標準 roleName プロパティを実装していません。

表8-1 標準データソース・プロパティ

名前	型	説明
databaseName	String	サーバー上の特定のデータベースの名前。
dataSourceName	String	基礎となるデータソース・クラスの名前。接続プーリングの場合、これはプーリングされた基礎となる接続データソース・クラスです。分散トランザクションの場合、これは基礎となる XA データソース・クラスです。
description	String	データソースの説明。
networkProtocol	String	サーバーとの通信のためのネットワーク・プロトコル。Oracle では、JDBC Oracle Call Interface(OCI)ドライバにのみ該当し、デフォルトは tcp です。
password	String	接続するユーザーのパスワード。
portNumber	int	サーバーがリクエストをリスニングするポートの番号。
serverName	String	データベース・サーバーの名前。
user	String	ログイン用の名前。



ノート:

セキュリティ上の理由により、getPassword() メソッドはありません。

表8-2 Oracle拡張データソース・プロパティ

名前	型	説明
connectionCacheName	String	キャッシュの名前を指定します。キャッシュを作成した後は変更できません。
connectionCacheProperties	java.util.Properties	暗黙的接続キャッシュのプロパティを指定します。
connectionCachingEnabled	Boolean	暗黙的接続キャッシュが使用中かどうかを

名前	型	説明
		指定します。
connectionProperties	java.util.Properties	接続プロパティを指定します。
driverType	String	Oracle JDBC ドライバのタイプを指定します。oci、thin または kprb のいずれかを指定します。
fastConnectionFailoverEnabled	Boolean	高速接続フェイルオーバーが使用中かどうかを指定します。
implicitCachingEnabled	Boolean	暗黙的文接続キャッシュが有効かどうかを指定します。
loginTimeout	int	このデータソースがデータベースへの接続の試行中に待機する最大時間(秒)を指定します。
logWriter	java.io.PrintWriter	このデータソースのログ・ライターを指定します。
maxStatements	int	アプリケーションのキャッシュ内に存在する文の最大数を指定します。
serviceName	String	このデータソースに対するデータベース・サービス名を指定します。
tnsEntry	String	TNS エントリ名を指定します。TNS エントリ名は、tnsnames.ora 構成ファイルに指定されている TNS エントリに対応します。 OCI ドライバでネイティブ XA 機能を使用する場合は、この OracleXADataSource プロパティを有効に設定して、Oracle リリース 8.1.6 以降のデータベースにアクセスしてください。ネイティブ XA 機能を使用するときに tnsEntry プロパティが設定されていないと、エラー・コード ORA-17207 の SQLException が発生しま

名前	型	説明
		す。
url	String	データベース接続文字列の URL を指定します。以前のバージョンの Oracle Database から移行する際の便宜を考えて提供されるものです。Oracle tnsEntry および driverType プロパティ、標準 portNumber、networkProtocol、serverName および databaseName プロパティのかわりに、このプロパティを使用できます。
nativeXA	Boolean	OCI ドライバでネイティブ XA 機能を使用する場合は、OracleXADataSource を有効に設定して、Oracle リリース 8.1.6 以降のデータベースにアクセスしてください。nativeXA プロパティを有効にする場合は、必ず tnsEntry プロパティも設定してください。このプロパティは OracleXADatasource 専用です。 この DataSource プロパティは、デフォルトで false に設定されます。
ONSConfiguration	String	FAN/ONS イベントのリモートでのサブスクライブに使用する ONS 構成を指定します。



ノート:

- この表では、非推奨となった接続キャッシュを OracleConnectionCache に基づいてサポートしていたプロパティは省略されています。
- ネイティブ XA は、Java XA よりパフォーマンスが高いため、可能なかぎりネイティブ XA を使用してください。

setConnectionPropertiesメソッドを使用して接続のプロパティを設定し、setConnectionCachePropertiesメソッドを使用して接続キャッシュのプロパティを設定します。

サーバー側内部ドライバを使用している場合、つまりdriverTypeプロパティがkprbに設定されている場合、その他のプロパティ設定は無視されます。

JDBC ThinドライバまたはOCIドライバを使用している場合は、次の点に注意してください。

- URL設定には、次の例のようにuserとpasswordの設定が含まれることがあります。この場合は、この設定が個々のuserおよびpasswordプロパティ設定よりも優先されます。

```
jdbc:oracle:thin:HR/hr@localhost:5221:orcl
```

- userとpasswordは、URL設定を介して直接設定するか、またはgetConnectionコールを介して設定する必要があります。getConnectionコールによるuserとpasswordの設定は、他のプロパティ設定よりも優先されます。
- urlプロパティを設定した場合は、tnsEntry、driverType、portNumber、networkProtocol、serverNameおよびdatabaseNameプロパティ設定は無視されます。
- tnsEntryプロパティを設定した場合(urlプロパティは設定されていないものとします)、databaseName、serverName、portNumberおよびnetworkProtocolの各設定は無視されます。
- OCIドライバを使用しているときに(driverTypeプロパティはociに設定されているものとします)、networkProtocolがipcに設定されている場合、他のプロパティ設定は無視されます。

またgetConnectionCacheName()は、データソースのキャッシュが有効になった後でデータソースのConnectionCacheNameプロパティが設定された場合にのみキャッシュの名前を戻します。

8.1.3 データソース・インスタンスの作成と接続

この項では、JNDI機能を使用せずにデータベースに接続する場合の、データソースの最も基本的な使用例を示します。ベンダーによってはベンダー固有のハードコードされたプロパティ設定が必要であることに注意してください。

次の例のようにOracleDataSourceインスタンスを作成し、必要に応じて接続プロパティを初期化して、接続インスタンスを取得します。

```
OracleDataSource ods = new OracleDataSource();
ods.setDriverType("oci");
ods.setServerName("localhost");
ods.setNetworkProtocol("tcp");
ods.setDatabaseName(<database_name>);
ods.setPortNumber(5221);
ods.setUser("HR");
ods.setPassword("hr");
Connection conn = ods.getConnection();
```

または、必要に応じて次のようにユーザー名とパスワードをオーバーライドします。

```
Connection conn = ods.getConnection("OE", "oe");
```

8.1.4 データソース・インスタンスの作成、JNDIへの登録および接続

この項では、データソースを使用してデータベースに接続する場合のJNDI機能を示します。ベンダー固有のハードコードされたプロパティ設定は、データソース・インスタンスをJNDI論理名にバインドするコードの部分でのみ必要になります。これ以降は、接続インスタンスを取得するデータソースを作成するために、論理名を使用して、移植可能なコードを作成できます。



ノート:

データソースの作成と登録は、一般に JDBC アプリケーションではなく、JNDI 管理者によって処理されます。

データソース・プロパティの初期化

次の例に示すように、OracleDataSourceインスタンスを作成し、必要に応じてそのプロパティを初期化します。

```
OracleDataSource ods = new OracleDataSource();
ods.setDriverType("oci");
ods.setServerName("localhost");
ods.setNetworkProtocol("tcp");
ods.setDatabaseName("816");
ods.setPortNumber(5221);
ods.setUser("HR");
ods.setPassword("hr");
```

データソースの登録

前述の例で示したように、OracleDataSourceインスタンスodsの接続プロパティを初期化した後は、次の例に示すように、このデータソース・インスタンスをJNDIに登録できます。

```
Context ctx = new InitialContext();
ctx.bind("jdbc/sampledb", ods);
```

JNDI InitialContext()コンストラクタをコールすると、初期JNDIネーミング・コンテキストを参照するJavaオブジェクトが作成されます。表示されていないシステム・プロパティによって、使用するサービス・プロバイダがJNDIに指示されます。

ctx.bindコールは、OracleDataSourceインスタンスを論理JNDI名にバインドします。つまり、ctx.bindをコールした後、いつでも論理名jdbc/sampledbを使用して、OracleDataSourceインスタンスodsのプロパティによって記述されるデータベースへの接続をオープンできます。論理名jdbc/sampledbは、このデータベースに論理的にバインドされます。

JNDIネームスペースの階層構造は、ファイル・システムの階層構造と似ています。この例で、JNDI名はルート・ネーミング・コンテキストでサブコンテキストjdbcを指定し、jdbcサブコンテキスト内で論理名sampledbを指定します。

ContextインスタンスとInitialContextクラスは、標準javax.namingパッケージ内にあります。



ノート:

JDBC 2.0 仕様では、すべての JDBC データソースを、JNDI ネームスペースの jdbc ネーミング・サブコンテキスト、または jdbc サブコンテキストの子サブコンテキストに登録する必要があります。

接続のオープン

lookupを実行して、JNDI名に論理的にバインドされたデータベースへの接続をオープンするには、論理JNDI名を使用します。これを実行するには、lookupの結果(Java Objectも可)をOracleDataSourceにキャストし、そのgetConnectionメソッドを使用して接続をオープンします。

次はその例です。

```
OracleDataSource odsconn = (OracleDataSource)ctx.lookup("jdbc/sampledb");
Connection conn = odsconn.getConnection();
```

8.1.5 サポートされている接続プロパティ

Oracle JDBCドライバがサポートする接続プロパティの詳細なリストは、『Oracle Database JDBC Java APIリファレンス』を参照してください。

ノート:

Oracle Databaseで以前の方法でDead Connection Detection (DCD)が有効になっている場合、JDBC Thinドライバ接続プロパティoracle.jdbc.ReadTimeoutおよびJDBCメソッドjava.sql.Connection.setNetworkTimeoutが予想どおりに動作しない場合があります。DCDの新しい実装の詳細は、次のMy Oracle Supportのノートを参照してください:
https://support.oracle.com/rs?type=doc&id=1591874.1

8.1.6 SYSログオンのためのロールの使用方法について

SYSでログオンするためのロールを指定するには、internal_logon接続プロパティを使用します。SYSでログオンするには、internal_logon接続プロパティを、SYSDBAまたはSYSOPERに設定します。



ノート:

ロールを指定する機能は、sys ユーザー名の場合のみサポートされています。

Bequeathed接続の場合、internal_logonプロパティを設定することにより、SYSとして接続を取得できます。リモート接続の場合、追加でパスワード・ファイルの設定手順を実行する必要があります。

8.1.7 データベース・リモート・ログインの構成

JDBC ThinドライバがSYSDBAとしてデータベースに接続できるようにするには、ユーザーを構成する必要があります。Oracle Databaseのセキュリティ・システムで管理者としてリモート接続するためにはパスワード・ファイルが必要です。次の手順を実行します。

1. orapwdパスワード・ユーティリティを使用してサーバー側またはリモート・データベースでパスワード・ファイルを設定します。ユーザーSYSのパスワード・ファイルを追加するには、次のようにします。
   - UNIXの場合

     ```
     orapwd file=$ORACLE_HOME/dbs/orapwORACLE_SID entries=200
     Enter password: password
     ```

   - Microsoft Windowsの場合

     ```
     orapwd file=%ORACLE_HOME%¥database¥PWDORACLE_SID.ora entries=200
     Enter password: password
     ```

   この場合、fileは、パスワード・ファイルの名前です。passwordは、ユーザーSYSのためのパスワードです。SQL Plusで

は、ALTER USER文を使用して変更することができます。entriesは、想定されるエントリ数より大きな値に設定してください。

パスワード・ファイル名の構文は、Microsoft WindowsとUNIXでは異なります。

関連項目:

Oracle Database管理者ガイド

2. SYSDBAとしてリモート・ログインを有効にします。このステップでは、SYSDBAおよびSYSOPERシステム権限を個別のユーザーに付与し、それらのユーザーがそのままのユーザーとして接続できるようにします。

   データベースを停止し、次の行を、UNIXの場合はinitservice_name.oraに、Microsoft Windowsの場合はinit.oraに追加します。

   ```
   remote_login_passwordfile=exclusive
   ```

   initservice_name.oraファイルは、ORACLE_HOME/dbs/およびORACLE_HOME/admin/db_name/pfile/に存在します。これらの2つのファイルは同期化してください。

   init.oraファイルは、%ORACLE_BASE%¥ADMIN¥db_name¥pfile¥に存在します。

3. SYSユーザーのパスワードを変更します。これはオプションのステップです。

   ```
   PASSWORD sys
          Changing password for sys
   New password: password
   Retype new password: password
   ```

4. SYSがSYSDBA権限を持っていることを確認します。

   ```
   SQL> select * from v$pwfile_users;
   USERNAME                         SYSDB      SYSOP
   ----------------------           ---------  ---------
   SYS                              TRUE       TRUE
   ```

5. リモート・データベースを再起動します。

例8-1 リモート接続でのSYSログインの使用

```
//This example works regardless of language settings of the database.
 /** case of remote connection using sys **/
import java.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;
// create an OracleDataSource
OracleDataSource ods = new OracleDataSource();
// set connection properties
java.util.Properties prop = new java.util.Properties();
prop.put("user", "sys");
prop.put("password", "sys");
prop.put("internal_logon", "sysoper");
ods.setConnectionProperties(prop);
// set the url
// the url can use oci driver as well as:
// url = "jdbc:oracle:oci8:@remotehost"; the remotehost is a remote database
String url = "jdbc:oracle:thin:@//localhost:5221/orcl";
```

```
ods.setURL(url);
// get the connection
Connection conn = ods.getConnection();
...
```

8.1.8 Bequeath接続とSYSログオンの使用

次の例は、SYSログインを指定するための、internal_logonおよびSYSDBA引数の使用方法を示しています。この例は、データベースの各国語設定に関係なく動作します。

```
/** Example of bequeath connection **/
import java.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;

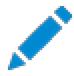
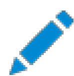
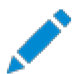
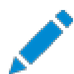
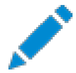
// create an OracleDataSource instance
OracleDataSource ods = new OracleDataSource();

// set neccessary properties
java.util.Properties prop = new java.util.Properties();
prop.put("user", "sys");
prop.put("password", "sys");
prop.put("internal_logon", "sysdba");
ods.setConnectionProperties(prop);

// the url for bequeath connection
String url = "jdbc:oracle:oci8:@";
ods.setURL(url);

// retrieve the connection
Connection conn = ods.getConnection();
...
```

8.1.9 Oracleパフォーマンス拡張機能のプロパティの設定

接続プロパティの一部は、Oracleパフォーマンス拡張機能で使用します。これらのプロパティの設定は、次のようにOracleConnectionオブジェクトで対応するメソッドを使用することと同等です。

- defaultRowPrefetchプロパティの設定は、setDefaultRowPrefetchのコールと同等です。
- remarksReportingプロパティの設定は、setRemarksReportingのコールと同等です。

関連項目:

「DatabaseMetaData TABLE_REMARKSのレポートについて」

例

次の例は、java.util.Propertiesクラスのputメソッドの使用方法を示します。この例では、Oracleパフォーマンス拡張要素のパラメータを設定する方法を示します。

```
//import packages and register the driver
import java.sql.*;
import java.math.*;
```

```
import oracle.jdbc.*;
import oracle.jdbc.pool.OracleDataSource;
//specify the properties object
java.util.Properties info = new java.util.Properties();
info.put ("user", "HR");
info.put ("password", "hr");
info.put ("defaultRowPrefetch","20");
info.put ("defaultBatchValue", "5");
//specify the datasource object
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:thin:@//localhost:5221/orcl");
ods.setUser("HR");
ods.setPassword("hr");
ods.setConnectionProperties(info);
...
```

8.1.10 ネットワーク・データ圧縮のサポート

Oracle Database 12cリリース2 (12.2.0.1)以降、JDBC Thinドライバではネットワーク・データ圧縮がサポートされています。ネットワーク・データ圧縮により、データ接続を介して送信されるセッション・データ・ユニット(SDU)のサイズが削減され、ネットワークを介してSQL問合せおよび結果を送信するのに必要な時間が短縮します。ワイヤレス・エリア・ネットワーク(WAN)の場合に、この利点がより重要になります。ネットワーク・データ圧縮を有効にするには、次のように接続プロパティを設定する必要があります。



ノート:

ネットワーク圧縮は、ストリーム処理されたデータでは機能しません。

```
...
OracleDataSource ds = new OracleDataSource();
Properties prop = new Properties();
prop.setProperty("user","user1");
prop.setProperty("password",<password>);
// Enabling Network Compression
prop.setProperty("oracle.net.networkCompression","on");
//Optional configuration for setting the client compression threshold.
prop.setProperty("oracle.net.networkCompressionThreshold","1024");
ds.setConnectionProperties(prop);
ds.setURL(url);
Connection conn = ds.getConnection();
...
```

8.2 データベースURLとデータベース指定子

データベースURLは文字列です。完全なURL構文は、次のとおりです。

```
jdbc:oracle:driver_type:[username/password]@database_specifier
```



ノート:

- 大カッコは、username/password ペアがオプションであることを示します。
- 内部サーバー側ドライバ kprb は、暗黙的な接続を使用します。サーバー側ドライバのデータベース URL は driver_type で終わります。

URLの最初の部分には、使用するJDBCドライバを指定します。サポートされているdriver_type値は、thin、ociおよびkprbです。

URLの残りの部分には、スラッシュで区切られたユーザー名とパスワード(オプション)、@、およびアプリケーションが接続されるデータベースを一意に識別するデータベース指定子が含まれます。データベース指定子には、JDBC Thinドライバにのみ有効なもの、JDBC OCIドライバにのみ有効なもの、両方に有効なものがあります。

8.2.1 インターネット・プロトコル・バージョン6のサポート

Oracle JDBCドライバの今回のリリースでは、インターネット・プロトコル・バージョン6(IPv6)のアドレスを、JDBC URLとIPv6のアドレスに解決するマシン名でサポートします。IPv6は、Internet Engineering Task Force(IETF)で設計された新しいネットワーク層プロトコルで、Internet Protocolの現行バージョンのInternet Protocol Version 4(IPv4)に代わるものです。IPv6の主な利点として、128ビット・アドレスの使用から導出される大容量のアドレス空間があります。IPv6ではまた、ルーティング、ネットワークの自動構成、セキュリティ、サービスの質などの分野でIPv4が改良されています。

ノート:

- IPv6 クライアントは IPv6 サーバーのみ、またはデュアル・プロトコル(IPv6 と IPv4 の両方のプロトコル)をサポートしているサーバーをサポートできます。また、IPv6 サーバーは IPv6 クライアントのみ、またはデュアル・プロトコル・クライアントをサポートできます。
- IPv6 は単一インスタンス・データベース・サーバーでのみサポートされています。Oracle RAC ではサポートされていません。

URLでリテラルIPv6アドレスを使用する場合は、リテラル・アドレスを左の大カッコ([)と右の大カッコ(])で囲む必要があります。たとえば、[2001:0db8:7654:3210:FEDC:BA98:7654:3210]のようになります。したがって、IPv6アドレスを使用するJDBC URLは、次のようになります。

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
  (HOST=[2001:0db8:7654:3210:FEDC:BA98:7654:3210])(PORT=5521))
  (CONNECT_DATA=(SERVICE_NAME=sales.example.com))
```



ノート:

データベースを初期化する際に Java が有効な場合、IPv6 のサポートに必要な新しいシステム・クラスがすべてロードされます。したがって、アプリケーションに IPv6 アドレスがない場合は、IPv6 機能を使用するために自分のコードを

変更する必要はありません。ただし、アプリケーションに IPv6 アドレスのみか、IPv6 と IPv4 アドレスの両方のいずれかがある場合、コマンドラインで java.net.preferIPv6Addresses システム・プロパティを設定する必要があります。これにより、Oracle JVM は適切なライブラリをロードできるようになります。これらのライブラリは一度ロードすると、Java プロセスを再起動しないかぎりリロードできません。

8.2.2 HTTPSプロキシ構成のサポート

Oracle Databaseリリース18cのJDBCドライバは、HTTPSプロキシ構成をサポートしています。HTTPSプロキシを使用すると、HTTP CONNECTメソッドを使用して、フォワードHTTPプロキシ上にセキュアな接続をトンネリングできます。これにより、クライアント側のファイアウォールにアウトバウンド・ポートをオープンする必要がなくなるため、パブリック・クラウド・データベースにアクセスしやすくなります。このパラメータはPROTOCOL=TCPSが指定されている接続記述子に対してのみ適用可能です。これは、インターネットのホストに接続する必要があるイントラネット・ユーザーのWebブラウザ設定に似ています。

HTTPSプロキシを構成するために、次のコード・スニペットに示すように、接続文字列のADDRESS部分に詳細を追加します。

```
(DESCRIPTION=
  (ADDRESS=(HTTPS_PROXY=sales-proxy)(HTTPS_PROXY_PORT=8080)(PROTOCOL=TCPS)(HOST=sales2-svr)(PORT=443))
  (CONNECT_DATA=(SERVICE_NAME=sales.us.example.com)))
```

8.2.3 データベース指定子

表8-3は、データベース指定子と各指定子がサポートするJDBCドライバを示しています。

ノート:

- Oracle Database 10g では、Oracle Service ID はサポートされません。
- Oracle Database 10g 以降、ネーミング・メソッドとしての Oracle Names はサポートされなくなりました。

表8-3 サポートされるデータベース指定子

指定子	サポートするドライバ	例

Oracle Net 接続記述子	Thin、OCI	Thin の場合。アドレス・リストを使用します。
		
		`url="jdbc:oracle:thin:@(DESCRIPTION=`
		`(LOAD_BALANCE=on) (ADDRESS_LIST=`
		`(ADDRESS=(PROTOCOL=TCP)(HOST=host1) (PORT=5221))`
		`(ADDRESS=(PROTOCOL=TCP)(HOST=host2)(PORT=5221)))`
		`(CONNECT_DATA=(SERVICE_NAME=orcl)))"`
		
		OCI の場合。クラスタを使用します。
		
		`"jdbc:oracle:oci:@(DESCRIPTION=`
		`(ADDRESS=(PROTOCOL=TCP)(HOST=cluster_alias) (PORT=5221))`

指定子	サポートするドライバ	例
		(CONNECT_DATA=(SERVICE_NAME=orcl)))"
Thin 形式のサービス名	Thin	詳細は、「Thin 形式のサービス名の構文」を参照してください。 "jdbc:oracle:thin:HR/hr@//localhost:5221/orcl"
LDAP 構文	Thin	詳細は、LDAP 構文を参照してください。 "jdbc:oracle:thin:@ldap://ldap.example.com:7777/sales,cn=OracleContext,dc=com"
Bequeath 接続	OCI	空。つまり、@の後には何も記述しません。 "jdbc:oracle:oci:HR/hr/@"
TNSNames 別名	Thin、OCI	詳細は、「TNSNames 別名の構文」を参照してください。 OracleDataSource ods = new OracleDataSource(); ods.setTNSEntryName("MyTNSAlias");

8.2.4 Thin形式のサービス名の構文

Thin形式のサービス名は、JDBC Thinドライバでのみサポートされます。構文は次のとおりです。

```
@//host_name:port_number/service_name
```

たとえば:

```
jdbc:oracle:thin:HR/hr@//localhost:5221/orcl
```



ノート:

JDBC Thin ドライバは、TCP/IP プロトコルのみをサポートします。

8.2.5 Easy Connect Plusのサポート

Oracle Databaseリリース19cのJDBCドライバでは、EasyConnect URLに多くの機能を備えた拡張機能が導入され、この新しい拡張された簡易接続URLは、Easy Connect Plusと呼ばれています。

EasyConnect URLでは、簡略化された接続文字列およびTCP転送プロトコルがサポートされていました。EasyConnect URLへの拡張の目的は、データベースへの接続時にクライアント側のデプロイメントと構成を簡素化することです。この拡張機能により、tnsnames.oraファイルを使用する必要がなくなり、TNS_ADMIN環境変数を指定する必要がなくなります。Easy

Connect Plusの主な機能は、次のとおりです。

- TCPSプロトコルのサポート
- 複数のホストとポートのサポート
- 接続文字列での接続プロパティの受渡しのサポート

8.2.5.1 TCPSプロトコルのサポート

EasyConnect URLはTCPトランスポート・プロトコルのみをサポートしていました。ただし、Easy Connect PlusはTCPプロトコルとTCPSプロトコルの両方をサポートしています。この拡張機能により、クライアント構成が、TCPS接続が必須であるOracle Database Cloud Serviceに簡略化されます。

たとえば、EasyConnect URLが次の形式であるとします。

```
jdbc:oracle:thin:@(DESCRIPTION=
(ADDRESS=(PROTOCOL=tcps)(HOST=salesserver1)(PORT=1521))
(CONNECT_DATA=(SERVICE_NAME=sales.us.example.com)))
```

この場合、Easy Connect PlusのURLは次のようになります。

```
jdbc:oracle:thin:@tcps:salesserver1:1521/sales.us.example.com
```

8.2.5.2 複数のホストとポートのサポート

Easy Connect Plusを使用すると、ホストのカンマ区切りリストを指定できます。

EasyConnect URLでは、単一のホスト名および単一のポートのみが許可されていました。しかし、Easy Connect Plusを使用すると、接続文字列で複数のホストおよび複数のポートを指定でき、ロード・バランシングが有効になっているデータベースに接続する場合にこの接続文字列を使用できます。

たとえば、次のEasyConnect URLを使用して、接続に単一のポート番号を持つ2つのホストを指定するとします。

```
jdbc:oracle:thin:@(DESCRIPTION=
(ADDRESS_LIST= (LOAD_BALANCE=ON) (ADDRESS=(PROTOCOL=tcp)(HOST=salesserver1)(PORT=1521))
(ADDRESS=(PROTOCOL=tcp)(HOST=salesserver2)(PORT=1521)))
(CONNECT_DATA=(SERVICE_NAME=sales.us.example.com)))
```

この場合、Easy Connect PlusのURLは次のようになります。

```
jdbc:oracle:thin:@tcp:salesserver1,salesserver2:1521/sales.us.example.com
```

また、次のEasyConnect URLを使用して、各リストが独自のポート番号に従うリストとして複数のホストを指定するとします。

```
jdbc:oracle:thin:@(DESCRIPTION= (ADDRESS_LIST= (LOAD_BALANCE=ON)
(ADDRESS=(PROTOCOL=tcp)(HOST=salesserver1)(PORT=1521))
(ADDRESS=(PROTOCOL=tcp)(HOST=salesserver2)(PORT=1522))
(ADDRESS=(PROTOCOL=tcp)(HOST=salesserver3)(PORT=1522)))
(CONNECT_DATA=(SERVICE_NAME=sales.us.example.com)))
```

この場合、Easy Connect PlusのURLは次のようになります。

```
jdbc:oracle:thin:@tcp:salesserver1:1521,salesserver2,salesserver3:1522/sales.us.example.com
```

8.2.5.3 接続文字列での接続プロパティの受渡しのサポート

Easy Connect Plusでは、接続URLでの名前/値ペアとしての接続プロパティの指定をサポートしています。?デリミタの後、Easy Connect Plusでは、名前/値ペアとしてのパラメータのリストがサポートされています。これは必要に応じて指定できます。

名前/値ペアを指定する場合は、次の点に注意してください。

- 疑問符記号(?)を使用して名前/値ペアの始まりを示し、各名前/値ペアの間にアンパサンド記号(&)をデリミタとして使用します。
- 接続文字列全体を単一の文字列として指定します。
- 値に特殊文字が含まれている場合は、バックスラッシュ(¥)エスケープ文字を使用します。
- 値の一部として空白が必要な場合は、パラメータ値内の先頭と末尾の空白は無視されるため、空白を二重引用符内に挿入します。

次の表に、サポートされているパラメータの一部を示します。

パラメータ名	古いURLの例	新しいURLの例
wallet_location	jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcps)(HOST=salesserver1)(PORT=1521))(CONNECT_DATA=(SERVICE_NAME=sales.us.example.com))	jdbc:oracle:thin:@tcps:salesserver1.com:1521/sales.us.example.com?wallet_location=/Users/jsmith/DBCloudService/wallet_JDBCTEST&oracle.net.ssl_server_cert_dn=¥"CN=salesserver2.com.com,OU=Oracle BMCS US,O=Oracle Corporation,L=Redwood City,ST=California,C=US¥"";
ssl_server_dn_match		
ssl_server_cert_dn	jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcps)(HOST=salesserver1)(PORT=1521)) (SECURITY=(SSL_SERVER_DN_MATCH=TRUE)(SSL_SERVER_CERT_DN=cn=sales,cn=OracleContext,dc=us,dc=example,dc=com))(CONNECT_DATA=(SERVICE_NAME=sales.us.example.com)))	jdbc:oracle:thin:@tcps:salesserver1:1521/sales.us.example.com?ssl_server_cert_dn="cn=sales,cn=OracleContext,dc=us,dc=example,dc=com"
https_proxy および https_proxy_port	jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcps)(HOST=salesserver1)(PORT=1521) (https_proxy=www-proxy.mycompany.com)(https_proxy_port=80))(CONNECT_DATA=(SERVICE_NAME=sales.us.example.com)))	jdbc:oracle:thin:@tcps:salesserver1:1521/sales.us.example.com?https_proxy=www-proxy.mycompany.com&https_proxy_port=80
connect_timeout	jdbc:oracle:thin:@(DESCRIPTION=(retry_count=3)(connect_timeout=60)(transport_connect_timeout=30)(ADDRESS=(PROTOCOL=tcp)(HOST=salesserver1)(PORT=1521))(CONNECT_DATA=(SERVICE_NAME=sales.us.example.com)))	jdbc:oracle:thin:@tcps:salesserver1:1521/sales.us.example.com?connect_timeout=60&transport_connect_timeout=30&retry_count=3

DESCRIPTIONの下でサポートされているキーワードのリストは次のとおりです。

- ENABLE
- FAILOVER
- LOAD_BALANCE
- RECV_BUF_SIZE
- SEND_BUF_SIZE
- SDU
- SOURCE_ROUTE
- RETRY_COUNT
- RETRY_DELAY
- CONNECT_TIMEOUT
- TRANSPORT_CONNECT_TIMEOUT



ノート:

アプリケーションで SOCKS プロキシを使用して Oracle Database に接続する場合(Oracle Cloud の要塞など)、TRANSPORT_CONNECT_TIMEOUT パラメータの値は無視されます。

8.2.6 接続の再試行での遅延のサポート

Oracle Database 12cリリース1 (12.1.0.2)以降、秒単位の接続の再試行の遅延を指定する新しい接続属性RETRY_DELAYがあります。次のコード・スニペットは、この属性の使用方法を示しています。

```
(DESCRIPTION_LIST=
  (DESCRIPTION=
   (CONNECT_TIMEOUT=10)(RETRY_COUNT=3)(RETRY_DELAY=3)
   (ADDRESS_LIST=
    (ADDRESS=(PROTOCOL=tcp)(HOST=myhost1)(PORT=1521))
    (ADDRESS=(PROTOCOL=tcp)(HOST=myhost2)(PORT=1521)))
   (CONNECT_DATA=(SERVICE_NAME=example1.com)))
  (DESCRIPTION=
   (CONNECT_TIMEOUT=60)(RETRY_COUNT=1)(RETRY_DELAY=5)
   (ADDRESS_LIST=
    (ADDRESS=(PROTOCOL=tcp)(HOST=myhost3)(PORT=1521))
    (ADDRESS=(PROTOCOL=tcp)(HOST=myhost4)(PORT=1521)))
   (CONNECT_DATA=(SERVICE_NAME=example2.com))))
```

8.2.7 TNSNames別名の構文

接続元のクライアント・コンピュータ上のtnsnames.oraファイルにリストされた、利用可能なTNSNAMESエントリを検索できます。Windowsの場合、このファイルはORACLE_HOME¥NETWORK¥ADMINディレクトリにあります。UNIXシステムの場合は、ORACLE_HOMEディレクトリ、またはTNS_ADMIN環境変数に示されているディレクトリに格納されています。

たとえば、ホストmyhost上のデータベースに、ユーザーHR、パスワードhr(MyHostStringのTNSNAMESエントリを持つ)で接続する場合は、次のように記述します。

```
OracleDataSource ods = new OracleDataSource();
ods.setTNSEntryName("MyTNSAlias");
ods.setUser("HR");
ods.setPassword("hr");
ods.setDriverType("oci");
Connection conn = ods.getConnection();
```

JDBC Thinドライバがtnsnames.oraファイルの位置を確認できるように、oracle.net.tns_adminシステム・プロパティをtnsnames.oraファイルの位置に設定する必要があります。たとえば:

```
System.setProperty("oracle.net.tns_admin", "c:¥¥Temp");
String url = "jdbc:oracle:thin:@tns_entry";
```



ノート:

TNSNames を JDBC Thin ドライバとともに使用する場合は、oracle.net.tns_admin プロパティを tnsnames.ora ファイルのあるディレクトリに設定する必要があります。

```
java -Doracle.net.tns_admin=$ORACLE_HOME/network/admin
```

8.2.8 LDAP構文

Lightweight Directory Access Protocol(LDAP)構文を使用するデータベース指定子の例を次に示します。

```
"jdbc:oracle:thin:@ldap:ldap.example.com:7777/sales,cn=OracleContext,dc=com"
```

TLSを使用する場合は、これを次のように変更します。

```
"jdbc:oracle:thin:@ldaps:ldap.example.com:7777/sales,cn=OracleContext,dc=com"
```



ノート:

JDBC Thin ドライバの場合、データベース指定子の ldap:を ldaps:に置き換えると、LDAP を使用し、TLS を介して Oracle Internet Directory と通信できます。TLS を使用するように LDAP サーバーを構成する必要があります。そうしない場合は、接続試行がハングします。

JDBC Thinドライバは、サービス名解決プロセスの際、LDAPサーバーのリストのフェイルオーバーをサポートします。ハードウェア・ロード・バランサは必要ありません。また、クライアント側ロード・バランシングが、LDAPサーバーへの接続のためにサポートされます。フェイルオーバーとロード・バランシングをサポートするために、スペースで区切られたLDAP URL構文のリストが使用されます。

LDAP URLのリストを指定すると、フェイルオーバーとロード・バランシングの両方がデフォルトで有効になります。oracle.net.ldap_loadbalance接続プロパティを使用するとロード・バランシングを無効にでき、oracle.net.ldap_failover接続プロパティを使用するとフェイルオーバーを無効にできます。

次の例は、クライアント側ロード・バランシングが無効になっているフェイルオーバーを示しています。

```
Properties prop = new Properties();
String url =
"jdbc:oracle:thin:@ldap:ldap1.example.com:3500/cn=salesdept,cn=OracleContext,dc=com/salesdb" +
"ldap:ldap2.example.com:3500/cn=salesdept,cn=OracleContext,dc=com/salesdb " +
"ldap:ldap3.example.com:3500/cn=salesdept,cn=OracleContext,dc=com/salesdb";
prop.put("oracle.net.ldap_loadbalance", "OFF" );
OracleDataSource ods = new OracleDataSource();
ods.setURL(url);
ods.setConnectionProperties(prop);
```

JDBC ThinドライバはLDAP非匿名バインドをサポートします。データソースに対して、認証情報を含む一連のJNDI環境プロパティを指定できます。LDAPサーバーが匿名バインドを許可しないよう設定されている場合は、LDAPサーバーに接続するために認証情報を提供する必要があります。次の例は、簡単なクリアテキストのパスワード認証を示しています。

```
String url =
"jdbc:oracle:thin:@ldap:ldap.example.com:7777/sales,cn=salesdept,cn=OracleContext,dc=com";
Properties prop = new Properties();
prop.put("oracle.net.ldap.security.authentication", "simple");
prop.put("oracle.net.ldap.security.principal","cn=salesdept,cn=OracleContext,dc=com");
prop.put("oracle.net.ldap.security.credentials", "mysecret");
OracleDataSource ods = new OracleDataSource();
ods.setURL(url);
ods.setConnectionProperties(prop);
```

JDBCがJNDIに3つのプロパティを渡すため、クライアントで選択される認証メカニズムが、これらのプロパティのJNDIでの解釈方法と矛盾することはありません。たとえば、クライアントがjava.naming.security.authenticationプロパティを明示的に指定せずに認証情報を指定する場合、デフォルトの認証メカニズムはsimpleです。

9 JDBCクライアント側セキュリティ機能

この章では、JDBC OCIおよびJDBC ThinドライバのOracle Advanced Securityのオプション機能に関して、Autonomous Databaseに対するIAM認証、ログイン認証、ネットワーク暗号化および整合性のサポートについて説明します。

ノート:

- サーバー側内部ドライバを介したすべての通信が完全にサーバー内で行われるため、この説明はサーバー側内部ドライバには関係しません。
- パラメータ SQLNET.CRYPTO_CHECKSUM_TYPES_CLIENT および SQLNET.CRYPTO_CHECKSUM_TYPES_SERVER を指定した SHA-1 (Secure Hash Algorithm 1)の使用は、このリリースでは非推奨であり、将来のリリースではサポートされなくなる可能性があります。DBMS_CRYPTO での SHA-1 暗号の使用も非推奨になりました(HASH_SH1、HMAC_SH1)。SHA1 を使用するのではなく、SHA-1 暗号にかわるより強力な SHA-2 暗号の使用を開始することをお薦めします。

以前Advanced Networking Option (ANO)またはAdvanced Security Option (ASO)と呼ばれていたOracle Advanced Securityは、業界標準に基づくネットワーク暗号化、ネットワーク整合性、サード・パーティ認証、シングル・サインオンおよびアクセス認可の機能を提供します。JDBC OCIドライバとJDBC Thinドライバは、どちらもOracle Advanced Security機能のすべてをサポートしています。

ノート:

JDBC ojdbc.policy のセキュリティ・ポリシー・ファイルを使用する場合、次のリンクからファイルをダウンロードできます。

http://www.oracle.com/technetwork/index.html

ojdbc.policy ファイルには、Java セキュリティ・マネージャの制御環境でアプリケーションを実行する必要がある付与された権限が含まれます。このファイル自体を Java ポリシー・ファイルとして使用するか、このファイルからコンテンツを取得して Java ポリシー・ファイルのコンテンツを追加できます。このファイルには、次のような権限が含まれます。

- 権限 java.util.PropertyPermission "user.name", "read";などの常に必要ないくつかの必須権限
- いくつかのドライバ固有の権限(たとえば、JDBC OCI ドライバでは権限 java.lang.RuntimePermission "loadLibrary.ocijdbc12";が必要です)
- XA、XDB、FCF などに関連する権限などのいくつかの機能ベースの権限

要件に応じてファイルに記述されているシステム・プロパティまたは権限のダイレクト値を設定できます。

この章の構成は、次のとおりです。

- IAMのトークンベース認証のサポート
- Azure ADのトークンベース認証のサポート
- Oracle Advanced Securityのサポート
- ログイン認証のサポート
- 厳密認証のサポート
- ネットワーク暗号化と整合性のサポート
- TLSのサポート
- Kerberosのサポート
- RADIUSのサポート
- セキュアな外部パスワード・ストアについて

9.1 IAMのトークンベース認証のサポート

Oracle Databaseリリース19.14以降では、JDBC Thinドライバは、Oracle Cloud Infrastructure (OCI) Identity Access Management (IAM)の拡張サポートを提供します。

データベースへの接続中、JDBCアプリケーションはデータベースにトークンを提供します。データベースは、認証サービスからリクエストする公開キーでトークンを検証し、対応するユーザー・グループ・メンバーシップ情報を取得し、データベース・スキーマとロール・マッピングを検索してデータベースへのユーザー認証を完了します。

さらに、アプリケーションは署名済ヘッダーを送信することで、トークンに埋め込まれた公開キーとペアになる秘密キーを保有していることを証明します。トークンと署名の両方が有効で、IAMユーザーとデータベース・ユーザーの間にマッピングが存在している場合は、JDBCアプリケーションにデータベースへのアクセスが許可されます。

トークンベースの認証は、次の方法でサポートされます。

9.1.1 ファイル・システムの使用

データベース・トークンがファイル・システムで使用できるようになっていると、たとえば、Oracle Cloud Infrastructureコマンドライン・インタフェース(OCI CLI)を使用している場合は、このトークンを使用してデータベースに接続するようにJDBCドライバを構成できます。

この目的には、CONNECTION_PROPERTY_TOKEN_AUTHENTICATIONを使用できます。これは、次の方法で指定できます。

- ojdbc.propertiesファイルとして
- JVMシステム・プロパティとして
- 接続文字列の問合せセクションのパラメータとして
- OracleDataSource.setConnectionProperties(Properties)に渡されたPropertiesオブジェクトによって
- Oracle Net記述子のSECURITYセクションのパラメータとして

JDBCは、このパラメータを指定できる次の2つのタイプの接続文字列をサポートしています。

- Oracle Net記述子として、このパラメータを次のように指定できます。

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(HOST=dbhost)(PORT=1522)(PROTOCOL=tcps))(SECURITY=(SSL
_SERVER_DN_MATCH=ON)
(TOKEN_AUTH=OCI_TOKEN))(CONNECT_DATA=(SERVICE_NAME=my.example.com)))
```

- 接続プロパティとして、このパラメータを次のように指定できます。

```
jdbc:oracle:thin:@tcps:dbhost:1522/my.example.com?oracle.jdbc.tokenAuthentication=OCI_TOKEN&or
acle.jdbc.tokenLocation
=" /path/to/my/token"
```

CONNECTION_PROPERTY_TOKEN_AUTHENTICATIONがOCI_TOKENに設定されている場合、CONNECTION_PROPERTY_TOKEN_LOCATIONは、ドライバによるアクセス・トークンの取得元になるファイル・システム・パスを指定します。デフォルトの場所は、$HOME/.oci/db-token/です。このプロパティは、デフォルト以外の場所を指定するために別の値を設定できます。このプロパティで指定するパスは、tokenおよびoci_db_key.pemという名前のファイルが含まれているディレクトリにする必要があります。



ノート:

- Oracle Net 記述子スタイルの URL に TOKEN_LOCATION パラメータが含まれている場合、そのパラメータの値は、CONNECTION_PROPERTY_TOKEN_LOCATION で定義した値より優先されます。
- トークン・ファイルには、JSON Web トークン(JWT)が UTF-8 エンコード・テキストの単一行で含まれている必要があります。JWT フォーマットは、RFC 7519 によって規定されています。
- トークンの場所には、oci_db_key.pem という名前の秘密キー・ファイルも含まれている必要があります。秘密キー・ファイルには PEM 形式を使用して、base64 エンコーディングの RSA 秘密キーが PKCS#8 エンコーディングで含まれている必要があります

接続プロパティCONNECTION_PROPERTY_TOKEN_AUTHENTICATION (oracle.jdbc.tokenAuthentication)をOAUTHに設定し、接続プロパティCONNECTION_PROPERTY_TOKEN_LOCATION (oracle.jdbc.tokenLocation)をファイル・システム上のベアラー・トークンを指すように設定することもできます。この場合、デフォルトの場所が設定されていないため、場所を次のいずれかに設定する必要があります。

- ディレクトリ(この場合、ドライバはtokenという名前のファイルをロードします)
- 完全修飾ファイル名

これは、次の方法で実行できます。

- ojdbc.propertiesファイルの構成

```
# Enable the OAUTH authentication mode
oracle.jdbc.tokenAuthentication=OAUTH
# Specify the location of the Bearer token location
oracle.jdbc.tokenLocation=/home/user1/mytokens/jwtbearertoken
```

- JDBC URLの使用

```
jdbc:oracle:thin:@tcps:adb.mydomain.oraclecloud.com:1522/xyz.adb.oraclecloud.com?
oracle.jdbc.tokenAuthentication=OAUTH&oracle.jdbc.tokenLocation=/home/user/token
```

- JDBC URLに対するTNS形式の使用:

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCPS)(PORT=1521)(HOST=adb.mydomain.oraclecloud.com))(CONNECT_DATA=
(SERVICE_NAME=xyz.adb.oraclecloud.com))(SECURITY=(TOKEN_AUTH=OAUTH)(TOKEN_LOCATION=/home/user1/mytokens/jwtbearertoken)))"
```

9.1.2 oracle.jdbc.accessToken接続プロパティの使用

CONNECTION_PROPERTY_ACCESS_TOKEN (oracle.jdbc.accessToken)をアクセス・トークン値に設定します。

これは、次の方法で実行できます。



ノート:

トークン値に含まれる可能性がある等号(=)をエスケープするには、トークン値を二重引用符("")で囲む必要があります。

- JDBC URLの使用

```
jdbc:oracle:thin:@tcps:adb.mydomain.oraclecloud.com:1522/xyz.adb.oraclecloud.com?oracle.jdbc.accessToken="ey...5c"
```

- 記述子URLの使用

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcps)(HOST=mydomain.com)(PORT=5525))
      (CONNECT_DATA=(SERVICE_NAME=myservice.com)))?
        oracle.jdbc.accessToken="ey...5c"
```

- ojdbc.propertiesファイルの構成

```
# Enable the OAUTH authentication mode
oracle.jdbc.accessToken="ey...5c"
```

または

```
# Enable the OAUTH authentication mode
oracle.jdbc.accessToken=${DATABASE_ACCESS_TOKEN}
```

ここで、アクセス・トークンはDATABASE_ACCESS_TOKEN環境変数の値です。



ノート:

アクセス・トークンが提供されている場合、ドライバによって OAUTH モードが自動的に設定されるため、oracle.jdbc.tokenAuthentication=OAUTH を設定する必要はありません。

9.1.3 OracleConnectionBuilderインタフェースの使用

データベース・アクセス・トークンによる認証のために、OracleConnectionBuilder.accessTokenメソッドをコールします。このメソッドは、アプリケーションが認証サービスから取得するトークン値を受け入れます。

このメソッドを使用してトークンを渡すと、TOKEN_AUTH=OCI_TOKEN接続文字列の設定がオーバーライドされます。つまり、JDBCは通常どおりファイル・システムからトークンを読み取らなくなるということです。かわりに、JDBCはaccessTokenメソッドに指定されたAccessTokenオブジェクトを使用するようになります。

この場合、JDBCは秘密キーを使用して署名を生成し、IAMデータベース・アクセス・トークンとともにデータベースに送信します。最初に、データベースはIAMの公開キーでトークンを検証します。その次に、JDBCによって生成された署名を、トークンに埋め込まれている公開キーで復号化して検証します。復号化された署名が有効な場合は、JDBCが秘密キーを保有していることの証明になります。

単一のURLで構成された単一インスタンスのOracleDataSourceクラスにより、OracleConnectionBuilderインタフェースのインスタンスが作成されます。こうしたインスタンスは、O5Logonによる従来の認証をサポートしながら、トークンベースの認証もサポートします。その次に、アプリケーションは、ユーザーとパスワードを構成するメソッドまたはトークンを構成するメソッドをコールします。ただし、このビルダーはトークンとユーザー名またはパスワードの両方を使用して構成することはできません。accessTokenメソッドとpasswordメソッドまたはuserメソッドの両方がNULL以外の値で起動されたると、このビルダーとの接続の作成時に無効な構成を示すSQLExceptionがスローされます。

9.1.4 OracleDataSourceクラスの使用

データベース・アクセス・トークンによる認証のために、OracleCommonDataSource.setTokenSupplier(AccessToken accessToken)メソッドをコールします。

このメソッドでは、このDataSourceとの接続を作成するときに、アクセス・トークンを生成するサプライヤ・ファンクションを設定します。サプライヤ・ファンクションは、このDataSourceによる接続の作成のたびに起動されます。サプライヤによって生成されたアクセス・トークンのインスタンスは、クライアント認証のためにOracle Databaseでサポートされているトークン・タイプを表している必要があります。サプライヤは、スレッド・セーフになっている必要があります。



ノート:

AccessToken.createJsonWebTokenCache(Supplier)メソッドは、ユーザー定義の Supplier からのトークンをキャッシュする、スレッド・セーフの Supplier を作成するために使用します。

このDataSourceはトークン・サプライヤとユーザー名またはパスワードの両方を使用して構成することはできません。setUser(String)、setPassword(String)、 setConnectionProperties(java.util.Properties)または setConnectionProperty(String, String)メソッドを起動して、このDataSourceをユーザー名またはパスワードで構成し、setTokenSuppliersetTokenSupplier(AccessToken accessToken)メソッドを起動してトークン・サプライヤを構成すると、このDataSourceとの接続を作成するときに無効な構成を示すSQLExceptionがスローされます。

アクセス・トークンは本質的に一時的なものであり、1時間以内に期限切れになります。そのため、OracleDataSourceクラスのインスタンスが接続を作成するたびに、新しく生成されたトークンを取得できるようにするSupplierタイプを使用してください。Supplierは、トークンの有効期限が切れるまで、同じトークンを複数回生成できます。トークンの有効期限が切れると、Supplierは、そのトークンを生成しなくなり、それよりも有効期限が後になる新しいトークンを生成するようになります。

関連項目:

- Oracle Autonomous DatabasesのIAMユーザーの認証と認可
- TLSプロトコル付きTCP/IPについて

9.2 Azure ADのトークンベース認証のサポート

Oracle Databaseリリース19.14以降では、JDBC Thinドライバは、Azure Active Directory (Azure AD) OAuth2アクセス・トークンのサポートを提供します。

データベースへの接続中、JDBCアプリケーションはデータベースにトークンを提供します。データベースは、認証サービスからリクエストする公開キーでトークンを検証し、対応するユーザー・グループ・メンバーシップ情報を取得し、データベース・スキーマとロール・マッピングを検索してデータベースへのユーザー認証を完了します。

トークンベースの認証は、次の方法でサポートされます。

9.2.1 ファイル・システムの使用

ファイル・システムでデータベース・トークンが使用可能な場合は、データベースへの接続にこのトークンを使用するようにJDBCドライバを構成できます。

この目的には、CONNECTION_PROPERTY_TOKEN_AUTHENTICATIONを使用できます。これは、次の方法で指定できます。

- ojdbc.propertiesファイルとして
- JVMシステム・プロパティとして
- 接続文字列の問合せセクションのパラメータとして
- OracleDataSource.setConnectionProperties(Properties)に渡されたPropertiesオブジェクトによって
- Oracle Net記述子のSECURITYセクションのパラメータとして

JDBCは、このパラメータを指定できる次の2つのタイプの接続文字列をサポートしています。

- Oracle Net記述子として、このパラメータを次のように指定できます。

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(HOST=dbhost)(PORT=1522)(PROTOCOL=tcps))(SECURITY=(SSL
_SERVER_DN_MATCH=ON)
(TOKEN_AUTH=OAUTH)(TOKEN_LOCATION=/path/to/my/token)))
```

- 接続プロパティとして、このパラメータを次のように指定できます。

```
jdbc:oracle:thin:@tcps://dbhost:1522/my.example.com?oracle.jdbc.tokenAuthentication=OAUTH&orac
le.jdbc.tokenLocation
=”/path/to/my/token”
```

CONNECTION_PROPERTY_TOKEN_AUTHENTICATIONがOAUTHに設定されている場合、CONNECTION_PROPERTY_TOKEN_LOCATIONは、アクセス・トークンの取得元になるファイル・システム・パスを指定します。この場合、デフォルトの場所はありません。トークンの場所を設定する必要があります。これは、tokenという名前のファイルにあるトークンを含むディレクトリです。たとえば、ディレクトリmytokendirectoryにtokenという名前のファイルが含まれている場合、次のようにトークンの場所を設定します。

```
/path/to/mytokendirectory
```



ノート:

- Oracle Net 記述子スタイルの URL に TOKEN_LOCATION パラメータが含まれている場合、そのパラメータの値は、CONNECTION_PROPERTY_TOKEN_LOCATION で定義した値より優先されます。

- トークン・ファイルには、JSON Web トークン(JWT)が UTF-8 エンコード・テキストの単一行で含まれている必要があります。JWT フォーマットは、RFC 7519 によって規定されています。

接続プロパティCONNECTION_PROPERTY_TOKEN_AUTHENTICATION (oracle.jdbc.tokenAuthentication)をOAUTHに設定し、接続プロパティCONNECTION_PROPERTY_TOKEN_LOCATION (oracle.jdbc.tokenLocation)をファイル・システム上のベアラー・トークンを指すように設定することもできます。この場合、デフォルトの場所が設定されていないため、場所を次のいずれかに設定する必要があります。

- ディレクトリ(この場合、ドライバはtokenという名前のファイルをロードします)
- 完全修飾ファイル名

これは、次の方法で実行できます。

- ojdbc.propertiesファイルの構成

```
# Enable the OAUTH authentication mode
oracle.jdbc.tokenAuthentication=OAUTH
# Specify the location of the Bearer token location
oracle.jdbc.tokenLocation=/home/user1/mytokens/jwtbearertoken
```

- JDBC URLの使用

```
jdbc:oracle:thin:@tcps://adb.mydomain.oraclecloud.com:1522/xyz.adb.oraclecloud.com?
oracle.jdbc.tokenAuthentication=OAUTH&oracle.jdbc.tokenLocation=/home/user/token
```

- JDBC URLに対するTNS形式の使用:

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCPS)(PORT=1521)(HOST=adb.mydomain.oraclecloud.com))(CONNECT_DATA=
(SERVICE_NAME=xyz.adb.oraclecloud.com))(SECURITY=(TOKEN_AUTH=OAUTH)(TOKEN_LOCATION=/home/user1
/mytokens/jwtbearertoken)))"
```

9.2.2 oracle.jdbc.accessToken接続プロパティの使用

CONNECTION_PROPERTY_ACCESS_TOKEN (oracle.jdbc.accessToken)をアクセス・トークン値に設定します。

これは、次の方法で実行できます。



ノート:

トークン値に含まれる可能性がある等号(=)をエスケープするには、トークン値を二重引用符("")で囲む必要があります。

- JDBC URLの使用

```
jdbc:oracle:thin:@tcps://adb.mydomain.oraclecloud.com:1522/xyz.adb.oraclecloud.com?oracle.jdbc
.accessToken="ey...5c"
```

- 記述子URLの使用

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcps)(HOST=mydomain.com)(PORT=5525))
      (CONNECT_DATA=(SERVICE_NAME=myservice.com)))?
        oracle.jdbc.accessToken="ey...5c"
```

- ojdbc.propertiesファイルの構成

```
# Enable the OAUTH authentication mode
oracle.jdbc.accessToken="ey...5c"
```

または

```
# Enable the OAUTH authentication mode
oracle.jdbc.accessToken=${DATABASE_ACCESS_TOKEN}
```

ここで、アクセス・トークンはDATABASE_ACCESS_TOKEN環境変数の値です。



ノート:

アクセス・トークンが提供されている場合、ドライバによって OAUTH モードが自動的に設定されるため、oracle.jdbc.tokenAuthentication=OAUTH を設定する必要はありません。

9.2.3 OracleConnectionBuilderインタフェースの使用

データベース・アクセス・トークンによる認証のために、OracleConnectionBuilder.accessTokenメソッドをコールします。

このメソッドは、アプリケーションが認証サービスから取得するトークン値を受け入れます。このメソッドを使用してトークンを渡すと、TOKEN_AUTH=OCI_TOKEN接続文字列の設定、その後TOKEN_AUTH=OAUTHがオーバーライドされます。つまり、JDBCは通常どおりファイル・システムからトークンを読み取らなくなるということです。かわりに、JDBCはaccessTokenメソッドに指定されたAccessTokenオブジェクトを使用するようになります。

単一のURLで構成された単一インスタンスのOracleDataSourceクラスにより、OracleConnectionBuilderインタフェースのインスタンスが作成されます。こうしたインスタンスは、O5Logonによる従来の認証をサポートしながら、トークンベースの認証もサポートします。その次に、アプリケーションは、ユーザーとパスワードを構成するメソッドまたはトークンを構成するメソッドをコールします。ただし、このビルダーはトークンとユーザー名またはパスワードの両方を使用して構成することはできません。accessTokenメソッドとpasswordメソッドまたはuserメソッドの両方がNULL以外の値で起動されたると、このビルダーとの接続の作成時に無効な構成を示すSQLExceptionがスローされます。

9.2.4 OracleDataSourceクラスの使用

データベース・アクセス・トークンによる認証のために、OracleCommonDataSource.setTokenSupplier(AccessToken accessToken)メソッドをコールします。

このメソッドでは、このDataSourceとの接続を作成するときに、アクセス・トークンを生成するサプライヤ・ファンクションを設定します。サプライヤ・ファンクションは、このDataSourceによる接続の作成のたびに起動されます。サプライヤによって生成されたアクセス・トークンのインスタンスは、クライアント認証のためにOracle Databaseでサポートされているトークン・タイプを表している必要があります。サプライヤは、スレッド・セーフになっている必要があります。



ノート:

AccessToken.createJsonWebTokenCache(Supplier)メソッドは、ユーザー定義の Supplier からのトークンを

キャッシュする、スレッド・セーフの Supplier を作成するために使用します。

このDataSourceはトークン・サプライヤとユーザー名またはパスワードの両方を使用して構成することはできません。setUser(String)、setPassword(String)、 setConnectionProperties(java.util.Properties)または setConnectionProperty(String, String)メソッドを起動して、このDataSourceをユーザー名またはパスワードで構成し、setTokenSuppliersetTokenSupplier(AccessToken accessToken)メソッドを起動してトークン・サプライヤを構成すると、このDataSourceとの接続を作成するときに無効な構成を示すSQLExceptionがスローされます。

アクセス・トークンは本質的に一時的なものであり、1時間以内に期限切れになります。そのため、OracleDataSourceクラスのインスタンスが接続を作成するたびに、新しく生成されたトークンを取得できるようにするSupplierタイプを使用してください。Supplierは、トークンの有効期限が切れるまで、同じトークンを複数回生成できます。トークンの有効期限が切れると、Supplierは、そのトークンを生成しなくなり、それよりも有効期限が後になる新しいトークンを生成するようになります。

関連項目:

- Autonomous DatabaseでのAzure Active Directory (Azure AD)の使用
- TLSプロトコル付きTCP/IPについて

9.3 Oracle Advanced Securityのサポート

この項では、次の概念について説明します。

- Oracle Advanced Securityの概要
- JDBC OCIドライバによるOracle Advanced Securityのサポート
- JDBC ThinドライバによるOracle Advanced Securityのサポート

9.3.1 Oracle Advanced Securityの概要

Oracle Advanced Securityでは、次のセキュリティ機能をサポートしています。

- ネットワーク暗号化

  エンタープライズ・ネットワークおよびインターネット上でやりとりされる機密情報は、復号化キーによってのみ解読できる形式に情報を変換する暗号化アルゴリズムにより保護できます。たとえば、AESです。

  送信中にネットワークの整合性を保証するために、Oracle Advanced Securityでは、暗号化によってセキュアなメッセージ・ダイジェストを生成します。Oracle Database 12cリリース1 (12.1)以降、ハッシュ・アルゴリズムのSHA-2リストもサポートされているので、Oracle Advanced Securityでは次のハッシュ・アルゴリズムを使用して、セキュアなメッセージ・ダイジェストを生成し、ネットワーク上で送信される各メッセージにこれを含めます。

  これにより、やりとりされるデータは、データの変更、パケットの削除、反射攻撃などの攻撃から保護されます。

  次のコードの抜粋に、前述のアルゴリズムのいずれかを使用したチェックサムの計算方法を示します。

```
prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_TYPES, "( SHA1)");
prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_LEVEL, "REQUIRED");
```

- 厳密認証

分散環境でネットワーク・セキュリティを確保するには、ユーザーを認証し、資格証明を確認する必要があります。パスワード認証は、認証で最も一般的な手段です。Oracle Databaseでは、Oracle認証アダプタを使用して厳密な認証を行います。Oracle認証アダプタは、デジタル証明書を使用したTLSを含む様々なサード・パーティ認証サービスをサポートします。Oracle Databaseは、業界で標準的な次の認証方式をサポートしています。

- Kerberos
- Remote Authentication Dial-In User Service(RADIUS)
- Transport Layer Security(TLS)

関連項目:

『Oracle Databaseセキュリティ・ガイド』

9.3.2 JDBC OCIドライバによるOracle Advanced Securityのサポート

JDBC OCIドライバを使用している場合は、Oracleクライアントがインストールされているコンピュータから実行していることが想定されるため、Oracle Advanced Securityと組込みサード・パーティ機能のサポートは、Oracleクライアントによって提供されるサポートとほとんど同じです。Advanced Security機能が使用されているかどうかは、クライアント・コンピュータのsqlnet.oraファイル内の、関連する設定の内容によって判断できます。



ノート:

Oracle Database 12c リリース 1 (12.1)以降、sqlnet.ora ファイルにある OCI 固有の構成パラメータのかわりに、新しい XML 構成ファイル oraaccess.xml にある構成パラメータを使用することをお薦めします。ただし、sqlnet.ora ファイル内の構成パラメータもまだサポートされています。

パスワードを指定せずにデータベースに接続しようとすると、JDBC OCIドライバは、外部認証を使用しようとします。JDBC OCIドライバを使用して、パスワードを入力しないでデータベースに接続する例を次に示します。

TLS認証

次のコードでは、TLS認証を使用してデータベースに接続する方法を示します:

例9-1 TLS認証を使用したデータベースへの接続

```
import java.sql.*;
import java.util.Properties;

public class test
{
    public static void main( String [] args ) throws Exception
    {
        String url = "jdbc:oracle:oci:@"
         +"(DESCRIPTION=(ADDRESS=(PROTOCOL=tcps)(HOST=localhost)(PORT=5221))"
         +"(CONNECT_DATA=(SERVICE_NAME=orcl)))";
        Driver driver = new oracle.jdbc.OracleDriver();
        Properties props = new Properties();
        Connection conn = driver.connect( url, props );
```

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

データソースの使用

次のコードでは、データソースを使用してデータベースに接続する方法を示します。

例9-2 データソースを使用したデータベースへの接続

```
import java.sql.*;
import javax.sql.*;
import java.util.Properties;
import oracle.jdbc.pool.*;

public class testpool {
    public static void main( String args ) throws Exception
    { String url = "jdbc:oracle:oci:@"
+"(DESCRIPTION=(ADDRESS=(PROTOCOL=tcps)(HOST=localhost)(PORT=5221))"
 +"(CONNECT_DATA=(SERVICE_NAME=orcl)))";
    OracleConnectionPoolDataSource ocpds = new OracleConnectionPoolDataSource();
    ocpds.setURL(url);
    PooledConnection pc = ocpds.getPooledConnection();
    Connection conn = pc.getConnection();
    }
    }
```



ノート:

Java に関する前述の重要な例外は、アプリケーションでネイティブ・スレッドを使用している場合にのみ、Transport Layer Security (TLS)プロトコルが Oracle JDBC OCI ドライバでサポートされているという点です。通常、グリーン・スレッドがデフォルトで設定されているため、特に注意が必要です。

9.3.3 JDBC ThinドライバによるOracle Advanced Securityのサポート

JDBC Thinドライバでは、Oracleクライアントがインストールされていること、またはsqlnet.oraファイルが存在していることを前もって想定できません。したがって、Javaアプローチを使用してOracle Advanced Securityをサポートします。Oracle Advanced Securityを実装するJavaクラスは、ojdbc8.jarファイルおよびojdbc10.jarファイルに含まれています。暗号化と整合性のためのセキュリティ・パラメータは、通常のようにsqlnet.oraファイル内に設定されるのではなく、JavaのPropertiesオブジェクトか、またはシステム・プロパティを使用して設定されます。

9.4 ログイン認証のサポート

JDBCを介した基本ログイン認証では、Oracleサーバーへの他のログイン方法と同様に、ユーザー名とパスワードが要求されます。Javaプロパティ・オブジェクトを使用するか、またはgetConnectionメソッドを直接コールして、ユーザー名とパスワードを指定します。これは、使用しているクライアント側Oracle JDBCドライバにかかわらず適用されますが、サーバー側内部ドライバを使用している場合には、特殊な直接接続を使用しており、ユーザー名またはパスワードを要求しないため、関係ありません。

Oracle Database 12cリリース1 (12.1.0.2)以降、Oracle JDBC Thinドライバは、PBKDF2-SHA2アルゴリズムをサポートしているJDK 8などのJDKを使用してアプリケーションを実行する場合、O7L_MRクライアント機能をサポートしています。JDK 7を使用してアプリケーションを実行する場合は、PBKDF2-SHA2アルゴリズムをサポートしているサードパーティ製セキュリティ・プロバイダを追加する必要があります。そうしないと、O7L_MRクライアント機能を必要とする新規の12aパスワード検証機能が、ドライバでサポートされません。

SQLNET.ALLOWED_LOGON_VERSION_SERVERパラメータを12aに設定してOracle Database 12cリリース1 (12.1.0.2)を使用している場合は、次の点に注意してください。

- 12.1.0.2 Oracle JDBC Thinドライバと、JDK 8、またはPBKDF2-SHA2アルゴリズムをサポートしているサードパーティ製セキュリティ・プロバイダを追加したJDK 7も、使用する必要があります
- 旧バージョンのOracle JDBC Thinドライバを使用すると、次のエラーが発生します。

```
ORA-28040: No matching authentication protocol
```

- 12.1.0.2 Oracle JDBC ThinドライバをJDK 7とともに使用する場合も、PBKDF2-SHA2アルゴリズムをサポートしているサードパーティ製セキュリティ・プロバイダを追加していないと、同じエラーが発生します。

9.5 厳密認証のサポート

Oracle Advanced Securityでは、Oracle Databaseユーザーに対する外部認証が可能です。外部認証には、RADIUS、Kerberos、証明書ベースの認証、トークン・カードおよびスマート・カードなどが使用されます。この認証方法は、厳密認証と呼ばれます。Oracle JDBCドライバでサポートしている厳密認証方法は次のとおりです。

- Kerberos
- RADIUS
- TLS

関連項目:

『Oracle Database Net Servicesリファレンス・ガイド』

9.6 ネットワーク暗号化と整合性のサポート

> ノート:
>
> Oracle では、古い暗号化およびハッシュ・アルゴリズムが非推奨になりました。非推奨のアルゴリズムには、DBMS_CRYPTO およびネイティブ・ネットワーク暗号化の場合は MD4、MD5、DES、3DES および RC4 関連アルゴリズム、透過的データ暗号化(TDE)の場合は 3DES があります。安全性の低い古い暗号化アルゴリズムを削除すると、これらのアルゴリズムが誤って使用されることがなくなります。セキュリティ要件を満たすために、Advanced Encryption Standard (AES)などの最新の暗号化アルゴリズムを使用することをお薦めします。

関連項目:

詳細は、『Oracle Databaseセキュリティ・ガイド』を参照してください

この項では、次の概念について説明します。

- JDBCによるネットワーク暗号化と整合性のサポートの概要
- JDBC OCIドライバによるデータ暗号化と整合性のサポート
- JDBC Thinドライバによるデータ暗号化と整合性のサポート
- Javaでの暗号化および整合性パラメータの設定

9.6.1 JDBCによるネットワーク暗号化と整合性のサポートの概要

サーバーでの関連部分の設定によっては、Javaデータベース・アプリケーションで、Oracle DatabaseとOracle Advanced Securityのネットワーク暗号化および整合化の機能を使用できます。JDBC OCIドライバを使用している場合は、Oracleクライアントの場合と同様にパラメータを設定します。Thinドライバを使用している場合は、Javaプロパティ・オブジェクトを使用してパラメータを設定します。

暗号化は、クライアント側暗号化レベル設定と、サーバー側暗号化レベル設定の組合せに基づいて、有効または無効になります。同様に、整合性はクライアント側整合性レベル設定と、サーバー側整合性レベル設定の組合せに基づいて、有効または無効になります。

暗号化と整合性は、REJECTED、ACCEPTED、REQUESTEDおよびREQUIREDの同じ設定レベルをサポートしています。表9-1は、クライアント側とサーバー側の設定の組合せごとに、この機能がオンになるかオフになるかを示しています。リモートOS認証(TCP経由)はセキュリティ上問題があるため、デフォルトで無効にされています。

表9-1 暗号化または整合性のためのクライアント/サーバー間ネゴシエーション

クライアント/サーバー設定のマトリックス	クライアントRejected	クライアントAccepted (デフォルト)	クライアントRequested	クライアントRequired
サーバーRejected	OFF	OFF	OFF	接続に失敗
サーバーAccepted (デフォルト)	OFF	OFF	ON	ON
サーバーRequested	OFF	ON	ON	ON
サーバーRequired	接続に失敗	ON	ON	ON

表9-1が示すように、たとえば、クライアントが暗号化を要求し、サーバーが拒否した場合は、暗号化はオフになります。整合性の場合も同様です。逆に、サーバーが暗号化を要求し、クライアントが受け入れる場合は、暗号化はオンになります。整合性の場合も同様です。

関連項目:

ネットワークの暗号化および整合性の機能の詳細は、『Oracle Databaseセキュリティ・ガイド』を参照してください。



ノート:

整合性パラメータ名には、まだ checksum という用語が使用されていますが、この用語はパラメータ以外では使用されません。事実上、checksum と integrity はシノニムです。

9.6.2 JDBC OCIドライバによるデータ暗号化と整合性のサポート

JDBC OCIドライバを使用している場合は、Oracleクライアントのインストールを伴うOracleクライアント設定が想定されるため、クライアントでのsqlnet.oraファイルの設定によって、すべてのOracleクライアントの場合と同様に、ネットワーク暗号化または整合性を使用可能または使用不可にし、関連するパラメータを設定できます。



ノート:

Oracle Database 12c リリース 1 (12.1)以降、sqlnet.ora ファイルにある OCI 固有の構成パラメータのかわりに、新しい XML 構成ファイル oraaccess.xml にある構成パラメータを使用することをお薦めします。ただし、sqlnet.ora ファイル内の構成パラメータもまだサポートされています。

表9-2に、クライアント・パラメータについてまとめます。

表9-2 暗号化と整合性のためのOCIドライバ・クライアント・パラメータ

パラメータの説明	パラメータ名	可能な設定
クライアント暗号化レベル	SQLNET.ENCRYPTION_CLIENT	REJECTED ACCEPTED REQUESTED REQUIRED
クライアント暗号化選択リスト	SQLNET.ENCRYPTION_TYPES_CLIENT	AES128、AES192、AES256
クライアント整合性レベル	SQLNET.CRYPTO_CHECKSUM_CLIENT	REJECTED ACCEPTED REQUESTED REQUIRED
クライアント整合性選択リスト	SQLNET.CRYPTO_CHECKSUM_TYPES_CLIENT	SHA-1

9.6.3 JDBC Thinドライバによるデータ暗号化と整合性のサポート

JDBC Thinドライバによるネットワーク暗号化および整合性のパラメータ設定のサポートは、前の項で説明したJDBC OCIドライバ・サポートと似ています。Javaプロパティ・オブジェクトを介して対応するパラメータを設定し、データベース接続のオープン中にこれを使用できます。

データ暗号化と整合性レベルのデフォルト値は、サーバー側もクライアント側もACCEPTEDです。これにより、サーバー側とクライアント側のいずれかの接続を構成して、接続ペアに必要なセキュリティ・レベルを獲得できます。これにより、Oracleサーバーに接続する複数のOracleクライアントがある場合、すべての接続の暗号化と整合性を有効にするために、サーバー側の

sqlnet.oraファイルで暗号化と整合性レベルをREQUESTEDに変更するだけでよいので、プログラムの効率性が向上します。クライアンの設定を個別に変更する必要がないので、時間と労力を節約できます。

表9-3は、JDBC Thinドライバのパラメータ情報です。これらのパラメータは、oracle.jdbc.OracleConnectionインタフェースで定義されています。

表9-3 暗号化と整合性のためのThinドライバ・クライアント・パラメータ

パラメータ名	パラメータ・タイプ	可能な設定
CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_LEVEL	String	REJECTED ACCEPTED REQUESTED REQUIRED
CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_TYPES	String	AES256、AES192、AES128
CONNECTION_PROPERTY_THIN_NET_CHECKSUM_LEVEL	String	REJECTED ACCEPTED REQUESTED REQUIRED
CONNECTION_PROPERTY_THIN_NET_CHECKSUM_TYPES	String	SHA1

ノート:



- Oracle Advanced Security の Thin ドライバのサポートは、JDBC クラスの JAR ファイルに直接組み込まれています。そのため、国内版と輸出版で別々のバージョンはありません。輸出版には適するパラメータ設定のみを使用できます。

9.6.4 Javaでの暗号化および整合性パラメータの設定

Javaプロパティ・オブジェクト(つまり、java.util.Propertiesのインスタンス)を使用して、JDBC Thinドライバでサポートされている、ネットワーク暗号化および整合性のパラメータを設定します。

次の例では、Javaプロパティ・オブジェクトをインスタンス化し、それを使用して表9-3の各パラメータを設定し、その後そのプロパティ・オブジェクトを使用してデータベース接続をオープンしています。

```
...
Properties prop = new Properties();
prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_LEVEL, "REQUIRED");
prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_TYPES, "( AES256 )");
prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_LEVEL, "REQUESTED");
prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_TYPES, "( SHA1 )");
OracleDataSource ods = new OracleDataSource();
ods.setConnectionProperties(prop);
ods.setURL("jdbc:oracle:thin:@localhost:5221:main");
Connection conn = ods.getConnection();
...
```

暗号化タイプおよびチェックサム・タイプの値は、丸カッコで囲み、リスト形式で指定できます。複数の値を指定した場合は、サー

バーとクライアントの間のネゴシエーションにより、実際に使用される値が決定されます。

例

例9-3は、データベースに接続して問合せを実行する前に、ネットワーク暗号化と整合性のパラメータを設定する完成したクラスの例です。



ノート:

この例にある文字列 REQUIRED は、AnoServices クラスと Service クラスの機能によって動的に取り出されます。この方法で文字列を取り出すことも、前の例のようにソフトウェア・コードに含めることもできます。

この例を実行する前に、sqlnet.oraファイルで暗号化を有効にする必要があります。たとえば、次のコードは、暗号化についてAES256、AES192およびAES128を、チェックサムについてはSHA1をオンにする場合の例です。

```
  SQLNET.ENCRYPTION_SERVER = ACCEPTED
  SQLNET.CRYPTO_CHECKSUM_SERVER = ACCEPTED
  SQLNET.CRYPTO_CHECKSUM_TYPES_SERVER = (SHA1)
  SQLNET.ENCRYPTION_TYPES_SERVER = (AES256, AES192, AES128)
```

例9-3 ネットワーク暗号化および整合性のパラメータの設定

```
import java.sql.*;
import java.util.Properties;
import oracle.net.ano.AnoServices;
import oracle.jdbc.*;

public class DemoAESAndSHA1
{
  static final String USERNAME= "HR";
  static final String PASSWORD= "hr";
  static final String URL =
"jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=localhost)(PORT=5221))"
  +"(CONNECT_DATA=(SERVICE_NAME=orcl)))";

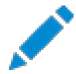
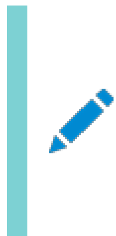
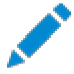
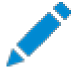
  public static final void main(String[] argv)
  {
    DemoAESAndSHA1 demo = new DemoAESAndSHA1();
    try
    {
      demo.run();
    }catch(SQLException ex)
    {
      ex.printStackTrace();
    }
  }

  void run() throws SQLException
  {
    OracleDriver dr = new OracleDriver();
    Properties prop = new Properties();

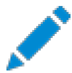
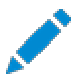
    // We require the connection to be encrypted with either AES256 or AES192.
    // If the database doesn't accept such a security level, then the connection attempt will fail.
```

```
prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_LEVEL,AnoServices.ANO_REQUIR
ED);
    prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_TYPES,"( " +
AnoServices.ENCRYPTION_AES256
     + "," + AnoServices.ENCRYPTION_AES192 + ")");

    // We also require the use of the SHA1 algorithm for network integrity checking.


prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_LEVEL,AnoServices.ANO_REQUIRED
);
    prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_TYPES,
"( " + AnoServices.CHECKSUM_SHA1 + " )");
    prop.setProperty("user",DemoAESAndSHA1.USERNAME);
    prop.setProperty("password",DemoAESAndSHA1.PASSWORD);
    OracleConnection oraConn = (OracleConnection)dr.connect(DemoAESAndSHA1.URL,prop);
    System.out.println("Connection created! Encryption algorithm is: " +
oraConn.getEncryptionAlgorithmName() + ", network
    integrity algorithm is: " + oraConn.getDataIntegrityAlgorithmName());
    oraConn.close();
  }

}
```

9.7 TLSのサポート

この節では、以下のトピックについて説明します。

- JDBCによるTLSのサポートの概要
- 証明書とウォレットの管理について
- キーと証明書のコンテナについて
- JDBC ThinおよびJKSを使用したTLSバージョン1.2でのデータベース接続
- TLS接続の自動構成
- キーストア・サービスのサポート

9.7.1 JDBCによるTLSのサポートの概要

Oracle Database 19cでは、Transport Layer Security (TLS)プロトコルがサポートされています。TLSは広く使用されている業界標準のプロトコルで、セキュアなネットワーク通信を提供します。TLSは、認証、データの暗号化およびデータの整合性を提供します。標準TCP/IPプロトコルのセキュリティ拡張で、インターネット通信に使用されます。

TLSでは、認証機能、および公開と秘密キー・ペアによる暗号化の機能を提供するのに、X.509v3標準準拠のデジタル証明書が使用されます。TLSではまた、データのプライバシと整合性を保証するために、秘密キー暗号化とデジタル署名も使用されます。TLSを介したネットワーク接続が開始されると、クライアントとサーバーでは次のようなステップを含むTLSハンドシェイクが実行されます:

- クライアントとサーバーとの間で、使用する暗号スイートを決めるためのネゴシエーションが行われます。これには、データ

転送に使用する暗号化アルゴリズムの決定も含まれます。

- サーバーが自分の証明書をクライアントに送信し、クライアントは、その証明書が信頼できる認証機関(CA)によって署名されているかどうかを検証します。このステップにより、サーバーの身元が検証されます。
- クライアント認証が必要な場合は、クライアントが自分自身の証明書をサーバーに送信し、サーバーは、その証明書が信頼できるCAによって署名されているかどうかを検証します。
- クライアントとサーバーが、公開キー暗号化を使用してキー情報を交換します。この情報に基づき、双方でセッション・キーが生成されます。クライアントとサーバーとの間の以降のやりとりはすべて、このセッション・キーのセットと、ネゴシエーションにより決定した暗号スイートを使用して、暗号化/復号化されます。

TLSに関する用語

TLSのコンテキストでは、次の用語がよく使用されます:

- 証明書: 証明書とは、公開キーとエンティティをバインドする、デジタル署名付きドキュメントのことです。証明書は、公開キーが適切なエンティティに属していることを検証するために使用します。
- 認証機関: 認証機関(CA)は、デジタル署名付き証明書(他のエンティティによって使用される)を発行するエンティティです。認証局とも呼ばれます。
- 暗号スイート: 暗号スイートとは、TLS対応のネットワーク上で送信されるデータを暗号化するために使用される、暗号化アルゴリズムとキー・サイズのセットのことです。
- 秘密キー: 秘密キーは、ネットワークを介して伝送されることのない、文字どおり秘密のキーです。秘密キーは、対応する公開キーを使用して暗号化されたメッセージを復号化するために使用されます。また、証明書に署名する際にも使用されます。その証明書を検証する際には、対応する公開キーが使用されます。
- 公開キー: 公開キーとは、公開したり、電子メール・メッセージのような通常の手段で送信したりできる暗号化キーのことです。公開キーは、TLS経由で送信されるメッセージを暗号化するために使用されます。また、対応する秘密キーによって署名された証明書を検証する際にも使用されます。
- キー・ストアまたはウォレット: ウォレットとは、資格証明の認証や署名を格納するために使用される、パスワード保護付きのコンテナのことです。ウォレット内には、TLSで必要となる、秘密キー、証明書、および信頼できる証明書が格納されます。
- セキュリティ・プロバイダ: セキュリティに関連したいくつかの機能を提供するJava実装。プロバイダは、キーストア・ファイルのデコードを行います。
- キーストア・サービス(KSS): Oracle Platform Securityサービスのコンポーネント。KSSを使用すると、キーストアはkss://スキームを使用したURIとして(ファイル名ではなく)参照されます。

JavaバージョンのTLS

Java Secure Socket Extension (JSSE)は、TLSプロトコルのJavaバージョンを使用するためのフレームワークと実装を提供するものです。JSSEは、データの暗号化、サーバーとクライアントの認証、およびメッセージの整合性保持をサポートします。アプリケーション開発者が直接アプリケーションに統合できる基礎要素が提供されるため、開発者は複雑なセキュリティ・アルゴリズムやハンドシェイク・メカニズムから解放され、アプリケーション開発が簡略化されます。JSSEは、Java Development Kit (JDK) 1.4以上に統合され、TLSバージョン2.0と3.0をサポートします。

Oracle JDBCドライバでTLSを使用する前に、JavaTM Secure Socket Extension (JSSE)フレームワークをよく理解しておくことをお薦めします。

JSSE標準のApplication Program Interface(API)は、javax.net、javax.net.sslおよびjavax.security.certの各パッケージで使用できます。これらのパッケージには、ソケット、サーバー・ソケット、TLSソケットおよびTLSサーバー・ソケットを作成および構成するためのクラスが用意されています。これらのパッケージには、セキュアなHTTP接続、JDK1.1ベースのプラットフォームと互換性がある公開キー証明書API、および信頼のおける主要なマネージャ用のインタフェースも用意されています。

TLSは、Oracle Database 18cの環境においても、他のあらゆるネットワーク環境の場合と同様に機能します。



ノート:

プログラムで JSSE を使用するには、JavaTM Secure Socket Extension (JSSE)フレームワークをよく理解している必要があります。

9.7.2 証明書とウォレットの管理について

Oracle Databaseサーバーは、JDBCクライアント(ThinまたはOCIのいずれか)とTLS接続を確立するために、ウォレットに格納されている自身の証明書を送信します。クライアントで証明書やウォレットが必要となるかどうかは、サーバーの構成によって決まります。

Oracle JDBC Thinドライバは、JSSEフレームワークを使用してTLS接続を作成します。デフォルトのプロバイダ(SunJSSE)を使用してTLSコンテキストを作成しますが、独自のプロバイダを指定できます。

サーバー上でSSL_CLIENT_AUTHENTICATIONパラメータが設定されていないかぎり、クライアント用の証明書は不要です。

9.7.3 キーと証明書のコンテナについて

Javaクライアントは、プロバイダさえあれば、Oracleウォレット、JKS、PKCS12など複数の種類のコンテナを使用することができます。Oracleウォレットの場合、SunJSSEプロバイダによって提供されるPKCS12サポートがPKCS12のすべての機能をサポートしているわけではないため、OraclePKIプロバイダを使用する必要があります。OraclePKIプロバイダを使用するには次のJARが必要です。

- oraclepki.jar
- osdt_cert.jar
- osdt_core.jar

これらすべてのJARファイルは$ORACLE_HOME/jlibディレクトリにあります。

9.7.4 JDBC ThinおよびJKSを使用したTLSバージョン1.2でのデータベース接続

この項では、TLS v1.2を使用してデータベースに接続するようにOracle JDBC Thinドライバを構成するステップについて説明します。

TLSバージョン1.2を使用してデータベースに接続するようにOracle JDBC Thinドライバを構成するには、次のステップを実行します。

- 必ずJDKの最新の更新を使用します

  更新されたバージョンにはTLSバージョン1.2の使用に必要なバグ修正が含まれているため、JDK 7またはJDK 8のい

ずれかの最新の更新を使用します。

- JCEファイルをインストールします

  JDKにJava Cryptography Extension (JCE)無制限強度の管轄ポリシー・ファイルをインストールします。使用しているJDKバージョンに関係なく、これらのファイルがなくては強力な暗号スイート(TLS_RSA_WITH_AES_256_CBC_SHA256など)が有効にならないためです。これらのファイルは次のページでダウンロードできます。

  http://www.oracle.com/technetwork/java/javase/downloads/jce-7-download-432124.html

  また、JDK 7を使用している場合、強力な暗号スイートを明示的に有効にする必要があります。たとえば、JDK 7でTLS_RSA_WITH_AES_256_CBC_SHA256などの強力な暗号スイートを使用している場合、oracle.net.ssl_cipher_suites接続プロパティでこれを有効にする必要があります。

- JKSファイルまたはウォレットを使用します



ノート:

Oracle Database リリース 18c 以降、ojdbc.properties という新しい構成ファイルで TLS の構成プロパティを指定できます。このファイルを使用すると、クラウドでのデータベース・サービスへの接続が容易になります。

関連項目:

Oracle Database JDBC Java APIリファレンス

前述のステップをすべて実行した後に問題が発生した場合、トレースをオンにして-Djavax.net.debug=allオプションを使用して問題を診断します。

9.7.5 TLS接続の自動構成

Oracle Databaseリリース18c以降、デフォルト値を使用するか、セキュリティ・プロバイダを手動で追加または更新することなく接続構成値を解決するためのプログラム・ロジックを使用することができます。次の2つの方法で構成値を解決できます。

- プロバイダの解決
- キーストア・タイプ(KSS)の自動解決

9.7.5.1 プロバイダの解決

特定のキーストア・タイプの場合、JDBCドライバはキーストアをロードするために使用されるプロバイダ実装を解決できます。これらのタイプの場合、Javaセキュリティを使用してプロバイダを登録する必要はありません。プロバイダ実装がCLASSPATHにあるかぎり、ドライバはセキュリティ・プロバイダをインスタンス化できます。

次のキーストア・タイプは、既知のプロバイダにマップされます。

- SSO: oracle.security.pki.OraclePKIProvider
- KSS: oracle.security.jps.internal.keystore.provider.FarmKeyStoreProvider

ドライバは、指定されたタイプにプロバイダが登録されていない場合にのみ、プロバイダを解決しようとします。

oraclepki.jarファイルがCLASSPATHにある場合、ドライバはOracle PKIプロバイダを次の方法で自動的にロードできます。

```
java -cp oraclepki.jar:ojdbc8.jar -D javax.net.ssl.keyStore=/path/to/wallet/cwallet.sso MyApp
```

同様に、oracle.net.wallet_location接続プロパティの指定値の場合、ドライバはOracle PKIプロバイダを次の方法で自動的にロードできます。

```
java -cp .:oraclepki.jar:ojdbc8.jar -D oracle.net.wallet_location=file:/path/to/wallet/cwallet.sso
MyApp
```

ノート:

orapki ツール(ewallet.p12 ファイル)によって作成された PKCS12 タイプの場合、orapki ツールによって作成された PKCS12 ファイルには ASN1 Key Bag 要素(タイプ・コード: 1.2.840.113549.1.12.10.1.1)が含まれているため、Java セキュリティを使用して OraclePKIProvider を登録する必要があります。Sun PKCS12 実装は Key Bag タイプをサポートしていないため、ewallet.p12 ファイルの読取りを試みると、エラーがスローされます。HotSpot および Open JDK ユーザーの場合、Sun プロバイダは、PKCS12 プロバイダとしてバンドルされます。つまり、PKCS12 プロバイダにはすでに登録済のプロバイダがあり、ドライバはこれをオーバーライドしようとしません。

9.7.5.2 キーストア・タイプ(KSS)の自動解決

JDBCドライバでは、javax.net.ssl.keyStoreおよびjavax.net.ssl.trustStoreプロパティの値に基づいて共通のキーストア・タイプを解決できるため、これらのプロパティを使用してタイプを指定する必要がなくなります。

認識済のファイル拡張子を持つキーストアまたはトラスト・ストア

認識済のファイル拡張子を持つキーストアまたはトラスト・ストアは、次のタイプにマップされます。

- ファイル拡張子.jksは、JKSとしてjavax.net.ssl.keyStoreTypeに解決されます。

  ```
  java -cp ojdbc8.jar -D javax.net.ssl.keyStore=/path/to/keystore/keystore.jks MyApp
  ```

- ファイル拡張子.ssoは、SSOとしてjavax.net.ssl.keyStoreTypeに解決されます。

  ```
  java -cp ojdbc8.jar -D
  javax.net.ssl.keyStore=/path/to/keystore/keystore.sso MyApp
  ```

- ファイル拡張子.p12は、PKCS12としてjavax.net.ssl.keyStoreTypeに解決されます。

  ```
  java -cp ojdbc8.jar -D
  javax.net.ssl.keyStore=/path/to/keystore/keystore.p12 MyApp
  ```

- ファイル拡張子.pfxは、PKCS12としてjavax.net.ssl.keyStoreTypeに解決されます。

  ```
  java -cp ojdbc8.jar -D
  javax.net.ssl.keyStore=/path/to/keystore/keystore.pfx MyApp
  ```

URIを使用したキーストアまたはトラスト・ストア

キーストアまたはトラスト・ストアがkss://スキームを使用したURIである場合、KSSタイプにマップされます。

```
java -cp ojdbc8.jar -D
javax.net.ssl.keyStore=kss://MyStripe/MyKeyStore MyApp
```



ノート:

デフォルト・タイプの解決をオーバーライドするために、javax.net.ssl.trustStoreType および javax.net.ssl.keyStoreType プロパティを設定できます。

9.7.6 デフォルトのTLSコンテキストのサポート

TLS構成に対してよりきめ細やかな制御を必要とするアプリケーションの場合は、SSLContext.getDefaultメソッドによって返されるSSLContextを使用するように、JDBCドライバを構成できます。ドライバでデフォルトのSSLContextを使用するには、次のいずれかの方法を使用します。

- javax.net.ssl.keyStore=NONE
- javax.net.ssl.trustStore=NONE

デフォルトのSSLContextを使用して、ファイルベースではないキーストア・タイプをサポートできます。このようなキーストア・タイプの一般的な例には、ハードウェアベースのスマート・カードが含まれます。load(KeyStore.LoadStoreParameter)メソッドへのプログラム的なコールを必要とするキーストア・タイプもこのカテゴリに属します。

関連項目:

- https://docs.oracle.com/javase/8/docs/api/javax/net/ssl/SSLContext.html#getDefault--
- https://docs.oracle.com/javase/8/docs/api/java/security/KeyStore.html#load-java.security.KeyStore.LoadStoreParameter-

9.7.7 キーストア・サービスのサポート

このリリースのOracle Databaseでは、JDBCドライバでのキーストア・サービス(KSS)のサポートが導入されています。つまり、WebLogic Serverでキーストア・サービスを構成した場合、JDBCアプリケーションを既存のキーストア・サービス構成と統合できるようになりました。

ドライバは、キーストア・サービスによって管理されるキーストアをロードできます。javax.net.ssl.keyStoreプロパティまたはjavax.net.ssl.trustStoreプロパティの値がkss://のスキームのURIである場合、ドライバはキーストア・サービスからキーストアをロードします。

権限ベースの保護の場合、次のパーミッションをojdbc JARファイルに付与する必要があります。

```
permission KeyStoreAccessPermission
"stripeName=*,keystoreName=*,alias=*", "read";
```

この権限は、すべてのキーストアへのアクセス権を付与します。アクセス権の範囲を制限するには、アスタリスクのワイルドカード(*)を、特定のアプリケーション・ストライプおよびキーストア名と置き換えます。ドライバは、キーストアを権限付きアクションとしてロードできません。これは、KeyStoreAccessPermissionもアプリケーションのコード・ベースに付与する必要があることを意味します。

9.8 Kerberosのサポート

この項では、次の項目について説明します。

- JDBCによるKerberosのサポートの概要
- Kerberosを使用するためのWindowsの構成
- Kerberosを使用するためのOracle Databaseの構成
- Kerberosを使用するコード例

9.8.1 JDBCによるKerberosのサポートの概要

Kerberosはネットワーク経由で認証ツールと強固な暗号化ツールを提供するネットワーク認証プロトコルです。Kerberosは秘密キー暗号化を使用する方式で、全社の情報システムのセキュリティ確保を容易に実現できます。Kerberosプロトコルは強固な暗号化を使用しているため、安全でないネットワーク接続を経由しても、クライアントまたはサーバーがサーバーまたはクライアントに対して身元を証明することができます。Kerberosを使用して身元を証明したクライアントおよびサーバーは、すべての通信を暗号化して、業務においてプライバシとデータ整合性を保全することができます。

Kerberosアーキテクチャでは、Key Distribution Center (KDC)と呼ばれる信頼できる認証サービスが軸となります。Kerberos環境のユーザーおよびサービスはプリンシパルと呼ばれ、各プリンシパルはKDCと秘密鍵(パスワードなど)を共有します。プリンシパルはHRなどのユーザーの場合も、データベース・サーバーのインスタンスの場合もあります。

12cリリース1以降、Oracle DatabaseではKerberosのレルム間認証もサポートしています。取得したレルムをKerberos構成ファイルのdomain_realmsセクションに適切に追加すると、ある特定のレルム内で、別のレルムのサービスにアクセスできます。

9.8.2 Kerberosを使用するためのWindowsの構成

klist、kinitなどのツールを提供する良質なKerberosクライアントは、次のリンクの場所にあります。

```
http://web.mit.edu/kerberos/dist/index.html
```

このクライアントでは便利なGUIも提供しています。

WindowsマシンでKerberosを構成するには、次の変更を行う必要があります。

1. デスクトップで「マイ コンピュータ」アイコンを右クリックします。
2. 「プロパティ」を選択します。「システムのプロパティ」ダイアログ・ボックスが表示されます。
3. 「詳細」タブを選択します。
4. 「環境変数」をクリックします。「環境変数」ダイアログ・ボックスが表示されます。
5. 「新規」をクリックして新しい利用者変数を追加します。「新しいユーザー変数」ダイアログ・ボックスが表示されます。
6. 「変数名」フィールドに「KRB5CCNAME」と入力します。
7. 「変数値」フィールドに「FILE:C:¥Documents and Settings¥<user_name>¥krb5cc」と入力します。
8. 「OK」をクリックして「新しいユーザー変数」ダイアログ・ボックスを閉じます。
9. 「OK」をクリックして「環境変数」ダイアログ・ボックスを閉じます。
10. 「OK」をクリックして「システムのプロパティ」ダイアログ・ボックスを閉じます。



ノート:

C:¥WINDOWS¥krb5.ini ファイルは、krb5.conf ファイルと内容が同じです。

9.8.3 Kerberosを使用するためのOracle Databaseの構成

Kerberosを使用するには、次のステップでOracle Databaseを構成します。

1. 次のコマンドを使用してデータベースに接続します。

```
SQL> connect system
Enter password: password
```

2. 次のコマンドを使用して、外部で身元が保証されたユーザーCLIENT@US.ORACLE.COMを作成します。

```
SQL> create user "CLIENT@US.ORACLE.COM" identified externally;
SQL> grant create session to "CLIENT@US.ORACLE.COM";
```

3. 次のコマンドを使用して、sysdbaとしてデータベースに接続し、ディスマウントします。

```
SQL> connect / as sysdba
SQL> shutdown immediate;
```

4. $T_WORK/t_init1.oraファイルに次の行を追加します。

```
OS_AUTHENT_PREFIX=""
```

5. 次のコマンドを使用してデータベースを再起動します。

```
SQL> startup pfile=t_init1.ora
```

6. sqlnet.oraファイルを編集して次の行を入れます。

```
names.directory_path = (tnsnames)
#Kerberos
sqlnet.authentication_services = (beq,kerberos5)
sqlnet.authentication_kerberos5_service = dbji
sqlnet.kerberos5_conf = /home/Jdbc/Security/kerberos/krb5.conf
sqlnet.kerberos5_keytab = /home/Jdbc/Security/kerberos/dbji.oracleserver
sqlnet.kerberos5_conf_mit = true
sqlnet.kerberos_cc_name = /tmp/krb5cc_5088
# logging (optional):
trace_level_server=16
trace_directory_server=/scratch/sqlnet/
```

7. 次のコマンドを使用して、SQL*Plusを経由して接続できることを確認します。

```
> kinit client
> klist
    Ticket cache: FILE:/tmp/krb5cc_5088
    Default principal: client@US.ORACLE.COM

    Valid starting     Expires            Service principal
    06/22/06 07:13:29  06/22/06 17:13:29  krbtgt/US.ORACLE.COM@US.ORACLE.COM


    Kerberos 4 ticket cache: /tmp/tkt5088
```

```
    klist: You have no tickets cached
> sqlplus '/@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=oracleserver.mydomain.com)(PORT=5221))
(CONNECT_DATA=(SERVICE_NAME=orcl)))'
```

9.8.4 Kerberosを使用するコード例

次の例は、Oracle Database 12cリリース1 (12.1)のJDBC Thinドライバに含まれている新しいKerberos認証機能を示しています。このデモは、次の2つの使用例を説明しています。

- 最初のシナリオでは、OSがユーザー名と資格証明をメンテナンスします。資格証明はキャッシュに格納され、ドライバは資格証明を取得してからサーバーの認証を受けようとします。この使用例はモジュールconnectWithDefaultUser()にあります。



ノート:

1. デモのこの部分を実行する前に、次のコマンドを使用して、有効な資格証明を持っていることを確認してください。

```
> /usr/kerberos/bin/kinit client
where, the password is welcome.
```

2. 次のコマンドを使用してチケットをリストします。

```
> /usr/kerberos/bin/klist
```

- 2番目の使用例は、アプリケーションがユーザーの資格証明を制御する事例です。これは、複数のWebユーザーが各自の資格証明を持っているアプリケーション・サーバーの場合です。この使用例はモジュールconnectWithSpecificUser()にあります。



ノート:

このデモを実行するには、作業用の設定として、稼働中の Kerberos サーバーと、Kerberos 認証を使用するように構成されている Oracle Database サーバーが必要です。例をコンパイルして実行するには、使用されている URL を変更する必要があります。

例9-4 Kerberos認証を使用したデータベースへの接続

```
import com.sun.security.auth.module.Krb5LoginModule;
import java.io.IOException;

import java.security.PrivilegedExceptionAction;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import java.util.HashMap;
import java.util.Properties;
import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
```

```
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.UnsupportedCallbackException;

import oracle.jdbc.OracleConnection;
import oracle.jdbc.OracleDriver;
import oracle.net.ano.AnoServices;
public class KerberosJdbcDemo
{
  String url ="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)"+
    "(HOST=oracleserver.mydomain.com)(PORT=5221))(CONNECT_DATA=" +
    "(SERVICE_NAME=orcl)))";

  public static void main(String[] arv)
  {
    /* If you see the following error message [Mechanism level: Could not load
     * configuration file c:¥winnt¥krb5.ini (The system cannot find the path
     * specified] it's because the JVM cannot locate your kerberos config file.
     * You have to provide the location of the file. For example, on Windows,
     * the MIT Kerberos client uses the config file: C¥WINDOWS¥krb5.ini:
     */
    // System.setProperty("java.security.krb5.conf","C:¥¥WINDOWS¥¥krb5.ini");
    System.setProperty("java.security.krb5.conf","/home/Jdbc/Security/kerberos/krb5.conf");

    KerberosJdbcDemo kerberosDemo = new KerberosJdbcDemo();
    try
    {
      System.out.println("Attempt to connect with the default user:");
      kerberosDemo.connectWithDefaultUser();
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
    try
    {
      System.out.println("Attempt to connect with a specific user:");
      kerberosDemo.connectWithSpecificUser();
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }


  void connectWithDefaultUser() throws SQLException
  {
    OracleDriver driver = new OracleDriver();
    Properties prop = new Properties();

    prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_AUTHENTICATION_SERVICES,
      "("+AnoServices.AUTHENTICATION_KERBEROS5+")");
    prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_AUTHENTICATION_KRB5_MUTUAL,
      "true");

    /* If you get the following error [Unable to obtain Princpal Name for
     * authentication] although you know that you have the right TGT in your
```

```
     * credential cache, then it's probably because the JVM can't locate your
     * cache.

     *
     * Note that the default location on windows is "C:¥Documents and Settings¥krb5cc_username".
     */

    // prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_AUTHENTICATION_KRB5_CC_NAME,

    /*
      On linux:
         > which kinit
         /usr/kerberos/bin/kinit
         > ls -l /etc/krb5.conf
         lrwxrwxrwx    1 root  root   47 Jun 22 06:56 /etc/krb5.conf ->
/home/Jdbc/Security/kerberos/krb5.conf

         > kinit client
         Password for client@US.ORACLE.COM:
         > klist
         Ticket cache: FILE:/tmp/krb5cc_5088
         Default principal: client@US.ORACLE.COM

         Valid starting     Expires            Service principal
         11/02/06 09:25:11  11/02/06 19:25:11  krbtgt/US.ORACLE.COM@US.ORACLE.COM


         Kerberos 4 ticket cache: /tmp/tkt5088
         klist: You have no tickets cached
    */
    prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_AUTHENTICATION_KRB5_CC_NAME,
                     "/tmp/krb5cc_5088");
    Connection conn  = driver.connect(url,prop);
    String auth = ((OracleConnection)conn).getAuthenticationAdaptorName();
    System.out.println("Authentication adaptor="+auth);
    printUserName(conn);
    conn.close();
  }


  void connectWithSpecificUser() throws Exception
  {
    Subject specificSubject = new Subject();

    // This first part isn't really meaningful to the sake of this demo. In
    // a real world scenario, you have a valid "specificSubject" Subject that
    // represents a web user that has valid Kerberos credentials.
    Krb5LoginModule krb5Module = new Krb5LoginModule();
    HashMap sharedState = new HashMap();
    HashMap options = new HashMap();
    options.put("doNotPrompt","false");
    options.put("useTicketCache","false");
    options.put("principal","client@US.ORACLE.COM");

    krb5Module.initialize(specificSubject,newKrbCallbackHandler(),sharedState,options);
    boolean retLogin = krb5Module.login();
    krb5Module.commit();
    if(!retLogin)
```

```
      throw new Exception("Kerberos5 adaptor couldn't retrieve credentials (TGT) from the cache");

    // to use the TGT from the cache:
    // options.put("useTicketCache","true");
    // options.put("doNotPrompt","true");
    // options.put("ticketCache","C:¥¥Documents and Settings¥¥user¥¥krb5cc");
    // krb5Module.initialize(specificSubject,null,sharedState,options);


    // Now we have a valid Subject with Kerberos credentials. The second scenario
    // really starts here:
    // execute driver.connect(...) on behalf of the Subject 'specificSubject':
    Connection conn =
      (Connection)Subject.doAs(specificSubject, new PrivilegedExceptionAction()
        {
          public Object run()
          {
            Connection con = null;
            Properties prop = new Properties();
            prop.setProperty(AnoServices.AUTHENTICATION_PROPERTY_SERVICES,
                             "(" + AnoServices.AUTHENTICATION_KERBEROS5 + ")");
            try
            {
              OracleDriver driver = new OracleDriver();
              con = driver.connect(url, prop);

            } catch (Exception except)
            {
              except.printStackTrace();
            }
            return con;
          }
        });

    String auth = ((OracleConnection)conn).getAuthenticationAdaptorName();
    System.out.println("Authentication adaptor="+auth);
    printUserName(conn);
    conn.close();
  }

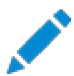
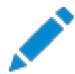
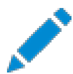
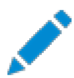
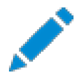
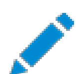
  void printUserName(Connection conn) throws SQLException
  {
    Statement stmt = null;
    try
    {
      stmt = conn.createStatement();
      ResultSet rs = stmt.executeQuery("select user from dual");
      while(rs.next())
        System.out.println("User is:"+rs.getString(1));
      rs.close();
    }
    finally
    {
      if(stmt != null)
        stmt.close();
    }
  }
}
```

```
class KrbCallbackHandler implements CallbackHandler
{
 public void handle(Callback[] callbacks) throws IOException,
                                                 UnsupportedCallbackException
 {
   for (int i = 0; i < callbacks.length; i++)
   {
     if (callbacks[i] instanceof PasswordCallback)
     {
       PasswordCallback pc = (PasswordCallback)callbacks[i];
       System.out.println("set password to 'welcome'");
       pc.setPassword((new String("welcome")).toCharArray());
     } else
     {
       throw new UnsupportedCallbackException(callbacks[i],
                                              "Unrecognized Callback");
     }
   }
 }
}
```

9.9 RADIUSのサポート

この項では、次の概念について説明します。

- JDBCによるRADIUSのサポートの概要
- RADIUSを使用するためのOracle Databaseの構成
- RADIUSを使用するコード例

9.9.1 JDBCによるRADIUSのサポートの概要

Oracle Database 11gリリース1で、リモート認証ダイアルイン・ユーザー・サービス(RADIUS)が導入されました。RADIUSは、最も広く知られている、リモートでの認証とアクセスを可能にするクライアント/サーバー・セキュリティ・プロトコルです。Oracle Advanced Securityは、この標準をクライアント/サーバー・ネットワーク環境で使用して、RADIUSプロトコルをサポートするあらゆる認証方式の使用を可能にします。RADIUSは、トークン・カードやスマートカードなど、いろいろな認証メカニズムで使用できます。

9.9.2 RADIUSを使用するためのOracle Databaseの構成

RADIUSを使用するには、次のステップでOracle Databaseを構成します。

1. 次のコマンドを使用してデータベースに接続します。

```
SQL> connect system
Enter password: password
```

2. 次のコマンドを使用して、データベース内から新規ユーザーasoを作成します。

```
SQL> create user aso identified externally;
SQL> grant create session to aso;
```

3. 次のコマンドを使用して、sysdbaとしてデータベースに接続し、ディスマウントします。

```
SQL> connect / as sysdba
SQL> shutdown immediate;
```

4. t_init1.oraファイルに次の行を追加します。

```
os_authent_prefix = ""
```

ノート:

テストが終了したら、t_init1.ora ファイルに加えられたこれまでの変更を元に戻す必要があります。

5. 次のコマンドを使用してデータベースを再起動します。

```
SQL> startup pfile=?/work/t_init1.ora
```

6. 次の行のみが含まれるようにsqlnet.oraファイルを変更します。

```
sqlnet.authentication_services = ( beq, radius)
sqlnet.radius_authentication = <RADUIUS_SERVER_HOST_NAME>
sqlnet.radius_authentication_port = 1812
sqlnet.radius_authentication_timeout = 120
sqlnet.radius_secret=/home/Jdbc/Security/radius/radius_key
# logging (optional):
trace_level_server=16
trace_directory_server=/scratch/sqlnet/
```

7. 次のコマンドを使用して、SQL*Plusを経由して接続できることを確認します。

```
>sqlplus
'aso/1234@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=oracleserver.mydomain.com)(PORT=5221))
(CONNECT_DATA=(SERVICE_NAME=orcl)))'
```

9.9.3 RADIUSを使用するコード例

次の例は、Oracle Database 12cリリース1 (12.1)のJDBC Thinドライバに含まれている新しいRADIUS認証機能を示しています。作業用の設定として、稼働中のRADIUSサーバーと、RADIUS認証を使用するように構成されているOracle Databaseサーバーが必要です。例をコンパイルして実行するには、指定されているURLを変更する必要があります。

例9-5 RADIUS認証を使用したデータベースへの接続

```
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.OracleDriver;
import oracle.net.ano.AnoServices;
public class RadiusJdbcDemo
{
  String url ="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)"+
    "(HOST=oracleserver.mydomain.com)(PORT=5221))(CONNECT_DATA=" +
```

```
  "(SERVICE_NAME=orcl)))";

public static void main(String[] arv)
{
  RadiusJdbcDemo radiusDemo = new RadiusJdbcDemo();
  try
  {
    radiusDemo.connect();
  }
  catch (Exception e)
  {
    e.printStackTrace();
  }
}

/*
 * This method attempts to logon to the database using the RADIUS
 * authentication protocol.
 *
 * It should print the following output to stdout:
 * -----------------------------------------------------
 * Authentication adaptor=RADIUS
 * User is:ASO
 * -----------------------------------------------------
 */
void connect() throws SQLException
{
  OracleDriver driver = new OracleDriver();
  Properties prop = new Properties();

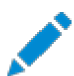
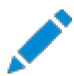
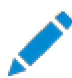
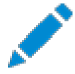
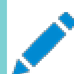
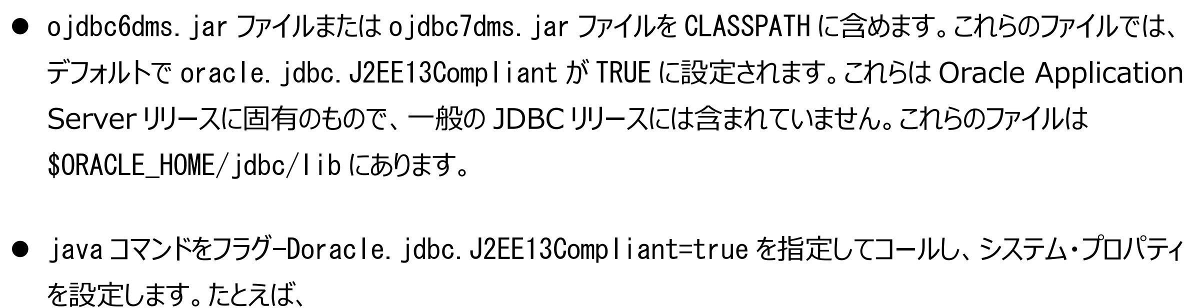
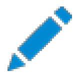
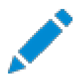
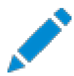
  prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_AUTHENTICATION_SERVICES,
    "("+AnoServices.AUTHENTICATION_RADIUS+")");
  // The user "aso" needs to be properly setup on the radius server with
  // password "1234".
  prop.setProperty("user","aso");
  prop.setProperty("password","1234");

  Connection conn  = driver.connect(url,prop);
  String auth = ((OracleConnection)conn).getAuthenticationAdaptorName();
  System.out.println("Authentication adaptor="+auth);
  printUserName(conn);
  conn.close();
}

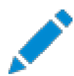
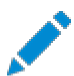
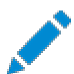
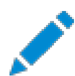
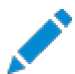
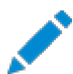
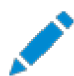

void printUserName(Connection conn) throws SQLException
{
  Statement stmt = null;
  try
  {
    stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery("select user from dual");
    while(rs.next())
      System.out.println("User is:"+rs.getString(1));
    rs.close();
  }
  finally
  {
```

```
      if(stmt != null)
        stmt.close();
    }
  }
}
```

9.10 セキュアな外部パスワード記憶域

データベースへの接続にアプリケーションでパスワード資格証明を使用するような大規模なデプロイメントの場合、クライアント側のOracleウォレットを使用してこのような資格証明を格納できます。Oracle Walletは、認証およびサインインの資格証明の格納に使用されるセキュアなソフトウェア・コンテナです。

クライアント側のOracleウォレットにデータベースのパスワード資格証明を格納すると、アプリケーション・コードやバッチ・ジョブ、スクリプトにユーザー名とパスワードを埋め込む必要がなくなります。このため、スクリプトやアプリケーション・コード内にパスワードが露出する危険性が減少し、ユーザー名やパスワードを変更するたびにコードを変更する必要がなくなるため、管理が単純化されます。また、アプリケーション・コードを変更する必要がないため、これらのユーザー・アカウントに対してパスワード管理ポリシーをさらに簡単に強制適用できるようになります。

`oracle.net.wallet_location`接続プロパティを設定して、ウォレットの場所を指定できます。JDBCドライバは、このウォレットからユーザー名とパスワードのペアを取得できます。



ノート:

Oracle ウォレットを JDBC アプリケーションで開くと、セキュリティ上の理由から、ウォレット所有者(作成者)のみがアクセスできるようにウォレット・ファイルの権限が調整されます。

関連項目:

- セキュアな外部パスワード・ストアを使用するようにクライアントを構成する方法およびそのパスワード・ストア内の資格証明を管理する方法の詳細は、『Oracle Database管理者ガイド』を参照してください。
- パスワード資格証明のためのセキュアな外部パスワード・ストアの管理方法の詳細は、『Oracle Databaseセキュリティ・ガイド』を参照してください。

10 プロキシ認証

Oracle Java Database Connectivity(JDBC)は、n層認証とも呼ばれるプロキシ認証を提供しています。この機能は、JDBC Oracle Call Interface(OCI)ドライバとJDBC Thinドライバの両方を介してサポートされています。この章の構成は、次のとおりです。

- プロキシ認証について
- 各種のプロキシ接続
- プロキシ接続の作成
- プロキシ・セッションのクローズ
- プロキシ接続のキャッシュ
- プロキシ接続の制限



ノート:

Oracle Database はプロキシ認証機能を 3 つの層のみでサポートしています。複数の中間層を横断してのサポートはありません。

10.1 プロキシ認証について

プロキシ認証はユーザー認証に中間層を使用する処理です。次の3つの形式のプロキシ認証を使用することによって、クライアントのセキュアなプロキシとして動作する中間層サーバーを設計することができます。

- 中間層サーバーはデータベース・サーバーおよびクライアントに対してそれ自身を認証します。この場合、アプリケーション・ユーザーまたは別のアプリケーションは中間層サーバーに対して自分自身を認証します。クライアントの識別情報は、データベースに到達するまで確実に保持されます。
- クライアント、つまりデータベース・ユーザーは、中間層サーバーによっては認証されません。クライアントのIDとデータベース・パスワードは、中間層サーバーを経由してデータベース・サーバーに渡され、認証に使用されます。
- クライアント、つまりグローバル・ユーザーは中間層サーバーによって認証され、クライアントのユーザー名を取得するために中間層を経由して識別名(DN)または証明書を渡します。



ノート:

中間層サーバーがクライアントの代理として行った処理は監査できます。

すべての場合で、クライアントのプロキシ(代理)として動作するために、管理者は中間層サーバーを認証する必要があります。たとえば、中間層サーバーが最初にユーザーHRとしてデータベースに接続し、プロキシ接続をユーザーjeffとしてアクティブにしてから、次の文を発行して、クライアントのプロキシとして動作するように中間層サーバーを認証するとします。

```
ALTER USER jeff GRANT CONNECT THROUGH HR;
```

また、次のことも可能です。

- クライアントとして接続する場合に中間層がアクティブにすることを許可されるロールを指定すること。たとえば、

```
CREATE ROLE role1;
GRANT SELECT ON employees TO role1;
ALTER USER jeff GRANT CONNECT THROUGH HR ROLE role1;
```

  このrole句は、ロール・リストに指定されたデータベース・オブジェクトのみにアクセスを制限します。空のロール・リストを指定することも可能です。

- PROXY_USERSデータ・ディクショナリ・ビューに問い合せて、中間層への接続を現在認可されているユーザーを検索すること。

- ALTER USER文のREVOKE CONNECT THROUGH句を使用してプロキシ接続の認可を取り消すこと。



ノート:

プロキシ認証の場合、認証中に、データベースへの JDBC 接続でデータベース・セッションが作成され、接続の存続中に他のセッションが作成されます。

各種のプロキシ接続を設定するには、oracle.jdbc.OracleConnectionインタフェースにある、各種のフィールドやメソッドを使用する必要があります。

10.2 各種のプロキシ接続

プロキシ接続は、次のいずれかのオプションを使用して作成できます。

- USER NAME

  このオプションは、ユーザー名またはパスワード(あるいはその両方)を指定して実行します。パスワードを使用する認証を指定するには、次のSQL文を使用します。

```
ALTER USER jeff GRANT CONNECT THROUGH HR AUTHENTICATED USING PASSWORD;
```

  この場合、jeffはユーザー名で、HRはjeffのプロキシです。

  パスワード・オプションは、セキュリティを強化するためのものです。authenticated句がない場合、パスワードがなく、ユーザー名のみを使用するデフォルトの認証が使用されます。デフォルト認証を指定するには、次のSQL文を使用します。

```
ALTER USER jeff GRANT CONNECT THROUGH HR
```

- DISTINGUISHED NAME

  この識別名は、プロキシされるユーザーのパスワードにかわるグローバル名です。識別名を使用する場合のSQL文の例を次に示します。

```
CREATE USER jeff IDENTIFIED GLOBALLY AS
'CN=jeff,OU=americas,O=oracle,L=redwoodshores,ST=ca,C=us';
```

  identified globally as句の後に続く文字列が識別名です。次に、この識別名を使用して認証する必要があります。識別名を使用して認証を指定するには、次のSQL文を使用します。

```
ALTER USER jeff GRANT CONNECT THROUGH HR AUTHENTICATED USING DISTINGUISHED NAME;
```

- CERTIFICATE

  これは、プロキシされるユーザーの資格証明をデータベースに渡す、より暗号化された方法です。証明書には、エンコードされた識別名が含まれます。証明書を生成する1つの方法として、ウォレットを作成した後、そのウォレットをデコードして証明書を取得する方法があります。ウォレットはrunutl mkwalletを使用して作成できます。その後、生成した証明書を使用して認証を受ける必要があります。証明書を使用する認証を指定するには、次のSQL文を使用します。

```
ALTER USER jeff GRANT CONNECT THROUGH HR AUTHENTICATED USING CERTIFICATE;
```



ノート:

証明書によるプロキシ認証は、将来の Oracle Database リリースではサポート対象外となります。

ノート:

- すべてのオプションは、ロールと関連付けることができます。
- プロキシとしての役割を果す新しい接続をオープンすると、新しいセッションがデータベース・サーバー上で開始されます。グローバル・トランザクションを開始してから、openProxySession メソッドをコールした場合、この時点で、グローバル・トランザクションの一部ではなくなります。そのかわり、新たに作成された JDBC 接続に含まれているようになります。通常は、グローバル・トランザクションを作成または再開する前に openProxySession メソッドがコールされるため、これが発生することはありません。この場合、まだグローバル・トランザクションの一部です。

10.3 プロキシ接続の作成

ユーザー(たとえば、jeff)は、別のユーザー(たとえば、HR)を介してデータベースに接続する必要があります。プロキシ・ユーザーHRには、アクティブな認証済接続が必要です。ドライバがサーバーにコマンドを送信してユーザーjeffのセッションを作成することで、このアクティブな接続上にプロキシ・セッションが作成されます。サーバーは新しいセッションIDを戻し、ドライバはセッション切替えコマンドを送信してこの新しいセッションに切り替えます。

JDBC OCIとThinドライバは、同じ方法でセッションを切り替えます。ドライバは、永続的に新しいセッションjeffに切り替えます。その結果、プロキシ・セッションHRは、新しいセッションjeffがクローズされるまで使用できません。



ノート:

oracle.jdbc.OracleConnection インタフェースから isProxySession メソッドを使用して、接続に関連付けられている現在のセッションがプロキシ・セッションであるかどうかをチェックすることができます。接続に関連付けられている現在のセッションがプロキシ・セッションの場合、このメソッドは true を戻します。

oracle.jdbc.OracleConnectionインタフェースから、次のメソッドを使用することで、新しいプロキシ・セッションが開かれます。

```
void openProxySession(int type, java.util.Properties prop) throws SQLExceptionOpens
```

ここで、

typeはプロキシ・セッションのタイプで、次の値を指定できます。

- OracleConnection.PROXYTYPE_USER_NAME

  このタイプのセッションは、ユーザー名を指定するために使用されます。

- OracleConnection.PROXYTYPE_DISTINGUISHED_NAME

  このタイプのセッションは、ユーザーの識別名を指定するために使用されます。

- OracleConnection.PROXYTYPE_CERTIFICATE

  このタイプのセッションは、プロキシ証明書を指定するために使用されます。

propはプロキシ・セッションのプロパティ値で、次の値を指定できます。

- PROXY_USER_NAME

  このプロパティ値はタイプOracleConnection.PROXYTYPE_USER_NAMEで使用できます。値はjava.lang.Stringになります。

- PROXY_DISTINGUISHED_NAME

  このプロパティ値はタイプOracleConnection.PROXYTYPE_DISTINGUISHED_NAMEで使用できます。値はjava.lang.Stringになります。

- PROXY_CERTIFICATE

  このプロパティ値はタイプOracleConnection.PROXYTYPE_CERTIFICATEで使用できます。値は、証明書が含まれるbytep[]配列です。

- PROXY_ROLES

  このプロパティ値は次のタイプで使用できます。

  - OracleConnection.PROXYTYPE_USER_NAME
  - OracleConnection.PROXYTYPE_DISTINGUISHED_NAME
  - OracleConnection.PROXYTYPE_CERTIFICATE

  値はjava.lang.Stringになります。

- PROXY_SESSION

  このプロパティ値は、プロキシ・セッションを閉じるためにcloseメソッドで使用されます。

- PROXY_USER_PASSWORD

  このプロパティ値はタイプOracleConnection.PROXYTYPE_USER_NAMEで使用できます。値はjava.lang.Stringになります。

次のコードは、openProxySessionメソッドの使用方法を示します。

```
java.util.Properties prop = new java.util.Properties();
prop.put(OracleConnection.PROXY_USER_NAME, "jeff");
String[] roles = {"role1", "role2"};
prop.put(OracleConnection.PROXY_ROLES, roles);
conn.openProxySession(OracleConnection.PROXYTYPE_USER_NAME, prop);
```

10.4 プロキシ・セッションのクローズ

OracleConnection.closeメソッドにOracleConnection.PROXY_SESSIONパラメータを渡すことによって、OracleConnection.openProxySessionメソッドでオープンしたプロキシ・セッションをクローズするには、次の操作を行います。

```
OracleConnection.close(OracleConnection.PROXY_SESSION);
```

これは、キャッシュされていない接続でのプロキシ・セッションのクローズと同じです。接続自体をクローズするには、標準のcloseメソッドを明示的にコールする必要があります。プロキシ・セッションをクローズせずにcloseメソッドを直接コールすると、プロキシ・セッションと接続の両方がクローズします。その方法は次のとおりです。

```
OracleConnection.close(OracleConnection.INVALID_CONNECTION);
```

10.5 プロキシ接続のキャッシュ

プロキシ接続は、通常の接続と同様にキャッシュできます。プロキシ接続をキャッシュすると、パフォーマンスが向上します。プロキシ接続をキャッシュするには、getConnectionメソッドのいずれかを使用して、キャッシュ対応のOracleDataSourceオブジェクトで接続を作成する必要があります。

プロキシ接続は、接続キャッシュの接続属性機能を使用して、接続キャッシュ内にキャッシュできます。接続属性はユーザー定義の名前/値ペアで、接続を接続キャッシュに戻して再利用する前に接続にタグを付けるのに役立ちます。タグ付けされた接続が取得されると、その接続を直接使用してプロキシ・セッションを作成またはクローズできるため、ラウンドトリップは必要ありません。ユニバーサル接続プールは、すべてのユーザー/パスワード認証接続のキャッシュをサポートします。したがって、ユーザー認証のすべてのプロキシ接続はキャッシュおよび取得できます。

接続属性を適用せずにプロキシ接続をクローズすることはお薦めしません。接続属性を適用せずにプロキシ接続をクローズすると、その接続は再利用のために接続キャッシュに戻されますが取得できません。接続キャッシュ・メカニズムでは、セッション状態を記憶したりリセットしたりしません。

プロキシ接続は、直接クローズすることで、接続キャッシュから削除できます。

10.6 プロキシ接続の制限

プロキシ接続をクローズすると、プロキシ・セッション中またはプロキシ・セッションより前にプロキシ接続によって作成されたすべてのSQL文が自動的にクローズされます。これがアプリケーション・プーリングまたは文キャッシュに予期しない結果を招く可能性があります。次のサンプル・コードは、プロキシ接続のこの制限について説明しています。

例1

```
....
public void displayName(String N)  // Any function using the Proxy feature
{
     Properties props = new Properties();
     props.put("PROXY_USER_NAME", proxyUser);
     c.openProxySession(OracleConnection.PROXYTYPE_USER_NAME, props);
     .......
     c.close(OracleConnection.PROXY_SESSION);
}
```

```
public static void main (String args[]) throws SQLException
{
    ............
    PreparedStatement pstmt = conn.prepareStatement("SELECT first_name FROM EMPLOYEES WHERE 
employee_id = ?");
    pstmt.setInt(1, 205);
    ResultSet rs = pstmt.executeQuery();
    while (rs.next())
    {
        displayName(rs.getString(1));
         if (rs.isClosed() // The ResultSet is already closed while closing the connection!
         {
             throw new Exception("Your ResultSet has been prematurely closed! 
Your Statement object is also dead now.");
         }
    }
}
```

この例では、displayNameメソッドでプロキシ接続をクローズすると、PreparedStatementオブジェクトとResultSetオブジェクトもクローズされます。したがって、ループ内でResultSetオブジェクトのステータスをチェックしないと、そのループはnextメソッドが2度目にコールされたときに失敗します。

例2

```
    ....
    PreparedStatement pstmt = conn.prepareStatement("SELECT first_name FROM EMPLOYEES WHERE 
employee_id = ?");
    pstmt.setString(1, "205");
    ResultSet rs = pstmt.executeQuery();
    while (rs.next())
    {
        ....
    }

    Properties props = new Properties();
    props.put("PROXY_USER_NAME", proxyUser);

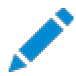
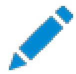
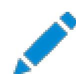
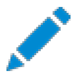
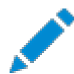
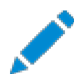
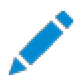
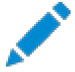
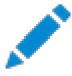
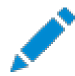
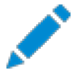
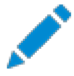
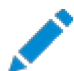
    conn.openProxySession(OracleConnection.PROXYTYPE_USER_NAME, props);
    .......
    conn.close(OracleConnection.PROXY_SESSION);

    // Try to use the PreparedStatement again
    pstmt.setString(1, "28960");
// This line of code will fail because the Statement is already closed while closing the connection!
    rs = pstmt.executeQuery();
```

この例では、PreparedStatementオブジェクトとResultSetオブジェクトは、プロキシ接続をオープンする前はうまく機能します。しかし、同じPreparedStatementオブジェクトをプロキシ接続のクローズ後に実行しようとすると、その文は失敗します。

第IV部 データへのアクセスと操作

この部では、Oracleデータへのアクセスと操作について説明します。また、ユーザー定義のオブジェクト型、ラージ・オブジェクト(LOB)とバイナリ・ファイル(BFILE)のロケータとデータ、オブジェクト参照およびOracleコレクション(ネストした表など)に対するJava Database Connectivity(JDBC)サポートについて説明します。さらに、JDBCでの結果セット機能、JDBC RowSet、およびJDBCドライバによって提供されるグローバリゼーション・サポートについても説明します。

第IV部の章の内容は次のとおりです。

- Oracleデータへのアクセスと操作
- JDBC内のJavaストリーム
- Oracleオブジェクト型の操作
- LOBとBFILEの操作
- Oracleオブジェクト参照の使用
- Oracleコレクションの操作
- 結果セット
- JDBC RowSet
- グローバリゼーション・サポート

11 Oracleデータへのアクセスと操作

この章では、Oracle拡張機能(oracle.sql.*形式)について説明し、標準Java形式(java.sql.*)と比較します。Oracle拡張機能を使用するには、結果セットと文を適宜OracleResultSet、OracleStatement、OraclePreparedStatementおよびOracleCallableStatementにキャストし、これらのクラスのgetOracleObject、setOracleObject、getXXXおよびsetXXXメソッド(XXXはoracle.sqlパッケージの型に対応)を使用する必要があります。

この章の内容は次のとおりです。

- データ型マッピング
- データ変換での考慮事項
- 結果セットと文拡張機能
- OracleのgetおよびsetメソッドとJDBCの比較
- 結果セット・メタデータ拡張機能の使用方法
- SQL CALL文とCALL INTO文の使用について

11.1 データ型マッピング

Oracle JDBCドライバは、Oracle固有のデータ型に加えて、標準JDBC型をサポートしています。この項では、標準およびOracle固有のSQL-Javaデフォルト型マッピングについて説明します。この項では、次の項目について説明します。

- マッピングの表
- マッピングに関するノート

11.1.1 マッピングの表

次の表は、SQLデータ型、JDBC型コード、標準Java型およびOracle拡張型間のデフォルト・マッピングを示しています。

SQLデータ型の列は、Oracle Database 12cリリース1 (12.1)に存在するSQL型を示しています。JDBC型コードの列は、JDBC標準によってサポートされ、java.sql.Typesクラス、またはoracle.jdbc.OracleTypesクラスでOracleによって定義されるデータの型コードを示しています。標準の型コードの場合、コードはこの2つのクラスで同一です。

標準Java型の列は、Java言語で定義される標準型を示しています。Oracle拡張機能Java型の列は、データベース内の各SQLデータ型に対応したoracle.sql.*Java型を示しています。これらは、oracle.sql.* Java型でSQLデータすべての取出しを可能にするOracle拡張機能です。



ノート:

一般に、Oracle JDBC ドライバは、標準 JDBC 型を使用して SQL データを操作するように最適化されています。いくつかの特殊な例では、oracle.sql パッケージで入手できる Oracle 拡張クラスを使用するのが有利な場合があります。しかし、できるかぎり Oracle 拡張機能ではなく標準 JDBC 型を使用することを強くお薦めします。

表11-1 SQL型とJava型間のデフォルト・マッピング

SQLデータ型	JDBC型コード	標準Java型	Ora
CHAR	java.sql.Types.CHAR	java.lang.String	orac
VARCHAR2	java.sql.Types.VARCHAR	java.lang.String	orac
LONG	java.sql.Types.LONGVARCHAR	java.lang.String	orac
NUMBER	java.sql.Types.NUMERIC	java.math.BigDecimal	orac
NUMBER	java.sql.Types.DECIMAL	java.math.BigDecimal	orac
NUMBER	java.sql.Types.BIT	boolean	orac
NUMBER	java.sql.Types.TINYINT	byte	orac
NUMBER	java.sql.Types.SMALLINT	short	orac
NUMBER	java.sql.Types.INTEGER	int	orac
NUMBER	java.sql.Types.BIGINT	long	orac
NUMBER	java.sql.Types.REAL	float	orac
NUMBER	java.sql.Types.FLOAT	double	orac
NUMBER	java.sql.Types.DOUBLE	double	orac
RAW	java.sql.Types.BINARY	byte[]	orac
RAW	java.sql.Types.VARBINARY	byte[]	orac
LONGRAW	java.sql.Types.LONGVARBINARY	byte[]	orac
DATE	java.sql.Types.DATE	java.sql.Date	orac
DATE	java.sql.Types.TIME	java.sql.Time	orac
TIMESTAMP	java.sql.Types.TIMESTAMP	javal.sql.Timestamp	orac
BLOB	java.sql.Types.BLOB	java.sql.Blob	orac

SQLデータ型	JDBC型コード	標準Java型	Orac
CLOB	java.sql.Types.CLOB	java.sql.Clob	orac
ユーザー定義オブジェクト	java.sql.Types.STRUCT	java.sql.Struct	orac
ユーザー定義参照	java.sql.Types.REF	java.sql.Ref	orac
ユーザー定義コレクション	java.sql.Types.ARRAY	java.sql.Array	orac
ROWID	java.sql.Types.ROWID	java.sql.RowId	orac
NCLOB	java.sql.Types.NCLOB	java.sql.NClob	orac
NCHAR	java.sql.Types.NCHAR	java.lang.String	orac
BFILE	oracle.jdbc.OracleTypes.BFILE (Oracle 拡張機能)	なし	orac
REF CURSOR	oracle.jdbc.OracleTypes.CURSOR (Oracle 拡張機能)	java.sql.ResultSet	orac
TIMESTAMP	oracle.jdbc.OracleTypes.TIMESTAMP (Oracle 拡張機能)	java.sql.Timestamp	orac
TIMESTAMPWITHTIMEZONE	oracle.jdbc.OracleTypes.TIMESTAMPTZ (Oracle 拡張機能)	java.sql.Timestamp	orac
TIMESTAMPWITHLOCALTIMEZONE	oracle.jdbc.OracleTypes.TIMESTAMPLTZ (Oracle 拡張機能)	java.sql.Timestamp	orac

脚注1

Oracle Database 12cリリース1 (12.1)以降、oracle.sql.BLOBクラスは非推奨となり、oracle.jdbc.OracleBlobインタフェースに置き換えられています。

脚注2

Oracle Database 12cリリース1以降、oracle.sql.CLOBクラスは非推奨となり、oracle.jdbc.OracleClobインタフェースに置き換えられています。

脚注3

Oracle Database 12cリリース1 (12.1)以降、oracle.sql.STRUCTクラスは非推奨となり、oracle.jdbc.OracleStructインタフェースに置き換えられています。

脚注4

Oracle Database 12cリリース1 (12.1)以降、oracle.sql.REFクラスは非推奨となり、oracle.jdbc.OracleRefインタフェースに置き換えられています。

脚注5

Oracle Database 12cリリース1 (12.1)以降、oracle.sql.ARRAYクラスは非推奨となり、oracle.jdbc.OracleArrayインタフェースに置き換えられています。



ノート:

バージョン 8.1.7 など、TIMESTAMP データ型をサポートしていないデータベースのバージョンの場合、TIMESTAMP は DATE にマップされます。

関連トピック

- 標準型とOracle型
- サポートされているSQLとJDBCデータ型のマッピング

11.1.2 マッピングに関するノート

この項では、NUMBERおよびユーザー定義型のマッピングに関する詳細を提供します。

NUMBER型

Oracle NUMBER値が対応できる様々な型コードでは、マッピングを正しく動作させるために、データのサイズに適切なgetterルーチンをコールします。たとえば、-128 < x < 128の項目xに対してJava tinyint値を取得するには、getByteをコールします。

ユーザー定義型

オブジェクト、オブジェクト参照およびコレクションなどのユーザー定義型は、デフォルトでjava.sql.Structなどの弱いJava型にマップされますが、かわりに強い型指定のカスタムJavaクラスにもマップできます。カスタムJavaクラスは、次のいずれかのインタフェースを実装できます。

- 標準java.sql.SQLData
- Oracle固有oracle.jdbc.OracleData

関連トピック

- Oracleオブジェクトのマッピングについて
- Oracleオブジェクト用のカスタム・オブジェクト・クラスの作成と使用方法について

11.2 データ変換での考慮事項

JDBCプログラムでSQLデータをJavaに取り込む場合は、標準Java型またはoracle.sqlパッケージの型を使用できます。この項の内容は次のとおりです。

- 標準型とOracle型
- SQL NULLデータの変換について
- NULLのテストについて

11.2.1 標準型とOracle型

oracle.sqlのOracleデータ型では、データベースで使用されているのと同じビット形式でデータが格納されます。Oracle Database 10gより前のバージョンのOracle JDBCドライバでは、通常、Oracleデータ型の方が効率的でした。Oracle Database 10g以降、JDBCドライバは大幅に更新されました。その結果、ほとんどの場合は、oracle.sqlのデータ型よりも標準Java型が優先されるようになりました。特に、java.lang.Stringは、oracle.sql.CHARよりはるかに効率的です。

一般的にはJava標準型の使用をお薦めします。例外として次のような場合があります。

- OracleDataの機能の方がニーズに合っている場合は、java.sql.SqlDataではなく、oracle.jdbc.OracleDataを使用します。
- 浮動小数点数の値を正確に保持する必要がある場合は、java.lang.Doubleではなくoracle.sql.NUMBERを使用します。Oracle NUMBERは10進表現ですが、Java DoubleおよびFloatは2進表現となります。ある形式から別の形式へ変換を行うと、実際の表現値が少し変化することがあります。さらに、2つの形式を使用して表現できる値の範囲が異なります。

  パフォーマンスが重大で、値を操作せずに、単に値の読取りや書込みを行う場合には、java.math.BigDecimalではなく、oracle.sql.NUMBERを使用します。
- JDK 6より前のJDKバージョンを使用している場合は、oracle.sql.DATEまたはoracle.sql.TIMESTAMPを使用します。JDK 6以降のバージョンを使用している場合、java.sql.Dateまたはjava.sql.Timestampを使用します。



ノート:

JDK 6 より前のすべてのバージョンでは、不具合により java.lang.Date および java.lang.Timestamp オブジェクトの作成処理が遅くなるという問題があります(特に、マルチスレッド環境において)。この不具合は、JDK 6 で修正されました。

- Oracle文字セットによるコードで表されている外部ソースのデータがある場合にかぎり、oracle.sql.CHARを使用します。それ以外の場合はすべて、java.lang.Stringを使用します。
- STRUCT、ARRAY、BLOB、CLOB、REFおよびROWIDはすべて、対応するJDBC標準インタフェース型の実装クラスです。したがって、JDBC標準型と同一のOracle拡張型を使用する利点はありません。
- BFILE、TIMESTAMPTZおよびTIMESTAMPLTZには、JDBC標準での表現がありません。これらのOracle拡張機能は使用する必要があります。
- 他のすべての場合には、Oracle拡張機能ではなく、標準JDBC型を使用する必要があります。



ノート:

oracle.sql データ型を Java 標準データ型に変換した場合、oracle.sql データ型を使用する利点は失われます。

11.2.2 SQL NULLデータの変換について

Javaは、SQL NULLデータを、Javaの値nullで表現します。Javaデータ型は、プリミティブ型(byte、int、floatなど)とオブジェクト型(クラス・インスタンスなど)の2つのカテゴリに分類されます。プリミティブ型でNULLを表すことはできません。かわりに、JDBC仕様で定義されているように、nullを0(ゼロ)値として格納します。これは、結果の解釈時にあいまいさが生じる原因ともなります。

一方、Javaオブジェクト型はNULLを表現できます。Java言語は、nullを表現可能なプリミティブ型ごとに対応したオブジェクト・コンテナ型を定義します。オブジェクト・コンテナ型は、SQLデータがSQL NULLを明確に検出するためのターゲットとして使用する必要があります。

11.2.3 NULLのテストについて

関係演算子を使用してNULL値同士、または他の値との比較をすることはできません。たとえば、次のSELECT文は、COMMISSION_PCT列にNULL値が1つ以上存在する場合でも行を戻しません。

```
PreparedStatement pstmt = conn.prepareStatement(
  "SELECT * FROM EMPLOYEES WHERE COMMISSION_PCT = ?");
pstmt.setNull(1, java.sql.Types.VARCHAR);
```

次の例では、戻り値にNULLが含まれる可能性がある場合に値が等しいかどうかを比較する方法を示します。このコードは、COMMに値205が存在しない場合、EMPLOYEES表からFIRST_NAMEをNULLとして戻します。

```
PreparedStatement pstmt = conn.prepareStatement("SELECT FIRST_NAME FROM EMPLOYEES
  WHERE COMMISSION_PCT =? OR  ((COMM IS NULL) AND (? IS NULL))");
pstmt.setBigDecimal(1, new BigDecimal(205));
pstmt.setNull(2, java.sql.Types.VARCHAR);
```

11.3 結果セットと文拡張機能

Statementオブジェクトは、java.sql.ResultSetを戻します。標準JDBCメソッドのみをオブジェクトに適用する場合は、オブジェクトのResultSet型を維持してください。ただし、オブジェクト上でOracle拡張機能を使用する場合は、Oracle拡張機能をOracleResultSetにキャストする必要があります。Oracleの結果セット拡張機能は、すべてoracle.jdbc.OracleResultSetインタフェースに含まれ、Statement拡張機能は、すべてoracle.jdbc.OracleStatementインタフェースに含まれています。

たとえば、標準Statementオブジェクトstmtがあり、標準JDBC ResultSetメソッドのみを使用する場合、次のように記述します。

```
ResultSet rs = stmt.executeQuery("SELECT * FROM employees");
```

JDBCに対するOracle拡張機能を使用する場合は、結果を選択して標準ResultSet変数に入れ、次にその変数をOracleResultSetにキャストします。

結果セット・クラスおよび文クラスへの主な拡張機能には、getOracleObjectおよびsetOracleObjectメソッドがあります。これらのメソッドは、oracle.sql.*形式でデータへのアクセスおよび操作を実行するために使用します。

11.4 OracleのgetおよびsetメソッドとJDBC標準の比較

この項では、getメソッドとsetメソッド、特にJDBC標準のgetObjectメソッドとsetObjectメソッドおよびOracle固有のgetOracleObjectメソッドとsetOracleObjectメソッドについて説明します。また、oracle.sql.*形式のデータへのアクセス方法をJava形式と比較しながら説明します。

すべてのOracle SQL型に標準のgetXXXメソッドを使用できます。

この項の内容は次のとおりです。

- 標準getObjectメソッド
- Oracle getOracleObjectメソッド
- getObjectおよびgetOracleObject戻り型のまとめ
- その他のgetXXXメソッド
- getObjectおよびgetXXXから戻されるオブジェクトのデータ型
- setObjectメソッドとsetOracleObjectメソッド
- その他のsetXXXメソッド



ノート:

表名を使用して列名を修飾し、その列名をパラメータとして getXXX メソッドに渡すことはできません。たとえば:

```
ResultSet rset = stmt.executeQuery("SELECT employees.department_id, department.department_id FROM emp
rset.getInt("employees.department_id");
```

このコード例の場合、getInt メソッドで例外が発生します。getXXX メソッドで列を一意に識別するには、列索引を使用するた
メソッドで使用します。

11.4.1 標準getObjectメソッド

結果セットまたはコール可能文の標準getObjectメソッドの戻り型は、java.lang.Objectです。戻されるオブジェクトのクラスは、次に示すようにSQL型に基づきます。

- Oracle固有ではないSQLデータ型の場合、JDBC仕様のマッピングに従い、getObjectメソッドは列のSQL型に対応するデフォルトのJava型を戻します。
- Oracle固有のデータ型の場合、getObjectは適切なoracle.sql.*クラス(oracle.sql.ROWIDなど)のオブジェクトを戻します。

- Oracle Databaseオブジェクトの場合、getObjectは、使用する型マップで指定されたクラスのJavaオブジェクトを戻します。型マップは、データベース型名からJavaクラスへのマッピングを指定します。getObject(parameter_index)メソッドでは、接続のデフォルト型マップを使用します。getObject(parameter_index, map)を使用すると、型マップを渡すことができます。型マップに特定のOracleオブジェクトに対応するマッピングがない場合、getObjectはoracle.sql.OracleStructオブジェクトを戻します。

11.4.2 Oracle getOracleObjectメソッド

結果セットまたはコール可能文からデータをoracle.sql.*オブジェクトとして取り出す場合は、特殊なプロセスに従う必要があります。OracleResultSetオブジェクトの場合、結果セットをoracle.jdbc.OracleResultSetにキャストしてから、getObjectではなく、getOracleObjectをコールする必要があります。CallableStatementおよびoracle.jdbc.OracleCallableStatementにも同じことが当てはまります。

getOracleObjectの戻り型はoracle.sql.Datumです。実際に戻されるオブジェクトは、該当するoracle.sql.*クラスのインスタンスです。メソッド・シグネチャは次のとおりです。

```
public oracle.sql.Datum getOracleObject(int parameter_index)
```

Datum変数に取り出したデータを入れると、標準Java instanceof演算子を使用して、実際にどのoracle.sql.*型であるかを確認できます。

例: 結果セットでのgetOracleObjectの使用方法

次の例では、CHARデータの列を含む表およびBFILEロケータを含む列を作成します。SELECT文では、表の内容を結果セットとして取り出します。その後、getOracleObjectで取り出したCHARデータをchar_datum変数に、BFILEロケータをbfile_datum変数に入れます。getOracleObjectはDatumオブジェクトを戻すため、戻り値をそれぞれCHARおよびBFILEにキャストする必要があります。

```
stmt.execute ("CREATE TABLE bfile_table (x VARCHAR2 (30), b BFILE)");
stmt.execute
    ("INSERT INTO bfile_table VALUES ('one', BFILENAME ('TEST_DIR', 'file1'))");
ResultSet rset = stmt.executeQuery ("SELECT * FROM bfile_table");
while (rset.next ())
{
   CHAR char_datum = (CHAR) ((OracleResultSet)rset).getOracleObject (1);
   BFILE bfile_datum = (BFILE) ((OracleResultSet)rset).getOracleObject (2);
   ...
}
```

例: コール可能文でのgetOracleObjectの使用方法

次の例では、文字列を日付と関連付けるプロシージャmyGetDateへのコールを作成します。このプログラムは、HRを準備済コールに渡し、DATE型を出力パラメータとして登録します。コールの実行後、getOracleObjectはHRに関連付けられている日付を取り出します。getOracleObjectはDatumオブジェクトを戻すため、結果はDATEにキャストされる点に留意してください。

```
OracleCallableStatement cstmt = (OracleCallableStatement)conn.prepareCall
                                   ("begin myGetDate (?, ?); end;");
cstmt.setString (1, "HR");
cstmt.registerOutParameter (2, Types.DATE);
cstmt.execute ();
```

```
DATE date = (DATE) ((OracleCallableStatement)cstmt).getOracleObject (2);
...
```

11.4.3 getObjectおよびgetOracleObject戻り型のまとめ

次の表は、各Oracle SQL型のgetObjectメソッドとgetOracleObjectメソッドの、基礎となる戻り型を示しています。

ただし、これらのメソッドを使用する際は、次のことを念頭に置いてください。

- getObjectは、データを常にjava.lang.Objectインスタンスで戻します。
- getOracleObjectは、データを常にoracle.sql.Datumインスタンスで戻します。

特殊機能を使用するには、戻されたオブジェクトをキャストする必要があります。

表11-2 getObjectとgetOracleObjectの戻り型

Oracle SQL型	getObjectの基礎となる戻り型	getOracleObjectの基礎となる戻り型
CHAR	String	oracle.sql.CHAR
VARCHAR2	String	oracle.sql.CHAR
NCHAR	String	oracle.sql.CHAR
LONG	String	oracle.sql.CHAR
NUMBER	java.math.BigDecimal	oracle.sql.NUMBER
RAW	byte[]	oracle.sql.RAW
LONGRAW	byte[]	oracle.sql.RAW
DATE	java.sql.Date	oracle.sql.DATE
TIMESTAMP	java.sql.Timestamp 脚注6	oracle.sql.TIMESTAMP
TIMESTAMPWITHTIMEZONE	oracle.sql.TIMESTAMPTZ	oracle.sql.TIMESTAMPTZ
TIMESTAMPWITHLOCALTIMEZONE	oracle.sql.TIMESTAMPLTZ	oracle.sql.TIMESTAMPLTZ
BINARY_FLOAT	java.lang.Float	oracle.sql.BINARY_FLOAT
BINARY_DOUBLE	java.lang.Double	oracle.sql.BINARY_DOUBLE
INTERVALDAYTOSECOND	oracle.sql.INTERVALDS	oracle.sql.INTERVALDS

Oracle SQL型	getObjectの基礎となる戻り型	getOracleObjectの基礎となる戻り型
INTERVALYEARTOMONTH	oracle.sql.INTERVALYM	oracle.sql.INTERVALYM
ROWID	oracle.sql.ROWID	oracle.sql.ROWID
REF CURSOR	java.sql.ResultSet	(未サポート)
BLOB	oracle.jdbc.OracleBlob 脚注7	oracle.jdbc.OracleBlob
CLOB	oracle.jdbc.OracleClob 脚注8	oracle.jdbc.OracleClob
NCLOB	java.sql.NClob	oracle.sql.NCLOB
BFILE	oracle.sql.BFILE	oracle.sql.BFILE
Oracle オブジェクト	型マップで指定されたクラス または oracle.sql.OracleStruct 脚注9 (型マップにエントリがない場合)	oracle.jdbc.OracleStruct
Oracle オブジェクト参照	oracle.jdbc.OracleRef 脚注10	oracle.jdbc.OracleRef
コレクション(VARRAY または NESTED TABLE)	oracle.jdbc.OracleArray 脚注11	oracle.sql.ARRAY

脚注6

ResultSet.getObjectは、oracle.jdbc.J2EE13Compliant接続プロパティがTRUEに設定されている場合のみjava.sql.Timestampを戻します。そうでない場合、このメソッドはoracle.sql.TIMESTAMPを戻します。

脚注7

Oracle Database 12cリリース1 (12.1)以降、oracle.sql.BLOBクラスは非推奨となり、oracle.jdbc.OracleBlobインタフェースに置き換えられています。

脚注8

Oracle Database 12cリリース1 (12.1)以降、oracle.sql.CLOBクラスは非推奨となり、oracle.jdbc.OracleClobインタフェースに置き換えられています。

脚注9

Oracle Database 12cリリース1 (12.1)以降、oracle.sql.STRUCTクラスは非推奨となり、

oracle.jdbc.OracleStructインタフェースに置き換えられています。

脚注10

Oracle Database 12cリリース1以降、oracle.sql.REFクラスは非推奨となり、oracle.jdbc.OracleRefインタフェースに置き換えられています。

脚注11

Oracle Database 12cリリース1以降、oracle.sql.ARRAYクラスは非推奨となり、oracle.jdbc.OracleArrayインタフェースに置き換えられています。

> ノート:
>
> ResultSet.getObject メソッドは、接続プロパティ oracle.jdbc.J2EE13Compliant が TRUE に設定されている場合のみ、TIMESTAMP SQL 型の java.sql.Timestamp を戻します。このプロパティは、接続の取得時に設定する必要があります。この接続プロパティが設定されていない場合、または接続の取得後に設定した場合、ResultSet.getObject メソッドは TIMESTAMP SQL 型の oracle.sql.TIMESTAMP を戻します。
>
> oracle.jdbc.J2EE13Compliant 接続プロパティは、コードを変更せずに、次の方法で設定できます。
>
> 
>
> - ojdbc6dms.jar ファイルまたは ojdbc7dms.jar ファイルを CLASSPATH に含めます。これらのファイルでは、デフォルトで oracle.jdbc.J2EE13Compliant が TRUE に設定されます。これらは Oracle Application Server リリースに固有のもので、一般の JDBC リリースには含まれていません。これらのファイルは $ORACLE_HOME/jdbc/lib にあります。
>
> - java コマンドをフラグ-Doracle.jdbc.J2EE13Compliant=true を指定してコールし、システム・プロパティを設定します。たとえば、
>
> ```
> java -Doracle.jdbc.J2EE13Compliant=true ...
> ```
>
> J2EE13Compliant が TRUE に設定されると、動作は JDBC 仕様の表 B-3 のようになります。

関連トピック

- サポートされているSQLとJDBCデータ型のマッピング

11.4.4 その他のgetXXXメソッド

標準JDBCでは、各標準Java型に対応したgetXXX(getByte、getInt、getFloatなど)が提供されます。これらの各メソッドは、そのメソッド名が意味する内容を正確に戻します。

さらに、OracleResultSetクラスとOracleCallableStatementインタフェースは、すべてのoracle.sql.*型に対応したgetXXXメソッドを補完します。各getXXXメソッドは、oracle.sql.XXXオブジェクトを戻します。たとえば、getROWIDメソッドは、oracle.sql.ROWIDオブジェクトを戻します。

特定のgetXXXメソッドを使用してもパフォーマンス上の利点はありません。ただし、戻り型は戻されるオブジェクトに固有であるた

め、面倒なキャストの必要がなくなります。

この項の内容は次のとおりです。

- getXXXメソッドの戻り型
- getXXXメソッドに関する特別なノート

11.4.4.1 getXXXメソッドの戻り型

各々のgetXXXメソッドの戻り型や、Java Development Kit (JDK) 6に含まれるOracle拡張機能については、JDBC Javadocを参照してください。戻されたオブジェクトをOracleResultSetまたはOracleCallableStatementにキャストして、Oracle拡張機能であるメソッドを使用する必要があります。

11.4.4.2 getXXXメソッドに関する特別なノート

この項では、いくつかのgetXXXメソッドについて追加の詳細情報を示します。

getBigDecimal

JDBC 2.0では、getBigDecimalメソッドのメソッド・シグネチャが単純になっています。以前の入力シグネチャは次のいずれかでした。

```
(int columnIndex, int scale) or (String columnName, int scale)
```

単純になった入力シグネチャは次のとおりです。

```
(int columnIndex) or (String columnName)
```

小数点の右にある数字の数を指定するために使用するscaleパラメータは不要になりました。Oracle JDBCドライバは、完全な精度で数値を取り出します。

getBoolean

BOOLEANデータベース型が存在しないため、getBooleanを使用すると必ずデータ型変換が実行されます。getBooleanメソッドは数値用の列に対してのみサポートされます。このような列に対してgetBooleanが適用されると、0(ゼロ)値はfalseとして、それ以外の値はtrueとして解釈されます。別の種類の列に適用された場合は、getBooleanは例外java.lang.NumberFormatExceptionを戻します。

11.4.5 getObjectおよびgetXXXから戻されるオブジェクトのデータ型

getObjectの戻り型はjava.lang.Objectです。戻り型は、java.lang.Objectのサブクラスのインスタンスです。同様に、getOracleObjectの戻り型はoracle.sql.Datumで、戻り値のクラスはoracle.sql.Datumのサブクラスです。通常、適切なクラスの特定のメソッドおよび機能を使用するには、戻されたオブジェクトをそのクラスにキャストします。

また、汎用のgetObjectメソッドやgetOracleObjectメソッドのかわりに、特定のgetXXXメソッドも使用できます。getXXXの戻り型は、戻されるオブジェクトの型に対応しているため、getXXXメソッドを使用することにより、キャストを回避できます。たとえば、getCLOBの戻り型は、java.lang.Objectではなくoracle.sql.CLOBです。

戻り値のキャストの例

この例では、NUMBERタイプのデータを結果セットの最初の列としてフェッチしていると想定しています。精度を下げずにNUMBERデータを操作する必要があるので、OracleResultSetに結果セットをキャストし、getOracleObjectを使用して、NUMBERデータをoracle.sql.*形式で戻します。結果セットをキャストしない場合は、getObjectを使用する必要があります。数値データがJava Floatに戻されるため、SQLデータの精度が多少下がります。

getOracleObjectメソッドは、出力をキャストしないかぎり、oracle.sql.NUMBERオブジェクトをoracle.sql.Datum戻り変数に戻します。NUMBER戻り変数を使用し、そのクラスの特殊機能のいずれかを使用する場合は、getOracleObjectの出力をoracle.sql.NUMBERにキャストしてください。

```
NUMBER x = (NUMBER)ors.getOracleObject(1);
```

11.4.6 setObjectメソッドとsetOracleObjectメソッド

結果セットとコール可能文に標準getObjectとOracle固有のgetOracleObjectがあるように、OraclePreparedStatementとOracleCallableStatementにも標準setObjectメソッドとOracle固有のsetOracleObjectメソッドがあります。setOracleObjectメソッドは、oracle.sql.*を入力パラメータとして取ります。

標準Java型をプリペアド文またはコール可能文にバインドするには、java.lang.Objectを入力にとるsetObjectメソッドを使用します。setObjectメソッドは、いくつかのoracle.sql.*型をサポートします。ただしこのメソッドは、JDBC標準型Blob、Clob、Struct、RefおよびArrayに対応するoracle.sql.*クラスのインスタンスを入力できるように実装されています。

oracle.sql.*型をプリペアド文またはコール可能文にバインドするには、oracle.sql.Datumのサブクラスを入力にとるsetOracleObjectメソッドを使用します。setOracleObjectを使用するには、プリペアド文またはコール可能文をOraclePreparedStatementまたはOracleCallableStatementにキャストする必要があります。

setObjectとsetOracleObjectの使用例

プリペアド文には、charVal変数により表現されたoracle.sql.CHARデータがsetOracleObjectメソッドによってバインドされます。oracle.sql.*データをバインドするには、プリペアド文をOraclePreparedStatementにキャストする必要があります。同様に、setObjectメソッドは、変数strValで表現されたJava Stringデータをバインドします。

```
PreparedStatement ps= conn.prepareStatement("text_of_prepared_statement");
((OraclePreparedStatement)ps).setOracleObject(1,charVal);
ps.setObject(2,strVal);
```

11.4.7 その他のsetXXXメソッド

getXXXメソッドと同様に、固有なsetXXXメソッドもいくつかあります。標準のsetXXXメソッドはJava型をバインドするために、Oracle固有のsetXXXメソッドはOracle固有の型をバインドするために提供されています。

同様に、setNullメソッドには次の2つの形式があります。

- void setNull(int parameterIndex, int sqlType)

  これは、標準のjava.sql.PreparedStatementインタフェースで指定されるシグネチャです。このシグネチャは、java.sql.Typesクラスまたはoracle.jdbc.OracleTypesクラスによって定義されるパラメータ索引とSQL型コードを取ります。REF、ARRAYまたはSTRUCT以外のオブジェクトをNULLに設定する場合は、このシグネチャを使用します。

- void setNull(int parameterIndex, int sqlType, String sql_type_name)

  JDBC 2.0の場合、このシグネチャは標準のjava.sql.PreparedStatementインタフェースにも指定されます。このメソッドは、パラメータ索引およびSQL型コードに加え、SQL型名を取ります。このメソッドは、SQL型コードがjava.sql.Types.REF、ARRAYまたはSTRUCTの場合に使用します。型コードがREF、ARRAYまたはSTRUCT以外の場合、指定されたSQL型名は無視されます。

同様に、registerOutParameterメソッドには、REF、ARRAYまたはSTRUCTデータとともに使用するためのシグネチャがあります。

```
void registerOutParameter
          (int parameterIndex, int sqlType, String sql_type_name)
```

標準Java型のバインドのためのメソッドではなく、適切なsetXXXメソッドを使用してOracle固有型をバインドすると、多少パフォーマンスが向上することがあります。

この項の内容は次のとおりです。

- 入力データのバインド
- WHERE句にCHARデータをバインドするためのメソッドsetFixedCHAR

11.4.7.1 入力データのバインド

入力用のデータのバインド方法は3つあります。

- データ自体をバインド・バッファに格納するダイレクト・バインド
- データをストリーム処理するストリーム・バインド
- 一時LOBを作成し、LOB APIを使用してデータをそのLOBに格納し、LOBロケータのバイトをバインド・バッファに格納するLOBバインド

3種類のバインドはパフォーマンスが異なり、バッチ処理に影響を与えます。ダイレクト・バインドは最も高速で、バッチ処理が正常に行われます。ストリーム・バインドはより低速で、複数のラウンドトリップが必要な場合があり、バッチ処理はオフになります。LOBバインドは非常に低速で、多数のラウンドトリップが必要です。バッチ処理は機能しますが、お薦めできません。これらのバインドでは、SQL文の型に応じてサイズの制限も異なります。

SQLパラメータについては、RAWやVARCHAR2など標準のパラメータ型の長さはターゲット列のサイズによって固定されます。PL/SQLパラメータについては、サイズは32766という固定バイト数に制限されます。

Oracle Database 10gリリース2では、PreparedStatementのsetString、setCharacterStream、setAsciiStream、setBytesおよびsetBinaryStreamメソッドが一部変更されています。APIの当初の動作は次のとおりでした。

- setString: キャラクタのダイレクト・バインド
- setCharacterStream: キャラクタのストリーム・バインド
- setAsciiStream: バイトのストリーム・バインド
- setBytes: バイトのダイレクト・バインド
- setBinaryStream: バイトのストリーム・バインド

Oracle Database 10gリリース2以降、データ・サイズおよびSQL文の型に基づくバインド・モードの自動切替えがサポートされています。

setBytesおよびsetBinaryStream

SQLでは、サイズが2000以下の場合はダイレクト・バインド、2000を超える場合はストリーム・バインドが使用されます。

PL/SQLでは、サイズが32766以下の場合はダイレクト・バインド、32766を超える場合はLOBバインドが使用されます。

setString、setCharacterStreamおよびsetAsciiStream

SQLでは、Javaキャラクタが32766以下の場合はダイレクト・バインド、32766を超える場合はストリーム・バインドが使用されます。これは文字セットに依存しません。

PL/SQLでは、使用形式パラメータの設定に応じて、データベース文字セットまたは各国語文字セット内のキャラクタ・データのバイト・サイズに注意する必要があります。バイト長が32766未満のデータにはダイレクト・バインド、32766以上のデータにはLOBバインドが使用されます。

固定長文字セットについては、Javaキャラクタ・データの長さにバイト単位の固定文字サイズをかけた値を上限値と比較します。可変長文字セットについては、Javaキャラクタの長さに応じて、次の3つの場合があります。

- 文字長が32766を最大文字サイズで割った値よりも小さい場合は、ダイレクト・バインドが使用されます。
- 文字長が32766を最小文字サイズで割った値よりも大きい場合は、LOBバインドが使用されます。
- 文字長がその間であり、変換済バイトの実際の長さが32766未満の場合は、ダイレクト・バインドが使用され、それ以外の場合はLOBバインドが使用されます。



ノート:

PL/SQL プロシージャが SQL 文に埋め込まれている場合、バインド処理は異なります。

サーバー側内部ドライバには、他にも次の制限事項があります。

- データ・サイズの文字数が32767バイトを超える場合、setString、setCharacterStreamおよびsetASCIIStreamの各APIはSQL CLOB列ではサポートされていません。
- データ・サイズが32767バイトを超える場合、setBytesおよびsetBinaryStreamの各APIはSQL BLOB列ではサポートされていません。



ノート:

サーバー側内部ドライバでこれらの API を使用する場合は、クライアント・コードのデータ・サイズを入念にチェックしてください。

関連トピック

- LOBのデータ・インタフェース

関連項目:

詳細な説明および考えられる対応策は、JDBCのリリース・ノートを参照してください。

11.4.7.2 WHERE句にCHARデータをバインドするためのメソッドsetFixedCHAR

データベース内のCHARデータは、列幅まで埋め込まれます。このため、SELECT文のWHERE句に文字データをバインドするためのsetCHARメソッドの使用に関して、制限が生じます。WHERE句の文字データも、SELECT文で合致させるために、列幅まで埋め込む必要があります。これは特に列幅がわからない場合に問題になります。

これを修正するために、OracleはOraclePreparedStatementクラスにsetFixedCHARメソッドを追加しました。このメソッドは埋込みなしの比較を実行します。



ノート:

- setFixedCHAR メソッドを使用するには、必ずプリペアド文オブジェクトを OraclePreparedStatement にキャストしてください。
- INSERT 文で、setFixedCHAR を使用する必要はありません。データベースは挿入時に、常にそのデータを列幅まで自動的に埋め込みます。

例

次の例では、setCHARメソッドとsetFixedCHARメソッドの違いを示します。

```
/* Schema is :
 create table my_table (col1 char(10));
 insert into my_table values ('JDBC');
*/
 PreparedStatement pstmt = conn.prepareStatement
                    ("select count(*) from my_table where col1 = ?");
 pstmt.setString (1, "JDBC");  // Set the Bind Value
 runQuery (pstmt);             // This will print " No of rows are 0"
 CHAR ch = new CHAR("JDBC      ", null);
 ((OraclePreparedStatement)pstmt).setCHAR(1, ch); // Pad it to 10 bytes
 runQuery (pstmt);             // This will print "No of rows are 1"
 ((OraclePreparedStatement)pstmt).setFixedCHAR(1, "JDBC");
  runQuery (pstmt);            // This will print "No of rows are 1"

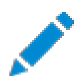
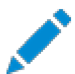
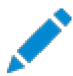
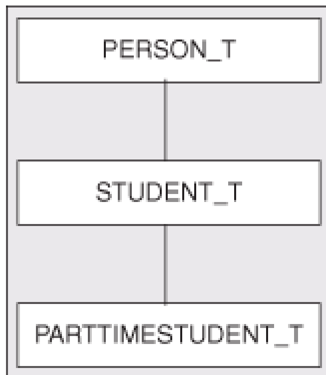
 void runQuery (PreparedStatement ps)
 {
   // Run the Query
   ResultSet rs = pstmt.executeQuery ();
   while (rs.next())
     System.out.println("No of rows are " + rs.getInt(1));

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

11.5 結果セット・メタデータ拡張機能の使用方法

oracle.jdbc.OracleResultSetMetaDataインタフェースはJDBC 2.0に準拠していますが、Oracle Databaseによってサ

ポートされていないため、getSchemaNameメソッドとgetTableNameメソッドは実装されません。

次のコードは、OracleResultSetMetadataインタフェースのメソッドをいくつか使用して、EMPLOYEES表から列数を取り出し、各列の数値型とSQL型名を取り出します。

```
DatabaseMetaData dbmd = conn.getMetaData();
ResultSet rset = dbmd.getTables("", "HR", "EMPLOYEES", null);
 while (rset.next())
 {
   OracleResultSetMetaData orsmd = ((OracleResultSet)rset).getMetaData();
   int numColumns = orsmd.getColumnCount();
   System.out.println("Num of columns = " + numColumns);
   for (int i=0; i<numColumns; i++)
   {
     System.out.print ("Column Name=" + orsmd.getColumnName (i+1));
     System.out.print (" Type=" + orsmd.getColumnType (i + 1) );
     System.out.println (" Type Name=" + orsmd.getColumnTypeName (i + 1));
  }
}
```

このプログラムは次の出力を戻します。

```
Num of columns = 5
Column Name=TABLE_CAT Type=12 Type Name=VARCHAR2
Column Name=TABLE_SCHEM Type=12 Type Name=VARCHAR2
Column Name=TABLE_NAME Type=12 Type Name=VARCHAR2
Column Name=TABLE_TYPE Type=12 Type Name=VARCHAR2
Column Name=TABLE_REMARKS Type=12 Type Name=VARCHAR2
```

11.6 SQL CALL文とCALL INTO文の使用について

次の2つの方法で、CALL文を使用してSQL内からルーチンを実行できます。



ノート:

ルーチンとは、スタンドアロンのプロシージャまたは関数や、型またはパッケージ内で定義されているプロシージャまたは関数です。スタンドアロンのルーチン上、またはルーチンが定義されている型やパッケージ上で、EXECUTE 権限を持っている必要があります。

- コールをルーチン自身に名前で送信するか、routine_clauseを使用します。
- 式の型の内部でobject_access_expressionを使用します。

ルーチンが引数を取る場合は、引数を指定できます。引数には、位置表記、名前表記または混合表記を使用できます。

CALL INTO文

INTO句は関数のコールにのみ適用されます。この句には次の型の変数を使用できます。

- ホスト変数
- 標識変数

関連項目:

詳細は、『Oracle Database SQL言語リファレンス』を参照してください

PL/SQLブロック

PL/SQLの基本単位は、ブロックです。すべてのPL/SQLプログラムはブロックで構成されており、ブロックは互いの内部にネストすることができます。PL/SQLブロックには、宣言部、実行部、例外処理部の3つの部分があります。アプリケーションでPL/SQLブロックを使用すると、次の利点があります。

- パフォーマンスの向上
- 生産性の向上
- 完全な移植性
- Oracleとの密接な統合
- 厳重なセキュリティ

12 JDBC内のJavaストリーム

この章では、Oracle Java Database Connectivity(JDBC)ドライバで様々なデータ型のJavaストリームを処理する方法を説明します。データ・ストリームを使用することにより、最大2GBのLONG型列データを読み取れます。

この章の構成は、次のとおりです。

- Javaストリームの概要
- LONGまたはLONG RAW列のストリームについて
- CHAR、VARCHARまたはRAW列のストリームについて
- LOBおよび外部ファイルのストリームについて
- データ・ストリームと複数列の関係
- ストリームと行のプリフェッチの関係
- ストリームのクローズ
- ストリームに関するノート

12.1 Javaストリームの概要

Oracle JDBCドライバは、サーバーとクライアント間の双方向のデータ・ストリーム操作をサポートします。ドライバは、すべてのストリーム変換、つまりバイナリ、ASCIIおよびUnicodeをサポートします。次に、ストリームの各タイプを簡単に説明します。

- バイナリ

  データのRAWバイトのために使用されます。getBinaryStreamメソッドに対応します。

- ASCII

  ISO-Latin-1エンコーディングでASCIIバイトのために使用されます。getAsciiStreamメソッドに対応します。

- Unicode

  UTF-16エンコーディングでUnicodeバイトのために使用されます。getUnicodeStreamメソッドに対応します。

getBinaryStream、getAsciiStreamおよびgetUnicodeStreamの各メソッドは、InputStreamオブジェクト内のデータ・バイトを戻します。



ノート:

Oracle Database 12c リリース 1 (12.1)以降、CONNECTION_PROPERTY_STREAM_CHUNK_SIZE は非推奨になり、ドライバは内部でこれを使用してストリーム・チャンク・サイズを設定しません。

関連項目:

LOBとBFILEの操作

12.2 LONGまたはLONG RAW列のストリームについて

この項の内容は次のとおりです。

- LONGまたはLONG RAW列のストリームの概要
- LONG RAWデータの変換
- LONGデータの変換
- 例: LONG RAWデータのストリーム
- LONGまたはLONG RAWのストリーム回避について

12.2.1 LONGまたはLONG RAW列のストリームの概要

問合せがLONGまたはLONG RAW列を1つ以上選択すると、JDBCドライバは、ストリーム・モードでこれらの列をクライアントに転送します。ストリーム・モードでは、JDBCドライバは、必要になるまでLONGまたはLONG RAW列のためにネットワークから列データを読み取りません。コードでgetXXXメソッドを呼び出して列データを読み取るまで、列データはネットワーク通信チャネルに残っています。コールの後でも、列データは、getXXXコールから戻り値にデータを移入する場合にのみ読み取られます。列データが通信チャネルに残っているため、ストリーム・モードは他のすべての接続の使用を妨げます。列データの読取り以外で接続を使用すると、チャネルから列データが廃棄されます。ストリーム・モードはメモリーを効率よく使用し、ネットワークのラウンドトリップを最小限に抑えますが、他の多くのデータベース操作の妨げになります。

ノート:

LONG および LONG RAW 列は使用しないことをお薦めします。かわりに LOB を使用してください。

LONG列のデータにアクセスするには、この列をJava InputStreamオブジェクトとして取得し、InputStreamオブジェクトのreadメソッドを使用します。また、データをStringまたはbyte配列として取得することもできます。この場合、ドライバによってストリーム処理が実行されます。

3種類のストリームのうち、任意のものを使用して、LONGまたはLONG RAWデータを取得できます。ドライバは、データベースおよびドライバの文字セットに応じて、変換を行います。



ノート:

LONG 列を使用した表を作成しないでください。かわりに、ラージ・オブジェクト(LOB)列 CLOB、NCLOB および BLOB を使用してください。LONG 列は、下位互換性のためにサポートされています。また、既存の LONG 列を LOB 列に変換することをお薦めします。LOB 列の制限の数は、LONG 列の制限の数よりも、かなり少なくなっています。

12.2.2 LONG RAWデータの変換

getBinaryStreamへのコールにより、RAWデータが戻されます。getAsciiStreamへのコールにより、RAWデータが16進データに変換され、ASCIIコードが戻されます。getUnicodeStreamへのコールにより、RAWデータが16進データに変換され、Unicode文

字が戻されます。

12.2.3 LONGデータの変換

getAsciiStreamを使用してLONGデータを取得した場合、ドライバは、データベースに格納されている基礎となるデータにUS7ASCIIまたはWE8ISO8859P1文字セットが使用されているものとみなします。実際にそうである場合、ドライバはASCII文字に対応するバイトを戻します。データベースでUS7ASCIIとWE8ISO8859P1のいずれの文字セットも使用されていない場合、getAsciiStreamへのコールを実行すると、意味のないコードが戻されます。

getUnicodeStreamを使用してLONGデータを取得すると、Unicode文字のストリームがUTF-16エンコーディングで戻されます。これは、Oracleがサポートする、データベースの基礎となる文字セットすべてに当てはまります。

getBinaryStreamを使用してLONGデータを取得する場合、次の2つのケースが考えられます。

- ドライバがJDBC OCIで、クライアントの文字セットがUS7ASCIIまたはWE8ISO8859P1でない場合、getBinaryStreamをコールするとUTF-8が戻されます。クライアントの文字セットがUS7ASCIIまたはWE8ISO8859P1に設定されている場合、US7ASCIIバイト・ストリームが戻されます。
- ドライバがJDBC Thinで、データベースの文字セットがUS7ASCIIまたはWE8ISO8859P1でない場合、getBinaryStreamをコールするとUTF-8が戻されます。サーバー側文字セットがUS7ASCIIまたはWE8ISO8859P1に設定されている場合、US7ASCIIバイト・ストリームが戻されます。



ノート:

LONG または LONG RAW 列をストリームとして受信するには、データベースから取り出す列の順序に注意する必要があります。

次の表に、LONGおよびLONG RAWデータ変換の概要をストリーム・タイプ別に示します。

表12-1 LONGおよびLONG RAWデータの変換

データ型	BinaryStream	AsciiStream	UnicodeStream
LONG	Unicode UTF-8 の文字を表すバイト。データベース文字セットが US7ASCII または WE8ISO8859P1 の場合、バイトは US7ASCII または WE8ISO8859P1 のキャラクタを表します。	ISO-Latin-1(WE8ISO8859P1)エンコーディングの文字を表すバイト。	Unicode UTF-16 エンコーディングの文字を表すバイト。
LONG RAW	データの変換なし	16 進バイトの ASCII 表現。	16 進バイトの Unicode 表現。

関連トピック

- グローバリゼーション・サポート
- データ・ストリームと複数列の関係

12.2.4 例: LONG RAWデータのストリーム

getXXXStreamメソッドには、データの増分的取得を可能にする機能があります。一方、getBytesは、すべてのデータを一度のコールでフェッチします。この項には、バイナリ・データ・ストリームの取得方法を示す2つの例が含まれています。LONG RAWデータを取得するのに、最初のバージョンではgetBinaryStreamメソッドを使用し、2番目のバージョンではgetBytesメソッドを使用します。

getBinaryStreamを使用したLONG RAWデータ列の取得

この例では、ローカル・ファイル・システム上のファイルにLONG RAW列の内容を書き込みます。この場合、ドライバはデータを増分的にフェッチします。

次のコードは、名前LESLIEに関連付けられたLONG RAWデータ列を格納する表を作成します。

```
-- SQL code:
create table streamexample (NAME varchar2 (256), GIFDATA long raw);
insert into streamexample values ('LESLIE', '00010203040506070809');
```

次のJavaコードの一部は、LONG RAW列のデータを、leslie.gifというファイルに書き込みます。

```
ResultSet rset = stmt.executeQuery
                 ("select GIFDATA from streamexample where NAME='LESLIE'");
// get first row
if (rset.next())
{
    // Get the GIF data as a stream from Oracle to the client
    InputStream gif_data = rset.getBinaryStream (1);
   try
   {
      FileOutputStream file = null;
      file = new FileOutputStream ("leslie.gif");
      int chunk;
      while ((chunk = gif_data.read()) != -1)
         file.write(chunk);
   }
   catch (Exception e)
   {
      String err = e.toString();
      System.out.println(err);
   }
   finally
   {
      if file != null()
         file.close();
   }
}
```

この例では、getBinaryStreamへのコールにより戻されるInputStreamオブジェクトは、データベース接続から直接データを読み取ります。

getBytesを使用したLONG RAWデータ列の取得

この例では、getBinaryStreamのかわりにgetBytesを使用して、GIFDATA列の内容を取得します。この場合、ドライバは一度

のコールですべてのデータをフェッチして、バイト配列に格納します。コードは次のようになります。

```
ResultSet rset2 = stmt.executeQuery
                  ("select GIFDATA from streamexample where NAME='LESLIE'");
// get first row
if (rset2.next())
{
   // Get the GIF data as a stream from Oracle to the client
   byte[] bytes = rset2.getBytes(1);
   try
   {
      FileOutputStream file = null;
      file = new FileOutputStream ("leslie2.gif");
      file.write(bytes);
   }
   catch (Exception e)
   {
      String err = e.toString();
      System.out.println(err);
   }
   finally
   {
      if file != null()
         file.close();
   }
}
```

LONG RAW列では最大2GBのデータを格納できるため、getBytesの使用例では、getBinaryStreamの使用例に比べて多量のメモリーを使用できます。LONGまたはLONG RAW列のデータの最大サイズが不明な場合は、ストリームを使用してください。

12.2.5 LONGまたはLONG RAWのストリーム回避について



ノート:

Oracle Database 12c リリース 1 (12.1)以降、このメソッドは非推奨になりました。

JDBCドライバは、任意のLONGおよびLONG RAW列を自動的にストリーム処理します。ただし、データ・ストリームを避ける状況が生じる場合もあります。たとえば、LONG列のサイズが非常に小さい場合、データが増分的ではなく、一度のコールで戻される方が望ましい場合があります。

ストリームを回避するには、defineColumnTypeメソッドを使用してLONG列の型を再定義します。たとえば、LONGまたはLONG RAW列をVARCHAR型またはVARBINARY型として再定義すると、ドライバがデータを自動的にストリーム処理することはなくなります。

defineColumnTypeを使用して列の型を再定義する場合、問合せの中で列の型を宣言する必要があります。列の型を宣言しないと、executeQueryが失敗します。また、Statementオブジェクトをoracle.jdbc.OracleStatementオブジェクトにキャストする必要があります。

さらに、defineColumnTypeを使用すると、問合せの実行時にOCIドライバがデータベースへラウンドトリップを行わずにすみます。defineColumnTypeを使用しない場合、これらのJDBCドライバは列タイプのデータ型を要求する必要があります。JDBC Thinドライバでは、ラウンドトリップの回数が常に最小であるため、defineColumnTypeを使用する利点はありません。

前の項の例を使用して、StatementオブジェクトstmtはOracleStatementへキャストされ、LONG RAWデータを含む列はVARBINARAY型として再定義されます。このデータはストリーム化されません。かわりに、バイト配列で戻されます。コードは次のようになります。

```
//cast the statement stmt to an OracleStatement
oracle.jdbc.OracleStatement ostmt =
   (oracle.jdbc.OracleStatement)stmt;
//redefine the LONG column at index position 1 to VARBINARY
ostmt.defineColumnType(1, Types.VARBINARY);
// Do a query to get the images named 'LESLIE'
ResultSet rset = ostmt.executeQuery
         ("select GIFDATA from streamexample where NAME='LESLIE'");
// The data is not streamed here
rset.next();
byte [] bytes = rset.getBytes(1);
```

関連トピック

- 非推奨となった機能

12.3 CHAR、VARCHARまたはRAW列のストリームについて



ノート:

Oracle Database 12c リリース 1 (12.1)以降、このメソッドは非推奨になりました。

defineColumnType Oracle拡張機能を使用してCHAR、VARCHARまたはRAW列をLONGVARCHARまたはLONGVARBINARYとして再定義する場合は、列をストリームとして取得できます。このプログラムは、列が実際にLONGまたはLONG RAW型であるかのように動作します。通常これらの列は短いため、これが問題になることはほとんどありません。

CHAR、VARCHARまたはRAW列を、列タイプを再定義することなくデータ・ストリームとして取得しようとすると、JDBCドライバはJava InputStreamを戻しますが、実際のストリームは発生しません。これらのデータ型の場合、JDBCドライバは、executeQueryメソッドまたはnextメソッドへのコール中に、データをインメモリーのバッファに完全にフェッチします。getXXXStreamエントリ・ポイントは、このバッファからデータを読み取るストリームを戻します。

関連トピック

- 非推奨となった機能

12.4 LOBおよび外部ファイルのストリームについて

ラージ・オブジェクト(LOB)という用語は、データベース表に直接格納するには大きすぎるデータ項目を指します。一方、ロケータはデータベース表に格納されて、実際のデータの場所を指します。外部ファイルも同様に管理されます。JDBCドライバは、ストリームの使用によって次の型をサポートできます。

- バイナリ・ラージ・オブジェクト(BLOB)

  非構造化バイナリ・データ用

- キャラクタ・ラージ・オブジェクト(CLOB)

  文字データ用

- 各国語キャラクタ・ラージ・オブジェクト(NCLOB)

  各国語キャラクタ・データ用

- バイナリ・ファイル(BFILE)

  外部ファイル用

LOBとBFILEは、この章で説明した他のストリーム・データとは動作が異なります。表に実際のデータが格納されるかわりに、ロケータが格納されます。ストリームとしてのデータの読取りおよび書込みなど、実際のデータを操作するには、このロケータを使用します。ストリーム処理するときでも、一部のデータ(サイズで定義済)のみがネットワーク間を移動します。これに対して、LONGまたはLONG RAWをストリーム処理するときは、すべてのデータがネットワーク間を移動します。

BLOB、CLOBおよびNCLOBのストリーム

問合せにより、1つ以上のBLOB、CLOBまたはNCLOB列をフェッチすると、JDBCドライバはデータをクライアントに転送します。このデータはストリームとしてアクセスできます。JDBCより取得したBLOB、CLOBまたはNCLOBデータを操作する場合、Oracleの拡張機能クラスoracle.sql.BLOB、oracle.sql.CLOBおよびoracle.sql.NCLOB内のメソッドを使用します。これらのクラスは、BLOB、CLOBまたはNCLOBから入力ストリーム内への読取り、出力ストリームからBLOB、CLOBまたはNCLOB内への書込み、BLOB、CLOBまたはNCLOBの長さの決定およびBLOB、CLOBまたはNCLOBのクローズなど、固有の機能を提供します。



ノート:

Oracle Database 12c リリース 1 (12.1)以降、oracle.sql パッケージの具象クラスは非推奨となり、oracle.jdbc パッケージのインタフェースに置き換えられています。標準互換性には(可能であれば)java.sql パッケージの使用可能なメソッドを使用し、Oracle 固有の拡張機能には oracle.jdbc パッケージの使用可能なメソッドを使用することをお薦めします。これらのインタフェースの詳細は、MoS ノート 1364193.1 を参照してください。

関連項目:

「LOBのデータ・インタフェース」

BFILEのストリーム

外部ファイルであるBFILEは、データベースの外部にあるファイルへのロケータを格納するために使用します。ファイルは、データ・サーバーのファイル・システム上のいずれかの場所に格納されます。ロケータは、ファイルが実際に格納された場所を指します。

問合せにより1つ以上のBFILE列をフェッチすると、JDBCドライバは必要に応じてファイルをクライアントに転送します。データにはストリームとしてアクセスできます。JDBCからBFILEデータを操作するには、Oracle拡張クラスoracle.sql.BFILEのメソッドを使用します。このクラスは、BFILEから入力ストリームへの読取り、出力ストリームからBFILEへの書込み、BFILEの長さの判定、BFILEのクローズなど、特定の機能を提供します。

12.5 データ・ストリームと複数列の関係

1つの問合せが複数の列をフェッチする場合、その列の1つにデータ・ストリームが含まれていると、ストリーム列に後続する列の内容はストリームが読み取られるまで利用できず、後続する列が読み取られると、ストリーム列はそれ以降利用できません。ストリーミング列の先にある列を読み取ろうとすると、ストリーミング列はクローズされます。

複数列によるストリーミングの例

次のコードについて検討します。

```
ResultSet rset = stmt.executeQuery
        ("select DATECOL, LONGCOL, NUMBERCOL from TABLE");
while rset.next()
{
   //get the date data
   java.sql.Date date = rset.getDate(1);
   // get the streaming data
   InputStream is = rset.getAsciiStream(2);
   // Open a file to store the gif data
   FileOutputStream file = new FileOutputStream ("ascii.dat");
   // Loop, reading from the ascii stream and
   // write to the file
   int chunk;
   while ((chunk = is.read ()) != -1)
      file.write(chunk);
   // Close the file
   file.close();
   //get the number column data
   int n = rset.getInt(3);
}
```

各行の受信データは、次の形式になります。

```
<a date><the characters of the long column><a number>
```

結果セットの各行を処理するときには、数値列を読み取る前にストリーム列の処理を完了する必要があります。

ストリーム・データ列のバイパス

ストリーム・データを含む列の読取りを回避することが望ましい場合があります。このようなデータの読取りを回避するには、ストリーム・オブジェクトのcloseメソッドをコールします。このメソッドを使用すると、ストリーム・データが廃棄され、ドライバが、ストリーム・データを含んだ列以降の非ストリーム・データを含んだ列すべてからデータを継続して読み取ることが可能になります。意図的にストリームを廃棄する場合でも、SELECT文で指定した順序で列を取り出すことは、よいプログラミング手法です。

次の例では、LONG列のストリーム・データを廃棄し、DATEおよびNUMBER列のデータのみをリカバリします。

```
ResultSet rset = stmt.executeQuery
        ("select DATECOL, LONGCOL, NUMBERCOL from TABLE");
while rset.next()
{
   //get the date
   java.sql.Date date = rset.getDate(1);

   // access the stream data and discard it with close()
```

```
    InputStream is = rset.getAsciiStream(2);
    is.close();

    // get the number column data
    int n = rset.getInt(3);
}
```

関連トピック

- ストリーム・データを使用する際の注意について
- LOBおよび外部ファイルのストリームについて

12.6 ストリームのクローズ

ストリームから取得したデータは、closeメソッドをコールすることによりいつでも廃棄できます。不要になったストリームはクローズするのが、望ましいプログラミング手法です。たとえば:

```
...
InputStream is = rset.getAsciiStream(2);
is.close();
```



ノート:

ストリームのクローズは、LONG または LONG RAW 列のパフォーマンスにはほとんど影響しません。データはすべて、そのままネットワーク経由で移動し、ドライバはネットワークからビットを読み取ることが必要です。

関連トピック

- データ・ストリームと複数列の関係
- ストリーム・データを使用する際の注意について

12.7 ストリームに関するノート

この項では、ストリームの使用に関して、次に示すいくつかの注意事項を説明します。

- ストリーム・データを使用する際の注意について
- setBytesとsetStringの制限を回避するためのストリームの使用方法について
- ストリームと行のプリフェッチの関係

12.7.1 ストリーム・データを使用する際の注意について

この項では、誤ってストリーム・データを廃棄または喪失することがないよう、事前に注意する必要がある点を説明します。現行ストリームの読取り以外の、データベースと通信する任意のJDBC操作を実行すると、ドライバは自動的にストリーム・データを廃棄します。2つの共通する注意事項について説明します。

- ストリーム・データはアクセス後に使用します。

  データ・ストリームを含む列からデータを取得するには、列のフェッチのみでは十分ではありません。列の内容をただちに

処理する必要があります。そうしないと、次の列のフェッチ時にその内容は廃棄されます。

- SELECT文で指定した順序でストリーム列をコールします。

  問合せを使用して複数列をフェッチすると、データベースは各行を、列を表すバイト・セットとして、SELECTで指定された順序で送信します。列の1つにストリーム・データが含まれる場合は、データベースは、次の列を処理する前にデータ・ストリーム全体を送信します。

  SELECT文で指定した順序を使用せずにデータへアクセスする場合は、ストリーム・データを失う可能性があります。つまり、ストリーム・データ列をバイパスして次の列のデータにアクセスすると、ストリーム・データは失われます。たとえば、ストリーム・データ列からデータを読み取る前にNUMBER列のデータにアクセスしようとすると、JDBCドライバは自動的に、まずストリーミング・データを読み取り、続いて廃棄します。LONG列に大量のデータが格納されている場合、これは非常に非効率的です。

  LONG列に後でアクセスしようとしても、データは使用できず、ドライバは「ストリームがクローズされています。」エラーを戻します。

次の例では、2番目の点について例証します。

```
ResultSet rset = stmt.executeQuery
        ("select DATECOL, LONGCOL, NUMBERCOL from TABLE");
while rset.next()
{
   int n = rset.getInt(3);  // This discards the streaming data
   InputStream is = rset.getAsciiStream(2);
                            // Raises an error: stream closed.
}
```

ストリームを取得して、それを使用する前にNUMBER列を取得した場合、ストリーム列はやはり自動的にクローズされます。

```
ResultSet rset = stmt.executeQuery
                 ("select DATECOL, LONGCOL, NUMBERCOL from TABLE");
while rset.next()
{
   InputStream is = rset.getAsciiStream(2); // Get the stream
   int n = rset.getInt(3);
   // Discards streaming data and closes the stream
}
int c = is.read(); // c is -1: no more characters to read-stream closed
```

12.7.2 setBytesとsetStringの制限を回避するためのストリームの使用方法について

Oracle Database 12c以降、setBytesメソッドおよびsetStringメソッドで使用されるデータ・サイズの上限は大幅に増加されました。どのJava byte配列もsetBytesに渡すことができ、どのJava StringもsetStringに渡すことができます。データのサイズ、SQL文またはPL/SQL文および使用するドライバによって、JDBCドライバは自動的に、setBinaryStreamかsetCharacterStreamの使用、またはsetBytesForBlobかsetStringForClobの使用に切り替わります。

以前のバージョンのOracle Database、およびサーバー側内部ドライバでは一部、制限があります。

関連トピック

- LOBのデータ・インタフェース

12.7.3 ストリームと行のプリフェッチの関係

JDBCドライバがデータ・ストリームを含む列に遭遇すると、行のフェッチ・サイズは1に再設定されます。行のフェッチ・サイズは、データベースにアクセスするたびに複数行のデータを取り出すことができるOracleのパフォーマンス強化点です。

13 Oracleオブジェクト型の操作

この章では、Java Database Database Connectivity(JDBC)でのユーザー定義オブジェクト型のサポートについて説明します。一般的な、弱い型指定のoracle.sql.STRUCTクラスの機能や、JDBC標準のSQLDataインタフェースまたはOracle固有のOracleDataインタフェースを実装するカスタムJavaクラスをマップする方法を説明します。



ノート:

Oracle Database 12c リリース 1 (12.1)以降、oracle.sql.STRUCT クラスは非推奨となり、oracle.jdbc.OracleStruct インタフェースに置き換えられています。このインタフェースは oracle.jdbc パッケージに属します。標準互換性には(可能であれば)java.sql パッケージの使用可能なメソッドを使用し、Oracle 固有の拡張機能には oracle.jdbc パッケージの使用可能なメソッドを使用することを強くお薦めします。oracle.jdbc.OracleStruct インタフェースの詳細は、MoS ノート 1364193.1 を参照してください。

内容は次のとおりです。

- Oracleオブジェクトのマッピングについて
- Oracleオブジェクト用のデフォルトSTRUCTクラスの使用方法について
- Oracleオブジェクト用のカスタム・オブジェクト・クラスの作成と使用方法について
- オブジェクト型の継承
- オブジェクト型の記述について

関連トピック

- PL/SQLタイプの使用について

13.1 Oracleオブジェクトのマッピングについて

Oracleオブジェクト型ではデータベースの複合データ構造がサポートされます。たとえば、name(CHAR型)、phoneNumber(CHAR型)およびemployeeNumber(NUMBER型)の属性を持つPerson型を定義できます。

Oracleでは、Oracleオブジェクト機能とJDBC機能が密接に統合されています。標準の汎用JDBC型を使用してOracleオブジェクトにマップすることも、カスタムJava型定義クラスを作成してマッピングをカスタマイズすることもできます。



ノート:

このマニュアルでは、Oracle オブジェクトにマップするために作成する Java クラスを、カスタム Java クラス、または、より具体的にカスタム・オブジェクト・クラスと呼びます。これは、オブジェクト参照にマップされる Java クラスであるカスタム参照クラスや、Oracle コレクションにマップされる Java クラスであるカスタム・コレクション・クラスに対比されるものです。

カスタム・オブジェクト・クラスは、データの読取りおよび書込みを行う標準JDBCインタフェースまたはOracle拡張機能インタ

フェースを実装できます。JDBCは、Oracleオブジェクトを特定のJavaクラスのインスタンスとして具体化します。JDBCを使用してOracleオブジェクトにアクセスするには、2つの主なステップを行います:

1. Oracleオブジェクト用にJavaクラスを作成します。
2. そのクラスにデータを移入します。次の選択肢があります。
    - JDBCを使用して、オブジェクトをSTRUCTオブジェクトとしてインスタンス化します。
    - OracleオブジェクトとJavaクラス間のマッピングを明示的に指定します。

      これにはオブジェクト・データ用にJavaクラスをカスタマイズすることも含まれます。そうすることにより、ドライバが指定されたカスタム・オブジェクト・クラスのインスタンスにデータを移入できるようになります。この場合、Javaクラスにいくつかの制約が生じます。これらの制約を満たすには、JDBC標準java.sql.SQLDataインタフェースまたはOracle拡張機能oracle.jdbc.OracleDataインタフェースを実装するようにクラスを定義します。



ノート:

SQLData インタフェースを使用する場合、弱い型指定の java.sql.Struct オブジェクトで十分な場合を除き、Java 型マップを使用して SQL と Java のマッピングを指定する必要があります。

13.2 Oracleオブジェクト用のデフォルトSTRUCTクラスの使用方法について

この項の内容は次のとおりです。

- Structクラスの使用の概要
- STRUCTオブジェクトと属性の取出し
- STRUCTオブジェクトの作成について
- STRUCTオブジェクトの文へのバインド
- STRUCT自動属性バッファリング

13.2.1 Structクラスの使用の概要

OracleオブジェクトのSQLとJavaのマッピングを行うカスタムJavaクラスを提供しない場合、Oracle JDBCはオブジェクトをjava.sql.Structインタフェースを実装するオブジェクトとしてインスタンス化します。

実際のSQL型が不明な場合は、通常、カスタムJavaオブジェクトのかわりにSTRUCTオブジェクトを使用します。たとえば、Javaアプリケーションを、エンド・ユーザー・アプリケーションではなく、データベースの任意のオブジェクト・データを操作するツールとして使用する場合があります。データベースから選択したデータをSTRUCTオブジェクトに挿入したり、データベースにデータを挿入するためにSTRUCTオブジェクトを作成したりできます。STRUCTオブジェクトは、データをSQL形式で維持するため、データを完全に保存します。アプリケーション固有の形式による情報が不要な場合は、STRUCTオブジェクトを使用すると、データをより効率的に、より正確に保存できます。

13.2.2 STRUCTオブジェクトと属性の取出し

この項では、Oracle固有の機能またはJDBC 2.0標準機能を使用して、Oracleオブジェクトとその属性を取り出し、操作する方法を説明します。



ノート:

JDBC ドライバは、埋込みオブジェクト(つまり、STRUCT オブジェクトの属性である STRUCT オブジェクト)を、通常のオブジェクトの場合と同様に、シームレスに処理します。オブジェクトである属性を取り出す JDBC ドライバは、同じ変換規則に従って、型マップ(使用可能な場合)またはデフォルトのマッピングを使用します。

java.sql.StructオブジェクトとしてのOracleオブジェクトの取出し

前述の例では、getObjectなどの標準JDBC機能を使用して、データベースからOracleオブジェクトをjava.sql.Structのインスタンスとして取り出すこともできます。getObjectメソッドによってjava.lang.Objectが戻されるため、メソッドの出力はStructにキャストする必要があります。たとえば:

```
ResultSet rs= stmt.executeQuery("SELECT * FROM struct_table");
java.sql.Struct jdbcStruct = (java.sql.Struct)rs.getObject(1);
```

oracle.sql型としての属性の取出し

STRUCTインスタンスまたはStructインスタンスからoracle.sql型としてOracleオブジェクト属性を取り出すには、次のように、oracle.sql.STRUCTクラスのgetOracleAttributesメソッドを使用します。

```
oracle.sql.Datum[] attrs = oracleSTRUCT.getOracleAttributes();
```

または

```
oracle.sql.Datum[] attrs = ((oracle.sql.STRUCT)jdbcStruct).getOracleAttributes();
```

標準Java型としての属性の取出し

STRUCTまたはStructインスタンスから標準Java型としてOracleオブジェクト属性を取り出すには、次のように標準getAttributesメソッドを使用します。

```
Object[] attrs = jdbcStruct.getAttributes();
```



ノート:

Oracle JDBC ドライバは、配列および構造の記述子をキャッシュします。そのためパフォーマンスは大きく向上します。ただし、データベース内で構造型の基礎となる型定義を変更すると、その構造型のキャッシュされた記述子は古くなり、アプリケーションで SQLException 例外が通知されます。

13.2.3 STRUCTオブジェクトの作成について

STRUCTオブジェクト作成の詳細は、「Package oracle.sql」を参照してください。

ノート:

データベースから適切な SQL オブジェクト型の STRUCT をフェッチしている場合、STRUCT 記述子を取得する最も簡単な方法は、フェッチした STRUCT オブジェクトのいずれかで getDescriptor をコールすることです。STRUCT 記述子は 1 つの SQL オブジェクト型に 1 つのみ必要です。

13.2.4 STRUCTオブジェクトの文へのバインド

oracle.sql.STRUCTオブジェクトをプリペアド文またはコール可能文にバインドするには、標準setObjectメソッド(型コードを指定)を使用するか、文オブジェクトをOracle文の型にキャストしてからOracle拡張機能のsetOracleObjectメソッドを使用します。たとえば:

```
PreparedStatement ps= conn.prepareStatement("text_of_prepared_statement");
Struct mySTRUCT = conn.createStruct (...);
ps.setObject(1, mySTRUCT, Types.STRUCT);
```

または

```
PreparedStatement ps= conn.prepareStatement("text_of_prepared_statement");
Struct mySTRUCT = conn.createStruct (...);
((OraclePreparedStatement)ps).setOracleObject(1, mySTRUCT);
```

13.2.5 STRUCT自動属性バッファリング

Oracle JDBCドライバには、STRUCT属性のバッファリングを有効または無効にするためのパブリック・メソッドが用意されています。



ノート:

Oracle Database 12c リリース 1 (12.1)以降、oracle.sql.STRUCT クラスは非推奨となり、oracle.jdbc.OracleStruct インタフェースに置き換えられています。このインタフェースは oracle.jdbc パッケージに属します。標準互換性には(可能であれば)java.sql パッケージの使用可能なメソッドを使用し、Oracle 固有の拡張機能には oracle.jdbc パッケージの使用可能なメソッドを使用することを強くお薦めします。oracle.jdbc.OracleStruct インタフェースの詳細は、MoS ノート 1364193.1 を参照してください。

oracle.sql.STRUCTクラスには、次のメソッドが含まれます。

- public void setAutoBuffering(boolean enable)
- public boolean getAutoBuffering()

setAutoBuffering(boolean)メソッドは自動バッファリングを有効または無効にします。getAutoBufferingメソッドは、現行の自動バッファリング・モードを戻します。デフォルトでは、自動バッファリングは無効です。

ARRAYデータをオーバーフローせずにJava仮想マシン(JVM)のメモリーに格納できると仮定し、STRUCT属性にgetAttributesメソッドとgetArrayメソッドで複数回アクセスする場合は、JDBCアプリケーションの自動バッファリングを有効にすることをお薦めします。



ノート:

変換した属性をバッファリングすると、JDBC アプリケーションでは、大量のメモリーが消費されます。

自動バッファリングを有効にすると、oracle.sql.STRUCTオブジェクトでは、変換したすべての属性のローカル・コピーが保持されます。このデータは保持されるため、この情報に後でアクセスするときにはデータ・フォーマット変換処理を実行しなくて済みます。

関連トピック

- ARRAY自動要素バッファリング

13.3 Oracleオブジェクト用のカスタム・オブジェクト・クラスの作成と使用方法について

この項の内容は次のとおりです。

- カスタム・オブジェクト・クラスの作成および使用の概要
- OracleDataとSQLDataの利点
- SQLDataを実装するための型マップについて
- SQLDataを実装するための型マップの作成とマッピングの定義について
- SQLData実装によるデータの読取りおよび書込みについて
- OracleDataインタフェースについて
- OracleData実装によるデータの読取りおよび書込みについて
- その他のOracleDataの使用方法

13.3.1 カスタム・オブジェクト・クラスの作成および使用の概要

Oracleオブジェクト用にカスタム・オブジェクト・クラスを作成する場合は、型マップのエントリを定義する必要があります。ドライバはこの型マップに従ってOracleオブジェクトに対応するカスタム・オブジェクト・クラスをインスタンス化します。

Oracleオブジェクトとその属性データからカスタム・オブジェクト・クラスのインスタンスを作成し、データを移入する方法も指定する必要があります。ドライバによって、カスタム・オブジェクト・クラスからのデータの読取り、およびカスタム・オブジェクト・クラスへのデータの移入が実行できる必要があります。また、カスタム・オブジェクト・クラスは、提供する必要があるかどうかにかかわらず、Oracleオブジェクトの属性に対応するgetXXXメソッドおよびsetXXXメソッドも提供できます。カスタム・クラスの作成とデータ移入、およびドライバの読取り/書込み機能の設定を行うには、次のインタフェースのいずれかを選択します。

- JDBC標準SQLDataインタフェース
- Oracleが提供するOracleDataインタフェースおよびOracleDataFactoryインタフェース

作成するカスタム・オブジェクト・クラスでは、これらのインタフェースのどちらかを実装する必要があります。OracleDataインタフェースは、カスタム・オブジェクト・クラスに対応するカスタム参照クラスを実装するときにも使用できます。ただし、SQLDataインタフェースを使用している場合、使用できるのは、java.sql.Refやoracle.sql.REFなど、弱いJava参照型のみです。SQLDataインタフェースは、SQLオブジェクトのマッピング専用です。

たとえば、データベースにEMPLOYEEというOracleオブジェクト型があり、そのオブジェクト型にはName(CHAR型)およびEmpNum(NUMBER型)という2つの属性が設定されているとします。型マップを使用して、EMPLOYEEオブジェクトがJEmployeeというカスタム・オブジェクト・クラスにマップされるように指定します。JEmployeeクラスでは、SQLDataまたはOracleDataインタフェースのどちらかを実装できます。

関連トピック

- オブジェクト型の継承

13.3.2 OracleDataとSQLDataの利点

2つのインタフェースのどちらを実装するかを決定する場合は、OracleDataとSQLDataの利点を考慮する必要があります。

SQLDataインタフェースは、SQLオブジェクトのマッピング専用です。OracleDataインタフェースは、より柔軟性が高く、他のSQL型と同じようにSQLオブジェクトをマップし、処理をカスタマイズできます。Oracle Databaseにあるどのデータ型からもOracleData実装を作成できます。これは、たとえば、JavaのRAWデータをシリアライズするときに役立ちます。

OracleDataインタフェースの利点

OracleDataインタフェースの利点は次のとおりです。

- Oracleオブジェクトの型マップ・エントリが必要ありません。
- Oracle拡張機能に対応しています。
- oracle.sql.STRUCTからOracleDataを作成できます。この方法は、ネイティブなJava型への変換が最小限で済むため、より効率的です。
- toJDBCObjectメソッドを使用して、OracleDataオブジェクトから、対応するJDBCオブジェクトを取得できます。

SQLDataの利点

SQLDataはJDBC標準であるため、コードの移植が可能です。

13.3.3 SQLDataを実装するための型マップについて

カスタム・オブジェクト・クラスでSQLDataインタフェースを使用する場合、Oracleオブジェクト型をJavaにマップする際に使用するカスタム・オブジェクト・クラスを指定する型マップ・エントリを作成する必要があります。接続オブジェクトのデフォルトの型マップか、結果セットからデータを取得するときに指定する型マップを使用できます。ResultSetインタフェースのgetObjectメソッドには、型マップの指定に使用できるシグネチャがあります。次のいずれかを使用できます。

```
rs.getObject(int columnIndex);
rs.getObject(int columnIndex, Map map);
```

SQLData実装を使用する場合、型マップ・エントリを含めないと、オブジェクトはデフォルトでoracle.jdbc.OracleStructインタフェースにマップされます。これに対し、OracleData実装には独自のマッピング機能があるため、型マップ・エントリが必要ありません。OracleData実装を使用する場合、OracleのgetObject(int columnindex, OracleDataFactory factory)メソッドを使用します。

型マップを使用して、JavaクラスをOracleオブジェクトのSQL型名に関連付けます。このマップは1対1のマッピングで、ハッシュ

表にキーワードと値のペアとして格納されます。Oracleオブジェクトからデータを読み取ると、JDBCドライバでは、型マップが参照され、Oracleオブジェクト型のデータのインスタンス化に使用されるJavaクラスが決定されます。Oracleオブジェクトにデータを書き込むと、JDBCドライバによって、SQLDataインタフェースのgetSQLTypeNameメソッドがコールされ、JavaクラスのSQL型名が取り出されます。SQLとJava間の実際の変換は、ドライバにより実行されます。

Oracleオブジェクトに対応しているJavaクラスの属性では、ネイティブなJava型またはOracleネイティブ型を使用して属性を格納できます。

関連トピック

- Oracleオブジェクト用のカスタム・オブジェクト・クラスの作成と使用方法について

13.3.4 SQLDataを実装するための型マップの作成とマッピングの定義について

この項の内容は次のとおりです。

- 型マップの作成およびマッピングの定義の概要
- 既存の型マップへのエントリの追加
- 新しい型マップの作成
- 型マップで指定されていないオブジェクト型のインスタンス化について

13.3.4.1 型マップの作成およびマッピングの定義の概要

SQLData実装を使用する場合、型マップを提供するのは、JDBCアプリケーション・プログラマの役割です。型マップは、標準のjava.util.Mapインタフェースを実装するクラスのインスタンスにする必要があります。

独自のクラスも作成できますが、標準のjava.util.Hashtableクラスが要件を満たしています。

型マップに使用するHashtableなどのクラスは、putメソッドを実装します。このメソッドは、キーワードと値のペアを入力として取ります。各キーは完全修飾SQL型名で、対応する値は指定されたJavaクラスのインスタンスです。

型マップは、接続インスタンスに関連付けられます。標準java.sql.ConnectionインタフェースとOracle固有oracle.jdbc.OracleConnectionクラスには、getTypeMapメソッドが含まれています。両方ともMapオブジェクトを戻します。

13.3.4.2 既存の型マップへのエントリの追加

最初に接続インスタンスが確立されたときには、デフォルト型マップは空です。デフォルト型マップにデータを移入する必要があります。

既存の型マップにエントリを追加するには、次のステップを実行してください。

1. OracleConnectionオブジェクトのgetTypeMapメソッドを使用して、接続の型マップ・オブジェクトを戻します。
   getTypeMapメソッドはjava.util.Mapオブジェクトを戻します。たとえば、OracleConnectionインスタンスoraconnがあるとします。

   ```
   java.util.Map myMap = oraconn.getTypeMap();
   ```



ノート:

OracleConnection インスタンスの型マップが初期化されていないと、getTypeMap を最初にコールしたとき、空のマップが戻されます。

2. 型マップのputメソッドを使用して、マップ・エントリを追加します。putメソッドでは2つの引数、つまり、SQL型名文字列およびそのSQL型をマップするJavaクラスのインスタンスを指定します。

```
myMap.put(sqlTypeName, classObject);
```

sqlTypeNameは、データベースのSQL型名の完全修飾名を表す文字列です。classObjectは、そのSQL型をマップするJavaクラス・オブジェクトです。次のようにClass.forNameメソッドを使用して、クラス・オブジェクトを取り出します。

```
myMap.put(sqlTypeName, Class.forName(className));
```

たとえば、CORPORATEデータベース・スキーマにPERSON SQLデータ型が定義されている場合は、そのSQLデータ型を、次の文でPersonとして定義されたPERSON Javaクラスにマップできます。

```
myMap.put("CORPORATE.PERSON", Class.forName("Person"));
oraconn.setTypeMap(newMap);
```

マップには、CORPORATEデータベースのPERSON SQLデータ型がPerson Javaクラスにマップされているエントリがあります。



ノート:

型マップの SQL 型名は、Oracle Database に大文字で格納されているので、すべて大文字にする必要があります。

13.3.4.3 新しい型マップの作成

新しい型マップを作成するには、次の一般的なステップを実行します。この例ではjava.util.Hashtableのインスタンスを使用しています。このインスタンスはjava.util.Dictionaryを拡張したもので、java.util.Mapを実装しています。

1. 新しい型マップ・オブジェクトを作成します。

```
Hashtable newMap = new Hashtable();
```

2. 型マップ・オブジェクトのputメソッドを使用して、マップにエントリを追加します。たとえば、CORPORATEデータベースにEMPLOYEEというSQL型が定義されている場合は、次のように、そのSQL型をEmployee.javaに定義されているEmployeeクラス・オブジェクトにマップできます。

```
newMap.put("CORPORATE.EMPLOYEE", class.forName("Employee"));
```

3. エントリをマップに追加した後に、OracleConnectionオブジェクトのsetTypeMapメソッドを使用して、接続の既存の型マップを上書きする必要があります。たとえば:

```
oraconn.setTypeMap(newMap);
```

この例では、setTypeMapメソッドによってoraconn接続オブジェクトの元のマップがnewMapに上書きされます。



ノート:

接続インスタンスのデフォルト型マップは、マッピングが必要なときに、マップ名を指定しないと使用されます。たとえば、入力としてマップを指定せずに、結果セットの getObject をコールしたときに使用されます。

13.3.4.4 型マップで指定されていないオブジェクト型のインスタンス化について

getObjectコールを使用するときに、適切なエントリを持つ型マップを指定しないと、JDBCドライバは oracle.jdbc.OracleStructインタフェースのインスタンスとしてOracleオブジェクトをインスタンス化します。Oracleオブジェクト型に埋込みオブジェクトが格納されていて、それらのオブジェクトが型マップに存在しない場合、ドライバは埋込みオブジェクトも oracle.jdbc.OracleStructのインスタンスとしてインスタンス化します。埋込みオブジェクトが型マップに存在する場合、getAttributesメソッドをコールすると、埋込みオブジェクトは型マップで指定されたJavaクラスのインスタンスとして戻されます。

13.3.5 SQLData実装によるデータの読取りおよび書込みについて

この項では、Oracleオブジェクトに対応するJavaクラスがSQLDataを実装している場合に、そのOracleオブジェクトに対してデータの読取りまたは書込みを行う方法について説明します。

結果セットからのSQLDataオブジェクトの読取り

ここでは、カスタム・オブジェクト・クラスに対してSQLData実装を選択したときに、OracleオブジェクトのデータをJavaアプリケーションに読み取るステップについて説明します。

このステップでは、すでにOracleオブジェクト型を定義し、対応するカスタム・オブジェクト・クラスを作成し、OracleオブジェクトとJavaクラス間のマッピングを定義する型マップを更新し、文オブジェクトstmtを定義しているものとします。

1. データベースに問合せを行い、OracleオブジェクトをJDBC結果セットに読み取ります。

   ```
   ResultSet rs = stmt.executeQuery("SELECT emp_col FROM personnel");
   ```

   表PERSONNELには、SQL型EMP_OBJECTのEMP_COLという列が1つ含まれています。このSQL型は、Javaクラス Employeeにマップされるように、型マップに定義されています。

2. Oracleの結果セットのgetObjectメソッドを使用して、カスタム・オブジェクト・クラスのインスタンスに結果セットの1行からデータを移入します。型マップにはEmployeeのエントリが含まれるため、getObject メソッドによりユーザー定義の SQLDataオブジェクトが戻されます。

   ```
   if (rs.next())
      Employee emp = (Employee)rs.getObject(1);
   ```

   型マップにオブジェクトのエントリが存在しない場合は、getObjectメソッドによりoracle.jdbc.OracleStructオブジェクトが戻されます。getObjectメソッド・シグネチャにより汎用のjava.lang.Object型が戻されるため、出力を OracleStruct型にキャストします。

   ```
   if (rs.next())
      OracleStruct empstruct = (OracleStruct)rs.getObject(1);
   ```

   getObjectメソッドがreadSQLをコールすると、SQLDataインタフェースからreadXXXがコールされます。



ノート:

定義済の型マップを使用しないようにするには、getSTRUCT メソッドを使用します。このメソッドでは、型マップにマッピング・エントリが定義されている場合でも、常に STRUCT オブジェクトが戻されます。

3. カスタム・オブジェクト・クラスにgetメソッドが定義されている場合、このメソッドを使用して、オブジェクト属性のデータを読み取ります。たとえば、EMPLOYEEに、CHAR型の属性EmpNameおよびNUMBER型の属性EmpNumがある場合は、Java Stringを戻すgetEmpNameメソッドおよびint値を戻すgetEmpNumメソッドを指定します。次に、Javaアプリケーションでこれらのメソッドを次の方法でコールします。

```
String empname = emp.getEmpName();
int empnumber = emp.getEmpNum();
```

コール可能文OUTパラメータからのSQLDataオブジェクトの取出し

PL/SQLファンクションGETEMPLOYEEをコールするCallableStatementインスタンスcsについて考えてみます。このファンクションには、プログラムによって従業員番号が渡されます。そのファンクションから対応するEmployeeオブジェクトが戻されます。このオブジェクトを取り出すには、次の手順を実行します。

1. GETEMPLOYEEファンクションをコールするCallableStatementを次のように準備します。

```
CallableStatement ocs = conn.prepareCall("{ ? = call GETEMPLOYEE(?) }");
```

2. empnumberを、GETEMPLOYEEの入力パラメータとして宣言します。SQLDataオブジェクトを、型コードOracleTypes.STRUCTを指定してOUTパラメータとして登録します。次に、文を実行します。これは、次の方法で行います。

```
cs.setInt(2, empnumber);
cs.registerOutParameter(1, OracleTypes.STRUCT, "EMP_OBJECT");
cs.execute();
```

3. getObjectメソッドを使用して、employeeオブジェクトを取り出します。

```
Employee emp = (Employee)cs.getObject(1);
```

型マップ・エントリが存在しない場合は、getObjectメソッドによりjava.sql.Structオブジェクトが戻されます。

```
Struct emp = cs.getObject(1);
```

SQLDataオブジェクトのINパラメータとしてのコール可能文への引渡し

EmployeeオブジェクトをINパラメータとして取り、そのオブジェクトをPERSONNEL表に追加するPL/SQLファンクションaddEmployee(?)があると仮定します。この例のempは、有効なEmployeeオブジェクトです。

1. addEmployee(?)ファンクションをコールするためにCallableStatementを準備します。

```
CallableStatement cs =
  conn.prepareCall("{ call addEmployee(?) }");
```

2. setObjectを使用して、empオブジェクトをINパラメータとしてコール可能文に渡します。次に、文をコールします。

```
cs.setObject(1, emp);
cs.execute();
```

SQLData実装を使用したデータのOracleオブジェクトへの書込み

ここでは、カスタム・オブジェクト・クラスに対してSQLData実装を選択したときに、Javaアプリケーションのデータを Oracleオブジェクトに書き込むステップについて説明します。

ここでは、あらかじめOracleオブジェクト型を定義し、対応するJavaクラスを作成し、OracleオブジェクトとJavaクラス間のマップを定義する型マップを更新していることを前提としています。

1. カスタム・オブジェクト・クラスにsetメソッドが定義されている場合、このメソッドを使用して、アプリケーションのJava変数のデータをJavaデータ型オブジェクトの属性に書き込みます。

```
emp.setEmpName(empname);
emp.setEmpNum(empnumber);
```

2. Javaデータ型オブジェクトのデータを使用して、データベース表の行に格納されているOracleオブジェクトを更新する文を、適切に準備します。

```
PreparedStatement pstmt = conn.prepareStatement
                          ("INSERT INTO PERSONNEL VALUES (?)");
```

3. プリペアド文のsetObjectメソッドを使用して、Javaデータ型オブジェクトをプリペアド文にバインドします。

```
pstmt.setObject(1, emp);
```

4. 文を実行すると、データベースが更新されます。

```
pstmt.executeUpdate();
```

13.3.6 OracleDataインタフェースについて

oracle.jdbc.OracleDataおよびoracle.jdbc.OracleDataFactoryインタフェースを実装するカスタム・オブジェクト・クラスを作成して、Javaアプリケーションで使用できるOracleオブジェクトおよびその属性データを作成できます。OracleDataおよびOracleDataFactoryインタフェースはOracle固有であり、JDBC標準の一部ではありません。

ノート:



Oracle Database 12c リリース 1 (12.1)以降、OracleData および OracleDataFactory インタフェースは ORAData および ORADataFactory インタフェースに置き換わりました。

OracleDataインタフェース機能の理解

OracleDataインタフェースには、次の利点があります。

- 標準JDBC型に対するOracle拡張機能をサポートします。
- 作成中のJavaカスタム・クラスの名前を指定するときに、型マップは必要ありません。
- これによりパフォーマンスが改善されます。OracleDataは、Oracleオブジェクトを保持するためにドライバが使用する内部書式であるDatum型を直接伴って機能します。

OracleDataインタフェースおよびOracleDataFactoryインタフェースは次のように実行します。

- OracleDataクラスのtoJDBCObjectメソッドでは、データをoracle.jdbc.*表現に変換します。
- OracleDataFactoryでは、カスタム・オブジェクト・クラスのコンストラクタと等価のcreateメソッドが指定されます。このメソッドでは、OracleDataインスタンスを作成して戻します。JDBCドライバは、createメソッドを使用して、カスタム・オブジェクト・クラスのインスタンスをJavaアプリケーションに戻します。このメソッドは、入力として、java.lang.Objectオブジェクトと、OracleTypesクラスで指定された対応するSQL型コードを示す整数を取ります。

OracleDataおよびOracleDataFactoryは、次のように定義されます。

```
package oracle.jdbc;
import java.sql.Connection;
import java.sql.SQLException;
public interface OracleData
{
   public Object toJDBCObject(Connection conn) throws SQLException;
}

package oracle.jdbc;
import java.sql.SQLException;
public interface OracleDataFactory
{
    public OracleData create(Object jdbcValue, int sqlType) throws SQLException;

}
```

connは接続オブジェクトを、jdbcValueは作成されるオブジェクトを初期化するために使用するjava.lang.object型のオブジェクトを、sqlTypeは指定されたDatumオブジェクトのSQL型をそれぞれ表します。

オブジェクト・データの取出しと挿入

オブジェクト・データをOracleDataのインスタンスとして取得および挿入する場合、JDBCドライバでは次のメソッドを使用します。

オブジェクト・データは、次のいずれかの方法で取り出すことができます。

- Oracle固有OracleResultSetインタフェースの次のgetObjectメソッドを使用します。

  ```
  ors.getObject(int col_index, OracleDataFactory factory
  );
  ```

  このメソッドでは、結果セットのデータの列索引およびOracleDataFactoryインスタンスを入力として取ります。たとえば、カスタム・オブジェクト・クラスにgetOracleDataFactoryメソッドを実装して、getObjectメソッドに入力するOracleDataFactoryインスタンスを作成できます。OracleDataを実装するJavaクラスを使用するときは、型マップは必要ありません。

- ResultSetインタフェースで指定される標準getObject(index, map)メソッドを使用して、OracleDataのインスタンスとしてデータを取り出します。この場合、指定されたオブジェクト型で使用されるファクトリ・クラスおよび対応するSQL型名を識別するには、型マップにエントリを定義する必要があります。

オブジェクト・データは、次のいずれかの方法で挿入できます。

- Oracle固有OraclePreparedStatementクラスの次のsetObjectメソッドを使用します。

  ```
  setObject(int bind_index, Object custom_object);
  ```

  このメソッドでは、入力としてバインド変数のパラメータ索引、および変数を含むオブジェクト名としてOracleDataのイン

スタンスを取ります。

● PreparedStatementインタフェースで指定される標準setObjectメソッドを使用します。このメソッドを別のフォームで使用して、型マップなしでOracleDataインスタンスを挿入することもできます。

この後の項では、getObjectメソッドとsetObjectメソッドについて説明します。

OracleオブジェクトEMPLOYEEの例を引き続き使用するには、Javaアプリケーションに次のコードを記述する必要があります。

```
OracleData obj = ors.getObject(1, Employee.getOracleDataFactory());
```

この例では、orsはOracleResultSetインタフェースのインスタンスで、getObjectはOracleDataオブジェクトを取得するために使用されるOracleResultSetインタフェースのメソッドで、EMPLOYEEは結果セットの列1です。static Employee.getOracleDataFactoryメソッドによって、OracleDataFactoryがJDBCドライバに戻されます。JDBCドライバでは、このオブジェクトからcreate()をコールし、結果セットのデータが移入されたEmployeeクラスのインスタンスをJavaアプリケーションに戻します。



ノート:

● OracleData および OracleDataFactory は、独立したインタフェースとして定義されるため、必要に応じて異なる Java クラスから実装できます。

● OracleData インタフェースを使用するには、カスタム・オブジェクト・クラスで oracle.jdbc.*をインポートする必要があります。

13.3.7 OracleData実装によるデータの読取りおよび書込みについて

この項では、対応するJavaクラスがOracleDataを実装する場合に、Oracleオブジェクトに対してデータの読取りまたは書込みを行う方法について説明します。

OracleData実装を使用したOracleオブジェクトからのデータの読取り

ここでは、データをOracleオブジェクトからJavaアプリケーションに読み取るステップについて説明します。これらのステップでは、OracleDataを手動で実装するか、Oracle JVM Webサービス・コールアウト・ユーティリティを使用して、カスタム・オブジェクト・クラスを作成します。

このステップで、すでにOracleオブジェクト型を定義して対応するカスタム・オブジェクト・クラスを作成したか、Oracle JVM Webサービス・コールアウト・ユーティリティを使用してこれを作成し、文オブジェクトstmtを定義したものとします。

1. データベースに問合せを行ってOracleオブジェクトを結果セットに読み取り、Oracle結果セットにキャストします。

   ```
   OracleResultSet ors = (OracleResultSet)stmt.executeQuery
                          ("SELECT Emp_col FROM PERSONNEL");
   ```

   PERSONNELは、1列の表です。列名はEmp_col、型はEmployee_objectです。

2. Oracleの結果セットのgetObjectメソッドを使用して、カスタム・オブジェクト・クラスのインスタンスに結果セットの1行からデータを移入します。getObjectメソッドはjava.lang.Objectオブジェクトを戻します。これを固有のカスタム・オブ

ジェクト・クラスにキャストすることができます。

```
if (ors.next())
   Employee emp = (Employee)ors.getObject(1, Employee.getOracleDataFactory());
```

または

```
if (ors.next())
   Object obj = ors.getObject(1, Employee.getOracleDataFactory());
```

この例では、Employeeがカスタム・オブジェクト・クラスの名前であり、orsがOracleResultSetインスタンスの名前であることを想定しています。

たとえば、オブジェクトのSQL型名がEMPLOYEEの場合は、対応するJavaクラスはEmployeeで、OracleDataが実装されます。対応するファクトリ・クラスはEmployeeFactoryで、OracleDataFactoryが実装されます。

次の文を使用して、型マップにEmployeeFactoryエントリを宣言します。

```
map.put ("EMPLOYEE", Class.forName ("EmployeeFactory"));
```

次に、getObjectの形式を使用し、マップ・オブジェクトを指定します。

```
Employee emp = (Employee) rs.getObject (1, map);
```

接続のデフォルトの型マップに、指定されたオブジェクト型と対応するSQL型名に対して使用されるファクトリ・クラスを識別するエントリがすでにある場合は、次の形式のgetObjectを使用できます。

```
Employee emp = (Employee) rs.getObject (1);
```

3. カスタム・オブジェクト・クラスにgetメソッドがある場合は、そのメソッドを使用して、オブジェクト属性のデータをアプリケーションのJava変数に読み取ります。たとえば、EMPLOYEEに、CHAR型のEmpNameおよびNUMBER型のEmpNumがある場合は、Java Stringを戻すgetEmpNameメソッドおよび整数値を戻すgetEmpNumメソッドを指定します。次に、Javaアプリケーションでこれらのメソッドを次の方法でコールします。

```
String empname = emp.getEmpName();
int empnumber = emp.getEmpNum();
```

OracleData実装を使用したデータのOracleオブジェクトへの書込み

ここでは、データをJavaアプリケーションからOracleオブジェクトに書き込むステップについて説明します。これらのステップでは、OracleDataを手動で実装するか、Oracle JVM Webサービス・コールアウト・ユーティリティを使用して、カスタム・オブジェクト・クラスを作成します。

このステップでは、すでにOracleオブジェクト型を定義し、対応するカスタム・オブジェクト・クラスを作成しているものとします。



ノート:

データベースの INSERT および UPDATE 操作の実行時には、型マップは使用されません。

1. カスタム・オブジェクト・クラスにsetメソッドが定義されている場合、このメソッドを使用して、アプリケーションのJava変数のデータをJavaデータ型オブジェクトの属性に書き込みます。

```
emp.setEmpName(empname);
emp.setEmpNum(empnumber);
```

2. Javaデータ型オブジェクトのデータを使用して、Oracleオブジェクトを更新するプリペアド文を、適切にデータベース表の行に書き込みます。

```
OraclePreparedStatement opstmt = conn.prepareStatement
   ("UPDATE PERSONNEL SET Employee = ? WHERE Employee.EmpNum = 28959);
```

この例では、connはConnectionオブジェクトです。

3. OraclePreparedStatementインタフェースのsetObjectメソッドを使用して、Javaデータ型オブジェクトをプリコンパイルされた文にバインドします。

```
opstmt.setObject(1,emp);
```

setObjectメソッドでは、カスタム・オブジェクト・クラス・インスタンスのtoJDBCObjectメソッドをコールすることにより、データベースに書き込むことができるoracle.jdbc.OracleStructオブジェクトが取り出されます。



ノート:

Java データ型オブジェクトを、IN または OUT バインド変数として使用できます。

13.3.8 その他のOracleDataの使用方法

OracleDataインタフェースには、SQLDataインタフェースより優れた柔軟性があります。SQLDataインタフェースは、Oracleオブジェクト型から目的のJava型へのマッピングをカスタマイズする目的で設計されています。SQLDataインタフェースを実装すると、JavaとSQL型間の変換が正常に終了してから、元のSQLオブジェクト・データからカスタムJavaクラス・インスタンスのフィールドへのデータ移入、およびその逆が、JDBCドライバによって実行されます。

OracleDataインタフェースの場合は、Oracleオブジェクト型からJava型へのカスタマイズがサポートされるのみでなく、その場合、Javaオブジェクト型と、oracle.sqlパッケージでサポートされる任意のSQL型の間のマッピングを用意します。

このインタフェースは、oracle.sql.*型をラップするためのカスタムJavaクラスを提供したり、カスタマイズされた変換または機能を実装したりするときに役立ちます。次の使用例が考えられます。

- データの暗号化および復号化、または妥当性チェックを実行します。
- 読取りまたは書込みを行った値のロギングを実行します。
- URL情報を含む文字フィールドなどのキャラクタ列を小さなコンポーネントに解析します。
- 文字列を数値定数にマップします。
- データをより適したJava形式にマップします。たとえば、DATEフィールドをjava.util.Date形式にマップします。
- データ表現をカスタマイズします。たとえば、表の列のデータがフィート単位のとき、選択後にメートルで表現します。
- Javaオブジェクトをシリアライズおよびデシリアライズします。

たとえば、OracleDataを使用すると、データベースの特定のSQLオブジェクト型に対応していないJavaオブジェクトのインスタンスを、SQL型RAWの列に格納できます。OracleDataFactoryのcreateメソッドでは、oracle.sql.RAW型のオブジェクトから目的のJavaオブジェクトへの変換を実装する必要があります。OracleDataのtoJDBCObjectメソッドでは、Javaオブジェクトからoracle.sql.RAW型のオブジェクトへの変換を実装する必要があります。Javaシリアライズを使用してこれを実行することもでき

ます。

JDBCドライバでは、データのRAWバイトをoracle.sql.RAW形式で透過的に取り出します。次に、OracleDataFactoryのcreateメソッドをコールし、oracle.sql.RAWオブジェクトを目的のJavaクラスに変換します。

データベースにJavaオブジェクトを挿入するときは、そのオブジェクトをRAW型の列にバインドするのみでデータベースに格納できます。ドライバは、OracleDataのtoJDBCObjectメソッドを透過的にコールして、Javaオブジェクトをoracle.sql.RAW型のオブジェクトに変換します。このオブジェクトは、データベースにRAW型の1列として格納されます。

JDBCドライバによって使用される内部書式であるoracle.sql.*書式を使用して変換が機能するように設計されているため、OracleDataインタフェースのサポートは非常に効率的でもあります。さらに、型マップはSQLDataインタフェースのために必要なものであり、OracleDataを実装するJavaクラスを使用する場合は不要です。

関連トピック

- OracleDataインタフェースについて

13.4 オブジェクト型の継承

オブジェクト型の継承により、別のオブジェクト型を拡張して新しいオブジェクト型を作成することができます。新しいオブジェクト型は、拡張元のオブジェクト型のサブタイプになります。サブタイプは、そのスーパータイプに定義されたすべての属性およびメソッドを自動的に継承します。サブタイプは、属性およびメソッドを追加し、スーパータイプから継承されたメソッドをオーバーロードまたはオーバーライドできます。

オブジェクト型の継承によって、代入可能性が導入されます。代入可能性とは、T型の任意のサブタイプに加えて、T型の値を保持するように宣言したスロットの機能です。Oracle JDBCドライバは、透過的に代入可能性を処理します。

データベース・オブジェクトは、情報が失われることなく、最も固有の型で戻されます。たとえば、STUDENT_TオブジェクトがPERSON_Tスロットに格納されている場合、Oracle JDBCドライバはSTUDENT_Tオブジェクトを表すJavaオブジェクトを戻します。

この項の内容は次のとおりです。

- サブタイプの作成について
- サブタイプに対してカスタマイズされたクラスの実装について
- サブタイプ・オブジェクトの取得について
- サブタイプ・オブジェクトの作成
- サブタイプ・オブジェクトの送信
- サブタイプ・データ・フィールドへのアクセス
- 継承メタデータ・メソッド

13.4.1 サブタイプの作成について

明示的にOracleオブジェクト型に対応するJavaクラスを作成するには、カスタム・オブジェクト・クラスを作成します。オブジェクト型の階層がある場合には、Javaクラスの対応する階層を作成できます。

JDBCでデータベース・サブタイプを作成する最も一般的な方法は、java.sql.StatementインタフェースのexecuteメソッドをSQLのCREATE TYPEコマンドを実行することです。たとえば、次の図で示すように型継承階層を作成する場合があります。

図13-1 型継承階層



この場合、JDBCコードは次のようになります。

```
Statement s = conn.createStatement();
s.execute ("CREATE TYPE Person_T (SSN NUMBER, name VARCHAR2(30),
  address VARCHAR2(255))");
s.execute ("CREATE TYPE Student_T UNDER Person_t (deptid NUMBER,
  major VARCHAR2(100))");
s.execute ("CREATE TYPE PartTimeStudent_t UNDER Student_t (numHours NUMBER)");
```

次のコードでは、ST型のfooメンバー・プロシージャがオーバーロードされ、printメンバー・プロシージャによってT型から継承されたコピーが上書きされます。

```
CREATE TYPE T AS OBJECT (...,
  MEMBER PROCEDURE foo(x NUMBER),
  MEMBER PROCEDURE Print(),
  ...
  NOT FINAL;
CREATE TYPE ST UNDER T (...,
  MEMBER PROCEDURE foo(x DATE),          <-- overload "foo"
  OVERRIDING MEMBER PROCEDURE Print(),    <-- override "print"
  STATIC FUNCTION bar(...) ...
  ...
  );
```

サブタイプを作成すると、実表の列またはオブジェクト型の属性として使用できます。

関連項目:

Oracle Databaseオブジェクト・リレーショナル開発者ガイド

13.4.2 サブタイプに対してカスタマイズされたクラスの実装について

一般に、カスタマイズされたJavaクラスはデータベース・オブジェクト・タイプを表します。サブタイプに対してカスタマイズされたJavaクラスを作成するときには、Javaクラスでデータベース・オブジェクト・タイプの階層をミラー化するかどうかを選択できます。

クラスの作成には、OracleDataまたはSQLDataソリューションのいずれかを使用して、オブジェクト型の階層をマップできます。

この項の内容は次のとおりです。

- 型継承階層のためのOracleDataの使用について
- 型継承階層のためのSQLDataの使用について

13.4.2.1 型継承階層のためのOracleDataの使用について

oracle.sql.OracleDataインタフェースを実装するJavaクラスを使用した、カスタマイズされたマッピングを使用することをお薦めします。OracleDataのマッピングでは、JDBCアプリケーションがOracleDataおよびOracleDataFactoryインタフェースを実装する必要があります。OracleDataFactoryインタフェースを実装するクラスは、オブジェクトを作成するファクトリ・メソッドを格納します。各オブジェクトはそれぞれ1つのデータベース・オブジェクトを表します。

OracleDataインタフェースを実装するクラスの階層は、データベース・オブジェクト・タイプ階層をミラー化できます。たとえば、PERSON_TおよびSTUDENT_Tに対するJavaクラスのマッピングは、次のようになります。

OracleDataを使用するPerson.java

次のコードは、OracleDataおよびOracleDataFactoryインタフェースを実装するPerson_javaクラスを示します。

```
public static OracleDataFactory getOracleDataFactory()
  {
    return _personFactory;
  }

  public Person () {}

  public Person(NUMBER ssn, CHAR name, CHAR address)
  {
    this.ssn = ssn;
    this.name = name;
    this.address = address;
  }

  public Object toJDBCObject(OracleConnection c) throws SQLException
  {
    Object [] attributes = { ssn, name, address };
 Struct struct = c.createStruct("HR.PERSON_T", attributes);
    return struct;
  }

  public OracleData create(Object jdbcValue, int sqlType) throws SQLException
  {
    if (d == null) return null;
    Object [] attributes = ((STRUCT) d).getOracleAttributes();
    return new Person((NUMBER) attributes[0],
                      (CHAR) attributes[1],
                      (CHAR) attributes[2]);
  }
}
```

Person.javaを拡張するStudent.java

次のコードは、Person.javaクラスを拡張するStudent.javaクラスを示します。

```
class Student extends Person
{
  static final Student _studentFactory = new Student ();

  public NUMBER deptid;
  public CHAR major;

  public static OracleDataFactory getOracleDataFactory()
  {
    return _studentFactory;
  }

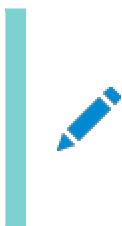
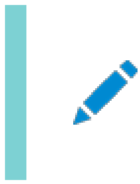
  public Student () {}

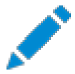
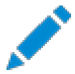
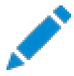
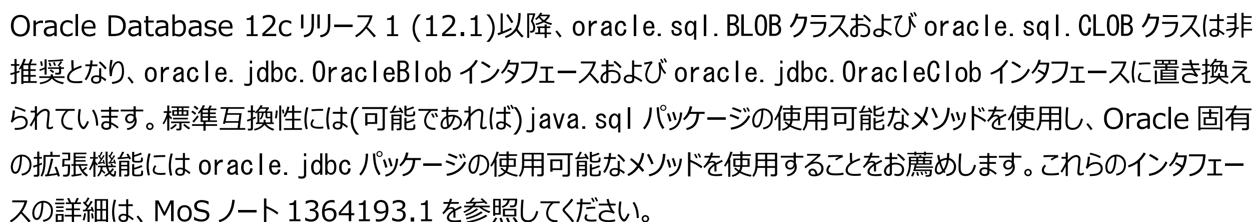
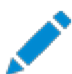
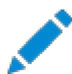
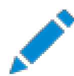
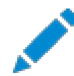
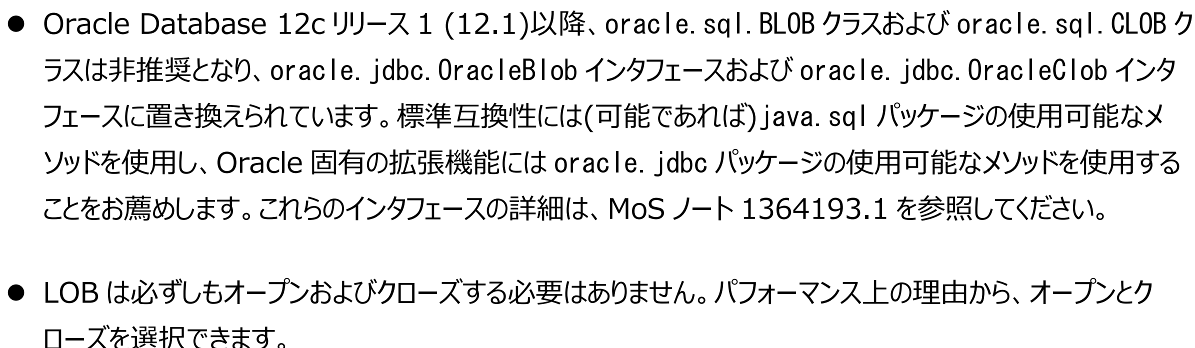
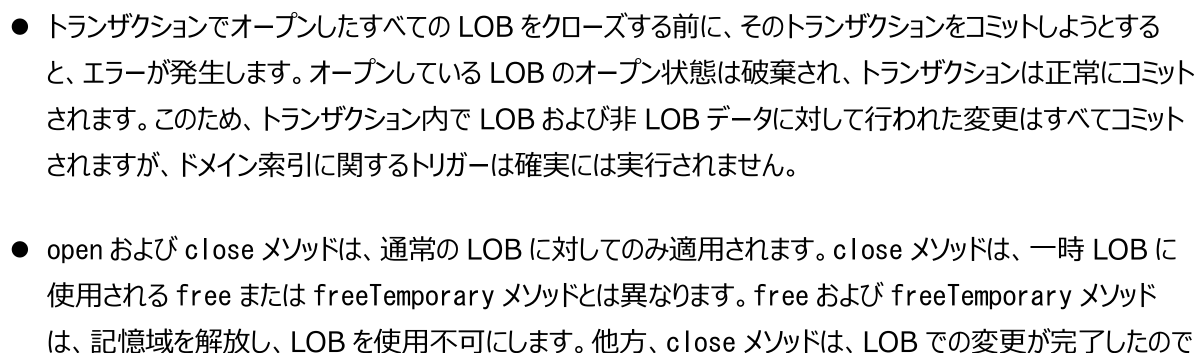
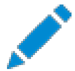
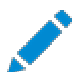
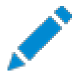
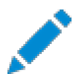
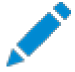
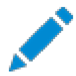
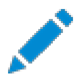
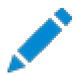
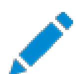
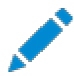
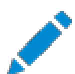
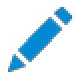
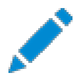
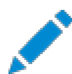
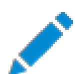
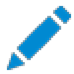
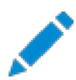
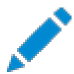
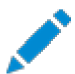
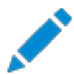
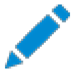
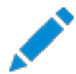
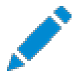
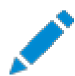
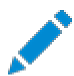
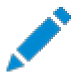
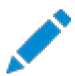
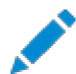
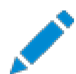
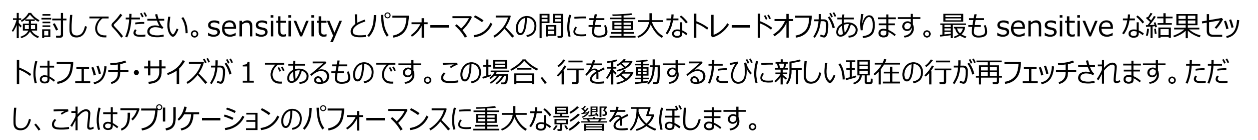
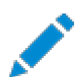
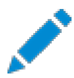
  public Student (NUMBER ssn, CHAR name, CHAR address,
                  NUMBER deptid, CHAR major)
  {
    super (ssn, name, address);
    this.deptid = deptid;
    this.major = major;
  }

  public Object toJDBCObject(OracleConnection c) throws SQLException
  {
    Object [] attributes = { ssn, name, address, deptid, major };
    Struct struct = c.createStruct("HR.STUDENT_T", attributes);
    return struct;
  }

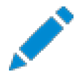
  public OracleData create(Object jdbcValue, int sqlType) throws SQLException
  {
    if (d == null) return null;
    Object [] attributes = ((STRUCT) d).getOracleAttributes();
    return new Student((NUMBER) attributes[0],
                       (CHAR) attributes[1],
                       (CHAR) attributes[2],
                       (NUMBER) attributes[3],
                       (CHAR) attributes[4]);
  }
}
```

OracleDataインタフェースを実装するカスタマイズされたクラスでは、必ずしもデータベース・オブジェクト型階層をミラー化する必要はありません。たとえば、スーパークラスを指定せずにStudentクラスを宣言できます。この場合、Studentには、PERSON_Tから継承された属性およびSTUDENT_Tによって宣言された属性を保持するフィールドが含まれます。

OracleDataFactory実装

次の例に示すように、JDBCアプリケーションでは、データベースの問合せにファクトリ・クラスを使用して、Personまたはそのサブクラスのインスタンスを戻します。

```
ResultSet rset = stmt.executeQuery ("select person from tab1");
while (rset.next())
{
  rset.getOracleData(1,Person.getOracleDataFactory());
```

```
  ...
}
```

OracleDataFactoryインタフェースを実装したクラスでは、関連付けられたカスタム・オブジェクト型のインスタンスおよび任意のサブタイプのインスタンス、または少なくともサポートされるすべての型のインスタンスの作成が可能になります。

次の例で、PersonFactory.getOracleDataFactoryメソッドは、Javaインスタンスperson、studentまたはparttimestudentを戻すことにより、PERSON_Tオブジェクト、STUDENT_TオブジェクトおよびPARTTIMESTUDENT_Tオブジェクトを処理できるファクトリを戻します。

```
class PersonFactory implements OracleDataFactory
{
  static final PersonFactory _factory = new PersonFactory ();

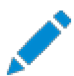
  public static OracleDataFactory getOracleDataFactory()
  {
    return _factory;
  }

  public OracleData create(Object jdbcValue, int sqlType) throws SQLException
  {
    STRUCT s = (STRUCT) jdbcValue;
    if (s.getSQLTypeName ().equals ("HR.PERSON_T"))
      return Person.getOracleDataFactory ().create (jdbcValue, sqlType);
    else if (s.getSQLTypeName ().equals ("HR.STUDENT_T"))
      return Student.getOracleDataFactory ().create(jdbcValue, sqlType);
    else if (s.getSQLTypeName ().equals ("HR.PARTTIMESTUDENT_T"))
      return ParttimeStudent.getOracleDataFactory ().create(jdbcValue, sqlType);
    else
      return null;
  }
}
```

次の例では、次のような表tabl1があることを想定しています。

```
CREATE TABLE tabl1 (idx NUMBER, person PERSON_T);
INSERT INTO tabl1 VALUES (1, PERSON_T (1000, 'HR', '100 Oracle Parkway'));
INSERT INTO tabl1 VALUES (2, STUDENT_T (1001, 'Peter', '200 Oracle Parkway', 101, 'CS'));
INSERT INTO tabl1 VALUES (3, PARTTIMESTUDENT_T (1002, 'David', '300 Oracle Parkway', 102, 'EE'));
```

13.4.2.2 型継承階層のためのSQLDataの使用について

java.sql.SQLDataインタフェースを実装するカスタマイズされたクラスを使用すると、データベース・オブジェクト・タイプ階層をミラー化できます。サブクラスのreadSQLおよびwriteSQLメソッドは、通常、対応するスーパークラスのメソッドをコールして、そのスーパークラスの属性の読取りまたは書込みを行ってから、サブクラス属性の読取りまたは書込みを行います。たとえば、PERSON_TおよびSTUDENT_Tに対するJavaクラスのマッピングは、次のようになります。

SQLDataを使用するPerson.java

次のコードは、SQLDataインタフェースを実装するPerson.javaクラスを示します。

```
import java.sql.*;
public class Person implements SQLData
{
```

```
  private String sql_type;
  public int ssn;
  public String name;
  public String address;
  public Person () {}
  public String getSQLTypeName() throws SQLException { return sql_type; }
  public void readSQL(SQLInput stream, String typeName) throws SQLException
  {
    sql_type = typeName;
    ssn = stream.readInt();
    name = stream.readString();
    address = stream.readString();
  }
  public void writeSQL(SQLOutput stream) throws SQLException
  {
    stream.writeInt (ssn);
    stream.writeString (name);
    stream.writeString (address);
  }
}
```

Student.javaを拡張するStudent.java

次のコードは、Person.javaクラスを拡張するStudent.javaクラスを示します。

```
import java.sql.*;
public class Student extends Person
{
  private String sql_type;
  public int deptid;
  public String major;
  public Student () { super(); }
  public String getSQLTypeName() throws SQLException { return sql_type; }
  public void readSQL(SQLInput stream, String typeName) throws SQLException
  {
    super.readSQL (stream, typeName);    // read supertype attributes
    sql_type = typeName;
    deptid = stream.readInt();
    major = stream.readString();
  }
  public void writeSQL(SQLOutput stream) throws SQLException
  {
    super.writeSQL (stream);        // write supertype
                                    // attributes
    stream.writeInt (deptid);
    stream.writeString (major);
  }
}
```

必須ではありませんが、SQLDataインタフェースを実装するカスタマイズされたクラスでは、データベース・オブジェクト型階層のミラー化をお薦めします。たとえば、スーパークラスを指定せずにStudentクラスを宣言できます。この場合、Studentには、PERSON_Tから継承された属性およびSTUDENT_Tによって宣言された属性を保持するフィールドが含まれます。

SQLDataを使用するStudent.java

次のコードは、Person.javaクラスを拡張せずに、SQLDataインタフェースを直接実装するStudent.javaクラスを示します。

```
import java.sql.*;
public class Student implements SQLData
{
  private String sql_type;
  public int ssn;
  public String name;
  public String address;
  public int deptid;
  public String major;
  public Student () {}
  public String getSQLTypeName() throws SQLException { return sql_type; }
  public void readSQL(SQLInput stream, String typeName) throws SQLException
  {
    sql_type = typeName;
    ssn = stream.readInt();
    name = stream.readString();
    address = stream.readString();
    deptid = stream.readInt();
    major = stream.readString();
  }
  public void writeSQL(SQLOutput stream) throws SQLException
  {
    stream.writeInt (ssn);
    stream.writeString (name);
    stream.writeString (address);
    stream.writeInt (deptid);
    stream.writeString (major);
  }
}
```

13.4.3 サブタイプ・オブジェクトの取得について

一般的なJDBCアプリケーションでは、サブタイプ・オブジェクトは次のいずれかとして戻されます。

- 問合せ結果
- PL/SQL OUTパラメータ
- 型属性

サブタイプの取得には、デフォルト・マッピング、SQLDataマッピングまたはOracleDataマッピングのいずれかを使用できます。

デフォルト・マッピングの使用

デフォルトで、データベース・オブジェクトは、oracle.jdbc.OracleStructインタフェースのインスタンスとして戻されます。このインスタンスは、宣言された型または宣言された型のサブタイプのオブジェクトを表します。OracleStructインタフェースがデータベースのサブタイプ・オブジェクトを表す場合、このインタフェースにはそのスーパータイプの属性と、サブタイプに定義された属性が含まれます。

Oracle JDBCドライバは、最も固有の型でデータベース・オブジェクトを戻します。JDBCアプリケーションは、OracleStructインタフェースのgetSQLTypeNameメソッドを使用して、STRUCTオブジェクトのSQL型を判断します。次のコードを参照してください。

```
// tab1.person column can store PERSON_T, STUDENT_T and PARTIMESTUDENT_T objects
ResultSet rset = stmt.executeQuery ("select person from tab1");
while (rset.next())
{
```

```
  oracle.sql.STRUCT s = (oracle.sql.STRUCT) rset.getObject(1);
  if (s != null)
    System.out.println (s.getSQLTypeName());    // print out the type name which
    // may be HR.PERSON_T, HR.STUDENT_T or HR.PARTTIMESTUDENT_T
}
```

SQLDataマッピングの使用

SQLDataマッピングの場合、JDBCドライバはSQLDataインタフェースを実装するクラスのインスタンスとして、データベース・オブジェクトを戻します。

データベース・オブジェクトの取得にSQLDataマッピングを使用するには、次の操作を実行します。

1. 目的のオブジェクト型のSQLDataインタフェースを実装するコンテナ・クラスを実装します。
2. 接続型マップに各Oracleオブジェクト型に対応するカスタムJava型を指定するエントリが移入されます。
3. SQLオブジェクト値へのアクセスには、getObjectメソッドを使用します。

   JDBCドライバによって、型マップと一致するエントリがチェックされます。一致するエントリが見つかると、ドライバはSQLDataインタフェースを実装するクラスのインスタンスとしてデータベース・オブジェクトを戻します。

次のコードは、カスタマイズされたSQLDataマッピングの全プロセスを示します。

```
// The JDBC application developer implements Person.java for PERSON_T,
// Student.java for STUDENT_T
// and ParttimeStudent.java for PARTTIMESTUDEN_T.
Connection conn = ...;  // make a JDBC connection
// obtains the connection typemap
java.util.Map map = conn.getTypeMap ();
// populate the type map
map.put ("HR.PERSON_T", Class.forName ("Person"));
map.put ("HR.STUDENT_T", Class.forName ("Student"));
map.put ("HR.PARTTIMESTUDENT_T", Class.forName ("ParttimeStudent"));
// tab1.person column can store PERSON_T, STUDENT_T and PARTTIMESTUDENT_T objects
ResultSet rset = stmt.executeQuery ("select person from tab1");
while (rset.next())
{
  // "s" is instance of Person, Student or ParttimeStudent
  Object s = rset.getObject(1);
  if (s != null)
  {
    if (s instanceof Person)
      System.out.println ("This is a Person");
    else if (s instanceof Student)
      System.out.println ("This is a Student");
    else if (s instanceof ParttimeStudent)
      System.out.pritnln ("This is a PartimeStudent");
    else
      System.out.println ("Unknown type");
  }
}
```

JDBCドライバにより、各コールの接続型マップと次のものが照合されます。

- java.sql.ResultSetおよびjava.sql.CallableStatementインタフェースのgetObjectメソッド
- java.sql.StructインタフェースのgetAttributeメソッド

- java.sql.ArrayインタフェースのgetArrayメソッド
- oracle.sql.REFインタフェースのgetValueメソッド

OracleDataマッピングの使用

OracleDataマッピングの場合、JDBCドライバはOracleDataインタフェースを実装するクラスのインスタンスとして、データベース・オブジェクトを戻します。

Oracle JDBCドライバは、Oracleオブジェクト型に対するJavaクラスのマッピングを認識する必要があります。Oracle JDBCドライバにこの情報を通知するには、次の2つの方法があります。

- JDBCアプリケーションは、getObject(int idx, OracleDataFactory f)メソッドを使用して、データベース・オブジェクトにアクセスします。getObjectメソッドの2番目のパラメータは、カスタマイズされたクラスを作成するファクトリ・クラスのインスタンスを指定します。getObjectメソッドはOracleResultSetおよびOracleCallableStatementインタフェースで使用できます。
- JDBCアプリケーションは、接続型マップに各Oracleオブジェクト型に対応するカスタムJava型を指定するエントリを移入します。Oracleオブジェクト値へのアクセスには、getObjectメソッドが使用されます。

2番目の方法では、標準getObjectメソッドが使用されます。次のサンプル・コードは、最初の方法を示します。

```
// tab1.person column can store both PERSON_T and STUDENT_T objects
ResultSet rset = stmt.executeQuery ("select person from tab1");
while (rset.next())
{
  Object s = rset.getObject(1, PersonFactory.getOracleDataFactory());
  if (s != null)
  {
    if (s instanceof Person)
      System.out.println ("This is a Person");
    else if (s instanceof Student)
      System.out.println ("This is a Student");
    else if (s instanceof ParttimeStudent)
      System.out.pritnln ("This is a PartimeStudent");
    else
      System.out.println ("Unknown type");
  }
}
```

13.4.4 サブタイプ・オブジェクトの作成

JDBCアプリケーションが、JDBCドライバを持つデータベース・サブタイプ・オブジェクトを作成する場合があります。これらのオブジェクトは、バインド変数としてデータベースに送られるか、JDBCアプリケーション内での情報交換に使用されます。

カスタマイズされたマッピングの場合、JDBCアプリケーションは、開発者が選択した方法に応じて、データベース・サブタイプ・オブジェクトを表すSQLDataまたはOracleDataベースのオブジェクトを作成します。デフォルト・マッピングの場合、データベース・サブタイプ・オブジェクトを表すSTRUCTオブジェクトをJDBCアプリケーションが作成します。スーパータイプから継承されたすべてのデータ・フィールド、およびサブタイプに定義されたすべてのフィールドに値が必要になります。次のコードでその方法を示します。

```
Connection conn = ...   // make a JDBC connection
...
Object[] attrs = {
  new Integer(1234), "HR", "500 Oracle Parkway", // data fields defined in
```

```
                                                    // PERSON_T
  new Integer(102), "CS",                           // data fields defined in
                                                    // STUDENT_T
  new Integer(4)                                    // data fields defined in
                                                    // PARTTIMESTUDENT_T
};
Struct s = conn.createStruct("HR.PARTTIMESTUDENT", attrs);
```

sは、PERSON_TとSTUDENT_Tから継承したデータ・フィールドおよびPARTTIMESTUDENT_Tで定義されているデータ・フィールドを使用して初期化されます。

13.4.5 サブタイプ・オブジェクトの送信

一般的なJDBCアプリケーションでは、データベース・オブジェクトを表すJavaオブジェクトは次のいずれかとしてデータベースに送られます。

- データ操作言語(DML)バインド変数
- PL/SQL INパラメータ
- オブジェクト型属性値

Javaオブジェクトは、STRUCTクラスのインスタンス、またはSQLDataまたはOracleDataインタフェースを実装するクラスのインスタンスです。Oracle JDBCドライバは、データベースのSQLエンジンが認識できるシリアライズされた形式にJavaオブジェクトを変換します。サブタイプ・オブジェクトは、標準のオブジェクトの場合と同じ方法でバインドされます。

13.4.6 サブタイプ・データ・フィールドへのアクセス

サブタイプ・データ・フィールドにアクセスするためのロジックはカスタマイズされたクラスの一部ですが、デフォルト・マッピングではこのロジックはJDBCアプリケーションそのものに定義されます。データベース・オブジェクトは、oracle.jdbc.OracleStructクラスのインスタンスとして戻されます。JDBCアプリケーションは、STRUCTクラスの次のいずれかのアクセス・メソッドをコールして、データ・フィールドにアクセスする必要があります。

- Object[] getAttribute()
- oracle.sql.Datum[] getOracleAttribute()

getAttributeメソッドのサブタイプ・データ・フィールド

java.sql.StructインタフェースのgetAttributeメソッドは、JDBC 2.0で、オブジェクトのデータ・フィールドにアクセスするために使用されます。このメソッドはjava.lang.Object配列を戻します。ここで、各配列要素はオブジェクト属性を表します。個々の要素型は、JDBC変換マトリックス内の対応する属性型を参照することで確認できます。たとえば、SQL NUMBER属性はjava.math.BigDecimalオブジェクトに変換されます。getAttributeメソッドは、オブジェクト型のスーパータイプで定義されるすべてのデータ・フィールドと、サブタイプで定義されるデータ・フィールドを戻します。最初にスーパータイプのデータ・フィールドがリストされ、次にサブタイプのデータ・フィールドがリストされます。

getOracleAttributeメソッドのサブタイプ・データ・フィールド

getOracleAttributeメソッドはOracleの拡張機能メソッドで、getAttributeメソッドよりも効率的です。getOracleAttributeメソッドは、データ・フィールドを保持するoracle.sql.Datum配列を戻します。oracle.sql.Datum配列の各要素は属性を表します。個々の要素型は、Oracle変換マトリックス内の対応する属性型を参照することで確認できま

す。たとえば、SQL NUMBER属性はoracle.sql.NUMBERオブジェクトに変換されます。getOracleAttributeメソッドは、そのオブジェクト型のスーパータイプに定義されたすべての属性と、そのサブタイプに定義された属性を戻します。最初にスーパータイプのデータ・フィールドがリストされ、次にサブタイプのデータ・フィールドがリストされます。

次のコードは、getAttributeメソッドの使用方法を示します。

```
// tab1.person column can store PERSON_T, STUDENT_T and PARTIMESTUDENT_T objects
ResultSet rset = stmt.executeQuery ("select person from tab1");
while (rset.next())
{
  oracle.sql.STRUCT s = (oracle.sql.STRUCT) rset.getObject(1);
  if (s != null)
  {
    String sqlname = s.getSQLTypeName();
    Object[] attrs = s.getAttribute();
    if (sqlname.equals ("HR.PERSON")
    {
      System.out.println ("ssn="+((BigDecimal)attrs[0]).intValue());
      System.out.println ("name="+((String)attrs[1]));
      System.out.println ("address="+((String)attrs[2]));
    }
    else if (sqlname.equals ("HR.STUDENT"))
    {
      System.out.println ("ssn="+((BigDecimal)attrs[0]).intValue());
      System.out.println ("name="+((String)attrs[1]));
      System.out.println ("address="+((String)attrs[2]));
      System.out.println ("deptid="+((BigDecimal)attrs[3]).intValue());
      System.out.println ("major="+((String)attrs[4]));
    }
    else if (sqlname.equals ("HR.PARTTIMESTUDENT"))
    {
      System.out.println ("ssn="+((BigDecimal)attrs[0]).intValue());
      System.out.println ("name="+((String)attrs[1]));
      System.out.println ("address="+((String)attrs[2]));
      System.out.println ("deptid="+((BigDecimal)attrs[3]).intValue());
      System.out.println ("major="+((String)attrs[4]));
      System.out.println ("numHours="+((BigDecimal)attrs[5]).intValue());
    }
    else
      throw new Exception ("Invalid type name: "+sqlname);
  }
}
rset.close ();
stmt.close ();
conn.close ();
```

13.4.7 継承メタデータ・メソッド

Oracle JDBCドライバでは、継承プロパティにアクセスするための一連のメタデータ・メソッドを提供します。継承メタデータ・メソッドは、oracle.sql.StructDescriptorクラスおよびoracle.jdbc.StructMetaDataクラスに定義されます。

StructMetaDataクラスは、サブタイプ属性の継承メタデータ・メソッドを提供します。StructDescriptorクラスのgetMetaDataメソッドは、その型のStructMetaDataのインスタンスを戻します。StructMetaDataクラスには、次の継承メタデータ・メソッドが含まれます。

13.5 オブジェクト型の記述について

Oracle JDBCには、構造化オブジェクト型の属性の名前と型に関する情報を取得する機能があります。これは考え方として、結果セットから列の名前と型に関する情報を取得するのに類似しており、実際ほとんど同一のメソッドを使用します。

この項の内容は次のとおりです。

- オブジェクト・メタデータの取出し機能
- オブジェクト・メタデータの取得

13.5.1 オブジェクト・メタデータの取出し機能

oracle.sql.StructDescriptorクラスには、構造化オブジェクト型に関するメタデータを取り出す機能が含まれています。StructDescriptorクラスには、結果セット・オブジェクトで使用可能な標準getMetaDataメソッドと同じ機能を持つgetMetaDataメソッドがあります。このメソッドは、属性名や型など、属性情報の集合を戻します。このメソッドをStructDescriptorオブジェクトでコールして、そのStructDescriptorオブジェクトによって記述されているOracleオブジェクト型に関するメタデータを取り出します。

StructDescriptorクラスのgetMetaDataメソッドのシグネチャは、標準ResultSetインタフェースのgetMetaDataに指定されているシグネチャと同じです。シグネチャは次のとおりです。

```
ResultSetMetaData getMetaData() throws SQLException
```

ただし、このメソッドは、実際にはoracle.jdbc.StructMetaDataのインスタンスを戻します。このクラスは、標準java.sql.ResultSetMetaDataインタフェースが結果セット・メタデータのサポートを指定するのと同じ方法で、構造化オブジェクト・メタデータをサポートします。

次のメソッドもStructMetaDataでサポートされます。

```
String getOracleColumnClassName(int column) throws SQLException
```

このメソッドは、指定した属性の値を取り出すためにOracleResultSetインタフェースのgetOracleObjectメソッドがコールされた場合にインスタンスが作成されるoracle.sql.Datumサブクラスの完全修飾名を戻します。たとえば、oracle.sql.NUMBERです。

getOracleColumnClassNameメソッドを使用するには、getMetaDataメソッドで戻されたResultSetMetaDataオブジェクトをStructMetaDataにキャストする必要があります。

-



ノート:

前述のメソッド・シグネチャの column は、誤解を招く表現です。column の値に 4 を指定すると、オブジェクトの 4 番目の属性を指すことになります。

13.5.2 オブジェクト・メタデータの取得

次のステップを使用して、構造化オブジェクト型のメタデータを取り出します。

1. 該当する構造化オブジェクト型を記述するStructDescriptorインスタンスを作成するか取り出します。
2. StructDescriptorインスタンスのgetMetaDataメソッドをコールします。
3. 適切なメタデータgetterメソッド(getColumnName、getColumnTypeおよびgetColumnTypeName)をコールします。



ノート:

構造化オブジェクトの属性自身が構造化オブジェクトの場合、ステップ 1 から 3 を繰り返します。

例13-1 例

次のメソッドは、構造化オブジェクト型の属性情報を取り出す方法を示します。この例には、StructDescriptorインスタンスを作成する最初のステップが含まれています。

```
//
// Print out the ADT's attribute names and types
//
void getAttributeInfo (Connection conn, String type_name) throws SQLException
{

  // get the type descriptor
  StructDescriptor desc = StructDescriptor.createDescriptor (type_name, conn);
  // get type metadata
  ResultSetMetaData md = desc.getMetaData ();
  // get # of attrs of this type
  int numAttrs = desc.length ();
  // temporary buffers
  String attr_name;
  int attr_type;
  String attr_typeName;
  System.out.println ("Attributes of "+type_name+" :");
  for (int i=0; i<numAttrs; i++)
  {
    attr_name = md.getColumnName (i+1);
    attr_type = md.getColumnType (i+1);
    System.out.println (" index"+(i+1)+" name="+attr_name+" type="+attr_type);
    // drill down nested object
    if (attrType == OracleTypes.STRUCT)
    {
      attr_typeName = md.getColumnTypeName (i+1);
      // recursive calls to print out nested object metadata
      getAttributeInfo (conn, attr_typeName);
    }
  }
}
```

14 LOBとBFILEの操作

この章では、データ・インタフェースまたはロケータ・インタフェースのいずれかを使用して、ラージ・オブジェクト(LOB)にアクセスし、操作するために、Java Database Connectivity(JDBC)を使用する方法について説明します。

以前のリリースでは、Oracle JDBCドライバには、Oracle Databaseで多くの操作を実行するために、標準JDBC型に対するOracle拡張機能が必要でした。JDBC 3.0ではOracle拡張機能を使用する必要が少なくなり、JDBC 4.0ではこの制限はほとんどなくなりました。java.sqlおよびjavax.sqlパッケージについてはJavasoft Javadocを、Oracle拡張機能の詳細はOracle JDBC Javadocを参照してください。

この章の構成は、次のとおりです。

- LOBデータ型
- Oracle SecureFiles
- LOBのデータ・インタフェース
- LOBロケータ・インタフェース
- 一時LOBの使用について
- OpenおよびCloseメソッドによる通常のLOBのオープンについて
- BFILEの操作について

ノート:

- Oracle Database 12c リリース 1 (12.1)では、Oracle JDBC ドライバで JDBC 4.0 java.sql.NClob インタフェースがサポートされています。
- Oracle Database 10g では、Oracle JDBC ドライバで JDBC 3.0 java.sql.Clob および java.sql.Blob インタフェースがサポートされています。Oracle Database の旧リリースの oracle.sql.CLOB および oracle.sql.BLOB で作成された Oracle 拡張機能は必要でなくなり、非推奨となりました。アプリケーションを標準 JDBC 3.0 インタフェースに移植する必要があります。
- Oracle Database 10g より前は、LOB の最大サイズが 2^{32} バイトでした。この制限は、Oracle Database 10g で削除されており、現在の最大サイズは使用可能な物理記憶域のサイズと同じです。Java LOB アプリケーション・プログラミング・インタフェース(API)は変更されていません。

14.1 LOBデータ型

Oracle Database 10gより前には、LOBの最大サイズが2^32バイトでした。この制限は、Oracle Database 10gで削除されており、現在の最大サイズは使用可能な物理記憶域のサイズと同じです。

Oracle Databaseでは、次の4つのLOBデータ型をサポートします。

- バイナリ・ラージ・オブジェクト(BLOB)

このデータ型は、非構造バイナリ・データに使用されます。

- キャラクタ・ラージ・オブジェクト(CLOB)

  このデータ型は、文字データに使用されます。

- 各国語キャラクタ・ラージ・オブジェクト(NCLOB)

  このデータ型は、各国語文字データに使用されます。

- BFILE

  このデータ型は、データベース表領域外の、オペレーティング・システム・ファイルに格納されるラージ・バイナリ・データ・オブジェクトに使用されます。

BLOB、CLOBおよびNCLOBは、データベース表領域に永続的に格納され、これらのデータ型に対する操作はすべて、トランザクション制御の下で実行されます。

BFILEは、Oracle独自のデータ型で、3次ストレージ・デバイス(ハード・ディスク、ネットワーク・マウント・ファイルシステム、CD-ROM、PhotoCDおよびDVDなど)上のデータベース表領域外にあるデータに読取り専用アクセスが可能です。BFILEデータは、トランザクション制御下にはなく、データベース・バックアップでは保存されません。

PL/SQL言語では、LOBデータ型がサポートされ、JDBCインタフェースにより、PL/SQLのプロシージャまたはファンクションへのINパラメータの引渡しや、OUTパラメータや戻り値の取出しが可能です。PL/SQLでは、LOBを含むすべてのデータ型に値セマンティクスを使用しますが、参照セマンティクスはBFILEに対してのみです。

14.2 Oracle SecureFiles

Oracle Database 11g リリース1 (11.1)では、LOBのまったく新しい記憶域としてOracle SecureFilesが導入されました。

Oracle SecureFilesの次の機能は、既存のAPIを介してJDBCプログラムで透過的に使用可能です。

- SecureFile圧縮により、データを圧縮してディスク容量を節約できます。
- SecureFile暗号化では、暗号化されたデータのランダムな読取りおよび書込みを可能にする新しい暗号化ツールが導入されました。
- 重複では、Oracle Databaseの重複LOBデータが自動検出され、データを一部のみ保存することで容量が節約されます。
- LOBデータ・パス・最適化には、記憶域レイヤーの上の論理キャッシュと新しいキャッシング・モードが含まれます。
- 高いパフォーマンスの容量管理。

`setLobOptions`および`getLobOptions` APIについては『Oracle Database PL/SQLパッケージおよびタイプ・リファレンス』を参照してください。コール可能文を使用してJDBCからアクセスすることもできます。

Oracle SecureFilesの次の機能は、既存のAPIに対する更新により、データベース内で実装されます。

- isSecureFileメソッド
- Oracle SecureFilesのゼロコピーI/O

isSecureFileメソッド

BLOBまたはCLOBデータによってOracle SecureFile記憶域が使用されているかどうかをチェックできます。このチェックを行うには、

oracle.jdbc.OracleBlobクラスまたはoracle.jdbc.OracleClobクラスの次のメソッドを使用します。

```
public boolean isSecureFile() throws SQLException
```

このメソッドからtrueが戻された場合、データはSecureFile記憶域を使用しています。

Oracle SecureFilesのゼロコピーI/O

Oracle Database 12cリリース2 (12.2) JDBCドライバの発表により、Oracle SecureFiles操作のパフォーマンスが大幅に向上しました。これは、Oracle Net Servicesで、バッファ管理を改善するためにゼロコピーI/Oフレームワークが使用されるようになったためです。

Oracle Database 11gリリース2では、新しい接続プロパティoracle.net.useZeroCopyIOが導入されました。このプロパティは、ゼロコピーI/Oプロトコルを有効または無効にするために使用できます。この接続プロパティは、定数OracleConnection.CONNECTION_PROPERTY_THIN_NET_USE_ZERO_COPY_IOとして定義されています。ゼロコピーI/Oフレームワークを無効にする場合は、この接続プロパティの値をfalseに設定してください。この接続プロパティのデフォルト値はtrueです。

14.3 LOBのデータ・インタフェース

この節では、以下のトピックについて説明します。

- 効率的なメカニズム
- 入力
- 出力
- CallableSatementとIN OUTパラメータ
- サイズの制限

14.3.1 効率的なメカニズム

Oracle Database 12cリリース1 (12.1) JDBCドライバでは、LOBの内容全体の書込みと読取りができる効率的なメカニズムを提供しています。これは、データ・インタフェースと呼ばれます。このデータ・インタフェースでは、標準JDBCメソッド(getStringやsetBytesなど)を使用して、LOBデータの読取りと書込みを行います。多くの場合、これによってコーディングが簡素化され、処理が速くなります。標準のjava.sql.Blob、java.sql.Clobおよびjava.sql.NClobインタフェースと異なり、これにはランダム・アクセス機能はなく、LOBロケータを使用していないので、2147483648要素を超えるデータにはアクセスできません。

14.3.2 入力

Oracle Database 12cリリース1 (12.2)では、PreparedStatementのsetBytes、setBinaryStream、setString、setCharacterStreamおよびsetAsciiStreamメソッドが、BLOB、CLOBおよびNCLOBターゲット列を処理する機能を強化するために拡張されています。データの長さが不明である場合は、パフォーマンスを向上させるために、データの長さをパラメータとして受け入れるsetBinaryStreamまたはsetCharacterStreamメソッドのバージョンを使用します。



ノート:

この強化は、読取り専用の BFILE データには影響がありません。

JDBC Oracle Call Interface(OCI)とThinドライバには、byte配列やStringのサイズに制限はなく、ストリーム・ファンクションに指定された長さには、Java言語の制限を除けば、制限はありません。



ノート:

Java では、配列サイズは正の Java int または 2147483648 要素に制限されています。

サーバー側内部ドライバについては、現在INSERT文などのSQL文の操作に32767バイトという制限があります。この制限はPL/SQL文には適用されません。INSERT文については、次のようにPL/SQLブロックにラップするという簡単な回避方法があります。

```
BEGIN
 INSERT id, c INTO clob_tab VALUES(?,?);
END;
```

大量のデータに対する次のような入力モードの自動切替えは留意する必要があります。

- 次の3つの入力モードがあります。
  - 直接バインディング

    このバインディングは、サイズに制限がありますが、最も効率的です。すべての入力列のデータを、サーバーに送信するデータのブロックにインラインで配置します。バッチの複数回実行を含め、すべてのデータは1回のネットワーク操作で送信されます。
  - ストリーム・バインディング

    このバインディングでは、データが最後に配置されます。バッチ・サイズは1つに制限され、完了に複数回のラウンドトリップが必要になる場合があります。
  - LOBバインディング

    このバインディングでは、一時LOBを作成し、データをLOBにコピーして、LOBロケータをバインドします。一時LOBは、実行後、自動的に解放されます。一時LOBを作成してから、LOBに書き込む操作には、複数回のラウンドトリップが必要です。ロケータの入力はバッチ処理されます。
- SQL文の場合
  - setBytesおよびsetBinaryStreamメソッドでは、32767バイト未満のデータには直接バインディングを使用します。
  - setBytesおよびsetBinaryStreamメソッドでは、32767バイトを超えるデータにはストリーム・バインディングを使用します。
  - JDBC 4.0では、新しい形式のsetAsciiStream、setBinaryStreamおよびsetCharacterStreamメソッドが導入されました。メソッドが長さとして長い引数を取る形式では、2147483648を超える長さにはLOBバイ

ンディングを使用します。長さを指定しない形式では、常にLOBバインディングを使用します。

- setString、setCharacterStreamおよびsetAsciiStreamメソッドでは、32767文字未満のデータには直接バインディングを使用します。
- setString、setCharacterStreamおよびsetAsciiStreamメソッドでは、32766文字を超えるデータにはストリーム・バインディングを使用します。
- 新しい形式のsetCharacterStreamメソッドは、長さの引数としてlong引数を取り、JDBC 4.0では、2147483647を超える長さにLOBバインディングを使用します。長さを指定しない形式では、常にLOBバインディングを使用します。

- PL/SQL文
    - setBytesおよびsetBinaryStreamメソッドでは、32767バイト未満のデータには直接バインディングを使用します。

ノート:

基礎となるデータベースが Oracle Database リリース 10.x の場合、12c リリース 1 (12.1) JDBC ドライバを使用していても、データ・サイズ制限は 32512 バイトです。

- setBytesおよびsetBinaryStreamメソッドでは、32766バイトを超えるデータにはLOBバインディングを使用します。
- setString、setCharacterStreamおよびsetAsciiStreamメソッドでは、データベース文字セットで32767バイト未満のデータには直接バインディングを使用します。

ノート:

基礎となるデータベースが Oracle Database リリース 10.x の場合、12c リリース 1 (12.1) JDBC ドライバを使用していても、データ・サイズ制限は 32512 バイトです。

- setString、setCharacterStreamおよびsetAsciiStreamメソッドでは、データベース文字セットで32766バイトを超えるデータにはLOBバインディングを使用します。

大量データの入力モードの自動切替えは、一部のプログラムに影響を与えます。以前は、32766文字を超えるString値にsetStringを使用しようとすると、ORA-17157エラーが発生していました。現在は、ターゲット・パラメータの型に応じて、文の実行中にエラーが発生する場合と、操作が成功する場合があります。

もう1つの影響は、自動切替えの結果、パラメータの型の変更に適合させるために追加のサーバー側解析が実行される場合があることです。その結果、文が繰り返し実行されてデータ・サイズが制限の上下に変動すると、パフォーマンスに影響が生じます。ストリーム・モードへの切替えはバッチ処理にも影響を与えます。

LOBへの変換の強制

oracle.jdbc.OraclePreparedStatementインタフェースにあるsetBytesForBlobおよびsetStringForClobメソッドでは、すべてのデータ・サイズにLOBバインディングを使用します。

Oracle Database 10gリリース1のSetBigStringTryClob接続プロパティは、使用されなくなるか不要になりました。

14.3.3 出力

ResultSetおよびCallableStatementのgetBytes、getBinaryStream、getString、getCharacterStreamおよびgetAsciiStreamメソッドは、BLOB、CLOBおよびBFILE列またはOUTパラメータと連係するよう拡張されています。これらのメソッドは、2147483648未満の長さのすべてのLOBに機能します。



ノート:

getString メソッドと getNString メソッドを使用して BLOB 列値を取得することはできません。

データ・インタフェースはドライバ内でのLOBロケータへのアクセスにより動作し、アプリケーション・プログラミングに対して透過的に行われます。サポートされているすべてのバージョンのデータベース(Oracle Database 10.1.x以降)で機能します。Oracle Database 11gリリース1以降のバージョンでは、LOBのプリフェッチを使用して、必要なデータベース・ラウンドトリップを減らすかなくすことができます。

defineColumnTypeメソッドを使用して、BFILEデータの読取り、およびBLOBまたはCLOBデータの読取りと書込みを行うことができます。読み取るには、列に対してdefineColumnType(nn, Types.LONGVARBINARY)またはdefineColumnType(nn,Types.LONGVARCHAR)を使用します。これによって、ダイレクト・ストリームがデータに対してLONG RAWまたはLONG列であるかのように作成されます。この技術はOracle Database 10gリリース1(10.1)以降に限定されます。

関連トピック

- JDK 6での各国語文字セット用の新しいメソッド
- LOBロケータ・インタフェース

14.3.4 CallableSatementとIN OUTパラメータ

PL/SQLの要件は、IN OUTパラメータの入力と出力に同じJava型を使用する必要があります。この章で説明した拡張機能による型の自動切替えが原因で、これに関して問題が発生する場合があります。

ストアド・プロシージャのIN OUT CLOBパラメータがあり、このパラメータの値を設定するためにsetStringを使用するとします。INおよびOUTパラメータについて、バインドを同じ型にする必要があります。データ・サイズが判明していないかぎり、入力モードの自動切替えによって問題が発生します。たとえば、入力データおよび出力データの両方が32766バイトを超えないことが判明している場合は、入力パラメータにsetStringメソッドを使用し、OUTパラメータをTypes.VARCHARとして登録し、出力パラメータにgetStringメソッドを使用できます。

よりよい解決方法は、ストアド・プロシージャを変更してINパラメータとOUTパラメータを個別にすることです。たとえば、次のストアド・プロシージャがあったとします。

```
CREATE PROCEDURE clob_proc( c IN OUT CLOB );
```

これを次のように変更します。

```
CREATE PROCEDURE clob_proc( c_in IN CLOB, c_out OUT CLOB );
```

もう1つの回避方法は、コンテナ・ブロックを使用してコールすることです。次のようにclob_procプロシージャをJava文字列でラップすると、prepareCall文に使用できます。

```
"DECLARE c_temp; BEGIN c_temp := ?; clob_proc( c_temp); ? := c_temp; END;"
```

どちらの場合でも、最初のパラメータにsetStringメソッドを使用し、2番目のパラメータにregisterOutParameterメソッドとTypes.CLOBを使用できます。

14.3.5 サイズの制限

非常に大きなbyte配列またはStringが作成されるために、Javaメモリー管理システムのパフォーマンスに影響が生じることに注意してください。Java仮想マシン(JVM)ベンダーが提供する情報を参照して、大量のデータ要素がメモリー管理に与える影響を理解し、かわりにストリーム・インタフェースを使用することを検討してください。

14.4 LOBロケータ・インタフェース

ロケータは少量データ構造体で、LOBの実際のデータへのアクセスに使用できる情報が格納されます。データベース表で、ロケータは表に直接格納されるのに対し、データはその表または別の記憶域に格納できます。大型のLOBの場合は、別の表領域を使用するのが一般的です。

JDBC 4.0では、LOBは、java.sql.Blob、java.sql.Clobおよびjava.sql.NClobインタフェースを使用して読込みまたは書込みを行います。これにより、LOBのデータにはランダム・アクセスできます。

Oracle実装クラスのoracle.sql.BLOB、oracle.sql.CLOBおよびoracle.sql.NCLOBではロケータを格納し、それを使用してデータにアクセスします。oracle.sql.BLOBクラスとoracle.sql.CLOBクラスは、それぞれjava.sql.Blobインタフェースとjava.sql.Clobインタフェースを実装します。ojdbc6.jarでは、oracle.sql.NCLOBはjava.sql.NClobを実装しますが、ojdbc5.jarでは、java.sql.Clobインタフェースを実装します。



ノート:

Oracle Database 12c リリース 1 (12.1)以降、oracle.sql.BLOB クラスおよび oracle.sql.CLOB クラスは非推奨となり、oracle.jdbc.OracleBlob インタフェースおよび oracle.jdbc.OracleClob インタフェースに置き換えられています。標準互換性には(可能であれば)java.sql パッケージの使用可能なメソッドを使用し、Oracle 固有の拡張機能には oracle.jdbc パッケージの使用可能なメソッドを使用することをお薦めします。これらのインタフェースの詳細は、MoS ノート 1364193.1 を参照してください。

Oracle Database 12cリリース1 (12.1)では、Oracle JDBCドライバは、ojdbc6.jarおよびojdbc7.jarでJDBC 4.0 java.sql.NClobインタフェースをサポートしています。これらのファイルは、それぞれ、JDK 6 (JRE 6とともに使用する必要があります)とJDK 7 (JRE 7とともに使用する必要があります)でコンパイルされます。

これに対し、oracle.sql.BFILEはOracle拡張機能で、対応するjava.sqlインタフェースはありません。

関連項目:

詳細は、JDBC Javadocを参照してください。

LOBプリフェッチ

Oracle Database 12cリリース1 (12.1) JDBCドライバの場合、通常のフェッチ操作中にロケータとともにLOBデータの始めをプリフェッチするのと同様に、LOBの長さやチャンク・サイズなどのメタデータをプリフェッチすることで、ラウンドトリップの回数を減らします。LOB列を選択して結果セットに入れる場合、ロケータがフェッチされると、データの一部またはすべてをクライアントにプリフェッチできます。ロケータからフェッチするまで、前述の操作を保留することによって、最初のラウンドトリップを行わずにデータを取得します。



ノート:

LOB のプリフェッチは LOB データのサイズに反比例しています。つまり、プリフェッチの利点は小さな LOB の場合は大きく、大きな LOB の場合は小さくなります。

プリフェッチ・サイズは、BLOBの場合はバイトで、CLOBの場合は文字数で指定されます。接続プロパティoracle.jdbc.defaultLobPrefetchSizeを設定することで指定できます。このプロパティの値は、次の2つの方法でオーバーライドできます。

- 文レベル: oracle.jdbc.OracleStatement.setLobPrefetchSize(int)メソッドを使用
- 列レベル: 長さを引数として取るdefineColumnTypeメソッドの形式を使用

デフォルトのプリフェッチ・サイズは4000です。



ノート:

大型 LOB のプリフェッチ・サイズを、行プリフェッチ・サイズや多数の LOB 列と一緒に設定するときには、予想されるメモリー消費に注意してください。

関連項目:

詳細は、JDBC Javadocを参照してください。

JDBC 4.0の新しいLOB API

Oracle Database 11gリリース1では、java.sql.NClobインタフェースが導入されました。Oracleドライバが、ojdbc6.jarとojdbc7.jarでoracle.sql.NCLOBおよびjava.sql.NCLOBインタフェースを実装します。

Oracleドライバにより、新しいファクトリ・メソッドcreateBlob、createClobおよびcreateNClobが、一時LOBを作成するためにjava.sql.Connectionインタフェースで実装されます。

JDK 6から、java.sql.Blob、java.sql.Clobおよびjava.sql.NClobインタフェースに、LOBを解放し、関連付けられたリソースを解放する新しいfreeメソッドが導入されました。Oracleドライバはこのメソッドを使用して、一時LOBを解放します。

14.5 一時LOBの使用について

一時LOBは、一時データの格納に使用できます。データは、通常の表領域ではなく、一時表領域に格納されます。不要になった一時LOBは解放する必要があります。解放しない場合、LOBによって使用されている一時表領域は再生されません。

一時LOBは、表に挿入できます。挿入すると、LOBの永続的なコピーが作成されて格納されます。



ノート:

一時 LOB を挿入する方が便利な場合があります。たとえば、LOB データが相対的に小さい場合は、空のロケータを取り出すデータベース・ラウンドトリップのコストに比べて、データをコピーするオーバーヘッドが小さくなります。データは、最初はサーバーの一時表領域に格納され、その後永続記憶域に移されるためです。

一時LOBは、oracle.sql.BLOBクラスとoracle.sql.CLOBクラスの両方で定義されているstaticメソッドcreateTemporaryで作成します。一時LOBを解放するには、freeTemporaryメソッドを使用します。

JDBC 4.0で利用できる接続ファクトリ・メソッドを使用することにより、一時LOB/CLOBまたはNCLOBを作成することもできます。

isTemporaryメソッドをコールすることで、LOBが一時LOBかどうかをテストできます。LOBがcreateTemporaryメソッドをコールして作成された場合、isTemporaryメソッドは、trueを戻し、それ以外の場合はfalseを戻します。

一時LOBを解放するには、freeTemporaryメソッドをコールします。セッションまたはコールを終了する前に、一時LOBを解放してください。



ノート:

- 一時 LOB を解放しない場合は、データベース内でその LOB によって使用されている記憶域を使用できないようにします。頻繁に一時 LOB を解放しないと、一時表領域が使用できない LOB 記憶域でいっぱいになってしまいます。
- 一時 LOB 列のある ReultSet からデータをフェッチする場合は、getString または getBytes のかわりに、getClob または getBlob メソッドを使用してください。
- java.sql.Blob、java.sql.Clob および java.sql.NClob インタフェースにある JDBC 4.0 の free メソッドは、freeTemporary メソッドに優先します。

関連トピック

- LOBの作成

14.6 OpenおよびCloseメソッドによる通常のLOBのオープンについて

この項では、LOBのオープンとクローズの方法について説明します。この機能は、oracle.sql.BLOBおよびoracle.sql.CLOBインタフェースの次のメソッドを使用して、JDBC実装されます。

- void open (int mode)
- void close()
- boolean isOpen()



ノート:

- Oracle Database 12c リリース 1 (12.1)以降、oracle.sql.BLOB クラスおよび oracle.sql.CLOB クラスは非推奨となり、oracle.jdbc.OracleBlob インタフェースおよび oracle.jdbc.OracleClob インタフェースに置き換えられています。標準互換性には(可能であれば)java.sql パッケージの使用可能なメソッドを使用し、Oracle 固有の拡張機能には oracle.jdbc パッケージの使用可能なメソッドを使用することをお薦めします。これらのインタフェースの詳細は、MoS ノート 1364193.1 を参照してください。
- LOB は必ずしもオープンおよびクローズする必要はありません。パフォーマンス上の理由から、オープンとクローズを選択できます。

Open/Closeコール操作内部でLOB操作をラップしない場合、LOBは変更のたびに暗黙的にオープンおよびクローズされ、ドメイン索引上でトリガーが起動されます。この場合、LOBを変更するとただちに、LOBのドメイン索引がすべて更新されることに注意してください。したがって、ドメインLOB索引は常に有効となり、同じトランザクション内でいつでも使用できます。

Open/Closeコール操作の内部でLOB操作をラップすると、LOBが変更されるたびにトリガーが起動されることはありません。かわりに、Closeがコールされた時点で、ドメイン索引に関するトリガーが起動します。たとえば、closeメソッドをコールするまでドメイン索引が更新されないようにアプリケーションを設計できます。ただし、これは、LOB上のドメイン索引がOpen/Closeコールの間は無効であることを意味します。

LOBを開くには、openメソッドまたはopen(int)メソッドをコールします。その後、そのLOBに関連付けられたトリガーを起動することなく、LOBの読取りまたは書込み操作を実行できます。LOBへのアクセスが終了後、closeメソッドをコールしてLOBをクローズします。LOBをクローズすると、そのLOBに関連付けられたトリガーが起動します。

isOpenメソッドをコールすると、LOBがオープンまたはクローズしているかどうかを確認できます。open(int)メソッドをコールしてLOBをオープンした場合は、引数の値がoracle.sql.BLOBクラスおよびoracle.sql.CLOBクラスで定義されたMODE_READONLYまたはMODE_READWRITEである必要があります。MODE_READONLYを使用してLOBをオープンした場合は、そのLOBに書込みを試みるとSQL例外が発生します。



ノート:

- トランザクションでオープンしたすべての LOB をクローズする前に、そのトランザクションをコミットしようとすると、エラーが発生します。オープンしている LOB のオープン状態は破棄され、トランザクションは正常にコミットされます。このため、トランザクション内で LOB および非 LOB データに対して行われた変更はすべてコミットされますが、ドメイン索引に関するトリガーは確実には実行されません。
- open および close メソッドは、通常の LOB に対してのみ適用されます。close メソッドは、一時 LOB に使用される free または freeTemporary メソッドとは異なります。free および freeTemporary メソッドは、記憶域を解放し、LOB を使用不可にします。他方、close メソッドは、LOB での変更が完了したので

トリガーを起動し索引を更新する必要があることを、データベースに指示します。LOB は、close メソッドへのコール後もまだ使用可能です。

14.7 BFILEの操作について

この項では、ファイル・ロケータを使用してBFILEからデータを読み取る方法を説明します。この項の内容は次のとおりです。

- BFILEロケータの取出し
- BFILESへの書込み

BFILEロケータの取出し

BFILEデータ型とoracle.sql.BFILEクラスは、Oracle独自のものです。したがって、これらには標準インタフェースはありません。この型のデータには、Oracle拡張機能を使用する必要があります。

標準JDBC結果セットまたはBFILEロケータを含むコール可能文オブジェクトがある場合は、標準結果セットのgetObject メソッドを使用して、ロケータにアクセスできます。このメソッドは、oracle.sql.BFILEオブジェクトを戻します。

また、結果セットをOracleResultSetにキャストするか、コール可能文をOracleCallableStatementにキャストし、getOracleObjectまたはgetBFILEメソッドを使用することにより、ロケータにアクセスすることもできます。



ノート:

getObject または getOracleObject メソッドを使用している場合は、必要に応じて出力をキャストしてください。

ロケータがあれば、oracle.sql.BFILE内のAPIによりBFILEデータにアクセスできます。これらのAPIは、java.sql.BLOBインタフェースの読取りメソッドに似ています。

BFILEへの書込み

BFILEの内容にデータは書き込めませんが、oracle.sql.BFILEのインスタンスをSQL文またはPL/SQLプロシージャへの入力として使用できます。これは次のいずれかを実行することで行えます。

- 標準のsetObjectメソッドを使用します。
- 文をOraclePreparedStatementまたはOracleCallableStatementにキャストし、setOracleObjectまたはsetBFILEメソッドを使用します。これらのメソッドは、パラメータ索引およびoracle.sql.BFILEオブジェクトを入力として取ります。



ノート:

- BFILE 用の標準 java.sql インタフェースはありません。
- getBFILE メソッドを OracleResultSet および OracleCallableStatement インタフェースで使用して、oracle.sql.BFILE オブジェクトを取り出します。OraclePreparedStatement および

OracleCallableStatement インタフェースの setBFILE メソッドは、引数として oracle.sql.BFILE オブジェクトを受け入れます。これらのメソッドを使用して BFILE に書き込みます。

- getBfile、setBfile および updateBfile メソッドのかわりに、getBFILE、setBFILE および updateBFILE メソッドを使用することをお薦めします。たとえば、setBfile メソッドのかわりに setBFILE メソッドを使用します。

BFILEは読取り専用です。データの本体はオペレーティング・システム(OS)のファイル・システムにあり、OSのツールとコマンドを使用しないと書き込めません。JDBCから、またはSQLを実行する別の方法を使用して、適切なSQL文を実行することにより、既存の外部ファイル用のBFILEを作成できます。しかし、BFILEが参照するOSファイルSQLまたはJDBCでは作成できません。これは、サーバー・ファイル・システムにアクセスするプロセスによって外部的にしか作成されません。



ノート:

旧版マニュアルのこの章で示されているサンプル・コードは、OTN でダウンロードできるサンプル・コードを参照するために削除されました。

15 Oracleオブジェクト参照の使用

この章では、オブジェクト参照へのアクセスおよび操作を行う、標準Java Database Connectivity(JDBC)について説明します。

この項では、次の項目について説明します。

- オブジェクト参照用Oracle拡張機能
- オブジェクト参照の取出しと引渡し
- オブジェクト値に対する、オブジェクト参照を介したアクセスと更新

15.1 オブジェクト参照用Oracle拡張機能

Oracleでは、データベース・オブジェクトへの参照を使用できます。Oracle JDBCでは、次のオブジェクト参照がサポートされます。

- SELECT句の列
- INまたはOUTバインド変数
- Oracleオブジェクトの属性
- コレクション型オブジェクトの要素

SQLでは、オブジェクト参照(REF)は厳密に型指定されています。たとえば、EMPLOYEEオブジェクトへの参照は、REFのみではなく、EMPLOYEE REFとして定義されます。

オブジェクト参照を選択する場合は、オブジェクト本体でなく、オブジェクトへのポインタのみを取得することに注意してください。移植性を重視して、参照をjava.sql.Refインスタンスとしてインスタンス化するか、事前に作成したカスタムJavaクラスのインスタンスとしてインスタンス化して強い型指定の利点を取るか、両方の選択肢があります。オブジェクト参照のために使用されるカスタムJavaクラスは、カスタム参照クラスとして参照されるため、oracle.jdbc.OracleDataインタフェースを実装する必要があります。

結果セットまたはコール可能文オブジェクトを介してREFインスタンスを取得し、プリペアド文またはコール可能文オブジェクトを介して更新されたREFにインスタンスをデータベースに戻すことができます。REFクラスには、基礎となるオブジェクト属性値を取得および設定し、基礎となるオブジェクトのSQLベース型名を取得する機能があります。

カスタム参照クラスには、これと同じ機能が含まれている他に、強い型指定が適用されるという利点があります。この強い型指定により、実行時まで検出できないコーディング・エラーを、コンパイル時に発見できます。



ノート:

- カスタム・オブジェクト・クラスに対して oracle.jdbc.OracleData インタフェースを使用する場合は、対応するカスタム参照クラスにも OracleData を使用します。ただし、カスタム・オブジェクト・クラスに対して標準 java.sql.SQLData インタフェースを使用する場合、参照に使用できるのは、弱い Java 型のみです。

SQLData インタフェースは、SQL オブジェクト型のマッピング専用です。

- JDBC アプリケーションで REF オブジェクトを作成して取り出すことができるのは、SQL 文を実行した場合のみです。REF オブジェクトを作成して取り出すための JDBC 固有の機能はありません。
- 配列はオブジェクトの同様に構造化型ですが、参照できません。

15.2 オブジェクト参照の取出しと引渡し

この項では、オブジェクト参照の取出しと引渡しを行うJDBC機能を説明します。内容は次のとおりです。

- 結果セットからのオブジェクト参照の取出し
- オブジェクト参照のコール可能文からの取出し
- オブジェクト参照のプリペアド文への引渡し

15.2.1 結果セットからのオブジェクト参照の取出し

オブジェクト参照を取り出す方法を示すために、次の例では、最初に、Oracleオブジェクト型ADDRESSを定義し、次にPEOPLE表でこのオブジェクト型を参照します。

```
create type ADDRESS as object
   (street_name      VARCHAR2(30),
    house_no         NUMBER);
create table PEOPLE
    (col1 VARCHAR2(30),
     col2 NUMBER,
     col3 REF ADDRESS);
```

ADDRESSオブジェクト型には、street nameとhouse numberという2つの属性があります。PEOPLE表には、文字データ用の列、数値データ用の列、およびADDRESSオブジェクトへの参照を含む列が設定されています。

オブジェクト参照を取り出すには、次のステップに従ってください。

1. 標準SQL SELECT文を使用して、データベース表のREF列から参照を取り出します。
2. getRefを使用して、結果セットからAddress参照を取り出し、OracleRefインスタンスに格納します。
3. AddressをSQLオブジェクト型のADDRESSに対応するJavaカスタム・クラスに変換します。
4. JavaクラスAddressとSQL型ADDRESS間の対応を、型マップに追加します。
5. getObjectメソッドを使用して、Address参照の内容を取り出します。出力をAddressにキャストします。

また、PEOPLEデータベース表の定義については、この項の始めの説明を参照してください。前述のステップを実行するコードは、Addressを型マップに追加するステップを除いて、次のようになります。

```
ResultSet rs = stmt.executeQuery("SELECT col3 FROM PEOPLE");
while (rs.next())
{
   OracleRef ref = rs.getRef(1);
   Address a = (Address)ref.getObject();
}
```



ノート:

前述のコードで、stmt はあらかじめ定義されている文オブジェクトです。

15.2.2 オブジェクト参照のコール可能文からの取出し

オブジェクト参照をPL/SQLブロックのOUTパラメータとして取り出すには、OUTパラメータのバインド型を登録する必要があります。

1. 次のように、コール可能文をOracleCallableStatementにキャストします。

   ```
   OracleCallableStatement ocs =
      (OracleCallableStatement)conn.prepareCall("{? = call func()}");
   ```

2. 次の形式のregisterOutParameterメソッドを使用して、OUTパラメータを登録します。

   ```
   ocs.registerOutParameter (int param_index, int sql_type, String sql_type_name);
   ```

   param_indexはパラメータ索引、sql_typeはSQL型コードです。sql_type_nameは、この参照が使用される構造化オブジェクト型の名前です。たとえば、OUTパラメータがADDRESSオブジェクトに対する参照の場合、ADDRESSは渡されるsql_type_nameです。

3. 次のように、コールを実行します。

   ```
   ocs.execute();
   ```

15.2.3 オブジェクト参照のプリペアド文への引渡し

オブジェクト参照をプリペアド文に渡す方法は、他のSQL型を渡す場合と同様です。プリペアド文のオブジェクトのsetObjectメソッドまたはsetREFメソッドを使用します。

次のプリペアド文を使用し、ROWIDに基づいてアドレス参照を更新します。

```
PreparedStatement pstmt =
   conn.prepareStatement ("update PEOPLE set ADDR_REF = ? where ROWID = ?");
pstmt.setRef (1, addr_ref);
    pstmt.setRowId (2, rowid);
```

15.3 オブジェクト値に対する、オブジェクト参照を介したアクセスと更新

RefオブジェクトのsetObjectメソッドを使用すると、データベースにあるオブジェクトの値をオブジェクト参照から更新できます。このためには、最初に、データベース・オブジェクトに対する参照を取り出し、データベース・オブジェクトに対応するJavaオブジェクトを作成する必要があります。

たとえば、次のコードのように、「オブジェクト参照の取出しと引渡し」のコードを使用して、データベースのADDRESSオブジェクトへの参照を取り出せます。

```
ResultSet rs = stmt.executeQuery("SELECT col3 FROM PEOPLE");
if (rs.next())
{
   Ref ref = rs.getRef(1);
   Address a = (Address)ref.getObject();
}
```

次に、データベースのADDRESSオブジェクトに対応するJavaのAddressオブジェクトを作成できます。次のように、RefインタフェースのsetObjectメソッドを使用して、データベース・オブジェクトの値を設定します。

```
Address addr = new Address(...);
ref.setObject(addr);
```

この例では、setValueメソッドによりデータベースのADDRESSオブジェクトが即時更新されます。

関連トピック

- オブジェクト参照の取出しと引渡し

16 Oracleコレクションの操作

この章では、Java配列とそのデータにマップされるOracleコレクションにアクセスして操作するための、標準Java Database Connectivity(JDBC)に対するOracleの拡張機能について説明します。次の内容について説明します。



ノート:

Oracle Database 12c リリース 1 (12.1)以降、oracle.sql.ARRAY クラスは非推奨となり、oracle.jdbc.OracleArray インタフェースに置き換えられています。このインタフェースは oracle.jdbc パッケージに属します。標準互換性には(可能であれば)java.sql パッケージの使用可能なメソッドを使用し、Oracle 固有の拡張機能には oracle.jdbc パッケージの使用可能なメソッドを使用することをお薦めします。oracle.jdbc.OracleArray インタフェースの詳細は、MoS ノート 1364193.1 を参照してください。

- コレクションのためのOracle拡張機能
- コレクション機能の概要
- ARRAYパフォーマンス拡張要素メソッド
- 配列の作成と使用方法
- 型マップを使用した配列要素のマップ

16.1 コレクションのためのOracle拡張機能

この項の内容は次のとおりです。

- Oracleコレクションの概要
- コレクションのインスタンス化に関する選択
- コレクションの作成
- マルチ・レベルのコレクション型の作成

16.1.1 Oracleコレクションの概要

データベース内の変数配列(VARRAY)またはNESTED TABLEのいずれかであるOracleコレクションは、Javaでは配列にマップされます。JDBC 2.0配列は、JavaでOracleコレクションをインスタンス化するために使用されます。コレクションおよび配列という用語は、どちらも同じ意味で使用されることがあります。ただし、データベース側ではコレクション、JDBCアプリケーション側では配列のほうが適切です。

サポートされるのは、SQL型名を指定することでコレクションの型を記述できる、名前付きコレクションのみです。JDBCを使用すると、次のものに配列を使用できます。

- SELECT句の列
- INまたはOUTバインド変数
- Oracleオブジェクトの属性

● 他の配列の要素

16.1.2 コレクションのインスタンス化に関する選択

アプリケーションでは、弱い型指定のoracle.sql.ARRAYクラスのインスタンスとして、または事前に作成した強い型指定のカスタムJavaクラスのインスタンスとしてコレクションをインスタンス化するオプションがあります。コレクションのために使用するカスタムJavaクラスは、カスタム・コレクション・クラスと呼びます。カスタム・コレクション・クラスは、Oracleのoracle.jdbc.OracleDataインタフェースを実装する必要があります。さらに、カスタム・クラスまたはコンパニオン・クラスは、oracle.jdbc.OracleDataFactoryを実装する必要があります。標準java.sql.SQLDataインタフェースは、SQLオブジェクト型のマッピング専用です。

oracle.sql.ARRAYクラスは、標準のjava.sql.Arrayインタフェースを実装します。

ARRAYクラスには、配列全体を取り出し、配列要素のサブセットを取り出し、配列要素のSQLベース型名を取り出すための機能があります。ただし、setterメソッドがないため、配列に書き込むことはできません。

カスタム・コレクション・クラスを使用すると、ARRAYクラスと同様に、配列のすべてまたは一部を取得して、SQLベース型名を取得できます。強い型指定により、実行時まで検出できない可能性のあるコーディング・エラーを、コンパイル時に発見できる利点もあります。



ノート:

VARRAY へのアクセスと NESTED TABLE へのアクセス間で、コード上の違いはありません。ARRAY クラス・メソッドは、VARRAY または NESTED TABLE に適用されているかどうかを判断し、適切なアクションを決定して応答します。

16.1.3 コレクションの作成

Oracleは名前付きコレクションのみをサポートしているため、特定のVARRAY型名またはNESTED TABLE型名を制限する必要があります。VARRAYとNESTED TABLEは、型自体ではなく型のカテゴリです。

次のSQL CREATE TYPE文を使用してコレクションを作成すると、コレクションにSQL型名が割り当てられます。

```
CREATE TYPE <sql_type_name> AS <datatype>;
```

VARRAYは、サイズ可変の配列です。順序付けられたデータ要素のセットを保持し、要素はすべて同じデータ型です。各要素は、索引を持ちますが、これはVARRAY.における要素の位置に対応する番号です。VARRAY内の要素数は、そのVARRAYのサイズを表します。VARRAY型を宣言する場合は、最大サイズを指定する必要があります。たとえば:

```
CREATE TYPE myNumType AS VARRAY(10) OF NUMBER;
```

この文では、10要素以下のNUMBER値を持つVARRAYが記述されたSQL型名として、myNumTypeが定義されています。

ネストした表は順序を設定していないデータ要素のセットで、すべて同じデータ型です。ネストした表はデータベースの別の表に格納されます。この表は単一の列を持ち、この列の型は組込み型またはオブジェクト型です。表がオブジェクト・タイプの場合、オ

ブジェクト・タイプの各属性の列を持つ複数列の表として表示されることがあります。ネストした表は、次のようにして作成することができます。

```
CREATE TYPE myNumList AS TABLE OF integer;
```

この文は、INTEGER型のNESTED TABLEに使用される表タイプが定義されたSQL型名として、myNumListを指定しています。

16.1.4 マルチ・レベルのコレクション型の作成

JDBCにおいてマルチ・レベルのコレクション型を作成する最も一般的な方法は、java.sql.StatementクラスのexecuteメソッドにSQLのCREATE TYPE文を渡すことです。次のコードは、first_levelという1レベルのNESTED TABLEと、second_levelという2レベルのNESTED TABLEを作成します。

```
Connection conn = ....                              // make a database
                                                    // connection
Statement stmt = conn.createStatement();            // open a database
                                                    // cursor
stmt.execute("CREATE TYPE first_level AS TABLE OF NUMBER");  // create a nested
                                                    // table of number
stmt.execute("CREATE TYPE second_level AS TABLE OF first_level"); // create a
           // two-levels nested table
...        // other operations here
stmt.close();                                       // release the
                                                    // resource
conn.close();                                       // close the
                                                    // database connection
```

マルチ・レベルのコレクション型を作成すると、実表の列またはオブジェクト型の属性として使用できます。



ノート:

マルチ・レベルのコレクション型は、Oracle9i 以降でのみ使用可能です。

16.2 コレクション機能の概要

結果セットまたはコール可能文によって配列インスタンス内のコレクション・データを取得し、プリペアド文またはコール可能文内のバインド変数として戻すことができます。

標準java.sql.Arrayインタフェースを実装するoracle.sql.ARRAYクラスは、Oracleコレクションのデータにアクセスし、更新するために必要な機能を提供します。

この項では、配列のgetterメソッドとsetterメソッドについて説明します。Java配列としてコレクションを取り出し、渡すには、次の結果セット、コール可能文およびプリペアド文メソッドを使用します。



ノート:

Oracle Database 12c リリース 1 (12.1)以降、oracle.sql.ARRAY クラスは非推奨となり、

oracle.jdbc.OracleArray インタフェースに置き換えられています。このインタフェースは oracle.jdbc パッケージに属します。標準互換性には(可能であれば)java.sql パッケージの使用可能なメソッドを使用し、Oracle 固有の拡張機能には oracle.jdbc パッケージの使用可能なメソッドを使用することをお薦めします。
oracle.jdbc.OracleArray インタフェースの詳細は、MoS ノート 1364193.1 を参照してください。

結果セットおよびコール可能文のgetterメソッド

OracleResultSetインタフェースとOracleCallableStatementインタフェースは、出力パラメータとして、つまりoracle.sql.ARRAYインスタンスまたはjava.sql.Arrayインスタンスとして ARRAYオブジェクトを取り出すために、getARRAY()メソッドとgetArray()メソッドをサポートします。getObjectメソッドも使用できます。これらのメソッドは、入力としてString列名またはint列索引を取ります。

ノート:



Oracle JDBC ドライバは、配列および構造の記述子をキャッシュします。これには多数のパフォーマンス上の利点がありますが、データベース内で配列型の基礎となる型定義を変更すると、その配列型のキャッシュされた記述子は古くなり、アプリケーションで SQLException が通知されます。

プリペアド文およびコール可能文のsetterメソッド

OraclePreparedStatementクラスとOracleCallableStatementクラスは、更新されたARRAYオブジェクトをバインド変数として取ってデータベースに渡すsetARRAYメソッドとsetArrayメソッドをサポートします。setObjectメソッドも使用できます。これらのメソッドは、oracle.sql.ARRAYインスタンスまたはjava.sql.Arrayインスタンスに加えて、Stringパラメータ名またはintパラメータ索引を入力として取ります。

16.3 ARRAYパフォーマンス拡張要素メソッド

この項では、次の項目について説明します。

- Javaプリミティブ型の配列としてのoracle.sql.ARRAY要素へのアクセスについて
- ARRAY自動要素バッファリング
- ARRAY自動索引作成

16.3.1 Javaプリミティブ型の配列としてのoracle.sql.ARRAY要素へのアクセスについて

oracle.sql.ARRAYクラスには、配列要素をJavaプリミティブ型として戻すメソッドが含まれています。これらのメソッドを使用すると、Datumインスタンスとしてコレクション要素にアクセスしてからDatumインスタンスをJavaプリミティブ型に変換する方法に比べ、より容易にコレクション要素にアクセスできます。



ノート:

oracle.sql.ARRAY クラスのこれらの特殊なメソッドは、数値コレクションに制限されます。

最初のシグネチャを使用している各メソッドは、コレクション要素をXXX[]の形式で戻します。ここで、XXXはJavaプリミティブ型です。第2のシグネチャを使用している各メソッドは、countによって指定される数の要素を含んでいて、indexの場所で開始されるコレクションのスライスを戻します。

16.3.2 ARRAY自動要素バッファリング

Oracle JDBCドライバには、ARRAYの内容のバッファリングを有効または無効にするためのパブリック・メソッドが用意されています。

oracle.sql.ARRAYクラスには、次のメソッドがあります。

- setAutoBuffering
- getAutoBuffering

ARRAY要素にgetAttributesおよびgetArrayの各メソッドで複数回アクセスし、ARRAYデータがオーバーフローせずにJava仮想マシン(JVM)のメモリーに格納されると想定する場合は、JDBCアプリケーションで自動バッファリングを有効にすることをお薦めします。



ノート:

変換した要素をバッファリングすると、JDBC アプリケーションでは、大量のメモリーが消費されます。

自動バッファリングを有効にすると、oracle.sql.ARRAYオブジェクトでは、変換したすべての要素のローカルのコピーが保持されます。このデータが保持されるため、この情報に2回目にアクセスするときにはデータ・フォーマット変換処理を実行しなくて済みます。

16.3.3 ARRAY自動索引作成

配列を自動索引作成モードにすると、配列オブジェクトは配列要素へのアクセスを迅速に行うために索引表をメンテナンスします。

oracle.sql.ARRAYクラスには、自動配列索引作成をサポートする次のメソッドが含まれています。

- setAutoIndexing(boolean)
- setAutoIndexing(boolean, int)

デフォルトでは、自動索引作成は有効にされていません。JDBCアプリケーションで、getArrayおよびgetResultSetの各メソッドを使用して配列要素のランダム・アクセスを実行する場合は、ARRAYオブジェクトの自動索引作成を有効にします。

16.4 配列の作成と使用方法

この項では、配列オブジェクトを作成する方法と、配列オブジェクトとしてコレクションを取り出し、渡す方法について説明します。内容は次のとおりです。

- ARRAYオブジェクトの作成
- 配列とその要素の取出し

- 配列の文オブジェクトへの引渡し

16.4.1 ARRAYオブジェクトの作成



ノート:

Oracle JDBC では、java.sql.Connection インタフェースの JDBC 4.0 の createArrayOf メソッドはサポートされていません。Oracle の配列型はすべて名前付きですが、このメソッドでは匿名の配列型のみが許容されています。かわりに Oracle 固有のメソッド、oracle.jdbc.OracleConnection.createARRAY を使用してください。

この項では、ARRAYオブジェクトを作成する方法について説明します。この項の内容は次のとおりです。

- ARRAYオブジェクトの作成に関するステップ
- 例16-1

ARRAYオブジェクトの作成に関するステップ

Oracle Database 11gリリース1以降、oracle.jdbc.OracleConnectionインタフェースのcreateARRAYファクトリ・メソッドを使用して配列オブジェクトを作成することができます。配列を作成するファクトリ・メソッドは次のように定義されています。

```
public ARRAY createARRAY(java.lang.String typeName,java.lang.Object elements)throws SQLException
```

ここで、typeNameは、作成されたオブジェクトのSQL型の名前で、elementsは作成されたオブジェクトの要素です。

配列を作成するには次の操作を実行します。

1. CREATE TYPE文を使用して、次のようにコレクションを作成します。

   ```
   CREATE TYPE elements AS varray(22) OF NUMBER(5,2);
   ```

   elementsの内容は、次の2つの場合があります。

   - Javaプリミティブ形の配列。たとえば、int[]です。
   - xxx[]などのJavaオブジェクトの配列。ここで、xxxはJavaクラスの名前です。たとえば、Integer[]です。

   

   ノート:

   OraclePreparedStatement クラスの setARRAY、setArray および setObject メソッドは、オブジェクトの配列ではなく、oracle.sql.ARRAY 型のオブジェクトを引数として使用します。

2. 配列のユーザー定義SQL型名を指定しているJava文字列、および配列に含まれる個別の要素が含まれているJavaオブジェクトを渡すことによって、ARRAYオブジェクトを構築します。

   ```
   ARRAY array = oracle.jdbc.OracleConnection.createARRAY(sql_type_name, elements);
   ```



ノート:

コレクション型の名前は、要素の型名と同じではありません。たとえば:

```
CREATE TYPE person AS object
            (c1 NUMBER(5), c2 VARCHAR2(30));
CREATE TYPE array_of_persons AS varray(10)
            OF person;
```

前述の文では、コレクション型の名前は、ARRAY_OF_PERSON です。コレクション要素の SQL 型名は、PERSON です。

例16-1 マルチ・レベル・コレクションの作成

JDBCアプリケーションは、シングル・レベル・コレクションと同様、oracle.sql.ARRAYインスタンスを作成してマルチ・レベル・コレクションを表し、インスタンスをデータベースに送信することができます。シングル・レベル・コレクションを作成するために使用するのと同じcreateARRAYファクトリ・メソッドを使用することによってマルチ・レベル・コレクションを作成することもできます。シングル・レベル・コレクションを作成する場合、要素は一次元のJava配列ですが、マルチ・レベル・コレクションを作成する場合、要素はoracle.sql.ARRAY[]要素の配列、ネストJava配列、またはその組合せです。

次のコードは、ネストしたJava配列を使用してコレクション型を作成する方法を示します。

```
// prepare the multilevel collection elements as a nested Java array
int[][][] elements = { {{1}, {1, 2}}, {{2}, {2, 3}}, {{3}, {3, 4}} };
// create the ARRAY using the factory method
ARRAY array = oracle.jdbc.OracleConnection.createARRAY(sql_type_name, elements);
```

16.4.2 配列とその要素の取出し

この項では、結果セットからARRAYインスタンス全体を取り出す方法を最初に説明し、次にARRAYインスタンスから要素を取り出す方法について説明します。この項の内容は次のとおりです。

- 配列の取出しについて
- データ取出しメソッド
- データ取出しメソッドの比較
- 型マップに従った構造化オブジェクト配列の要素の取出し
- 配列要素のサブセットの取出し
- oracle.sql.Datum配列への配列要素の取出し
- マルチ・レベル・コレクション要素へのアクセスについて

16.4.2.1 配列の取出しについて

oracle.sql.ARRAYオブジェクトを戻すgetARRAYメソッドを使用して、結果セットをOracleResultSetオブジェクトにキャストすることで、SQL配列を結果セットから取り出すことができます。結果セットをキャストするのを避けるには、java.sql.ResultSetインタフェースによって指定された標準のgetObjectメソッドを使用してデータを取得し、その出力をoracle.sql.ARRAYにキャストします。

16.4.2.2 データ取出しメソッド

ARRAYオブジェクトを作成した後は、次に示すoracle.sql.ARRAYクラスの3つのオーバーロード・メソッドのいずれかを使用して、データを取り出すことができます。

- getArray
- getOracleArray
- getResultSet

Oracleは配列のすべての要素、またはサブセットを取り出すことができるメソッドも提供しています。



ノート:

構造化オブジェクトの配列を操作している場合は、Oracle は、オブジェクトを Java にマップする方法を選択できるように、型マップを指定できる次の 3 つのメソッドのバージョンを提供します。

getOracleArray

getOracleArray()メソッドは、標準Arrayインタフェースでは指定されていないOracle固有の拡張機能です。getOracleArrayメソッドでは、配列の要素値を取り出してDatum[]配列に格納します。この要素は、元の配列のSQL型データに対応するoracle.sql.*データ型です。

構造化オブジェクトの配列では、このメソッドは要素のためにoracle.jdbc.OracleStructインスタンスを使用します。

また、getOracleArray(index,count)メソッドを使用すると、配列要素のサブセットを取得できます。

getResultSet

getResultSetメソッドは、ARRAYオブジェクトによって示される配列の要素が含まれる結果セットを戻します。結果セットは配列要素ごとに1行を含み、各行には2つの列があります。最初の列には、配列内でその要素を参照する索引が格納され、2つめの列には要素値が格納されます。VARRAYの場合、索引は配列内での要素の位置を表します。定義上、順序付けられていないネスト表の場合、索引は特定の問合せにおける要素の戻り順のみを反映します。

NESTED TABLEからデータを取り出すときは、getResultSetを使用することをお薦めします。NESTED TABLEの要素数には、制限はありません。メソッドから戻されたResultSetオブジェクトのポインタの初期値は、データの第1行です。nextメソッドおよび適切なgetXXXメソッドを使用すると、NESTED TABLEの内容を取得できます。また、getArrayを使用すると、NESTED TABLEのすべての内容が一度に戻されます。

getResultSetメソッドでは、接続のデフォルト型マップを使用して、OracleオブジェクトのSQL型とその対応するJavaデータ型間のマッピングを決定します。接続のデフォルト型マップを使用しない場合は、getResultSet(map)を使用して別の型マップを指定できます。

また、getResultSet(index,count)およびgetResultSet(index,count,map)メソッドを使用すると、配列のサブセットを取り出すことができます。

getArray

getArrayメソッドは、必要に応じてキャストできるjava.lang.Objectとして配列要素を戻す標準JDBCメソッドです。要素は、元の配列のSQL型データに対応するJava型データに変換されます。

また、getArray(index,count)メソッドを使用すると、配列要素のサブセットを取り出すことができます。

16.4.2.3 データ取出しメソッドの比較

配列要素を戻すためにgetOracleArrayを使用する場合は、そのメソッドがoracle.sql.Datumインスタンスを使用するため、SQLからJavaへのデータ変換を行う必要がなくなります。Datumクラスまたはそのサブクラスのインスタンス内にある文字以外のデータは、RAW SQL形式で保持されます。

getResultSetを使用してプリミティブ・データ型の配列を戻す場合、JDBCドライバはResultSetオブジェクトを戻します。このオブジェクトには、各要素ごとに、その要素を配列内で参照する索引と要素値が含まれます。たとえば:

```
ResultSet rset = intArray.getResultSet();
```

この場合、結果セットには、1つの配列要素ごとに1つの行があり、各行には2つの列があります。最初の列には、配列内でその要素を参照する索引が格納され、2つめの列には要素値が格納されます。

配列の要素が、Java型にマップするSQL型の場合、getArrayはこのJava型の要素の配列を戻します。getArrayメソッドの戻り型はjava.lang.Objectです。したがって、結果は使用する前にキャストする必要があります。

```
BigDecimal[] values = (BigDecimal[]) intArray.getArray();
```

ここで、intArrayはoracle.sql.ARRAYで、NUMBER型のVARRAYに対応しています。values配列にはjava.math.BigDecimal型の要素の配列が含まれます。これはSQL NUMBERデータ型が、デフォルトで、Oracle JDBCドライバによって、Java BigDecimalにマップされるためです。

ノート:



BigDecimal の使用は、リソースを大量に消費する Java の操作です。Oracle JDBC は数値 SQL データをデフォルトで BigDecimal にマップするため、getArray を使用するとパフォーマンスに影響を与える可能性があり、数値のコレクションの場合は推奨されません。

16.4.2.4 型マップに従った構造化オブジェクト配列の要素の取出し

デフォルトでは、構造化オブジェクトの要素を持つ配列を操作しているときに、getArrayまたはgetResultSetを使用すると、デフォルト・マッピングに従って配列のOracleオブジェクトが、対応するJavaデータ型にマップされます。これは、これらのメソッドでは、接続のデフォルト型マップを使用してマッピングが決定されるためです。

ただし、デフォルトの処理を変更する場合は、getArray(map)またはgetResultSet(map)メソッドを使用して、別のマッピングを含む型マップを指定できます。配列のOracleオブジェクトに対応するエントリが型マップに存在する場合は、配列の各オブジェクトは、その型マップで指定されている、対応するJava型にマップされます。たとえば:

```
Object[] object = (Object[])objArray.getArray(map);
```

この例のobjArrayはoracle.sql.ARRAYオブジェクトを表し、mapはjava.util.Mapオブジェクトを表します。

型マップに特定のOracleオブジェクトに対応するエントリが含まれない場合、要素はoracle.jdbc.OracleStructオブジェクトとして戻されます。

getResultSet(map)メソッドは、getArray(map)メソッドと同じように動作します。

関連トピック

- 型マップを使用した配列要素のマップ

16.4.2.5 配列要素のサブセットの取出し

配列の内容全体を取り出さない場合は、サブセットを取り出すことができるgetArray、getResultSetおよびgetOracleArrayのシグネチャを使用できます。配列のサブセットを取り出すには、索引と件数を渡して、取出しを開始する配列の位置および取り出す要素数を指定します。前の例と同様に、接続に対して型マップを指定するか、またはデフォルトの型マップを使用して、Java型に変換します。たとえば:

```
Object object = arr.getArray(index, count, map);
Object object = arr.getArray(index, count);
```

getResultSetを使用した例です。

```
ResultSet rset = arr.getResultSet(index, count, map);
ResultSet rset = arr.getResultSet(index, count);
```

getOracleArrayを使用した例です。

```
Datum[] arr = arr.getOracleArray(index, count);
```

arrはoracle.sql.ARRAYオブジェクト、indexはlong型、countはint型、mapはjava.util.Mapオブジェクトを表します。



ノート:

配列全体ではなく、配列のサブセットを取り出すことにパフォーマンス上の利点はありません。

16.4.2.6 oracle.sql.Datum配列への配列要素の取出し

oracle.sql.Datum[]配列を戻すには、getOracleArrayを使用します。戻される配列の要素は、元の配列要素のSQLデータ型に対応するoracle.sql.*型です。たとえば:

```
Datum arraydata[] = arr.getOracleArray();
```

arrはoracle.sql.ARRAYオブジェクトを表します。

次の例は、接続オブジェクトconnおよび文オブジェクトstmtがすでに作成済であることを前提としています。この例では、SQL型名NUM_ARRAYの配列がNUMBERデータのVARRAYを格納するために作成されます。NUM_ARRAYは、表VARRAY_TABLEに格納されます。

問合せは、VARRAY_TABLEの内容を選択します。結果セットはOracleResultSetにキャストされます。それに対してgetARRAYメソッドが適用され、配列データがmy_arrayに取得されます。これはoracle.sql.ARRAYオブジェクトです。

my_arrayはoracle.sql.ARRAY型の配列であるため、getSQLTypeNameメソッドおよびgetBaseTypeメソッドをこの配列に適用すれば、その配列の要素ごとに、その整数コードでSQL型名を戻すことができます。

次に、プログラムにより配列の内容が出力されます。NUM_ARRAYの内容はSQLデータ型NUMBERであるため、my_arrayの要素はBigDecimal型です。要素を使用するには、最初に、その要素をBigDecimalにキャストする必要があります。forループでは、配列の各値はBigDecimalにキャストされ、標準出力に出力されます。

```
stmt.execute ("CREATE TYPE num_varray AS VARRAY(10) OF NUMBER(12, 2)");
stmt.execute ("CREATE TABLE varray_table (col1 num_varray)");
stmt.execute ("INSERT INTO varray_table VALUES (num_varray(100, 200))");
ResultSet rs = stmt.executeQuery("SELECT * FROM varray_table");
ARRAY my_array = ((OracleResultSet)rs).getARRAY(1);
// return the SQL type names, integer codes,
// and lengths of the columns
System.out.println ("Array is of type " + array.getSQLTypeName());
System.out.println ("Array element is of type code " + array.getBaseType());
System.out.println ("Array is of length " + array.length());
// get Array elements
BigDecimal[] values = (BigDecimal[]) my_array.getArray();
for (int i=0; i<values.length; i++)
{
   BigDecimal out_value = (BigDecimal) values[i];
   System.out.println(">> index " + i + " = " + out_value.intValue());
}
```

getResultSetを使用して配列を取得する場合は、最初に結果セット・オブジェクトを取得し、次にnextメソッドを使用して操作を反復する必要があるので、注意が必要です。getIntメソッドでは、パラメータの索引を使用して、要素の索引および要素値を取り出します。

```
ResultSet rset = my_array.getResultSet();
while (rset.next())
{
    // The first column contains the element index and the
    // second column contains the element value
    System.out.println(">> index " + rset.getInt(1)+" = " + rset.getInt(2));
}
```

16.4.2.7 マルチ・レベル・コレクション要素へのアクセスについて

oracle.sql.ARRAYクラスは、コレクション要素にアクセスするために、オーバーロードされた3つのメソッドを提供します。JDBCドライバは、これらのメソッドをマルチ・レベル・コレクションをサポートするように拡張します。これらのメソッドを次に示します。

- getArrayメソッド
- getOracleArrayメソッド
- getResultSetメソッド

getArrayメソッドは、コレクション要素を保持するJava配列を戻します。配列要素の型は、コレクション要素の型とJDBCのデフォルトの変換マトリックスによって決まります。

たとえば、getArrayメソッドは、SQL NUMBER型のコレクションのjava.math.BigDecimal配列を戻します。getOracleArrayメソッドは、Datum形式でコレクション要素を保持するDatum配列を戻します。マルチ・レベル・コレクションの場合、getArrayメソッ

ドおよびgetOracleArrayメソッドはいずれも、oracle.sql.ARRAY要素のJava配列を戻します。

getResultSetメソッドは、マルチ・レベル・コレクション要素をラップするResultSetオブジェクトを戻します。マルチ・レベル・コレクションの場合、JDBCアプリケーションはResultSetクラスのgetObject、getARRAYまたはgetArrayメソッドを使用して、oracle.sql.ARRAYのインスタンスとしてコレクション要素にアクセスします。

次のコードは、getOracleArray、getArrayおよびgetResultSetメソッドの使用方法を示します。

```
Connection conn = ...;          // make a JDBC connection
Statement stmt = conn.createStatement ();
ResultSet rset = stmt.executeQuery ("select col2 from tab2 where idx=1");
while (rset.next())
{
  ARRAY varray3 = (ARRAY) rset.getObject (1);
  Object varrayElems = varray3.getArray (1);
// access array elements of "varray3"
  Datum[] varray3Elems = (Datum[]) varrayElems;
  for (int i=0; i<varray3Elems.length; i++)
  {
    ARRAY varray2 = (ARRAY) varray3Elems[i];
    Datum[] varray2Elems = varray2.getOracleArray();
    // access array elements of  "varray2"
    for (int j=0; j<varray2Elems.length; j++)
    {
      ARRAY varray1 = (ARRAY) varray2Elems[j];
      ResultSet varray1Elems = varray1.getResultSet();
      // access array elements   of "varray1"
      while (varray1Elems.next())
        System.out.println ("idx="+varray1Elems.getInt(1)+"
          value="+varray1Elems.getInt(2));
    }
  }
}
rset.close ();
stmt.close ();
conn.close ();
```

16.4.3 配列の文オブジェクトへの引渡し

この項では、プリペアド文オブジェクトまたはコール可能文オブジェクトに配列を渡す方法について説明します。

配列のプリペアド文への引渡し

次の手順に従って、配列をプリペアド文に渡します。



ノート:

配列は、IN または OUT バインド変数として使用できます。

1. プリペアド文にoracle.sql.ARRAYオブジェクトとして渡す配列を定義します。

```
ARRAY array = oracle.jdbc.OracleConnection.createARRAY(sql_type_name, elements);
```

sql_type_nameは配列のユーザー定義SQL型名を指定するJava文字列を表し、elementsは要素のJava配列を含むjava.lang.Objectを表します。

2. 実行するSQL文を含むjava.sql.PreparedStatementオブジェクトを作成します。
3. プリペアド文をOraclePreparedStatementにキャストし、setARRAY()を使用して、プリペアド文に配列を渡します。

```
(OraclePreparedStatement)stmt.setARRAY(parameterIndex, array);
```

parameterIndexはパラメータ索引を表し、arrayは手順2で作成したoracle.sql.ARRAYオブジェクトを表します。

4. プリペアド文を実行します。

配列のコール可能文への引渡し

コレクションをPL/SQLブロックのOUTパラメータとして取り出すには、次の手順を実行してOUTパラメータのバインド型を登録します。

1. 次のように、コール可能文をOracleCallableStatementにキャストします。

```
OracleCallableStatement ocs = (OracleCallableStatement)conn.prepareCall("{? = call func()}");
```

2. 次の形式のregisterOutParameterメソッドを使用して、OUTパラメータを登録します。

```
ocs.registerOutParameter
       (int param_index, int sql_type, string sql_type_name);
```

param_indexはパラメータ索引、sql_typeはSQL型コード、sql_type_nameは配列型の名前です。この場合、sql_typeはOracleTypes.ARRAYです。

3. 次のように、コールを実行します。

```
ocs.execute();
```

4. 次のように、値を取得します。

```
oracle.sql.ARRAY array = ocs.getARRAY(1);
```

16.5 型マップを使用した配列要素のマップ

配列にOracleオブジェクトが含まれる場合は、型マップを使用して、配列のオブジェクトを、対応するJavaクラスに関連付けることができます。型マップを指定しない場合または型マップに特定のOracleオブジェクトのエントリが含まれない場合、各要素はoracle.jdbc.OracleStructオブジェクトとして戻されます。

型マップを使用して、配列のOracleオブジェクトと対応するJavaクラスとのマッピングを決定するときは、マップに適切なエントリを追加する必要があります。

次の例では、型マップを使用して配列要素をカスタムJavaオブジェクト・クラスにマップする方法を説明します。この例の配列は、NESTED TABLEです。この例では、まず、名前属性と従業員番号属性が設定されているEMPLOYEEオブジェクトを定義します。EMPLOYEE_LISTはEMPLOYEEオブジェクトのNESTED TABLE型です。次に、会社内の部署名および各部署の従業員名を格納するEMPLOYEE_TABLEを作成します。EMPLOYEE_TABLEの中では、従業員はEMPLOYEE_LIST表の形式で格納されます。

```
stmt.execute("CREATE TYPE EMPLOYEE AS OBJECT
            (EmpName VARCHAR2(50),EmpNo INTEGER))");
stmt.execute("CREATE TYPE EMPLOYEE_LIST AS TABLE OF EMPLOYEE");
stmt.execute("CREATE TABLE EMPLOYEE_TABLE (DeptName VARCHAR2(20),
```

```
     Employees EMPLOYEE_LIST) NESTED TABLE Employees STORE AS ntable1");
stmt.execute("INSERT INTO EMPLOYEE_TABLE VALUES ("SALES", EMPLOYEE_LIST
            (EMPLOYEE('Susan Smith', 123), EMPLOYEE('Lee Brown', 124)))");
```

SALES部門に属するすべての従業員をカスタム・オブジェクト・クラスEmployeeObjのインスタンスの配列に取り込む場合は、EMPLOYEE SQL型とEmployeeObjカスタム・オブジェクト・クラス間のマッピングを指定するために、型マップにエントリを追加する必要があります。

これを実行するには、まず文と結果セット・オブジェクトを作成し、次に、SALES部門に関連付けられているEMPLOYEE_LISTを結果セットに選択します。getARRAYメソッドを使用してEMPLOYEE_LISTをARRAYオブジェクト(次の例ではemployeeArray)に取得できるように、結果セットをOracleResultSetにキャストします。

この例のEmployeeObjカスタム・オブジェクト・クラスは、SQLDataインタフェースを実装しています。

```
Statement s = conn.createStatement();
OracleResultSet rs = (OracleResultSet)s.executeQuery
       ("SELECT Employees FROM employee_table WHERE DeptName = 'SALES'");
// get the array object
ARRAY employeeArray = ((OracleResultSet)rs).getARRAY(1);
```

EMPLOYEE_LISTオブジェクトを取り出したので、既存の型マップを取得し、SQL型EMPLOYEEをJava型EmployeeObjにマップするエントリを追加します。

```
// add type map entry to map SQL type
// "EMPLOYEE" to Java type "EmployeeObj"
Map map = conn.getTypeMap();
map.put("EMPLOYEE", Class.forName("EmployeeObj"));
```

次に、EMPLOYEE_LISTからSQL EMPLOYEEオブジェクトを取り出します。この操作をするには、employeeArray配列オブジェクトのgetArrayメソッドをコールします。このメソッドにより、オブジェクト配列が戻されます。getArrayメソッドでは、employeesオブジェクト配列にEMPLOYEEオブジェクトが戻されます。

```
// Retrieve array elements
Object[] employees = (Object[]) employeeArray.getArray();
```

最後にループを作成して、各EMPLOYEE SQLオブジェクトに、EmployeeObj Javaオブジェクトのempを割り当てます。

```
// Each array element is mapped to EmployeeObj object.
for (int i=0; i<employees.length; i++)
{
   EmployeeObj emp = (EmployeeObj) employees[i];
   ...
}
```

17 結果セット

Java Development Kit(JDK)の標準Java Database Connectivity(JDBC)機能には、結果セット機能の拡張が含まれています。拡張には、前方または後方への処理、相対的または絶対的な位置指定、内部または外部で生じたデータベースに対する変更の参照、および結果セット・データの更新と更新による変更内容のデータベースへのコピーがあります。

この章では、以下のトピックについて説明します。

- 結果セットのサポートのOracle JDBC実装概要
- 結果セットの制限事項およびダウングレード・ルール
- 更新の競合回避について
- 行フェッチ・サイズ
- 行の再フェッチについて
- 内部的および外部的に加えられたデータベース変更の参照について

17.1 結果セットのサポートのOracle JDBC実装概要

この項では、スクロール可能性(クライアント側キャッシュを使用)および更新可能性(ROWIDを使用)を実現する結果セットのサポートのOracle JDBC実装に関する主要な視点について説明します。

ユーザーは、独自のクライアント側キャッシュ・メカニズムを実装できます。そのためのインタフェースが提供されています。

結果セットのスクロール可能性のためのOracle JDBC実装

基礎となるサーバーでスクロール可能なカーソルがサポートされないため、Oracle JDBCが、別々のレイヤーでスクロール可能性を実装する必要があります。

この機能は、スクロール可能な結果セットの行をクライアント側のメモリー・キャッシュに格納することにより、実現されていることに注意してください。



ノート:

あらゆるスクロール可能な結果セットのすべての行がクライアント側キャッシュに格納されるため、結果セットに多くの行、多くの列または非常に大きな列が含まれる場合、クライアント側 Java 仮想マシン(JVM)に障害が発生する原因になる可能性があります。大きな結果セットに対してはスクロール可能性を指定しないでください。

結果セットの更新可能性のためのOracle JDBC実装

更新可能性をサポートするために、Oracle JDBCはROWIDを使用して、結果セットに出現するデータベース行を一意に識別します。更新可能な結果セットに問い合せるたびに、選択された列とともに、Oracle JDBCドライバによって自動的にROWIDが取り出されます。



ノート:

更新可能性そのものは、クライアント側のキャッシュを必要としません。特に、forward-only 更新可能結果セットは、クライアント側のキャッシュを必要としません。

17.2 結果セットの制限事項およびダウングレード・ルール

一部の結果セット型は、ある種類の問合せには不適切です。実行する問合せに対して不適切な結果セット型または同時実行性型が指定されている場合、JDBCドライバは次のルール・セットに従い、かわりに使用する最適な型を判断します。

指定された結果セット型または同時実行性型が不適切な場合、実際の結果セット型と同時実行性型は、文が実行されるときに、文オブジェクト上で SQLWarning を発行しているドライバを使用して決定されます。SQLWarning オブジェクトには、要求した型が不適切だった理由が含まれます。要求した結果セットの型を受け取ったかどうか確認するには、警告を確認してください。

結果セットの制限

拡張された結果セットの問合せには、次の制限事項が適用されます。次の指針に当てはまらない場合、JDBCドライバによって、代替の結果セット型または同時実行性型が選択されます。

更新可能な結果セットを作成するには、次の制限事項を考慮します。

- 問合せによって選択できるのは、単一の表のみです。結合操作を含めることはできません。

  さらに、挿入するためには、NULL化可能でない列およびデフォルト値が設定されていない列をすべて、問合せによって選択する必要があります。

- 問合せではSELECT *を使用できません。

  ただし、回避方法があります。

- 問合せで選択できるのは、表列のみです。

  導出列や、列の集合のSUMまたはMAXなど、集合体を選択できません。

scroll-sensitive結果セットを作成するには、次の制限事項を考慮します。

- 問合せではSELECT *を使用できません。

  ただし、回避方法があります。

- 問合せによって選択できるのは、単一の表のみです。

スクロール可能結果セットおよび更新可能結果セットでは、Streamなどの列を使用できません。サーバーでStream列をフェッチする必要がある場合は、フェッチ・サイズを1に減らし、Stream列の読込みが終了するまで、Stream列に続くすべての列をブロックします。このため、列を一括でフェッチし、スクロールすることはできません。

回避策

SELECT *の制限事項を回避するには、次の例のように、表の別名を使用します。

```
SELECT t.* FROM TABLE t ...
```



ノート:

ある問合せから、scroll-sensitive または更新可能な結果セットを作成できるかどうかを判断する簡単な方法があります。ROWID 列を問合せリストに追加できる場合は、その問合せから scroll-sensitive または更新可能な結果セットを作成できます。

結果セットのダウングレード・ルール

指定された結果セット型または同時実行性型が不適切な場合、Oracle JDBCドライバは次のルールを使用して、代替型を選択します。

- 指定した結果セット型がTYPE_SCROLL_SENSITIVEであるにもかかわらず、JDBCドライバがその要求を完全には満たせない場合、JDBCドライバはTYPE_SCROLL_INSENSITIVEにダウングレードしようとします。
- 指定した結果セット型またはダウングレード後の結果セット型がTYPE_SCROLL_INSENSITIVEであるにもかかわらず、JDBCドライバがその要求を完全には満たせない場合、JDBCドライバはTYPE_FORWARD_ONLYにダウングレードしようとします。
- 指定した同時実行性型がCONCUR_UPDATABLEであるにもかかわらず、JDBCドライバがその要求を完全には満たせない場合、JDBCドライバはCONCUR_READ_ONLYにダウングレードしようとします。



ノート:

JDBC ドライバによる結果セット型および同時実行性型の操作は、それぞれ独立しています。

結果セット型および同時実行性型の検証

問合せを実行した後、結果セット・オブジェクトのメソッドをコールすることにより、JDBCドライバが実際に使用した結果セット型と同時実行性型を検証できます。

- `int getType() throws SQLException`

  このメソッドは、問合せに対して使用された結果セット型のint値を戻します。ResultSet.TYPE_FORWARD_ONLY、ResultSet.TYPE_SCROLL_SENSITIVEまたはResultSet.TYPE_SCROLL_INSENSITIVEという値が戻されます。
- `int getConcurrency() throws SQLException`

  このメソッドは、問合せに対して使用された同時実行性型のint値を戻します。ResultSet.CONCUR_READ_ONLYまたはResultSet.CONCUR_UPDATABLEという値が戻されます。

17.3 更新の競合回避について

JDBCドライバで使用される更新可能な結果セットに関しては、次の事項に注意してください。

- ドライバは、更新可能な結果セットに対する書込みロックを施行しません。
- ドライバは、結果セットのDELETEまたはUPDATE操作の競合をチェックしません。

DELETEまたはUPDATE操作を、コミットされた別のトランザクションによって更新された行に対して実行すると、競合が発生します。

Oracle JDBCドライバは、ROWIDを使用し、データベース表の行を一意に識別します。ドライバがUPDATEまたはDELETE操作を

データベースに送信しようとしたとき、ROWIDが有効であれば、操作は実行されます。

コミットされた別のトランザクションによる変更は、レポートされません。競合は警告なしに無視され、新しい変更によって前の変更は上書きされます。

競合を回避するには、結果セットを作成する問合せを実行するときに、OracleのFOR UPDATE機能を使用します。これにより、競合を回避できますが、データへの同時アクセスも禁止されます。データ項目に同時に設定できる書込みロックは1つのみです。

17.4 行フェッチ・サイズ

デフォルトでは、Oracle JDBCは問合せを実行するとき、データベース・カーソルから一度に10行の結果セットを受け取ります。これが、デフォルトのOracle行フェッチ・サイズ値です。データベース・カーソルへの1回のラウンドトリップで取り出される行の数を変更するには、行フェッチ・サイズ値を変更します。

標準JDBCでは、1つの問合せの1回のデータベース・ラウンドトリップでフェッチされる行の数を指定することもできます。この数はフェッチ・サイズと呼ばれます。Oracle JDBCでは、行プリフェッチ値が、文オブジェクトのデフォルト・フェッチ・サイズとして使用されます。フェッチ・サイズを設定すると、行プリフェッチ設定より優先され、その文オブジェクトを介して実行される後続の問合せに適用されます。

フェッチ・サイズは、結果セットでも使用されます。文オブジェクトが問合せを実行するとき、文オブジェクトのフェッチ・サイズが、その問合せで作成される結果セット・オブジェクトに渡されます。ただし、結果セット・オブジェクトでフェッチ・サイズを設定して、渡された文フェッチ・サイズをオーバーライドすることもできます。



ノート:

結果セットが作成された後で文オブジェクトのフェッチ・サイズを変更しても、結果セットには影響を与えません。

明示的に設定された場合でも、渡された文フェッチ・サイズと等しいデフォルトでも、結果セット・フェッチ・サイズによって、その結果セットでの後続のデータベースへのラウンドトリップで取り出される行の数が判断されます。このラウンドトリップには、元の問合せを完了するために必要なラウンドトリップと、結果セットへのデータの再フェッチに必要なラウンドトリップが含まれます。scroll-sensitiveまたはscroll-insensitive/更新可能な結果セットを更新するために、データは明示的または暗黙的に再フェッチされる可能性があります。

17.4.1 フェッチ・サイズの設定

Statement、PreparedStatement、CallableStatementおよびResultSetオブジェクトのすべてで、フェッチ・サイズを設定および取り出す次のメソッドが使用可能です。

- void setFetchSize(int rows) throws SQLException
- int getFetchSize() throws SQLException

問合せのフェッチ・サイズを設定するには、問合せを実行する前に文オブジェクトのsetFetchSizeをコールします。フェッチ・サイズをNに設定すると、1回のデータベースへのラウンドトリップで、N行がフェッチされます。

問合せを実行した後、結果セット・オブジェクト上でsetFetchSizeをコールすることで、渡された文オブジェクト・フェッチ・サイズを

オーバーライドすることができます。これは、元の問合せに応じてさらに行を取得するための後続のアクセスすべてと、それ以降の、行の再フェッチすべてに適用されます。

17.4.2 フェッチ方向のプリセット

標準JDBCでは、結果セットの処理に使用するために、フェッチ方向と呼ばれる方向を事前に指定できます。これにより、JDBCドライバの処理を最適化できます。次の結果セット・メソッドが指定されます。

- void setFetchDirection(int direction) throws SQLException
- int getFetchDirection() throws SQLException

Oracle JDBCドライバは前方へのプリセット値のみをサポートします。これは、ResultSet.FETCH_FORWARD静的定数値を入力することにより、指定できます。

ResultSet.FETCH_REVERSEおよびResultSet.FETCH_UNKNOWNという値はサポートされていません。指定しようとすると、SQL警告が発生し、設定は無視されます。

17.5 行の再フェッチについて

結果セットのrefreshRowメソッドは、一部の種類の結果セットで、データを再フェッチするためにサポートされています。具体的には、データベースに戻って、結果セットの現在の行からのn行に対応するデータベース行を再取得します。ここで、nはフェッチ・サイズです。これを使用すると、周囲のトランザクションの分離レベルに応じて、結果セットの外部で行われたデータベースの最新の更新を参照できます。

再フェッチすると、すでに結果セットにある行に対応する行のみを再取得します。元の問合せ以降にデータベースで挿入や削除が行われた行については何もしません。挿入された行は無視されます。また、対応する行がデータベースから削除された後も、結果セットの行はそのまま残ります。データベースで削除された行を再フェッチしようとした場合、それに対応する結果セットの行は、元の値を維持します。



ノート:

いくつかの基準を持つ問合せに基づいて TYPE_SCROLL_SENSITIVE 結果セットを宣言してから、列の値が問合せの基準に一致しなくなるように外部的に行を更新すると、ドライバは、行がデータベースから削除され、送信された問合せでは取り出せないかのように動作します。このため、refreshRow メソッドをコールしても個別行の更新は確認できません。

refreshRowメソッドのシグネチャを示します。

```
void refreshRow() throws SQLException
```

このメソッドをコールするときは、行境界の外側や挿入行ではなく、有効な現在行に位置指定しておく必要があります。

次の結果セットのカテゴリで、refreshRowメソッドがサポートされています。

- scroll-sensitive/読取り専用
- scroll-sensitive/更新可能

- scroll-insensitive/更新可能



ノート:

Scroll-Sensitive 結果セット機能は、`refreshRow` の暗黙的なコールによって実装されます。

17.6 内部的および外部的に加えられたデータベース変更の参照について

この項では、次の事項を参照する結果セットの機能について説明します。

- 内部変更と呼ばれる結果セット自体の変更
- 外部変更と呼ばれる、結果セット以外から加えられた変更(ユーザー自身による結果セット外のトランザクションによる変更、またはその他のコミットされたトランザクションによる変更)



ノート:

標準 JDBC の仕様では、外部変更は「他からの変更」と呼ばれています。

この項の内容は次のとおりです。

- 外部変更の可視性と検出
- 内部変更および外部変更の可視性の概要
- Scroll-Sensitive結果セットのOracle実装

17.6.1 外部変更の可視性と検出

外部ソースによって基礎となるデータベースに加えられる変更に関して、2つの概念があります。これらは似ていますが、ローカルな結果セットからの変更の可視性という観点から見ると異なります。

- 変更の可視性
- 変更の検出

「可視である」変更とは、結果セットの行を見たとき、外部ソースによってデータベースの対応する行に加えられた変更からの新しいデータ値を参照できるという意味です。

一方、「検出される」変更とは、結果セットに最初に値が移入されて以降の新しい値であることを、結果セットが認識するという意味です。

Oracle結果セットでは、新しいデータを参照したときでも(scroll-sensitive結果セットでの外部UPDATEのように)、このデータが結果セットへの移入後に変更されたことは認識されません。このような変更は、検出されません。

17.6.2 内部変更および外部変更の可視性の概要

表17-1では、Oracle JDBC実装の結果セット・オブジェクトが、結果セット自体が内部的に実行した変更および基礎となるデータベースに対してユーザー自身のトランザクション、またはその他のコミットされたトランザクションから外部的に実行された変

更を参照できるかどうかについての状況を要約します。

表17-1 Oracle JDBCでの内部変更および外部変更の可視性

結果セット型	内部的なDELETEの参照が可能か	内部的なUPDATEの参照が可能か	内部的なINSERTの参照が可能か	外部的なDELETEの参照が可能か	外部的なUPDATEの参照が可能か	外部的なINSERTの参照が可能か
forward-only	不可	可	不可	不可	不可	不可
scroll-sensitive	可	可	不可	不可	可	不可
scroll-insensitive	可	可	不可	不可	不可	不可



ノート:

- `refreshRow` メソッドの明示的な使用は、外部変更の可視性の概念とは別です。
- scroll-sensitive 結果セットを基礎とする `UPDATE` 操作のように、外部変更が可視であるときでも、これらは検出されません。結果セットの `rowDeleted`、`rowUpdated` および `rowInserted` メソッドは常に `false` を戻します。

17.6.3 Scroll-Sensitive結果セットのOracle実装

scroll-sensitive結果セットのOracle実装にはウィンドウの概念があり、フェッチ・サイズに基づくウィンドウ・サイズを持ちます。ウィンドウ・サイズは、結果セット内で行が更新される頻度に影響を及ぼします。

指定された行へ移動することによって現在の行を確定した場合、ウィンドウは結果セットのその行から始まるn行です。ここで、nは、結果セットによって使用されているフェッチ・サイズです。結果セットが先に作成された場合、現在の行がなく、したがってウィンドウもない点に注意してください。デフォルトの位置は最初の行の前です。これは有効な現在の行ではありません。

行を移動するとき、現在行がウィンドウ内にある間は、ウィンドウは変更されません。しかし、ウィンドウの外にある新しい現在行に移動すると、新しい現在行から始まるN行のウィンドウが再定義されます。

ウィンドウが再定義されると、`refreshRow`メソッドが暗黙的にコールされ、新しいウィンドウの行に対応するデータベースのN行が自動的に再フェッチされます。これにより、新しいウィンドウのデータが更新されます。

そのため、外部更新は、scroll-sensitive結果セットですぐには参照できません。前述のように、自動再フェッチの後で参照できるようになります。



ノート:

この種の再フェッチは、高度に効率的または最適化されている方法論ではなく、パフォーマンスについては無視できない不安があります。現在実装されているような scroll-sensitive 結果セットを使用する場合は、あらかじめ慎重に



検討してください。sensitivity とパフォーマンスの間にも重大なトレードオフがあります。最も sensitive な結果セットはフェッチ・サイズが 1 であるものです。この場合、行を移動するたびに新しい現在の行が再フェッチされます。ただし、これはアプリケーションのパフォーマンスに重大な影響を及ぼします。

18 JDBC RowSet

この章の構成は、次のとおりです。

- JDBC RowSetの概要
- CachedRowSetについて
- JdbcRowSetについて
- WebRowSetについて
- FilteredRowSetについて
- JoinRowSetについて

18.1 JDBC RowSetの概要

RowSetは、Java Database Connectivity(JDBC)の結果セットまたは表形式データソースの一連の行をカプセル化するオブジェクトです。RowSetは、JavaBeansなどのコンポーネント・ベースの開発モデル、一連の標準プロパティおよびイベント通知メカニズムをサポートしています。

RowSetsは、JDBC 2.0で、オプションのパッケージによって導入されました。ただし、RowSetsの実装はJDBC RowSet Implementations Specification(JSR-114)で標準化され、Java Platform, Standard Edition(Java SE)5.0以降、オプションでないパッケージとして利用できます。Java SE 6.0 RowSetsには、RowIdや各国語文字セットなどのような機能をサポートするAPIがより多く含まれます。Java SE Javadocには、JDBC RowSetの実装のための、標準インタフェースと基本クラスに関する情報が含まれています。

JSR-114仕様には、次の5種類のRowSetの実装詳細が記載されています。

- `CachedRowSet`
- `JdbcRowSet`
- `WebRowSet`
- `FilteredRowSet`
- `JoinRowSet`

Oracle JDBCは、5種類のRowSetすべてを、`oracle.jdbc.rowset`パッケージにあるインタフェースおよびクラスを介してサポートしています。Oracle Database 11gリリース1以降、サーバー側ドライバにRowSetsサポートが追加されました。このため、Oracle Database 11gリリース1からは、すべてのOracle JDBCドライバの型にわたり、RowSetサポートは均一です。標準のOracle JDBC Java Archive(JAR)ファイル、たとえば、`ojdbc6.jar`および`ojdbc7.jar`には`oracle.jdbc.rowset`パッケージが含まれます。



ノート:

- 接尾辞名の異なる他の JAR ファイル、たとえば、`ojdbc6_g.jar` および `ojdbc6dms.jar` にも、`oracle.jdbc.rowset` パッケージが含まれます。

- Oracle Database 10g リリース 2 では、実装クラスは ojdbc14.jar ファイルにパッケージ化されていました。
- Oracle Database 10g リリース 2 より前では、実装クラスは ojdbc12.jar ファイルにパッケージ化されていました。
- Oracle Database 11g リリース 1 より前は、RowSets サポートはサーバー側ドライバでは利用できませんでした。

ノート:



Oracle Database 10g リリース 2 (10.2)では、このパッケージは標準 Oracle JDBC JAR ファイル (classes12.jar、ojdbc5.jar および ojdbc6.jar)に含まれています。Oracle Database 10g リリース 2 (10.2)より前では、row set 実装クラスは ocrs12.jar ファイルにパッケージ化されていました。

Oracle RowSet実装を使用するには、必要なRowSet型のパッケージからoracle.jdbc.rowsetパッケージ全体、または特定のクラスおよびインタフェースをインポートする必要があります。またクライアント側では、CLASSPATH環境変数にojdbc6.jarまたはojdbc7.jarなどのOracle JARファイルをインクルードする必要があります。

この項の内容は次のとおりです。

- RowSetのプロパティ
- イベントおよびイベント・リスナー
- コマンド・パラメータおよびコマンド実行
- RowSetの横断について

18.1.1 RowSetのプロパティ

javax.sql.RowSetインタフェースは、単一のインタフェースでデータソースのデータにアクセスするために変更できるJavaBeansプロパティ一式を提供します。プロパティには、たとえば接続文字列、ユーザー名、パスワード、接続の種類および問合せ文字列があります。

関連項目:

プロパティおよびプロパティの説明の完全なリストは、http://docs.oracle.com/javase/1.5.0/docs/api/javax/sql/RowSet.htmlのJava2 Platform, Standard Edition (J2SE)のJavadocを参照

このインタフェースは、プロパティ値の設定および取出しに必要な標準アクセッサ・メソッドを提供します。次のコードは、一部のRowSetプロパティを設定します。

```
...
rowset.setUrl("jdbc:oracle:oci:@");
```

```
rowset.setUsername("HR");
rowset.setPassword("hr");
rowset.setCommand("SELECT employee_id, first_name, last_name, salary FROM employees");
...
```

この例では、URL、ユーザー名、パスワードおよびSQL問合せが、従業員番号、従業員名および全従業員の給与をRowSetオブジェクト内に取り出すためのRowSetプロパティとして設定されています。

18.1.2 イベントおよびイベント・リスナー

RowSetはJavaBeansイベントをサポートします。RowSetインタフェースは、次のタイプのイベントをサポートします。

- cursorMoved

  このイベントはカーソルが移動するたびに生成されます。たとえば、nextメソッドまたはpreviousメソッドがコールされると生成されます。

- rowChanged

  このイベントは、RowSetに行が挿入されるか、RowSetの行が更新されるか、またはRowSetから行が削除されると生成されます。

- rowSetChanged

  このイベントは、RowSet全体が作成または変更されると生成されます。たとえば、executeメソッドがコールされると生成されます。

アプリケーション・コンポーネントでは、RowSetリスナーを実装し、これらのRowSetイベントをリスニングして、イベントの発生時に必要な操作を実行できます。これらのイベントに関係のあるアプリケーション・コンポーネントの場合は、標準のjavax.sql.RowSetListenerインタフェースを実装し、RowSetオブジェクトにこれらのリスナー・オブジェクトを登録する必要があります。リスナーの登録にはRowSet.addRowSetListenerメソッドを使用し、登録を解除するにはRowSet.removeRowSetListenerメソッドを使用します。複数のリスナーを同じRowSetオブジェクトに登録できます。

次のコードは、RowSetリスナーを登録します。

```
...
MyRowSetListener rowsetListener = new MyRowSetListener ();
// adding a rowset listener
rowset.addRowSetListener (rowsetListener);
...
```

次のコードは、リスナーを実装します。

```
public class MyRowSetListener implements RowSetListener
 {
   public void cursorMoved(RowSetEvent event)
   {
     // action on cursor movement
   }

   public void rowChanged(RowSetEvent event)
   {
     // action on change of row
   }

   public void rowSetChanged(RowSetEvent event)
```

```
    {
      // action on changing of rowset
    }
  }// end of class MyRowSetListener
```

選択されたイベントのみを処理するアプリケーションの場合、すべてのイベント処理メソッド用の空の実装を持つ抽象クラスであるoracle.jdbc.rowset.OracleRowSetListenerAdapterクラスを使用することにより、必要なイベント処理メソッドのみを実装することができます。次のコードでは、rowSetChangedイベントのみが処理され、残りのイベントは処理されません。

```
...
rowset.addRowSetListener(new oracle.jdbc.rowset.OracleRowSetListenerAdapter ()
    {
      public void rowSetChanged(RowSetEvent event)
       {
         // your action for rowSetChanged
       }
    }
  );
...
```

18.1.3 コマンド・パラメータおよびコマンド実行

RowSetオブジェクトのcommandプロパティは、通常はSQL問合せ文字列を表し、処理時に実際のデータをRowSetオブジェクトに移入します。通常のJDBC処理と同様に、この問合せ文字列は入力やバインド・パラメータをとることができます。javax.sql.RowSetインタフェースにも、このSQL問合せに入力パラメータを設定するメソッドがあります。必要な入力パラメータが設定された後でSQL問合せが処理され、基礎となるデータソースのデータをRowSetオブジェクトに移入できます。次のコードは、この単純な手順を示しています。

```
...
 rowset.setCommand("SELECT first_name, last_name, salary FROM employees WHERE employee_id = ?");
 // setting the employee number input parameter for employee named "Douglas"
 rowset.setInt(1, 199);
 rowset.execute();
 ...
```

前述の例では、従業員番号199が、RowSetオブジェクトのcommandプロパティで指定されたSQL問合せの入力パラメータまたはバインド・パラメータとして設定されています。SQL問合せが処理されると、RowSetオブジェクトは、従業員番号199の従業員の従業員名と給与情報で満たされます。

18.1.4 RowSetの横断について

javax.sql.RowSetインタフェースは、java.sql.ResultSetインタフェースを拡張します。そのため、RowSetインタフェースでは、カーソル移動および位置指定の各メソッドがResultSetインタフェースから継承され、RowSetオブジェクト内のデータの横断に使用されます。継承されるメソッドの中には、absolute、beforeFirst、afterLast、nextおよびpreviousがあります。

RowSetインタフェースは、データを取得および更新するためのResultSetインタフェースと同じように使用できます。RowSetインタフェースでは、オプションでスクロール可能および更新可能な結果セットを実装できます。ResultSetインタフェースに含まれるすべてのフィールドおよびメソッドが、RowSetに実装されます。



ノート:

java.sql.ResultSet インタフェースのスクロール可能で更新可能なプロパティは、ResultSet の Oracle 実装でも提供されます。

次のコードは、RowSetをスクロールする方法を示します。

```
/**
 * Scrolling forward, and printing the empno in
 * the order in which it was fetched.
 */
...
rowset.setCommand("SELECT empno, ename, sal FROM emp");
rowset.execute();
...
// going to the first row of the rowset
rowset.beforeFirst ();
while (rowset.next ())
  System.out.println ("empno: " +rowset.getInt (1));
```

前述のコードでは、beforeFirstメソッドによって、カーソル位置がRowSetの最初の行の前に初期化されます。行は、nextメソッドを使用して前方方向に取り出されます。

次のコードは、RowSetを逆方向にスクロールする方法を示します。

```
/**
 * Scrolling backward, and printing the empno in
 * the reverse order as it was fetched.
 */
//going to the last row of the rowset
rowset.afterLast ();
while (rowset.previous ())
  System.out.println ("empno: " +rowset.getInt (1));
```

前述の例では、RowSetの最後の行の後にカーソル位置が初期化されます。行は、RowSetのpreviousメソッドを使用して逆方向に取り出されます。

行の挿入、更新および削除は、結果セット機能と同様に行セット機能でもサポートされます。行セットを更新可能にするには、setReadOnly(false)およびacceptChangesの各メソッドをコールする必要があります。

次のコードは、行セットの5番目の位置に1行を挿入します。

```
...
/**
  * Make rowset updatable
  */
rowset.setReadOnly (false);
/**
 * Inserting a row in the 5th position of the rowset.
 */
// moving the cursor to the 5th position in the rowset
if (rowset.absolute(5))
{
```

```
  rowset.moveToInsertRow ();
  rowset.updateInt (1, 193);
  rowset.updateString (2, "Smith");
  rowset.updateInt (3, 7200);
  // inserting a row in the rowset
  rowset.insertRow ();
  // Synchronizing the data in RowSet with that in the database.
  rowset.acceptChanges ();
}
...
```

前述の例では、パラメータ5が指定されたabsoluteメソッドをコールすることにより、RowSetの5番目の位置にカーソルが移動し、moveToInsertRowメソッドをコールすることにより、RowSetに新しい行を挿入するための場所が作成されます。新しく作成された行を更新するには、updateXXXメソッドを使用します。行のすべての列が更新されると、insertRowがコールされ、そのRowSetが更新されます。変更はacceptChangesメソッドを使用してコミットします。

18.2 CachedRowSetについて

CachedRowSetは、行がキャッシュされ、切断されている(つまり、データベースとのアクティブ接続を持たない)RowSetです。oracle.jdbc.rowset.OracleCachedRowSetクラスは、CachedRowSetのOracle実装です。これは標準のリファレンス実装と相互運用できます。ojdbc6.jarファイルおよびojdbc7.jarファイルのOracleCachedRowSetクラスは、標準のJSR-114インタフェースjavax.sql.rowset.CachedRowSetを実装します。

次のコードでは、OracleCachedRowSetオブジェクトを作成し、プロパティとしてRowSetオブジェクトの接続URL、ユーザー名、パスワードおよびSQL問合せを設定します。RowSetオブジェクトには、executeメソッドを使用してデータが移入されます。executeメソッドを処理した後、java.sql.ResultSetオブジェクトとしてRowSetオブジェクトを使用できるため、データの取得、スクロール、挿入、削除または更新を実行できます。

```
...
RowSet rowset = new OracleCachedRowSet();
rowset.setUrl("jdbc:oracle:oci:@");
rowset.setUsername("HR");
rowset.setPassword("hr");
rowset.setCommand("SELECT employee_id, first_name, last_name, salary FROM employees");
rowset.execute();
while (rowset.next ())
{
  System.out.println("employee_id: " +rowset.getInt (1));
  System.out.println("first_name: " +rowset.getString (2));
  System.out.println("last_name: " +rowset.getString (3));
  System.out.println("sal: "    +rowset.getInt (4));
}
...
```

問合せによりCachedRowSetオブジェクトにデータを移入するには、次のステップを実行します。

1. OracleCachedRowSetをインスタンス化します。
2. Url(接続URL)、Username、PasswordおよびCommand(問合せ文字列で、RowSetオブジェクトのプロパティ)を設定します。接続型を設定することもできますが、それはオプションです。

3. executeメソッドをコールして、CachedRowSetオブジェクトにデータを移入します。executeのコールにより、問合せセットがこのRowSetのプロパティとして実行されます。

```
OracleCachedRowSet rowset = new OracleCachedRowSet ();
rowset.setUrl ("jdbc:oracle:oci:@");
rowset.setUsername ("HR");
rowset.setPassword ("hr");
rowset.setCommand ("SELECT employee_id, first_name, last_name, salary FROM employees");
rowset.execute ();
```

populateメソッドを使用して、CachedRowSetオブジェクトに既存のResultSetオブジェクトを移入できます。これを行うには、次のステップを完了します。

1. OracleCachedRowSetをインスタンス化します。

2. すでに使用可能なResultSetオブジェクトをpopulateメソッドに渡して、RowSetオブジェクトにデータを移入します。

```
// Executing a query to get the ResultSet object.
ResultSet rset = pstmt.executeQuery ();

OracleCachedRowSet rowset = new OracleCachedRowSet ();
// the obtained ResultSet object is passed to the populate method
// to populate the data in the rowset object.
rowset.populate (rset);
```

前述の例では、問合せを実行してResultSetオブジェクトを取得し、そのResultSetオブジェクトがCachedRowSetオブジェクトのpopulateメソッドに渡されて、結果セットの内容がCachedRowSetに移入されています。

ノート:



既存の ResultSet オブジェクトを CachedRowSet オブジェクトへのデータの移入に使用する場合には、プロパティが適用される接続または結果セットがすでに作成されているため、トランザクションの分離や結果セットの同時実行性モードなどの接続プロパティおよびバインド・プロパティを設定することはできません。

次のコードは、OracleCachedRowSetオブジェクトをファイルに対してシリアライズしてから取得します。

```
// writing the serialized OracleCachedRowSet object
{
  FileOutputStream fileOutputStream = new FileOutputStream("emp_tab.dmp");
  ObjectOutputStream ostream = new ObjectOutputStream(fileOutputStream);
  ostream.writeObject(rowset);
  ostream.close();
  fileOutputStream.close();
}
// reading the serialized OracleCachedRowSet object
{
  FileInputStream fileInputStream = new FileInputStream("emp_tab.dmp");
  ObjectInputStream istream = new ObjectInputStream(fileInputStream);
  RowSet rowset1 = (RowSet) istream.readObject();
  istream.close();
  fileInputStream.close();
}
```

前述のコードでは、emp_tab.dmpファイル用にFileOutputStreamオブジェクトがオープンされ、データを移入されたOracleCachedRowSetオブジェクトがObjectOutputStreamを使用してこのファイルに書き込まれます。シリアライズされたOracleCachedRowSetオブジェクトは、FileInputStreamおよびObjectInputStreamオブジェクトを使用して取り出されます。

InputStream、OutputStream、バイナリ・ラージ・オブジェクト(BLOB)およびキャラクタ・ラージ・オブジェクト(CLOB)などの非シリアライズ形式のデータは、OracleCachedRowSetによってシリアライズされます。また、OracleCachedRowSetsは独自のメタデータを実装するため、追加のラウンドトリップを必要とせずに取得できます。次のコードは、RowSetのメタデータを取得する方法を示します。

```
...
ResultSetMetaData metaData = rowset.getMetaData();
int maxCol = metaData.getColumnCount();
for (int i = 1; i <= maxCol; ++i)
   System.out.println("Column (" + i +") " + metaData.getColumnName(i));
...
```

OracleCachedRowSetクラスはシリアライズ可能なので、Remote Method Invocation(RMI)の場合と同様に、ネットワークや別のJava仮想マシン(JVM)の間で渡すことができます。OracleCachedRowSetクラスにデータが移入されると、任意のJVMまたはJDBCドライバを使用していない環境での移動が可能になります。RowSetにデータをコミットするには、JDBCドライバがインストールされている必要があります。

データを取得して、OracleCachedRowSetクラスでそれにデータを移入する処理の全体はサーバーで実行され、データを移入されたRowSetは、RMIやEnterprise Java Beans(EJB)などの適切なアーキテクチャを使用してクライアントに渡されます。クライアントは、取得、スクロール、挿入、更新および削除などすべての操作を、RowSet上で、データベースに接続しないで実行できます。データベースにデータがコミットされる場合は常に、RowSet内のデータとデータベース内のデータを同期させるacceptChangesメソッドがコールされます。このメソッドはJDBCドライバを利用するため、JVM環境にJDBC実装が含まれる必要があります。このアーキテクチャは、携帯情報端末(PDA)などのThinクライアントが関係するシステムに適しています。

CachedRowSetオブジェクトにデータを移入すると、このオブジェクトをResultSetオブジェクト、またはRMIやその他の適切なアーキテクチャを使用してネットワークを通じて渡すことが可能な他のオブジェクトとして使用できます。

CachedRowSetには、この他に次のような主要機能があります。

- RowSetのクローニング
- RowSetのコピーの作成
- RowSetの共有コピーの作成

CachedRowSetの制約事項

OracleCachedRowSetはシリアライズ可能なので、更新可能な結果セットに適用される、シリアライズ以外のすべての制約をここで適用できます。SQL問合せには、次の制約があります。

- データベース内の1つの表のみを参照できます。
- 結合操作は含まれません。
- 参照先の表の主キーを選択します。

また、新しい行を挿入するには、SQL問合せが次の条件を満たしている必要があります。

- 基礎となる表のNULL化可能でない列すべてを選択します。
- デフォルト値のないすべての列を選択します。

ノート:

データはすべてメモリーにキャッシュされるため、CachedRowSet には大量のデータを格納できません。したがって、大量のデータを戻す可能性のある OracleCachedRowSet を問合せで使用しないことをお薦めします。

結果セットのトランザクションの分離や同時実行性モードなど接続プロパティは、RowSetへのデータの移入後には設定できません。このプロパティは、データを取り出した後には接続に適用できないためです。

18.3 JdbcRowSetについて

JdbcRowSetは、ResultSetオブジェクトをラップするRowSetです。これは、接続RowSetであり、JavaBeanインタフェース形式のJDBCインタフェースを提供します。JdbcRowSetのOracle実装は、oracle.jdbc.rowset.OracleJDBCRowSetです。ojdbc6.jarファイルおよびojdbc7.jarファイルのOracleJDBCRowSetクラスは、標準のJSR-114インタフェースjavax.sql.rowset.JdbcRowSetを実装します。

表18-1は、JdbcRowSetインタフェースとCachedRowSetインタフェースの違いを示しています。

表18-1 JDBCのRowSetとCachedRowSetの比較

RowSetのタイプ	シリアライズ可能	データベースへの接続	JVM内での移動が可能	データベースに対するデータの同期化	JDBCドライバの存在
JDBC	あり	あり	なし	なし	あり
キャッシュ	あり	なし	あり	あり	なし

JdbcRowSetは接続されているRowSetで、データベースへのライブ接続を持ち、JdbcRowSet上のすべてのコールは、JDBC接続、文または結果セットのマッピング・コールに反映されます。CachedRowSetは、開いているデータベースへの接続をいっさい持ちません。

JdbcRowSetではJDBCドライバが存在する必要があるのに対して、CachedRowSetの場合は、操作時にJDBCドライバが存在する必要はありません。ただし、JdbcRowSetおよびCachedRowSetでは、RowSetへのデータ移入時とRowSetの変更をコミットするときには、JDBCドライバが必要です。

次のコードは、JdbcRowSetの使用方法を示します。

```
...
RowSet rowset = new OracleJDBCRowSet();
rowset.setUrl("java:oracle:oci:@");
rowset.setUsername("HR");
rowset.setPassword("hr");
```

```
rowset.setCommand("SELECT empno, ename, sal FROM emp");
rowset.execute();
while (rowset.next())
{
  System.out.println("empno: " + rowset.getInt(1));
  System.out.println("ename: " + rowset.getString(2));
  System.out.println("sal: " + rowset.getInt(3));
}
...
```

前述の例では、接続URL、ユーザー名、パスワードおよびSQL問合せがRowSetオブジェクトのプロパティとして設定され、SQL問合せがexecuteメソッドを使用して処理され、行がRowSetオブジェクトに移入されたデータを横断して取得され出力されます。

18.4 WebRowSetについて

WebRowSetは、CachedRowSetの拡張機能です。このRowSetは一連のフェッチされる行または表データを表し、データソースとのアクティブな接続を保持せずに、各層およびコンポーネント間で渡すことができます。WebRowSetインタフェースは、結果セットの生成と使用をサポートし、結果セットとデータソースとの同期をExtensible Markup Language(XML)形式および接続切断方式の両方でサポートします。これにより、層と層をまたがり、またインターネット・プロトコルを介して、結果セットを移送できます。

WebRowSetのOracle実装は、oracle.jdbc.rowset.OracleWebRowSetです。このクラスは、ojdbc6.jarファイルおよびojdbc7.jarファイル内にあり、標準のJSR-114インタフェースjavax.sql.rowset.WebRowSetを実装します。このクラスは、oracle.jdbc.rowset.OracleCachedRowSetクラスも拡張します。OracleWebRowSetクラスは、OracleCachedRowSetクラスで使用可能なメソッドの他に、次のメソッドも提供します。

- `public OracleWebRowSet() throws SQLException`

  これは、OracleWebRowSetオブジェクトの作成に使用するコンストラクタであり、OracleCachedRowSetオブジェクトのデフォルト値、デフォルトのOracleWebRowSetXmlReaderおよびデフォルトのOracleWebRowSetXmlWriterによって初期化されます。

- ```
  public void writeXml(java.io.Writer writer) throws SQLException
  public void writeXml(java.io.OutputStream ostream) throws SQLException
  ```

  これらのメソッドは、OracleWebRowSetオブジェクトを、提供されたWriterまたはOutputStreamオブジェクトにJSR-114 XMLスキーマ準拠のXML形式で書き込みます。RowSetデータの他に、RowSetのプロパティとメタデータも書き込まれます。

- ```
  public void writeXml(ResultSet rset, java.io.Writer writer) throws SQLException
  public void writeXml(ResultSet rset, java.io.OutputStream ostream) throws SQLException
  ```

  これらのメソッドは、OracleWebRowSetオブジェクトを作成し、そのオブジェクトに指定したResultSetオブジェクトのデータを移入し、提供されたWriterまたはOutputStreamオブジェクトにJSR-114 XMLスキーマ準拠のXML形式で書き込みます。

- ```
  public void readXml(java.io.Reader reader) throws SQLException
  public void readXml(java.io.InputStream istream) throws SQLException
  ```

  これらのメソッドは、提供されたReaderまたはInsputStreamオブジェクトを使用して、OracleWebRowSetオブジェクトを
  ```

JSR-114 XMLスキーマ準拠のXML形式で読み取ります。

Oracle WebRowSet実装は、Java API for XML Processing (JAXP) 1.2をサポートしています。Simple API for XML (SAX) 2.0とDocument Object Model (DOM) JAXPのどちらに準拠しているXMLパーサーもサポートされます。WebRowSetの現在のJSR-114 W3C XMLスキーマに準拠します。

readXml(...)メソッドを使用するアプリケーションの場合は、メソッドのコール前に、次の2つの標準JAXPシステム・プロパティのいずれかを設定する必要があります。

- javax.xml.parsers.SAXParserFactory

  このプロパティは、SAXパーサー用です。

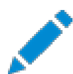
- javax.xml.parsers.DocumentBuilderFactory

  このプロパティは、DOMパーサー用です。

次のコードでは、XML形式での書込みおよび読取り両方にOracleWebRowSetを使用します。

```
import java.sql.*;
import java.io.*;
import oracle.jdbc.rowset.*;
...
String url = "jdbc:oracle:oci8:@";
Connection conn = DriverManager.getConnection(url,"HR","hr");
Statement stmt = conn.createStatement();
ResultSet rset = stmt.executeQuery("select * from employees");
// Create an OracleWebRowSet object and populate it with the ResultSet object
OracleWebRowSet wset = new OracleWebRowSet();
wset.populate(rset);
try
{
  // Create a java.io.Writer object
  FileWriter out = new FileWriter("xml.out");

  // Now generate the XML and write it out
  wset.writeXml(out);
}
catch (IOException exc)
{
  System.out.println("Couldn't construct a FileWriter");
}
System.out.println("XML output file generated.");
// Create a new OracleWebRowSet for reading from XML input
OracleWebRowSet wset2 = new OracleWebRowSet();
// Use Oracle JAXP SAX parser
System.setProperty("javax.xml.parsers.SAXParserFactory","oracle.xml.jaxp.JXSAXParserFactory");
try
{
  // Use the preceding output file as input
  FileReader fr = new FileReader("xml.out");

  // Now read XML stream from the FileReader
  wset2.readXml(fr);
}
catch (IOException exc)
{
  System.out.println("Couldn't construct a FileReader");
}
```

```
...
```



ノート:

前述のコードでは、スキーマの妥当性チェックをサポートする Oracle SAX XML パーサーを使用しています。

18.5 FilteredRowSetについて

FilteredRowSetは、WebRowSetの拡張機能であり、プログラム内容のフィルタリングに必要なプログラム・サポートを提供します。これにより、問合せの指定および関連処理に要するオーバーヘッドを回避できます。FilteredRowSetのOracle実装は、oracle.jdbc.rowset.OracleFilteredRowSetです。ojdbc7.jarファイルのOracleFilteredRowSetクラスは、標準のJSR-114インタフェースjavax.sql.rowset.FilteredRowSetを実装します。

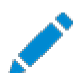
OracleFilteredRowSetクラスは、次の新しいメソッドを定義します。

- `public Predicate getFilter();`

  このメソッドは、OracleFilteredRowSetオブジェクトでアクティブなフィルタリング基準を定義するPredicateオブジェクトを戻します。

- `public void setFilter(Predicate p) throws SQLException;`

  このメソッドは、Predicateオブジェクトをパラメータとして取ります。Predicateオブジェクトは、OracleFilteredRowSetオブジェクトに適用するフィルタリング基準を定義します。このメソッドでは、SQLException例外が発生します。



ノート:

FilteredRowSet 機能に対して ojdbc5.jar および ojdbc6.jar のかわりに classes12.jar を使用している場合、Predicate のかわりに OraclePredicate を使用します。Oracle 固有の oracle.jdbc.rowset.OraclePredicate インタフェースは、Predicate と同等です。このインタフェースは、JSR-114 パッケージを使用できない場合に使用します。

OracleFilteredRowSetオブジェクトに設定される条件は、参照できる行のセットを取得するためにオブジェクト内の行すべてに適用するフィルタリング基準を定義します。また、行の挿入、削除および変更に適用する基準も定義します。設定されたフィルタリング基準は、OracleFilteredRowSetオブジェクトの参照および更新すべてに適用されるゲーティング・メカニズムとして機能します。OracleFilteredRowSetオブジェクトを更新しようとすると、フィルタリング基準に違反するため、SQLException例外が発生します。

OracleFilteredRowSetオブジェクトに設定されているフィルタリング基準を変更するには、新しいPredicateオブジェクトを適用します。新しい基準は即時にオブジェクトに適用されるため、適用以後の参照および更新では、すべてこの新しい基準に準拠する必要があります。新しいフィルタリング基準を適用できるのは、OracleFilteredRowSetオブジェクトへの参照がない場合のみです。

オブジェクトに設定されたフィルタリング基準の範囲外の行は、そのフィルタリング基準が削除されるか、新しいフィルタリング基準が適用されるまで変更できません。また、このオブジェクトの同期化を続行しても、データソースと同期化される行は、フィルタリング基準の範囲内の行のみです。

次のコードは、OracleFilteredRowSetの使用例を示します。表test_tableに、NUMBER型のcol1およびcol2という2つの列があるものとします。この表からコードが取り出す行には、col1に50から100までの値、col2に100から200までの値が含まれています。

フィルタリング基準を定義する条件を次に示します。

```
public class PredicateImpl implements Predicate
{
  private int low[];
  private int high[];
  private int columnIndexes[];

  public PredicateImpl(int[] lo, int[] hi, int[] indexes)
  {
    low = lo;
    high = hi;
    columnIndexes = indexes;
  }

  public boolean evaluate(RowSet rs)
  {
    boolean result = true;
    for (int i = 0; i < columnIndexes.length; i++)
    {
      int columnValue = rs.getInt(columnIndexes[i]);
      if (columnValue < low[i] || columnValue > high[i])
        result = false;
    }
    return result;
  }
// the other two evaluate(...) methods simply return true
}
```

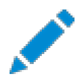
前述のコードで定義した条件を使用して、OracleFilteredRowSetオブジェクトの内容を次のようにフィルタリングします。

```
...
OracleFilteredRowSet ofrs = new OracleFilteredRowSet();
int low[] = {50, 100};
int high[] = {100, 200};
int indexes[] = {1, 2};
ofrs.setCommand("select col1, col2 from test_table");
// set other properties on ofrs like usr/pwd ...
...
ofrs.execute();
ofrs.setPredicate(new PredicateImpl(low, high, indexes));
// this will only get rows with col1 in (50,100) and col2 in (100,200)
while (ofrs.next()) {...}
...
```

18.6 JoinRowSetについて

JoinRowSetは、WebRowSetの拡張機能であり、異なるRowSetの関連データで構成されています。データソースに接続していない状態では、切断されたRowSet間でSQL JOINを確立する標準的な方法はありません。JoinRowSetはこの問題に対応しています。JoinRowSetのOracle実装は、oracle.jdbc.rowset.OracleJoinRowSetクラスです。このクラスは、ojdbc7.jarファイル内にあり、標準のJSR-114インタフェースjavax.sql.rowset.JoinRowSetを実装します。

Joinableインタフェースを実装する任意の数のRowSetオブジェクトは、SQL JOINに関連付けることが可能な場合、JoinRowSetオブジェクトに追加できます。RowSetの5つのタイプはすべて、Joinableインタフェースをサポートします。Joinableインタフェースには、JOINを実行するときに使用する列(一致する列)を指定するためのメソッドがあります。



ノート:

JoinRowSet 機能に対して ojdbc5.jar および ojdbc6.jar のかわりに classes12.jar を使用している場合、Joinable のかわりに OracleJoinable を使用します。Oracle 固有の oracle.jdbc.rowset.OracleJoinable インタフェースは、Joinable と同等です。このインタフェースは、JSR-114 パッケージを使用できない場合に使用します。

一致する列は次の方法で指定できます。

- setMatchColumnメソッドの使用

  このメソッドは、Joinableインタフェースで定義されます。RowSetオブジェクトをJoinRowSetオブジェクトに追加する前に、一致する列の設定に使用できるのは、このメソッドのみです。また、一致する列の再設定にいつでも使用できます。

- addRowSetメソッドの使用

  このメソッドは、JoinRowSetでオーバーロードされます。このメソッドの5つの実装のうち4つは、一致する列をパラメータとして取ります。この4つのメソッドを使用して、RowSetオブジェクトをJoinRowSetオブジェクトに追加するときに、一致する列を設定または再設定できます。

OracleJoinRowSetは、継承されるメソッド以外に、次のメソッドを提供します。

- ```
  public void addRowSet(Joinable joinable) throws SQLException;
  public void addRowSet(RowSet rowSet, int i) throws SQLException;
  public void addRowSet(RowSet rowSet, String s) throws SQLException;
  public void addRowSet(RowSet arowSet[], int an[]) throws SQLException;
  public void addRowSet(RowSet arowSet[], String as[]) throws SQLException;
  ```

  これらのメソッドを使用して、RowSetオブジェクトをOracleJoinRowSetオブジェクトに追加します。1つ以上のRowSetオブジェクトをOracleJoinRowSetオブジェクトに渡して追加できます。一致する列として設定する必要がある1つ以上の列の名前または索引を渡すこともできます。

- ```
  public Collection getRowSets() throws SQLException;
  ```

  このメソッドは、OracleJoinRowSetに追加されたRowSetオブジェクトを取り出します。また、RowSetオブジェクトを含むjava.util.Collectionオブジェクトを戻します。

- ```
  public String[] getRowSetNames() throws SQLException;
  ```
  ```

このメソッドは、OracleJoinRowSetオブジェクトに追加されるRowSetオブジェクトの名前を含む文字列の配列を戻します。

- ```
  public boolean supportsCrossJoin();
  public boolean supportsFullJoin();
  public boolean supportsInnerJoin();
  public boolean supportsLeftOuterJoin();
  public boolean supportsRightOuterJoin();
  ```

  これらのメソッドは、OracleJoinRowSetオブジェクトが、対応するJOIN型をサポートしているかどうかを示すブール値を戻します。

- `public void setJoinType(int i) throws SQLException;`

  このメソッドを使用して、OracleJoinRowSetオブジェクトにJOIN型を設定します。このメソッドは、javax.sql.rowset.JoinRowSetインタフェースで定義されたJOIN型を指定する整数定数を取ります。

- `public int getJoinType() throws SQLException;`

  このメソッドは、OracleJoinRowSetオブジェクトに設定されたJOIN型を示す整数値を戻します。このメソッドでは、SQLException例外が発生します。

- `public CachedRowSet toCachedRowSet() throws SQLException;`

  このメソッドは、OracleJoinRowSetオブジェクトのデータを含むCachedRowSetオブジェクトを作成します。

- `public String getWhereClause() throws SQLException;`

  このメソッドは、OracleJoinRowSetオブジェクトで使用されるWHERE句のSQLなどの記述を含む文字列を戻します。このメソッドでは、SQLException例外が発生します。

次のコードは、OracleJoinRowSetを使用して、2つの異なる表のデータを含む2つのRowSetで内部結合を実行します。結合されたRowSetには、2つの表で内部結合が実行された場合と同じデータが含まれます。表が2つあり、その一方はNUMBER型の2つの列Order_idおよびPerson_idを備えたOrder表、他方はNUMBER型の列Person_idとVARCHAR2型の列Nameを備えたPerson表であるものとします。

```
...
// RowSet holding data from table Order
OracleCachedRowSet ocrsOrder = new OracleCachedRowSet();
...
ocrsOrder.setCommand("select order_id, person_id from order");
...
// Join on person_id column
ocrsOrder.setMatchColumn(2);
ocrsOrder.execute();
// Creating the JoinRowSet
OracleJoinRowSet ojrs = new OracleJoinRowSet();
ojrs.addRowSet(ocrsOrder);
// RowSet holding data from table Person
OracleCachedRowSet ocrsPerson = new OracleCachedRowSet();
...
ocrsPerson.setCommand("select person_id, name from person");
...
// do not set match column on this RowSet using setMatchColumn().
//use addRowSet() to set match column
ocrsPerson.execute();
```
```

```
// Join on person_id column, in another way
ojrs.addRowSet(ocrsPerson, 1);
// now we can go the JoinRowSet as usual
ojrs.beforeFirst();
while (ojrs.next())
System.out.println("order id = " + ojrs.getInt(1) + ", " + "person id = " +
ojrs.getInt(2) + ", " + "person's name = " + ojrs.getString(3));
...
```

19 グローバリゼーション・サポート

Oracle Java Database Connectivity(JDBC)ドライバは、グローバリゼーション・サポート(以前のNational Language Support(NLS))を提供しています。グローバリゼーション・サポートにより、Oracleがサポートする任意の文字セットで、データの取出しやデータベースへのデータの挿入が可能です。クライアントとサーバーで異なる文字セットを使用する場合、ドライバでは、データベース文字セットとクライアント文字セット間の変換がサポートされます。

この章の構成は、次のとおりです。

- グローバリゼーション・サポートの提供について
- NCHAR、NVARCHAR2、NCLOBおよびdefaultNCharプロパティ
- JDK 6での各国語文字セット用の新しいメソッド

ノート:

- Oracle Database 10g 以上では、NLS_LANG 変数が JDBC グローバリゼーション・メカニズムに組み込まれていません。JDBC ドライバは、NLS 環境をチェックしません。そのため、この変数を設定しても効果はありません。
- JDBC サーバー側内部ドライバは完全なグローバリゼーション・サポートを提供し、グローバリゼーション拡張ファイルを必要としません。
- JDBC 4.0 には各国語文字セットの値を読書きするためのメソッドが含まれています。JSE 6 以上を使用する場合はこれらのメソッドを使用してください。

関連トピック

- Oracle文字データ型のサポート
- Oracle Databaseグローバリゼーション・サポート・ガイド

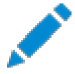
19.1 グローバリゼーション・サポートの提供について

基本Javaアーカイブ(JAR)ファイル(ojdbc7.jar)には、次の項目に対して完全なグローバリゼーション・サポートを提供するために必要なすべてのクラスが含まれています。

- Oracleオブジェクトまたはコレクション型のデータ・メンバーとして取得または挿入されないCHAR、VARCHAR、LONGVARCHARまたはCLOBデータ用のOracle文字セット
- 文字セットUS7ASCII、WE8DEC、WE8ISO8859P1、WE8MSWIN1252およびUTF8に対するオブジェクトおよびコレクションのCHARまたはVARCHARデータ・メンバー

前述以外の文字セットをオブジェクトまたはコレクションのCHARまたはVARCHARデータ・メンバーで使用するには、CLASSPATH環境変数にorai18n.jarを含める必要があります。

```
ORACLE_HOME/jlib/orai18n.jar
```



ノート:

以前のリリースでは、nls_charset12.zip ファイルが必要でした。このファイルは現在使用されません。

orai18n.jarの縮小

orai18n.jarファイルには、多数の重要な文字セットとグローバリゼーション・サポート・ファイルが含まれています。組込みのカスタマイズ・ツールを使用して、次のようにorai18n.jarファイルのサイズを縮小できます。

```
java -jar orai18n.jar -custom-charsets-jar [jar/zip_filename] -charset characterset_name
[characterset_name ...]
```

たとえば、文字セットJA16SJISおよびJA16EUCを含むカスタムの文字セット・ファイルcustom_orai18n_ja.jarを作成する場合は、次のコマンドを発行します。

```
$ java -jar orai18n.jar -custom-charsets-jar custom_orai18n_ja.jar -charset JA16SJIS JA16EUC
```

コマンドの出力は次のようになります。

```
Added Character set : JA16SJIS
Added Character set : JA16EUC
```

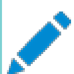
カスタムのJAR/ZIPファイルにファイル名を指定しない場合、jdbc_orai18n_cs.jarという名前のファイルが現行作業ディレクトリに作成されます。また、カスタムのJAR/ZIPファイルに、orai18nで始まる名前を指定することはできません。

無効またはサポートされていない文字セットをコマンドに指定した場合、JAR/ZIPファイルは出力されません。カスタムのJAR/ZIPファイルは更新または削除されません。

カスタムの文字セットJAR/ZIPはコマンドを受け入れません。ただし、バージョン情報およびJAR/ZIPファイルの生成に使用されたコマンドは出力されます。たとえば、jdbc_orai18n_cs.zipを設定した場合、情報を表示するコマンドおよび表示される情報は次のようになります。

```
$ java -jar jdbc_orai18n_cs.jar
Oracle Globalization Development Kit - 12.1.X.X.X Release
This custom character set jar/zip file was created with the following command:
java -jar orai18n.jar -custom-charsets-jar jdbc_orai18n_cs.jar -charset WE8ISO8859P15
```

指定できる文字セットの数は、オペレーティング・システムのシェルまたはコマンド・プロンプトの制限によって決まります。サポートされている文字セットはすべてコマンドで指定できます。



ノート:

カスタム文字セットを使用する場合、JDBC がカスタム文字セットをサポートするためには、次の操作を実行する必要がありま

1. カスタム文字セットの作成プロセスで.nlt および.nlb ファイルを作成した後で、次のコマンドを使用して、新たに作成
   ファイルを作成します。

```
java -classpath $ORACLE_HOME/jlib/orai18n.jar:$ORACLE_HOME/lib/xmlparserv2.jar Ginstall -[add
```

2. カスタム文字セットを使用してデータベースに接続するJDBCコードを実行する場合は、生成したファイルと$ORACLE_
環境変数に追加します。

19.2 NCHAR、NVARCHAR2、NCLOBおよびdefaultNCharプロパティ

デフォルトでは、oracle.jdbc.OraclePreparedStatementインタフェースにより、すべての列のデータ型は、列がデータベース文字セットでエンコードされるのと同じように処理されます。ただし、Oracle Database 10g以上では、oracle.jdbc.defaultNCharの値をtrueに設定すると、すべてのキャラクタ列がJDBCにより各国語として処理されます。

defaultNCharのデフォルト値はfalseです。defaultNCharの値がfalseである場合、特に各国語文字が必要な列のために、setFormOfUse(<column_Index>, OraclePreparedStatement.FORM_NCHAR)メソッドをコールする必要があります。たとえば:

```
PreparedStatement pstmt =
conn.prepareStatement("insert into TEST values(?,?,?)");
pstmt.setFormOfUse(1, OraclePreparedStatement.FORM_NCHAR);
pstmt.setString(1, myUnicodeString1); // NCHAR column
pstmt.setFormOfUse(2, OraclePreparedeStatement.FORM_NCHAR);
pstmt.setString(2, myUnicodeString2); // NVARCHAR2 column
```

defaultNCharの値をtrueと設定する場合、コマンドラインで次のように指定します。

```
java -Doracle.jdbc.defaultNChar=true myApplication
```

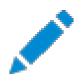
必要に応じて、アプリケーションでdefaultNCharを接続プロパティとして指定してNCHAR、NVARCHAR2またはNCLOBデータにアクセスすることもできます。

```
Properties props = new Properties();
props.put(OracleConnection.CONNECTION_PROPERTY_DEFAULTNCHAR, "true");
// set URL, username, password, and so on.
...
Connection conn = DriverManager.getConnection(props);
```

defaultNCharの値がtrueである場合、各国語文字が必要でない列のために、setFormOfUse(<column_Index>, FORM_CHAR)をコールする必要があります。たとえば:

```
PreparedStatement pstmt =
conn.prepareStatement("insert into TEST values(?,?,?)");
pstmt.setFormOfUse(3, OraclePreparedStatement.FORM_CHAR);
pstmt.setString(3, myString); // CHAR column
```

ノート:



defaultNCharの値をtrueに設定してからCHAR列にアクセスした場合、データベースでは、すべてのCHARデータを暗黙的にNCHARに変換します。この変換は、パフォーマンスに多大な影響を与えます。

ノート:

- 文字データには、必ずoracle.sql.CHARでなくjava.lang.Stringを使用します。CHARは下位互換性のみのために用意されています。
- setObjectメソッドを使用して各国語文字セットにアクセスすることもできますが、setObjectメソッドを使用する場合、ターゲット・データ型をTypes.NCHAR、Types.NCLOB、Types.NVARCHARまたはTypes.LONGNVARCHARとして設定する必要があります。

ノート:

Oracle Databaseでは、SQL文字列はデータベース文字セットに変換されます。そのため、次のことを覚えておく必要があります。

- Oracle Database 10gリリース1(10.1)以前のリリースでは、JDBCドライバは、データベース文字セットで表示できないUnicode文字を含むNCHARリテラル(n'...')をサポートしていません。データベース文字セットで表示できないUnicode文字はすべて破損します。
- Oracle Database 10gリリース2(10.2)のJDBCドライバがOracle Database 10gリリース2(10.2)のデータベース・サーバーに接続されている場合、NCHARリテラル(n'...')はすべてUnicodeリテラル(u'...')に変換され、ASCII以外の文字はすべて対応するUnicodeエスケープ・シーケンスに変換されます。この変換は、データ破損を防ぐために自動的に実行されます。
- Oracle Database 10gリリース2(10.2)のJDBCドライバがOracle Database 10gリリース1(10.1)以前のデータベース・サーバーに接続されている場合、NCHARリテラル(n'...')は変換されないため、データベース文字セットで表示できない文字はすべて破損します。

19.3 JDK 6での各国語文字セット用の新しいメソッド

JDBC 4.0では、各国語文字セット型にアクセスするために、次の4つの追加SQL型のサポートが導入されています。

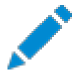
- NCHAR
- NVARCHAR
- LONGNVARCHAR
- NCLOB

これらの型は、各国語文字セットを使用して値がエンコードされる点を除き、CHAR、VARCHAR、LONGVARCHARおよびCLOB型に類似しています。JDBCの仕様では、Stringクラスを使用してNCHAR、NVARCHARおよびLONGNVARCHARデータを表し、NClobクラスを使用してNCLOB値を表します。

各国語の文字値を取得するには、アプリケーションから次のいずれかのメソッドをコールします。

- getNString
- getNClob
- getNCharacterStream

- getObject

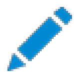


ノート:

NClob が Clob を実装しているため、getClob メソッドを使用して、NClob オブジェクトを戻します。

各国語キャラクタ・タイプのパラメータ・マーカーの値を指定するには、アプリケーションから次のいずれかのメソッドをコールします。

- setNString
- setNCharacterStream
- setNClob
- setObject



ノート:

setFormOfUse メソッドを使用して、JDK 6 の各国語キャラクタ値を指定できます。しかし、このメソッドは今後のリリースで非推奨になるため、利用はお薦めしません。この項で説明したメソッドを使用することをお薦めします。

関連項目:

setObjectメソッドが使用される場合、ターゲット・データ型をTypes.NCHAR、Types.NCLOB、Types.NVARCHARまたはTypes.LONGNVARCHARとして設定する必要があります。

第V部　パフォーマンスとスケーラビリティ

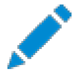
この部では、文キャッシュおよびOracle Call Interface(OCI)接続プーリングなど、Oracle Java Database Connectivity(JDBC)のパフォーマンス強化機能について説明します。また、バッチ更新や行のプリフェッチなどのOracleパフォーマンス拡張機能についても説明します。

第V部の章の内容は次のとおりです。

- 文キャッシュと結果セット・キャッシュ
- パフォーマンス拡張機能
- OCI接続プーリング
- データベース常駐接続プーリング
- Oracle Advanced Queuing
- 連続問合せ通知

20 文キャッシュと結果セット・キャッシュ

この章では、Oracle Java Database Connectivity(JDBC)拡張要素である文キャッシュの利点と使用方法について説明します。



ノート:

表の構造がデータベース内で同じ場合にのみ、文キャッシュを使用します。表の構造を変更し、変更前に作成および実行した文を再利用した場合、エラーが発生することがあります。

この章の構成は、次のとおりです。

- 文キャッシュについて
- 文キャッシュの使用について
- 文オブジェクトの再利用について
- 結果セット・キャッシュについて

20.1 文キャッシュについて

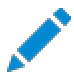
文キャッシュにより、繰り返しコールされるループやメソッドなどで何度も使用する実行文がキャッシュされるため、パフォーマンスが向上します。JDBC 3.0以降では、JDBC標準で文キャッシュ・インタフェースが定義されています。

文キャッシュでは、次のことができます。

- カーソル作成の繰返しによるオーバーヘッドを回避します。
- 文の解析と作成の繰返しを回避します。
- クライアント内のデータ構造を再利用します。

この項の内容は次のとおりです。

- 文キャッシュの基本
- 暗黙的文キャッシュ
- 明示的文キャッシュ



ノート:

暗黙的文キャッシュの使用を強くお薦めします。Oracle JDBC ドライバは暗黙的文キャッシュが有効になっているという前提で設計されています。このため、文キャッシュを使用しないとパフォーマンスに悪影響を及ぼします。

20.1.1 文キャッシュの基本

アプリケーションでは、特定の物理接続に関連付けられている文をキャッシュするために文キャッシュを使用します。このキャッシュ

はOracleConnectionオブジェクトに関連付けられています。OracleConnectionには、文キャッシュを有効にするメソッドが含まれています。文キャッシュを有効にすると、closeメソッドをコールするときに文オブジェクトがキャッシュされます。

物理接続ごとに独自のキャッシュがあるため、複数の物理接続に対して文キャッシュを有効にすると複数のキャッシュが存在することになります。接続キャッシュで文キャッシュを有効にすると、基礎となる物理接続で有効な文キャッシュが論理接続で利用されます。接続キャッシュによって保持されている論理接続で文キャッシュを有効にしようとすると、例外が発生します。

文キャッシュには、暗黙的文キャッシュと明示的文キャッシュの2つのタイプがあります。文キャッシュの各タイプは、互いに関係なく有効または無効にできます。いずれかを有効にするか、いずれも有効にしないか、または両方とも有効にすることができます。両方のタイプの文キャッシュで、接続ごとに1つのキャッシュが共有されます。

20.1.2 暗黙的文キャッシュ

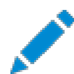
暗黙的文キャッシュを有効にすると、JDBCでは、プリペアド文またはコール可能文の文オブジェクトのcloseメソッドをコールしたときに、その文が自動的にキャッシュされます。プリペアド文およびコール可能文のキャッシュおよび取出しには、標準の接続オブジェクトおよび文オブジェクト・メソッドを使用します。

暗黙的文キャッシュはSQL文字列をキーとして使用しますが、プレーン文はSQL文字列を使用せずに作成されるため、プレーン文は暗黙的にキャッシュされません。このため、暗黙的文キャッシュは、SQL文字列を使用して作成されるOraclePreparedStatementおよびOracleCallableStatementオブジェクトにのみ適用されます。OracleStatementとともに暗黙的文キャッシュを使用することはできません。OraclePreparedStatementまたはOracleCallableStatementを作成するとき、JDBCドライバはキャッシュを自動的に検索し、一致する文を探します。一致基準は、次のとおりです。

- 文内のSQL文字列は、キャッシュのSQL文字列と同一である必要があります。
- 文の種類は、同じにする必要があります。つまり、プリペアド文またはコール可能文にします。
- 文によって生成される結果セットのスクロール可能な型は、同じにする必要があります。つまり、forward-onlyまたはscrollableにします。

キャッシュ検索中に一致するものが見つかった場合は、キャッシュされた文が戻されます。一致するものが見つからなかった場合は、新しい文が作成されて戻されます。どちらの場合でも、文は、そのカーソルおよび状態とともに、文オブジェクトのcloseメソッドをコールするとキャッシュされます。

キャッシュされたOraclePreparedStatementまたはOracleCallableStatementオブジェクトが取り出されると、状態およびデータ情報は自動的に再初期化されて、デフォルト値にリセットされますが、メタデータは保存されます。最大サイズに準拠するため、最低使用頻度(LRU)アルゴリズムを使用してキャッシュから文が削除されます。



ノート:

JDBC ドライバは、メタデータをクリアしません。メタデータはパフォーマンスの理由から保存されますが、セマンティクスへの影響はありません。暗黙的キャッシュによる文は、新たに作成された場合と同様に動作します。

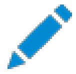
特定の文を暗黙的なキャッシュの対象から外すことができます。

関連トピック

- 暗黙的文キャッシュの使用方法について

20.1.3 明示的文キャッシュ

明示的文キャッシュによって、プリペアド文およびコール可能文を選択してキャッシュし、取り出すことができます。明示的文キャッシュは、ユーザーが指定する任意のJava Stringであるキーに依存します。



ノート:

プレーン文はキャッシュできません。

明示的文キャッシュは文のデータ、状態およびメタデータを保持するため、メタデータのみを保持する暗黙的文キャッシュに対してパフォーマンス上は有利です。ただし、このタイプのキャッシュを使用する場合は慎重である必要があります。明示的文キャッシュは再利用のために3つの種類の情報をすべて保存するため、前回の文の使用から、どのようなデータと状態が保持されているかわからない場合があります。

暗黙的文キャッシュと明示的文キャッシュは次の点が異なります。

- 文の取出し

  暗黙的文キャッシュの場合は、文をキャッシュから取り出すための特別な処理は不要です。かわりに、prepareStatementまたはprepareCallをコールするたびに、JDBCでは一致する文についてキャッシュが自動的にチェックされ、見つかった場合はその文が戻されます。しかし、明示的文キャッシュの場合は、専用のOracle WithKeyメソッドを使用して、文オブジェクトをキャッシュし、取り出します。

- キーの指定

  暗黙的文キャッシュでは、プリペアド文またはコール可能文のSQL文字列がキーとして使用され、ユーザーは特にアクションを実行する必要はありません。これに対し明示的文キャッシュでは、キーとして使用するJava Stringを指定する必要があります。

- 文の戻り

  暗黙的文キャッシュの実行中に、JDBCドライバでキャッシュ内に一致する文を見つけられない場合は、新しい文が自動的に作成されます。しかし、明示的文キャッシュの実行中にJDBCドライバでキャッシュ内に一致する文を見つけられない場合は、NULL値が戻されます。

表20-1で、暗黙的文キャッシュと明示的文キャッシュで使用されるメソッドの違いを比較します。

表20-1 文キャッシュで使用されるメソッドの比較

キャッシュのタイプ	割当て	キャッシュへの挿入	キャッシュからの取出し
暗黙的	prepareStatement prepareCall	close	prepareStatement prepareCall
明示的	createStatement prepareStatement	closeWithKey	getStatementWithKey getCallWithKey

キャッシュのタイプ	割当て	キャッシュへの挿入	キャッシュからの取出し
	prepareCall		

20.2 文キャッシュの使用について

この項では、次の項目について説明します。

- 文キャッシュの有効化および無効化について
- キャッシュされた文のクローズについて
- 暗黙的文キャッシュの使用方法について
- 明示的文キャッシュの使用方法について

20.2.1 文キャッシュの有効化および無効化について

OracleConnection APIを使用する場合は、暗黙的および明示的な文キャッシュは、互いに関係なく有効または無効にできます。いずれかを有効にするか、いずれも有効にしないか、または両方とも有効にすることができます。

暗黙的文キャッシュの有効化

暗黙的文キャッシュを有効にする方法は2つあります。最初のメソッドは、プールされていない物理接続上で文キャッシュを有効にします。ここでは、setStatementCacheSize メソッドを使用して、すべての接続に対する文サイズを明示的に指定する必要があります。2番目のメソッドは、プールされている論理接続上で文キャッシュを有効にします。プール内のそれぞれの接続は、MaxStatementsLimitプロパティを設定することで指定できる最大サイズが同一である、独自の文キャッシュを持っています。

メソッド1

次のステップを実行します。

- 接続でOracleDataSource.setImplicitCachingEnabled(true)メソッドをコールして、OracleDataSourceプロパティimplicitCachingEnabledをtrueに設定します。たとえば:

```
OracleDataSource ods =  new OracleDataSource();
...
ods.setImplicitCachingEnabled(true);
...
```

- 物理接続でOracleConnection.setStatementCacheSizeメソッドをコールします。引数にはキャッシュ内の文の最大数を指定します。たとえば、次のコードでは、キャッシュ・サイズを10の文に指定します。

```
((OracleConnection)conn).setStatementCacheSize(10);
```

メソッド2

次のステップを実行します。

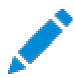
- OracleDataSourceプロパティのimplicitCachingEnabledおよびconnectionCachingEnabledをtrueに設定します。たとえば:

```
OracleDataSource ods =  new OracleDataSource();
...
ods.setConnectionCachingEnabled( true );
ods.setImplicitCachingEnabled( true );
...
```

- 接続キャッシュを使用する場合は、接続キャッシュでMaxStatementsLimitプロパティを正の整数に設定します。たとえば:

```
Properties cacheProps = new Properties();
...
cacheProps.put( "MaxStatementsLimit", "50" );
```

暗黙的キャッシュが有効になっているかどうかを判断するには、getImplicitCachingEnabledをコールします。暗黙的キャッシュが有効な場合はtrueが、それ以外の場合はfalseが戻されます。



ノート:

文キャッシュを有効にすると、暗黙的文キャッシュと明示的文キャッシュの両方が有効になります。

暗黙的文キャッシュの無効化

暗黙的文キャッシュを無効にするには、接続でsetImplicitCachingEnabled(false)をコールするか、または ImplicitCachingEnabledプロパティをfalseに設定します。

明示的文キャッシュの有効化

明示的文キャッシュを有効にするには、最初に文キャッシュ・サイズを設定する必要があります。キャッシュ・サイズを設定するために、物理接続でOracleConnection.setStatementCacheSizeをコールします。引数にはキャッシュ内の文の最大数を指定します。0(ゼロ)を指定すると、キャッシングは行われません。キャッシュ・サイズのチェックには、次のようにgetStatementCacheSizeメソッドを使用します。

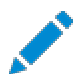
```
System.out.println("Stmt Cache size is " +
   ((OracleConnection)conn).getStatementCacheSize());
```

次のコードでは、キャッシュ・サイズを10の文に指定します。

```
((OracleConnection)conn).setStatementCacheSize(10);
```

接続でsetExplicitCachingEnabled(true)をコールすることにより、明示的文キャッシュを有効にします。

明示的キャッシュが有効になっているかどうかを判断するには、getExplicitCachingEnabledをコールします。明示的キャッシュが有効な場合はtrueが、それ以外の場合はfalseが戻されます。



ノート:

- 特定の物理接続に対しては、暗黙的キャッシュと明示的キャッシュを別々に有効にします。このため、同じセッション中で暗黙的文キャッシュと明示的文キャッシュの両方を実行できるわけです。

- 暗黙的文キャッシュと明示的文キャッシュは、同一のキャッシュを共有します。文キャッシュのサイズ設定では、この点を考慮してください。

明示的文キャッシュの無効化

setExplicitCachingEnabled(false)をコールすることにより、明示的文キャッシュを無効にします。キャッシュを無効化またはクローズすると、そのキャッシュは消去されます。次の例では、明示的文キャッシュを無効にします。

```
((OracleConnection)conn).setExplicitCachingEnabled(false);
```

20.2.2 キャッシュされた文のクローズについて

文をクローズし、それが確実にキャッシュに戻されないようにするには次の操作を行います。

J2SE 5.0の場合

- その文のキャッシュを無効にします。

  ```
  stmt.setDisableStmtCaching(true);
  ```

- 文オブジェクトのcloseメソッドをコールします。

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

JSE 6.0の場合

```
stmt.setPoolable(false);
stmt.close();
```

キャッシュされた文の物理的なクローズ

暗黙的文キャッシュを有効にすると、文の物理的なクローズを手動で実行できなくなります。文オブジェクト・キャッシュのcloseメソッドは、文をクローズするのではなく、キャッシュします。文は、次の3つの条件の1つで自動で物理的にクローズされます。

- 関連付けられている接続がクローズされた場合
- キャッシュがそのサイズ制限に達し、LRUアルゴリズムにより、最低使用頻度の文オブジェクトが優先的に取得される場合
- 文キャッシュが無効にされている文でcloseメソッドをコールした場合

20.2.3 暗黙的文キャッシュの使用方法について

暗黙的文キャッシュを有効にすると、デフォルトではすべてのプリペアド文およびコール可能文が自動的にキャッシュされます。暗黙的文キャッシュでは、次のステップが実行されます。

1. 暗黙的文キャッシュを有効にします。
2. 標準メソッドの1つを使用して文を割り当てます。
3. キャッシュの対象としない特定の文に対する暗黙的文キャッシュを無効にします。これはオプションのステップです。
4. closeメソッドを使用して文をキャッシュします。

5. 暗黙的にキャッシュされた文を取り出すには、適切な標準のprepareメソッドをコールします。

暗黙的キャッシュのための文の割当て

暗黙的文キャッシュに文を割り当てるには、通常prepareStatementまたはprepareCallメソッドを使用します。

次のコードでは、pstmtと呼ばれる新しい文オブジェクトを割り当てます。

```
PreparedStatement pstmt = conn.prepareStatement
   ("UPDATE emp SET ename = ? WHERE rowid = ?");
```

特定の文に対する暗黙的文キャッシュの無効化

ある接続の暗黙的文キャッシュを有効にすると、デフォルトでは、その接続のすべてのコール可能文およびプリペアド文が自動的にキャッシュされます。特定のコール可能文またはプリペアド文が暗黙的にキャッシュされないようにするには、文オブジェクトのsetDisableStmtCachingメソッドを使用します。キャッシュ領域を管理するためには、使用頻度の低い文でsetDisableStmtCachingメソッドをコールします。

次のコードでは、pstmtに対する暗黙的文キャッシュを無効にします。

```
PreparedStatement pstmt = conn.prepareStatement("SELECT 1 from DUAL");
((OraclePreparedStatement)pstmt).setDisableStmtCaching(true);
pstmt.close ();
```

ノート:

JSE 6 を使用している場合は、標準 JDBC 4.0 メソッド setPoolable を使用して文キャッシュを無効にします。

```
PreparedStatement.setPoolable(false);
```

次のコマンドを使用して Statement オブジェクトがプール可能かどうかを確認します。

```
Statement.isPoolable();
```

文の暗黙的なキャッシュ

割り当てられた文をキャッシュするには、文オブジェクトのcloseメソッドをコールします。OraclePreparedStatementまたはOracleCallableStatementオブジェクトでcloseメソッドをコールすると、この文のキャッシュを無効にしていないかぎり、JDBCドライバではこの文がキャッシュ内に自動的に挿入されます。

次のコードでは、pstmt文をキャッシュします。

```
pstmt.close ();
```

暗黙的にキャッシュされた文の取出し

暗黙的にキャッシュされた文を取り出すには、文の種類に応じて、prepareStatementまたはprepareCallメソッドをコールします。

次のコードでは、prepareStatementメソッドを使用して、キャッシュからpstmtを取り出します。

```
pstmt = conn.prepareStatement ("UPDATE emp SET ename = ? WHERE rowid = ?");
```

20.2.3.1 文の割当ておよび暗黙的文キャッシュで使用されるメソッド

文を割り当てたり、暗黙的にキャッシュされた文を取り出したりするメソッドについて、表20-2で説明します。

表20-2 文の割当ておよび暗黙的文キャッシュで使用されるメソッド

メソッド	暗黙的文キャッシュの機能
prepareStatement	目的のキャッシュされた OraclePreparedStatement オブジェクトを検索して戻すキャッシュ検索を実行します。一致するオブジェクトが見つからない場合は、新しい OraclePreparedStatement オブジェクトを割り当てます。
prepareCall	目的のキャッシュされた OracleCallableStatement オブジェクトを検索して戻すキャッシュ検索を実行します。一致するオブジェクトが見つからない場合は、新しい OracleCallableStatement オブジェクトを割り当てます。

例20-1は、暗黙的文キャッシュを有効にする方法を示すサンプル・コードです。

例20-1 暗黙的文キャッシュの使用

```
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Properties;
import javax.sql.DataSource;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.pool.OracleDataSource;
public class TestJdbc
{
  /**
   * Get a Connection, prepare a statement, execute a query, fetch the results, close the connection.
   * @param ods the DataSource used to get the connection.
   */
  private static void doSQL( DataSource ods ) throws SQLException
  {
    final String SQL = "select username from all_users";
    OracleConnection  conn = null;
    PreparedStatement ps = null;
    ResultSet rs = null;
    try
    {
      conn = (OracleConnection) ods.getConnection();
      System.out.println( "Connection:" + conn );
      System.out.println( "Connection getImplicitCachingEnabled:" +
conn.getImplicitCachingEnabled() );
      System.out.println( "Connection getStatementCacheSize:" + conn.getStatementCacheSize() );
      ps = conn.prepareStatement( SQL );
      System.out.println( "PreparedStatement:" + ps );
      rs = ps.executeQuery();
      while ( rs.next() )
```

```
      {
        String owner = rs.getString( 1 );
        System.out.println( owner );
      }
    }
    finally
    {
      if ( rs != null )
      {
        rs.close();
      }
      if ( ps != null )
      {
        ps.close();
      conn.close();
    }
  }
  }
  public static void main( String[] args )
  {
    try
    {
      OracleDataSource ods =  new OracleDataSource();
      ods.setDriverType( "thin" );
      ods.setServerName( "localhost" );
      ods.setPortNumber( 5221 );
      ods.setServiceName( "orcl" );
      ods.setUser( "HR" );
      ods.setPassword( "hr" );
      ods.setConnectionCachingEnabled( true );
      ods.setImplicitCachingEnabled( true );
      Properties cacheProps = new Properties();
      cacheProps.put( "InitialLimit", "1" );
      cacheProps.put( "MinLimit", "1" );
      cacheProps.put( "MaxLimit", "5" );
      cacheProps.put( "MaxStatementsLimit", "50" );
      ods.setConnectionCacheProperties( cacheProps );
      System.out.println( "DataSource getImplicitCachingEnabled: " +
ods.getImplicitCachingEnabled() );
      for ( int i = 0; i < 5; i++ )
      {
        doSQL( ods );
      }
    }
    catch ( Exception ex )
    {
      ex.printStackTrace();
    }
  }
}
```

20.2.4 明示的文キャッシュの使用方法について

明示的文キャッシュを有効にすると、プリペアド文またはコール可能文を明示的にキャッシュできます。明示的文キャッシュでは、次のステップが行われます。

1. 明示的文キャッシュを有効にします。
2. 標準メソッドの1つを使用して文を割り当てます。
3. 文を明示的にキャッシュするには、closeWithKeyメソッドを使用して、キーを付けてその文をクローズします。
4. 明示的にキャッシュされた文を取り出すには、適切なWithKeyメソッドをコールして、適切なキーを指定します。
5. オープンしている、明示的にキャッシュされた文を再キャッシュするには、closeWithKeyメソッドを使用して文を再度クローズします。キャッシュされた文をクローズするたびに、そのキーを使用して文を再キャッシュします。

明示的なキャッシュのための文の割当て

明示的文キャッシュに文を割り当てるには、通常createStatement、prepareStatementまたはprepareCallメソッドを使用します。

次のコードでは、pstmtと呼ばれる新しい文オブジェクトを割り当てます。

```
PreparedStatement pstmt =
   conn.prepareStatement ("UPDATE emp SET ename = ? WHERE rowid = ?");
```

文の明示的なキャッシュ

割り当てられた文を明示的にキャッシュするには、文オブジェクトのcloseWithKeyメソッドをコールして、キーを指定します。キーは、ユーザーが指定する任意のJava Stringです。closeWithKeyメソッドは、文をそのままキャッシュします。つまり、データ、状態およびメタデータはクリアされずに保持されます。

次のコードでは、"mykey"キーを使用してpstmt文をキャッシュします。

```
((OraclePreparedStatement)pstmt).closeWithKey ("mykey");
```

明示的にキャッシュされた文の取出し

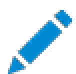
明示的にキャッシュされた文を再度コールするには、文の種類によって、getStatementWithKeyまたはgetCallWithKeyメソッドをコールします。

指定したキーを使用して文を取り出す場合は、指定したキーに基づいて、JDBCドライバがキャッシュ内でその文を検索します。一致する文が見つかった場合、一致する文は、状態、データおよびメタデータとともに戻されます。情報は、文が最後にクローズしたときの状態で戻されます。一致する文が見つからない場合は、JDBCドライバでnullが戻されます。

次のコードでは、getStatementWithKeyメソッドで"mykey"キーを使用して、キャッシュからpstmtを再度コールします。pstmt文オブジェクトは"mykey"キーでキャッシュされていました。

```
pstmt = ((OracleConnection)conn).getStatementWithKey ("mykey");
```

pstmt文オブジェクトでcreationStateメソッドをコールすると、メソッドはEXPLICITを戻します。



ノート:

明示的にキャッシュされた文を取り出す場合は、キーを指定するときに文の種類に適したメソッドを使用してください。たとえば、prepareStatement メソッドを使用して文を割り当てた場合は、getStatementWithKey メソッドを

使用してキャッシュから文を取り出します。JDBC ドライバは、戻す文の型を検証しません。

20.2.4.1 明示的にキャッシュされた文を取り出す場合に使用するメソッド

明示的にキャッシュされた文を取り出す場合に使用するメソッドについて、表20-3で説明します。

表20-3 明示的にキャッシュされた文を取り出す場合に使用するメソッド

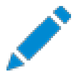
メソッド	明示的文キャッシュの機能
getStatementWithKey	キャッシュからプリペアド文を取り出すのに必要なキーを指定します。
getCallWithKey	キャッシュからコール可能文を取り出すのに必要なキーを指定します。

20.3 文オブジェクトの再利用について

JDBC 3.0の仕様で導入された文プーリングでは、アプリケーションは、Connectionオブジェクトの使用と同様にしてPreparedStatementオブジェクトを再利用することができます。PreparedStatementオブジェクトは、複数の論理接続が透過的に再利用できます。

この項の内容は次のとおりです。

- プールされた文の使用について
- プールされた文のクローズについて



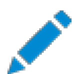
ノート:

Oracle JDBC ドライバでは、文のプーリングをサポートするために、暗黙的文キャッシュを使用します。

20.3.1 プールされた文の使用について

StatementインタフェースからisPoolableメソッドをコールすることによって、データソースが文プーリングをサポートするかどうかを認識できます。戻り値がtrueの場合、アプリケーションはPreparedStatementオブジェクトがプールされていることを認識していま す。アプリケーションはまた、StatementインタフェースからsetPoolableメソッドをコールすることによって、文がプールされるようにするかどうかを指定することができます。

プールされた文の再利用は、アプリケーションに対して完全に透過的である必要があります。つまり、PreparedStatementオブジェクトが文プーリングに関与するかどうかにかかわらず、アプリケーション・コードが同じままであることが必要です。アプリケーションがPreparedStatementオブジェクトをクローズした場合、再利用する際はConnection.prepareStatementメソッドをコールする必要があります。



ノート:

アプリケーションは、文がどのようにプールされるかを、直接的に制御しません。文のプールは PooledConnection オブジェクトに関連付けられます。このオブジェクトの動作は、それを作成した ConnectionPoolDataSource オブジェクトのプロパティによって決定されます。

20.3.2 プールされた文のクローズについて

アプリケーションがプールされた文をクローズする方法は、プールされていない文をクローズする方法と完全に同じです。プールされた文であってもプールされていない文であっても、一度クローズされると、その文をアプリケーションが使用することはできず、再利用しようとすると例外がスローされる原因になります。認識できる唯一の相違点は、プールされる物理文をアプリケーションが直接クローズすることができないということです。その操作はプール・マネージャが行います。メソッドPooledConnection.closeAll は、指定の物理接続上でオープンしている文をすべてクローズします。これにより、それらの文に関連付けられているリソースが解放されます。

次のメソッドは、プールされた文をクローズすることができます。

- close

  このjava.sql.Statementインタフェース・メソッドはアプリケーションによってコールされます。文がプールされる場合、アプリケーションが使用する論理文はクローズされますが、プールされた物理文はクローズされません。

- close

  このjava.sql.Connectionインタフェース・メソッドはアプリケーションによってコールされます。このメソッドは、文を使用する接続がプールされるかどうかに応じて、異なる動作をします。

  - プールされていない接続

    このメソッドは、物理接続が作成したすべての接続とすべての文をクローズします。こうする必要があるのは、外部的に管理されているリソースをいつ解放できるかは、ガーベジ収集メカニズムで検出できないためです。

  - プールされた接続

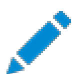
    このメソッドは論理接続とそれが戻した論理文をクローズしますが、その基礎となるPooledConnectionオブジェクトや、関連付けられているプールされた文はオープンされた状態に残します。

- PooledConnection.closeAll

  このメソッドは、PooledConnectionオブジェクトによってプールされる物理文をすべてクローズするために接続プール・マネージャによってコールされます。

20.4 結果セット・キャッシュについて

アプリケーションでは、データベースに問合せを繰返し送信することがあります。繰返しの問合せのレスポンス時間を改善するには、問合せの結果、問合せフラグメントおよびPL/SQLファンクションをメモリーにキャッシュできます。結果キャッシュには、すべてのセッションにわたって共有される問合せの結果が格納されます。これらの問合せを繰返し実行すると、結果がキャッシュ・メモリーから直接取り出されます。



ノート:

結果セットが非常に大きい場合、サイズ制限のためにキャッシュされない場合があります。

結果が問合せ結果キャッシュに格納されることを示すには、問合せまたは問合せフラグメントに結果キャッシュのヒントで注釈を付ける必要があります。

問合せ結果セットは、次の方法でキャッシュできます。

- サーバー側結果セット・キャッシュ
- クライアント側結果セット・キャッシュ

ノート:

- サーバー側およびクライアント結果セットのキャッシュは、読取り専用または大部分が読取りのデータに最も便利です。非常に動的な結果となる問合せの場合は、パフォーマンスが低下する可能性があります。
- サーバー側およびクライアント結果セットのキャッシュはどちらも、メモリーを使用します。したがって、非常に大きな結果セットをキャッシュすると、パフォーマンス上の問題が発生する可能性があります。

20.4.1 サーバー側結果セット・キャッシュ

サーバー側結果セット・キャッシュは、Oracle Database 11gリリース1から、JDBC ThinおよびJDBC Oracle Call Interface(OCI)の両方のドライバに対してサポートされるようになりました。サーバー側結果キャッシュは、現在の問合せ、問合せフラグメントおよびPL/SQLファンクションの結果をメモリーにキャッシュし、その問合せ、問合せフラグメントまたはPL/SQLファンクションをその後実行する際に、キャッシュした結果を使用するためのものです。キャッシュした結果はSGAの結果キャッシュ・メモリー部分に存在します。作成に使用されたデータベース・オブジェクトが正常に修正されると、キャッシュ結果は自動的に無効化されます。サーバー側キャッシュには、次の2つのタイプがあります。

- SQL問合せ結果キャッシュ
- PL/SQLファンクション結果キャッシュ

関連項目:

- SQL問合せ結果キャッシュの詳細は、『Oracle Databaseパフォーマンス・チューニング・ガイド』を参照してください。
- PL/SQLファンクション結果キャッシュの詳細は、『Oracle Database PL/SQL言語リファレンス』を参照してください。

20.4.2 クライアント側結果セット・キャッシュ

クライアント側結果セット・キャッシュ機能により、クライアント側でのSQL問合せ結果セットのクライアント・メモリーへのキャッシュが可能になります。このようにして、アプリケーションでは、クライアント・メモリーを使用することで、繰返しの問合せのレスポンス時間を改善するためにクライアント側結果セット・キャッシュを利用できます。

この項の内容は次のとおりです。

- クライアント側結果セット・キャッシュの有効化

- クライアント側結果セット・キャッシュの利点
- JDBCでの使用ガイドライン

20.4.2.1 クライアント側結果セット・キャッシュの有効化

Oracle Databaseリリース18cでは、JDBC Thinドライバでクライアント側結果セット・キャッシュがサポートされています。この機能を有効にするには、新しいoracle.jdbc.enableQueryResultCache接続プロパティを使用できます。このプロパティのデフォルト値はtrueで、この機能がデフォルトで有効になっていることを意味します。プロパティをfalseに設定することで、この機能を無効にできます。

ノート:

- Oracle Database 12c リリース 2 (12.2)では、enableQueryResultCache プロパティは enableResultSetCache として使用可能であり、デフォルト値は false です。enableResultSetCache プロパティを true に設定することで、この機能を有効にできます。
- JDBC OCI ドライバは、クライアント側結果セット・キャッシュをすでにサポートしています。

関連項目:

『Oracle Call Interfaceプログラマーズ・ガイド』

この機能を使用するには、次のデータベース初期化パラメータを次のように設定する必要があります。

```
CLIENT_RESULT_CACHE_SIZE=100M
CLIENT_RESULT_CACHE_LAG=1000
```

CLIENT_RESULT_CACHE_SIZEパラメータのこの値は、Thinドライバがキャッシュに使用できるメモリー量を制御します。

読取り専用または大部分が読取りの表に注釈を付けることができるようになり、表のデータをドライバにキャッシュできるようになります。たとえば、RESULT_CACHE(MODE FORCE)です。

キャッシュの対象となる問合せを識別するSQLヒント/*+RESULT_CACHE */も使用できます。

関連項目:

Oracle Database JDBC Java APIリファレンス

20.4.2.2 クライアント側結果セット・キャッシュの利点

クライアント側結果セット・キャッシュには次のような利点があります。

- クライアント側結果セット・キャッシュは、アプリケーションに対して完全に透過的です。結果セット・データのキャッシュは、結果セットに影響を与えるすべてのセッションまたはデータベース変更と一致するように維持されています。
- 表注釈を使用すると、クライアント側の結果セットがJDBCアプリケーションに対して透過的に動作します。そうでない場

合、ヒントを使用してこれを有効にします。キャッシュ・ヒットはサーバーへの問合せとラウンドトリップの実行を回避して、結果セットを取得します。このため、サーバーCPUやサーバーI/Oなどのサーバー・リソースにとって、非常に大きなパフォーマンスの節約になります。

関連項目:

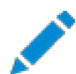
表注釈およびSQLヒント

- クライアントの結果キャッシュはプロセス単位であるため、複数のクライアント・セッションで一致するキャッシュ内の結果セットを同時に使用できます。
- クライアントの結果キャッシュにより、各アプリケーションが独自のカスタム結果セット・キャッシュを持つ必要性が最小限に抑えられます。
- クライアントの結果キャッシュではクライアント・メモリーが使用され、サーバー・メモリーよりコストが低くなります。

20.4.2.3 JDBCでの使用ガイドライン

結果セット・キャッシュを有効化するには次の3つの方法があります。

- RESULT_CACHE_MODEパラメータ
- 表注釈
- SQLヒント



ノート:

- クライアント側結果セット・キャッシュを使用する場合、アプリケーション・レベルで JDBC 文キャッシュまたはキャッシュ文を使用する必要があります。
- SQL ヒントは、セッション・パラメータ RESULT_CACHE_MODE および表注釈よりも優先されます。表注釈 FORCE はセッション・パラメータよりも優先されます。

関連トピック

- 文キャッシュと結果セット・キャッシュ

20.4.2.3.1 RESULT_CACHE_MODEパラメータ

RESULT_CACHE_MODEパラメータを使用して、問合せで使用する表の結果キャッシュ・モードを決定できます。ALTER SESSIONおよびALTER SYSTEM文でこの句を使用するか、またはサーバー・パラメータ・ファイル(init.ora)内でこの句を使用して、結果キャッシュを決定します。RESULT_CACHE_MODEパラメータを設定して、SQL問合せ結果キャッシュをすべての問合せで使用するか、SQLヒントまたは表注釈を使用して結果キャッシュ・ヒントで注釈を付けられている問合せのみで使用するかを制御できます。

20.4.2.3.2 表注釈

表注釈を使用して、コードを変更しないで結果キャッシュを有効にできます。ALTER TABLEおよびCREATE TABLE文を使用すると、結果キャッシュ・モードで表に注釈を付けることができます。構文は次のとおりです。

```
CREATE|ALTER TABLE [<schema>.]<table> ... [RESULT_CACHE (MODE {FORCE|DEFAULT})]
```

次の例では、CREATE TABLE文で表注釈を使用する方法を示しています。

```
CREATE TABLE foo (a NUMBER, b VARCHAR2(20)) RESULT_CACHE (MODE FORCE);
```

次の例では、ALTER TABLE文で表注釈を使用する方法を示しています。

```
ALTER TABLE foo RESULT_CACHE (MODE DEFAULT);
```

20.4.2.3.3 SQLヒント

SQLヒントを使用し、/*+ result_cache */または/*+ no_result_cache */のヒントの付いている問合せに注釈を付けることによってキャッシュする問合せを指定できます。たとえば、次のコードを見てください。

```
String  query  = "select /*+ result_cache */ * from employees where employee_id < : 1";
   ((oracle.jdbc.OracleConnection)conn).setImplicitCachingEnabled(true);
   ((oracle.jdbc.OracleConnection)conn).setStatementCacheSize(10);
   PreparedStatement  pstmt;
   ResultSet rs;

    for (int j = 0 ; j < 10 ; j++ )
    {
       pstmt  = conn.prepareStatement (query);
       pstmt.setInt(1,7500);
       rs  = pstmt.executeQuery();
         while (rs.next( ) )
        {     // see the values  }
          rs.close;
          pstmt.close( ) ;
        }
    }
```

この例では、クライアント結果キャッシュのヒント/*+ result_cache */が、実際の問合せであるselect * from employees where employee_id < : 1に注釈として付けられています。したがって、問合せは最初にデータベースに対して実行され、その結果セットが、問合せの残り9回の実行用にキャッシュされます。これにより、アプリケーションのパフォーマンスは大幅に向上します。これは主として読取り専用データの場合に便利です。

SQLヒントの例を次に示します。次の例はすべて、dept表が結果キャッシュのために次のコマンドによって注釈を付けられていることを前提にしています。

```
ALTER TABLE dept result_cache (MODE FORCE);
```

例

- SELECT * FROM employees

  結果セットはキャッシュされません。

- SELECT * FROM departments

  結果セットはキャッシュされます。

- SELECT /*+ result_cache */ employee_id FROM employees

  結果セットはキャッシュされます。

- SELECT /*+ no_result_cache */ department_id FROM departments

  結果セットはキャッシュされません。

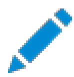
- SELECT /*+ result_cache */ * FROM departments

  問合せヒントは不要ですが、結果セットはキャッシュされます。

- SELECT e.first_name FROM employees e, departments d WHERE e.department_id = d.department_id

  問合せヒントが使用不可で、すべての表がFORCEとして注釈を付けられていないため、結果セットはキャッシュされません。

ノート:



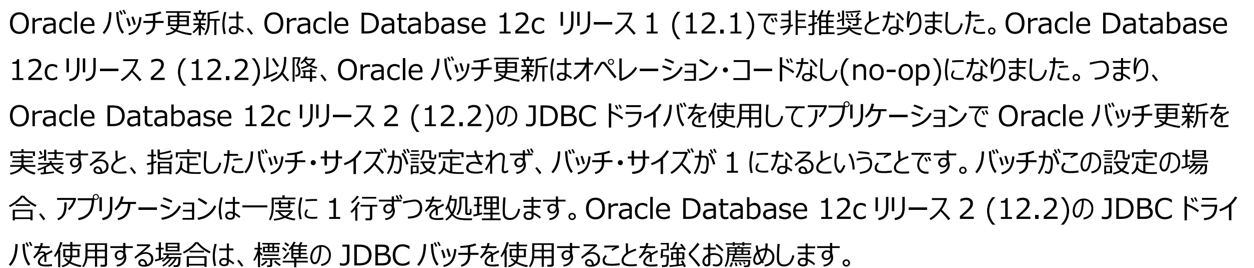
使用ガイドライン、クライアント・キャッシュの整合性、デプロイ時間の設定、クライアント・キャッシュ統計情報、クライアント結果キャッシュの検証および OCI クライアント結果キャッシュとサーバー結果キャッシュの詳細は、『Oracle Call Interface プログラマーズ・ガイド』を参照してください。

21 パフォーマンス拡張機能

この章では、Java Database Connectivity(JDBC)標準に対するOracleパフォーマンス拡張機能について説明します。

この章の構成は、次のとおりです。

- バッチ更新機能
- その他のOracleパフォーマンス拡張機能



ノート:

Oracle バッチ更新は、Oracle Database 12c リリース 1 (12.1)で非推奨となりました。Oracle Database 12c リリース 2 (12.2)以降、Oracle バッチ更新はオペレーション・コードなし(no-op)になりました。つまり、Oracle Database 12c リリース 2 (12.2)の JDBC ドライバを使用してアプリケーションで Oracle バッチ更新を実装すると、指定したバッチ・サイズが設定されず、バッチ・サイズが 1 になるということです。バッチがこの設定の場合、アプリケーションは一度に 1 行ずつを処理します。Oracle Database 12c リリース 2 (12.2)の JDBC ドライバを使用する場合は、標準の JDBC バッチを使用することを強くお薦めします。

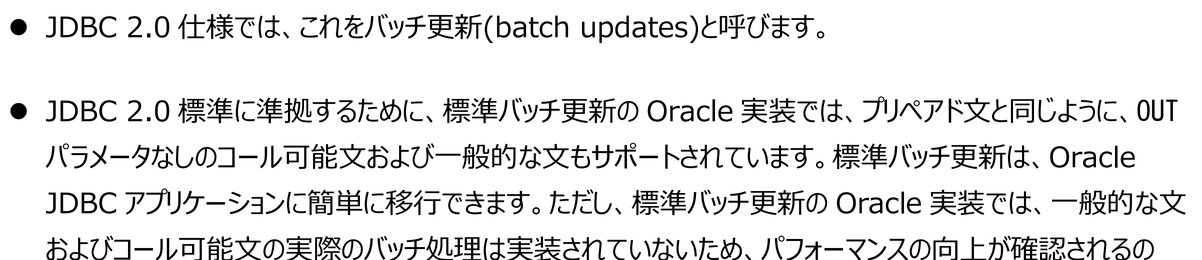
21.1 バッチ更新

この項の内容は次のとおりです。

- バッチ更新の概要
- 標準バッチ更新
- 早期バッチ・フラッシュ

21.1.1 バッチ更新の概要

複数のUPDATE文、DELETE文またはINSERT文を単一のバッチにグループ化して、バッチ全体を一度にデータベースに送信して処理することによって、データベースへのラウンドトリップの回数を減らし、それによってアプリケーションのパフォーマンスを向上させることができます。これは、バッチ更新(update batching)と呼ばれます。これは、特にプリペアド文で、同じ文をバインド変数を変えて繰り返し使用する場合に効果的です。



ノート:

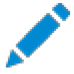
- JDBC 2.0 仕様では、これをバッチ更新(batch updates)と呼びます。
- JDBC 2.0 標準に準拠するために、標準バッチ更新の Oracle 実装では、プリペアド文と同じように、OUT パラメータなしのコール可能文および一般的な文もサポートされています。標準バッチ更新は、Oracle JDBC アプリケーションに簡単に移行できます。ただし、標準バッチ更新の Oracle 実装では、一般的な文およびコール可能文の実際のバッチ処理は実装されていないため、パフォーマンスの向上が確認されるの

は、PreparedStatement オブジェクトの場合のみです。

21.1.2 標準バッチ更新

JDBC標準バッチ更新は、addBatchメソッドを使用して明示的に文をバッチに追加し、executeBatchメソッドを使用して明示的にバッチを処理します。

ノート:



バッチ更新を使用する場合、自動コミット・モードは無効化します。バッチの処理中にエラーが発生した場合、エラーの前に正常に実行された操作をコミットするかロールバックするかを選択できます。

21.1.2.1 標準バッチ処理のOracle実装の制限事項

この項では、標準バッチ更新のOracle実装に関する、制限事項と実装の詳細を説明します。

Oracle JDBCアプリケーションでバッチ更新を使用すると、バインド変数の設定を変えてプリペアド文を繰り返し処理できます。

標準バッチ更新のOracle実装では、一般的な文およびコール可能文の実際のバッチ処理は実装されていません。Oracle JDBCはStatementおよびCallableStatementオブジェクトに対する標準バッチ処理の使用をサポートしますが、パフォーマンスは向上しません。

21.1.2.2 バッチに対する操作の追加について

いかなる文オブジェクトも、最初に作成されたとき、その文バッチは空です。文バッチに操作を追加するには、標準のaddBatchメソッドを使用します。このメソッドは標準のjava.sql.Statementインタフェース、PreparedStatementインタフェースおよびCallableStatementインタフェースで指定され、それぞれ、oracle.jdbc.OracleStatementインタフェース、OraclePreparedStatementインタフェースおよびOracleCallableStatementインタフェースで実装されます。

Statementオブジェクトの場合、addBatchメソッドは、入力としてSQL操作のJava Stringを取ります。たとえば:

```
...
Statement stmt = conn.createStatement();
stmt.addBatch("INSERT INTO emp VALUES(1000, 'Joe Jones')");
stmt.addBatch("INSERT INTO dept VALUES(260, 'Sales')");
stmt.addBatch("INSERT INTO emp_dept VALUES(1000, 260)");
...
```

この時点で、バッチには3つの操作が入っています。

プリペアド文の場合、バッチ更新は、異なる組合せのバインド・パラメータを使用して、同じ文を複数回実行するために使用されます。PreparedStatementオブジェクトまたはOraclePreparedStatementオブジェクトの場合、addBatchメソッドは、入力を取りません。適切なsetXXXメソッドによって最後に設定されたバインド・パラメータを使用して、操作をバッチに追加するだけです。CallableStatementオブジェクトやOracleCallableStatementオブジェクトの場合も同様です。ただし、標準更新バッチのOracle実装では、コール可能文をバッチ処理してもパフォーマンスはおそらく向上しないことに注意してください。

たとえば:

```
...
PreparedStatement pstmt =
          conn.prepareStatement("INSERT INTO employees VALUES(?, ?)");
pstmt.setInt(1, 2000);
pstmt.setString(2, "Milo Mumford");
pstmt.addBatch();
pstmt.setInt(1, 3000);
pstmt.setString(2, "Sulu Simpson");
pstmt.addBatch();
...
```

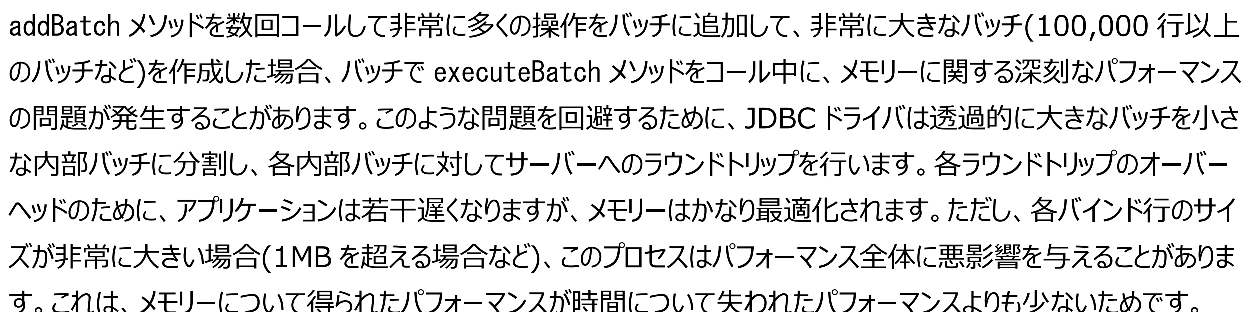
この時点で、バッチには2つの操作が入っています。

1つのバッチは単一のプリペアド文オブジェクトに関連付けられるので、バッチ処理できるのは、この例のような単一のプリペアド文の繰返し実行のみです。

21.1.2.3 バッチの処理について

現在の操作バッチを処理するには、文オブジェクトのexecuteBatchメソッドを使用します。このメソッドは標準のStatementインタフェースで指定されます。これは標準PreparedStatementインタフェースとCallableStatementインタフェースで拡張されています。

ノート:


addBatch メソッドを数回コールして非常に多くの操作をバッチに追加して、非常に大きなバッチ(100,000 行以上のバッチなど)を作成した場合、バッチで executeBatch メソッドをコール中に、メモリーに関する深刻なパフォーマンスの問題が発生することがあります。このような問題を回避するために、JDBC ドライバは透過的に大きなバッチを小さな内部バッチに分割し、各内部バッチに対してサーバーへのラウンドトリップを行います。各ラウンドトリップのオーバーヘッドのために、アプリケーションは若干遅くなりますが、メモリーはかなり最適化されます。ただし、各バインド行のサイズが非常に大きい場合(1MB を超える場合など)、このプロセスはパフォーマンス全体に悪影響を与えることがあります。これは、メモリーについて得られたパフォーマンスが時間について失われたパフォーマンスよりも少ないためです。

次の例では、前の例で示したプリペアド文のaddBatchコールを繰り返した後、バッチを処理します。

```
...
PreparedStatement pstmt =
          conn.prepareStatement("INSERT INTO employees VALUES(?, ?)");
pstmt.setInt(1, 2000);
pstmt.setString(2, "Milo Mumford");
pstmt.addBatch();
pstmt.setInt(1, 3000);
pstmt.setString(2, "Sulu Simpson");
pstmt.addBatch();
int[] updateCounts = pstmt.executeBatch();
...
```

21.1.2.4 配列DMLの反復ごとの行数

Oracle Database 12cリリース1 (12.1)以降、executeBatchメソッドは改良され、バッチのレコード数と同じサイズのint配列を戻します。戻り配列の各項目は、バッチの対応するレコードの影響を受けたデータベース表の行数です。たとえば、バッチ・

サイズが5の場合、executeBatchメソッドはサイズ5の配列を戻します。バッチの実行中にエラーが発生した場合、executeBatchメソッドは値を戻すことができず、かわりにBatchUpdateExceptionをスローします。この場合、例外自体がサイズnのint配列をそのデータとして保持します。nは、成功したレコード実行の数です。たとえば、バッチのサイズが5で、4番目のレコードでエラーが発生した場合、BatchUpdateExceptionはサイズ3の配列(3レコードは正常に実行)を保持し、配列内の各項目は各レコードの影響を受けた行数を表します。

21.1.2.5 標準バッチ処理のOracle実装による変更のコミットについて

バッチを処理した後、変更をコミットする必要があります。自動コミットは使用禁止(推奨)にしてあるものとします。

commitをコールすると、処理済の文バッチの場合、バッチ処理された操作とされなかった操作がコミットされますが、標準バッチのOracle実装では、未処理の保留文バッチには影響がありません。

21.1.2.6 バッチのクリアについて

現在の操作バッチを処理しないで消去するには、文オブジェクトのclearBatchメソッドを使用します。このメソッドは標準のStatementインタフェースで指定されます。これは標準PreparedStatementインタフェースとCallableStatementインタフェースで拡張されています。

次の点に注意してください。

- バッチが処理されるとき、操作は、バッチに入れられた順に実行されます。
- addBatchをコールした後、executeUpdateをコールする前に、executeBatchまたはclearBatchをコールする必要があります。コールしないと、SQL例外が発生します。
- clearBatchまたはexecuteBatchをコールすると、文バッチは空にリセットされます。
- 接続がROLLBACK要求を受信すると、文のバッチが空にリセットされません。リセットするには、clearBatchを明示的にコールする必要があります。
- ロールバック後のclearBatchメソッドのコールは、すべてのリリースで有効です。
- 文オブジェクトの現行結果セットがある場合、この結果セットはexecuteBatchコールによってクローズされます。
- clearBatchメソッドの戻り値はありません。

次の例では、前の例で示したプリペアド文のaddBatchコールを繰り返した後、特定の条件でバッチをクリアします。

```
...
PreparedStatement pstmt =
          conn.prepareStatement("INSERT INTO employees VALUES(?, ?)");
pstmt.setInt(1, 2000);
pstmt.setString(2, "Milo Mumford");
pstmt.addBatch();
pstmt.setInt(1, 3000);
pstmt.setString(2, "Sulu Simpson");
pstmt.addBatch();
if (...condition...)
{
   int[] updateCounts = pstmt.executeBatch();
   ...
}
else
```

```
{
   pstmt.clearBatch();
   ...
}
```

21.1.2.7 標準バッチ処理のOracle実装の更新件数

文バッチが正常に処理された場合、文のexecuteBatchコールから戻される整数配列、つまり、更新件数配列には、常にバッチ操作1つに対して1つの要素が含まれます。標準バッチ更新のOracle実装では、配列要素の値は次のようになります。

- プリコンパイル文のバッチの場合、配列には、各操作で影響を受けた行数を示す実際の更新件数を格納します。
- 一般的な文のバッチの場合、配列には、各操作で影響を受けた行数を示す実際の更新件数を格納します。標準バッチのOracle実装では、一般的な文の場合のみ、実際の更新件数がわかります。
- コール可能文のバッチの場合、配列には、各操作で影響を受けた行数を示す実際の更新件数を格納します。

コードの側では、バッチの正常な処理に対して、配列要素に-2、1または実際の更新件数のいずれが設定されても処理できるように準備しておく必要があります。正常なバッチ処理では、配列にはすべて-2が含まれるか、すべて1が含まれるか、またはすべて正の整数が含まれます。

例21-1は、標準バッチ更新の使用方法を示しています。

例21-1 標準バッチ更新

この例は、前の項のサンプル・コードを組み合せたもので、次のステップを行います。

1. 自動コミット・モードの無効化。どちらかのバッチ更新モデルを使用する場合、無効にする必要があります。
2. プリペアド文オブジェクトの作成。
3. プリペアド文オブジェクトに関連付けられたバッチへの操作の追加。
4. バッチの処理
5. バッチの操作のコミット。

```
conn.setAutoCommit(false);
PreparedStatement pstmt =
          conn.prepareStatement("INSERT INTO employees VALUES(?, ?)");
pstmt.setInt(1, 2000);
pstmt.setString(2, "Milo Mumford");
pstmt.addBatch();
pstmt.setInt(1, 3000);
pstmt.setString(2, "Sulu Simpson");
pstmt.addBatch();
int[] updateCounts = pstmt.executeBatch();
conn.commit();
pstmt.close();
...
```

更新件数配列を処理して、バッチが正常に処理されたかどうかを判断できます。

21.1.2.8 標準バッチ処理のOracle実装におけるエラー処理

executeBatchがコールされたとき、バッチ処理された操作のうち1つでも失敗したり、結果セットを戻そうとしたりすると、処理は

停止し、java.sql.BatchUpdateExceptionが生成されます。

バッチ例外の後、BatchUpdateExceptionオブジェクトのgetUpdateCountsメソッドを使用して、更新件数配列を取り出せます。このメソッドからは、executeBatchメソッドと同じように、更新件数のint配列が戻されます。標準バッチ更新のOracle実装では、バッチが処理された後の更新件数配列の内容は次のようになります。

- プリペアド文バッチにおいて、バッチの実行中にエラーが発生した場合、executeBatchメソッドは値を戻すことができず、かわりにBatchUpdateExceptionをスローします。この場合、例外自体がサイズnのint配列をそのデータとして保持します。nは、成功したレコード実行の数です。たとえば、バッチのサイズが5で、4番目のレコードでエラーが発生した場合、BatchUpdateExceptionはサイズ3の配列(3レコードは正常に実行)を保持し、配列内の各項目は各レコードの影響を受けた行数を表します。
- 一般的な文のバッチまたはコール可能文のバッチの場合、更新件数配列は、エラーの時点までの実際の更新件数を格納する、部分的な配列になります。標準バッチ更新のOracle実装では、Oracle JDBCは一般の文およびコール可能文の本当の意味でのバッチ処理を使用できないので、実際の更新件数がわかります。

  たとえば、バッチに20の操作が含まれているとき、最初の13は正常終了し、14番目で例外が生成された場合、更新件数配列には13の要素が含まれ、正常終了した操作の実際の更新件数が設定されます。

  この場合、正常終了した操作をコミットすることも、ロールバックすることもできます。

コードの側では、例外が発生した場合、バッチの失敗した処理に対して、配列要素に-3または実際の更新件数のどちらが設定されても処理できるように準備しておく必要があります。失敗したバッチ処理では、すべてに-3が含まれる完全な配列か、正の整数が含まれる部分配列が作成されます。

21.1.2.9 バッチ処理される文とバッチ処理されない文の混在について

文オブジェクトに操作の保留バッチがある場合、通常のバッチ処理されない操作の処理を行うためにexecuteUpdateをコールできません。

ただし、文バッチに操作を追加する前か、バッチを処理した後で、バッチ処理されない操作を処理する場合は、バッチ処理される操作とバッチ処理されない操作を単一文オブジェクトに混在させることができます。つまり、文オブジェクトのexecuteUpdateは、バッチ更新が空のときにのみコールできます。バッチが空でない場合、例外が生成されます。

たとえば、次のような順序は有効です。

```
...
PreparedStatement pstmt =
          conn.prepareStatement("INSERT INTO employees VALUES(?, ?)");
pstmt.setInt(1, 2000);
pstmt.setString(2, "Milo Mumford");
int scount = pstmt.executeUpdate();   // OK; no operations in pstmt batch
pstmt.setInt(1, 3000);
pstmt.setString(2, "Sulu Simpson");
pstmt.addBatch();                    // Now start a batch
pstmt.setInt(1, 4000);
pstmt.setString(2, "Stan Leland");
pstmt.addBatch();
int[] bcounts = pstmt.executeBatch();
pstmt.setInt(1, 5000);
pstmt.setString(2, "Amy Feiner");
int scount = pstmt.executeUpdate();   // OK; pstmt batch was executed
...
```

ある文オブジェクトのバッチ処理されない操作と、別の文オブジェクトのバッチ処理される操作を、コード上に混在させることはできます。異なる文オブジェクトは、バッチ更新操作に関して、互いに無関係です。COMMIT要求は、バッチ処理されない操作すべてと、処理済バッチの正常な操作すべてに影響を与えますが、保留バッチには影響を与えません。

21.1.3 早期バッチ・フラッシュ

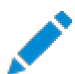
早期バッチ・フラッシュは、キャッシュされたメタデータが変更されると発生します。キャッシュされたメタデータは、次のような様々な理由から変更されることがあります。

- 最初のバインドがNULLで後続のバインドが非NULLである場合。
- 最初は文字列としてスカラー型がバインドされ、後でスカラー型としてバインドされた場合。あるいはその逆の場合。

早期バッチ・フラッシュ・カウントは、次のexecuteUpdateメソッドまたはsendBatchメソッドの戻り値に追加されます。

以前の機能では、ここで取得できるすべてのバッチ・フラッシュ値が失われました。以前の機能に切り替えるには、次に示すようにAccumulateBatchResultプロパティをfalseに設定します。

```
java.util.Properties info = new java.util.Properties();
info.setProperty("user", "HR");
info.setProperty("passwd", "hr");
// other properties
...
// property: batch flush type
info.setProperty("AccumulateBatchResult", "false");
OracleDataSource ods = new OracleDataSource();
ods.setConnectionProperties(info);
ods.setURL("jdbc:oracle:oci:@"");
Connection conn = ods.getConnection();
```



ノート:

AccumulateBatchResult プロパティはデフォルトで true に設定されます。

例21-2は、早期バッチ・フラッシュを示しています。

例21-2 早期バッチ・フラッシュ

```
((OraclePreparedStatement)pstmt).setExecuteBatch (2);
pstmt.setNull(1, OracleTypes.NUMBER);
pstmt.setString(2, "test11");
int count = pstmt.executeUpdate(); // returns 0
/*
* Premature batch flush happens here.
*/
pstmt.setInt(1, 22);
pstmt.setString(2, "test22");
int count = pstmt.executeUpdate(); // returns 0
pstmt.setInt(1, 33);
pstmt.setString(2, "test33");
/*
*  returns 3 with the new batching scheme where as,
*  returns 2 with the old batching scheme.
```

```
*/
int count = pstmt.executeUpdate();
```

21.2 その他のOracleパフォーマンス拡張機能

Oracle JDBCドライバでは、バッチ更新に加えて、次の拡張機能がサポートされています。これにより、データベースへのラウンドトリップが減少し、パフォーマンスが向上します。

- 行のプリフェッチ

  行をプリフェッチすることにより、データがフェッチされるたびに複数行がフェッチされるので、データベースへのラウンドトリップが減少します。余分にフェッチされたデータは、後で使用するためにクライアント側バッファに格納されます。プリフェッチの行数は、目的に応じて設定できます。

- 列型の指定

  問合せの実行および問合せ結果の取出しを行うときに、標準のJDBCプロトコルで生じる非効率性を回避できます。

- データベース・メタデータTABLE_REMARKS列の抑止

  コストの高い外部結合操作を回避できます。

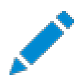
Oracleでは、これらのパフォーマンス拡張機能をサポートするために、接続プロパティ・オブジェクトにいくつかの拡張機能を提供します。これらの拡張機能により、`remarksReporting`フラグと、行プリフェッチとバッチ更新のデフォルト値を設定できます。

この項の内容は次のとおりです。

- LOBデータのプリフェッチについて
- Oracle行プリフェッチの制限事項
- 列型の定義について
- DatabaseMetaData TABLE_REMARKSのレポートについて

21.2.1 LOBデータのプリフェッチについて

Oracle Database 11gリリース2 JDBCドライバより前のJDBCドライバの場合、1回のラウンドトリップでLOBデータを取得するには、データをVARCHAR2型としてフェッチする必要があります。つまり、`OracleTypes.VARCHAR`または`OracleTypes.LONGVARCHAR`を、JDBC `defineColumnType`メソッドとともに使用する必要があります。この手法の制限は、LOBデータをCHAR型としてフェッチする場合、ロケータをデータとともにフェッチすることができないということです。このため、アプリケーションがLOBデータを後から取得する場合、またはアプリケーションが他のLOB操作を実行する場合、アプリケーションがLOBロケータを取得できないため、LOBロケータを取得するためにラウンドトリップがもう1回必要になります。



ノート:

LOB ロケータでの配列操作は、JDBC API でサポートされていません。

Oracle Database 11gリリース2以降のJDBCドライバの場合、通常のフェッチ操作中にロケータとともにLOBデータの始めをプリフェッチするのと同様に、LOBの長さやチャンク・サイズなどの使用頻度の高いメタデータをプリフェッチすることで、ラウンドトリップの

回数を減らします。LOBが小さい場合、ただ1回のラウンドトリップでデータをすべてプリフェッチできる場合があります。select、パース、実行およびフェッチが1回のラウンドトリップで行われるため、パフォーマンスが大幅に向上します。LOBがプリフェッチ・サイズの5倍より大きい場合、チャンク・サイズを取得するためのラウンドトリップが必要ないだけなので、パフォーマンスはそれほど向上しません。

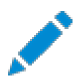
defaultLobPrefetchSize接続プロパティ

Oracle Database 11gリリース2から、接続のためのデフォルトLOBプリフェッチ・サイズを設定するために使用できる新しい接続プロパティoracle.jdbc.defaultLobPrefetchSizeが導入されました。この接続プロパティは、定数OracleConnection.CONNECTION_PROPERTY_DEFAULT_LOB_PREFETCH_SIZEとして定義されています。このプロパティの値が、現行接続のデフォルトのLOBプリフェッチ・サイズとして使用されます。この接続プロパティのデフォルト値は4000です。文レベルでこのデフォルト値を変更する場合は、oracle.jdbc.OracleStatementインタフェースで定義されているsetLobPrefetchSizeメソッドを使用します。デフォルト値は、次の値に変更できます。

- 現行接続のLOBプリフェッチを無効にするには-1
- メタデータ専用のLOBプリフェッチを有効にするには0
- フェッチ操作中にロケータとともにプリフェッチするBLOBのバイト数およびCLOBの文字数を指定するには、0より大きな任意の値

LOBプリフェッチ・サイズを取り出すには、oracle.jdbc.OracleStatementインタフェースで定義されているgetLobPrefetchSizeメソッドを使用します。

defineColumnTypeメソッドを使用して列レベルでLOBプリフェッチ・サイズの値も設定できます。列レベルの値は、接続または文レベルで設定されているどの値よりも優先されます。



ノート:

LOB プリフェッチが接続レベルまたは文レベルで無効でない場合、列レベルでも無効にできません。

21.2.2 Oracle行プリフェッチの制限事項

最大のプリフェッチ量は設定されていません。デフォルト値は10です。問合せから予想される行数および列数に基づいて、値を増減させた方が適している場合もあります。Propertiesオブジェクトを使用して、接続のデフォルト行プリフェッチ値を設定できます。

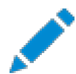
Statementオブジェクトは、作成されると、関連付けられた接続から行のプリフェッチ設定のデフォルト値を受け取ります。接続の行プリフェッチ設定のデフォルト値を後で変更しても、文の行プリフェッチ設定は変更されません。

結果セットの列のデータ型が、データ・インタフェースから戻されたLONG、LONG RAWまたはLOB、つまりストリーム型の場合は、いずれの型の値も実際に読み取らなくても、文の行プリフェッチ設定はJDBCによって1に変更されます。

プリフェッチ・サイズの設定は、アプリケーションのパフォーマンスに影響を与える場合があります。プリフェッチ・サイズを大きくすると、全データの取得に必要なラウンドトリップ数は減りますが、メモリーの使用量は増えます。これは、問合せ内の列の数とサイズ、および戻されることが予想される行数に依存します。また、JDBCクライアント・マシンのメモリーおよびCPU負荷にも依存します。ス

タンドアロン・クライアント・アプリケーションの最適値は、負荷の大きいアプリケーション・サーバーの場合と異なります。ネットワーク接続の速度および待機時間も考慮してください。

ノート:



Oracle Database 11g リリース 1 以降、Thin ドライバは、最初のラウンドトリップで、最初の prefetch_size 行をサーバーからフェッチすることができます。これにより、SELECT 文でラウンドトリップを 1 回減らせます。

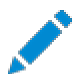
10gリリース1(10.1)以上のOracle JDBCドライバにそれより前のリリースのOracle JDBCドライバからアプリケーションを移行する場合は、以前検討した最適化を再検討する必要があります。メモリーの使用量およびパフォーマンス特性が大きく変化している場合があるからです。

一意なキーを選択する問合せなどは、よく実行する処理です。そのような問合せは、行をゼロ個または1つ戻します。プリフェッチ・サイズを1に設定すると、メモリーおよびCPUの使用量は減り、ラウンドトリップ数は増えません。if(rs.next())のかわりにwhile(rs.next())を記述して余分なフェッチを要求するというミスを避けるよう、注意する必要があります。

JDBC Thinドライバを使用する場合、useFetchSizeWithLongColumn接続プロパティを使用してください。PARSE、EXECUTEおよびFETCHが単一のラウンドトリップ内で実行されるからです。

プリフェッチ・サイズのチューニングは、実際のアプリケーションの現実的な負荷に基づき、JVM内でのメモリー管理のチューニングに合せて行う必要があります。

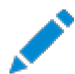
ノート:



- JDBC 2.0 フェッチ・サイズ Application Program Interface(API)と Oracle 行プリフェッチ API をアプリケーションで混在させることはできません。どちらも使用できますが、両方は使用できません。
- Oracle 行フェッチ・サイズ値を設定すると、問合せ以外に、結果セットの refreshRow メソッドによる結果セットの行の明示的な再フェッチ(scroll-sensitive/読取り専用、scroll-sensitive/更新可能およびscroll-insensitive/更新可能結果セットに関係します)行数および scroll-sensitive 結果セットのウィンドウ・サイズ(自動再フェッチの実行頻度に影響します)にも影響を与えることがあります。ただし、Oracle 行フェッチ・サイズ値は、フェッチ・サイズの設定によってオーバーライドされます。

21.2.3 列型の定義について

ノート:



Oracle Database 12c リリース 1 (12.1)以降、defineColumnType メソッドは非推奨です。

Oracle Database 10gでは、defineColumnTypeの実装が大幅に変更されました。これまでは、defineColumnTypeはパフォーマンスの最適化とデータ型変換の強制実行の両方に使用されていました。以前のリリースでは、すべてのドライバがdefineColumnTypeのコールによる利点を得ていました。Oracle Database 10gから、JDBC Thinドライバはこの情報の提

供を必要としなくなりました。JDBC Thinドライバは、defineColumnTypeをコールせずに最大のパフォーマンスを実現します。アプリケーションでdefineColumnTypeを使用すると、JDBC Oracle Call Interface(OCI)およびサーバー側内部ドライバのパフォーマンスが向上します。

コードでJDBC ThinドライバおよびJDBC OCIドライバの両方を使用する場合は、Thinドライバの使用時に接続プロパティdisableDefineColumnTypeをtrueに設定することで、defineColumnTypeメソッドを無効にできます。その結果、defineColumnTypeは無視されます。JDBC OCIまたはサーバー側内部ドライバを使用する場合は、この接続プロパティをtrueに設定しないでください。

また、defineColumnTypeを使用して、クライアント側が割り当てるメモリー量を制御することや、可変長データのサイズを制限することもできます。

問合せの列型を定義するには、次のステップを実行します。

1. 必要に応じて、StatementオブジェクトをOracleStatement、OraclePreparedStatementまたはOracleCallableStatementにキャストします。
2. 必要に応じて、StatementオブジェクトのclearDefinesメソッドを使用して、このStatementオブジェクトの以前の列定義を消去します。
3. 各列で、StatementオブジェクトのdefineColumnTypeメソッドをコールして次のパラメータを渡します。
    - 列索引(整数)
    - 型コード(整数)

      java.sql.Typesクラスまたはoracle.jdbc.OracleTypesクラスのstatic定数を使用します(Types.INTEGER、Types.FLOAT、Types.VARCHAR、OracleTypes.VARCHAR、OracleTypes.ROWIDなど)。この2つのクラスで、標準型の型コードは同一です。
    - 型名(文字列)

      構造化オブジェクト、オブジェクト参照および配列の場合は、型名も指定する必要があります。たとえば、Employee、EmployeeRef、EmployeeArrayなどです。
    - 最大フィールド・サイズ(整数)

      オプションで、この列の最大データ長も指定できます。

      構造化オブジェクト、オブジェクト参照または配列の列型を定義する場合、最大フィールド・サイズ・パラメータは指定できません。このパラメータを含めても、無視されます。
    - 使用する形式(short)

      オプションで、この列の使用する形式も指定できます。データベース文字セットを使用する場合はOraclePreparedStatement.FORM_CHARに、各国語文字セットを使用する場合はOraclePreparedStatement.FORM_NCHARに設定します。このパラメータを省略した場合、デフォルトはFORM_CHARです。

   たとえば、stmtをOracle文と仮定して、次のように使用します。

   ```
   stmt.defineColumnType(column_index, typeCode);
   ```

   列がVARCHARまたは等価な型で、長さの制限がわかっている場合は、次のようにします。

```
stmt.defineColumnType(column_index, typeCode, max_size);
```

列がNVARCHARで、元の最大長が必要、かつデータベース文字セットへの変換が要求される場合は、次のようにします。

```
stmt.defineColumnType(column_index, typeCode, 0,
   OraclePreparedStatement.FORM_CHAR );
```

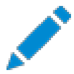
列が構造化オブジェクト、オブジェクト参照および配列の場合は、次のようにします。

```
stmt.defineColumnType(column_index, typeCode, typeName);
```

デフォルトのデータ長をすべて受け取る必要がない場合は、最大フィールド・サイズを設定します。標準JDBC StatementクラスのsetMaxFieldSizeメソッドをコールして、戻されるデータ量の制限を設定します。つまり、戻されるデータのサイズは、次の値の最小値になります。

- defineColumnTypeで設定された最大フィールド・サイズ
- setMaxFieldSizeで設定された最大フィールド・サイズ
- データ型固有の最大サイズ

これらのステップを完了した後に、文のexecuteQueryメソッドを使用して問合せを実行します。



ノート:

必要な結果セットのすべての列に対してデータ型を指定する必要はありません。

この機能の使用例を次に示します。この例では、oracle.jdbc.*インタフェースがインポート済であることを前提にしています。

例21-3 列型の定義

```
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:thin:@localhost:5221:orcl");
ods.setUser("HR");
ods.setPassword("hr");
Connection conn = ods.getConnection();
Statement stmt = conn.createStatement();
// Allocate only 2 chars for this column (truncation will happen)
((OracleStatement)stmt).defineColumnType(1, Types.VARCHAR, 2);
ResultSet rset = stmt.executeQuery("select ename from emp");
while(rset.next() )
  System.out.println(rset.getString(1));
stmt.close();
```

この例が示すように、defineColumnTypeメソッドの起動では、StatementオブジェクトstmtをOracleStatementにキャストする必要があります。接続のcreateStatementメソッドは、defineColumnTypeメソッドとclearDefinesを持たず、型がjava.sql.Statementであるオブジェクトを戻します。これらのメソッドは、OracleStatement実装でのみ提供されています。

定義の拡張要素では、JDBC型を使用して目的の型を指定します。使用可能な列型は、列のOracle内部型によって異なります。

すべての列は、本来のJDBC型に定義できます。多くの場合、Types.CHARまたはTypes.VARCHAR型コードに定義できます。

次の表に、defineColumnTypeメソッドで使用できる有効な列定義引数を示します。

表21-1 有効な列型

列内のOracle SQL型	defineColumnTypeで定義できる型
NUMBER、VARNUM	BIGINT、TINYINT、SMALLINT、INTEGER、FLOAT、REAL、DOUBLE、NUMERIC、DECIMAL、CHAR、VARCHAR
CHAR、VARCHAR2	CHAR、VARCHAR
LONG	CHAR、VARCHAR、LONGVARCHAR
LONGRAW	LONGVARBINARY、VARBINARY、BINARY
RAW	VARBINARY、BINARY
DATE	DATE、TIME、TIMESTAMP、CHAR、VARCHAR
ROWID	ROWID
BLOB	VARBINARY、BINARY
CLOB	LONG、CHAR、VARCHAR

defineColumnTypeを列の元のデータ型で使用することは、常に可能です。

21.2.4 DatabaseMetaData TABLE_REMARKSのレポートについて

データベース・メタデータ・クラスのgetColumns、getProcedureColumns、getProceduresおよびgetTablesメソッドを使用してTABLE_REMARKS列をレポートすると、コストの高い外部結合を必要とするため処理が遅くなります。この理由でJDBCドライバは、デフォルトではTABLE_REMARKS列を報告しません。

OracleConnectionオブジェクトのsetRemarksReportingメソッドにtrue引数を渡すと、TABLE_REMARKSレポートが使用可能になります。

Java Propertiesオブジェクトを使用して接続を確立した場合、setRemarksReportingをコールするかわりに、remarksReporting Javaプロパティを設定することもできます。

標準java.sql.Connectionオブジェクトを使用している場合、setRemarksReportingを使用するには、オブジェクトをOracleConnectionにキャストする必要があります。

次のコードでは、TABLE_REMARKSレポートを使用可能にする方法を示します。

```
((oracle.jdbc.OracleConnection)conn ).setRemarksReporting(true);
```

ここで、connは標準Connectionオブジェクト名を表しています。次の文を使用するとTABLE_REMARKSレポートが使用可能にな

ります。

getColumnsの考慮事項

シノニムが指定されている場合、デフォルトでは、getColumnsメソッドは列の情報を取り出しません。シノニムが指定されている場合に情報の取出しを可能にするには、次のように、接続でsetIncludeSynonymsメソッドをコールする必要があります。

```
((oracle.jdbc.OracleConnection)conn ).setIncludeSynonyms(true)
```

これによって、その接続での後続のgetColumnsメソッドのコールすべてにシノニムが含まれます。これは、setRemarksReportingと同じです。あるいは、includeSynonyms接続プロパティを設定できます。これは、remarksReporting接続プロパティと同じです。

ただし、includeSynonymsがtrueに設定されている場合、シノニムが存在すると、table_name列に戻されるオブジェクト名は、シノニム名であることに留意してください。これは、表名をgetColumnsに渡す場合でも同じです。

getProceduresおよびgetProcedureColumnsメソッドの考慮事項

JDBCバージョン1.1および1.2では、getProceduresおよびgetProcedureColumnsメソッドは、catalog、schemaPattern、columnNamePatternおよびprocedureNamePatternパラメータを同じ方法で処理します。これらのメソッドに関するOracleの定義では、パラメータの処理方法は次のように異なります。

- catalog

  Oracleには複数カタログはありませんが、複数パッケージはあります。したがって、catalogパラメータは、パッケージ名として扱われます。これは入力(catalogパラメータ)と出力(戻されたResultSetのcatalog列)の両方に該当します。入力時に、構成" "(空の文字列)は、プロシージャと引数を、パッケージ(スタンドアロンのオブジェクト)なしで取得します。null値は、選択基準(スタンドアロン・オブジェクトとパッケージ化されたオブジェクトの両方に関する戻し情報)から除外されることを意味します。つまり、パーセント記号(%)を渡すのと同じ効果です。それ以外の場合は、catalogパラメータには、パッケージ名パターンを指定します。必要に応じてSQLワイルドカードを付けられます。

- schemaPattern

  Oracle Database内のオブジェクトはすべてはスキーマを持つ必要があるため、スキーマを持たないオブジェクトの情報を戻すことには意味がありません。このため、構成" "(空の文字列)は、入力時に、現在のスキーマ、つまり現在接続しているスキーマを持つオブジェクトを意味するものと解釈されます。catalogパラメータの動作との整合性を取るため、nullは選択基準からスキーマを除外するものと解釈されます。つまり、%を渡すのと同じ効果です。SQLワイルドカードを使用したパターンとしても使用できます。

- procedureNamePatternおよびcolumnNamePattern

  プロシージャと引数はすべて名前を持っているため、空の文字列(" ")はいずれのパラメータについても意味を持ちません。このため、構成" "を使用すると例外が発生します。他のパラメータの動作との整合性を取るため、nullはパーセント記号(%)を渡すのと同じ効果を持ちます。

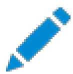
22 OCI接続プーリング

Java Database Connectivity(JDBC)Oracle Call Interface(OCI)ドライバ接続プーリング機能は、JDBCクライアントの一部です。この機能は、OracleOCIConnectionPoolクラスによって提供されます。

JDBCアプリケーションでは、同時に複数のプールを保持できます。複数のプールを保持すると、複数のアプリケーション・サーバーや異なるデータソースの各プールに対応できます。JDBC OCIドライバで提供される接続プーリングでは、アプリケーションで少数の物理接続をすべて使用して、複数の論理接続を保持できます。この論理接続でのコールは、コールの時点で使用可能な物理接続にルーティングされます。

この章の構成は、次のとおりです。

- OCIドライバ接続プーリングの背景
- OCIドライバ接続プーリングと共有サーバーの比較
- OCI接続プールの定義について
- OCI接続プールへの接続について
- OCI接続プーリングのサンプル・コード
- 文の処理とキャッシュ
- JNDIおよびOCI接続プール



ノート:

セッションの多重化が必要な場合は、OCI 接続を使用してください。そうでない場合は、ユニバーサル接続プールの使用をお薦めします。

22.1 OCIドライバ接続プーリングの背景

Oracle JDBC OCIドライバは、Oracleセッションおよび接続の細かな管理などのいくつかのトランザクション・モニター機能を提供します。ハイエンド・アプリケーション・サーバーまたはトランザクション・モニターでは、コール・レベルで少数の物理接続を通じて、複数のセッションを多重化できるため、接続とバックエンドのOracleサーバー・プロセスをプーリングすることで、高度な拡張性を実現できます。

OracleOCIConnectionPoolインタフェースで提供される接続プーリングは、物理接続プールの管理を見えなくさせ、セッションと接続が分離されたインタフェースを単純化します。Oracleセッションとは、OracleOCIConnectionPoolから取得されたOracleOCIConnectionオブジェクトです。接続プール自体は通常、はるかに小さい複数の物理接続共有プールで構成され、同じ数の専用サーバー・プロセスを含むバックエンド・サーバー・プールに変換されます。少ない数の共有接続とバックエンドOracleプロセスを含むこのプールを介して、はるかに多い数のOracleセッションを多重化できる点に注目してください。

22.2 OCIドライバ接続プーリングと共有サーバーの比較

ある意味では、OCIドライバ接続プーリングが中間層で提供する機能は、共有サーバー・プロセスがバックエンドで提供する機能と同じです。OCIドライバ接続プーリングでは、中間層でセッションの多重化ロジックを管理することで、専用サーバー・インスタンスが共有インスタンスと同じように動作します。したがって、専用サーバー・プロセスと専用サーバー・プロセスへの接続のプーリングは、中間層にあるOCI接続プールで制御されます。

OCI接続プーリングと共有サーバーの主な違いは、共有サーバーを使用する場合、通常はデータベース・インスタンス内のディスパッチャに対してクライアントから接続が行われることです。ディスパッチャには、クライアント要求を適切な共有サーバーに送信します。他方、OCI接続プールからの物理接続は、バックエンド・サーバー・プールにあるOracle専用サーバー・プロセスに対して中間層から直接確立されます。

OCI接続プールが有効なのは、中間層がマルチスレッドである場合のみです。各スレッドは、データベースに対して1つのセッションを保持できます。データベースに対する実際の接続は、OracleOCIConnectionPoolによって保持され、これらの接続は、専用データベース・サーバー・プロセスのプールも含めて、中間層のすべてのスレッドで共有されます。

22.3 OCI接続プールの定義について

この項では、次の概念について説明します。

- OCI接続プールの作成の概要
- oracle.jdbc.poolパッケージおよびoracle.jdbc.ociパッケージのインポート
- OCI接続プールの作成
- OCI接続プール・パラメータの設定
- OCI接続プール・ステータスのチェック

22.3.1 OCI接続プールの作成の概要

OCI接続プールは、アプリケーションの開始時に作成されます。プールから接続を作成する方法は、OracleDataSourceクラスを使用して接続を作成する方法とほぼ同じです。

OCI接続プールの作成には、OracleDataSourceクラスを拡張するoracle.jdbc.pool.OracleOCIConnectionPoolクラスを使用します。OracleOCIConnectionPoolインスタンスから、論理接続オブジェクトを取得できます。これらの接続オブジェクトは、OracleOCIConnectionクラス型です。このクラスは、OracleConnectionインタフェースを実装します。OracleOCIConnectionインスタンスから作成したStatementオブジェクトには、OracleConnectionインスタンスから作成したOracleStatementオブジェクトと同じフィールドおよびメソッドがあります。

次のコードは、OracleOCIConnectionPoolクラスのヘッダー情報を示します。

```
/*
   * @param us   ConnectionPool user-id.
   * @param p    ConnectionPool password
   * @param name  logical name of the pool. This needs to be one in the
   *                   tnsnames.ora configuration file.
     @param config (optional)  Properties of the pool, if the default does not
```

```
                suffice. Default connection configuration is min =1, max=1,
                incr=0
                  Please refer setPoolConfig for property names.
                 Since this is optional, pass null if the default configuration
                 suffices.
   * @return
   *
   * Notes: Choose a userid and password that can act as proxy for the users
   *        in the getProxyConnection() method.
            If config is null, then the following default values will take
            effect
            CONNPOOL_MIN_LIMIT = 1
            CONNPOOL_MAX_LIMIT = 1
            CONNPOOL_INCREMENT = 0
*/
public synchronized OracleOCIConnectionPool
  (String       user,    String   password,  String name, Properties config)
  throws SQLException
/*
 * This will use the user-id, password and connection pool name values set
   LATER using the methods setUser, setPassword, setConnectionPoolName.
 * @return
 *
 * Notes:
     No OracleOCIConnection objects can be created on
     this class unless the methods setUser, setPassword, setPoolConfig
     are invoked.
     When invoking the setUser, setPassword later, choose a userid and
     password that can act as proxy for the users
 *   in the getProxyConnection() method.
 */
  public synchronized OracleOCIConnectionPool ()
    throws SQLException
```

22.3.2 oracle.jdbc.poolパッケージおよびoracle.jdbc.ociパッケージのインポート

OCI接続プールを作成する前に、Oracle OCI接続プーリング機能に次のパッケージをインポートする必要があります。

```
import oracle.jdbc.pool.*;
import oracle.jdbc.oci.*;
```

22.3.3 OCI接続プールの作成

次のコードは、cpoolという名前のOracleOCIConnectionPoolクラスのインスタンスを作成する方法を示します。

```
OracleOCIConnectionPool cpool = new OracleOCIConnectionPool
    ("HR", "hr", "jdbc:oracle:oci:@(description=(address=(host=
    localhost)(protocol=tcp)(port=5221))(connect_data=(INSTANCE_NAME=orcl)))",
    poolConfig);
```

poolConfigは、接続プールを指定する一連のプロパティです。poolConfigがNULLの場合は、デフォルト値が使用されます。たとえば、次の例を考えてみます:

- poolConfig.put (OracleOCIConnectionPool.CONNPOOL_MIN_LIMIT, "4");
- poolConfig.put (OracleOCIConnectionPool.CONNPOOL_MAX_LIMIT, "10");

- poolConfig.put (OracleOCIConnectionPool.CONNPOOL_INCREMENT, "2");

コンストラクタ・コール以外にも、個々のメソッドを使用してユーザー、パスワードおよび接続文字列を指定して、OracleOCIConnectionPoolクラスのインスタンスを作成する方法もあります。

```
OracleOCIConnectionPool cpool = new OracleOCIConnectionPool ( );
cpool.setUser("HR");
cpool.setPassword("hr");
cpool.setURL("jdbc:oracle:oci:@(description=(address=(host=
    localhost)(protocol=tcp)(port=5221))(connect_data=(INSTANCE_NAME=orcl)))");
cpool.setPoolConfig(poolConfig);  // In case you want to specify a different
                                   // configuration other than the default
                                   // values.
```

22.3.4 OCI接続プール・パラメータの設定

接続プールの構成は、次に示すOracleOCIConnectionPoolクラス属性によって決まります。

- CONNPOOL_MIN_LIMIT

  プールで保持できる物理接続の最小数を指定します。

- CONNPOOL_MAX_LIMIT

  プールで保持できる物理接続の最大数を指定します。

- CONNPOOL_INCREMENT

  既存の接続がすべて使用中で、さらに接続が必要になったとき、オープンする物理接続の増分数を指定します。オープンしている物理接続の数が、そのプールについてオープンできる接続の最大数を超えない場合にのみ、接続がオープンされます。

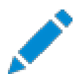
- CONNPOOL_TIMEOUT

  物理接続が切断されるまでのアイドル状態の時間を指定します。これは論理接続には影響しません。

- CONNPOOL_NOWAIT

  この属性を有効に設定すると、プール内の最大数の接続が使用されている場合に、コールに物理接続が必要になるとエラーが返されます。この属性を無効に設定すると、接続が使用可能になるまでコールは待機します。この属性をtrueに設定した後で、falseにリセットすることはできません。

これらの属性はすべて動的に構成できます。したがって、アプリケーションでは現行ロード、つまりオープンしている接続の数と使用中の接続の数を読み取って、setPoolConfigメソッドを使用してこれらの属性を適切に調整できます。



ノート:

CONNPOOL_MIN_LIMIT、CONNPOOL_MAX_LIMIT および CONNPOOL_INCREMENT パラメータのデフォルト値は、それぞれ 1、1 および 0 です。

OCI接続プールのプロパティを構成するには、setPoolConfigメソッドを使用します。次は、OracleOCIConnectionPoolクラス属性の代表的な設定例です。

```
...
java.util.Properties p  = new java.util.Properties( );
p.put (OracleOCIConnectionPool.CONNPOOL_MIN_LIMIT, "1");
```

```
p.put (OracleOCIConnectionPool.CONNPOOL_MAX_LIMIT, "5");
p.put (OracleOCIConnectionPool.CONNPOOL_INCREMENT, "2");
p.put (OracleOCIConnectionPool.CONNPOOL_TIMEOUT, "10");
p.put (OracleOCIConnectionPool.CONNPOOL_NOWAIT, "true");
cpool.setPoolConfig(p);
...
```

これらの属性を設定するときは、次の規則に従ってください。

- CONNPOOL_MIN_LIMIT、CONNPOOL_MAX_LIMITおよびCONNPOOL_INCREMENTパラメータは必須です。
- CONNPOOL_MIN_LIMITの値はゼロより大きくしてください。
- CONNPOOL_MAX_LIMITは、CONNPOOL_MIN_LIMITとCONNPOOL_INCREMENTを足した値以上にしてください。
- CONNPOOL_INCREMENTの値はゼロ以上にしてください。
- CONNPOOL_TIMEOUTの値はゼロより大きくしてください。
- CONNPOOL_NOWAITは、trueまたはfalseにしてください。

関連項目:

『Oracle Call Interfaceプログラマーズ・ガイド』

22.3.5 OCI接続プール・ステータスのチェック

接続プールのステータスをチェックするには、OracleOCIConnectionPoolクラスの次のメソッドを使用します。

- int getMinLimit()

  プールで保持できる物理接続の最小数を取得します。

- int getMaxLimit()

  プールで保持できる物理接続の最大数を取得します。

- int getConnectionIncrement()

  既存の接続がすべて使用中に、コールに接続が必要になったときにオープンする物理接続の増分数を取得します。

- int getTimeout()

  プールの物理接続が切断されるまでのアイドル状態の時間(秒単位)を取得します。接続の持続時間は最低使用頻度(LRU)アルゴリズムに基づいて決定されます。

- String getNoWait()

  NOWAITプロパティが有効かどうかを取得します。trueまたはfalseの文字列が戻されます。

- int getPoolSize()

  オープンしている物理接続の数を取得します。統計分析用にあくまで目安として使用してください。

- int getActiveSize()

  オープンしていて、かつ使用中である物理接続の数を取得します。統計分析用にあくまで目安として使用してください。

- boolean isPoolCreated()

  プールが作成されているかどうかを取得します。実際にプールが作成されるのは、OracleOCIConnection (user,

password, url, poolConfig)をコールしたとき、またはOracleOCIConnection()をコールした後にsetUser、setPasswordおよびsetURLを実行したときです。

22.4 OCI接続プールへの接続について

OracleOCIConnectionPoolクラスは、getConnectionメソッドをコールして、OracleOCIConnectionクラスのインスタンスを作成します。このインスタンスは1つの接続を表します。

OracleOCIConnectionクラスはOracleConnectionクラスを拡張したものなので、このクラスの機能も持っています。ユーザー・セッションが終了したらOracleOCIConnectionオブジェクトをクローズしてください。そうしない場合は、プール・インスタンスがクローズされるときにクローズされます。

getConnectionをコールするには、次の2つの方法があります。

- OracleConnection getConnection()

  ユーザー名とパスワードを指定しない場合、接続プールの作成に使用されたデフォルトのユーザー名とパスワードを使用して、接続オブジェクトが作成されます。

- OracleConnection getConnection(String user, String password)

  このメソッドを使用すると、指定されたユーザー名とパスワードで識別される論理接続を取得します。このユーザー名とパスワードは、プールの作成に使用されたものとは異なります。

次のコードは、オーバーロードされたgetConnectionメソッドのシグネチャを示します。

```
public synchronized OracleConnection getConnection( )
                    throws SQLException
/*
 * For getting a connection to the database.
 *
 * @param us  Connection user-id
 * @param p   Connection password
 * @return     connection object
 */
public synchronized OracleConnection getConnection(String us, String p)
throws SQLException
```

OracleConnectionの拡張機能として、ユーザーのパスワードを変更するために次のような新しいメソッドがOracleOCIConnectionに追加されています。

```
void passwordChange (String user, String oldPassword, String newPassword)
```

22.5 OCI接続プーリングのサンプル・コード

次のコードは、サンプル・アプリケーションでのOCI接続プールの使用方法を示しています。

```
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Properties;
import oracle.jdbc.OracleDriver;
```

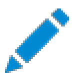
```
import oracle.jdbc.pool.OracleOCIConnectionPool;

public class conPoolAppl extends Thread
{
  public static final String query = "SELECT object_name FROM all_objects WHERE rownum < 300";
  static public void main(String args[]) throws SQLException
  {
    int _maxCount = 10;
    Connection []conn = new Connection[_maxCount];
    try
    {
      String s = null;   //System.getProperty ("JDBC_URL");
      String url = "jdbc:oracle:oci8:@localhost";
      OracleOCIConnectionPool cpool = new OracleOCIConnectionPool("HR", "hr", url, null);

      // Print out the default configuration for the OracleOCIConnectionPool
      System.out.println ("-- The default configuration for the OracleOCIConnectionPool --");
      displayPoolConfig(cpool);

      //Set up the initial pool configuration
      Properties p1  = new Properties();
      p1.put (OracleOCIConnectionPool.CONNPOOL_MIN_LIMIT, Integer.toString(1));
      p1.put (OracleOCIConnectionPool.CONNPOOL_MAX_LIMIT, Integer.toString(_maxCount));
      p1.put (OracleOCIConnectionPool.CONNPOOL_INCREMENT, Integer.toString(1));

      // Enable the initial configuration
      cpool.setPoolConfig(p1);

      Thread []t = new Thread[_maxCount];
      for (int i = 0; i < _maxCount; ++i)
      {
        conn[i] = cpool.getConnection("HR", "hr");
        if ( conn[i] == null )
        {
          System.out.println("Unable to create connection.");
          return;
        }
        t[i] = new conPoolAppl (i, conn[i]);
        t[i].start ();
        //displayPoolConfig(cpool);
      }

      ((conPoolAppl)t[0]).startAllThreads ();
      try
      {
        Thread.sleep (200);
      }
      catch (Exception ea) {}

      displayPoolConfig(cpool);
      for (int i = 0; i < _maxCount; ++i)
        t[i].join ();
    }
    catch(Exception ex)
    {
      System.out.println("Error: " + ex);
      ex.printStackTrace ();
      return;
```

```
  }
  finally
  {
    for (int i = 0; i < _maxCount; ++i)
      if (conn[i] != null)
      conn[i].close ();
  }
} //end of main

private Connection m_conn;
private static boolean m_startThread = false;
private int m_threadId;

public conPoolAppl (int i, Connection conn)
{
  m_threadId = i;
  m_conn = conn;
}

public void startAllThreads ()
{
  m_startThread = true;
}

public void run ()
{
  while (!m_startThread) Thread.yield ();
  try
  {
    doQuery (m_conn);
  }
  catch (SQLException ea)
  {
    System.out.println ("*** Thread id: " + m_threadId);
    ea.printStackTrace ();
  }
} // end of run

private static void doQuery (Connection conn) throws SQLException
{
  PreparedStatement pstmt = null;
  ResultSet rs = null;
  try
  {
    pstmt = conn.prepareStatement (query);
    rs = pstmt.executeQuery ();
    while (rs.next ())
    {
      //System.out.println ("Object name: " +rs.getString (1));
    }
  }
  catch (Exception ea)
  {
    System.out.println ("Error during execution: " +ea);
    ea.printStackTrace ();
  }
  finally
  {
```

```
      if (rs != null)
        rs.close ();
      if (pstmt != null)
        pstmt.close ();
      if (conn != null)
        conn.close ();
    }
  }  // end of doQuery (Connection)

  // Display the current status of the OracleOCIConnectionPool
  private static void displayPoolConfig (OracleOCIConnectionPool cpool) throws SQLException
  {
    System.out.println (" Min poolsize Limit: " + cpool.getMinLimit());
    System.out.println (" Max poolsize Limit: " + cpool.getMaxLimit());
  /*
    System.out.println (" Connection Increment: " + cpool.getConnectionIncrement());
    System.out.println (" NoWait: " + cpool.getNoWait());
    System.out.println (" Timeout: " + cpool.getTimeout());
  */
    System.out.println (" PoolSize: " + cpool.getPoolSize());
    System.out.println (" ActiveSize: " + cpool.getActiveSize());
  }

}  // end of class conPoolAppl
```

22.6 文の処理とキャッシュ

文キャッシュは、OracleOCIConnectionPoolでサポートされます。キャッシュを使用すると、カーソルをオープン、解析およびクローズする必要がなくなるため、パフォーマンスが向上します。OracleOCIConnection.prepareStatement("a_SQL_query")を処理すると、SQL問合せに一致する文が文キャッシュから検索されます。一致する文が見つかった場合は、別のStatementオブジェクトを作成するかわりに、同じStatementオブジェクトを再利用できます。キャッシュ・サイズは動的に増減できます。デフォルトのキャッシュ・サイズは0(ゼロ)です。



ノート:

OracleOCIConnection クラスから作成した OracleStatement オブジェクトは、OracleConnection から作成したオブジェクトと同じように動作します。

22.7 JNDIおよびOCI接続プール

Java Naming and Directory Interface(JNDI)機能によりJavaオブジェクトのプロパティが存続するため、そのオブジェクトの新しいインスタンスを、それらのプロパティを使用して作成することができます(例: オブジェクトのクローニングなど)。利点は、古いオブジェクトを解放して、正確に同じプロパティを持つ新しいオブジェクトを後で作成できることです。InitialContext.lookupメソッドは永続ストアからプロパティを取得し、そのプロパティを使用して新しいオブジェクトを作成しますが、InitialContext.bindメソッドは、ファイルまたはデータベースにあるプロパティを存続させます。

JNDI機能を使用して、OracleOCIConnectionPoolオブジェクトをバインドしたり、検索したりすることができます。OracleOCIConnectionPoolで新しくインタフェースをコールする必要はありません。

23 データベース常駐接続プーリング

データベース常駐接続プール(DRCP)は、多数のクライアント間で共有されるサーバーの接続プールです。アクティブ接続数がオープン接続数より大幅に少ない場合、接続プールのDRCPを使用する必要があります。DRCPプールからの接続を共有できる接続プールのインスタンスの数が増加すると、DRCPを使用する利点も増加します。DRCPを使用すると、データベース・サーバーのスケーラビリティが向上し、中間層の接続プーリングに関連したリソースの消費の問題が解決されます。

この章の構成は、次のとおりです。

- データベース常駐接続プーリングの概要
- データベース常駐接続プーリングを使用可能にする方法
- 複数の接続プール間でのプールされたサーバーの共有について
- DRCPのタグ付け
- セッション状態の修正のためのPL/SQLコールバック
- DRCPを使用するためのAPI

23.1 データベース常駐接続プーリングの概要

中間層の接続プールでは、すべての接続キャッシュは、サーバーに対して最低限の数の接続を維持します。各接続は、サーバーの使用済リソースを示します。常に、すべてのオープン接続が使用されているわけではありません。つまり、これは、サーバーのリソースを無駄に消費する未使用のリソースがあるということです。複数の中間層のシナリオでは、これらの接続は他の中間層と共有されておらず、アイドルの接続があってもキャッシュにそのまま保持されています。ただし、すべての接続が同時にアクティブになるわけではないため、このような中間層の多数の接続プールによってデータベース・サーバーへの非アクティブな接続の数がかなり増加し、多くのデータベース・リソースが消費されています。

たとえば、中間層の接続プールで、最小のプール・サイズが200の場合、接続プールにはサーバーに対して200の接続があり、データベース・サーバーにはこれらの接続に関連付けられた200のサーバー・プロセスがあります。最小サイズ200の接続プールで30の中間層がある場合、このサーバーには、これに対応する実行中サーバー・プロセスが6000 (200 * 30)個あります。通常、サーバー・プロセスは平均5%の接続にしか使用されていません。したがって、常に、6,000個のサーバー・プロセスのうちアクティブな接続は300個のみです。これは、サーバー上で5,700個以上のサーバー・プロセスが使用されていないということです。このような使用されていないプロセスはサーバー上のリソースを無駄に消費しています。

データベース常駐接続プールを実装すると、サーバー側でプールが作成され、複数のクライアント・プール間でこれが共有されます。これによって、サーバー上でサーバー・プロセス数が減少するため、サーバーでのメモリーの消費がかなり抑えられ、データベース・サーバーのスケーラビリティが向上します。

関連項目:

- Oracle Database概要
- Oracle Database管理者ガイド

23.2 データベース常駐接続プーリングを使用可能にする方法

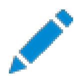
この項では、DRCPをサーバー側とクライアント側で使用可能にする方法について説明します。

- サーバー側でDRCPを使用可能にする方法
- クライアント側でDRCPを使用可能にする方法

23.2.1 サーバー側でDRCPを使用可能にする方法

プールを起動し終了するには、データベース管理者(DBA)がSYSDBAとしてログインする必要があります。この項では、次の概念について説明します。

- デフォルトの接続プールの起動
- デフォルトの接続プールの構成
- プールの終了
- 文のキャッシュ・サイズの設定



ノート:

JDBC にはデフォルトのプールがないので、DRCP の機能はクライアントの接続プールでのみ利用できます。クライアント接続プールがなく、自動コミットが false に設定されたデータベースになんらかの変更を加えた場合、接続を閉じる際にその変更はデータベースにコミットされません。

デフォルトの接続プールの起動

Oracle Databaseのデフォルトの接続プールSYS_DEFAULT_CONNECTION_POOLを起動するには、dbms_connection_pool.start_poolメソッドをデフォルト設定で実行します。たとえば:

```
sqlplus /nolog
connect / as sysdba
execute dbms_connection_pool.start_pool();
```

デフォルトの接続プールの構成

デフォルトの接続プールはデフォルトのパラメータ値を使用して構成されます。DBMS_CONNECTION_POOLパッケージ内のプロシージャを使用すると、データベース常駐接続プーリングの接続プールを構成できます。

Oracle Database 12cリリース2 (12.2.0.1)では、MAX_TXN_THINK_TIMEパラメータが導入されました。これは、進行中のトランザクションを含むプールされたサーバーの思考タイムアウトを指定するための新規パラメータです。思考タイムアウトは、クライアントがプールからプールされたサーバーを取得した後で非アクティブ状態でいられる最大時間(秒単位)です。

関連項目:

構成パラメータの詳細は、『Oracle Database管理者ガイド』を参照してください

プールの終了

プールを終了するには、dbms_connection_pool.stop_poolメソッドをデフォルト設定で実行します。たとえば:

```
sqlplus /nolog
connect / as sysdba
execute dbms_connection_pool.stop_pool();
```

文のキャッシュ・サイズの設定

DRCPを使用する場合、キャッシュもサーバー側では行われません。したがって、次のように、サーバー側で文のキャッシュ・サイズを指定する必要があります。50が最適サイズです。

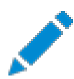
```
execute DBMS_CONNECTION_POOL.CONFIGURE_POOL (session_cached_cursors=>50);
```

関連トピック

- 文キャッシュについて

23.2.2 クライアント側でDRCPを使用可能にする方法

クライアント側でDRCPを使用可能にするには、次のステップを実行します。



ノート:

この項の例は、クライアント側の接続プールとしてユニバーサル接続プールを使用します。他の接続プールの場合、次の 2 つのステップに従い、oracle.jdbc.pool.OracleConnectionPoolDataSource をコネクション・ファクトリとして使用して、DRCP を使用可能にできます。

- nullでなく空でないString値を接続プロパティoracle.jdbc.DRCPConnectionClassに渡します
- (SERVER=POOLED)をCONNECT_DATAに長い接続文字列で追加します。

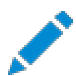
次のように、短いURLで(SERVER=POOLED)を指定することもできます。

```
jdbc:oracle:thin:@//<host>:<port>/<service_name>[:POOLED]
```

たとえば:

```
jdbc:oracle:thin:@//localhost:5221/orcl:POOLED
```

次の例は、クライアント側でDRCPを使用可能にする方法を示します。



ノート:

UCP では、接続クラスを指定しない場合、接続プール名がデフォルトで接続クラス名として使用されます。

例23-1 ユニバーサル接続プールを使用してクライアント側でDRCPを使用可能にする方法

```
String url = "jdbc:oracle:thin:@//localhost:5521/orcl:POOLED";
PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
```

```
pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
// Set DataSource Property
pds.setUser("HR");
pds.setPassword("hr");
System.out.println ("Connecting to " + url);
pds.setURL(url);
pds.setConnectionPoolName("HR-Pool1");
pds.setMinPoolSize(2);
pds.setMaxPoolSize(3);
pds.setInitialPoolSize(2);
Properties prop = new Properties();
prop.put("oracle.jdbc.DRCPConnectionClass", "HR-Pool1");
pds.setConnectionProperties(prop);
```

23.3 複数の接続プール間でのプールされたサーバーの共有について

複数の接続プール間でサーバー上にプールされたサーバー・プロセスを共有するには、同じDRCP接続クラス名をすべてのプールされたサーバー・プロセス用にサーバー上に設定します。接続プロパティoracle.jdbc.DRCPConnectionClassを使用して、DRCP接続クラス名を設定できます。

23.4 DRCPのタグ付け

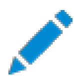
DRCPを使用すると、サーバーの接続プールに、サーバー・プロセスを特定のタグ名と関連付けるように要求できます。特定の接続にタグを適用し、後でそのタグ付けされた接続を取得できます。接続にタグを付けると、特定のセッションを簡単に取り出せるようになり、セッション・プーリングが強化されます。

Oracle Database 12cリリース2 (12.2.0.1)以降、DRCPでは複数のタグ付けがサポートされています。デフォルトでは、既存のDRCPアプリケーションとの互換性ファクタのために、この機能が無効になっています。この機能をDRCPアプリケーションで有効にするには、oracle.jdbc.UseDRCPMultipletag接続プロパティをTRUEに設定します。

複数のタグ付け機能を有効にすると、DRCPタグを設定する際に使用したものと同じAPIを使用して、複数のDRCPタグを使用できます。違いはセパレータを使用する点のみです。DRCPタグのキーと値は等号 (=)で区切り、複数のタグはセミコロン(;)で区切ります。

DRCPタグの使用時には次の点に注意してください。

- タグのキーおよび値はNULLまたは空にすることはできません。
- 複数のタグを指定する場合、左端のタグの優先度が最も高く、右端のタグの優先度が最も低くなります。
- タグ付けされた接続を取得する際に、完全一致が見つからない(すべてのタグが一致しない)場合、部分一致を検索します。



ノート:

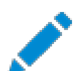
Oracle Database 12c リリース 2 (12.2.0.1)以降、同じデータベース・ユーザーに属するがプロキシ・ユーザーが異なる DRCP セッションは、プロキシ・ユーザー間で共有できます。

関連項目:

セッション・プーリングと接続のタグ付けの詳細は、『Oracle Call Interfaceプログラマーズ・ガイド』を参照してください。

23.5 セッション状態の修正のためのPL/SQLコールバック

Oracle Database 12cリリース2 (12.2.0.1)以降、セッション状態のPL/SQLベースの修正コールバックをサーバーで提供できるようになりました。このアプリケーションで提供されたコールバックは、プールからチェックアウトされたセッションをアプリケーションで要求された必要な状態に変換します。このコールバックは、データベース常駐接続プーリング(DRCP)があってもなくても動作します。



ノート:

PL/SQL ベースの修正コールバックは、複数のタグ付けの場合にのみ適用できます。

このコールバックを使用すると、修正ロジックがサーバー上のセッション状態に対して実行されるため、アプリケーションのパフォーマンスを改善できます。したがって、修正ロジックのために、この機能によりデータベースへのアプリケーションのラウンドトリップがなくなります。関連するパッケージで実行権限を付与されている適切なインストール・ユーザーは、アプリケーションのインストール時に修正コールバックを登録する必要があります。

例23-2 PL/SQLの修正コールバックの例

次に、セッション・プロパティSCHEMAおよびCURRENCYを修正するためのPL/SQL修正コールバックの実装例を示します。

```
CREATE OR REPLACE PACKAGE mycb_pack AS
PROCEDURE mycallback (
desired_props IN VARCHAR2,
actual_props IN VARCHAR2
);
END;
/
CREATE OR REPLACE PACKAGE BODY mycb_pack AS
PROCEDURE mycallback (
desired_props IN VARCHAR2,
actual_props IN VARCHAR2
) IS
property VARCHAR2(64);
key VARCHAR2(64);
value VARCHAR2(64);
pos number;
pos2 number;
pos3 number;
idx1 number;
BEGIN
idx1:=1;
pos:=1;
pos2:=1;
pos3:=1;
property := 'tmp';
```

```
-- To check if desired properties are part of actual properties
while (pos > 0 and length(desired_props)>pos)
loop
pos := instr (desired_props, ';', 1, idx1);
if (pos=0)
then
property := substr (desired_props, pos2);
else
property := substr (desired_props, pos2, pos-pos2);
end if ;
pos2 := pos+1;
pos3 := instr (property, '=', 1, 1);
key := substr (property, 1, pos3-1);
value := substr (property, pos3+1);
if (key = 'CURRENCY') then
EXECUTE IMMEDIATE 'ALTER SESSION SET NLS_CURRENCY=''' || value || '''';
elsif (key = 'SCHEMA') then
EXECUTE IMMEDIATE 'ALTER SESSION SET CURRENT_SCHEMA=' || value;
end if;
idx1 := idx1+1;
end loop;
END; -- mycallback
END mycb_pack;
/
```

関連項目:

Oracle Database JDBC Java APIリファレンス

23.6 DRCPを使用するためのAPI

カスタムの接続プールを実装するためにさらに詳細に制御してDRCPを利用する場合、`oracle.jdbc.OracleConnection`インタフェースで宣言された次のAPIを使用する必要があります。

- `attachServerConnection`
- `detachServerConnection`
- `isDRCPEnabled`
- `isDRCPMultitagEnabled`
- `getDRCPReturnTag`
- `needToPurgeStatementCache`
- `getDRCPState`

関連項目:

Oracle Database JDBC Java APIリファレンス

24 データベース・シャーディングのJDBCによるサポート

この章では、次の項でデータベース・シャーディングのJDBCによるサポートについて説明します。

- JDBCユーザー用のデータベース・シャーディングの概要
- シャーディング・キーの作成について
- データベース・シャーディングのサポート用API
- JDBCシャーディングの例

24.1 JDBCユーザー用のデータベース・シャーディングの概要

現在、Webアプリケーションには大量のデータのスケーラビリティに関する新しい課題があります。この問題に共通して受け入れられているソリューションはシャーディングです。シャーディングは、独立したデータベース間でデータが水平にパーティション化されるデータ階層アーキテクチャです。このような構成内の各データベースをシャードと呼びます。すべてのシャードが集まったものが単一の論理データベースで、これをシャード・データベース(SDB)と呼びます。シャードはCPU、メモリー、記憶域デバイスなどの物理リソースを共有しないため、シャーディングは何も共有しないデータベース・アーキテクチャです。

シャーディングはグローバル・データ・サービス(GDS)を使用します。GDSは可用性、負荷、ネットワーク待機時間およびレプリケーション・ラグに基づいて、クライアント・リクエストを適切なデータベースにルーティングします。GDSプールは、レプリケートされたデータベースのセットであり、同じグローバル・サービスを提供します。GDSプールのデータベースは、異なるリージョンにある複数のデータ・センターに配置できます。シャードされたGDSプールにはシャードされたデータベースのすべてのシャードおよびそのレプリカが含まれ、データベース・クライアントには単一のシャードされたデータベースのように見えます。

Oracle Database 12cリリース2 (12.2.0.1)以降、Oracle JDBCではデータベース・シャーディングがサポートされています。JDBCドライバは指定されたシャーディング・キーおよびスーパー・シャーディング・キーを認識し、データが含まれている該当シャードに接続します。シャードに対する接続が確立されると、DML、SQL問合せなどのサポートされているデータベース操作は通常どおりに実行されます。次の項では、このガイドで使用されるシャーディングの用語について説明します。

関連項目:

Oracle Database管理者ガイド

シャーディング、シャードおよびシャードされたデータベース

シャーディングは、独立したデータベース間でデータが水平にパーティション化されるデータ階層アーキテクチャです。このような構成の各データベースはシャードと呼ばれます。すべてのシャードが集まったものが単一の論理データベースで、これをシャード・データベース(SDB)と呼びます。

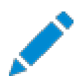
シャーディング・キー、複合シャーディング・キーおよびスーパー・シャーディング・キー

シャーディング・キーは、範囲、リストまたは一貫性のあるハッシュによって単一レベルのシャーディングで使用されるパーティション化キーです。すべてのシャーディング・キーは、あわせて複合シャーディング・キーと呼ばれます。スーパー・シャーディング・キーは、

範囲またはリストによって最上位レベルのシャーディングの複合シャーディングで使用されるパーティション化キーです。シャーディング・キーとスーパー・シャーディング・キーの両方は、各行が格納されるシャードを決定する1つ以上の列を含めることができます。シャーディング・キーは、VARCHAR2、CHAR、DATE、NUMBER、TIMESTAMP型などにすることができます。

JDBCユーザーの場合、データベースから接続を取得したときにシャーディング・キーおよびスーパー・シャーディング・キーを渡すことをお薦めします。ただし、シャーディング・キーは接続文字列でCONNECT_DATAの別の属性として指定できます。シャーディング・キーを接続文字列で渡すと、1つのシャードのみへの接続に制限されます。したがって、この方法の使用はお薦めしません。次のコードでは、シャーディング・キーを接続文字列でCONNECT_DATAの別の属性として指定する方法を示します。

```
(DESCRIPTION=(…)(CONNECT_DATA=(SERVICE_NAME=ORCL (SHARDING_KEY=…) (SUPER_SHARDING_KEY=...)))
```



ノート:

データベースで指定されている NLS 書式に準拠したシャーディング・キーを指定する必要があります。

複数のシャード問合せ

複数のシャード問合せにより、複数のシャードに格納されたデータにアクセスする問合せとトランザクションをルーティングおよび処理できます。複数のシャード問合せはシャーディング・キーを指定しないで実行されます。複数のシャード操作は、データの単純な集計およびシャード間のレポート作成時に使用されます。

シャード・カタログ

シャード・カタログは、シャードされたデータベースの格納および複数のシャード問合せのサポートに使用される特殊なデータベースです。これは、シャードされたデータベースの集中管理にも役立ちます。

シャード・ディレクタ

シャード・ディレクタは、グローバル・サービス・マネージャ(GSM)の特定の実装であり、SDBに接続するクライアント用のリージョン・リスナーとして機能し、SDBの現在のトポロジ・マップを維持します。ディレクタは、接続リクエストで渡されたシャーディング・キーに基づいて、接続を適切なシャードにルーティングします。

シャード・トポロジ

シャード・トポロジは、特定のシャードに格納されたシャーディング・キーの範囲マッピングです。ユニバーサル接続プール(UCP)はシャード・トポロジをキャッシュできます。これにより、シャードへの接続の確立時にシャード・ディレクタをバイパスできるようになります。したがって、UCPを使用して作成するアプリケーションは、シャードへのファスト・パスが得られます。

関連項目:

Oracle Universal Connection Pool開発者ガイド

チャンク

チャンクは表ファミリの各表の単一パーティションです。これは、シャード間のデータ移行の単位です。

チャンクの分割

チャンクの分割は、チャンクが大きくなりすぎた場合や、チャンクの一部のみを別のシャードに移行する必要がある場合に必要となるプロセスです。

チャンクの移行

チャンクの移行は、データまたはワークロードのスキューが発生したときに、シャード数を変更せずに、チャンクを1つのシャードから他のシャードへ移動するプロセスです。これは、ホット・スポットを回避するために、DBAで開始されます。

再シャーディング

再シャーディングは、データがシャード間に再分散されるプロセスで、シャード数の変化によってトリガーされます。チャンクは、シャード間でチャンクを均等に分散させるためにシャード間で移動します。ただし、チャンクの内容は変化しません。つまり、再シャーディング時に再ハッシュは行われません。

24.2 シャーディング・キーの作成について

シャード認識アプリケーションは、シャードされたデータベースへの接続の確立に必要なシャーディング・キーおよびスーパー・シャーディング・キーを識別して作成する必要があります。これを実行するには、シャード認識アプリケーションでOracleShardingKeyおよびOracleShardingKeyBuilderインタフェースを使用する必要があります。

OracleShardingKeyBuilderは、データ型が異なる複合キーをサポートするために次のビルダー・メソッドを使用します。

```
subkey(Object subkey, java.sql.SQLTYPE subkeyDataType)
```

各サブキーのデータ型が異なる可能性がある複合シャーディング・キーを作成するには、ビルダーでsubkeyメソッドを複数回起動します。データ型は、oracle.jdbc.OracleType列挙またはjava.sql.JDBCTypeを使用して定義できます。

例24-1 シャーディング・キーの作成

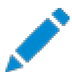
次の例は、シャーディング・キーの作成方法を示します。

```
import java.sql.Connection;
import java.sql.Date;
import java.sql.SQLException;
import java.sql.Statement;
import oracle.jdbc.OracleShardingKey;
import oracle.jdbc.OracleType;
import oracle.ucp.jdbc.PoolDataSource;
import oracle.ucp.jdbc.PoolDataSourceFactory;
  public class ShardExample
  {
    public static void main(String[] args) throws SQLException
    {
      String url =
"jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(HOST=myhost)(PORT=3216)(PROTOCOL=tcp))(CONNECT_DATA=(SERVICE
_NAME=myservice)(REGION=east)))";
      String user="testuser1";
      String pwd = "password";

      PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
```

```
        pds.setURL(url);
        pds.setUser(user);
        pds.setPassword(pwd);
        pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
        pds.setInitialPoolSize(5);
        pds.setMinPoolSize(5);
        pds.setMaxPoolSize(20);

        // build the sharding key object
        Date shardingKeyVal = new java.sql.Date(0L);
        OracleShardingKey sdkey = pds.createShardingKeyBuilder()
                                     .subkey(shardingKeyVal, OracleType.DATE)
                                     .build();

        Connection conn = pds.createConnectionBuilder()
                              .shardingKey(sdkey)
                              .build();

        Statement stmt = conn.createStatement();
        stmt.execute("... SQL statement here ...");
        stmt.close();
        conn.close();
      }
   }
```

次のコードでは、Stringデータ型とDateデータ型で構成される複合シャーディングの作成方法を示します。

```
...
Date shardingKeyVal = new java.sql.Date(0L);
...
OracleShardingKey shardingKey = datasource.createShardingKeyBuilder()
                                          .subkey("abc@xyz.com", JDBCType.VARCHAR)
                                          .subkey(shardingKeyVal, OracleType.DATE)
                                          .build();
...
```



ノート:

- 有効でサポートされているデータ型の固定のセットがあります。サポートされていないデータ型をキーとして使用すると、例外が発生します。次のリストに、サポートされているデータ型を示します。
  - `OracleType.VARCHAR2/JDBCType.VARCHAR`
  - `OracleType.CHAR/JDBCType.CHAR`
  - `OracleType.NVARCHAR/JDBCType.NVARCHAR`
  - `OracleType.NCHAR/JDBCType.NCHAR`
  - `OracleType.NUMBER/JDBCType.NUMERIC`
  - `OracleType.FLOAT/ JDBCType.FLOAT`

- OracleType.DATE/ JDBCType.DATE
- OracleType.TIMESTAMP/JDBCType.TIMESTAMP
- OracleType.TIMESTAMP_WITH_LOCAL_TIME_ZONE
- OracleType.RAW

- データベースで指定されている NLS 書式に準拠したシャーディング・キーを指定する必要があります。

24.3 データベース・シャーディングのサポート用API

Oracle Database 12cリリース2 (12.2.0.1)では、データベース・シャーディングを実装するための一連のAPIが導入されました。次の各項では、これらのAPIについて詳細に説明します。

- OracleShardingKeyインタフェース
- OracleShardingKeyBuilderインタフェース
- OracleConnectionBuilderインタフェース
- データベース・シャーディングのサポートのための他の新規クラスおよびメソッド

24.3.1 OracleShardingKeyインタフェース

このインタフェースは、現在のオブジェクトがOracleのシャードされたデータベースで使用されるOracleシャーディング・キーを表していることを示します。

構文

```
public interface OracleShardingKey extends Comparable <OracleShardingKey>
```

24.3.2 OracleShardingKeyBuilderインタフェース

OracleShardingKeyBuilderでは、サポートされている様々なデータ型のサブキーを持つ複合シャーディング・キーを作成するためのインタフェースを提供します。このインタフェースでは、新しいJDK 8ビルダー・パターンを使用してシャーディング・キーを作成します。

構文

```
public interface OracleShardingKeyBuilder
```

例24-2 シャーディング・キーの作成

```
OracleDataSource ods = new OracleDataSource();
...
//set datasource properties..
...
OracleShardingKey shardingKey = ods.createShardingKeyBuilder()
                                   .subkey("Customer_Name_XYZ", JDBCType.VARCHAR)
                                   .subkey(94002, JDBCType.NUMERIC)
                                   .build();
```

24.3.3 OracleConnectionBuilderインタフェース

OracleConnectionBuilderをユーザー名およびパスワード以外に追加パラメータともに使用して、接続オブジェクトを作成します。接続を作成するには、接続リクエストに含まれている必要があるパラメータごとにビルダー・メソッドをコールし、その後build()メソッドをコールする必要があります。ビルダー・メソッドをコールする順序は重要ではありません。ただし、同じビルダー属性を複数回適用する場合、最新の値のみが接続の作成時に考慮されます。ビルダーのbuild()メソッドをコールできるのは、ビルダー・オブジェクトで1回のみです。

構文

```
public interface OracleConnectionBuilder
```

例24-3 接続ビルダーの作成

```
...
OracleDataSource ods=new OracleDataSource();
...
OracleConnection conn = ods.createConnectionBuilder()
                           .shardingKey(shardingKey)
                           .superShardingKey(superShardingKey)
                           .build();
```

24.3.4 データベース・シャーディングのサポートのための他の新規クラスおよびメソッド

この項では、データベース・シャーディングのサポートの実装で導入されたその他の新規クラスおよびメソッドについて説明します。

OracleDataSourceクラスの新規メソッド

createConnectionBuilderおよびcreateShardingKeyBuliderメソッドが、データベース・シャーディングのサポート用にOracleDataSourceクラスに導入されました。

```
OracleConnectionBuilder createConnectionBuilder() throws SQLException;
OracleShardingKeyBuilder createShardingKeyBuilder()
```

OracleXADataSourceクラスの新規メソッド

createConnectionBuilderメソッドが、データベース・シャーディングのサポート用にOracleXADataSourceクラスに導入されました。

```
OracleConnectionBuilder createConnectionBuilder() throws SQLException;
```

OracleConnectionクラスの新規メソッド

setShardingKeyIfValidおよびsetShardingKeyメソッドが、データベース・シャーディングのサポート用にOracleConnectionクラスに導入されました。

```
boolean setShardingKeyIfValid(OracleShardingKey shardingKey, OracleShardingKey superShardingKey, int
timeout) throws SQLException;
void setShardingKey(OracleShardingKey shardingKey, OracleShardingKey superShardingKey) throws
SQLException;
```

OracleXAConnectionクラスの新規メソッド

setShardingKeyIfValidおよびsetShardingKeyメソッドが、データベース・シャーディングのサポート用にOracleConnectionクラスに導入されました。

```
boolean setShardingKeyIfValid(OracleShardingKey shardingKey, OracleShardingKey superShardingKey, int timeout) throws SQLException;
void setShardingKey(OracleShardingKey shardingKey, OracleShardingKey superShardingKey) throws SQLException;
```

24.4 JDBCシャーディングの例

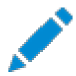
次のコードでは、JDBCシャーディングAPIの使用方法を示します。

例24-4 JDBCシャーディングの例

```
      OracleDataSource ods = new OracleDataSource();

ods.setURL("jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(HOST=myhost)(PORT=1521)(PROTOCOL=tcp))(CONNECT_DATA=(SERVICE_NAME=myorcldbservicename)))");
      ods.setUser("hr");
      ods.setPassword("hr");
     // Employee name is the sharding Key in this example.
     // Build the Sharding Key using employee name as shown below.
      OracleShardingKey employeeNameShardKey = ods.createShardingKeyBuilder()
                                                  .subkey("Mary", JDBCType.VARCHAR)// First Name
                                                  .subkey("Claire", JDBCType.VARCHAR)// Last Name
                                                  .build();
      OracleShardingKey locationSuperShardKey = ods.createShardingKeyBuilder() // Building a super
sharding key using location as the key
                                                   .subkey("US", JDBCType.VARCHAR)
                                                   .build();
      OracleConnection connection = ods.createConnectionBuilder()
                                       .shardingKey(employeeNameShardKey)
                                       .superShardingKey(locationSuperShardKey)
                                       .build();
```

25 Oracleアドバンスト・キューイング

Oracleアドバンスト・キューイング(AQ)は、データベース統合型のメッセージ・キューイング機能を提供します。Oracle Streams上で構築され、Oracle Databaseの機能を最適化して、メッセージを永続的に格納し、異なるコンピュータやデータベース上のキューにメッセージを伝播し、Oracle Netサービス、HTTPおよびHTTPSを使用してメッセージを送信できます。Oracle AQはデータベース表に実装されるため、運用上の利点である高可用性、スケーラビリティ、信頼性のすべてがキュー・データにも適用されます。この章では、Oracle AQへのJavaインタフェースに関する情報を提供します。



ノート:

- Oracle アドバンスト・キューイング(AQ)は、Oracle JDBC Thin ドライバの機能であり、JDBC OCI ドライバではサポートされていません。
- Oracle Database 12c リリース 1 (12.1)では、XMLType キューのサポートが追加されました。Oracle Database 11g リリース 1 までは、サポートされているキューの型は、RAW、ADT および ANYDATA 型でした。

関連項目:

Oracle Databaseアドバンスト・キューイング・ユーザーズ・ガイド

この章の内容は次のとおりです。

- Oracleアドバンスト・キューイングの機能とフレームワーク
- データベースの変更
- AQ非同期イベント通知
- メッセージの作成について
- メッセージのエンキュー
- メッセージのデキュー
- 例: エンキューとデキュー

25.1 Oracleアドバンスト・キューイングの機能とフレームワーク

Oracle JDBCパッケージであるoracle.jdbc.aqは、高速JavaインタフェースをAQに提供しています。このパッケージには、次のものが含まれています。

- クラス
    - AQDequeueOptions

      デキュー操作で利用可能なオプションを指定します。

- AQEnqueueOptions

  エンキュー操作で利用可能なオプションを指定します。

- AQFactory

  AQのファクトリ・クラスです。

- AQNotificationEvent

  通知を有効にするよう登録したキューに新しいメッセージがエンキューされると作成されます。

- インタフェース
  - AQAgent

    キューのユーザーや、メッセージのプロデューサまたはコンシューマを、表したり特定したりするために使用します。

  - AQMessage

    エンキューまたはデキュー対象のメッセージを表します。

  - AQMessageProperties

    相関、送信者、遅延、有効期限、受信者、優先度、順序付けなどのメッセージ・プロパティが含まれます。

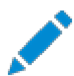
  - AQNotificationListener

    AQ通知イベントを受信するためのリスナー・インタフェースです。

  - AQNotificationRegistration

    特定のキューに新しいメッセージがエンキューされた場合に通知を受けることを表します。

これらのクラスとインタフェースを使用して、既存のキューにアクセスしたり、メッセージを作成したり、メッセージをエンキューおよびデキューできます。



ノート:

Oracle JDBC ドライバには、キューを作成するための API は一切用意されていません。キューは、DBMS_AQADM PL/SQL パッケージを使用して作成する必要があります。

関連項目:

APIの詳細は、Oracle Database JDBC Java APIリファレンスを参照してください。

25.2 データベースの変更

この章で使用されているコードの抜粋では、ユーザーHRがデータベースに接続しているものとします。そのため、データベース内でHRに次の権限を付与する必要があります。

```
GRANT EXECUTE ON DBMS_AQ to HR;
GRANT EXECUTE ON DBMS_AQADM to HR;
GRANT AQ_ADMINISTRATOR_ROLE TO HR;
GRANT ADMINISTER DATABASE TRIGGER TO HR;
```

メッセージをエンキューおよびデキューする前に、データベース内にキューが存在する必要があります。手順は次のとおりです。

1. 次のようにキュー表を作成します。

```
BEGIN
    DBMS_AQADM.CREATE_QUEUE_TABLE(
            QUEUE_TABLE =>'HR.RAW_SINGLE_QUEUE_TABLE',
            QUEUE_PAYLOAD_TYPE =>'RAW',
            COMPATIBLE => '10.0');
END;
```

2. 次のようにキューを作成します。

```
BEGIN
    DBMS_AQADM.CREATE_QUEUE(
            QUEUE_NAME =>'HR.RAW_SINGLE_QUEUE',
            QUEUE_TABLE =>'HR.RAW_SINGLE_QUEUE_TABLE',
END;
```

3. 次のようにキューを起動します。

```
BEGIN
    DBMS_AQADM.START_QUEUE(
 'HR.RAW_SINGLE_QUEUE',
END;
```

キューの停止やデータベースからのキュー表の削除が必要な場合があります。次の方法で対処できます。

1. 次のようにキューを停止します。

```
BEGIN
    DBMS_AQADM.STOP_QUEUE(
 HR.RAW_SINGLE_QUEUE',
END;
```

2. 次のようにデータベースからキュー表を削除します。

```
BEGIN
    DBMS_AQADM.DROP_QUEUE_TABLE(
            QUEUE_TABLE =>'HR.RAW_SINGLE_QUEUE_TABLE',
            FORCE => TRUE
END;
```

25.3 AQ非同期イベント通知

JDBCアプリケーションでは次のことができます。

- AQネームスペースを登録し、エンキュー発生時に通知を受け取ります。その方法は次のとおりです。

```
public AQNotificationRegistration registerForAQEvents(
    OracleConnection conn,
    String queueName) throws SQLException
  {
    Properties globalOptions = new Properties();
    String[] queueNameArr = new String[1];
    queueNameArr[0] = queueName;
    Properties[] opt = new Properties[1];
    opt[0] = new Properties();
    opt[0].setProperty(OracleConnection.NTF_AQ_PAYLOAD,"true");
```

```
    AQNotificationRegistration[] regArr =
conn.registerAQNotification(queueNameArr,opt,globalOptions);
    AQNotificationRegistration reg = regArr[0];
    return reg;
  }
```

- データベース・イベントにサブスクリプションを登録し、イベントのトリガー時に通知を受け取ります。

  登録されたクライアントは、イベントがトリガーされた場合または明示的なAQエンキューの場合に非同期に通知されます(または、通知希望を登録したキューに新規メッセージがエンキューされた場合)。クライアントは、データベースに接続する必要はありません。

  次のコードは、データベース・イベントにサブスクライブし、イベントのトリガー時に通知を受け取る方法を示します。

```
class DemoAQRawQueueListener implements AQNotificationListener
{
  OracleConnection conn;
  String queueName;
  String typeName;
  int eventsCount = 0;

  public DemoAQRawQueueListener(String _queueName, String _typeName)
   throws SQLException
  {
   queueName = _queueName;
   typeName = _typeName;
   conn = (OracleConnection)DriverManager.getConnection
     (DemoAQRawQueue.URL, DemoAQRawQueue.USERNAME, DemoAQRawQueue.PASSWORD);
  }

  public void onAQNotification(AQNotificationEvent e)
  {
    try
    {
     AQDequeueOptions deqopt = new AQDequeueOptions();
     deqopt.setRetrieveMessageId(true);
     if(e.getConsumerName() != null)
       deqopt.setConsumerName(e.getConsumerName());
     if((e.getMessageProperties()).getDeliveryMode()
        == AQMessageProperties.DeliveryMode.BUFFERED)
     {
       deqopt.setDeliveryMode(AQDequeueOptions.DEQUEUE_BUFFERED);
       deqopt.setVisibility(AQDequeueOptions.DEQUEUE_IMMEDIATE);
     }
     AQMessage msg = conn.dequeue(queueName,deqopt,typeName);
     byte[] msgId = msg.getMessageId();
     if(msgId != null)
     {
       String mesgIdStr = DemoAQRawQueue.byteBufferToHexString(msgId,20);
       System.out.println("ID of message dequeued = "+mesgIdStr);
     }
     System.out.println(msg.getMessageProperties().toString());
     byte[] payload = msg.getPayload();
     if(typeName.equals("RAW"))
     {
       String payloadStr = new String(payload,0,10);
       System.out.println("payload.length="+payload.length+", value="+payloadStr);
     }
```

```
    }
    catch(SQLException sqlex)
    {
     System.out.println(sqlex.getMessage());
    }
    eventsCount++;
  }
  public int getEventsCount()
  {
    return eventsCount;
  }
  public void closeConnection() throws SQLException
  {
    conn.close();
  }
}
```

- 次のようにリスナーに登録します。

```
AQNotificationRegistration reg = registerForAQEvents(conn,queueName+":BLUE");
DemoAQRawQueueListener demo_li = new DemoAQRawQueueListener(queueName,queueType);
reg.addListener(demo_li);
```

25.4 メッセージの作成について

この項では、次の概念について説明します。

- メッセージの作成
- AQメッセージのプロパティ
- AQメッセージのペイロード

25.4.1 メッセージの作成

メッセージをエンキューするには、まずそのメッセージを作成する必要があります。AQメッセージは、AQMessageインタフェースを実装したクラスのインスタンスによって表されます。各AQメッセージには、一連のプロパティ(メタデータ)と1つのペイロード(データ)が含まれます。AQメッセージを作成するには次の操作を実行します。

1. 次のようにAQMessagePropertiesのインスタンスを作成します。

```
AQMessageProperties msgprop = AQFactory.createAQMessageProperties();
```

2. 次のようにプロパティ属性を設定します。

```
msgprop.setCorrelation("mycorrelation");
msgprop.setExceptionQueue("MY_EXCEPTION_QUEUE");
msgprop.setExpiration(0);
msgprop.setPriority(1);
```

3. 次のようにAQMessagePropertiesオブジェクトを使用してAQメッセージを作成します。

```
AQMessage mesg = AQFactory.createAQMessage(msgprop);
```

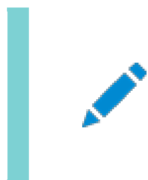
4. 次のようにペイロードを設定します。

```
byte[] rawPayload = "Example_Payload".getBytes();
mesg.setPayload(new oracle.sql.RAW(rawPayload));
```

25.4.2 AQメッセージのプロパティ

AQメッセージのプロパティは、AQMessagePropertiesインタフェースのインスタンスによって表されます。設定または取得できるメッセージ・プロパティは次のとおりです。

- デキュー試行回数: メッセージのデキューを試行した回数を示します。このプロパティは、設定できません。
- 相関: メッセージのエンキュー時に、そのメッセージのプロデューサによって提供されるIDです。
- 遅延: メッセージをいつまでWAITING状態にしておくかを示す秒数です。指定された遅延時間が経過すると、メッセージはREADY状態になり、デキューできるようになります。メッセージID(msgid)を使用してメッセージをデキューした場合、遅延時間の指定はオーバーライドされます。



ノート:

バッファ済メッセージでは delay はサポートされません。

- 配信モード: メッセージがバッファ・メッセージであるか永続メッセージであるかを示します。このプロパティは、設定できません。
- エンキュー時間: メッセージがエンキューされた時刻を示します。この値はシステムにより判断されるため、ユーザーは設定できません。
- 例外キュー: メッセージを正常に処理できない場合にメッセージの移動先となるキューの名前を指定します。メッセージが移動されるのは、次の2つの場合です。
    - デキューに失敗した回数がmax_retriesを超えた場合。
    - メッセージが期限切れになった場合。
- 有効期限: メッセージがREADY状態になってから、そのメッセージをデキューできなくなるまでの秒数。期限切れになる前にデキューされない場合、メッセージはEXPIRED状態で例外キューに移されます。
- メッセージ状態: メッセージがデキューされた時点のメッセージの状態を示します。このプロパティは、設定できません。
- 直前のキューにおけるメッセージID: 現在のメッセージを生成した最後のキューに入っているメッセージのIDです。メッセージがあるキューから別のキューに伝播される際に、この属性はメッセージの最後の伝播元であるキューのIDを識別します。このプロパティは、設定できません。
- 優先度: メッセージの優先順位を指定します。マイナスの整数を含む任意の整数です。小さい値ほど、高い優先度を表します。
- 受信者リスト: 受信者を表すAQAgentオブジェクトのリストです。デフォルトの受信者はキューのサブスクライバです。このパラメータは、複数コンシューマのキューに対してのみ有効です。
- 送信者: メッセージのエンキュー時にプロデューサによって指定される識別子です。これはAQAgentのインスタンスです。
- トランザクション・グループ: キューがトランザクション・グループ対応である場合に、メッセージのトランザクション・グループを示します。このプロパティは、dequeueArrayメソッドに対するコールが成功した後に設定されます。

25.4.3 AQメッセージのペイロード

AQメッセージのペイロードは、キューの型に応じ、AQMessageインタフェースのsetPayloadメソッドを使用して指定します。次のコードは、ペイロードの設定方法の例を示しています。

```
...
byte[] rawPayload = "Example_Payload".getBytes();
mesg.setPayload(new oracle.sql.RAW(rawPayload));
...
```

AQメッセージのペイロードを取得するには、次のようにgetPayloadメソッドまたは適切なgetXXXPayloadメソッドを使用します。

```
byte[] payload = mesg.getPayload();
```

これらのメソッドは、AQMessageインタフェースで定義されています。

25.5 例: メッセージの作成およびペイロードの設定

この項では、メッセージを作成しペイロードを設定する方法の例を示します。

例25-1 メッセージの作成およびペイロードの設定

この例では、AQMessagePropertiesのインスタンスの作成、プロパティ属性の設定、AQメッセージの作成およびペイロードの設定の方法を示します。

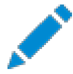
```
 AQMessageProperties msgprop = AQFactory.createAQMessageProperties();
    msgprop.setCorrelation("mycorrelation");
    msgprop.setExceptionQueue("MY_EXCEPTION_QUEUE");
    AQAgent ag = AQFactory.createAQAgent();
    ag.setName("MY_SENDER_AGENT_NAME");
    ag.setAddress("MY_SENDER_AGENT_ADDRESS");
    msgprop.setSender(ag);
    // handle multi consumer case:
    if(recipients != null)
      msgprop.setRecipientList(recipients);
    System.out.println(msgprop.toString());
    AQMessage mesg = AQFactory.createAQMessage(msgprop);
byte[] rawPayload = "Example_Payload".getBytes();
mesg.setPayload(new oracle.sql.RAW(rawPayload));
```

25.6 メッセージのエンキュー

メッセージを作成し、メッセージのプロパティとペイロードを設定したら、OracleConnectionインタフェースのenqueueメソッドを使用してメッセージをエンキューできます。メッセージをエンキューする前には、いくつかのエンキュー・オプションを指定できます。具体的には、AQEnqueueOptionsクラスを使用して、次のエンキュー・オプションを指定できます。

- 配信モード: 配信モードを指定します。配信モードは、永続(ENQUEUE_PERSISTENT)またはバッファ(ENQUEUE_BUFFERED)に設定できます。
- メッセージIDの取得: メッセージのエンキュー時にサーバーからメッセージIDを取得するかどうかを指定します。デフォルトでは、メッセージIDは取得されません。

● 変換: エンキューの前にメッセージに適用する変換を指定します。変換ファンクションの戻り型は、キューの型と一致する必要があります。



ノート:

変換は、DBMS_TRANSFORM.CREATE_TRANSFORMATION(...)を使用して、PL/SQL 内で作成する必要があります。

● 可視性: エンキュー・リクエストの、トランザクション上の動作を指定します。このオプションのデフォルト値はENQUEUE_ON_COMMITです。この設定値は、エンキュー操作が現在のトランザクションの一部であることを示します。ENQUEUE_IMMEDIATEを指定した場合、エンキュー操作は自律型トランザクションとなり、操作の完了時にコミットされます。バッファ・メッセージの場合は、ENQUEUE_IMMEDIATEを使用する必要があります。

次のコードは、エンキュー・オプションを設定してメッセージをエンキューする方法の例を示しています。

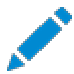
```
...
AQEnqueueOptions opt = new AQEnqueueOptions();opt.setRetrieveMessageId(true);
conn.enqueue(queueName, opt, mesg);
...
```

25.7 メッセージのデキュー

エンキューされたメッセージをデキューするには、OracleConnectionインタフェースのdequeueメソッドを使用します。メッセージをデキューする前には、デキュー・オプションを設定する必要があります。具体的には、AQDequeueOptionsクラスを使用して、次のデキュー・オプションを指定できます。

● 条件: メッセージ・プロパティ、メッセージ・データ・プロパティおよびPL/SQLファンクションに基づいて条件式を指定します。デキュー条件は、SQL問合せのWHERE句に似た構文を使用し、Boolean式として指定します。

● コンシューマ名: これを指定すると、そのコンシューマ名に一致するメッセージのみがアクセスされます。



ノート:

キューが単一コンシューマ・キューの場合は、このオプションを設定しないでください。

● 相関: デキュー操作に適用する相関基準(検索基準)を指定します。

● 配信フィルタ: デキューするメッセージの型を指定します。指定できるのは、バッファ・メッセージのみ(DEQUEUE_BUFFERED)、永続メッセージのみ(DEQUEUE_PERSISTENT、デフォルト)またはその両方(DEQUEUE_PERSISTENT_OR_BUFFERED)です。

● デキューするメッセージのID: デキューするメッセージのメッセージIDを指定します。このオプションは、IDがわかっている一意のメッセージをデキューする場合に使用できます。

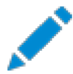
● デキュー・モード: デキュー操作に関連付けるロック動作を指定します。次のいずれかの値を指定できます。

  ● DequeueMode.BROWSE: ロックを取得せずにメッセージをデキューします。

  ● DequeueMode.LOCKED: トランザクションの完了まで持続する書込みロックをかけてメッセージをデキューします。

- DequeueMode.REMOVE: (デフォルト)メッセージをデキュー後に削除します。保持プロパティが永続であれば、メッセージはキュー内に保持されます。
- DequeueMode.REMOVE_NO_DATA: メッセージを、更新済または削除済としてマークします。

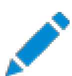
- 最大バッファ長: メッセージがRAWキューからデキューされる場合に、メッセージに割り当てることができる最大バイト数を指定します。デフォルトの最大値はDEFAULT_MAX_PAYLOAD_LENGTHですが、これはゼロを除く任意の他の値に変更できます。メッセージ全体を格納できるほどバッファが大きくない場合、超過したバイト数は警告なしに無視されます。
- ナビゲーション: どの位置のメッセージを取得対象とするかを指定します。次のいずれかの値を指定できます。
    - NavigationOption.FIRST_MESSAGE: 検索基準に一致するメッセージのうち、利用可能な最初のメッセージをデキューします。
    - NavigationOption.NEXT_MESSAGE: (デフォルト)検索基準に一致する、次のデキュー可能なメッセージをデキューします。前のメッセージがメッセージ・グループに属する場合は、そのメッセージ・グループに属するメッセージの中から、検索基準に一致する次に使用可能なメッセージがデキューされます。
    - NavigationOption.NEXT_TRANSACTION: 現在のトランザクション・グループ内のメッセージをスキップし、次のトランザクション・グループの最初のメッセージをデキューします。この設定を使用できるのは、キューでメッセージのグループ化が可能な場合のみです。
- メッセージIDの取得: デキューするメッセージのメッセージIDを取得する必要があるかどうかを指定します。デフォルトでは取得されません。
- 変換: デキューの後にメッセージに適用する変換を指定します。変換のソース・タイプは、キューのタイプと一致させる必要があります。



ノート:

変換は、DBMS_TRANSFORM.CREATE_TRANSFORMATION(...)を使用して、PL/SQL 内で作成する必要があります。

- 可視性: メッセージを、現在のトランザクションの一部としてデキューするかどうかを指定します。次のいずれかの値を指定できます。
    - VisibilityOption.ON_COMMIT: (デフォルト)現在のトランザクションの一部としてデキューします。
    - VisibilityOption.IMMEDIATE: 操作の完了時にコミットされる自律型トランザクションとしてデキューします。

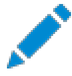


ノート:

デキュー・モードが DequeueMode.BROWSE の場合、可視性オプションは無視されます。配信フィルタが DEQUEUE_BUFFERED または DEQUEUE_PERSISTENT_OR_BUFFERED である場合、このオプションは VisibilityOption.IMMEDIATE に設定する必要があります。

- 待機: 検索基準に一致するメッセージが1つもない場合にデキュー操作に適用する待機時間を指定します。デフォルト値のDEQUEUE_WAIT_FOREVERは、デキュー操作を無期限に待機状態にすることを示します。DEQUEUE_NO_WAITに設定した場合、デキュー操作は待機状態になりません。数値を指定した場合、デキュー操作は指定された秒数の間待機状態になります。

ノート:



DEQUEUE_WAIT_FOREVER を指定すると、デキュー操作は検索基準に一致するメッセージがキュー内でデキュー可能になるまで待機状態が続きます。ただし、OracleConnection オブジェクトに対して cancel メソッドをコールすると、デキュー操作を中断できます。

次のコードは、デキュー・オプションを設定してメッセージをデキューする方法の例を示しています。

```
...
AQDequeueOptions deqopt = new AQDequeueOptions();
deqopt.setRetrieveMessageId(true);
deqopt.setConsumerName(consumerName);
AQMessage msg = conn.dequeue(queueName,deqopt,queueType);
```

25.8 例: エンキューとデキュー

この項では、メッセージのエンキュー方法とデキュー方法を、いくつかの例で示します。

例25-2ではメッセージのエンキュー方法を、例25-3ではメッセージのデキュー方法を示しています。

例25-2 単一メッセージのエンキュー

この例では、キューにアクセスし、メッセージを作成して、メッセージをエンキューする方法を示しています。

```
AQMessageProperties msgprop = AQFactory.createAQMessageProperties();
msgprop.setPriority(1);
msgprop.setExceptionQueue("EXCEPTION_QUEUE");
msgprop.setExpiration(0);
AQAgent agent = AQFactory.createAQAgent();
agent.setName("AGENTNAME");
agent.setAddress("AGENTADDRESS");
msgprop.setSender(agent);
AQMessage mesg = AQFactory.createAQMessage(msgprop);
mesg.setPayload(buffer); // where buffer is a byte array (for a RAW queue)
AQEnqueueOptions options = new AQEnqueueOptions();
conn.enqueue("HR.MY_QUEUE", options, mesg);
```

例25-3 単一メッセージのデキュー

この例では、キューにアクセスし、デキュー・オプションを設定して、メッセージをデキューする方法を示しています。

```
AQDequeueOptions options = new AQDequeueOptions();
options.setDeliveryFilter(AQDequeueOptions.DeliveryFilter.BUFFERED);
AQMessage mesg = conn.dequeue("HR.MY_QUEUE", options, "RAW");
```

26 連続問合せ通知

この章では、連続問合せ通知機能の仕組みについて説明します。

この章の内容は次のとおりです。

- 連続問合せ通知の概要
- クライアント開始の連続問合せ通知の概要
- 登録エントリの作成
- 登録エントリへの問合せの関連付け
- データベース変更イベントの通知
- 登録の削除

26.1 連続問合せ通知の概要

一般に、中間層データ・キャッシュは、バックエンド・データベース・サーバーの一部のデータを複製します。その目的は、データベースに対する冗長な問合せを回避することです。ただし、これが効率的なのは、データベース内でのデータの変更頻度が非常に低い場合に限られます。データベース内でデータが変更された際には、データ・キャッシュを更新するか、無効にする必要があります。11gリリース1以降、Oracle JDBCドライバは、Oracle Databaseの連続問合せ通知機能をサポートしています。この機能を使用すると、JDBCドライバから無効化イベントを受信することにより、多重化システムのデータ・キャッシュを可能なかぎり最新の状態に保つことができます。

JDBCドライバでは、SQL問合せをデータベースに登録して、次のイベントの発生時に通知を受け取ることができます。

- 問合せに関連付けられたオブジェクトに対するDMLまたはDDL変更。
- 結果セットに影響を与えるDMLまたはDDL変更。

通知は、DMLまたはDDLトランザクションのコミット時にパブリッシュされます(ローカル・トランザクションで行われた変更は、コミットされるまでイベントを生成しません)。

Oracle JDBCドライバの連続問合せ通知機能は、次のような流れで使用します。

1. 登録: まず登録エントリを作成します。
2. 問合せの関連付け: 登録エントリを作成したら、その登録エントリにSQL問合せを関連付けできます。これらの問合せは、登録エントリの一部となります。
3. 通知: 表または結果セットに変更が生じると、通知が作成されます。Oracle Databaseは専用のネットワーク接続を使用してこれらの通知をJDBCドライバに伝え、JDBCドライバはそれらの通知をJavaイベントに変換します。

これらに加え、ユーザーにCHANGE NOTIFICATION権限を付与することも必要です。たとえば、HRというユーザー名を使用してデータベースに接続する場合は、データベース内で次のコマンドを実行する必要があります。

```
grant change notification to HR;
```

26.2クライアント開始の連続問合せ通知の概要

Oracle Databaseリリース19c以降、JDBC Thinドライバではクライアント開始の連続問合せ通知機能をサポートしていま

す。この場合、クライアント・アプリケーションは、通知を受信するためにデータベース・サーバーへの接続を開始します。

まず、クライアント・アプリケーションは連続問合せ通知登録エントリを作成する前に、新しいデータベース接続を開始します。アプリケーションが登録エントリを作成すると、JDBCドライバは内部的に新しいスレッドを開始し、データベース・サーバーとの新しい接続を作成します。次に、データベース・サーバーはこの新しい接続を使用して、変更通知をクライアントに送信します。

デフォルトでは、この機能はオンプレミス・データベースに対して無効になっています。この機能を有効にするには、`OracleConnection.DCN_CLIENT_INIT_CONNECTION`をtrueに設定する必要があります。

関連項目:

連続問合せ通知登録オプション

26.3 登録エントリの作成

CQN登録エントリの作成は、1回かぎりのプロセスであり、現在使用中のトランザクションの外部で実行します。サーバーに登録エントリを作成するためのAPIは、専用のトランザクション内で実行され、即時にコミットされます。

登録エントリを作成するにはJDBC接続が必要です。ただし、登録エントリは接続にアタッチされません。登録エントリの作成後に接続をクローズしても、登録エントリ自体は有効な状態を維持します。Oracle RAC環境においては、登録エントリはすべてのノードに存在する永続的エンティティとなります。登録エントリはデータベース内に存在しています。そのため、ノードがダウンした場合でも、登録エントリは存在し続け、表の変更時には通知を受けます。

登録エントリの作成方法は、次の2種類に分けられます。

- JDBCスタイルの登録: JDBCドライバを使用し、サーバー上に登録エントリを作成します。JDBCドライバは、サーバーからの通知を(専用チャネルを介して)リスニングする新しいスレッドを起動し、それらの通知メッセージをJavaイベントに変換します。その後、ドライバは作成したエントリに登録されているすべてのリスナーに通知を送ります。
- PL/SQLスタイルの登録: 通知の処理にPL/SQLのストアド・プロシージャを使用する場合は、PL/SQLスタイルの登録エントリを作成します。JDBCスタイルの登録の場合と同様に、登録エントリに文(問合せ)をアタッチするにはJDBCドライバを使用します。ただし、通知はPL/SQLのストアド・プロシージャによって処理されるため、JDBCドライバはサーバーからの通知を受け取りません。

ノート:

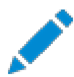
この方法は、PHP などマルチスレッド以外の言語の場合にのみ、便利です。

既存の登録エントリから特定の(1つの)オブジェクト(表)を削除する方法はありません。次善策としては、そのオブジェクトを含まない登録エントリを新規作成するか、そのオブジェクトに関連付けられているイベントを無視します。

JDBCスタイルの登録エントリの作成には、`oracle.jdbc.OracleConnection`インタフェースの`registerDatabaseChangeNotification`メソッドを使用できます。このメソッドでは、`options`パラメータにより、特定の登録オプションを設定できます。次の項の「連続問合せ通知登録オプション」表には、設定できる登録オプションの一部を示します。これらのオプションは、`java.util.Properties`オブジェクトを使用して設定します。これらのオプションは、

oracle.jdbc.OracleConnectionインタフェースで定義されています。これらの登録オプションは、JDBCドライバによって作成される通知イベントに直接影響を与えます。連続問合せ通知機能の使用方法は、(この章の終わりの)例に示します。

registerDatabaseChangeNotificationメソッドは、指定されたオプションを反映した、新しいデータベース変更登録エントリをデータベース・サーバー内に作成します。このメソッドはDatabaseChangeRegistrationオブジェクトを返し、そのオブジェクトを使用して登録エントリに文が関連付けられます。このメソッドはさらに、通知を送信するためにデータベースによって使用されるリスナー・ソケットをオープンします。



ノート:

別の登録エントリによって作成されたリスナー・ソケットがすでに存在している場合は、新しいデータベース変更登録エントリもそのソケットを使用します。

26.3.1 連続問合せ通知登録オプション

次の表では、連続問合せ通知登録オプションを示します。

表26-1 連続問合せ通知登録オプション

オプション	説明
DCN_IGNORE_DELETEOP	true に設定された場合、DELETE 操作が実行されてもデータベース変更イベントは生成されません。
DCN_IGNORE_INSERTOP	true に設定された場合、INSERT 操作が実行されてもデータベース変更イベントは生成されません。
DCN_IGNORE_UPDATEOP	true に設定された場合、UPDATE 操作が実行されてもデータベース変更イベントは生成されません。
DCN_NOTIFY_CHANGELAG	クライアントにトランザクションいくつ分までの遅延を許可するかを指定します。 ノート: このオプションを 0 以外の値に設定した場合、DCN_NOTIFY_ROWIDS オプションを true に設定しても、ROWID レベルの粒度の情報は一連のイベントで利用できなくなります。
DCN_NOTIFY_ROWIDS	データベース変更イベントに、操作タイプや ROWID など、行レベルの詳細情報を含めます。
DCN_QUERY_CHANGE_NOTIFICATION	オブジェクト変更通知のかわりに、問合せ変更通知をアクティブにします。 ノート: このオプションは、11.0 データベースに対して実行する場合にのみ利用できます。

オプション	説明
DCN_CLIENT_INIT_CONNECTION	クライアント開始の連続問合せ通知を指定します。これは、クライアント・アプリケーションがデータベース接続を開始し、その接続をサーバーが使用して変更通知をクライアントに送信します。
NTF_LOCAL_HOST	サーバーからの通知を受信するコンピュータの IP アドレスを指定します。
NTF_LOCAL_TCP_PORT	リスナー・ソケット用としてドライバに使用させる TCP ポートを指定します。
NTF_QOS_PURGE_ON_NTFN	最初の通知イベント発生時に登録エントリを消去するかどうかを指定します。
NTF_QOS_RELIABLE	通知を永続的に維持するかどうかを指定します(永続的に維持する場合、パフォーマンス・コストが高くなります)。
NTF_TIMEOUT	このオプションで秒数を指定すると、登録エントリは指定した秒数の経過後にデータベースによって自動的に消去されます。

すでに登録エントリが存在している場合は、getDatabaseChangeRegistrationメソッドを使用して、既存の登録エントリを新しいDatabaseChangeRegistrationオブジェクトにマップすることもできます。このメソッドは、PL/SQLを使用した登録エントリをすでに作成してあり、それに文を関連付ける場合に特に便利です。

26.4 登録エントリへの問合せの関連付け

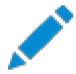
登録エントリの作成、または既存の登録エントリに対するマップが終わったら、登録エントリに問合せを関連付けることができます。登録エントリの作成と同様、登録エントリと問合せの関連付けは1回かぎりのプロセスであり、現在使用中の登録の外部で実行します。問合せの関連付けは、ローカル・トランザクションがロールバックされても実行されます。

登録エントリと問合せの関連付けは、OracleStatementクラスで定義されているsetDatabaseChangeRegistrationメソッドを使用して実行します。このメソッドは、DatabaseChangeRegistrationオブジェクトをパラメータとして取ります。次のコードは、登録エントリに問合せを関連付ける方法の例を示しています。

```
...
// conn is an OracleConnection object.
// prop is a Properties object containing the registration options.
DatabaseChangeRegistration dcr = conn.registerDatabaseChangeNotifictaion(prop);
...
Statement stmt = conn.createStatement();
// associating the query with the registration
((OracleStatement)stmt).setDatabaseChangeRegistration(dcr);
// any query that will be executed with the 'stmt' object will be associated with
// the registration 'dcr' until 'stmt' is closed or
// '((OracleStatement)stmt).setDatabaseChangeRegistration(null);' is executed.
...
```

26.5 データベース変更イベントの通知

連続問合せ通知を受け取るには、登録エントリにリスナーをアタッチします。データベース変更イベントが発生すると、データベース・サーバーはJDBCドライバに通知します。ドライバは、新しいJavaイベントを作成し、通知対象の登録エントリを特定した後、その登録エントリにアタッチされているリスナーに通知を送ります。イベントには、変更されたデータベース・オブジェクトのオブジェクトIDと、変更の原因となった操作のタイプが含まれます。登録オプションによっては、行レベルの詳細情報も含まれます。その後、そのイベントを使用して、データ・キャッシュに関する判断を行うリスナー・コードを実行することもできます。



ノート:

リスナー・コードは、JDBC 通知メカニズムの処理速度を低下させないように作成する必要があります。データベースへの問合せによってデータ・キャッシュをリフレッシュするなど、リスナー・コードの処理時間が長い場合は、専用のスレッド内で実行する必要があります。

登録エントリにリスナーをアタッチするには、addListenerメソッドを使用します。次のコードは、登録エントリにリスナーをアタッチする方法の例を示しています。

```
...
// conn is an OracleConnection object.
// prop is a Properties object containing the registration options.
DatabaseChangeRegistration dcr = conn.registerDatabaseChangeNotifictaion(prop);
...
// Attach the listener to the registration.
// Note: DCNListener is a custom listener and not a predefined or standard
// lsiener
DCNListener list = new DCNListener();
dcr.addListener(list);
...
```

26.6 登録エントリの削除

登録エントリをサーバーから削除し、ドライバ内のリソースをリリースするには、登録エントリを明示的に解除する必要があります。登録の解除には、作成時に使用したのとは別の接続を使用できます。登録の解除には、oracle.jdbc.OracleConnectionで定義されているunregisterDatabaseChangeNotificationメソッドを使用します。

このメソッドには、パラメータとしてDatabaseChangeRegistrationオブジェクトを渡す必要があります。このメソッドは、サーバーおよびドライバから登録を削除し、リスナー・ソケットをクローズします。

PL/SQLを使用するなどして、登録エントリをJDBCの外部で作成した場合は、DatabaseChangeRegistrationオブジェクトのかわりに登録IDを渡す必要があります。このメソッドは、サーバーから登録を削除しますが、ドライバ内のリソースは解放しません。

連続問合せ通知機能の使用方法は、例26-1に示しています。この例では、ユーザーHRがデータベースに接続しています。そのため、データベース内でこのユーザーに次の権限を付与する必要があります。

```
grant change notification to HR;
```

このコードは、Oracle Database 10gリリース2(10.2)でも動作します。このコードでは、表登録を使用しています。この場合、

SELECT問合せを登録しても、登録されるのは問合せ自体ではなく、それに関与する表の名前です。つまり、表内の1行を選択した場合、それとは別の行が更新されていて、問合せの結果に影響がなくても、通知を受けることになります。

このコード例では、登録エントリをクローズせずにオープン状態のままにしておくと、連続問合せ通知スレッドは動作を継続するようになっています。したがって、HR.DEPARTMENTS表を変更するDML問合せを(SQL*Plusなどから)起動およびコミットすると、Javaプログラムによって通知が出力されます。

例26-1 連続問合せ通知

```
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.OracleDriver;
import oracle.jdbc.OracleStatement;
import oracle.jdbc.dcn.DatabaseChangeEvent;
import oracle.jdbc.dcn.DatabaseChangeListener;
import oracle.jdbc.dcn.DatabaseChangeRegistration;

public class DBChangeNotification
{
  static final String USERNAME= "HR";
  static final String PASSWORD= "hr";
  static String URL;

  public static void main(String[] argv)
  {
    if(argv.length < 1)
    {
      System.out.println("Error: You need to provide the URL in the first argument.");
      System.out.println("  For example: > java -classpath .:ojdbc6.jar DBChangeNotification
¥"jdbc:oracle:thin:
@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=yourhost.yourdomain.com)(PORT=5221))(CONNECT_DATA=
(SERVICE_NAME=orcl)))¥"");

      System.exit(1);
    }
    URL = argv[0];
    DBChangeNotification demo = new DBChangeNotification();
    try
    {
      demo.run();
    }
    catch(SQLException mainSQLException )
    {
      mainSQLException.printStackTrace();
    }
  }

  void run() throws SQLException
  {
    OracleConnection conn = connect();

    // first step: create a registration on the server:
    Properties prop = new Properties();
```

```
    // if connected through the VPN, you need to provide the TCP address of the client.
    // For example:
    // prop.setProperty(OracleConnection.NTF_LOCAL_HOST,"14.14.13.12");

    // Ask the server to send the ROWIDs as part of the DCN events (small performance
    // cost):
    prop.setProperty(OracleConnection.DCN_NOTIFY_ROWIDS,"true");
//
//Set the DCN_QUERY_CHANGE_NOTIFICATION option for query registration with finer granularity.
 prop.setProperty(OracleConnection.DCN_QUERY_CHANGE_NOTIFICATION,"true");

    // The following operation does a roundtrip to the database to create a new
    // registration for DCN. It sends the client address (ip address and port) that
    // the server will use to connect to the client and send the notification
    // when necessary. Note that for now the registration is empty (we haven't registered
    // any table). This also opens a new thread in the drivers. This thread will be
    // dedicated to DCN (accept connection to the server and dispatch the events to
    // the listeners).
    DatabaseChangeRegistration dcr = conn.registerDatabaseChangeNotification(prop);

    try
    {
      // add the listenerr:
      DCNDemoListener list = new DCNDemoListener(this);
      dcr.addListener(list);

      // second step: add objects in the registration:
      Statement stmt = conn.createStatement();
      // associate the statement with the registration:
      ((OracleStatement)stmt).setDatabaseChangeRegistration(dcr);
      ResultSet rs = stmt.executeQuery("select * from dept where deptno='45'");
      while (rs.next())
      {}
      String[] tableNames = dcr.getTables();
      for(int i=0;i<tableNames.length;i++)
        System.out.println(tableNames[i]+" is part of the registration.");
      rs.close();
      stmt.close();
    }
    catch(SQLException ex)
    {
      // if an exception occurs, we need to close the registration in order
      // to interrupt the thread otherwise it will be hanging around.
      if(conn != null)
        conn.unregisterDatabaseChangeNotification(dcr);
      throw ex;
    }
    finally
    {
      try
      {
        // Note that we close the connection!
        conn.close();
      }
      catch(Exception innerex){ innerex.printStackTrace(); }
    }
```

```
    synchronized( this )
    {
      // The following code modifies the dept table and commits:
      try
      {
        OracleConnection conn2 = connect();
        conn2.setAutoCommit(false);
        Statement stmt2 = conn2.createStatement();
        stmt2.executeUpdate("insert into dept (deptno,dname) values ('45','cool dept')",
Statement.RETURN_GENERATED_KEYS);
        ResultSet autoGeneratedKey = stmt2.getGeneratedKeys();
        if(autoGeneratedKey.next())
          System.out.println("inserted one row with ROWID="+autoGeneratedKey.getString(1));
        stmt2.executeUpdate("insert into dept (deptno,dname) values ('50','fun dept')",
Statement.RETURN_GENERATED_KEYS);
        autoGeneratedKey = stmt2.getGeneratedKeys();
        if(autoGeneratedKey.next())
          System.out.println("inserted one row with ROWID="+autoGeneratedKey.getString(1));
        stmt2.close();
        conn2.commit();
        conn2.close();
      }
      catch(SQLException ex) { ex.printStackTrace(); }

      // wait until we get the event
      try{ this.wait();} catch( InterruptedException ie ) {}
    }

    // At the end: close the registration (comment out these 3 lines in order
    // to leave the registration open).
    OracleConnection conn3 = connect();
    conn3.unregisterDatabaseChangeNotification(dcr);
    conn3.close();
  }

  /**
   * Creates a connection the database.
   */
  OracleConnection connect() throws SQLException
  {
    OracleDriver dr = new OracleDriver();
    Properties prop = new Properties();
    prop.setProperty("user",DBChangeNotification.USERNAME);
    prop.setProperty("password",DBChangeNotification.PASSWORD);
    return (OracleConnection)dr.connect(DBChangeNotification.URL,prop);
  }
}
/**
 * DCN listener: it prints out the event details in stdout.
 */
class DCNDemoListener implements DatabaseChangeListener
{
  DBChangeNotification demo;
  DCNDemoListener(DBChangeNotification dem)
  {
    demo = dem;
  }
  public void onDatabaseChangeNotification(DatabaseChangeEvent e)
```

```
  {
    Thread t = Thread.currentThread();
    System.out.println("DCNDemoListener: got an event ("+this+" running on thread "+t+")");
    System.out.println(e.toString());
    synchronized( demo ){ demo.notify();}
  }
}
```

第VI部 高可用性

この部では、Oracle Database 12cリリース2 (12.2.0.1)の高可用性機能について説明します。

第VI部は、次の章で構成されます。

- Java用のトランザクション・ガード
- Java用のアプリケーション・コンティニュイティ
- Oracle JDBCによるFANイベントのサポート
- 透過的アプリケーション・フェイルオーバー
- 単一クライアント・アクセス名

27 Java用のトランザクション・ガード

Oracle Database 12cリリース1 (12.1)では、トランザクション・ガード機能を導入し、計画済および計画外の停止と重複発行の際に最大1回の実行のための汎用インフラストラクチャを提供します。この章では、次の項で、Java用のトランザクション・ガードについて説明します。

- Java用のトランザクション・ガードの概要
- XAトランザクションのためのトランザクション・ガードのサポート
- Java API用のトランザクション・ガード
- 完全な例: トランザクション・ガードAPIの使用
- サーバー側トランザクション・ガードAPIの使用について

27.1 Java用のトランザクション・ガードの概要

現在のアプリケーションでは、保証されていて拡張性のある方法で最後のコミット操作の結果を判断する方法やサーバーとの通信エラーの後の処理は未解決の問題です。多くの場合、エンドユーザーは、重複したリクエストの再発行を回避するために特定のステップに従うように要求されます。たとえば、一部のアプリケーションでは、ユーザーに送信ボタンを2回クリックしないように警告します。これは、2回クリックした場合、ユーザーが意図せずに、同じものを2回購入し、2回分の支払いを行うことがあるためです。

この問題を解決するため、Java用のトランザクション・ガードではトランザクションが重複発行されても1件のみが有効になります。つまり、すべてのトランザクションの実行は最大1回であるため、アプリケーションは重複するトランザクションを送信できません。すべてのトランザクションは論理トランザクションID(LTXID)でタグ付けされており、これは、トランザクションが障害の前にコミットしたかどうかを確認するために、障害の発生後に、アプリケーションによって使用されます。たとえば、コミット・コールが戻らない場合、アプリケーションはLTXIDを使用して、コールが正常終了したかどうかを検出できます。

関連項目:

『Oracle Database開発ガイド』

Java用のアプリケーション・コンティニュイティ機能はJava用のトランザクション・ガードを内部で使用しているため、透過的なセッション・リカバリと、処理中のトランザクションの初めからSQL文(問合せおよびDML)のリプレイが可能になります。アプリケーション・コンティニュイティによって計画済または計画外の停止発生後に作業のリカバリが可能になり、Java用のトランザクション・ガードによってトランザクションは重複発行されても必ず1件のみが有効になります。停止が発生すると、リカバリでは、障害が発生する前と同じ状態がリストアされます。

関連トピック

- Java用のアプリケーション・コンティニュイティ

27.2 XAトランザクションのためのトランザクション・ガードのサポート

Oracle Database 12cリリース2 (12.2.0.1)以降、トランザクション・ガードでは、1フェーズ・コミットの最適化、読取り専用最適化および昇格可能なXAのためのXAトランザクションがサポートされています。XAを使用したトランザクション・ガードでは、XAトランザクションでのリカバリ可能な停止に続く安全なリプレイを提供します。XAサポートを追加すると、トランザクション・マネージャは、トランザクション・ガードによる冪等性を備えたリプレイを簡単に行えるようになります。

ノート:

接続プールからセッションをチェックアウトする際に XA を使用したトランザクション・ガードを使用するには、データベースのバージョンが Oracle 12c リリース 2 (12.2.0.1)であり、トランザクション・ガードが有効になっていることを確認する必要があります。

新しいサーバー・プロトコルでは、コミットがデータベースによって1フェーズ管理されている場合にコミット結果が保証され、トランザクション・マネージャがそのセッションのトランザクションを協調させたときに、無効化されたモードに切り替えます。新しいプロトコルでは、LTXIDにステータス・フラグを設定し、このフラグにより、現在のトランザクション所有者に基づいてLTXIDへのアクセスが有効化および無効化されます。

このプロトコルは、インテリジェントに処理を行い、XAは1つのXAトランザクションに対して多数のセッションおよびブランチを網羅できます。さらに、1つのブランチが保留されると、元のトランザクションはアクティブなまま、1つのセッションが別のトランザクションに対して使用可能になります。元のブランチを作成した同じセッションまたはRACインスタンスでXAトランザクションを準備またはコミットするための要件はありません。XA用のトランザクション・ガードでは、1フェーズXAトランザクションのデータベースでコミット結果を処理するための次の2つの新しい方法を使用し、一方で、トランザクション・マネージャは2フェーズ・トランザクションのコミット結果の処理を続行します。

- 最初の方法を使用すると、リカバリ可能なエラーまたは他の指定された条件がそのセッションで発生するまで、ドライバはLTXIDを暫定とマークします。リカバリ可能なエラー(または他の条件)が発生すると、クライアントのLTXIDは最終とマークされます。保証されたコミット結果が提供されるのは、LTXIDがクライアントで最終の場合、およびLTXIDのステータスがVALIDである(データベースがそのトランザクションを所有していることを示す)サーバーで最終の場合のみです。他のアクセス試行はエラーが戻されます。
- 2番目の方法を使用すると、リカバリ可能なエラーまたは他の指定した条件がそのセッションで発生するまで、クライアント・ドライバはLTXIDをアプリケーションに提供しません。

27.3 XAによるトランザクション・ガードの使用方法

この項では、次の項について説明します。

昇格可能なXAを使用したコミット結果の取得

ローカル・トランザクションの場合、リカバリ可能な例外があると、リクエストではLTXIDをトランザクション・キーとして取得します。2番目のブランチが開始されると、リクエストはXAに昇格されるか、XAに変換され、グローバル・トランザクション識別子(GTRID)がこれに割り当てられます。コミットの処理中にリカバリ可能な停止が発生し、アプリケーションがトランザクション・マネージャから応答を受信しない場合、アプリケーションはトランザクション・マネージャに結果を要求します。データベースへのほとん

どのリクエストでは、ローカル・トランザクションまたは単一ブランチの最適化のいずれかを使用します。ローカル・トランザクションまたは昇格可能なXAを使用している場合、XAに対するラウンドトリップおよび管理でオーバーヘッドはありません。これは、ほとんどのトランザクションがローカルであるためです。これらのトランザクションのワークフローは次のとおりです。

1. XAに変換する前は、トランザクション処理はローカルです。認証、SELECT文およびローカル・トランザクションが実行されると、ローカル・トランザクションのLTXIDが使用されます。
2. トランザクション・マネージャがGTRIDをトランザクションに割り当てるのは、トランザクションの2番目のブランチをオープンしたためにXAの使用を開始した場合のみです。
3. リカバリ可能なエラーの後に、アプリケーションでコミット結果を受信しない場合、トランザクションがローカルであれば、トランザクション・マネージャはLTXIDをGET_LTXID_OUTCOMEプロシージャとともに使用してコミット結果を取得し、アプリケーションにCOMMITTEDまたはUNCOMMITTEDの結果を戻します。

昇格可能なXAが追加された場合のリプレイ

昇格される前に、昇格可能なXAでは、静的XAでサポートされていないコールと設定を使用してRDBMSコミットをサポートしています。これらのコールには、自動コミット・モード、PL/SQLに組み込まれたDDL、DCL、COMMIT、およびリモート・プロシージャ・コールを使用したCOMMITが含まれています。これらのユーザー・コールおよびモードに対するCOMMIT結果はRDBMSで制御され、エラーの後、トランザクション・ガードを使用してコミット結果を確認できます。

昇格されるまで、トランザクション・マネージャではリクエストがCOMMITで発行されたかどうかを認識していません。トランザクション・マネージャはリカバリ可能なエラーの後でリクエストをリプレイする場合に、RDBMS COMMITが発生したかどうかを判定する必要があります。RDBMS COMMITが発生したか発生する可能性がある場合、リプレイは発生しません。GET_LTXID_OUTCOMEプロシージャでは現在のトランザクション結果の報告しか行われないため、これを判定する際には不十分です。LTXIDが変更されると、トランザクションがコミットされます。したがって、LTXIDのコールバックの起動により、トランザクションがコミットされたことが示されます。

27.4 Java API用のトランザクション・ガード

この項では、次のアクティビティのためにJava用のトランザクション・ガードに関連付けられたAPIについて説明します。

- 論理トランザクションIDの取得
- 更新済論理トランザクションIDの取得

27.4.1 論理トランザクションIDの取得

サーバーによって送信される現在の論理トランザクションIDを取得するには、oracle.jdbc.OracleConnectionインタフェースのgetLogicalTransactionIdメソッドを使用します。このメソッドをコールしても、データベースのラウンドトリップは発生しません。

例

```
OracleConnection oconn = (OracleConnection) ods.getConnection();
...
// Getting the 1st LTXID after connecting
LogicalTransactionId firstLtxid = oconn.getLogicalTransactionId();
```

27.4.2 更新済論理トランザクションIDの取得

論理トランザクションIDに対する更新を受信するには、oracle.jdbc.LogicalTransactionIdEventListenerインタフェースを使用します。論理トランザクションIDイベントを処理するために、このインタフェースをアプリケーションに実装する必要があります。

27.4.2.1 イベント・リスナーの登録

リスナーを論理トランザクションIDイベントに登録するには、addLogicalTransactionIdEventListenerメソッドを使用します。

例

```
OracleConnection oconn = (OracleConnection) ods.getConnection();
...
// The subsequent LTXID updates can be obtained through the listener
oconn.addLogicalTransactionIdEventListener(this);
```

addLogicalTransactionIdEventListener(LogicalTransactionIdEventListener listener, java.util.concurrent.Executor executor)メソッドを使用して、リスナーをエグゼキュータに登録することもできます。

27.4.2.2 イベント・リスナーの登録解除

リスナーを論理トランザクションIDイベントから登録解除するには、removeLogicalTransactionIdEventListenerメソッドを使用します。

例

```
OracleConnection oconn = (OracleConnection) ods.getConnection();
...
// The subsequent LTXID updates can be obtained through the listener
oconn.removeLogicalTransactionIdEventListener(this);
```

27.5 完全な例: トランザクション・ガードAPIの使用

次に、トランザクション・ガードAPIを使用する完全な例を示します。

```
import oracle.jdbc.pool.OracleDataSource;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.LogicalTransactionId;
import oracle.jdbc.LogicalTransactionIdEvent;
import oracle.jdbc.LogicalTransactionIdEventListener;

public class transactionGuardExample
{
    ...
    ...
    OracleDataSource ods = new OracleDataSource();
    ods.setURL(url);
    ods.setUser("user");
    ods.setPassword("password");

    OracleConnection oconn = (OracleConnection) ods.getConnection();

    // Getting the 1st LTXID after connecting
    LogicalTransactionId firstLtxid = oconn.getLogicalTransactionId();
```

```
            // The subsequent LTXID updates can be obtained via the listener
            oconn.addLogicalTransactionIdEventListener(this);
        }

      public class LtxidListenerImpl
        implements LogicalTransactionIdEventListener
      {
        ...

        public void onLogicalTransactionIdEvent(LogicalTransactionIdEvent ltxidEvent)
        {
          LogicalTransactionId newLtxid = ltxidEvent.getLogicalTransactionId();
          // process newLtxid ......
        }
      }
```

27.6 サーバー側トランザクション・ガードAPIの使用について

DBMS_APP_CONTパッケージには、サーバー側トランザクション・ガードAPIが含まれているGET_LTXID_OUTCOMEプロシージャが含まれています。このプロシージャを実行すると、トランザクションの結果が出力されます。トランザクションがコミットされていない場合、偽のトランザクションがコミットされます。そうでない場合、トランザクションの状態が戻されます。デフォルトでは、このパッケージのEXECUTE権限がデータベース管理者に付与されています。

構文

```
  PROCEDURE GET_LTXID_OUTCOME(CLIENT_LTXID        IN  RAW,
                              committed           OUT BOOLEAN,
                              USER_CALL_COMPLETED OUT BOOLEAN);
```

入力パラメータ

CLIENT_LTXIDは、クライアント・ドライバのLTXIDを指定します。

出力パラメータ

COMMITTEDは、トランザクションがコミットされることを指定します。

USER_CALL_COMPLETEDは、トランザクションをコミットしたユーザー・コールが完了したことを指定します。

例外

サーバーがクライアントよりも先行している場合、SERVER_AHEADがスローされます。したがって、トランザクションは古くなり、すでにコミットされている必要があります。

クライアントがサーバーよりも先行している場合、CLIENT_AHEADがスローされます。これは、サーバーがフラッシュバックされるか、LTXIDが破損している場合にのみ、発生することがあります。これらの状況のいずれかの場合、結果は判定できません。

処理中にエラーが発生した場合、ERRORがスローされ、結果は判定できません。現在のプロシージャの実行中に発生したエラー・コードを指定します。

例

例27-1に、GET_LTXID_OUTCOMEプロシージャをコールし、LTXIDの結果を確認する方法を示します。

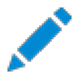
例27-1 LTXIDの結果の確認

```
...
OracleConnection oconn = (OracleConnection) ods.getConnection();
LogicalTransactionId ltxid = oconn.getLogicalTransactionId();
boolean committed = false;
boolean call_completed = false;

try
{
  CallableStatement cstmt = oconn.prepareCall(GET_LTXID_OUTCOME);
  cstmt.setObject(1, ltxid);
  cstmt.registerOutParameter(2, OracleTypes.BIT);
  cstmt.registerOutParameter(3, OracleTypes.BIT);

  cstmt.execute();

  committed = cstmt.getBoolean(2);
  call_completed = cstmt.getBoolean(3);

  System.out.println("LTXID committed ? " + committed);
  System.out.println("User call completed ? " + call_completed);
}
catch (SQLException sqlexc)
{
  System.out.println("Calling GET_LTXID_OUTCOME failed");
  sqlexc.printStackTrace();
}
```

28 Javaのアプリケーション・コンティニュイティ

アプリケーション・コンティニュイティは、汎用目的のアプリケーションから独立したソリューションを提供します。これにより、計画済または計画外の停止発生後にアプリケーションからの作業のリカバリが可能になります。停止は、修復、構成変更またはパッチ適用後のシステム、通信またはハードウェアに関連している可能性があります。

基礎となるソフトウェア、ハードウェア、通信および記憶域レイヤーが停止すると、アプリケーションの実行が失敗することがあります。最悪の場合、ログオン・ストーム[脚注1]を処理するために、中間層のサーバーを再起動しなければならないことがあります。アプリケーション・コンティニュイティ機能は、データベースの停止をアプリケーションにマスクし、エンド・ユーザーがそのような停止にさらされないため、このような問題を克服するのに役立ちます。



ノート:

- この機能を使用するには、トランザクション・ガードを使用する必要があります。
- アプリケーション・コンティニュイティは、Oracle JDBC Thin ドライバの機能であり、JDBC OCI ドライバではサポートされていません。

この章では、次の項でアプリケーション・コンティニュイティに関するJDBCの側面について説明します。

- Java用のアプリケーション・コンティニュイティに対するOracle JDBCの構成について
- Java用のアプリケーション・コンティニュイティに対するOracle Databaseの構成について
- アプリケーション・コンティニュイティにおけるXAデータソースのサポート
- Java用のアプリケーション・コンティニュイティにおけるリクエスト境界の識別について
- 透過的アプリケーション・コンティニュイティのサポート
- アプリケーション・コンティニュイティのリプレイ前の初期状態の確立
- Java用のアプリケーション・コンティニュイティにおける再接続の遅延について
- Java用のアプリケーション・コンティニュイティにおける可変値の保持について
- アプリケーション・コンティニュイティの統計
- Java用のアプリケーション・コンティニュイティにおけるリプレイの無効化について

関連トピック

- Java用のトランザクション・ガード

28.1 Java用のアプリケーション・コンティニュイティに対するOracle JDBCの構成について

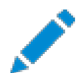
oracle.jdbc.replay.OracleDataSourceImpl、oracle.jdbc.replay.OracleConnectionPoolDataSourceImplまたはoracle.jdbc.replay.driver.OracleXADataSourceImplデータソースを使用して、JDBC接続を取得する必要がありま

す。oracle.jdbc.replay.OracleDataSourceImplとoracle.jdbc.replay.OracleConnectionPoolDataSourceImplの両方をスタンドアロン方式で使用できます。あるいは、ユニバーサル接続プール(UCP)またはOracle WebLogic Server接続プールなどの接続プール用コネクション・ファクトリとして構成することもできます。

Oracle Database 12cリリース2 (12.2.0.1)以降、JDBCリプレイ・ドライバでは、新しいデータソース(XAリプレイ・データソース)が提供されています。これは、JDBC操作のリプレイをサポートしており、高速接続フェイルオーバー、ランタイム接続ロード・バランシングおよびすべてのタイプのRACインスタンス・アフィニティを含め、すべてのOracle RAC機能に対してUCPデータソースとWebLogicのアクティブなGridLinkの単一プールのデータソースとともに動作します。このデータソースを使用するには、oracle.jdbc.replay.OracleXADataSourceインタフェースを実装する必要があります。実際のデータソースの実装クラスは、oracle.jdbc.replay.driver.OracleXADataSourceImplです。接続ファクトリとして、実装クラスをUCPデータソースとOracle WebLogic Server GridLinkデータソースに指定できます。JNDIのファクトリ・クラスは、oracle.jdbc.replay.OracleXADataSourceFactoryです。

ノート:

- XA リプレイ・データソースでは、構成可能なリプレイ・モードが提供されていません。リプレイを有効にするには、リプレイ・データソースを使用して、サーバー側のデータベース・サービス上で FAILOVER_TYPE を TRANSACTION に設定する必要があります(設定されていない場合)。
- Oracle Database リリース 18c 以降、FAILOVER_TYPE を AUTO に設定して、透過的アプリケーション・コンティニュイティを使用することもできます。



関連項目:

Oracle Real Application Clusters 管理およびデプロイメント・ガイド

- リプレイを動的に有効および無効にするには、リプレイ接続プロキシで使用可能な別の API を使用する必要があります。XA リプレイ・データソースでは、接続プーリングが提供されていません。getXAConnection メソッドをコールすると、新しい JDBC XAConnection プロキシが動的に作成されます。これにより、代理として新しい JDBC 物理接続が保持されます。代理となるのは Oracle JDBC ドライバ・オブジェクトです。

次のコードでは、スタンドアロンのJDBCアプリケーションでのoracle.jdbc.replay.OracleDataSourceImplおよびoracle.jdbc.replay.OracleConnectionPoolDataSourceImplの使用方法を示します。

```
import java.sql.Connection;
import javax.sql.PooledConnection;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.replay.OracleDataSourceFactory;
import oracle.jdbc.replay.OracleDataSource;
import oracle.jdbc.replay.OracleConnectionPoolDataSource;

...
{
......
OracleDataSource rds = OracleDataSourceFactory.getOracleDataSource();
```

```
rds.setUser(user);
rds.setPassword(passwd);
rds.setURL(url);
......  // Other data source configuration like callback, timeouts, etc.

Connection conn = rds.getConnection();
((OracleConnection) conn).beginRequest();  // Explicit request begin
......  // JDBC calls protected by Application Continuity
((OracleConnection) conn).endRequest();  // Explicit request end
conn.close();

OracleConnectionPoolDataSource rcpds = OracleDataSourceFactory.getOracleConnectionPoolDataSource();
rcpds.setUser(user);
rcpds.setPassword(passwd);
rcpds.setURL(url);
......  // other data source configuration like callback, timeouts, and so on

PooledConnection pc = rcpds.getPooledConnection();
Connection conn2 = pc.getConnection();  // Implicit request begin
......  // JDBC calls protected by Application Continuity
conn2.close();  // Implicit request end
......
```

接続URLの使用時には次の点に注意する必要があります。

- 接続URLで、常にThinドライバを使用します。
- 常にサービスに接続します。instance_nameまたはSIDは使用しないでください。これらを使用しても、既知の適切なインスタンスにマッピングされません。また、SIDは非推奨です。
- クライアントのADDRESS_LIST内のアドレスがデータベースのREMOTE_LISTENER設定と一致しない場合、接続が行われず、「サービスが見つかりません」というメッセージが表示されます。このため、クライアントのADDRESS_LIST内のアドレスはデータベースのREMOTE_LISTENER設定と一致する必要があります。
  - REMOTE_LISTENERがSCAN_VIPに設定されている場合、ADDRESS_LISTにはSCAN_VIPが使用されます。
  - REMOTE_LISTENERがホストVIPに設定されている場合、ADDRESS_LISTでは同じホストVIPが使用されます。
  - REMOTE_LISTENERがSCAN_VIPとホストVIPの両方に設定されている場合、ADDRESS_LISTにはSCAN_VIPおよび同じホストVIPが使用されます

ノート:

リリース 11.2 より前の Oracle クライアントの場合、SCAN を使用するには、ADDRESS_LIST をアップグレードする必要があります。つまり、ADDRESS_LIST を 3 つの SCAN IP アドレスに対応する 3 つの ADDRESS エントリにまで拡張します。

Database Upgrade Assistant を使用して以前のリリースからアップグレードされるデータベースにこのようなクライアントが接続する場合、HOST VIP に設定されたこれらのクライアントの ADDRESS_LIST を保持する必要があります。ただし、REMOTE_LISTENER が ONLY SCAN に変更されると、またはクライアントが新しくインストールされた Oracle Database 12c リリース 1 (REMOTE_LISTENER が ONLY SCAN)に移動されると、完全なサービス・マップが取得されず、接続

できないことがあります。

- 接続文字列にRETRY_COUNT、RETRY_DELAY、CONNECT_TIMEOUTおよびTRANSPORT_CONNECT_TIMEOUTパラメータを設定します。Oracle Databaseリリース12.1.0.2以降、これは、JDBC Thinドライバ接続を構成する際の一般的な推奨事項です。これらの設定により、ランタイム時、リプレイ時、および計画済停止の作業排出時に新規接続の取得効率が向上します。

  CONNECT_TIMEOUTは、sqlnet.oraファイル内のSQLNET.OUTBOUND_CONNECT_TIMEOUTパラメータと等価であり、完全接続に適用されます。TRANSPORT_CONNECT_TIMEOUTパラメータはADDRESSパラメータに従って適用されます。サービスがフェイルオーバーまたは再起動に対して登録されていない場合、SCANを使用する際に再試行が重要になります。たとえば、SCANアドレスを指すリモート・リスナーを使用する場合、次の設定を使用する必要があります。

```
jdbc:oracle:thin:@(DESCRIPTION =
 (TRANSPORT_CONNECT_TIMEOUT=3000)
 (RETRY_COUNT=20)(RETRY_DELAY=3)(FAILOVER=ON)
 (ADDRESS_LIST =(ADDRESS=(PROTOCOL=tcp)
  (HOST=CLOUD-SCANVIP.example.com)(PORT=5221))
 (CONNECT_DATA=(SERVICE_NAME=orcl)))

REMOTE_LISTENERS=CLOUD-SCANVIP.example.com:5221
```

  同様に、データベースでVIPを指すリモート・リスナーを使用する場合、次の設定を使用する必要があります。

```
jdbc:oracle:thin:@(DESCRIPTION =
(TRANSPORT_CONNECT_TIMEOUT=3000)
(CONNECT_TIMEOUT=60)(RETRY_COUNT=20)(RETRY_DELAY=3)(FAILOVER=ON)
 (ADDRESS_LIST=
 (ADDRESS=(PROTOCOL=tcp)(HOST=CLOUD-VIP1.example.com)(PORT=5221) )
(ADDRESS=(PROTOCOL=tcp)(HOST=CLOUD-VIP2.example.com)(PORT=5221) )
 (ADDRESS=(PROTOCOL=tcp)(HOST=CLOUD-VIP3.example.com)(PORT=5221) ))
(CONNECT_DATA=(SERVICE_NAME=orcl)))
REMOTE_LISTENERS=CLOUD-VIP1.example.com:5221
```

関連項目:

- ローカル・ネーミング・パラメータの詳細は、『Oracle Database Net Servicesリファレンス』を参照してください
- 『Oracle Real Application Clusters管理およびデプロイメント・ガイド』

関連トピック

- データソースおよびURL

28.1.1 アプリケーション・コンティニュイティでの具象クラスのサポート

Oracle Databaseリリース18c以降、JDBCドライバは、アプリケーション・コンティニュイティで次の具象クラスをサポートします。

- oracle.sql.CLOB
- oracle.sql.NCLOB
- oracle.sql.BLOB
- oracle.sql.BFILE

- oracle.sql.STRUCT
- oracle.sql.REF
- oracle.sql.ARRAY

28.2 Java用のアプリケーション・コンティニュイティに対するOracle Databaseの構成について

Java用のアプリケーション・コンティニュイティを使用するには、次のようにOracle Databaseを構成する必要があります。

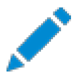
- Oracle Database 12cリリース1 (12.1)以上を使用します
- Oracle Real Application Clusters (Oracle RAC)またはOracle Data Guardを使用している場合、Oracle Notification System(ONS)を使用してFANを構成し、Oracle WebLogic Serverまたはユニバーサル接続プール(UCP)と通信するようにします。
- すべてのデータベース処理に対してアプリケーション・サービスを使用します。サービスを作成するには、次の操作を実行します。
  - Oracle RACを使用している場合、SRVCTLコマンドを実行します。
  - Oracle RACを使用していない場合、DBMS_SERVICEパッケージを使用します。
- リプレイおよびロード・バランシングに必要なプロパティをサービスに設定します。たとえば、次のように設定します。
  - aq_ha_notifications = TRUE: FAN通知を有効にします
  - FAILOVER_TYPE = TRANSACTIONまたはFAILOVER_TYPE = AUTO: アプリケーション・コンティニュイティを使用する場合
  - トランザクション・ガードを有効にする場合: COMMIT_OUTCOME = TRUE
  - リプレイが発生する期間(秒)を設定する場合: REPLAY_INITIATION_TIMEOUT = 900
  - FAILOVER_RETRIES = 30: リプレイごとに接続の再試行回数を指定します
  - FAILOVER_DELAY = 10: 接続の再試行間の遅延時間を秒単位で設定します
  - GOAL = SERVICE_TIME (Oracle RACを使用している場合、これが推奨設定です。)
  - CLB_GOAL = LONG: 通常、クローズしたワークロードに便利です。Oracle RACを使用している場合、これが推奨設定です。他のほとんどのワークロードの場合、SHORTが推奨設定です。
- データベース・サービス(つまり、DB_NAMEまたはDB_UNIQUE_NAMEに対応するデフォルトのサービス)は使用しないでください。このサービスは、Oracle Enterprise ManagerおよびDBA用として予約されています。高可用性のためにデータベース・サービスを使用することは推奨していません。これは、このサービスについて次の作業を行うことができないためです。
  - 有効化と無効化
  - Oracle RACへの再配置
  - Oracle Data Guardへのスイッチオーバー

関連項目:

アプリケーション・コンティニュイティの操作および使用方法の詳細は、『Oracle Database開発ガイド』を参照してください。

28.3 アプリケーション・コンティニュイティにおけるXAデータソースのサポート

Oracle Database 12cリリース2 (12.2.0.1)では、非XAデータソース(javax.sql.DataSource)と同様に、Oracle XAデータソース(javax.sql.XADataSource)をサポートすることによって、アプリケーション・コンティニュイティを強化する新しい機能が導入されました。JDBCとJava Transaction API (JTA)の両方を使用すると、JDBC接続がローカル・トランザクションにもグローバル/XAトランザクションにも同じように参加できます。ただし、多数のカスタマ・アプリケーションがXAデータソースから接続を取得しますが、これらの接続を使用するのはローカル・トランザクションを実行する場合のみです。新機能のアプリケーション・コンティニュイティでは、グローバル/XAトランザクションに昇格可能なローカル・トランザクションを含め、ローカル・トランザクション以外のXA対応のデータソースを使用するアプリケーションにも対応しています。したがって、フェイルオーバーおよびフォワードリカバリなどのアプリケーション・コンティニュイティの利点がこれらのアプリケーションに拡張されました。



ノート:

この機能を使用するには、トランザクション・ガード 12.2 を使用する必要があります。

基礎となる物理接続がグローバル/XAトランザクションに参加したり、XA操作を行うときは常に、リプレイはその接続で無効になっています。他のすべてのXA操作は正常に機能しますが、アプリケーションではアプリケーション・コンティニュイティによる保護が得られません。

前述の理由でリプレイが接続で無効になると、次のリクエストが開始されるまで無効のままです。グローバル/XAトランザクション・モードからローカル・トランザクション・モードに切り替えても、接続でリプレイは自動的に有効になりません。

次のコードでは、接続がOracleXADataSourceから取得された場合に、ローカル・トランザクションからリプレイをサポートし、XAトランザクションに対して無効にする方法を示します。

```
import javax.transaction.xa.*;
import oracle.jdbc.replay.OracleXADataSource;
import oracle.jdbc.replay.OracleXADataSourceFactory;
import oracle.jdbc.replay.ReplayableConnection;
OracleXADataSource xards = OracleXADataSourceFactory.getOracleXADataSource();
xards.setURL(connectURL);
xards.setUser(<user_name>);
xards.setPassword(<password>);
XAConnection xaconn = xards.getXAConnection();
// Implicit request begins
Connection conn = xaconn.getConnection();
/* Local transaction case */
// Request-boundary detection OFF
((ReplayableConnection) conn).beginRequest();
conn.setAutoCommit(false);
PreparedStatement pstmt=conn.prepareStatement( “select cust_first_name,cust_last_name from customers
where customer_id=1”);
ResultSet rs=pstmt.executeQuery();
// Outage happens at this point
// Replay happens at this point
rs.next();
```

```
rs.close();
pstmt.close();
((ReplayableConnection) conn).endRequest();
...
/* Global/XA transaction case */
((ReplayableConnection) conn).beginRequest();
conn.setAutoCommit(false);
XAResource xares = xaconn.getXAResource();
Xid xid = createXid();
// Replay is disabled here
xares.start(xid, XAResource.TMNOFLAGS);
conn.prepareStatement( “INSERT INTO TEST_TAB VALUES(200, 'another new record')” );
// outage happens at this point
try {
//No replay here and throws exception
conn.executeUpdate();
}
// sqlrexc.getNextException() shows the reason for the replay failure
catch (SQLRecoverableException sqlrexc) {

    …...
}
```

28.4 Java用のアプリケーション・コンティニュイティにおけるリクエスト境界の識別について

リクエストは、アプリケーション・コンティニュイティによって保護されているOracle Databaseへの物理的な接続における作業の単位です。リクエスト境界は、ユースケースのシナリオによって異なります。リクエストが開始するのは接続がユニバーサル接続プール(UCP)またはWebLogic Server接続プールから流用されているときで、終了するのは接続が接続プールに返却されたときです。

JDBCドライバには、oracle.jdbc.OracleConnectionインタフェースで明示的なリクエスト境界宣言APIのbeginRequestおよびendRequestが用意されています。これらのAPIを使用すると、アプリケーション、フレームワークおよび接続プールが境界ポイントをJDBCリプレイ・ドライバに示すことができます。この境界ポイントで、安全にコール履歴を解放し、前のリクエストでリプレイが無効になっていた場合にこれを有効にすることができます。リクエストの終了時に、JDBCリプレイ・ドライバは、APIがコールされた接続について記録された履歴を削除します。これによって、アプリケーションは接続をプールに返却しないで延長して同じ接続を使用するので、メモリー消費がさらに節約されます。

接続プールが動作するには、必要な場合にアプリケーションが接続を取得し、使用していない場合に接続を解放する必要があります。これによって、リクエスト境界をより正しく見積もり、透過的に提供できます。APIは、リソースの消費量、リカバリ、ロード・バランシングのパフォーマンスを改善する以外に、アプリケーションに影響を及ぼしません。これらのAPIでは、JDBCメソッド、SQLまたはPL/SQLをコールすることによって、接続状態が変更されません。ローカル・トランザクションのオープン中にリクエストの開始や終了の試行を行った場合、エラーが返されます。

28.5 透過的アプリケーション・コンティニュイティのサポート

Oracle Databaseリリース18cには、アプリケーション・コンティニュイティの機能モードである透過的アプリケーション・コンティニュ

イティ機能が導入されました。透過的アプリケーション・コンティニュイティは、セッションおよびトランザクションの状態を透過的に追跡および記録し、リカバリ可能な停止の後にデータベース・セッションをリカバリできるようにします。これは、安全に、アプリケーションの知識を必要とすることなく、アプリケーション・コードも変更せずに実行されます。透過性は、状態を追跡するインフラストラクチャを使用して実現されます。このインフラストラクチャは、アプリケーションがユーザー・コールを発行すると、セッション状態の使用状況を分類します。この機能により、ドライバは可能なリクエスト境界(暗黙的なリクエスト境界と呼ばれる)を検出して挿入できます。暗黙的なリクエスト境界では次のようになります。

- オブジェクトはオープンされない
- カーソルはドライバ文キャッシュに戻される
- トランザクションはオープンされない

このような場合のセッション状態はリストア可能であることがわかっています。無効化イベントがあった場合、ドライバは現在の取得をクローズして新しいイベントを起動するか、取得を有効にします。サーバーへの次コールで、サーバーは検証を行い、必要に応じて、以前は明示的な境界がなかった場所に、リクエスト境界を作成します。

透過的アプリケーション・コンティニュイティを使用するには、データベース・サービス上のサーバー側サービス属性FAILOVER_TYPEをAUTOに設定する必要があります。

暗黙的なリクエストのサポートは、アプリケーション・フェイルオーバーのリカバリ時間の短縮、およびアプリケーション・コンティニュイティの最適化に役立ちます。透過的アプリケーション・コンティニュイティを使用して、サーバーおよびドライバはトランザクションとセッション状態の使用状況を追跡できます。ただし、リクエスト中にサーバーのセッション状態を変更するアプリケーションの場合、この機能は注意して使用する必要があります。JDBC Thinドライバには、必要な場合に暗黙的にリクエストをオフにするoracle.jdbc.enableImplicitRequestsプロパティが用意されています。このプロパティは、システム・レベル(すべての接続に適用される)または接続レベル(特定の接続に適用される)で設定できます。デフォルトでは、このプロパティの値はtrueです。これは、暗黙的なリクエストのサポートが有効になっていることを意味します。

関連項目:

Oracle Real Application Clusters管理およびデプロイメント・ガイド

Oracle Database Release 19c以降では、FAILOVER_TYPEサービス属性の値をAUTOに設定すると、Oracle JDBCドライバは、リプレイ・データ・ソースによって作成された新しい物理接続ごとに暗黙的にリクエストを開始します。この機能により、サード・パーティの接続プールを使用するアプリケーションは、リクエスト境界を挿入するためのコード変更なしで、透過的アプリケーション・コンティニュイティ(TAC)を簡単に使用できるようになります。

この暗黙のbeginRequestは、リプレイ・データ・ソースにのみ適用されます。また、TACが有効なときには、実行時に作成された物理接続にのみ適用されます。ドライバは、リプレイ試行中の各再接続後に暗黙的にリクエストを開始しません(つまり、beginRequestがリプレイ接続時に暗黙的に挿入されません)。また、手動アプリケーション・コンティニュイティ・モード(FAILOVER_TYPEサービス属性をTRANSACTIONに設定)の場合も暗黙的にリクエストを開始しません。

この機能を明示的にオフにする場合は、Javaシステム・プロパティoracle.jdbc.beginRequestAtConnectionCreationの値をfalseに設定します。このプロパティのデフォルト値はtrueです。

関連トピック

- FAILOVER_RESTOREの有効化について

28.6 アプリケーション・コンティニュイティのリプレイ前の初期状態の確立

非トランザクション・セッション状態(NTSS)とは、データベース・トランザクションの外部に存在し、リカバリによって保護されていないデータベース・セッションの状態です。ステートフル・リクエストを使用するアプリケーションの場合、非トランザクション状態は再構築済セッションとして再確立されます。

リクエストの開始時のみに状態を設定するアプリケーションの場合、または事前設定済の状態で接続を使用してパフォーマンス上の利点を得るステートフル・アプリケーションの場合、次のコールバック・オプションの中からいずれかを指定します。

- コールバックなし
- 接続ラベリング
- 接続初期化コールバック
- FAILOVER_RESTOREの有効化について

28.6.1 コールバックなし

このシナリオの場合、アプリケーションは各リクエスト時に独自の状態を構築します。

28.6.2 接続ラベリング

このシナリオは、ユニバーサル接続プール(UCP)およびOracle WebLogic Serverのみに適用可能です。接続に事前設定された状態を活用するようアプリケーションを変更できます。接続ラベリングAPIにより、接続の一致状況が判別され、接続が流用される際にコールバックを使用してギャップが移入されます。

関連項目:

Oracle Universal Connection Pool開発者ガイド

28.6.3 接続初期化コールバック

このシナリオの場合、リプレイ・ドライバがアプリケーション・コールバックを使用して、ランタイムおよびリプレイ時にセッションの初期状態を設定します。JDBCリプレイ・ドライバは、オプションの接続の初期化コールバック・インタフェースおよびこのようなコールバックの登録と登録解除のメソッドを提供します。

接続初期化コールバックは、登録されると、リカバリ可能なエラーの後に成功した再接続ごとに実行されます。アプリケーションは、初期化アクションがフェイルオーバー前の元の接続時のものと同じであることを確認する必要があります。コールバックの起動が失敗した場合、リプレイはその接続で無効になります。

次の項で、初期化コールバックについて説明します。

- 初期化コールバックの作成

● 初期化コールバックの登録

● 初期化コールバックの削除または登録解除

28.6.3.1 初期化コールバックの作成

JDBC接続の初期化コールバックを作成するには、アプリケーションで

oracle.jdbc.replay.ConnectionInitializationCallbackインタフェースを実装します。

oracle.jdbc.replay.OracleDataSourceインタフェースのインスタンスごとに、1つのコールバックが許可されます。

ノート:

このコールバックは、再接続の成功後、フェイルオーバー中にのみ起動されます。

例

次のコードでは、単純な初期化コールバックの実装を示します。

```
import oracle.jdbc.replay.ConnectionInitializationCallback;
class MyConnectionInitializationCallback implements ConnectionInitializationCallback
{
    public MyConnectionInitializationCallback()
    {
        ...
    }
    public void initialize(java.sql.Connection connection) throws SQLException
    {
        // Reset the state for the connection, if necessary (like ALTER SESSION)
        ...
    }
 }
```

XAデータソースを使用するアプリケーションの場合、接続初期化コールバックがXAリプレイ・データソース上で登録されます。次の両方が発生した場合は常に、コールバックが実行されます。

● 接続が接続プールから流用されるとき。

● リプレイXAデータソースがフェイルオーバーで新しい物理接続を取得したとき。

ノート:

接続の初期化は冪等にする必要があります。接続がすでに初期化されている場合、これを繰り返す必要はありません。これにより、アプリケーションでは、フェイルオーバー後でリプレイか開始される前にセッションの最初の開始ポイントを再確立できるようになります。コールバックの実行では、オープンしているローカル・トランザクションをコミットまたはロールバックしないで残す必要があります。これに違反すると、例外が発生します。

コールバックの起動が失敗した場合、リプレイはその接続で無効になります。たとえば、アプリケーションでは接続の設定フェーズをこのコールバックに埋め込みます。

28.6.3.2 初期化コールバックの登録

JDBCリプレイ・ドライバがoracle.jdbc.replay.OracleDataSourceインタフェースで提供する次のメソッドを使用して、接続の初期化コールバックを登録します。

```
registerConnectionInitializationCallback(ConnectionInitializationCallback cbk)
```

OracleDataSourceインタフェースのインスタンスごとに、1つのコールバックが許可されます。

XAデータソースを使用している場合、oracle.jdbc.replay.OracleXADataSourceインタフェースでregisterConnectionInitializationCallback(ConnectionInitializationCallback cbk)メソッドを使用します。

28.6.3.3 初期化コールバックの削除または登録解除

JDBCリプレイ・ドライバがoracle.jdbc.replay.OracleDataSourceインタフェースで提供する次のメソッドを使用して、接続の初期化コールバックの登録を解除します。

```
unregisterConnectionInitializationCallback(ConnectionInitializationCallback cbk)
```

XAデータソースを使用している場合、oracle.jdbc.replay.OracleXADataSourceインタフェースでunregisterConnectionInitializationCallback(ConnectionInitializationCallback cbk)メソッドを使用します。

28.6.4 FAILOVER_RESTOREの有効化について

FAILOVER_RESTOREサービス属性は、Oracle Database 12cリリース2 (12.2.0.1).で導入されました。FAILOVER_RESTOREをLEVEL1に設定すると、リクエストをリプレイする前の共通の初期状態に自動的にリストアされます。デフォルトでは、FAILOVER_RESTORE属性の値がNONEに設定されていますが、これは無効になっているということです。

Oracle Databaseリリース18c以降、この属性の値をAUTOに設定することもできます。また、FAILOVER_TYPE属性の値をAUTOに設定すると、FAILOVER_RESTOREは自動的にAUTOに設定されます。FAILOVER_TYPEがAUTOに設定されているかぎり、FAILOVER_RESTOREの値をそれ以外に変更することはできません。FAILOVER_RESTOREがAUTOに設定されている場合、共通の初期状態も設定されます。セッション状態のリストアに関するかぎり、この設定は、FAILOVER_RESTOREをLEVEL1に設定した場合と同じ機能を提供します。

関連項目:

『Oracle® Real Application Clusters管理およびデプロイメント・ガイド』

ノート:

Oracle Database リリース 18c で使用可能な Java 用のアプリケーション・コンティニュイティの場合、次の初期状態が FAILOVER_RESTORE でサポートされています。

- NLS_CALENDAR
- NLS_CURRENCY

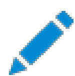
- NLS_DATE_FORMAT
- NLS_DATE_LANGUAGE
- NLS_DUAL_CURRENCY
- NLS_ISO_CURRENCY
- NLS_LANGUAGE
- NLS_LENGTH_SEMANTICS
- NLS_NCHAR_CONV_EXCP
- NLS_NUMERIC_CHARACTER
- NLS_SORT
- NLS_TERRITORY
- NLS_TIME_FORMAT
- NLS_TIME_TZ_FORMAT
- NLS_TIMESTAMP_FORMAT
- NLS_TIMESTAMP_TZ_FORMAT
- TIME_ZONE （OCI, ODP.NET 12201）
- CURRENT_SCHEMA
- MODULE
- ACTION
- CLIENT_ID
- ECONTEXT_ID
- ECONTEXT_SEQ
- DB_OP
- AUTOCOMMIT 状態(Java および SQL*Plus)
- CONTAINER (PDB)および SERVICE (OCI および ODP.NET の場合)

Oracle Database リリース 19c では、FAILOVER_RESTORE に対して次の追加の初期状態がサポートされています。

- ERROR_ON_OVERLAP_TIME
- EDITION

- SQL_TRANSLATION_PROFILE
- ROW ARCHIVAL VISIBILITY
- ROLEs
- CLIENT_INFO

ノート:



Oracle Database リリース 19c では、FAILOVER_RESTORE が透過アプリケーション・コンティニュイティ(TAC)モードで自動的に有効になります。 次に示す状態は、Thin ドライバでサポートされていないため、自動リストア・オプションから除外されます。

- NLS_COMP
- CALL_COLLECT_TIME

多くのアプリケーションの場合、ACリプレイに必要な初期状態を自動的にリストアするには、コールバックの使用は不要で、FAILOVER_RESTOREの有効化で十分です。アプリケーションで前述のリストに記載されていない初期状態が必要な場合、または、アプリケーションで初期状態の設定の明示的な制御が適切な場合、アプリケーションでは接続ラベリングまたは初期化コールバックのいずれかのコールバックを使用する必要があります。コールバックを構成すると、初期化コールバックが同時に有効になっている場合にFAILOVER_RESTOREでリストアされた初期状態がオーバーライドされます。

28.7 Java用のアプリケーション・コンティニュイティにおける再接続の遅延について

デフォルトでは、JDBCリプレイ・ドライバは、フェイルオーバーを開始する際に、サービスが使用可能なインスタンスで進行中の作業をリカバリしようとします。これを実行するには、ドライバは最初に、実行中のインスタンスに対して適切な接続を再確立する必要があります。サービスが再配置および発行される前にデータベースまたはインスタンスを再起動する必要がある場合、再接続に時間がかかる場合があります。したがって、サービスが別のインスタンスまたはデータベースから利用可能になるまでに、フェイルオーバーを遅延させる必要があります。

最大遅延はFAILOVER_RETRIESにFAILOVER_DELAYを掛けた値として計算されるため、遅延を維持するためにFAILOVER_RETRIESパラメータとFAILOVER_DELAYパラメータを使用する必要があります。これらのパラメータは計画済停止と連動して使用すると効果があります。たとえば、サービスが数分間使用不可になる可能性がある停止の場合などです。FAILOVER_DELAYおよびFAILOVER_RETRIESパラメータを設定する場合は、REPLAY_INITIAITION_TIMEOUTパラメータの値を最初に確認します。このパラメータのデフォルト値は900秒です。FAILOVER_DELAYパラメータの値が高い場合、リプレイが取り消されることがあります。

パラメータ名	使用可能な値	デフォルト値
FAILOVER_RETRIES	正の整数ゼロ以上	30
FAILOVER_DELAY	秒数	10

28.7.1 Java用のアプリケーション・コンティニュイティに関する構成例

次の項では、サービスの作成と変更の構成例について説明します。

- Oracle RACでのサービスの作成
- 単一インスタンス・データベースでのサービスの変更

28.7.1.1 Oracle RACでのサービスの作成

Oracle RACまたはOracle RAC Oneを使用している場合、SRVCTLコマンドを使用して次の方法でサービスを作成および変更します。

透過的アプリケーション・コンティニュイティの場合

次のように、透過的アプリケーション・コンティニュイティを使用するサービスを作成できます。

ポリシー管理データベースの場合:

```
$ srvctl add service -db codedb -service GOLD -serverpool ora.Srvpool -clbgoal SHORT -rlbgoal
SERVICE_TIME -failover_restore AUTO -failoverretry 30
-failoverdelay 10 -commit_outcome TRUE -failovertype AUTO -replay_init_time 1800 -retention 86400  -
notification TRUE
```

管理者管理データベースの場合:

```
$ srvctl add service -db codedb -service GOLD -preferred serv1 -available serv2 -clbgoal SHORT -
rlbgoal SERVICE_TIME -failover_restore AUTO
-failoverretry 30 -failoverdelay 10 -commit_outcome TRUE -failovertype AUTO -replay_init_time 1800 -
retention 86400 -notification TRUE
```

手動によるアプリケーション・コンティニュイティの場合

次のように、手動によるアプリケーション・コンティニュイティを使用するサービスを作成できます。

ポリシー管理データベースの場合:

```
$ srvctl add service -db codedb -service GOLD -serverpool ora.Srvpool -clbgoal SHORT -rlbgoal
SERVICE_TIME -failover_restore LEVEL1 -failoverretry 30
-failoverdelay 10 -commit_outcome TRUE -failovertype TRANSACTION -replay_init_time 1800 -retention
86400 -notification TRUE
```

管理者管理データベースの場合:

```
$ srvctl add service -db codedb -service GOLD -preferred serv1 -available serv2 -clbgoal SHORT -
rlbgoal SERVICE_TIME -failover_restore LEVEL1
-failoverretry 30 -failoverdelay 10 -commit_outcome TRUE -failovertype TRANSACTION -replay_init_time
1800 -retention 86400 -notification TRUE
```

28.7.1.2 単一インスタンス・データベースでのサービスの変更

単一インスタンス・データベースを使用している場合、DBMS_SERVICEパッケージを使用して次の方法でサービスを変更します。

```
declare
params dbms_service.svc_parameter_array;
```

```
begin
params('FAILOVER_TYPE'):='TRANSACTION';
params('REPLAY_INITIATION_TIMEOUT'):=1800;
params('RETENTION_TIMEOUT'):=604800;
params('FAILOVER_DELAY'):=10;
params('FAILOVER_RETRIES'):=30;
params('commit_outcome'):='true';
params('aq_ha_notifications'):='true';
dbms_service.modify_service('[your service]',params);
end;
/
```

28.8 Java用のアプリケーション・コンティニュイティにおける可変値の保持について

可変オブジェクトとは、変数、関数からの戻り値、またはコールされるたびに異なる値を戻す構造体です。たとえば、Sequence.NextVal、SYSDATE、SYSTIMESTAMPおよびSYS_GUIDです。リプレイ時に名前付きの関数の結果を保持するには、DBAは関数を起動するユーザーにKEEP権限を付与する必要があります。このセキュリティの制限は、ユーザーが所有していないコードに対してリプレイが関数の結果の保存およびリストアを行うことが有効であることを保証するために強制されています。

関連項目:

『Oracle Database開発ガイド』

28.8.1 インタフェースの付与または取消し

次のように標準的なGRANTおよびREVOKEインタフェースを使用して、可変値を使用できます。

- DateおよびSYS_GUID構文
- Sequence構文
- GRANT ALL文
- 可変値における付与のルール

28.8.1.1 DateおよびSYS_GUID構文

DATE_TIMEおよびSYS_GUID構文は次のとおりです。

```
GRANT [KEEP DATE TIME|SYSGUID]..[to USER}
REVOKE [KEEP DATE TIME | KEEP SYSGUID] … [from USER]
```

たとえば、元の日付をEBSで標準的に使用する場合:

```
Grant KEEP DATE TIME,  KEEP SYSGUID  to [custom user];
Grant KEEP DATE TIME,  KEEP SYSGUID  to [apps user];
```

28.8.1.2 Sequence構文

Sequence構文には次のタイプがあります。

- 所有済Sequence構文
- その他のSequence構文

所有済Sequence構文

```
ALTER SEQUENCE [sequence object] [KEEP|NOKEEP];
```

このコマンドは、リプレイ後にキーが一致するように、リプレイ用にsequence.nextvalの元の値を保持します。ほとんどのアプリケーションはリプレイ時にシーケンスの値を保持する必要があります。ALTER SYNTAXは所有済シーケンス専用です。

その他のSequence構文

```
GRANT KEEP SEQUENCE..[to USER] on [sequence object];
REVOKE  KEEP SEQUENCE … [from USER]  on [sequence object];
```

たとえば、元のシーケンス値をEBSで標準的に使用する場合、次のコマンドを使用します。

```
Grant KEEP SEQUENCE  to [apps user] on [sequence object];
Grant KEEP SEQUENCE  to [custom user] on [sequence object];
```

28.8.1.3 GRANT ALL文

GRANT ALL文では、KEEP権限をユーザーのすべてのオブジェクトに付与します。ただし、ここでは可変値は除外されます。つまり、可変値には明示的な付与が必要です。

28.8.1.4 可変値における付与のルール

可変オブジェクトに権限を付与する場合、次のルールに従います。

- 可変値に対してKEEP権限が付与されているユーザーの場合、SYS_GUID、SYSDATEおよびSYSTIMESTAMP関数がコールされたとき、オブジェクトは可変アクセスを継承します。
- シーケンス・オブジェクトの可変値に対するKEEP権限が取り消された場合、このオブジェクトを使用するSQLまたはPL/SQLブロックは、このシーケンスに対して可変コレクションまたはアプリケーションを許可しません。
- 付与された権限がランタイムとフェイルオーバーの間に取り消されると、収集された可変値はリプレイに適用されません。
- 新しい権限が実行時とフェイルオーバー間で付与された場合、可変値は収集されず、これらの値はリプレイに適用されません。

28.9 アプリケーション・コンティニュイティの統計

JDBCリプレイ・ドライバでは、アプリケーション・コンティニュイティを使用したアプリケーションの次の統計をサポートしています。

- リクエストの合計数
- 完了したリクエストの合計数
- 合計コール数
- 保護されたコールの合計数
- 停止の影響を受けるコールの合計数
- リプレイをトリガーするコールの合計数

- リプレイ中に停止の影響を受けるコールの合計数
- 成功したリプレイの合計数
- 失敗したリプレイの合計数
- 無効になったリプレイの合計数
- リプレイ試行の合計数

これらのすべてのメトリックは、接続単位でも接続全体でも使用できます。これらの統計を取得するには、次のメソッドを使用できます。

- getReplayStatistics(StatisticsReportType)

  oracle.jdbc.replay.ReplayableConnection.getReplayStatistics(StatisticsReportType)メソッドを使用して、スナップショット統計を取得します。このメソッドの引数は、同じReplayableConnectionインタフェースでも定義済の列挙型です。接続全体の統計を取得するには、アプリケーションのメイン・ロジックの後にこのメソッドをコールすることをお薦めします。アプリケーションでは、オープンしているoracle.jdbc.replay.ReplayableConnectionを使用することも、同じデータソースに対して新規接続をオープンすることもできます。これは、UCPとWLSの両方のデータソースを使用しているアプリケーション、およびリプレイ・データソースを直接使用しているアプリケーションに適用されます。

- getReplayStatistics()

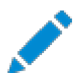
  oracle.jdbc.replay.OracleDataSource.getReplayStatistics()メソッドを使用して接続全体の統計を取得します。これは、リプレイ・データソースを直接使用しているアプリケーションにのみ適用されます。

両方のメソッドは、個々のリプレイ・メトリックを取得できる位置からoracle.jdbc.replay.ReplayStatisticsオブジェクトを戻します。次にReplayStatisticsオブジェクトを文字列として出力するサンプル出力を示します。

```
AC Statistics:
===============================================
TotalRequests = 1
TotalCompletedRequests = 1
TotalCalls = 19
TotalProtectedCalls = 19
===============================================
TotalCallsAffectedByOutages = 3
TotalCallsTriggeringReplay = 3
TotalCallsAffectedByOutagesDuringReplay = 0
===============================================
SuccessfulReplayCount = 3
FailedReplayCount = 0
ReplayDisablingCount = 0
TotalReplayAttempts = 3
===============================================
```

接続単位またはすべての接続の累積したリプレイ統計をクリアする場合、次のメソッドを使用できます。

- oracle.jdbc.replay.ReplayableConnection.clearReplayStatistics(ReplayableConnection.StatisticsReportType reportType)
- oracle.jdbc.replay.OracleDataSource.clearReplayStatistics()



ノート:

すべての統計には、直近のクリア以降の更新のみが反映されています。

28.10 Java用のアプリケーション・コンティニュイティにおけるリプレイの無効化について

この項では、次の概念について説明します。

- リプレイを無効にする方法
- リプレイを無効にするタイミング
- 診断とトレース

28.10.1 リプレイを無効にする方法

アプリケーション・モジュールがリプレイに不適切な設計を使用する場合、リプレイの無効化APIがリプレイごとにリプレイを無効にします。リプレイの無効化は、oracle.jdbc.replay.ReplayableConnectionインタフェースのdisableReplayメソッドを使用して、コールバックまたはメイン・コードに追加できます。たとえば:

```
if (connection instanceof oracle.jdbc.replay.ReplayableConnection)
{
    ((oracle.jdbc.replay.ReplayableConnection)connection).disableReplay();

}
```

リプレイを無効にしても、JDBCメソッド、SQLまたはPL/SQLの再実行によって接続状態が変更されません。リプレイの無効化APIを使用してリプレイを無効にした場合、リクエストが終了するまで記録もリプレイも無効になります。リプレイされたリクエストで時間のギャップのあるデータベース・セッションを再確立しても無効であるため、リプレイを有効にするAPIはありません。これにより、必要なコールのすべての履歴が記録された場合にのみ、リプレイが実行されます。

28.10.2 リプレイを無効にするタイミング

デフォルトでは、JDBCリプレイ・ドライバは次のリカバリ可能なエラーをリプレイします。リプレイの無効化APIは、データベース・セッションを失ってリカバリできないアプリケーション・モジュールのエントリ・ポイントで使用できます。たとえば、アプリケーションがUTL_SMTPパッケージを使用し、メッセージを繰り返す必要がない場合、disableReplay APIは無効にする必要があるリクエストのみに影響します。他のすべてのリクエストは引き続きリプレイされます。

リプレイに対してアプリケーションを構成する前に検討する必要があるシナリオは、次のとおりです。

- アプリケーションが繰り返す必要のない外部PL/SQLアクションをコールする場合
- アプリケーションが独立セッションを同期化する場合
- アプリケーションが実行ロジックの中間層で時間を使用する場合
- アプリケーションでROWIDが変更されないことが想定される場合
- アプリケーションで悪影響が一度発生することが想定される場合
- アプリケーションでロケーション値が変更されないことが想定される場合

28.10.2.1 繰り返さない外部システムをコールするアプリケーション

リプレイ中に、自律型トランザクションと外部システム(PL/SQLコールやJavaコールなど)が、メイン・トランザクションとは別の悪影響を及ぼす可能性があります。リプレイしないように指定しないと、これらの悪影響がリプレイされ、結果が持続します。これらの悪影響には、外部テーブルへの書込み、電子メールの送信、PL/SQLまたはJavaからのセッションの分岐化、ファイルの転送、外部URLへのアクセスなどがあります。たとえば、PL/SQLメッセージを送信する場合、コミットせずに作業中にその作業を離れ、セッションがタイムアウトしたとします。そこで、Ctrl+Cコマンドを発行した場合、フォアグラウンドのコンポーネントが失敗します。作業を再発行したとき、この悪影響も繰り返される可能性があります。

関連項目:

発生する可能性のあるアプリケーション・コンティニュイティの悪影響の詳細は、『Oracle Real Application Clusters管理およびデプロイメント・ガイド』を参照してください

外部アクションのリプレイを有効にするかどうかを意識的に決定する必要があります。たとえば、このような決定が重要である、次の状況を考慮します。

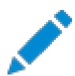
- UTL_HTTPパッケージを使用してSOAコールを発行する。
- UTL_SMTPパッケージを使用してメッセージを送信する。
- UTL_URLパッケージを使用してWebサイトにアクセスする。

外部アクションがリプレイしないようにする場合、disableReplay APIを使用します。

28.10.2.2 アプリケーションが独立セッションを同期化する場合

コミット、ロールバックまたはセッションの喪失まで維持される揮発エンティティを使用して、アプリケーションが独立セッションを同期する場合、リプレイ用にアプリケーションを構成できます。このような場合、アプリケーションは、データベースのロックなどのリソースを使用して相互に依存している、いくつかのデータソースに接続した複数のセッションを同期します。アプリケーションでこれらのセッションのみがシリアライズされ、いずれかのセッションが失敗する可能性があることが認識されている場合、このような同期が有効なことがあります。ただし、1つのデータソースによって保持されているロックまたは他の高揮発リソースが、他の接続から同一または別個のデータソースのデータに対する排他的アクセスを実現することをアプリケーションが想定している場合、この想定はリプレイ時に否定される可能性があります。

リプレイ中、ドライバは、ロックまたは他の高揮発リソースを保持している1つのセッションにこれらのセッションが依存していることを認識していません。また、リソース(セマフォ、デバイスまたはソケットなどの)を使用するパイプ、バッファ・キュー、ストアド・プロシージャを使用して、失敗によって失われる同期を実行することもできます。



ノート:

DBMS_LOCK は、制限されたバージョンではリプレイしません。

28.10.2.3 アプリケーションが実行ロジックの中間層で時間を使用する場合

この場合、アプリケーションは実行ロジックに従って中間層の実時間を使用します。JDBCリプレイ・ドライバは、中間層の時刻ロジックは繰り返さず、このロジックの一部として実行されるデータベース・コールを使用します。たとえば、中間層の時間を使用するアプリケーションが明示的にこれを使用していなければ、時間T1で実行された文が時間T2で再実行されていないと想定することがあります。

28.10.2.4 アプリケーションでROWIDが変更されないことが想定される場合

アプリケーションがROWIDをキャッシュする場合、データベースが変更されているためにROWIDへのアクセスが無効になることがあります。ROWIDが表内の一意の行を特定しても、次のような状況ではROWIDの値が変更されることがあります。

- 基礎となる表が再編成された場合
- 表に索引が作成された場合
- 基礎となる表がパーティション化された場合
- 基礎となる表が移行された場合
- 基礎となる表がEXP/IMP/DULを使用してエクスポートおよびインポートされた場合
- 基礎となる表がGolden Gateまたはロジカル・スタンバイまたは他のレプリケーション・テクノロジを使用して再構築された場合
- 基礎となる表のデータベースがフラッシュバックされたかリストアされた場合

アプリケーションがROWIDを将来的に使用するために保存することは、対応する行が存在しないかまったく異なるデータを含んでいることがあるため、適切ではありません。

28.10.2.5 アプリケーションで悪影響が一度発生することが想定される場合

この場合、リプレイ中にリプレイされる処理は、次のとおりです。

- 自律型トランザクション
- メイン・トランザクションの悪影響と別のバック・チャネルのオープン

メイン・トランザクションと別のバック・チャネルの例には、外部表への書込み、電子メールの送信、PL/SQLまたはJavaからのセッションの分岐化、出力ファイルへの書込み、ファイルの転送および例外ファイルへの書込みがあります。これらのアクションのいずれかは、リプレイがないときも悪影響が持続します。バック・チャネルに、引き続き結果が残っていることがあります。たとえば、ユーザーがトランザクションをコミットしないままにしてセッションがタイムアウトした場合、Ctrl+Cを押すと、フォアグラウンドのコンポーネントが失敗します。ユーザーが作業を再発行した場合、悪影響が繰り返されることがあります。

28.10.2.6 アプリケーションでロケーション値が変更されないことが想定される場合

SYSCONTEXTオプションは、各国語サポート(NLS)設定、ISDBA、CLIENT_IDENTIFIER、MODULEおよびACTIONなどのロケーション非依存セットと、物理ロケータを使用するロケーション依存セットで構成されています。通常、アプリケーションではテスト環境を除いて、物理識別子を使用しません。物理ロケータがメインライン・コードで使用されている場合、リプレイによって不一致が検出され、拒否されます。ただし、コールバックで物理的なロケータを使用することは適切です。

例

```
select
```

```
    sys_context('USERENV','DB_NAME')
    ,sys_context('USERENV','HOST')
    ,sys_context('USERENV','INSTANCE')
    ,sys_context('USERENV','IP_ADDRESS')
    ,sys_context('USERENV','ISDBA')
    ,sys_context('USERENV','SESSIONID')
    ,sys_context('USERENV','TERMINAL')
    ,sys_context('USERENV',ID')
from dual
```

28.10.3 診断とトレース

JDBCリプレイ・ドライバは、標準のJDKロギングをサポートしています。ロギングは、Javaのコマンドラインの-Djava.util.logging.config.file=<file>オプションを使用して有効にします。ログ・レベルは、ログ構成ファイルのoracle.jdbc.internal.replay.level属性で制御されます。たとえば:

```
oracle.jdbc.internal.replay.level = FINER|FINEST
```

ここで、FINERは外部APIを作成し、FINESTは大量のトレースを作成します。管理されている場合にのみ、FINESTを使用する必要があります。

java.util.logging.XMLFormatterクラスを使用してログ・レコードをフォーマットする場合、ログは読み取りやすくなりますが、大きくなります。UCPまたはWebLogic Serverで有効になっているFANでリプレイを使用している場合、FANを処理するロギングも有効にする必要があります。

関連項目:

『Oracle Universal Connection Pool for JDBC開発者ガイド』

28.10.3.1 リプレイ・トレースのコンソールへの書込み

次に、ロギング構成用の構成ファイルの例を示します。

```
oracle.jdbc.internal.replay.level = FINER
handlers = java.util.logging.ConsoleHandler
java.util.logging.ConsoleHandler.level = ALL
java.util.logging.ConsoleHandler.formatter = java.util.logging.XMLFormatter
```

28.10.3.2 リプレイ・トレースのファイルへの書込み

次に、ロギング構成用のpropertiesファイルの例を示します。

```
oracle.jdbc.internal.replay.level = FINEST

# Output File Format (size, number and style)
# count: Number of output files to cycle through, by appending an integer to the base file name:
# limit: Limiting size of output file in bytes
handlers = java.util.logging.FileHandler
java.util.logging.FileHandler.pattern = [file location]/replay_%U.trc
java.util.logging.FileHandler.limit = 500000000
java.util.logging.FileHandler.count = 1000
```

```
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter
```

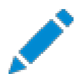
脚注の凡例

脚注1:

ログオン・ストームとは、クライアントの接続リクエストの数が突然増加することです。

29 Oracle JDBCによるFANイベントのサポート

Oracle Database 12cリリース2 (12.2.0.1)以降、Oracle JDBCドライバでは計画済または計画外の停止にOracle RAC高速アプリケーション通知(FAN)イベントがサポートされています。これにより、サードパーティの接続プールが高可用性のためにOracle RAC機能を容易に利用できるようになりました。Oracleユニバーサル接続プール(UCP)またはWebLogic Serverを使用しないJavaアプリケーションが、このサポートを利用できます。たとえば、Oracle RACサーバー側のローリング・アップグレードなどのシナリオでは、アプリケーション内でJDBCエラーが発生しません。



ノート:

Oracle JDBC ドライバでは FAN イベントがサポートされていますが、Oracle UCP ではすべての FAN イベントに対してより包括的なサポートが提供されています。

関連項目:

Oracle Universal Connection Pool開発者ガイド

- Oracle JDBCによるFANイベントのサポートの概要
- 計画済メンテナンスの安全なドレイニングAPI
- FANイベントのサポートのためのOracle JDBCドライバのインストールおよび構成

29.1 Oracle JDBCによるFANイベントのサポートの概要

この機能を使用するには、単一インスタンスのデータベースでOracle RACデータベースまたはOracle Restartを使用する必要があります。この機能により、次がサポートされています。

- 計画済のメンテナンス

  これはOracle RACサーバー上で計画済メンテナンスに対応しており、Oracle RACサービスは正常に停止できます。この場合、接続プールから流用された接続または使用中の接続は中断されず、安全なドレイニングを行うAPIが起動されて初めてクローズされます。たとえば、アプリケーションが接続などの作業を終了してこれを接続プールに戻すときなどです。

- 計画外停止

  この場合、無効な接続はただちに検出されて中断され、これによりネットワーク接続がハングしないように切断されます。この場合、流用された接続および使用中の接続には、計画外停止時に割込みが発生します。アプリケーションでは影響を受ける接続の例外を処理し、そのアプリケーション自体で、またはアプリケーション・コンティニュイティなどのOracle高可用性ソリューションを使用して、必要なリカバリを実行します。

関連トピック

- Java用のアプリケーション・コンティニュイティ
- 計画済メンテナンスの安全なドレイニングAPI

関連項目:

Oracle RACを使用するサーバー側の構成は、『Oracle Real Application Clusters管理およびデプロイメント・ガイド』を参照してください。この章では、Oracle JDBCドライバによるFANイベントのサポートを使用している場合にアプリケーションが実行する必要がある、クライアント側の構成ステップについてのみ説明します。

29.2 計画済メンテナンスの安全なドレイニングAPI

計画済Oracle RACメンテナンスの場合、JDBCドライバでは安全なドレイニングを行うAPIのリストがサポートされています。これは、サードパーティのJava接続プールで追加のハンドシェイクまたは統合作業に必要になります。これらのAPIはドレイニングポイントとして機能し、ここでドライバは、アプリケーションでエラーが表示されずに、計画済メンテナンスで影響を受ける接続を安全にクローズします。次に、ドライバFANがサポートしている、安全なドレイニングAPIのリストを示します。

- java.sql.Connection.isValid(int timeout)
- oracle.jdbc.OracleConnection.pingDatabase()
- oracle.jdbc.OracleConnection.pingDatabase(int timeout)
- oracle.jdbc.OracleConnection.endRequest()
- すべての標準JDBCおよびOracle JDBC拡張機能のEXECUTE***がStatement、PreparedStatementおよびCallableStatementインタフェース上でコールされます

標準JDBCおよびOracle JDBC拡張機能のEXECUTE***コールの場合、実行したSQLコマンド文字列に次のSQLヒントがSQL文字列内で最初のコメントのないトークンとして含まれている必要があります。

```
/*+ CLIENT_CONNECTION_VALIDATION */
```

修飾SQLが接続の検証SQLとして処理されます。たとえば:

```
/*+ CLIENT_CONNECTION_VALIDATION */ SELECT 1 FROM DUAL
```

通常、サードパーティの接続プールは、これらのAPIにコールを実行します。このような起動での無効な接続の検出時に、アプリケーションにエラーが表示されないように、サードパーティの接続プールはプールから関連する接続をクローズし削除します。アプリケーション自体がこれらのAPIをコールすると、アプリケーションは基礎となる接続をアクティブに検証し、検出された無効な接続をクローズして削除します。

29.3 FANイベントのサポートのためのOracle JDBCドライバのインストールおよび構成

Oracle JDBCドライバは、接続するデータベース・サーバーをチェックすることにより、FANイベント用にOracle JDBCサポートを有効にするかどうか、および次の必要なJARファイルがアプリケーション環境およびOracle JDBCドライバで使用可能かどうかを自動的に判定します。

- simplefan.jarファイルおよびons.jarファイル

  次のリンクから12.2.0.1バージョンのsimplefan.jarファイルおよびons.jarファイルをインストールし、これらを

CLASSPATHに含める必要があります。

http://www.oracle.com/technetwork/database/application-development/jdbc/jdbc-ucp-122-3110062.html

いずれかのファイルが存在しないか、ドライバがこれをロードできない場合、この機能は無効になります。サードパーティの接続プールで使用する場合、これらのJARファイルは同じ場所に配置する必要があります。ここで接続プールがドライバのJARファイルを取得しロードします。

- Oracle JDBCデータソース

  JDBC接続の取得時に、oracle.jdbc.pool.OracleDataSourceまたはoracle.jdbc.OracleDriverなどの通常の同じOracle JDBCデータソースを使用できます。サードパーティの接続プールとともに使用する場合、アプリケーションではこれらのクラスを接続プールの接続ファクトリとして指定する必要があります。

この機能を明示的に無効にする場合、アプリケーションではoracle.jdbc.fanEnabledプロパティをFALSEに設定できます。このプロパティは、システム・プロパティとしても接続プロパティとしても使用できます。ユニバーサル接続プール(UCP)またはWebLogic ServerのアクティブなGridLink (AGL)を使用するアプリケーションの場合、このプロパティはデフォルトでFALSEに設定されます。それ以外の場合は、デフォルト値はTRUEです。

ノート:

- JDBC ドライバが FAN イベントのサポートを自動的に有効化する場合、simplefan.jar および ons.jar ファイルが CLASSPATH に存在する状態で getConnection メソッドをコールすると、java.lang.IllegalArgumentException などの例外がスローされる可能性があります。これを避けるには、次のいずれかを実行します。
  - CLASSPATH から simplefan.jar または ons.jar を削除します。
  - oracle.jdbc.fanEnabled プロパティを FALSE に設定して、この機能を明示的に無効化します。
- oracle.jdbc.fanEnabled プロパティを TRUE に設定しても、この機能が他の要因に依存しているために、FAN イベントの Oracle JDBC サポートが有効にならない場合があります。

JDBCドライバでは、Oracle FANイベントをサポートするために、サードパーティの接続プールに対して最小限の構成の変更またはコードの変更が必要になります。構成の変更またはコードの変更の必要がない接続プールの場合、次の基準のいずれかを満たしていると想定されます。

- プールには、プールのチェックアウト時にJDBC接続を検証するための構成オプションがある。
- プールはjavax.sql.PooledConnectionを使用し、javax.sql.ConnectionPoolDataSource実装にプラグインするための構成オプションがある。このような接続プールは、接続が戻るときにクローズした物理接続または無効な物理接続をチェックできることも想定されます。

一部のサードパーティのJava接続プールの接続検証オプションのいくつかを次に示します。これらのオプションの大部分は、検証APIではなく、SQLに基づいています。

Java接続プール	接続検証オプション
Oracle WebLogic 汎用データソースおよび MDS データソース	TestConnectionsOnReserve、TestConnectionsOnRelease、TestConnectionsOnCreate
IBM WebSphere	PreTest 接続
RedHat JBoss	check-valid-connection-sql
Apache TomCat	TestonBorrow、TestonRelease

Oracle RACサーバー・リリース11gとともにFANイベント機能のOracle JDBCによるサポートを使用する場合、アプリケーションでは、oracle.jdbc.fanONSConfigシステム・プロパティを介してOracle RAC FANのリモートONS構成文字列を明示的に設定する必要があります。プロパティの値および書式は、UCP高速接続フェイルオーバー(FCF)と同様です。

関連項目:

Universal Connection Pool開発者ガイド

29.4 計画済メンテナンスの場合のOracle JDBCドライバによるFANのサポートの例

次の例では、Oracle RACの計画済メンテナンスで、JDBC Oracle FANサポートを有効にして使用する通常の方法を示します。次の手順の実行後、アプリケーションでは計画済メンテナンス時に例外を受信しません。

1. Oracle JDBCドライバをリリース12.2.0.1にアップグレードして、ojdbc8.jarファイルを使用します。
2. 12.2.0.1バージョンのons.jarファイルおよびsimplefan.jarファイルをインストールして使用します。
3. oracle.jdbc.pool.OracleDataSourceクラスを使用して、物理接続を取得するか、サードパーティのJava接続プールでこのクラスを接続ファクトリとして構成します。後者の場合、特定のプール・プロパティを設定して接続検証を有効にする必要があります。

   Oracle RACリリース11gに対して実行している場合に、オプションで、リモートONSを構成するようにシステム・プロパティoracle.jdbc.fanONSConfigを指定します。
4. アプリケーションは、Oracle RAC上のローリング・アップグレードと同様に、計画済メンテナンス・アクティビティを実行する準備ができるまで実行されます。計画済メンテナンス時に、使用パターンに基づいたサービスごとに、DBAは拡張12.2 srvctlインタフェースを使用して、次のアクティビティを実行します。
   - -f (force)を指定しないで、次にアップグレードするインスタンスでサービスを再配置または停止します
   - このサービスへのすべての接続がドライバFANでドレインされるまで待機します
   - タイムアウトに達したときに、定義した停止モードでセッションを切断します(トランザクションをお薦めします)
   - すべてのサービスを再配置または停止したときに、インスタンスを停止してアップグレードまたはパッチを適用しま

す

- インスタンスを再起動し、サービスが停止している場合はサービスを再起動します
- すべてのインスタンスがアップグレード/パッチ適用されるまで、繰り返します

30 透過的アプリケーション・フェイルオーバー

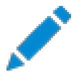
この章の構成は、次のとおりです。

- 透過的アプリケーション・フェイルオーバーの概要
- フェイルオーバー・タイプ・イベント
- TAFコールバック
- Java TAFコールバック・インタフェース
- TAFと高速接続フェイルオーバーの比較

30.1 透過的アプリケーション・フェイルオーバーの概要

透過的アプリケーション・フェイルオーバー(TAF)は、Java Database Connectivity(JDBC)Oracle Call Interface(OCI)ドライバの機能の1つです。この機能により、接続先のデータベース・インスタンスに障害が発生した場合でも、アプリケーションはデータベースに自動的に再接続できます。この場合、アクティブ・トランザクションはロールバックされます。

接続先インスタンスで障害が発生したり、このインスタンスがシャットダウンしたりすると、クライアント側の接続が無効になり、その接続を使用しようとしたコール元に例外が表示されます。TAFにより、アプリケーションは、事前に構成した2次インスタンスに透過的に再接続し、最初の元のインスタンスで確立したのと同じ接続を新たに作成します。つまり、この接続プロパティは、直前の接続のプロパティと同じです。接続がどのように切断されても関係ありません。



ノート:

- TAF は常にアクティブなので、設定する必要はありません。
- TAF は LOB 型および XML 型でサポートされていません。

30.2 フェイルオーバー・タイプ・イベント

OracleOCIFailoverインタフェースで発生する可能性があるフェイルオーバー・イベントは、次のとおりです。

- FO_SESSION

  tnsnames.oraファイルCONNECT_DATAフラグのFAILOVER_MODE=SESSIONに相当します。この結果、サーバー側ではユーザー・セッションのみが再度認証され、一方OCIアプリケーションのオープン・カーソルは再実行する必要があります。

- FO_SELECT

  tnsnames.oraファイルCONNECT_DATAフラグのFAILOVER_MODE=SELECTに相当します。サーバー側でユーザー・セッションが認証されるのみでなく、OCIのオープン・カーソルもフェッチを続行できます。これは、クライアント側のロジックで各オープン・カーソルのフェッチ状態が保持されることを意味します。

- FO_NONE

  tnsnames.oraファイルCONNECT_DATAフラグのFAILOVER_MODE=NONEに相当します。これはデフォルトで、フェイルオーバー機能は使用されません。フェイルオーバーが実行されないように、明示的に指定することもできます。さらに、

FO_TYPE_UNKNOWNは、OCIドライバから不正なフェイルオーバー型が戻されたことを意味します。

- FO_BEGIN

  フェイルオーバーで失われた接続が検出され、フェイルオーバーが開始されることを意味します。

- FO_END

  フェイルオーバーの正常な完了を示します。

- FO_ABORT

  フェイルオーバーが失敗し、再試行できないことを意味します。

- FO_REAUTH

  ユーザー・ハンドルが再認証されたことを意味します。

- FO_ERROR

  フェイルオーバーが一時的に失敗したが、アプリケーションでエラーを処理してフェイルオーバーを再試行できることを意味します。sleepメソッドを発行した後、値FO_RETRYを戻して再試行するのが、エラー処理の一般的な方法です。

- FO_RETRY

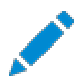
  アプリケーションでフェイルオーバーを再試行することを意味します。

- FO_EVENT_UNKNOWN

  不正なフェイルオーバー・イベントを意味します。

30.3 TAFコールバック

TAFコールバックは、1つのデータベース接続に障害が起きた場合に、別のデータベース接続にフェイルオーバーする際に使用されます。TAFコールバックは、フェイルオーバーが発生したときに登録されるコールバックです。コールバックは、フェイルオーバー中にイベントが発生したことをJDBCアプリケーションに通知するためにコールされます。アプリケーション側でもフェイルオーバーをある程度制御できます。



ノート:

コールバックの設定はオプションです。

30.4 Java TAFコールバック・インタフェース

OracleOCIFailoverインタフェースには、次の型とイベントをサポートするcallbackFnメソッドが含まれます。

```
public interface OracleOCIFailover{
// Possible Failover Types
public static final int FO_SESSION = 1;
public static final int FO_SELECT  = 2;
public static final int FO_NONE  = 3;
public static final int;
// Possible Failover events registered with callback
public static final int FO_BEGIN   = 1;
public static final int FO_END     = 2;
public static final int FO_ABORT   = 3;
public static final int FO_REAUTH  = 4;
```

```
public static final int FO_ERROR  = 5;
public static final int FO_RETRY  = 6;
public static final int FO_EVENT_UNKNOWN = 7;
public int callbackFn (Connection conn,
                       Object ctxt, // ANy thing the user wants to save
                       int type, // One of the possible Failover Types
                       int event ); // One of the possible Failover Events
```

FO_ERRORイベントの処理

新しい接続にフェイルオーバーするときにエラーが発生した場合、JDBCアプリケーションはそのフェイルオーバーを再試行できます。一般に、アプリケーションは一度スリープ状態に入った後、コールバックでFO_RETRYを戻すことで、無限または一定の時間のみ再試行します。

FO_ABORTイベントの処理

登録されたコールバックは、FO_ERRORイベントが渡された場合、FO_ABORTイベントを戻す必要があります。

30.5 TAFと高速接続フェイルオーバーの比較

透過的アプリケーション・フェイルオーバー(TAF)と高速接続フェイルオーバーは、次の点で異なります。

- アプリケーションレベルでの接続の再試行

  TAFでは、接続の再試行はOCI/Netレイヤーでのみサポートしています。高速接続フェイルオーバーは、アプリケーションレベルでの接続の再試行をサポートしています。これにより、アプリケーションは接続のフェイルオーバーへの対応を制御できます。アプリケーションは接続を再試行するか、または例外を再発生させるかを選択できます。

- ユニバーサル接続プールとの統合

  TAFは、接続単位でネットワーク・レベルで動作するため、接続キャッシュに障害の発生を通知できません。高速接続フェイルオーバーはユニバーサル接続プールと高度に統合されているため、Connection Cache Managerはキャッシュを管理して高可用性を維持できます。たとえば、障害が発生した接続は、キャッシュ内で自動的に無効になります。

- イベントベース

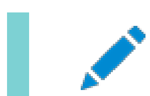
  高速接続フェイルオーバーは、Oracle RACイベント・メカニズムに基づいています。したがって、高速接続フェイルオーバーは効率的で、アクティブおよび非アクティブな接続の両方で障害をすばやく検出できます。

- ロード・バランシングのサポート

  高速接続フェイルオーバーは、接続のUPイベント・ロード・バランシング、およびアクティブなOracle RACインスタンス間での実行時作業要求分散をサポートしています。

関連項目:

『Oracle Universal Connection Pool for JDBC開発者ガイド』



ノート:

TAF と高速接続フェイルオーバーを同じアプリケーションで使用しないことをお薦めします。

31 単一クライアント・アクセス名

単一クライアント・アクセス名(SCAN)は、クラスタで実行中のOracle Databaseにアクセスするために、クライアントに単一の名前を提供するOracle Real Application Clusters (Oracle RAC)の機能です。この章では、SCANに関連する次の概念について説明します。

- 単一クライアント・アクセス名の概要
- SCANを使用したデータベースの構成について
- 最大可用性アーキテクチャ環境でのSCANの使用
- Oracle Connection ManagerでのSCANの使用

31.1 単一クライアント・アクセス名の概要

SCANは、ドメイン・ネーム・サービス(DNS)またはグリッド・ネーミング・サービス(GNS)のいずれかにある、1つ以上3つ以下のIPアドレスに登録されたドメイン名です。GNSおよびDynamic Host Configuration Protocol (DHCP)を使用している場合、Oracle Clusterwareでは、クラスタの構成時に指定されるSCAN名の仮想IP (VIP)アドレスが構成されます。ノードVIPおよび3つのSCAN VIPは、GNSを使用している場合、DHCPサーバーから取得されます。

関連項目:

GNSの詳細は、『Oracle Clusterware管理およびデプロイメント・ガイド』を参照してください。

新しいサーバーがクラスタに追加されると、Oracle Clusterwareでは、必要なVIPアドレスはDHCPサーバーから動的に取得されてクラスタ・リソースが更新され、GNSを介してサーバーにアクセスできるようになります。SCANを使用する利点は、クラスタ内でノードを追加または削除しても、クライアントの接続情報を変更する必要がないことです。クラスタにアクセスするための単一名を保持することで、クライアントではEZConnectクライアントとシンプルなJDBC thin URLを使用して、クラスタ内のアクティブ・サーバーに関係なく、クラスタで実行中のあらゆるデータベースにアクセスできます。SCANにより、データベースへのクライアント接続のロード・バランシングおよびフェイルオーバーが実現します。SCANは、クラスタ内のデータベースのクラスタ別名として機能します。

31.2 SCANを使用したデータベースの構成について

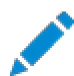
SCANは、データベース構成に不可欠です。そのため、標準のOracleツールを使用してデータベースを作成すると、REMOTE_LISTENERパラメータはデフォルトでSCANに設定されます。これにより、インスタンスがリモート・リスナーとしてSCANリスナーに登録され、そのインスタンスによって提供されているサービスおよび現在のロードに関する情報や、そのインスタンスに送られる着信接続数に関する推奨が提供されます。

こうした場合、LOCAL_LISTENERパラメータをノードVIPに設定する必要があります。完全修飾ドメイン名が必要な場合は、LOCAL_LISTENERパラメータに完全修飾ドメイン名を必ず設定します。デフォルトでは、クラスタの構成時に、クラスタ内の各ノードでノード・リスナーが作成されます。Oracle Grid Infrastructureでは、ノード・リスナーはOracle Grid Infrastructure

ホームから実行され、指定されたポートを使用してノードVIPをリスニングします。デフォルトのポートは1521です。

以前のバージョンのデータベースとは異なり、REMOTE_LISTENERパラメータを、HOST=sales1-scanなど、アドレス・リスト・エントリでホストをSCANに解決するサーバー側のTNSNAMES別名に設定することはお薦めしません。かわりに、次の表に示すように、簡略化されたSCAN:port構文を使用することをお薦めします。この表では、LOCAL_LISTENERおよびREMOTE_LISTENERの一般的な設定を示します。

名前	型	値
LOCAL_LISTENER	string	(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)(HOST=localhost)(PORT=5221))))
REMOTE_LISTENER	string	example.us.oracle.com:5221



ノート:

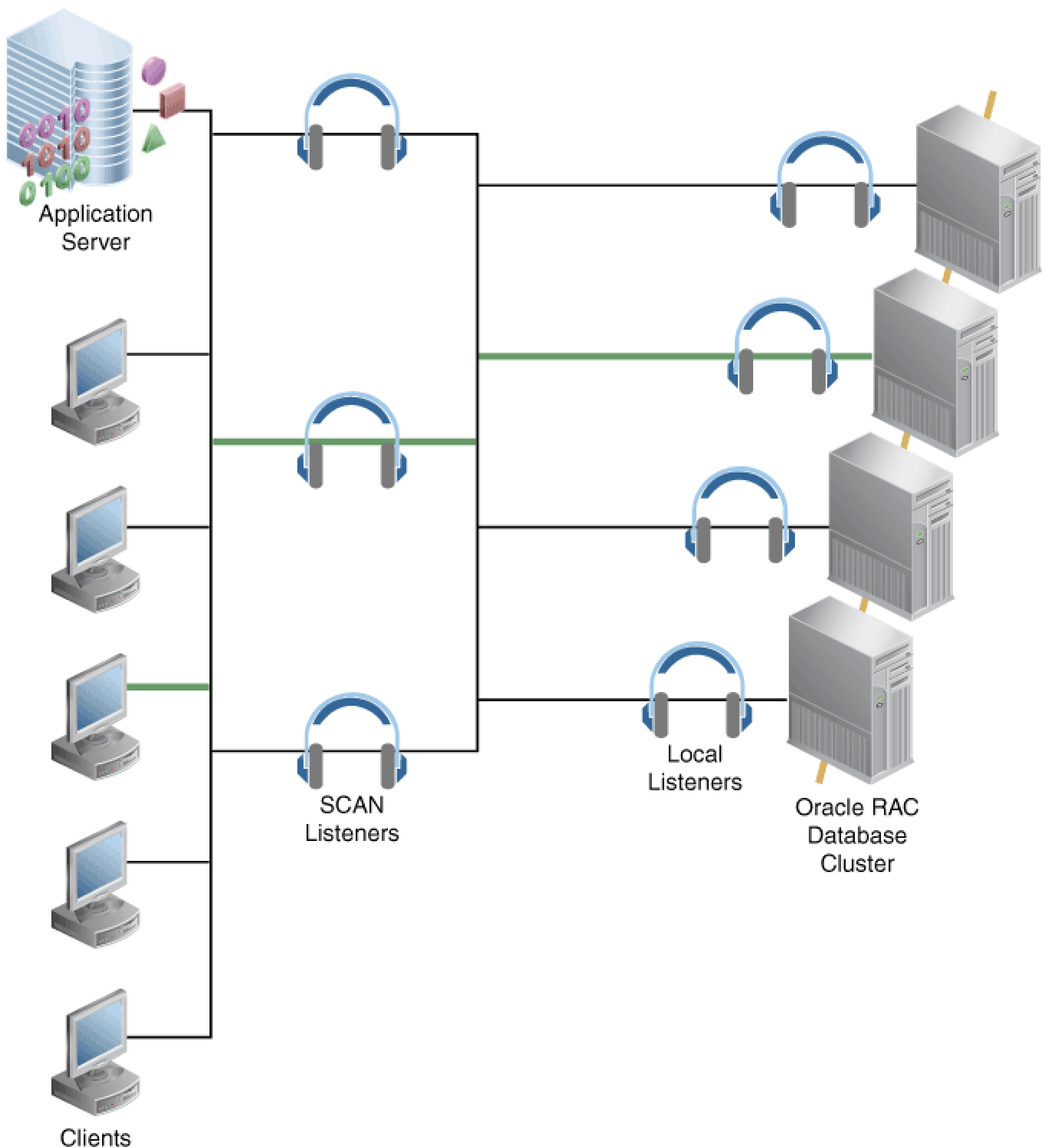
簡易接続ネーミング方法を使用している場合は、クライアント名の解決の参照に使用されるネーミング方法の順序を指定するときに、SQLNET.ORA ファイルを変更して EZCONNECT が必ずリストに含まれるようにする必要があります。

31.3 SCANを使用した接続ロード・バランシングの動作

Oracle SQL*Netを使用したクライアント接続の場合、DNSによってSCAN名を解決することで、クライアントは3つのIPアドレスを受け取ります。次に、クライアントは、DNSから受け取ったリストを調べ、リスト内のIPアドレスの1つを使用して接続しようとします。エラーを受け取ると、クライアントはユーザーまたはアプリケーションにエラーを戻す前に別のアドレスに接続しようとします。これは、クライアント接続文字列にアドレス・リストが指定されているときの、以前のリリースのデータベースにおけるクライアント接続フェイルオーバーの動作に似ています。

SCANリスナーが接続要求を受け取ると、SCANリスナーは最もロードされていない、要求されたサービスを提供しているインスタンスをチェックします。次に、最もロードされていないインスタンスが実行中のノードのローカル・リスナーに接続要求をリダイレクトします。続いて、クライアントにローカル・リスナーのアドレスが付与されます。最後に、このローカル・リスナーがデータベース・インスタンスへの接続を作成します。

図31-1 SCANを使用した接続ロード・バランシング



例

この例では、デフォルトのTNSNAMES.oraファイルを使用するOracleクライアントを想定しています。

```
ORCLservice =(DESCRIPTION =(ADDRESS = (PROTOCOL = TCP)(HOST = sales1-scan.example.com)(PORT =
1521))(CONNECT_DATA =(SERVER = DEDICATED)(SERVICE_NAME = MyORCLservice)))
```

31.4 バージョンと下位互換性

SCANを正しく使用してクラスタ内のOracle RACデータベースに接続できるかどうかは、次の2つの要因によって決まります。

- SCANを認識して使用するクライアントの能力

- データベースのREMOTE_LISTENERパラメータ設定の正しい構成

データベースに接続するOracleクライアントのバージョンと使用されるOracle DatabaseのバージョンがいずれもOracle Database 11g リリース2であり、前述の項で説明されているようにデフォルト構成を使用する場合は、通常、システムに変更を加える必要はありません。

Oracleクライアントのバージョンとクライアントが接続しているOracle Databaseのバージョンがいずれも Oracle Database 11g リリース2より前である場合は、通常、システムに変更を加える必要はありません。この場合、クライアントはクラスタのノードVIPに解決するTNS接続記述子を使用する一方で、Oracle DatabaseはノードVIPを指すREMOTE_LISTENERエントリを使用します。この構成の短所は、SCANが使用されないことです。そのため、バックエンドでクラスタが変更されるたびに、クライアントを変更する必要があります。

Oracle Database 11g リリース2を使用しているのに、クライアントが以前のバージョンのデータベース上にある場合は、OracleクライアントまたはOracle DatabaseのREMOTE_LISTENERパラメータ設定、あるいはその両方を適宜変更する必要があります。このような場合、次のケースを考慮する必要があります。

表31-1 OracleクライアントおよびOracle DatabaseのSCANに関するバージョン互換性

Oracleクライアントのバージョン	**Oracle Databaseのバージョン**	**備考**
Oracle Database 11 g リリース 2	Oracle Database 11 g リリース 2	変更は不要です。
Oracle Database 11 g リリース 2	Oracle Database 11g リリース 2 より前の Oracle Database のバージョン	SCAN VIP をホストとして REMOTE_LISTENER パラメータに追加します。
Oracle Database 11g リリース 2 より前の Oracle Database のバージョン	Oracle Database 11 g リリース 2	クライアントの TNSNAMES.ora ファイルを更新して SCAN VIP を含めます。Database Upgrade Assistant (DBUA)を使用してデータベースを 11g リリース 2 より前のデータベースからアップグレードする場合、DBUA によって REMOTE_LISTENER パラメータはノード VIP および SCAN を指すように構成されます。
Oracle Database 11g リリース 2 より前の Oracle Database のバージョン	Oracle Database 11g リリース 2 より前の Oracle Database のバージョン	SCAN を使用する場合(推奨)は、SCAN VIP をホストとして REMOTE_LISTENER パラメータに追加し、クライアントの TNSNAMES.ora ファイルを更新して SCAN VIP を含めます。それ以外の場合、変更は不要です。

Oracle Database 11g リリース2より前のクライアントを使用している場合は、SCANの利点を十分に活用できません。これ

は、DNSから戻されるSCAN用の3つのIPアドレスのセットをOracleクライアントが処理できないためです。かわりに、リストに戻された1つ目のアドレスにのみ接続しようとし、他の2つのアドレスを無視します。この特定のIPアドレスをリスニングしているSCANリスナーが使用できない場合、またはIPアドレス自体が使用できない場合は、接続できません。Oracle Database 11g リリース2より前のクライアントでロード・バランシングおよび接続フェイルオーバーが確実に行われるようにするには、クライアントのTNSNAMES.oraファイルを更新して、3つのアドレス行がクライアントで使用され、各アドレス行がSCAN VIPのいずれかに解決されるようにする必要があります。次の例に、Oracle Database 11g リリース2より前のクライアントのサンプルTNSNAMES.oraファイルを示します。

```
sales.example.com =(DESCRIPTION=
(ADDRESS_LIST= (LOAD_BALANCE=on)(FAILOVER=ON)
(ADDRESS=(PROTOCOL=tcp)(HOST=133.22.67.192)(PORT=1521))
(ADDRESS=(PROTOCOL=tcp)(HOST=133.22.67.193)(PORT=1521))
(ADDRESS=(PROTOCOL=tcp)(HOST=133.22.67.194)(PORT=1521)))
(CONNECT_DATA=(SERVICE_NAME= salesservice.example.com)))
```

31.5 最大可用性アーキテクチャ環境でのSCANの使用

最大可用性アーキテクチャ(MAA)環境を実装し、その環境でOracle Data Guardを使用して同期されるプライマリ・データベースとスタンバイ・データベースの両方にOracle RACを使用している場合、SCANを使用すると、現在アクティブなデータベースがプライマリ・データベースであるかスタンバイ・データベースであるかに関係なく、クライアントがデータベースへの接続に使用できるTNSNAMES構成を簡略化できます。

この簡略化された構成を使用するために、Oracle Database 11g リリース2には、個々のクライアントの接続文字列に使用できる次の2つのSQL*Netパラメータが導入されています。

- CONNECT_TIMEOUTパラメータ

  クライアントがOracle DatabaseへのOracle Net接続を確立するときのタイムアウト時間を秒数で指定します。このパラメータは、SQLNET.ORAファイルのSQLNET.OUTBOUT_CONNECT_TIMEOUTパラメータをオーバーライドします。

- RETRY_COUNTパラメータ

  接続試行を終了するまでに、ADDRESS_LISTを反復する回数を指定します。

この2つのパラメータを使用すると、プライマリ・サイトとスタンバイ・サイトの両方のSCANをクライアント接続文字列で使用できます。また、ランダムに選択されたアドレスが現在アクティブではないサイトを指している場合、クライアントが不当に長時間待機する前に、タイムアウトによって接続要求をフェイルオーバーすることができます。次の例に、MAA環境用のサンプルTNSNAMES.ORAエントリを示します。

```
sales.example.com =(DESCRIPTION= (CONNECT_TIMEOUT=10)(RETRY_COUNT=3)
(ADDRESS_LIST= (LOAD_BALANCE=on)(FAILOVER=ON)
(ADDRESS=(PROTOCOL=tcp)(HOST=sales1-scan)(PORT=1521))
(ADDRESS=(PROTOCOL=tcp)(HOST=sales2-scan)(PORT=1521)))
(CONNECT_DATA=(SERVICE_NAME= salesservice.example.com)))
```

31.6 Oracle Connection ManagerでのSCANの使用

Oracle RACデータベースでOracle Connection Manager (CMAN)を使用する場合は、Oracle RACインスタンスのREMOTE_LISTENERパラメータにCMANサーバーを含めて、CMANサーバーでロード・バランシング関連の情報を受け取り、使用

可能なインスタンスに対して接続のロード・バランシングを実行できるようにする必要があります。これを実行する最も簡単な方法は、クライアントがCMANを経由して接続するデータベースのREMOTE_LISTENERパラメータにCMANサーバーをエントリとして追加します。また、クライアントのTNSNAMES接続記述子からSCANを削除してCMANサーバーを構成する必要があります。次の例に、CMANを使用するときのサーバー側のTNSNAMES.oraのサンプル・エントリを示します。

```
SQL> show parameters listener
NAME                 TYPE         VALUE
-------------------------- ----------- ------------------------------
listener_networks    string
local_listener       string       (DESCRIPTION=(ADDRESS_LIST=
                                  (ADDRESS=(PROTOCOL=TCP)
                                  (HOST=148.87.58.109)(PORT=1521))))
remote_listener      string       stscan3.oracle.com:1521,(DESCRIPTION=
                                  (ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)
                                  (HOST=CMANserver)(PORT=1521))))
```

関連項目:

CMANサーバーの構成の詳細は、『Oracle Database Net Servicesリファレンス』を参照してください。

第VII部 トランザクション管理

この部では、Oracle Java Database Connectivity(JDBC)でのトランザクション管理について説明します。分散トランザクションのOracle JDBC実装についても説明します。

この部の構成は、次のとおりです。

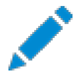
- 分散トランザクション

32 分散トランザクション

この章では、分散トランザクションのOracle Java Database Connectivity(JDBC)実装について説明します。これは多フェーズ・トランザクションで、しばしば複数のデータベースを使用し、協調してコミットされる必要があります。Java固有でなく分散トランザクションの一般的な標準であるXAについての関連説明もあります。

次の内容について説明します。

- 分散トランザクションについて
- XAコンポーネント
- エラー処理と最適化
- 分散トランザクションの実装について
- Oracle JDBCドライバのネイティブXA



ノート:

この章では、javax パッケージで使用可能な、JDBC 2.0 Standard Extension Application Program Interface (API)と呼ばれる、JDBC 2.0 Optional Package の機能について説明します。

分散トランザクションの詳細および一般情報は、JDBC 2.0 Optional PackageおよびJava Transaction API(JTA)の仕様を参照してください。

32.1 分散トランザクションについて

この項の内容は次のとおりです:

- 分散トランザクションの概要
- 分散トランザクションのコンポーネントおよびシナリオ
- 分散トランザクションの概念
- ローカル・トランザクションとグローバル・トランザクションの切替えについて
- Oracle XAパッケージ

32.1.1 分散トランザクションの概要

分散トランザクション(グローバル・トランザクションとも呼ばれる)は、複数の関連トランザクションの組合せで、協調して管理する必要があります。分散トランザクションを構成するトランザクションは、同じデータベース内に存在することもありますが、通常は異なるデータベースに存在し、多くの場合は異なる場所に存在します。分散トランザクションを構成する各トランザクションをトランザクション・ブランチと呼びます。

たとえば、分散トランザクションには、ある銀行のある口座から、別の銀行の口座への送金があります。両方の処理が正常に完了する保証がなければ、トランザクションはコミットできません。

JDBCでは、分散トランザクション機能が接続プーリング機能の最上位に構築されます。この分散トランザクション機能は、分散トランザクションのオープンXA標準にも基づいて構築されます。XAはX/Open標準の一部で、Java固有ではありません。

データベース・リソースに接続するには、JDBCを使用します。ただし、複数のデータベースに対するすべての変更を1回のトランザクションで更新するには、JTAグローバル・トランザクションでJDBC接続を使用する必要があります。トランザクションにデータベースSQL更新を含める処理は、データベース・リソースの参加と呼ばれます。

32.1.2 分散トランザクションのコンポーネントおよびシナリオ

この項の以降の部分を読むには、次のポイントを覚えておくと役に立ちます。

- 分散トランザクション・システムは、通常、標準JTA機能を実装するソフトウェア・コンポーネントなど、外部のトランザクション・マネージャを使用して、個々のトランザクションを協調させます。

  多くのベンダーが、XA準拠のJTAモジュールを提供しています。たとえば、Oracleからは、Oracle9i Application ServerおよびOracle Application Server 10gでJTAが提供されています。

- XA機能は、通常、クライアント・アプリケーションから孤立しています。クライアント・アプリケーションではなく、アプリケーション・サーバーなど、中間層環境に実装されます。

  多くの場合、アプリケーション・サーバーとトランザクション・マネージャはともに中間層に位置します。同時に、アプリケーション・コードの一部も中間層に位置します。

- この項の説明は、主に、中間層の開発者を対象にしています。
- 分散トランザクションの説明では、リソース・マネージャという用語が頻繁に使用されます。リソース・マネージャとは、単に、データまたはその他のリソースを管理するエンティティです。この章では、データベースの意味で使用されています。

ノート:

JTA 機能を使用するには、`CLASSPATH` 環境変数に `jta.jar` が含まれている必要があります。このファイルは、`ORACLE_HOME/jlib` にあります。このファイルは JDBC 製品に含まれています。

32.1.3 分散トランザクションの概念

XA機能を使用する場合、トランザクション・マネージャが、XAリソース・インスタンスを使用して各トランザクション・ブランチを準備し、協調させ、すべてのトランザクション・ブランチを適切にコミットまたはロールバックします。

XA機能には、次のキー・コンポーネントが含まれています。

- XAデータソース

  接続プーリング・データソースおよびその他のデータソースの拡張機能で、概念および機能の面で似ています。

  分散トランザクションで使用されるリソース・マネージャごとに1つのXAデータソース・インスタンスがあります。通常、XAデータソース・インスタンスは、中間層ソフトウェアで作成します。

  XAデータソースは、XA接続を作成します。

- XA接続

プーリングされた接続の拡張機能で、概念および機能の面で似ています。XA接続は、物理的なデータベース接続をカプセル化します。個々の接続インスタンスは、これら物理接続の一時的なハンドルです。

1つのXA接続インスタンスは、1つのOracleセッションに対応します。ただし、1つのセッションは、プーリングされた接続インスタンスのように、複数の論理接続インスタンスで順に使用できます。

通常、XA接続インスタンスは、XAデータソース・インスタンスから、中間層ソフトウェアで取得します。分散トランザクションが、同じデータベースの複数セッションにかかわる場合、単一XAデータソース・インスタンスから複数のXA接続インスタンスを取得できます。

XA接続は、OracleXAResourceインスタンスおよびJDBC接続インスタンスを作成します。

- XAリソース

  分散トランザクションのトランザクション・ブランチを協調させるために、トランザクション・マネージャで使用されます。

  通常、OracleXAResourceインスタンスは、各XA接続インスタンスから1つずつ、中間層ソフトウェアで取得します。OracleXAResourceインスタンスとXA接続インスタンスの間には1対1の相関関係があります。同様に、OracleXAResourceインスタンスとOracleセッションの間には、1対1の相関関係があります。

  一般的なシナリオでは、中間層コンポーネントがOracleXAResourceインスタンスをトランザクション・マネージャに渡します。トランザクション・マネージャは、これを使用して分散トランザクションを協調させます。

  各OracleXAResourceインスタンスは1つのOracleセッションに対応しています。したがって、OracleXAResourceインスタンスに関連付けられたトランザクション・ブランチでアクティブになれるものは常に1つのみです。ただし、他のトランザクション・ブランチを保留しておくことはできます。

  各OracleXAResourceインスタンスには、そのOracleXAResourceインスタンスに関連付けられたセッションで実行中のトランザクション・ブランチの操作を、開始、終了、準備、コミットまたはロールバックする機能があります。

  準備ステップは、2フェーズ・コミット操作の最初のステップです。トランザクション・マネージャは、各OracleXAResourceインスタンスにPREPAREを発行します。トランザクション・マネージャは、各トランザクション・ブランチの操作が正常に準備されたことを確認すると、各OracleXAResourceインスタンスにCOMMITを発行し、すべての変更をコミットします。

- トランザクションID

  トランザクション・ブランチを識別するために使用されます。各トランザクションIDには、トランザクション・ブランチIDコンポーネントと分散トランザクションIDコンポーネントが含まれています。これにより、ブランチが分散トランザクションに関連付けられます。ある分散トランザクションに関連付けられたOracleXAResourceインスタンスにはすべて、同じ分散トランザクションIDコンポーネントが含まれているトランザクションIDが設定されます。

- OracleXAResource.ORATRANSLOOSE

  トランザクションIDがxidの疎結合トランザクションを起動します。

32.1.4 ローカル・トランザクションとグローバル・トランザクションの切替えについて

アプリケーションは、ローカル・トランザクションとグローバル・トランザクションの間で接続を共有できます。また、アプリケーションは、ローカル・トランザクションとグローバル・トランザクションの間で接続を切り替えることもできます。

接続は必ず次のモードのいずれかです。

- NO_TXN

  この接続をアクティブの状態で使用しているトランザクションはありません。

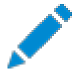
- LOCAL_TXN

  自動コミットがオフまたは無効になっているローカル・トランザクションが、アクティブの状態でこの接続を使用しています。

- GLOBAL_TXN

  グローバル・トランザクションが、アクティブの状態でこの接続を使用しています。

各接続では、その接続で実行されている操作によって、これらのモードが自動的に切り替わります。接続は、インスタンス化されたときは常にNO_TXNモードです。



ノート:

各モードは Oracle Database に関連付けられた JDBC ドライバによって内部的に保持されます。

表32-1では、接続モードの推移のルールについて説明します。

表32-1 接続モードの推移

現在のモード	NO_TXNに切り替わるとき	LOCAL_TXNに切り替わるとき	GLOBAL_TXNに切り替わるとき
NO_TXN	N/A	自動コミット・モードが false で、Oracle Data Manipulation Language(DML)文が実行された場合	現在の接続を提供した XAconnection から取得された XAResource で start メソッドがコールされた場合。
LOCAL_TXN	次のいずれかの場合 ● Oracle Data Definition Language(DDL)文が実行された場合 ● commit がコールされた場合 ● rollback がパラメータなしでコールされた場合	N/A	現在の接続を提供した XAconnection から取得された XAResource で start メソッドがコールされた場合。この機能は、Oracle Database 12c リリース 1 (12.1.0.2)以上で使用できます。
GLOBAL_TXN	この接続でオープンしているグローバル・トランザクション内で、この接続を提供した XAconnection から取得された XAResource で end がコールさ	なし	N/A

現在のモード	NO_TXNに切り替わるとき	LOCAL_TXNに切り替わるとき	GLOBAL_TXNに切り替わるとき
	れた場合		

前述のルールのいずれにも該当しない場合、モードは変わりません。

操作に関するモードの制限

現在の接続モードによって、トランザクション内で有効な操作が制限されます。

- LOCAL_TXNモードの場合、アプリケーションはprepare、commit、rollback、forgetまたはendをXAResourceでコールできません。これらを起動すると、XAExceptionが発生します。
- GLOBAL_TXNモードの場合、アプリケーションではcommit、rollback、rollback(Savepoint)、setAutoCommit(true)またはsetSavepointをjava.sql.Connectionでコールすることや、OracleSetSavepointまたはoracleRollbackをoracle.jdbc.OracleConnectionでコールすることはできません。これらを起動すると、SQLExceptionが発生します。

ノート:

このモードに関する制限のエラー・チェックは、トランザクションとセーブポイント API に関する標準のエラー・チェックに追加されます。

32.1.5 Oracle XAパッケージ

Oracleからは、XA標準に従い分散トランザクション機能を実装するクラスを含む、次の3つのパッケージが提供されます。

- oracle.jdbc.xa
- oracle.jdbc.xa.client
- oracle.jdbc.xa.server

XAデータソース、XA接続およびXAリソースのための各クラスは、clientパッケージとserverパッケージのどちらにもあります。それぞれの抽象クラスは、最上位レベルのパッケージにあります。OracleXidクラスとOracleXAExceptionクラスは最上位レベルのoracle.jdbc.xaパッケージにあります。それらの機能が、コードが実行される場所に依存しないからです。

中間層のシナリオでは、OracleXid、OracleXAExceptionおよびoracle.jdbc.xa.clientパッケージをインポートします。

ただし、XAコードをターゲットOracle Databaseで実行する場合は、oracle.jdbc.xa.clientパッケージのかわりに、oracle.jdbc.xa.serverパッケージをインポートします。

ターゲット・データベースの内部で実行するコードでリモート・データベースにもアクセスする必要がある場合は、どちらのパッケージもインポートしません。かわりに、clientパッケージから使用するクラス(リモート・データベースにアクセスするとき)またはserverパッケージから使用するクラス(ローカル・データベースにアクセスするとき)の名前を完全に修飾する必要があります。クラス名は、これらのパッケージで重複しています。

32.2 XAコンポーネント

この項では、XAコンポーネント、つまりJDBC標準で指定される標準XAインタフェースと、それを実装するOracleクラスについて説明します。内容は次のとおりです。

- XADatasourceインタフェースとOracle実装
- XAConnectionインタフェースとOracle実装
- XAResourceインタフェースとOracle実装
- OracleXAResourceメソッドの機能と入力パラメータ
- XidインタフェースとOracle実装

32.2.1 XADatasourceインタフェースとOracle実装

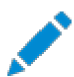
javax.sql.XADataSourceインタフェースは、XA接続のためのファクトリであるXAデータソースの標準機能の大枠を定めています。オーバーロードされたgetXAConnectionメソッドは、XA接続インスタンスを戻し、次のように、オプションでユーザー名とパスワードを入力に取ります。

```
public interface XADataSource
{
   XAConnection getXAConnection() throws SQLException;
   XAConnection getXAConnection(String user, String password)
      throws SQLException;
   ...
}
```

Oracle JDBCは、XADataSourceインタフェースをOracleXADataSourceクラスで実装します。どちらも、oracle.jdbc.xa.clientパッケージおよびoracle.jdbc.xa.serverパッケージに含まれています。

OracleXADataSourceクラスは、OracleConnectionPoolDataSourceクラス(OracleDataSourceクラスの拡張)の拡張も行います。そのため、すべての接続プロパティも継承されます。

OracleXADataSourceクラスのgetXAConnectionメソッドは、XA接続インスタンスのOracle実装を戻します。このインスタンスは、OracleXAConnectionインスタンスです。



ノート:

前述の非プーリング・データソースの場合と同じネーミング規則を使用して、XA データソースを Java Naming Directory and Interface(JNDI)に登録できます。

関連項目:

高速接続フェイルオーバーの詳細は、『Oracle Universal Connection Pool for JDBC開発者ガイド』を参照してください。

32.2.2 XAConnectionインタフェースとOracle実装

XA接続インスタンスは、プーリングされた接続インスタンスと同様に、データベースへの物理接続をカプセル化します。このデータベースは、そのXA接続インスタンスを作成したXAデータソース・インスタンスの接続プロパティで指定されたデータベースです。

各XA接続インスタンスには、対応するOracleXAResourceインスタンスを作成する機能もあります。このインスタンスは、分散トランザクションを協調させるために使用されます。

XA接続インスタンスは、標準javax.sql.XAConnectionインタフェースを実装するクラスのインスタンスです。

```
public interface XAConnection extends PooledConnection
{
   javax.jta.xa.XAResource getXAResource() throws SQLException;
}
```

すでに説明したとおり、XAConnectionインタフェースはjavax.sql.PooledConnectionインタフェースを拡張します。そのため、getConnection、close、addConnectionEventListenerおよびremoveConnectionEventListenerメソッドも継承されます。

Oracle JDBCは、XAConnectionインタフェースをOracleXAConnectionクラスで実装します。どちらも、oracle.jdbc.xa.clientパッケージおよびoracle.jdbc.xa.serverパッケージに含まれています。

OracleXAConnectionクラスは、OraclePooledConnectionクラスも拡張します。

OracleXAConnectionクラスのgetXAResourceメソッドは、OracleXAResourceインスタンスのOracle実装を戻します。これはOracleXAResourceインスタンスです。getConnectionメソッドは、OracleConnectionインスタンスを戻します。

XA接続インスタンスから戻されるJDBC接続インスタンスは、物理接続のカプセル化ではなく、物理接続への一時的なハンドルとして機能します。物理接続は、XA接続インスタンスでカプセル化されます。XAConnectionオブジェクトから取得された接続は、グローバル・トランザクションに加わるまでは通常の接続とまったく同じように動作します。グローバル・トランザクションに加わった時点で、自動コミット・ステータスがfalseに設定されます。グローバル・トランザクションが終了すると、自動コミット・ステータスはグローバル・トランザクション前の値に戻ります。XAConnectionから取得された接続でのデフォルトの自動コミット・ステータスは、Oracle Database 10gより前のすべてのリリースでは、falseです。Oracle Database 10gから、デフォルトのステータスはtrueです。

XA接続インスタンスgetConnectionメソッドは、コールされるたびに、デフォルトの動作を示す新しい接続インスタンスを戻します。また、以前の接続インスタンスで、残存していて、同じXA接続インスタンスによって戻されたものをすべてクローズします。ただし、新しい接続インスタンスをオープンする前に、以前の接続インスタンスはすべて明示的にクローズしておくことをお薦めします。

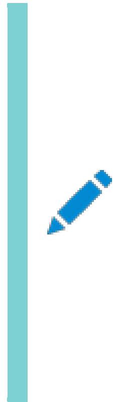
XA接続インスタンスのcloseメソッドをコールすると、データベースへの物理接続がクローズされます。これは、通常、中間層で実行します。

32.2.3 XAResourceインタフェースとOracle実装

トランザクション・マネージャはOracleXAResourceインスタンスを使用して、分散トランザクションを構成するすべてのトランザクション・ブランチを協調させます。

各OracleXAResourceインスタンスは、次の主要な機能を提供します。通常、トランザクション・マネージャから起動されます。

- 分散トランザクションと、このOracleXAResourceインスタンスを作成したXA接続インスタンスで動作するトランザクション・ブランチの、関連付けおよび関連付けの解除を行います。基本的には、分散トランザクションを物理接続またはXA接続インスタンスでカプセル化されたセッションに関連付けます。これは、トランザクションIDを使用して実行されます。
- 分散トランザクションの2フェーズ・コミット機能を実行します。すべてのトランザクション・ブランチで正常に処理されることが保証されるまで、その変更がどれか1つのトランザクション・ブランチでコミットされることはありません。



ノート:

- XA 接続インスタンスと OracleXAResource インスタンスの間には常に 1 対 1 の相関関係があるので、関連付けられた XA 接続インスタンスがクローズされると、OracleXAResource インスタンスは暗黙的にクローズされます。
- ある OracleXAResource インスタンスがトランザクションをオープンした場合、そのトランザクションは同じ OracleXAResource インスタンスによってクローズする必要があります。

OracleXAResourceインスタンスは、標準javax.transaction.xa.XAResourceインタフェースを実装するクラスのインスタンスです。Oracle JDBCは、XAResourceインタフェースをOracleXAResourceクラスとともに実装します。どちらも、oracle.jdbc.xa.clientパッケージおよびoracle.jdbc.xa.serverパッケージに含まれています。

Oracle JDBCドライバは、OracleXAConnectionクラスのgetXAResourceメソッドがコールされると、OracleXAResourceインスタンスを作成して戻します。OracleXAResourceインスタンスを、接続インスタンスおよびその接続によって実行されるトランザクション・ブランチに関連付けるのも、Oracle JDBCドライバの役割です。

このメソッドは、特定の接続およびその接続で実行されるトランザクション・ブランチにOracleXAResourceインスタンスを関連付けるときのメソッドです。

32.2.4 OracleXAResourceメソッドの機能と入力パラメータ

OracleXAResourceクラスには、トランザクション・ブランチを、関連付けられた分散トランザクションと協調させるためのメソッドがいくつかあります。この機能は、通常、2フェーズ・コミット操作で起動されます。

通常、トランザクション・マネージャが、アプリケーション・サーバーなどの中間層コンポーネントからOracleXAResourceインスタンスを受け取り、この機能を起動します。

これらのメソッドはそれぞれ、Xidインスタンスの形で、トランザクションIDを入力に取ります。それにはトランザクション・ブランチのIDコンポーネントと分散トランザクションのIDコンポーネントが含まれます。すべてのトランザクション・ブランチが一意のトランザクションIDを持っていますが、同一のグローバル・トランザクションに属するトランザクション・ブランチは、トランザクションIDの一部として、同じグローバル・トランザクション・コンポーネントを持ちます。

start

トランザクション・ブランチの処理を開始します。トランザクション・ブランチを分散トランザクションに関連付けます。

```
void start(Xid xid, int flags)
```

flagsパラメータには、次の値のうち1つ以上を設定する必要があります。

- XAResource.TMNOFLAGS

  このXAリソース・インスタンスに関連付けられたセッションにおける後続の操作のための新しいトランザクション・ブランチが開始される箇所にフラグを設定します。このブランチはトランザクションID xidを持ちます。これは、トランザクション・マネージャによって作成されたOracleXidインスタンスです。これは、トランザクション・ブランチを適切な分散トランザクションにマップします。

- XAResource.TMJOIN

  このXAリソース・インスタンスに関連付けられたセッションの後続の操作を、xidで指定した既存のトランザクション・ブランチに結合します。

- XAResource.TMRESUME

  xidで指定したトランザクション・ブランチを再開します。

ノート:

再開できるのは、前に保留されていたトランザクション・ブランチのみです。

- OracleXAResource.TMPROMOTE

  ローカル・トランザクションからグローバル・トランザクションへ昇格します

- OracleXAResource.ORATMSERIALIZABLE

  トランザクションIDがxidのシリアライズ可能なトランザクションを起動します。

- OracleXAResource.ORATMREADONLY

  トランザクションIDがxidの読取り専用トランザクションを起動します。

- OracleXAResource.ORATMREADWRITE

  トランザクションIDがxidの読取り/書込みトランザクションを起動します。

- OracleXAResource.ORATRANSLOOSE

  トランザクションIDがxidの疎結合トランザクションを起動します。

TMNOFLAGS、TMJOIN、TMRESUME、TMPROMOTE、ORATMSERIALIZABLE、ORATMREADONLYおよびORATMREADWRITE は、XAResourceインタフェースおよびOracleXAResourceクラスのstaticメンバーとして定義されています。ORATMSERIALIZABLE、ORATMREADONLYおよびORATMREADWRITEは、分離モード・フラグです。デフォルトの分離動作は、READ COMMITTEDです。

ノート:

- TMRESUME を指定して start メソッドを使用するかわりに、トランザクション・マネージャで OracleXAResource にキャストし、Oracle 拡張機能 resume(Xid xid) メソッドを使用することもできます。

- TMRESUME を使用する場合は、start(xid, XAResource.TMRESUME | OracleXAResource.TMNOMIGRATE) の場合と同じように、TMNOMIGRATE も使用する必要があります。これ

により、アプリケーションがエラー「ORA 1002: フェッチ順序が無効です。」を受け取らなくなります。

- 分離モード・フラグを不正確に使用すると、コード XAER_INVAL の例外が発生します。さらに、グローバル・トランザクションを再開する場合、分離モード・フラグを使用することができません。これは、既存のトランザクションの分離レベルを設定できないためです。トランザクションを再開するときに分離モード・フラグを使用しようとすると、コード ORA-24790 の外部 Oracle 例外が発生します。

- エラー「ORA 1002: フェッチ順序が無効です。」の発生を防ぐには、start メソッドの一部として TMNOMIGRATE フラグを含めます。たとえば:

```
start(xid, XAResource.TMSUSPEND | OracleXAResource.TMNOMIGRATE);
```

- OracleXAResource で定義されているフラグはすべて Oracle 拡張機能です。これらのフラグを使用するトランザクション・マネージャの作成時には、この点を留意してください。

トランザクション・ブランチを起動するときに適切なトランザクションIDを作成するには、そのトランザクション・ブランチが属する分散トランザクションを、トランザクション・マネージャに指示する必要があります。このメカニズムは、中間層とトランザクション・マネージャの間で処理されます。

end

xidで指定されたトランザクション・ブランチの処理を終了します。トランザクション・ブランチと分散トランザクションの関連付けを解除します。

```
void end(Xid xid, int flags)
```

flagsパラメータには、次の値のうち1つを設定できます。

- XAResource.TMSUCCESS

  このトランザクション・ブランチが正常であることを示します。

- XAResource.TMFAIL

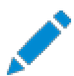
  このトランザクション・ブランチが異常であることを示します。

- XAResource.TMSUSPEND

  xidで指定したトランザクション・ブランチを保留することを示します。トランザクション・ブランチを保留することにより、複数のトランザクション・ブランチを単一セッションで使用できます。ただし、常にアクティブにできるのは1つのみです。また、多くの場合、リソースの点から見て、2つのセッションを使用するよりもコストがかかります。

TMSUCCESS、TMFAILおよびTMSUSPENDは、XAResourceインタフェースおよびOracleXAResourceクラスの静的メンバーとして定義されています。

ノート:



- TMSUSPEND を指定して end メソッドを使用するかわりに、トランザクション・マネージャで OracleXAResource にキャストし、Oracle 拡張機能 suspend(Xid xid) メソッドを使用することもできます。

- トランザクションを保留するこの XA 機能によって、単一 JDBC 接続で、各種トランザクションを切り替えることができます。分散トランザクション環境になく、XA クラスを必要としない場合でも、この機能を実現するために XA クラスを使用できます。

- TMSUSPEND を使用する場合は、end(xid, XAResource.TMSUSPEND | OracleXAResource.TMNOMIGRATE)の場合と同じように、TMNOMIGRATE も使用する必要があります。これにより、アプリケーションがエラー「ORA 1002: フェッチ順序が無効です。」を受け取らなくなります。

- エラー「ORA 1002: フェッチ順序が無効です。」の発生を防ぐには、end メソッドの一部として TMNOMIGRATE フラグを含めます。たとえば:

```
end(xid, XAResource.TMSUSPEND | OracleXAResource.TMNOMIGRATE);
```

- OracleXAResource で定義されているフラグはすべて Oracle 拡張機能です。これらのフラグを使用するトランザクション・マネージャの場合は、この点を留意してください。

prepare

xidで指定されたトランザクション・ブランチで実行される変更の準備をします。これは、データベースへのアクセスと、変更の正常なコミットを保証する、2フェーズ・コミット操作の最初のフェーズです。

```
int prepare(Xid xid)
```

このメソッドでは、次の整数値を戻します。

- XAResource.XA_RDONLY

  トランザクション・ブランチが、SELECT文など、読取り専用操作のみを実行する場合に戻されます。

- XAResource.XA_OK

  トランザクション・ブランチが更新を実行する場合、すべて準備済で、エラーがなければ、この値が戻されます。

XA_RDONLYおよびXA_OKは、XAResourceインタフェースおよびOracleXAResourceクラスのstaticメンバーとして定義されています。

ノート:

- prepare メソッドでは、トランザクション・ブランチが更新を実行する場合、そのうち 1 つでも準備中にエラーが発生すると、値が戻されないことがあります。この場合、XA 例外がスローされます。

- prepare メソッドをコールする前に、必ずブランチの end メソッドをコールする必要があります。

- 分散トランザクションにあるトランザクション・ブランチが 1 つのみの場合、prepare メソッドをコールする必要はありません。準備なしに、OracleXAResource の commit メソッドをコールできます。

commit

xidで指定したトランザクション・ブランチで準備された変更をコミットします。これは2フェーズ・コミットの2番目のフェーズで、トラン

ザクション・ブランチをすべて正常に準備できた場合のみ実行されます。

```
void commit(Xid xid, boolean onePhase)
```

onePhaseパラメータは、次のように設定します。

- true

  トランザクション・ブランチをコミットするときに、2フェーズ・プロトコルでなく1フェーズ・プロトコルを使用します。これは、分散トランザクションにトランザクション・ブランチが1つのみ存在する場合に適しています。prepareステップは省略されます。

- false

  トランザクション・ブランチをコミットするときに、2フェーズ・プロトコルを使用します。

rollback

xidで指定したトランザクション・ブランチで準備された変更をロールバックします。

```
void rollback(Xid xid)
```

forget

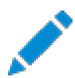
ヒューリスティックに決着されたトランザクション・ブランチを無視するようにリソース・マネージャに通知します。

```
public void forget(Xid xid)
```

recover

トランザクション・マネージャは回復時にこのメソッドを呼び出して、現在準備状態またはヒューリスティックに決着された状態にあるトランザクション・ブランチのリストを取得します。

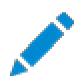
```
public Xid[] recover(int flag)
```



ノート:

TMSTARTRSCAN、TMENDRSCAN または TMNOFLAGS 以外の値を flag に使用すると、例外が発生します。そうでない場合、flag は無視されます。

リソース・マネージャは、現在準備中またはヒューリスティックに完了した状態のトランザクション・ブランチに対して、0(ゼロ)個以上のXidを戻します。操作中にエラーが発生した場合は、リソース・マネージャにより適切なXAExceptionが発行されます。



ノート:

recover メソッドは、Oracle Database サーバーの DBA_PENDING_TRANSACTIONS 上で SELECT 権限、SYS.DBMS_XA 上で EXECUTE 権限が必要です。Oracle Database 11g リリース 1 より前のデータベース・バージョンでは、Oracle Bug#5945463 の修正を含む Oracle パッチが利用できないか、個別のアプリケーション使用例にそのパッチを適用できない場合、recover メソッドは、SYSBDBA 権限も必要とします。SYSDBA 権限を定期的に使用すると、セキュリティ上の問題があります。したがって、recover メソッドを使用する必要がある場合、データベース

をアップグレードするか、または Bug#5945463 の修正を適用することを強くお薦めします。

isSameRM

2つのOracleXAResourceインスタンスが同じリソース・マネージャに対応しているかどうかを判断するには、一方のOracleXAResourceインスタンスから、入力としてもう一方のOracleXAResourceインスタンスを指定して、isSameRMメソッドをコールします。次の例では、xares1およびxares2はOracleXAResourceインスタンスとします。

```
boolean sameRM = xares1.isSameRM(xares2);
```

32.2.5 XidインタフェースとOracle実装

トランザクション・マネージャは各トランザクションIDインスタンスを作成し、分散トランザクションの各ブランチを協調させる際に使用します。各トランザクション・ブランチには一意のトランザクションIDが割り当てられます。これには次の情報が含まれます。

- フォーマット識別子

  フォーマット識別子は、Javaトランザクション・マネージャを指定します。たとえば、フォーマット識別子orclがあります。このフィールドは、nullにできません。フォーマット識別子のサイズは4バイトです。

- グローバル・トランザクション識別子

  分散トランザクションIDコンポーネントとも呼ばれます。グローバル・トランザクション識別子のサイズは64バイトです。

- ブランチ修飾子

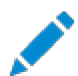
  トランザクション・ブランチIDコンポーネントとも呼ばれます。ブランチ修飾子のサイズは64バイトです。

64バイトのグローバル・トランザクション識別子の値は、同じ分散トランザクションに属するトランザクション・ブランチすべてのトランザクションIDで同一です。ただし、トランザクションIDの全体は、トランザクション・ブランチごとに一意です。

XAトランザクションIDインスタンスは、標準javax.transaction.xa.Xidインタフェースを実装するクラスのインスタンスです。このインタフェースは、X/Openトランザクション識別子XID構造体のJavaマッピングです。

Oracleでは、このインタフェースをoracle.jdbc.xaパッケージのOracleXidクラスに実装しています。OracleXidインスタンスは、トランザクション・マネージャでのみ使用されますが、アプリケーション・プログラムまたはアプリケーション・サーバーにとって透過的です。

ノート:



Oracle では、OracleXAResource のコールに OracleXid を使用する必要はありません。かわりに、javax.transaction.xa.Xid インタフェースを実装するクラスを使用します。

トランザクション・マネージャでは、次のメソッドを使用して、OracleXidインスタンスを作成できます。

```
public OracleXid(int fId, byte gId[], byte bId[]) throws XAException
```

fIdは、フォーマット識別子を表す整数値です。gId[]は、グローバル・トランザクション識別子を表すバイト配列です。bId[]は、ブランチ修飾子を表すバイト配列です。

Xidインタフェースでは、次のgetterメソッドが指定されています。

- public int getFormatId()
- public byte[] getGlobalTransactionId()
- public type[] getBranchQualifier()

32.3 エラー処理と最適化

この項では、XA例外の機能とエラー処理およびXA実装でのOracle最適化について説明します。内容は次のとおりです。

- XAExceptionクラスとメソッド
- OracleエラーとXAエラーのマッピング
- XAエラー処理
- Oracle XA最適化

例外とエラー処理の説明には、標準XA例外クラスとOracle固有のXA例外クラスが含まれます。また、特定のXAエラー・コードおよびエラー処理の方法も含まれます。

32.3.1 XAExceptionクラスとメソッド

XAメソッドでは、一般例外やSQLExceptionsではなく、XA例外が発生します。XA例外は、標準クラスjavax.transaction.xa.XAExceptionまたはそのサブクラスのインスタンスです。

Oracle XAExceptionは、Oracleエラーの部分とXAエラーの部分で構成されています。Oracleには、標準javax.transaction.xa.XAExceptionクラスのoracle.jdbc.xa.OracleXAExceptionサブクラスが用意されています。OracleXAExceptionインスタンスは、次のコンストラクタの1つを使用して構成されます。

```
public OracleXAException()
public OracleXAException(int error)
```

エラー値は、Oracle SQLエラー値とXAエラー値を組み合せたエラー・コードです。Oracleエラー値とXAエラー値を組み合せる正確な方法は、JDBCドライバによって判断されます。

OracleXAExceptionクラスには、次のメソッドが含まれます。

- public int getOracleError()

  このメソッドは、例外に含まれるOracle SQLエラー・コード(標準ORAエラー番号)を戻します。Oracle SQLエラーがなければ、0を戻します。

- public int getXAError()

  このメソッドは、例外に含まれるXAエラー・コードを戻します。XAエラー値は、javax.transaction.xa.XAExceptionクラスで定義されています。

32.3.2 OracleエラーとXAエラーのマッピング

表32-2で示すように、OracleエラーはOracleXAExceptionインスタンスのXAエラーに対応しています。

表32-2 OracleとXAのエラー・マッピング

Oracleエラー・コード	XAエラー・コード
ORA 24756	XAException.XAER_NOTA
ORA 24764	XAException.XA_HEURCOM
ORA 24765	XAException.XA_HEURRB
ORA 24766	XAException.XA_HEURMIX
ORA 24767	XAException.XA_RDONLY
ORA 25351	XAException.XA_RETRY
ORA 30006	XAException.XA_RETRY
ORA 24763	XAException.XAER_PROTO
ORA 24769	XAException.XAER_PROTO
ORA 24770	XAException.XAER_PROTO
ORA 24776	XAException.XAER_PROTO
ORA 2056	XAException.XAER_PROTO
ORA 17448	XAException.XAER_PROTO
ORA 24768	XAException.XAER_PROTO
ORA 24775	XAException.XAER_PROTO
ORA 24761	XAException.XA_RBROLLBACK
ORA 2091	XAException.XA_RBROLLBACK
ORA 2092	XAException.XA_RBROLLBACK
ORA 24780	XAException.XAER_RMERR
その他すべてのORAエラー	XAException.XAER_RMFAIL

32.3.3 XAエラー処理

次のコードでは、OracleXAExceptionクラスを使用して、XA例外を処理します。

```
try {
   ...
   ...Perform XA operations...
   ...
} catch(OracleXAException oxae) {
  int oraerr = oxae.getOracleError();
  System.out.println("Error " + oraerr);
}
  catch(XAException xae)
{...Process generic XA exception...}
```

XA操作によってOracle固有のXA例外が発生しなかった場合、このコードは一般XA例外の処理を行いません。

32.3.4 Oracle XA最適化

Oracle JDBCには、分散トランザクションの2つ以上のブランチが同じデータベース・インスタンスを使用する場合、つまり、これらのブランチに関連付けられたOracleXAResourceインスタンスが同じリソース・マネージャに関連付けられている場合、パフォーマンスを改善する機能があります。

このような場合、これらのOracleXAResourceインスタンスのうち1つのprepareメソッドのみがXA_OKを戻すか、または失敗します。残りは、更新が行われる場合でも、XA_RDONLYを戻します。これにより、トランザクション・マネージャは、すべてのトランザクション・ブランチを暗黙的に結合し、XA_OKを戻した(または失敗した)OracleXAResourceインスタンスによって、結合されたトランザクションをコミット(失敗した場合はロールバック)できます。

トランザクション・マネージャは、OracleXAResourceクラスのisSameRMメソッドを使用して、2つのOracleXAResourceインスタンスが同じリソース・マネージャを使用しているかどうかを判断できます。このようにして、XA_RDONLY戻り値の意味を解析できます。

32.4 分散トランザクションの実装について

この項では、Oracle XA機能を使用して分散トランザクションを実装する方法の例を示します。この項の内容は次のとおりです。

- Oracle XAのインポートの概要
- OracleのXAコード・サンプル

32.4.1 Oracle XAのインポートの概要

Oracle XA機能を使用するには、次のパッケージをインポートする必要があります。

```
import oracle.jdbc.xa.OracleXid;
import oracle.jdbc.xa.OracleXAException;
import oracle.jdbc.pool.*;
import oracle.jdbc.xa.client.*;
import javax.transaction.xa.*;
```

oracle.jdbc.poolパッケージには、接続プーリング機能のクラスが含まれています。この一部には、XA関連クラスがサブクラス

として含まれています。

また、コードがOracle Databaseの内部で実行され、そのデータベースにアクセスしてSQL操作を行う場合は、oracle.jdbc.xa.clientではなく、oracle.jdbc.xa.serverをインポートする必要があります。

```
import oracle.jdbc.xa.server.*;
```

サーバー側Thinドライバを使用して、アプリケーションがXAトランザクションの一部として別のOracle Databaseにアクセスする必要がある場合は、コードで完全に修飾されたoracle.xa.clientクラスの名前を使用できます。

clientおよびserverパッケージには、それぞれのバージョンのOracleXADataSource、OracleXAConnectionおよびOracleXAResourceクラスがあります。これら3つのクラスの抽象バージョンは、最上位のoracle.jdbc.xaパッケージに含まれています。

32.4.2 OracleのXAコード・サンプル

このサンプルでは、異なるデータベースに対する2つのトランザクション・ブランチで、2フェーズ分散トランザクションを使用します。

簡単にするため、このサンプルでは、通常は中間層に置くコードと、通常はトランザクション・マネージャに置くコード(OracleXAResourceメソッドの起動や、トランザクションIDの作成など)を組み合せているので、注意してください。

短くするため、トランザクションID作成の指定およびSQL操作の実行は、ここでは示しません。完全な例は製品に付属しています。

このサンプルは、次の順序で実行します。

1. トランザクション・ブランチ1を開始します。
2. トランザクション・ブランチ2を開始します。
3. ブランチ1のDML操作を実行します。
4. ブランチ2のDML操作を実行します。
5. トランザクション・ブランチ1を終了します。
6. トランザクション・ブランチ2を終了します。
7. ブランチ1を準備します。
8. ブランチ2を準備します。
9. ブランチ1をコミットします。
10. ブランチ2をコミットします。

```
// You need to import the java.sql package to use JDBC
import java.sql.*;
import javax.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;
import oracle.jdbc.xa.OracleXid;
import oracle.jdbc.xa.OracleXAException;
import oracle.jdbc.xa.client.*;
import javax.transaction.xa.*;
```

```
class XA4
{
  public static void main (String args [])
       throws SQLException
  {
    try
    {
        String URL1 = "jdbc:oracle:oci:@";
        // You can put a database name after the @ sign in the connection URL.
        String URL2 ="jdbc:oracle:thin:@(description=(address=(host=localhost)
                     (protocol=tcp)(port=5521))(connect_data=(service_name=orcl)))";
        // Create first DataSource and get connection
        OracleDataSource ods1 = new OracleDataSource();
        ods1.setURL(URL1);
        ods1.setUser("HR");
        ods1.setPassword("hr");
        Connection conna = ods1.getConnection();
        // Create second DataSource and get connection
        OracleDataSource ods2 = new OracleDataSource();
        ods2.setURL(URL2);
        ods2.setUser("HR");
        ods2.setPassword("hr");
        Connection connb = ods2.getConnection();
        // Prepare a statement to create the table
        Statement stmta = conna.createStatement ();
        // Prepare a statement to create the table
        Statement stmtb = connb.createStatement ();
        try
        {
          // Drop the test table
          stmta.execute ("drop table my_table");
        }
        catch (SQLException e)
        {
          // Ignore an error here
        }
        try
        {
          // Create a test table
          stmta.execute ("create table my_table (col1 int)");
        }
        catch (SQLException e)
        {
          // Ignore an error here too
        }
        try
        {
          // Drop the test table
          stmtb.execute ("drop table my_tab");
        }
        catch (SQLException e)
        {
          // Ignore an error here
        }
        try
        {
          // Create a test table
          stmtb.execute ("create table my_tab (col1 char(30))");
```

```
    }
    catch (SQLException e)
    {
      // Ignore an error here too
    }
    // Create XADataSource instances and set properties.
    OracleXADataSource oxds1 = new OracleXADataSource();
    oxds1.setURL("jdbc:oracle:oci:@");
    oxds1.setUser("HR");
    oxds1.setPassword("hr");
    OracleXADataSource oxds2 = new OracleXADataSource();
    oxds2.setURL("jdbc:oracle:thin:@(description=(address=(host=localhost)
               (protocol=tcp)(port=5521))(connect_data=(service_name=orcl)))");
    oxds2.setUser("HR");
    oxds2.setPassword("hr");

    // Get XA connections to the underlying data sources
    XAConnection pc1  = oxds1.getXAConnection();
    XAConnection pc2  = oxds2.getXAConnection();
    // Get the physical connections
    Connection conn1 = pc1.getConnection();
    Connection conn2 = pc2.getConnection();
    // Get the XA resources
    XAResource oxar1 = pc1.getXAResource();
    XAResource oxar2 = pc2.getXAResource();
    // Create the Xids With the Same Global Ids
    Xid xid1 = createXid(1);
    Xid xid2 = createXid(2);
    // Start the Resources
    oxar1.start (xid1, XAResource.TMNOFLAGS);
    oxar2.start (xid2, XAResource.TMNOFLAGS);
    // Execute SQL operations with conn1 and conn2
    doSomeWork1 (conn1);
    doSomeWork2 (conn2);
    // END both the branches -- IMPORTANT
    oxar1.end(xid1, XAResource.TMSUCCESS);
    oxar2.end(xid2, XAResource.TMSUCCESS);
    // Prepare the RMs
    int prp1 =  oxar1.prepare (xid1);
    int prp2 =  oxar2.prepare (xid2);
    System.out.println("Return value of prepare 1 is " + prp1);
    System.out.println("Return value of prepare 2 is " + prp2);
    boolean do_commit = true;
    if (!((prp1 == XAResource.XA_OK) || (prp1 == XAResource.XA_RDONLY)))
       do_commit = false;
    if (!((prp2 == XAResource.XA_OK) || (prp2 == XAResource.XA_RDONLY)))
       do_commit = false;
   System.out.println("do_commit is " + do_commit);
    System.out.println("Is oxar1 same as oxar2 ? " + oxar1.isSameRM(oxar2));
    if (prp1 == XAResource.XA_OK)
      if (do_commit)
         oxar1.commit (xid1, false);
      else
         oxar1.rollback (xid1);
    if (prp2 == XAResource.XA_OK)
      if (do_commit)
         oxar2.commit (xid2, false);
      else
```

```
            oxar2.rollback (xid2);
          // Close connections
        conn1.close();
        conn1 = null;
        conn2.close();
        conn2 = null;
        pc1.close();
        pc1 = null;
        pc2.close();
        pc2 = null;
        ResultSet rset = stmta.executeQuery ("select col1 from my_table");
        while (rset.next())
          System.out.println("Col1 is " + rset.getInt(1));

        rset.close();
        rset = null;
        rset = stmtb.executeQuery ("select col1 from my_tab");
        while (rset.next())
          System.out.println("Col1 is " + rset.getString(1));

        rset.close();
        rset = null;
        stmta.close();
        stmta = null;
        stmtb.close();
        stmtb = null;
        conna.close();
        conna = null;
        connb.close();
        connb = null;
    } catch (SQLException sqe)
    {
      sqe.printStackTrace();
    } catch (XAException xae)
    {
      if (xae instanceof OracleXAException) {
        System.out.println("XA Error is " +
                       ((OracleXAException)xae).getXAError());
        System.out.println("SQL Error is " +
                       ((OracleXAException)xae).getOracleError());
      }
    }
  }
  static Xid createXid(int bids)
    throws XAException
  {...Create transaction IDs...}
  private static void doSomeWork1 (Connection conn)
   throws SQLException
  {...Execute SQL operations...}
  private static void doSomeWork2 (Connection conn)
   throws SQLException
  {...Execute SQL operations...}
}
```

32.5 Oracle JDBCドライバのネイティブXA

通常、XAコマンドは次の方法でサーバーに送信できます。

- 非ネイティブAPIの使用
- ネイティブAPIの使用

この2つの方法でサーバーにXAコマンドを送信する場合、パフォーマンスに大きな相違があります。非ネイティブAPIと比較して、ネイティブAPIを使用するほうが、高いパフォーマンスを達成できます。

Oracle Database 10gより前は、ThinネイティブのAPIが利用できなかったため、ThinドライバはネイティブでないAPIを使用してXAコマンドをサーバーに送信していました。ネイティブでないAPIはPL/SQLプロシージャを使用します。このため、次の短所があります。

- ワイヤ上で異なるメッセージが必要です。
- データベースとのラウンドトリップが増加します。
- オープン状態のカーソルが増加します。
- 文キャッシュの領域を占有することによって、文キャッシュを破損します。

さらに、非ネイティブAPIの実装がサーバー内にあります。したがって、XAコマンドを送信する際の問題を解決するために、サーバーのパッチが必要です。パッチを適用するとサーバーを再起動する必要があるので、大きな問題になります。

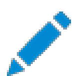
Oracle Database 10gから、ThinネイティブAPIを使用できるようになったため、デフォルトではこのAPIを使用してXAコマンドを送信します。ネイティブAPIは、非ネイティブAPIよりも処理が10倍高速です。

この項の内容は次のとおりです。

- OCIネイティブXA
- ThinネイティブXA

32.5.1 OCIネイティブXA

ネイティブXAは、`OracleXADataSource`クラスの`tnsEntry`および`nativeXA`プロパティを使用して有効にします。



ノート:

現在のところ、OCI Native XA は、マルチスレッド環境では機能しません。OCI ドライバは、Oracle の C/XA ライブラリを使用して分散トランザクションをサポートします。このため、グローバル・トランザクションを再開する前に、各スレッドごとに XAConnection を取得する必要があります。

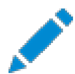
構成およびインストール

SolarisまたはLinuxシステムの場合、ネイティブXA機能を使用可能にするには、`libheteroxa11.so`共有ライブラリが必要です。ネイティブXA機能を正しく動作させるには、このライブラリをインストールして、検索パスで使用可能にする必要があります。

Microsoft Windowsシステムの場合、ネイティブXA機能を使用可能にするには、`heteroxa11.dll`ファイルが必要です。ネイティブXA機能を正しく動作させるには、このファイルをインストールして、WindowsのDLLパスで使用可能にする必要があります。

例外処理

分散トランザクションでネイティブXA機能を使用する場合は、アプリケーションでOracleXAExceptionまたはOracleSQLExceptionをチェックするのではなく、単純にXAExceptionまたはSQLExceptionをチェックすることをお薦めします。



ノート:

標準の XA エラー・コードに対する SQL エラー・コードのマッピングは、ネイティブ XA 機能には適用されません。

ネイティブXAのコード・サンプル

次のコードは、ネイティブXA機能を使用可能にする方法を示します。

```
...
// Create a XADataSource instance
OracleXADataSource oxds = new OracleXADataSource();
oxds.setURL(url);
// Set the nativeXA property to use Native XA feature
oxds.setNativeXA(true);
// Set the tnsEntry property to an older DB as required
oxds.setTNSEntryName("ora805");
...
```

関連トピック

- データソースの機能とプロパティ
- ネイティブXAメッセージ

32.5.2 ThinネイティブXA

JDBC OCIドライバと同様に、JDBC ThinドライバもネイティブXAをサポートしています。ただし、JDBC ThinドライバはデフォルトでネイティブXAをサポートします。一方、JDBC OCIドライバの場合、ネイティブXAのサポートがデフォルトでは使用可能になりません。

次のようにXAデータソースでsetNativeXA(false)をコールすることで、ネイティブXAを無効にできます。

```
...
// Create a XADataSource instance
OracleXADataSource oxds = new OracleXADataSource();
...
// Disabling Native XA
oxds.setNativeXA(false);
...
```

たとえば、ネイティブXAコードの不具合に対する回避策として、ネイティブXAの無効化が必要になる場合があります。

第VIII部 管理性

この部では、Oracle Java Database Connectivity(JDBC)ドライバのデータベース管理および診断機能のサポートについて説明します。

第VIIIの構成は次のとおりです。

- データベース管理
- JDBCの診断機能
- JDBC DMSメトリック

33 データベース管理

この章では、Oracle Database 11gリリース1で導入されたデータベース管理方法について説明します。この章の構成は、次のとおりです。

- データベース管理メソッドの使用
- startupメソッドの使用
- shutdownメソッドの使用
- 完全な例

33.1 データベース管理メソッドの使用

Oracle Database 11gリリース1以降、2つのJDBCメソッド、startupおよびshutdownがoracle.jdbc.OracleConnectionインタフェースに追加されており、これによりOracle Databaseインスタンスの起動と停止を実行できます。SQL*Plusからデータベース・インスタンスを起動または停止する方法と同様です。

startupおよびshutdownメソッドを使用するには、次の点に従う必要があります。

- サーバー専用の接続を行うこと。ディスパッチャを介して共有サーバーには接続できません。
- SYSDBAまたはSYSOPERとして接続すること。SYSDBAまたはSYSOPERとして、Oracle JDBCドライバ経由で接続するには、INTERNAL_LOGON接続プロパティに適切な値を設定する必要があります。

SYSDBA権限を使用し、JDBC Thinドライバ経由でログオンできるようにするには、パスワード・ファイルを使用するようにサーバーを構成しておく必要があります。たとえば、SYSDBA権限を使用し、JDBC Thinドライバ経由で接続するようにsystem/managerを構成するには、次の手順を実行します。

1. コマンドラインから次のように入力します。

```
orapwd file=$ORACLE_HOME/dbs/orapw entries=5
Enter password: password
```

2. SQL*Plusから、SYSDBAとしてデータベースに接続し、次のコマンドを実行します。

```
GRANT SYSDBA TO system;
PASSWORD system
       Changing password for system
       New password: password
       Retype new password: password
```

3. init.oraを編集し、次の行を追加します。

```
REMOTE_LOGIN_PASSWORDFILE=EXCLUSIVE
```

4. データベース・インスタンスを再起動します。

JDBC OCIドライバの場合は、JDBC Thinドライバと違い、サーバー上のパスワード・ファイルを指定することなく、SYSDBAまたはSYSOPERとしてローカルに接続できます。

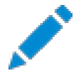
33.2 startupメソッドの使用

startupメソッドを使用してデータベース・インスタンスを起動するには、SYSDBAまたはSYSOPER権限を使用して、PRELIM_AUTHモード(データベースがダウンしているときに許可される唯一の接続モード)でデータベースに接続する必要があります。そのためには、接続プロパティであるPRELIM_AUTHをtrueに設定する必要があります。PRELIM_AUTHモードでは、ダウンしているデータベース・インスタンスの起動のみ実行できます。このモードでは、SQL文の実行はできません。

例

次のコードは、ダウンしているデータベース・インスタンスを起動する方法を示しています。

```
OracleDataSource ds = new OracleDataSource();
    Properties prop = new Properties();
    prop.setProperty("user","sys");
    prop.setProperty("password","manager");
    prop.setProperty("internal_logon","sysdba");
    prop.setProperty("prelim_auth","true");
    ds.setConnectionProperties(prop);
    ds.setURL("jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=XYZ.com)(PORT=5221))"
+ "(CONNECT_DATA=(SERVICE_NAME=rdbms.devplmt.XYZ.com)))");
    OracleConnection conn = (OracleConnection)ds.getConnection();
    conn.startup(OracleConnection.DatabaseStartupMode.NO_RESTRICTION);
    conn.close();
```



ノート:

startup メソッドは、サーバーのパラメータ・ファイルを使用してデータベースを起動します。Oracle JDBC ドライバは、クライアントのパラメータ・ファイルを使用したデータベースの起動はサポートしていません。

33.2.1 データベース起動オプション

startupメソッドには、データベース起動オプションを指定するパラメータを指定できます。表33-1は、サポートされているデータベース起動オプションを示しています。これらのオプションは、oracle.jdbc.OracleConnection.DatabaseStartupModeクラスで定義されています。

表33-1 サポートされているデータベース起動オプション

オプション	説明
FORCE	新しいインスタンスを起動する前に、データベースで中断モードになっている現行インスタンスがある場合は停止します。
NO_RESTRICTION	アクセス制限なしでデータベースを起動します。
RESTRICT	データベースを起動し、データベース・アクセスを、CREATE SESSION と RESTRICTED SESSION の両方の権限を持つユーザー(通常は DBA)のみに許可します。

startupメソッドは、データベース・インスタンスを起動するのみです。データベース・インスタンスのマウントもオープンも実行しません。データベース・インスタンスをマウントおよびオープンするには、SYSDBAまたはSYSOPERとして、PRELIM_AUTH以外のモードで再接続する必要があります。

例

次のコードは、データベース・インスタンスをマウントおよびオープンする方法を示しています。

```
    OracleDataSource ds1 = new OracleDataSource();
    Properties prop1 = new Properties();
    prop1.setProperty("user","sys");
    prop1.setProperty("password","manager");
    prop1.setProperty("internal_logon","sysdba");
    ds1.setConnectionProperties(prop1);
    ds1.setURL(DB_URL);
    OracleConnection conn1 = (OracleConnection)ds1.getConnection();
    Statement stmt = conn1.createStatement();
    stmt.executeUpdate("ALTER DATABASE MOUNT");
    stmt.executeUpdate("ALTER DATABASE OPEN");
```

33.3 shutdownメソッドの使用

shutdownメソッドを使用すると、Oracle Databaseインスタンスを停止できます。このメソッドを使用するには、SYSDBAまたはSYSOPERとしてデータベースに接続する必要があります。

例

次のコードは、データベース・インスタンスをシャットダウンする方法を示しています。

```
    OracleDataSource ds2 = new OracleDataSource();
...
    OracleConnection conn2 = (OracleConnection)ds2.getConnection();
    conn2.shutdown(OracleConnection.DatabaseShutdownMode.IMMEDIATE);
    Statement stmt1 = conn2.createStatement();
    stmt1.executeUpdate("ALTER DATABASE CLOSE NORMAL");
    stmt1.executeUpdate("ALTER DATABASE DISMOUNT");
    stmt1.close();
    conn2.shutdown(OracleConnection.DatabaseShutdownMode.FINAL);
    conn2.close();
```

33.3.1 データベース停止オプション

startupメソッドと同様、shutdownメソッドにもパラメータを指定できます。このパラメータには、データベースの停止に関するオプションを指定します。表33-2に、サポートされているデータベース停止オプションを示します。それらのオプションは、oracle.jdbc.OracleConnection.DatabaseShutdownModeクラスで定義されています。

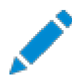
表33-2 サポートされているデータベース停止オプション

オプション	説明

オプション	説明
ABORT	現在のコールが完了したり、ユーザーがデータベースから切断するまで待機しません。
CONNECT	新しい接続を拒否し、既存の接続の終了を待ちます。
FINAL	データベースを停止します。
IMMEDIATE	現在のコールが完了したり、ユーザーがデータベースから切断するまで待機しません。
TRANSACTIONAL	新しいトランザクションを拒否し、アクティブなトランザクションの終了を待ちます。
TRANSACTIONAL_LOCAL	新しいローカル・トランザクションを拒否し、アクティブなローカル・トランザクションの終了を待ちます。

ABORTおよびFINAL以外の停止オプションの場合、データベースを実際に停止するには、FINALオプションを指定してshutdownメソッドを再度コールする必要があります。

ノート:



shutdown(DatabaseShutdownMode.FINAL)メソッドをコールする場合は、事前に、CONNECT、TRANSACTIONAL、TRANSACTIONAL_LOCAL または IMMEDIATE のいずれかのオプションを指定した shutdown メソッドをコールする必要があります。そうしないとコールがハングします。

33.3.2 標準的なデータベース停止プロセス

データベースを停止するための標準的な方法は、次のとおりです。

1. 停止するための準備として、データベース内における以降の接続とトランザクションを禁止します。指定できる停止オプションは、CONNECT、TRANSACTIONAL、TRANSACTIONAL_LOCALまたはIMMEDIATEです。
2. 適切なALTER DATABASEコマンドをコールして、データベースをディスマウントおよびクローズします。
3. 最後に、FINALオプションを使用して停止を実行します。

データベースを可能なかぎり早く停止する必要があるような特殊な状況においては、ABORTオプションを使用することもできます。これは、SQL*PlusのSHUTDOWN ABORTに相当します。

33.4 完全な例

例33-1は、startupメソッドおよびshutdownメソッドの使用方法の例を示しています。

例33-1 データベースの起動と停止

```
import java.sql.Statement;
import java.util.Properties;
```

```
import oracle.jdbc.OracleConnection;
import oracle.jdbc.pool.OracleDataSource;
/**
 * To logon as sysdba, you need to create a password file for user "sys":
 *   orapwd file=/path/orapw password=password entries=300
 * and add the following setting in init.ora:
 *   REMOTE_LOGIN_PASSWORDFILE=EXCLUSIVE
 * then restart the database.
 */
public class DBStartup
{
  static final String DB_URL =
"jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=localhost)(PORT=5221))"
+ "(CONNECT_DATA=(SERVICE_NAME=rdbms.devplmt.XYZ.com)))";
  public static void main(String[] argv) throws Exception
  {
// Starting up the database:
    OracleDataSource ds = new OracleDataSource();
    Properties prop = new Properties();
    prop.setProperty("user","sys");
    prop.setProperty("password","manager");
    prop.setProperty("internal_logon","sysdba");
    prop.setProperty("prelim_auth","true");
    ds.setConnectionProperties(prop);
    ds.setURL(DB_URL);
    OracleConnection conn = (OracleConnection)ds.getConnection();
    conn.startup(OracleConnection.DatabaseStartupMode.NO_RESTRICTION);
    conn.close();
// Mounting and opening the database
    OracleDataSource ds1 = new OracleDataSource();
    Properties prop1 = new Properties();
    prop1.setProperty("user","sys");
    prop1.setProperty("password","manager");
    prop1.setProperty("internal_logon","sysdba");
    ds1.setConnectionProperties(prop1);
    ds1.setURL(DB_URL);
    OracleConnection conn1 = (OracleConnection)ds1.getConnection();
    Statement stmt = conn1.createStatement();
    stmt.executeUpdate("ALTER DATABASE MOUNT");
    stmt.executeUpdate("ALTER DATABASE OPEN");
    stmt.close();
    conn1.close();
// Shutting down the database
    OracleDataSource ds2 = new OracleDataSource();
    Properties prop = new Properties();
    prop.setProperty("user","sys");
    prop.setProperty("password","manager");
    prop.setProperty("internal_logon","sysdba");
    ds2.setConnectionProperties(prop);
    ds2.setURL(DB_URL);
    OracleConnection conn2 = (OracleConnection)ds2.getConnection();
    conn2.shutdown(OracleConnection.DatabaseShutdownMode.IMMEDIATE);
    Statement stmt1 = conn2.createStatement();
    stmt1.executeUpdate("ALTER DATABASE CLOSE NORMAL");
    stmt1.executeUpdate("ALTER DATABASE DISMOUNT");
    stmt1.close();
    conn2.shutdown(OracleConnection.DatabaseShutdownMode.FINAL);
    conn2.close();
```

```
    }
}
```

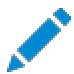
34 JDBCの診断機能

Oracle Database 12cリリース1 (12.1)の診断機能を使用すると、Oracle JDBCドライバを使用するアプリケーションの問題や、ドライバ自体の問題を診断できます。また、Oracle JDBCドライバを使用してOracle Databaseインスタンスにアクセスする Javaアプリケーションを開発、保守するために必要な労力を低減できます。

Oracle JDBCドライバには、ユーザーがJDBCアプリケーションの問題を識別および修正することを可能にする次の診断機能が用意されています。

- Oracle JDBCドライバのロギング機能
- 診断能力管理

ノート:



JDBC ドライバの診断機能は、標準の java.util.logging フレームワークおよび javax.management MBean フレームワークに基づいています。これらの標準フレームワークについての情報はこの文書では扱いません。

34.1 Oracle JDBCドライバのロギング機能について

この項では、次の概念について説明します。

- Oracle JDBCドライバのロギング機能の概要
- JDBCロギングの有効化と使用
- 実行時における機能固有のロギングの有効化または無効化
- 機能固有のロギング用のロギング構成ファイルの使用
- パフォーマンス、スケーラビリティおよびセキュリティに関する問題点

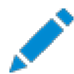
34.1.1 Oracle JDBCドライバのロギング機能の概要

JDBCドライバのコードが実行されるときに発生するイベントの情報を記録します。イベントには、SQL例外などユーザーの目に触れるイベントも、内部JDBCメソッドの出入りなどの詳細JDBC内部イベントも含めることができます。ユーザーがこの機能を有効にすると、特定のイベントまたはすべてのイベントを記録することができます。

Oracle Database 11gより前は、JDBCドライバは、J2SE 2.0と3.0をサポートしていました。J2SEのこれらのバージョンには、java.util.loggingが含まれませんでした。このため、Oracle Database 11gより前のJDBCドライバ・リリースによって提供されるロギング機能は、java.util.loggingフレームワークと異なります。

Oracle Database 11g以降、JDBCドライバではJ2SE 2.0と3.0はもうサポートされません。このため、JDBCドライバのロギング機能は、標準java.util.loggingパッケージをフル活用します。強化されたロギング・システムではログ・レベルが有効利用され、ユーザーはログ出力を関心のものに制限できます。特定のクラスの情報がより一貫して記録されるため、ログ・ファイルがわかりやすくなりました。

この機能では新しいAPIや構成ファイルは導入されません。既存の標準java.util.logging構成ファイルに新しいパラメータが追加されたのみです。これらのパラメータはjava.util.loggingの使用に不可欠なもので、既存のパラメータと同様に使用されます。



ノート:

生成されたログの内容の正確性は保証されません。ログの内容は大部分、実装の詳細に依存します。実装の細部はリリースのたびに変更されるため、ログの正確な内容もリリースごとに変更される傾向があります。

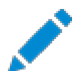
34.1.2 JDBCロギングの有効化と使用

Javaアプリケーションのデバッグを開始する前に、JDBCロギングを有効にして、構成する必要があります。この項では、JDBCロギングを有効にして使用するために実行する必要があるステップを説明します。内容は次のとおりです。

- CLASSPATHの構成について
- ロギングの有効化
- ロギングの構成
- ログ出力の使用
- ロギングの例

34.1.2.1 CLASSPATHの構成について

JDBCドライバのそれぞれのバージョンに対応するいくつかのJARファイルが付属しています。最適化されたJARファイルにはロギング・コードが含まれていないため、使用時にログ出力が生成されません。ログ出力を取得するには、デバッグJARファイル(ojdbc6_g.jarやojdbc7_g.jarのようにファイル名に_gが付いています)を使用する必要があります。デバッグJARファイルはCLASSPATHに含まれている必要があります。



ノート:

デバッグ JAR ファイル、たとえば ojdbc6_g.jar や ojdbc7_g.jar が CLASSPATH 内の唯一の Oracle JDBC JAR ファイルであることを確認してください。

34.1.2.2 ロギングの有効化

ロギングを有効化する方法には次のようなものがあります。

- Javaシステム・プロパティの設定

  oracle.jdbc.Traceシステム・プロパティを設定することによってロギングを有効にできます。

  ```
  java -Doracle.jdbc.Trace=true ...
  ```

  システム・プロパティの設定によりグローバル・ロギングを有効にできます。ロギングがアプリケーション全体に対して有効になります。アプリケーション全体をデバッグする場合、またはアプリケーションのソース・コードを変更しないか変更できない

場合は、グローバル・ロギングを使用できます。

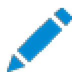
- プログラムによる有効化

ロギングをプログラム的に有効または無効にするには、次のようにします。

最初に、Diagnosability MBeanのObjectNameを取得します。ObjectNameは次のようになります

```
com.oracle.jdbc:type=diagnosability,name=<loader>
```

loaderは、Oracle JDBCドライバをロードしたクラス・ローダー・インスタンスに基づいた一意の名前です。

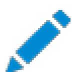


ノート:

ドライバは、1 つの仮想マシンに複数回ロードできます。したがって、それぞれ一意の名前を持つ複数の MBean が存在することになります。

次のとおりにコードを作成します。

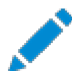
```
ClassLoader l = oracle.jdbc.OracleDriver.getClassLoader();
String loader = l.getName() + "@" + l.hashCode();
// compute the ObjectName

javax.management.ObjectName name = new
javax.management.ObjectName("com.oracle.jdbc:type=diagnosability,
name="+loader);
// get the MBean server
javax.management.MBeanServer mbs =
java.lang.management.ManagementFactory.getPlatformMBeanServer();
// find out if logging is enabled or not
System.out.println("LoggingEnabled = " + mbs.getAttribute(name, "LoggingEnabled"));
// enable logging
mbs.setAttribute(name, new javax.management.Attribute("LoggingEnabled", true));
// disable logging
mbs.setAttribute(name, new javax.management.Attribute("LoggingEnabled", false));
```



ノート:

- 同じクラス・ローダーで JDBC ドライバが複数回数をロードすると、一意の名前を作成するために、MBean が追加されるたびに l.hashCode()メソッドの値が増加します。どの MBean がどの JDBC ドライバ・インスタンスに関連付けられているかを識別することが難しくなる可能性があります。
- ロードされている JDBC ドライバのインスタンスが 1 つしかない場合は、名前を*に設定します。

プログラムでロギングを有効および無効にすると、ログ出力の生成に必要なアプリケーションの部分を変更しやすくなります。



ノート:

上のいずれかの方法を使用してロギングを有効にしても、重大なエラーの最小限のログが生成されるだけです。通

常、それではあまり役に立ちません。より有用で詳細なログを生成するには、java.util.logging を構成する必要があります。

34.1.2.3 ロギングの構成

有用で詳細なログを生成するには、java.util.loggingを構成する必要があります。これには、構成ファイルを使用する方法と、プログラム的に行う方法があります。

JDBCをインストールすると、サンプル構成ファイル、OracleLog.propertiesがdemoディレクトリに配置されます。そこにはjava.util.loggingの構成方法に関する基本情報や、最初に使用できるいくつかの初期設定が含まれています。このサンプル・ファイルをそのまま使用することも、ファイルを編集して使用することも、ファイルの名前を変更して使用することも、任意の名前を持つ完全に新しいファイルを作成することも可能です。

構成ファイルを使用するには、Javaランタイムにそれを認識させる必要があります。それには、システム・プロパティを設定します。たとえば:

```
java -Djava.util.logging.config.file=/jdbc/demo/OracleLog.properties.
```

ファイルはjava.util.loggingシステムによって読み取られます。このファイルはどこに配置してもかまいません。

java.util.logging.config.fileとoracle.jdbc.Traceは同時に使用できます。

```
java -Djava.util.logging.config.file=/jdbc/demo/OracleLog.properties -Doracle.jdbc.Trace=true
```

デフォルトのOracleLog.propertiesファイルを使用できます。希望する出力が得られる場合も、得られない場合もあります。独自の構成ファイルを作成して使用することもできます。次のステップに従ってください。

1. myConfig.propertiesという名前のファイルを作成します。どのような名前を使用してもかまいません。
2. そのファイルに次の行のテキストを挿入します。

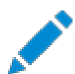
   ```
   .level=SEVERE
   oracle.jdbc.level=INFO
   oracle.jdbc.handlers=java.util.logging.ConsoleHandler
   java.util.logging.ConsoleHandler.level=INFO
   java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFormatter
   ```

3. ファイルを保存します。
4. この構成ファイルを使用するようにシステム・プロパティを設定します。

   ```
   java -Djava.util.logging.config.file=<filepath>/myConfig.properties ...
   ```

   filepathは、myConfig.propertiesファイルを保存したフォルダのパスです。

ノート:



ステップ 2 で指定した設定を使用すると、大量のログ出力が生成されます。また、ログ出力はコンソールに表示されます。

34.1.2.4 ログ出力のファイルへのリダイレクト

また、java.util.loggingを構成して、ログ出力をファイルにリダイレクトできます。そうする場合は、構成ファイルを次のように変更します。

```
.level=SEVERE
oracle.jdbc.level=INFO
oracle.jdbc.handlers=java.util.logging.FileHandler
java.util.logging.FileHandler.level=INFO
java.util.logging.FileHandler.pattern=jdbc.log
java.util.logging.FileHandler.count=1
java.util.logging.FileHandler.formatter=java.util.logging.SimpleFormatter
```

これにより、正確に同じログ出力が生成され、現在のディレクトリにjdbc.logという名前のファイルで保存されます。

レベル設定を変更することで、詳細度を制御することができます。定義されているレベルは、最も詳細度の低いものから最も高いものまで、次のようになります。

- OFF

  ロギングをオフにします。

- SEVERE

  SQLExceptionと内部エラーを記録します。

- WARNING

  SQLWarningと、好ましくないが致命的ではない内部の状況を記録します。

- INFO

  まれではあるが重要なイベントおよびエラーを記録します。生成されるログ・メッセージは比較的小量です。

- CONFIG

  実行されるSQL文字列を記録します。

- FINE

  すべてのパブリック・メソッドへの出入りが記録されるため、JDBC操作を詳細にトレースできます。生成されるログ・メッセージはかなり大量です。

- FINER

  内部メソッドのコールを記録します。

- FINEST

  大量の内部メソッドのコールを記録します。

- ALL

  詳細をすべて記録します。ロギング詳細度が最も高いレベルです。

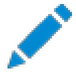
ノート:

詳細度がFINEを超えるレベルでは、非常に大量のログが生成されます。

上で示された例で詳細の出力量を削減するには、java.util.logging.FileHandler.level設定をALLからINFOに変更し

ます。

```
java.util.logging.FileHandler.level=INFO
```



ノート:

INFO では、実行される SQL 文字列を記録します。

oracle.jdbcログ出力のレベルを変更することは可能ですが、必要ありません。FileHandlerレベルを設定することで、ログ・ファイルにダンプされるログ・メッセージを制御できます。

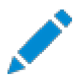
34.1.2.5 ログ出力の使用

レベルを設定すると、JDBCからのすべてのロギング出力が減少します。しかし、コードのある部分からの出力は大量に必要ですが、他の部分からはほとんど必要ない場合があります。そうするには、ログ出力に関する深い理解が必要です。

ログ出力は、名前で定義されたツリー構造をなしています。ルート・ログ出力の名前は「」(空の文字列)です。構成ファイルの最初の行には、.level=SEVEREと記述されています。これは、ルート・ログ出力のレベル設定です。次の行はoracle.jdbc.level=INFOです。これは、oracle.jdbcという名前のログ出力のレベルを設定しています。oracle.jdbcログ出力は、ログ出力ツリーのメンバーです。その親はoracleという名前です。oracleログ出力の親がルート・ログ出力(空の文字列)です。

ロギング・メッセージは、特定のログ出力(たとえば、oracle.jdbc)に送信されます。メッセージがそのレベルのレベル・チェックに合格すると、メッセージは、そのレベルのハンドラ(存在する場合)と親ログ出力に渡されます。このため、ずっと見ていくと、oracle.logに送信されるログ・メッセージはそのログ出力のレベル、INFOに対して比較されます。レベルが同じか低い(詳細度が低い)場合、FileHandlerと親ログ出力(oracle)に送信されます。この場合も、レベルに対して確認されます。この例のような場合は、レベルが設定されていないため、親レベル(SEVERE)が使用されます。メッセージ・レベルが同じか低い場合は、ハンドラに渡され(存在しません)、親に送信されます。この場合は、ルート・ログ出力の親です。このようなツリー構造は、出力の量を減らす役には立ちませんでした。役に立つのは、JDBCドライバがいくつかのサブ・ログ出力を使用するということです。ログ・メッセージをいずれかのサブ・ログ出力に制限すると、出力が大幅に少なくなります。Oracle JDBCドライバによって使用されるログ出力は、次のとおりです。

- oracle.jdbc
- oracle.jdbc.pool
- oracle.jdbc.rowset
- oracle.jdbc.xa
- oracle.sql



ノート:

ドライバが使用するログ出力は、リリースによって異なることがあります。

34.1.2.6 ロギングの例

oracle.sqlコンポーネントで発生していることを追跡し、残りのドライバに関する基本情報の一部を取得する場合を考えます。これは、ロギングのより複雑な使用方法です。configファイルのエントリを次に示します。

```
     #
     # set levels
     #
.level=SEVERE
oracle.level=INFO
oracle.jdbc.driver.level=INFO
oracle.jdbc.pool.level=OFF
oracle.jdbc.util.level=OFF
oracle.sql.level=INFO
     #
     # configure handlers
     #
oracle.handlers=java.util.logging.ConsoleHandler
java.util.logging.ConsoleHandler.level=INFO
java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFormatter
```

構成ファイルのそれぞれの行で行われている処理を検討します。

```
.level=SEVERE
```

ルート・ログ出力のロギング・レベルをSEVEREに設定します。深刻な障害の場合を除き、他の、Oracleのものでないコンポーネントからのロギングを確認する必要はありません。このため、すべてのログ出力のデフォルト・レベルをSEVEREに設定します。明示的に設定された場合を除いて、各ログ出力はレベルを親から継承します。ルート・ログ出力のレベルをSEVEREに設定すると、あらためて設定したログ出力を除いて、他のすべてのログ出力が確実にそのレベルを継承するようにできます。

```
oracle.level=INFO
```

oracle.sqlとoracle.jdbc.driverログ出力の両方からログを出力します。共通の祖先はoracleです。このため、oracleログ出力のレベルをINFOに設定します。より低いレベルでは詳細度をより明示的に制御します。

```
oracle.jdbc.driver.level=INFO
```

ここでは、oracle.jdbc.driverからのSQL実行の表示のみが必要です。このため、レベルをINFOに設定します。これはかなり小量のレベルですが、このテストの処理内容を追跡するためには役立ちます。

```
oracle.jdbc.pool.level=OFF
```

このテストではDataSourceを使用しており、そのロギングすべてを表示する必要はありません。したがって、OFFにします。

```
oracle.jdbc.util.level=OFF
```

oracle.jdbc.utilパッケージからのロギングを表示する必要はありません。XAまたはRowsetを使用していた場合、それもオフにします。

```
oracle.sql.level=INFO
```

oracle.sqlで発生していることを表示します。このため、レベルをINFOに設定します。これにより、大量の詳細が出力されるの

を回避しながら、パブリック・メソッドのコールについて多くの情報が得られます。

```
oracle.handlers=java.util.logging.ConsoleHandler
```

stderrに全内容をダンプします。テストを実行する場合、stderrをファイルにリダイレクトします。

```
java.util.logging.ConsoleHandler.level=INFO
```

System.errであるコンソールに全内容をダンプします。この場合、ハンドラでなくログ出力でフィルタリングしています。

```
java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFormatter
```

簡単でほぼ判読可能な形式を使用します。

この構成ファイルでテストを実行すると、oracle.sqlパッケージからは適度に詳細な情報を取得し、コア・ドライバ・コードからは少量の情報を取得します。他のコードからは何も取得しません。

XMLFormatterを使用して、Oracleサポートにログを送信することもできます。

カスタムjava.util.logging.Filterを実装および使用して、ログに書き込まれたデータをより詳細に制御できます。これは標準java.util.logging機能で、JSEのJavaDocに記述されています。カスタム・フィルタを使用すると、次のことが可能になります。

- マルチスレッド・アプリケーションのスレッドの取得(1つのみ)
- 長時間実行中のアプリケーションの断続的なエラーの取得

34.1.3 実行時における機能固有のロギングの有効化または無効化

Oracle Database 12cリリース2 (12.2.0.1)以降、JDBCでは実行時に選択した機能に対して機能固有のロギングの有効化または無効化がサポートされています。たとえば、ロード・バランシング機能のみのロギングを有効にして、JDBCの他の機能のロギングを無効にすることができます。また、同じ実行で高速接続フェイルオーバー機能のロギングを有効にして、ロード・バランシング機能のロギングを無効にすることもできます。

すべての機能のロギングはデフォルトで有効になっています。

JDBCのロギングの切替えノブはOracleDiagnosabilityMBeanに含まれています。このBeanを使用するには、JConsoleを起動してアプリケーションに接続します。

サポートされている機能の表示

サポートされている機能のリストを表示するには、次のメソッドを使用します。

```
getTraceController().getSupportedFeatures()
```

有効になっている機能の表示

現在有効になっている機能のリストを表示するには、次のメソッドを使用します。

```
getTraceController().getEnabledFeatures()
```

機能のロギングの有効化

特定の機能またはすべての機能のロギングを有効にするには、次のようにtraceメソッドを使用します。

```
trace(boolean enable, String feature_name)
trace(boolean enable, ALL)
```

機能のロギングの無効化

特定の機能またはすべての機能のロギングを無効にするには、次のようにtraceメソッドを使用します。

```
trace(boolean disable, String feature_name)
trace(boolean disable, ALL)
```

ロギングの一時停止および再開

ロギングを一時停止および再開するには、それぞれ、次のメソッドを使用します。

```
suspend()
resume()
```

34.1.4 機能固有のロギング用のロギング構成ファイルの使用

Oracle Database 12cリリース2 (12.2.0.1)以降、ロギング構成ファイルにプロパティを追加することにより、特定の機能のロギングを有効または無効にすることができます。ロギングはデフォルトですべての機能について有効になっています。そうでない場合、次の構文を使用してすべての機能のロギングを有効にすることができます。

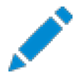
```
clio.feature.all = on
```

機能固有のロギングを有効にするには、次のプロパティを使用できます。

```
clio.feature.pool_statistics = on
clio.feature.check_in = on
clio.feature.check_out = on
clio.feature.labeling = on
clio.feature.conn_construction = on
clio.feature.conn_destruction = on
clio.feature.high_availability = on
clio.feature.load_balancing = on
clio.feature.transaction_affinity = on
clio.feature.web_affinity = on
clio.feature.data_affinity = on
clio.feature.conn_harvesting = on
clio.feature.ttl_conn_timeout = on
clio.feature.abandoned_conn_timeout = on
clio.feature.admin = on
clio.feature.sharding = on
```

34.1.5 パフォーマンス、スケーラビリティおよびセキュリティに関する問題点

ロギング機能を使用すると、アプリケーションの追跡やデバッグ、および詳細ログ出力の生成が可能になりますが、パフォーマンス、スケーラビリティおよびセキュリティに関する問題が多少あります。



注意:

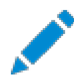
トレース・ファイルには、ユーザー名、パスワードやユーザー・データなどの機密情報が含まれていることがあります。Oracle では、このような機密情報を保護するために、本番データや資格証明が含まれる JDBC デバッグ JAR ファイルを使用しないことをお薦めします。さらに、トレース・ファイルの作成のための適切なセキュリティ・プラクティスに従うことをお薦めします。

セキュリティ上の問題

フル・ロギングを有効にすると、機密情報がトレース・ファイルに公開されるというリスクが発生します。これはロギング機能の内在的問題です。ただし、特定のJDBC JARファイルにのみ、JDBCロギング機能が含まれます。次のJARファイルにはフル・ロギングが含まれるため、本番環境での使用はお薦めしません。

- ojdbc8_g.jar
- ojdbc8dms_g.jar

ojdbc8dms.jar JARファイルには、制限付きのロギング機能が含まれています。



ノート:

データベースのユーザー名およびパスワードは、ojdbc8_g.jar および ojdbc8dms_g.jar JAR ファイルで作成されたログ・ファイルに表示されません。ただし、SQL 文、定義された値またはバインド値の一部である機密ユーザー・データは、これらの JAR ファイルのいずれかを使用して作成されたログに表示されます。

トレース・ファイルのセキュアな処理について

トレース・ファイルをセキュアに処理するには、次のようにする必要があります。

- トレース・ファイルの機密情報の量を最小限に抑えるために、実行のトレースは必要な分のみにします。
- ユーザーが所有するディレクトリでトレース・ファイルを作成します。/tmpディレクトリなどの共通のパブリック・ディレクトリにファイルを作成しないでください。
- トレース・ファイルを作成するディレクトリにUMASKを設定します。これにより、トレース・ファイルへのユーザー・アクセスが制限されます。
- java.util.logging.FileHandlerでappendオプションを有効にしないでください。これにより、トレース・ファイルに対して所有者と権限を適切に制御できます。
- ojdbc8.jarファイルの使用時に、LoggingPermissionをJDBCコード・ベースに付与しないでください。ojdbc8dms.jarファイルはログ出力が制限されており、LoggingPermissionが必要です。デバッグJARファイルojdbc8_g.jarおよびojdbc8dms_g.jarには拡張トレースが含まれており、LoggingPermissionが必要です。

パフォーマンスとスケーラビリティの問題

ロギングは、パフォーマンスにかなり影響します。本番システムでロギングが有効になっていないことを確認する必要があります。また、本番環境でデバッグJARファイルを使用しないでください。ロギングが無効の場合、パフォーマンスへの影響はありません。

ロギングには、多くの共有リソースへの保護アクセスが伴うため、結果として、スケーラビリティが大幅に低下します。これはjava.util.loggingフレームワークの問題です。

34.2 診断能力管理

JDBC診断能力管理機能には、MBean、oracle.jdbc.driver.OracleDiagnosabilityMBeanが導入されています。このMBeanを使用して、JDBCロギングの有効化および無効化が可能です。

関連項目:

OracleDiagnosabilityMBean APIの詳細は、JDBC Javadocを参照してください。

将来のリリースでは、このMBeanはJDBC内部関数に関する他の統計情報を提供するように拡張されます。

セキュリティ上の問題

この機能により、JDBCロギングを有効にできます。JDBCロギングを有効にするには、特別な権限は必要ありません。ただし、ロギングが有効になると、ログ出力を生成するために標準のJava権限LoggingPermissionが必要になります。この権限がない場合、ログを生成するすべてのJDBC処理でセキュリティ例外がスローされます。これは標準のJavaメカニズムです。

35 JDBC DMSメトリック

DMSメトリックはアプリケーション・コンポーネントのパフォーマンスを計測するために使用されます。

この章では、以下のトピックについて説明します。

- JDBC DMSメトリックの概要
- 生成されるメトリックの種類の決定について
- SQLTextメトリックの生成について
- JMXを使用したDMSメトリックへのアクセスについて

ノート:

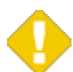
エンドツーエンド・メトリックと呼ばれる別のメトリックがあります。エンドツーエンド・メトリックは、アプリケーション・コードへのエントリから JDBC を使用してデータベースまで(およびその逆)の、アプリケーション・アクティビティにタグ付けするために使用されます。

JDBC では、次のエンドツーエンド・メトリックをサポートしています。

- `Action`
- `ClientId`
- `ExecutionContextId`
- `Module`
- `State`

以前のリリースでは、前述のメトリックを使用する場合、`oracle.jdbc.OracleConnection` インタフェースの `setEndToEndMetrics` および `getEndToEndMetrics` メソッドを使用できました。しかしながら、Oracle Database 12c リリース 1 (12.1)以降、これらのメソッドは非推奨になりました。`setEndToEndMetrics` および `getEndToEndMetrics` メソッドのかわりに、`setClientInfo` および `getClientInfo` メソッドを使用することをお薦めします。

Oracle Database 10g では、Oracle Java Database Connectivity(JDBC)は、エンドツーエンド・メトリックをサポートしています。Oracle Database 12c リリース 1 (12.1)で、エンドツーエンド・メトリックをアプリケーションが直接設定できるのは、DMS 対応の JAR ファイルを使用していない場合のみです。アプリケーションが DMS 対応の JAR ファイルを使用している場合は、DMS を経由してのみエンドツーエンド・メトリックを設定できます。



警告:

アプリケーションが DMS 対応の JAR ファイルを使用する場合、DMS メトリックを使用することを強くお薦めしま

す。

関連項目:

エンドツーエンド・メトリックの詳細は、「Oracle Database JDBC Java APIリファレンス」を参照

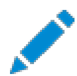
35.1 JDBC DMSメトリックの概要

DMSメトリックを使用すると、アプリケーションおよびシステム開発者は、カスタマイズされたパフォーマンス・メトリックを特定のソフトウェア・コンポーネントのために測定およびエクスポートできます。DMSメトリックはすべて、次のDMS対応のJARファイルで使用できます。

- `ojdbc6dms.jar`
- `ojdbc6dms_g.jar`
- `ojdbc7dms.jar`
- `ojdbc7dms_g.jar`

他のJDBC JARファイルはいずれもDMSメトリックを生成しません。Oracle JDBC 12cリリース1 (12.1)で生成されるメトリックは、Oracle JDBCの10.2、10.1、9.2以前のバージョンと異なっており、以前のバージョンとの互換性を維持しようとしていません。互換性モードもありません。以前のバージョンのJDBCで生成されたDMSメトリックの正確な詳細に依存しているシステムは、Oracle JDBC 12cが生成したメトリックを処理すると、想定外の動作をすることがあります。これは設計どおりの動作で、変更できません。

文メトリックは、ある接続におけるすべての文について一括してレポートすることも、各文について個々にレポートすることもできます。DMSメトリックは、個別の文に関連するメトリックを除いて、すべて常時有効です。



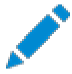
ノート:

`SQLText` 文メトリックは、有効にも無効にもできます。デフォルトでは無効です。有効の場合、すべての文で有効です。

35.2 生成されるメトリックの種類の決定について

一括メトリックと個別メトリックのどちらを使用するかを決定するために、JDBCはDMSConsoleセンサーの重みをチェックします。センサーの重みが`DMSConsole.NORMAL`以下の場合は、JDBCは一括文メトリックを生成します。センサーの重みが`DMSConsole.NORMAL`を超えている場合は、JDBCは個別文メトリックを生成します。

プリペアド文またはコール可能文を作成する場合、JDBCはDMSConsoleセンサーの重みをチェックし、文作成時点のセンサーの重みに応じて、メトリックが生成されます。文が作成された後でセンサーの重み値を変更しても、文により一括メトリックと個別メトリックが切り替えられることはありません。



ノート:

文キャッシュが存在する場合、文は新たに作成されず、キャッシュから取得されるため、センサーの重みを変更しても変化がないように見えることがあります。

一括文メトリックと個別文メトリックの両方とも、生成される文メトリックのリストは1つのみです。これら2つのリストの唯一の相違は文の集計です。個別文メトリックが生成される場合、JDBCドライバによって作成された別個の文オブジェクトごとにそれぞれ1セットのメトリックが生成されます。一方、一括文メトリックが生成される場合、指定された接続によって作成されたすべての文が、同一の文メトリック・セットを使用します。

たとえば、'execute'フェーズ・イベントを検討してみましょう。個々の文メトリックが使用される場合、作成される各々の文は異なった'execute'フェーズのイベントを持ちます。このため、そのような文2つから、2つの異なる文に対応する実行統計を識別することが可能です。片方の実行時間が1秒で、もう一方の実行時間が3秒の場合、'execute'フェーズの異なる2つのイベント(合計時間と平均時間が1秒のイベントと、合計時間と平均時間が3秒のイベント)があることになります。しかし、一括文メトリックを使用すると、すべての文が、その接続に共通の、単一の'execute'フェーズのイベントを使用することになります。このため、同じ接続によって作成されるそのような2つの文から、2つの文の実行統計を識別することができなくなります。片方の実行時間が1秒で、もう一方の実行時間が3秒の場合、'execute'フェーズが共通のイベント(合計時間が4秒で平均時間が2秒)が報告されます。

35.3 SQLTextメトリックの生成について

DMSのバージョンに応じて、SQLText文メトリックの生成を決定するメカニズムには次の2つがあります。

- DMS JARファイルの12cバージョンがclasspath環境変数に存在する場合、JDBCはDMS更新SQLテキスト・フラグをチェックします。このフラグがtrueの場合、SQLTextメトリックは更新されます。
- DMS JARファイルの12cバージョンがclasspath環境変数に存在しない場合、JDBCはDMSStatementMetrics接続プロパティの値を使用します。この文プロパティがtrueの場合、SQLTextメトリックは更新されます。この接続プロパティのデフォルト値はfalseです。

SQLTextメトリックが生成されるかどうかは、使用される文メトリックの種類、つまり個別文メトリックか一括文メトリックかには関係ありません。

35.4 JMXを使用したDMSメトリックへのアクセスについて

JMX(Java Management Extensions)は、アプリケーション、システム・オブジェクト、デバイス、サービス指向のネットワーク、JVM(Java仮想マシン)を管理および監視するツールを提供するJavaテクノロジです。JMXをサポートする管理アプリケーションを使用して、実行時に容易にDMSメトリックにアクセスできます。JMXを使用したDMSデータへのアクセスの詳細は、URL http://www.oracle.com/technetwork/middleware/toplink/overview/index.htmlを参照してください。

関連項目:

JMXの詳細は、『Oracle Database Java開発者ガイド』を参照してください。

付録

この部では、Java Database Connectivity(JDBC)のリファレンス情報、JDBCアプリケーションのコーディングのヒント、JDBCエラー・メッセージおよびJDBCアプリケーションのトラブルシューティングについて説明します。

第IX部の構成は次のとおりです。

- JDBCリファレンス情報
- Oracle RAC高速アプリケーション通知
- JDBCコーディングのヒント
- JDBCエラー・メッセージ
- トラブルシューティング

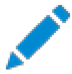
A JDBCリファレンス情報

この付録では、Java Database Connectivity(JDBC)リファレンス詳細情報について説明します。内容は次のとおりです。

- サポートされているSQLとJDBCデータ型のマッピング
- サポートされているSQLおよびPL/SQLデータ型
- PL/SQLタイプの使用について
- 埋込みJDBCエスケープ構文の使用
- Oracle JDBCのノートおよび制限事項

A.1 サポートされているSQLとJDBCデータ型のマッピング

次の表は、特定のSQLデータ型のマッピング先として有効な、すべてのJava型のリストです。Oracle JDBCドライバでは、これらの非デフォルト・マッピングをサポートしています。たとえば、SQL CHARデータをoracle.sql.CHARオブジェクトとしてインスタンス化するには、getCHARメソッドを使用します。java.math.BigDecimalオブジェクトとしてインスタンス化するには、getBigDecimalメソッドを使用します。



ノート:

oracle.jdbc.OracleData がイタリックで表示されているクラスは、Oracle JVM Web サービス・コールアウト・ユーティリティで生成できます。

表A-1 有効なSQLデータ型-Javaクラス・マッピング

SQLデータ型	Java型
CHAR、VARCHAR2、LONG	java.lang.String oracle.sql.CHAR
NUMBER	boolean char byte short int long float

SQLデータ型	Java型
	double java.lang.Byte java.lang.Short java.lang.Integer java.lang.Long java.lang.Float java.lang.Double java.math.BigDecimal oracle.sql.NUMBER
BINARY_INTEGER	boolean char byte short int long
BINARY_FLOAT	oracle.sql.BINARY_FLOAT
BINARY_DOUBLE	oracle.sql.BINARY_DOUBLE
DATE	oracle.sql.DATE
RAW	oracle.sql.RAW
BLOB	oracle.jdbc.OracleBlob 脚注 1
CLOB	oracle.jdbc.OracleClob 脚注 2
BFILE	oracle.sql.BFILE
ROWID	oracle.sql.ROWID

SQLデータ型	Java型
TIMESTAMP	oracle.sql.TIMESTAMP
TIMESTAMPWITHTIMEZONE	oracle.sql.TIMESTAMPTZ
TIMESTAMPWITHLOCALTIMEZONE	oracle.sql.TIMESTAMPLTZ
REF CURSOR	java.sql.ResultSet sqlj.runtime.ResultSetIterator
ユーザー定義の名前付き型、ADT	oracle.jdbc.OracleStruct 脚注 3
opaque 名前付き型	oracle.jdbc.OracleOpaque 脚注 4
NESTED TABLE および VARRAY 名前付き型	oracle.jdbc.OracleArray 脚注 5
名前付き型の参照	oracle.jdbc.OracleRef 脚注 6

脚注1

Oracle Database 12cリリース1 (12.1)以降、oracle.sql.BLOBクラスは非推奨となり、oracle.jdbc.OracleBlobインタフェースに置き換えられています。

脚注2

Oracle Database 12cリリース1 (12.1)以降、oracle.sql.CLOBクラスは非推奨となり、oracle.jdbc.OracleClobインタフェースに置き換えられています。

脚注3

Oracle Database 12cリリース1 (12.1)以降、oracle.sql.STRUCTクラスは非推奨となり、oracle.jdbc.OracleStructインタフェースに置き換えられています。

脚注4

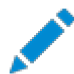
Oracle Database 12cリリース1 (12.1)以降、oracle.sql.OPAQUEクラスは非推奨となり、oracle.jdbc.OracleOpaqueインタフェースに置き換えられています。

脚注5

Oracle Database 12cリリース1 (12.1)以降、oracle.sql.ARRAYクラスは非推奨となり、oracle.jdbc.OracleArrayインタフェースに置き換えられています。

脚注6

Oracle Database 12cリリース1 (12.1)以降、oracle.sql.REFクラスは非推奨となり、oracle.jdbc.OracleRefインタフェースに置き換えられています。



ノート:

- UROWID 型はサポートされていません。
- oracle.sql.Datum は抽象クラスです。oracle.sql.Datum 型のパラメータに渡す値は、基礎となる SQL 型に対応する Java 型にする必要があります。同様に、戻り型 oracle.sql.Datum のメソッドから戻される値は、基礎となる SQL 型に対応する Java 型にする必要があります。

A.2 サポートされているSQLおよびPL/SQLデータ型

この項の表は、SQLとPL/SQLのデータ型、およびこれらのデータ型のOracle JDBCドライバのサポート状況のリストです。次の表は、SQLデータ型に対するOracle JDBCドライバのサポート状況のリストです。

表A-2 SQLデータ型に対するサポート

SQLデータ型	JDBCドライバによるサポート
BFILE	可
BLOB	可
CHAR	可
CLOB	可
DATE	可
NCHAR	不可脚注 7
NCHAR VARYING	不可
NUMBER	可
NVARCHAR2	可脚注 8
RAW	可
REF	可

SQLデータ型	JDBCドライバによるサポート
ROWID	可
UROWID	不可
VARCHAR2	可

脚注7

NCHAR型が間接的にサポートされています。対応するjava.sql.Types型はありませんが、アプリケーションがformOfUse(NCHAR)メソッドをコールする場合には、これらの型にアクセスできます。

脚注8

JSE 6では、NVARCHAR2型が間接的にサポートされています。J2SE 5.0では、NVARCHAR2型が間接的にサポートされています。対応するjava.sql.Types型はありませんが、アプリケーションがformOfUse(NCHAR)メソッドをコールする場合には、これらの型にアクセスできます。

次の表は、ANSIでサポートされているSQLデータ型に対するOracle JDBCドライバのサポート状況のリストです。

表A-3 ANSI-92 SQLデータ型に対するサポート

ANSIでサポートされているSQLデータ型	JDBCドライバによるサポート
CHARACTER	可
DEC	可
DECIMAL	可
DOUBLE PRECISION	可
FLOAT	可
INT	可
INTEGER	可
NATIONAL CHARACTER	不可
NATIONAL CHARACTER VARYING	不可

ANSIでサポートされているSQLデータ型	JDBCドライバによるサポート
NATIONAL CHAR	可
NATIONAL CHAR VARYING	不可
NCHAR	可
NCHAR VARYING	不可
NUMERIC	可
REAL	可
SMALLINT	可
VARCHAR	可

次の表は、SQLユーザー定義型に対するOracle JDBCドライバのサポート状況のリストです。

表A-4 SQLユーザー定義型に対するサポート

SQLユーザー定義型	JDBCドライバによるサポート
OPAQUE	可
参照型	可
オブジェクト型(JAVA_OBJECT)	可
NESTED TABLE 型および VARRAY 型	可

次の表は、PL/SQLデータ型に対するOracle JDBCドライバのサポート状況のリストです。PL/SQLデータ型には、次のカテゴリが含まれます。

- スカラー型
- スカラー文字列型(DATEデータ型を含みます。)
- コンポジット型
- 参照型
- ラージ・オブジェクト(LOB)型

表A-5 PL/SQLデータ型に対するサポート

PL/SQLデータ型	JDBCドライバによるサポート
スカラー型:	
BINARY INTEGER	可
DEC	可
DECIMAL	可
DOUBLE PRECISION	可
FLOAT	可
INT	可
INTEGER	可
NATURAL	可
NATURALn	不可
NUMBER	可
NUMERIC	可
PLS_INTEGER	可
POSITIVE	可
POSITIVEn	不可
REAL	可
SIGNTYPE	可
SMALLINT	可
BOOLEAN	可

PL/SQLデータ型	JDBCドライバによるサポート
スカラー文字列型	
CHAR	可
CHARACTER	可
LONG	可
LONG RAW	可
NCHAR	不可(「ノート」を参照)
NVARCHAR2	不可(「ノート」を参照)
RAW	可
ROWID	可
STRING	可
UROWID	不可
VARCHAR	可
VARCHAR2	可
DATE	可
コンポジット型	
RECORD	不可
TABLE	不可
VARRAY	可
参照型:	

PL/SQLデータ型	JDBCドライバによるサポート
REF CURSOR 型	可
オブジェクト参照型	可
LOB 型:	
BFILE	可
BLOB	可
CLOB	可
NCLOB	可

ノート:

- NATURAL、NATURALn、POSITIVE、POSITIVEn および SIGNTYPE 型は、BINARY INTEGER のサブタイプです。
- DEC、DECIMAL、DOUBLE PRECISION、FLOAT、INT、INTEGER、NUMERIC、REAL および SMALLINT 型は、NUMBER のサブタイプです。
- NCHAR 型および NVARCHAR2 型は間接的にサポートされています。対応する java.sql.Types 型はありませんが、アプリケーションが formOfUse(NCHAR)をコールする場合には、これらの型にアクセスできます。

関連トピック

- NCHAR、NVARCHAR2、NCLOBおよびdefaultNCharプロパティ

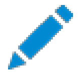
A.3 PL/SQLタイプの使用について

Oracle Database 12cリリース1 (12.1)以降、スキーマ・レベルのPL/SQLタイプを汎用のjava.sql.Structタイプとして、PL/SQLコレクションをjava.sql.Arrayタイプとしてマップできます。そのため、バインディング用にPL/SQLパッケージ・タイプにマップされるスキーマ・レベルのタイプを作成するかわりに、JDBC APIのみを使用して、PL/SQLタイプの記述およびバインドを行うことができます。

たとえば、Connection.createStruct(type_name)メソッドをコールして、まずPL/SQLタイプの記述に使用できる記述子を作成し、次にクライアントでこのタイプの新しいSTRUCT表現を作成することができます。Oracle Database 12c リリース1 (12.1)以降、type_nameを"schema.package.typename"または"package.typename"として指定して、このAPIを再利用できます。

PL/SQLパッケージ・タイプはすべて、システム全体で一意の名前にマップされます。この名前は、サーバー側のタイプ・メタデータを取得するためにJDBCで使用できます。名前の形式は次のとおりです。

```
[SCHEMA.]<PACKAGE>.<TYPE>
```



ノート:

スキーマがパッケージ名と同じ場合、また、PL/SQL タイプと同じ名前のタイプがある場合、2 つのパート名の形式(<package>.<type>)のオブジェクトを識別できません。このような場合、3 つのパート名(<schema>.<package>.<type>)を使用する必要があります。

次のコードでは、PL/SQLパッケージで宣言されたタイプをバインドする方法を説明します。

```
/*
---------------------------
# Perform the following SQL operations prior to running this sample
---------------------------
conn HR/hr;
create or replace package TEST_PKG is
   type V_TYP is varray(10) of varchar2(200);
   type R_TYP is record(c1 pls_integer, c2 varchar2(100));
   procedure VARR_PROC(p1 in V_TYP, p2 OUT V_TYP);
   procedure REC_PROC(p1 in R_TYP, p2 OUT R_TYP);
end;
/
create or replace package body TEST_PKG is
 procedure VARR_PROC(p1 in V_TYP, p2 OUT V_TYP) is
  begin
    p2 := p1;
  end;
  procedure REC_PROC(p1 in R_TYP, p2 OUT R_TYP) is
  begin
    p2 := p1;
  end;
end;
/
*/
import java.sql.Array;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Struct;
import java.sql.Types;

import oracle.jdbc.OracleConnection;
public class PLSQLTypesSample
{
  public static void main(String[] args) throws SQLException
  {
    System.out.println("begin...");
    Connection conn = null;
    oracle.jdbc.pool.OracleDataSource ods = new oracle.jdbc.pool.OracleDataSource();
    ods.setURL("jdbc:oracle:oci:localhost:5521:orcl");
    ods.setUser("HR");
```

```
    ods.setPassword("hr");
    //get connection
    conn = ods.getConnection();

    //call procedure TEST_PKG.VARR_PROC
    CallableStatement cstmt = null;
    try {
      cstmt = conn.prepareCall("{ call TEST_PKG.VARR_PROC(?,?) }");
      //PLSQL VARRAY type binding
      Array arr = ((OracleConnection)conn).createArray("TEST_PKG.V_TYP", new String[]{"A", "B"});
      cstmt.setArray(1, arr);
      cstmt.registerOutParameter(2, Types.ARRAY, "TEST_PKG.V_TYP");
      cstmt.execute();
      //get PLSQL VARRAY type out parameter value
      Array outArr = cstmt.getArray(2);
      //...
    }
    catch( Exception e) {
      e.printStackTrace();
    }finally {
      if (cstmt != null)
        cstmt.close();
    }

    //call procedure TEST_PKG.REC_PROC
    try {
      cstmt = conn.prepareCall("{ call TEST_PKG.REC_PROC(?,?) }");
      //PLSQL RECORD type binding
      Struct struct = conn.createStruct("TEST_PKG.R_TYP", new Object[]{12345, "B"});
      cstmt.setObject(1, struct);
      cstmt.registerOutParameter(2, Types.STRUCT, "TEST_PKG.R_TYP");
      cstmt.execute();
      //get PLSQL RECORD type out parameter value
      Struct outStruct = (Struct)cstmt.getObject(2);
      //...
    }
    catch( Exception e) {
      e.printStackTrace();
    }finally {
      if (cstmt != null)
        cstmt.close();
    }

    if (conn != null)
      conn.close();

    System.out.println("done!");
  }
}
```

%ROWTYPE属性を使用する各行のJavaレベルのオブジェクトの作成方法

%ROWTYPE属性を使用して、Javaレベルのオブジェクトを作成できます。この場合、表の各行はjava.sql.Structオブジェクトとして作成されます。たとえば、パッケージpack1では、次のように指定します。

関連項目:

%ROWTYPE属性の詳細は、『Oracle Database PL/SQL言語リファレンス』を参照してください。

```
CREATE OR REPLACE PACKAGE PACK1 AS

  TYPE EMPLOYEE_ROWTYPE_ARRAY IS TABLE OF EMPLOYEES%ROWTYPE;
END PACK1;
/
```

次のコードの抜粋に、JDBC APIを使用してEMPLOYEE_ROWTYPE_ARRAY配列の値を取得する方法を示します。

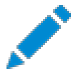
この例では、java.sql.Structオブジェクトのjava.sql.Arrayを戻します。そのStruct要素はそれぞれEMPLOYEES表の1行を表します。

例A-1 データベース表の行のためのSTRUCTオブジェクトの作成

```
CallableStatement cstmt = conn.prepareCall("BEGIN SELECT * BULK COLLECT INTO :1 FROM EMPLOYEE; END;");
cstmt.registerOutParameter(1,OracleTypes.ARRAY, "PACK1.EMPLOYEE_ROWTYPE_ARRAY");
cstmt.execute();
Array a = cstmt.getArray(1);
```

A.4 埋込みJDBCエスケープ構文の使用

Oracle JDBCドライバは、いくつかの埋込みJDBCエスケープ構文(中カッコで囲んで指定する構文)をサポートしています。現在のサポートは初歩的なものです。



ノート:

JDBC エスケープ構文は、以前は SQL92 構文または SQL92 エスケープ構文と呼ばれていました。

この項では、ドライバによって提供される次の構文のサポートについて説明します。

- 時刻および日付リテラル
- スカラー関数
- LIKEエスケープ文字
- MATCH_RECOGNIZE句
- 外部結合
- ファンクション・コール構文

ドライバのサポートが制限されている場合、これらの項では、選択可能な回避策についても説明します。

エスケープ処理の無効化

JDBCエスケープ構文の処理はデフォルトで有効です。このため、SQLコードをデータベースに送信する前にJDBCドライバがエスケープ置換を実行します。ドライバが通常のOracle SQL構文(JDBCエスケープ構文処理より効率的です)を使用するようにするには、次の文を使用します。

```
stmt.setEscapeProcessing(false);
```

A.4.1 時刻および日付リテラル

日付、時刻およびタイムスタンプのリテラルに使用する構文は、データベースによって異なります。JDBCでは、特定の形式で記述された日付および時刻のみがサポートされています。この項では、SQL文内で使用する必要のある日付、時刻およびタイムスタンプのリテラルについて説明します。

A.4.1.1 日付リテラル

JDBCドライバは、次の形式で記述されたSQL文内の日付リテラルをサポートしています。

```
{d 'yyyy-mm-dd'}
```

yyyy-mm-ddは、年、月および日を表します。たとえば:

```
{d '1995-10-22'}
```

JDBCドライバは、このエスケープ句を等価のOracleの表現「22 OCT 1995」に置換します。

次のコードの抜粋には、SQL文内での日付リテラルの使用例が含まれています。

```
// Connect to the database
// You can put a database name after the @ sign in the connection URL.
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:oci:@");
ods.setUser("HR");
ods.setPassword("hr");
Connection conn = ods.getConnection();
// Create a Statement
Statement stmt = conn.createStatement ();
// Select the first name column from the employees table where the hire date is Jan-23-1982
ResultSet rset = stmt.executeQuery
                ("SELECT first_name FROM employees WHERE hire_date = {d '1982-01-23'}");
// Iterate through the result and print the employee names
while (rset.next ())
   System.out.println (rset.getString (1));
```

A.4.1.2 時刻リテラル

JDBCドライバは、次の形式で記述されたSQL文内の時刻リテラルをサポートしています。

```
{t 'hh:mm:ss'}
```

hh:mm:ssは、時、分および秒を表します。たとえば:

```
{t '05:10:45'}
```

JDBCドライバは、このエスケープ句を等価のOracleの表現「05:10:45」に置換します。

次のように時刻が指定されているとします。

```
{t '14:20:50'}
```

サーバーが24時間制のクロックを使用しているとすれば、等価のOracleの表現は「14:20:50」になります。

次のコードの抜粋には、SQL文内での時刻リテラルの使用例が含まれています。

```
ResultSet rset = stmt.executeQuery
                 ("SELECT first_name FROM employees WHERE hire_date = {t '12:00:00'}");
```

A.4.1.3 タイムスタンプ・リテラル

JDBCドライバは、次の形式で記述されたSQL文内のタイムスタンプ・リテラルをサポートしています。

```
{ts 'yyyy-mm-dd hh:mm:ss.f...'}
```

yyyy-mm-dd hh:mm:ss.f...は、年、月、日、時、分および秒を表します。端数秒の部分(.f...)は省略できます。たとえば、{ts '1997-11-01 13:22:45'}は、Oracle形式ではNOV 01 1997 13:22:45と表されます。

次のコードの抜粋には、SQL文内でのタイムスタンプ・リテラルの使用例が含まれています。

```
ResultSet rset = stmt.executeQuery
    ("SELECT first_name FROM employees WHERE hire_date = {ts '1982-01-23 12:00:00'}");
```

Oracleオブジェクト型からSQL DATEデータ型へのマッピング

Oracle Database 8iおよびそれ以前のリリースではTIMESTAMPデータはサポートされませんでしたが、Oracle DATEデータには、SQL標準を拡張するtimeコンポーネントがありました。このため、JDBCドライバのOracle Database 8iおよびそれ以前のリリースは、oracle.sql.DATEをjava.sql.Timestampにマップしてtimeコンポーネントを維持していました。Oracle Database 9.0.1で初めてTIMESTAMPのサポートが含まれ、9i JDBCドライバがoracle.sql.DATEをjava.sql.Dateにマップするようになりました。このマッピングは不正確で、Oracle DATEデータのtimeコンポーネントを切り詰めました。この問題を解決するために、Oracle Database 11gリリース1では、新しいフラグmapDateToTimestampが導入されました。このフラグのデフォルト値はtrueで、これは、ドライバがoracle.sql.DATEをデフォルトでjava.sql.Timestampに正しくマップすることを意味します。不正確でも10gとの互換性があるoracle.sql.DATEからjava.sql.Dateへのマッピングを使用する場合は、mapDateToTimestampフラグの値をfalseに設定します。

ノート:

- Oracle Database 11g 以降、SQL 問合せが使用するための索引が DATE 列上にある場合、高速で正確な結果を入手するには、次のように setObject メソッドを使用する必要があります。

  ```
  Date d = parseIsoDate(val);
  Timestamp t = new Timestamp(d.getTime());
  stmt.setObject(pos, new oracle.sql.DATE(t, (Calendar)UTC_CAL.clone()));
  ```

  これは、setDate メソッドを使用する場合は Oracle DATE データの time コンポーネントが失われ、setTimestamp メソッドを使用する場合は DATE 列上の索引が使用されないためです。

- oracle.sql.DATE から java.sql.Date へのマッピングの問題を解決するために、Oracle Database 9.2 でフラグ V8Compatible が導入されました。このフラグのデフォルト値は false で、これにより

java.sql.Date データへの Oracle DATE データのマッピングが可能になっています。ユーザーは、このフラグの値を true に設定することによって、Oracle DATE データの time コンポーネントを維持できていました。このフラグは 11g 以降はサポートされなくなりました。それにより制御される Oracle Database 8i との互換性がもうサポートされないためです。

A.4.2 スカラー関数

Oracle JDBCドライバでサポートしていないスカラー関数もあります。ドライバがサポートする関数を調べるには、Oracle固有のoracle.jdbc.OracleDatabaseMetaDataクラスおよび標準Javaのjava.sql.DatabaseMetadataインタフェースでサポートされている次のメソッドを使用します。

- getNumericFunctions()

  ドライバによってサポートされている数値演算関数をカンマで区切られたリストで戻します。たとえば、「ABS, COS, SQRT」です。

- getStringFunctions()

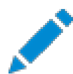
  ドライバによってサポートされている文字列関数をカンマで区切られたリストで戻します。たとえば、「ASCII, LOCATE」です。

- getSystemFunctions()

  ドライバによってサポートされているシステム関数をカンマで区切られたリストで戻します。たとえば、「DATABASE, USER」です。

- getTimeDateFunctions()

  ドライバによってサポートされている時刻および日付関数をカンマで区切られたリストで戻します。たとえば、「CURDATE, DAYOFYEAR, HOUR」です。



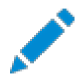
ノート:

Oracle JDBC ドライバは、ファンクション・キーワードの fn をサポートしています。

A.4.3 LIKEエスケープ文字

SQL LIKE句では、文字%および_には特別な意味があります。%は0文字以上の一致、_は1文字のみの一致に使用します。これらの文字を文字列内で文字どおりに使用する場合は、その前に特別なエスケープ文字を置きます。たとえば、アンパサンド&をエスケープ文字として使用する場合は、SQL文内では次のように識別させます。

```
Statement stmt = conn.createStatement ();
// Select the empno column from the emp table where the ename starts with '_'
ResultSet rset = stmt.executeQuery
          ("SELECT empno FROM emp WHERE ename LIKE '&_%' {ESCAPE '&'}");
// Iterate through the result and print the employee numbers
while (rset.next ())
   System.out.println (rset.getString (1));
```



ノート:

円記号(¥)をエスケープ文字として使用する場合は、2 回入力する(¥¥とする)必要があります。たとえば:

```
ResultSet rset = stmt.executeQuery("SELECT empno FROM emp
                WHERE ename LIKE '¥¥_%' {escape '¥¥'}");
```

A.4.4 MATCH_RECOGNIZE句

?の文字は、Oracle Database 11g以上のバージョンでMATCH_RECOGNIZE句でのトークンとして使用されます。JDBC標準が?文字をパラメータ・マーカーとして定義するため、JDBCドライバおよびServer SQLエンジンは同じトークンの異なる使用を区別できません。

旧バージョンのJDBCドライバでは、?文字をパラメータ・マーカーではなくMATCH_RECOGNIZEトークンとして変換する場合、PreparedStatementのかわりにStatementを使用してエスケープ処理を無効にする必要があります。ただし、Oracle Database 12cリリース1 (12.1.0.2)以降からは、?文字を使用すると同時に'{¥¥ ... ¥¥}'構文を使用することで、JDBCドライバでパラメータ・マーカーとして処理されることなく、SQLエンジンで処理できます。次のコード・スニペットは、'{¥¥ ... ¥¥}'構文の使用方法を示しています。

```
String sql =
  "select T.firstW, T.lastZ, ? " +  // use of parameter marker
  "from tkpattern_S11 " +
  "MATCH_RECOGNIZE ( " +
  "    MEASURES A.c1 as firstW, last(Z.c1) as lastZ " +
  "    ALL MATCHES " +
  "    {¥¥ PATTERN(A? X*? Y+? Z??)¥¥} " +  // use of escape sequence
  "    DEFINE " +
  "        X as X.c2 > prev(X.c2), " +
  "        Y as Y.c2 < prev(Y.c2), " +
  "        Z as Z.c2 > prev(Z.c2)" +
  ") as T";
PreparedStatement ps = conn.prepareStatatement(sql);
ps.setString(1, "test");
ResultSet rs = ps.executeQuery();
```

関連トピック

- 埋込みJDBCエスケープ構文の使用

A.4.5 外部結合

OracleのJDBCドライバは、外部結合構文をサポートしていません。回避策は、Oracle外部結合構文を使用することです。

次の構文のかわりに、

```
Statement stmt = conn.createStatement ();
ResultSet rset = stmt.executeQuery
     ("SELECT ename, dname
       FROM {OJ dept LEFT OUTER JOIN emp ON dept.deptno = emp.deptno}
       ORDER BY ename");
```

Oracle SQL構文を使用します。

```
Statement stmt = conn.createStatement ();
ResultSet rset = stmt.executeQuery
     ("SELECT ename, dname
       FROM emp b, dept a WHERE a.deptno = b.deptno(+)
       ORDER BY ename");
```

A.4.6 ファンクション・コール構文

Oracle JDBCドライバは、次のプロシージャおよびファンクション・コール構文をサポートしています。

プロシージャ・コール:

```
{ call procedure_name (argument1, argument2,...) }
```

ファンクション・コール:

```
{ ? = call procedure_name (argument1, argument2,...) }
```

A.4.7 JDBCエスケープ構文からOracle SQL構文変換例

JDBCエスケープ構文をOracle SQL構文に変換する簡単なプログラムを記述できます。次のプログラムは、ファンクション・コール、日付リテラル、時刻リテラルおよびタイムスタンプ・リテラルのためのJDBCエスケープ構文を使用する文に対して、それに対応するOracle SQL構文を出力します。このプログラムでは、oracle.jdbc.OracleSqlクラスのparse()メソッドで変換します。

```
public class Foo
{
   static oracle.jdbc.OracleDriver driver = new oracle.jdbc.OracleDriver();
   public static void main (String args[]) throws Exception
   {
      show ("{call foo(?, ?)}");
      show ("{? = call bar (?, ?)}");
      show ("{d '1998-10-22'}");
      show ("{t '16:22:34'}");
      show ("{ts '1998-10-22 16:22:34'}");
   }

   public static void show (String s) throws Exception
   {
      System.out.println (s + " => " +
         driver.processSqlEscapes(s));
   }
}
```

対応するSQL構文の出力です。

```
{call foo(?, ?)} => BEGIN foo(:1, :2); END;
{? = call bar (?, ?)} => BEGIN :1 := bar (:2, :3); END;
{d '1998-10-22'} => TO_DATE ('1998-10-22', 'YYYY-MM-DD')
{t '16:22:34'} => TO_DATE ('16:22:34', 'HH24:MI:SS')
{ts '1998-10-22 16:22:34'} => TO_TIMESTAMP ('1998-10-22 16:22:34', 'YYYY-MM-DD
HH24:MI:SS.FF')
```

A.5 Oracle JDBCのノートおよび制限事項

Oracle JDBC実装には次の制限がありますが、すべて、ほとんど影響がないか、簡単な対処法があります。この項の内容は次のとおりです。

- CursorName
- JDBC外部結合エスケープ
- IEEE 754浮動小数点との互換性
- DatabaseMetaDataコールへのCatalog引数
- SQLWarningクラス
- DDL文の実行
- 名前付きパラメータのバインド

A.5.1 CursorName

Oracle JDBCドライバは、getCursorNameメソッドおよびsetCursorNameメソッドをサポートしていません。それらをOracle構造にマップする便利な手段がないためです。かわりにROWIDを使用することをお薦めします。

関連トピック

- Oracle ROWID型

A.5.2 JDBC外部結合エスケープ

Oracle JDBCドライバでは、JDBC外部結合エスケープがサポートされません。かわりに、Oracle SQL構文で+を使用してください。

関連トピック

- 埋込みJDBCエスケープ構文の使用

A.5.3 IEEE 754浮動小数点との互換性

Oracle NUMBER型の算術演算は、浮動小数演算に関してIEEE 754規格に準拠していません。このため、Oracleによる演算結果とJavaによる演算結果に小さな誤差が生じることがあります。

Oracleでは、10進数演算と互換性のある形式で数値を格納し、小数点以下38桁までの精度を保証しています。このため、0(ゼロ)、無限大の負数、無限大の正数を正確に表現できます。各正数に対して、同じ絶対値の負数も表せます。

10^{-30}から$(1 - 10^{-38}) * 10^{126}$ のすべての正数を38桁の完全精度で表せます。

A.5.4 DatabaseMetaDataコールへのCatalog引数

特定のDatabaseMetaDataメソッドによりcatalogパラメータが定義されます。このパラメータは、そのメソッドの選択条件の1つです。Oracleには複数カタログはありませんが、複数パッケージはあります。

関連トピック

- DatabaseMetaData TABLE_REMARKSのレポートについて

A.5.5 SQLWarningクラス

java.sql.SQLWarningクラスは、データベースのアクセス警告に関する情報を提供します。警告には、通常、警告の説明と警告を識別するコードが含まれます。警告は、オブジェクトが報告される原因になったメソッドに、通知なしで関連付けられます。Oracle JDBCドライバは、通常、SQLWarningをサポートしません。その例外として、スクロール可能な結果セットの操作ではSQL警告が生成されますが、SQLWarningインスタンスは、データベース内でなくクライアント上で作成されます。

関連トピック

- SQL例外の処理について

A.5.6 DDL文の実行

Data Definition Language (DDL)文は、Statementオブジェクトとともに実行する必要があります。PreparedStatementsオブジェクトまたはCallableStatementsオブジェクトを使用する場合、DDL文は初回の実行時にのみ有効です。SQL文が文キャッシュ内にある場合、これにより予期しない動作になる可能性があります。

A.5.7 名前付きパラメータのバインド

setXXXメソッドを使用する場合、名前によるバインドはサポートされていません。特定の環境下では、以前のバージョンのOracle JDBCドライバは、setXXXメソッドの使用時にも名前によって文変数をバインドできました。次の文では、名前付き変数EmpIdが整数314159とバインドされます。

```
PreparedStatement p = conn.prepareStatement
  ("SELECT name FROM emp WHERE id = :EmpId");
  p.setInt(1, 314159);
```

set XXXメソッドを使用して名前を基準にバインドするこの機能は、JDBC仕様には含まれず、Oracleではサポートされません。JDBCドライバは、SQLExceptionをスローしたり、予想外の結果になったりすることがあります。Oracle Database 10g のJDBCドライバ以降、名前を基準としたバインドはsetXXXAtNameメソッドを使用してサポートされています。

executeメソッドをコールするまで、ドライバはバインド値をコピーしません。このためexecuteメソッドをコールする前にバインド値を変更すると、バインド値が変更されることがあります。たとえば、次のコードを考えてみましょう。

```
PreparedStatement p;
.......
Date d = new Date(1181676033917L);
p.setDate(1, d);
d.setTime(0);
p.executeUpdate();
```

このコードはデータベースにDate(1181676033917L)でなくDate(0)を挿入します。これは、パフォーマンス上の理由でJDBCドライバ実装によりバインド値がコピーされていないためです。

関連トピック

* インタフェースoracle.jdbc.OracleCallableStatement
* インタフェースoracle.jdbc.OraclePreparedStatement

B Oracle RAC高速アプリケーション通知

Oracle Database 12cリリース1 (12.1)以降、ユニバーサル接続プールまたはアクティブ・グリッド・リンク(AGL)を備えたOracle WebLogic Serverを使用しない場合にも、Oracle RAC高速アプリケーション通知(FAN) APIを使用すると、Oracle Databaseの高可用性(HA)機能を利用できます。

ここで説明する項目は、次のとおりです。

- Oracle RAC高速アプリケーション通知の概要
- Oracle RAC高速アプリケーション通知のインストールおよび構成
- Oracle RAC高速アプリケーション通知の使用
- 接続プールの実装

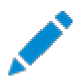
この機能はOracle Notification System(ONS)メッセージ・トランスポート・メカニズムに依存しています。ONSを使用するには、システム、サーバーおよびクライアントで構成が必要です。

Oracle RAC高速アプリケーション通知を使用するには、simplefan.jarファイルがCLASSPATHに存在するか、ons.jarファイルがCLASSPATHに存在するか、Oracle Notification Services (ONS)クライアントがクライアント・システムにインストールされ、稼働している必要があります。

B.1 Oracle RAC高速アプリケーション通知の概要

Oracle RAC高速アプリケーション通知(FAN)機能は、コールバック・メカニズムによるFANイベントへのアクセス用に簡略化されたAPIを提供します。このメカニズムを使用すると、サードパーティのドライバ、接続プールおよびコンテナでFANイベントをサブスクライブ、受信および処理できるようになります。これらのAPIを、この付録ではOracle RAC FAN APIと呼んでいます。

Oracle RAC FAN APIは、Oracle Database HA機能をフルに活用して、反応のよいアプリケーションを開発するために役立つFANイベント通知です。ユニバーサル接続プールを使用するのでなく、独自の接続プールを実装してFANイベントと連携するには、Oracle RAC高速アプリケーション通知を使用することをお薦めします。



ノート:

- 独自の接続プールを実装しない場合は、Oracle ユニバーサル接続プールを使用し、Oracle RAC 高速アプリケーション通知の利点を他の多くの利点とあわせてすべて活用することをお薦めします。
- Oracle Database 12c リリース 1 (12.1)以降、暗黙的接続キャッシュ(ICC)はサポート対象外になりました。かわりに、ユニバーサル接続プールの使用をお薦めします。

アプリケーションは、次の方法によりFANイベントに応答できるようになります。

- Oracle RACサービスの停止イベントおよびノードの停止イベントのリスニング
- Oracle RACまたはグローバル・データ・サービス(GDS)の起動または再起動を表すOracle RACサービスの起動イベントのリスニング。これらの起動イベントで、FANパラメータ・ステータスはUPで、event_typeは、database、instance、

serviceまたはservicememberのいずれかです。

- Oracle Databaseリリース12c以前のFAN ONSイベント構文およびフィールドのサポート。たとえば、各イベントに追加されたevent_typeフィールドおよびtimezoneフィールド、ノードまたはパブリックネットワーク・イベント以外のすべてのイベントに追加されたdb_domainフィールド、実行時ロード・バランシング(RLB)イベントに追加されたpercentfフィールドなどです。
- ロード・バランシング・アドバイザ・イベントのリスニングと、それに対する応答

この機能により、FANイベントが公開されます。これはOracle RACを実行しているクラスタによって送信される通知で、サブスクライバにサービス・レベルまたはノード・レベルで発生しているイベントを通知します。サポートされているFANイベントは、次のとおりです。

- サービス起動

  このサービスは、新しいインスタンスが使用可能になったため、次のインスタンスでセッションを作成できることを接続プールに通知します。ServiceUpEventクライアントAPIは、現在のリリースのOracle RAC FAN API、つまりsimplefan.jarファイルでサポートされています。

- サービス停止

  サービス停止イベントは、管理されているリソースが停止していてアクセスで使用できないことを通知します。サービス停止イベントには、次の2種類があります。

  - サービスの特定のインスタンスが停止していることを通知するイベント。このインスタンスは作業を受け入れることができません。
  - 1つを除いたすべてのサービス・インスタンスが停止していることを通知するイベント。このサービスは作業を受け入れることができません。

- ノード停止

  ノード停止イベントは、ホスト識別子で識別されるOracle RACノードが停止していて使用不可であることを通知します。クラスタは、ノードで作業を受け入れられなくなると、ノード停止イベントを送信します。

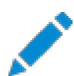
- 計画済停止

  計画済停止には、ノード停止イベント以外のすべての停止イベントが含まれます。これらのイベントには、2つのフィールド(status=downおよびreason=user)が設定されています。

- ロード・バランシング・アドバイザ

  ロード・バランシング・アドバイザ・イベントはロード・バランシング・アルゴリズムのためのメトリックを提供します。ロード・バランシング・アドバイザは、使用可能なノード間で推奨される作業分散についてサブスクライバに知らせるために定期的に送信されます。

ノート:



独自の接続プールを実装する場合にかぎり、Oracle RAC 高速アプリケーション通知を使用することをお薦めします。それ以外の場合は、Oracle ユニバーサル接続プールを使用して、Oracle RAC 高速アプリケーション通知の利点を、他の多くの利点ともに、すべて活用することをお薦めします。

関連トピック

- Oracle Universal Connection Pool開発者ガイド

B.2 Oracle RAC高速アプリケーション通知のインストールおよび構成

Oracle RAC FAN APIをインストールするには、次のステップを実行します。

1. 次のリンクからsimplefan.jarファイルをダウンロードします。:

   https://www.oracle.com/database/technologies/appdev/jdbc-downloads.html
2. simplefan.jarファイルをクラスパスに追加します。
3. Javaコードで次の操作を実行します。
   a. getInstanceメソッドを使用して、FanManagerクラスのインスタンスを取得します。
   b. FanManagerクラスのconfigureメソッドを使用して、イベント・デーモンを構成します。configureメソッドでは、次のプロパティを設定します。

      onsNodes: ONSランタイムが通信するONSデーモンのhost:portペアのカンマ区切りリスト。host:portペアのhostは、ONSデーモンを実行しているシステムのホスト名です。portは、そのデーモンのローカルのポート構成パラメータです。

      onsWalletFile: ONSウォレット・ファイルのパス名。ウォレット・ファイルは、TLS証明書の格納にTLSが使用するローカルのウォレット・ファイルのパスです。ONSデーモンのウォレット・ファイル構成パラメータと同様です。

      onsWalletPassword: ONSウォレット・ファイルにアクセスするためのパスワード。

関連項目:

- Oracle RAC FAN APIの詳細は、『Oracle Database RAC FAN Events Java API Reference』を参照してください。
- Oracle Universal Connection Pool開発者ガイド

B.3 Oracle RAC高速アプリケーション通知の使用

次のコードでは、FAN停止イベントの処理方法について説明します。このコード例では、イベント・データを標準出力デバイスに出力します。

このコード例は、handleFanEventメソッドをオーバーロードして異なるFANイベント通知を引数として受け入れることによってOracle RAC FAN APIを使用する方法を示しています。また、次のイベント・データも示しています。:

- FANイベント通知を送信するシステムの名前
- FANイベント通知のタイムスタンプ
- FANイベント通知のロード・ステータス

例B-1 FAN停止イベントにOracle RAC FAN APIを使用するサンプル・コードの例

```
...
...Properties props = new Properties();
props.putProperty( “serviceName”,  “gl”);
FanSubscription sub = FanManager.getInstance().subscribe(props);
```

```
sub.addListener(new FanEventListener()) {
  public void handleFanEvent(ServiceDownEvent se) {
    try {
      System.out.println(event.getTimestamp());
      System.out.println(event.getServiceName());
      System.out.println(event.getDatabaseUniqueName());
      System.out.println(event.getReason());
      ServiceMemberEvent me = se.getServiceMemberEvent();
      if (me != null) {
        System.out.println(me.getInstanceName());
        System.out.println(me.getNodeName());
        System.out.println(me.getServiceMemberStatus());
      }
      ServiceCompositeEvent ce = se.getServiceCompositeEvent();
      if (ce != null) {
        System.out.println(ce.getServiceCompositeStatus());
      }
    }
    catch (Throwable t) {
      // handle all exceptions and errors
      t.printStackTrace(System.err);
    }
  }
  public void handleFanEvent(NodeDownEvent ne) {
    try {
      System.out.println(event.getTimestamp());
      System.out.println(ne.getNodeName());
      System.out.println(ne.getIncarnation());
    }
    catch (Throwable t) {
      // handle all exceptions and errors
      t.printStackTrace(System.err);
    }
  }
  public void handleFanEvent(LoadAdvisoryEvent le) {
    try {
      System.out.println(event.getTimestamp());
      System.out.println(le.getServiceName());
      System.out.println(le.getDatabaseUniqueName());
      System.out.println(le.getInstanceName());
      System.out.println(le.getPercent());
      System.out.println(le.getServiceQuality());
      System.out.println(le.getLoadStatus());
    }
    catch (Throwable t) {
      // handle all exceptions and errors
      t.printStackTrace(System.err);
    }
  }
});
```

例B-2 FAN起動イベントにOracle RAC FAN APIを使用するサンプル・コードの例

次のコードでは、サービス起動イベントにOracle RAC高速アプリケーション通知を使用する方法について説明します。このコードでは、Oracle Database 12cリリース2 (12.2.0.1)で導入された新しいOracle RAC FAN API (FanEventListenerインタフェースを拡張したFanUpEventListener、ServiceUpEvent)を使用しています。FanEventListenerインタフェースを実

装したように、クライアント・アプリケーションでFanUpEventListenerインタフェースを実装する必要があります。

```
import oracle.simplefan.*;
...
FanEventListener fanListener = new FanUpEventListener() {
 public void handleEvent(ServiceUpEvent event) { ...... }
// Specify the next action here, when the node comes up
public void handleEvent(NodeUpEvent event) { ...... }
    ......
}
FanManager fanMngr = FanManager.getInstance();
Properties onsProps = new Properties();
onsProps.setProperty("onsNodes", ......);
fanMngr.configure(onsProps);
Properties subscriptionProps = new Properties();
subscriptionProps.setProperty("serviceName", ......);
fanSubscription = fanMngr.subscribe(subscriptionProps);
fanSubscription.addListener(fanListener);
...
```

B.4 接続プールの実装

Oracle RAC FAN APIを使用するには、独自の接続プールを実装する必要があります。次の点を検討してから、Oracle RAC FAN APIを使用するために接続プールを実装します。

- Oracle RAC FAN APIでは、FANイベントのサブセットを提供します。
- Oracle RAC FAN APIは、ONSイベントのみサポートします。対応するスーパークラスタ・イベントをアプリケーションでサポートする場合、サブスクリプション・プロパティへの追加が必要になることがあります。

C JDBCコーディングのヒント

この付録では、Java Database Connectivity (JDBC)アプリケーションの最適化の方法について説明します。次の内容について説明します。

- JDBCとマルチスレッド
- JDBCプログラムのパフォーマンスの最適化
- JDBCのトランザクション分離レベルとアクセス・モード

C.1 JDBCとマルチスレッド

Oracle JDBCドライバは、Javaマルチスレッドを使用するアプリケーションを完全にサポートし、それに対応するように最適化されています。接続キャッシュによって提供されるアクセスなど、接続に対する制御されたシリアル・アクセスが必要であり、そのようなアクセス方法をお薦めします。ただし、複数スレッド間でのデータベース接続の共有はお薦めしません。複数のスレッドが1つの接続に同時にアクセスできないようにしてください。複数のスレッドで接続を共有する必要がある場合は、規則的な使用開始/使用終了の方法を使用してください。

マルチスレッド・アプリケーションでの作業時には、次の点に注意してください。

- Connectionオブジェクトをローカル変数として使用します。
- メソッドを終了する前にfinallyブロックで接続をクローズします。たとえば:

```
Connection conn = null;
try
{
    ...
}
finally
{
   if(conn != null) conn.close();
}
```

- スレッド間でConnectionオブジェクトを共有しないでください。
- 同期は内部でドライバによって行われるので、JDBCオブジェクトを同期しないでください。
- 別のスレッドから時間がかかる問合せを取り消すのではなく、Statement.setQueryTimeoutメソッドを使用して問合せを実行する時間を設定します。
- SELECT、UPDATEまたはDELETEなどのSQL操作のためにStatement.cancelメソッドを使用します。
- COMMIT、ROLLBACKなどのSQL操作のためにConnection.cancelメソッドを使用します。
- Thread.interruptメソッドを使用しないでください。

C.2 JDBCプログラムのパフォーマンスの最適化

次の機能を使用すると、JDBCプログラムのパフォーマンスを大幅に向上させることができます。

- 自動コミット・モードの無効化

- 標準フェッチ・サイズとOracle行プリフェッチ
- セッション・データ・ユニット・サイズの設定について
- JDBCバッチ更新機能
- 文キャッシング
- 組込みSQL型とJava型間のマッピング

C.2.1 自動コミット・モードの無効化

自動コミット・モードは、SQL操作を実行するたびに自動的にCOMMIT操作を発行するかどうかを、データベースに対して指示します。自動コミット・モードにすると、異なるバインド変数で同じ文を繰り返すような場合などに、時間と処理能力の面で大きな負荷がかかる場合があります。

デフォルトでは、新規の接続オブジェクトは自動コミット・モードが有効になります。ただし、接続オブジェクトjava.sql.Conectionまたはoracle.jdbc.OracleConnectionのどちらかのsetAutoCommitメソッドで自動コミット・モードを無効にできます。

自動コミット・モードでは、文が完了した時点または次の実行が発生した時点のうち、どちらか早い方でCOMMIT操作が発生します。ResultSetオブジェクトを戻す文の場合は、結果セットの最後の行が取り出されたとき、または結果セットがクローズしたときに文が完了します。より複雑なケースでは、1つの文で出力パラメータ値や複数の結果が返されることがあります。この場合、すべての結果および出力パラメータ値が取り出された時点でCOMMITが発生します。

setAutoCommit(false)をコールして自動コミット・モードを無効にした場合、接続オブジェクトのcommitまたはrollbackメソッドを使用して、操作のグループを手動でコミットまたはロールバックする必要があります。

例

次に、ドライバをロードしてデータベースに接続する例を示します。新しい接続はデフォルトで自動コミット・モードが有効になるので、この例では自動コミットを無効にする方法を示します。この例では、connはConnectionオブジェクトを、stmtはStatementオブジェクトを表します。

```
// Connect to the database
// You can put a database host name after the @ sign in the connection URL.
   OracleDataSource ods = new OracleDataSource();
   ods.setURL("jdbc:oracle:oci:@");
   ods.setUser("HR");
   ods.setPassword("hr");
   Connection conn = ods.getConnection();

// It's faster when auto commit is off
conn.setAutoCommit (false);
// Create a Statement
Statement stmt = conn.createStatement ();
...
```

C.2.2 標準フェッチ・サイズとOracle行プリフェッチ

Oracle JDBC接続オブジェクトや文オブジェクトでは、データベースにアクセスするたびにクライアントにプリフェッチする行数を指

定できますが、結果セットは問合せ時に設定されます。接続オブジェクトに値を設定して、その接続で作成される各々の文に適用することも、任意の個別の文オブジェクト内でその値をオーバーライドすることもできます。接続オブジェクトのデフォルト値は、10です。クライアントにデータをプリフェッチすると、サーバーへのラウンドトリップの回数を減らすことができます。

JDBC 2.0では、同様かつさらに柔軟に、文オブジェクトにも(後続の問合せに影響)、結果セット・オブジェクトにも(行の再フェッチに影響)、1回のラウンドトリップでフェッチする行の数を指定できます。デフォルトでは、その結果セットを作成した文オブジェクトの値が、結果セットで使用されます。JDBC 2.0フェッチ・サイズを設定しないと、Oracle接続行プリフェッチ値がデフォルトとして使用されます。

関連トピック

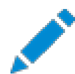
- 行フェッチ・サイズ

C.2.3 セッション・データ・ユニット・サイズの設定について

セッション・データ・ユニット(SDU)は、Oracle Netがネットワーク間でデータを転送する前にデータを格納するバッファです。Oracle Netがバッファ内のデータを送信するのは、リクエストが完了したときか、バッファがいっぱいのときです。

SDUを構成すると、特に次の利点があります。

- ネットワーク間でのSQL問合せおよび結果の転送に要する時間の削減
- 大量のデータの転送



ノート:

SDU サイズの設定を大きくすると、クライアント・プロセスおよびサーバー・プロセスのフットプリントは増加します。

関連項目:

『Oracle Database Net Services管理者ガイド』

ここでは、以下の項目について説明します。

- データベース・サーバーのSDUサイズの設定について
- JDBC ThinクライアントのSDUサイズの設定について

C.2.3.1 データベース・サーバーのSDUサイズの設定について

データベース・サーバーのSDUサイズを設定するには、sqlnet.oraファイルでDEFAULT_SDU_SIZEパラメータを構成します。

C.2.3.2 JDBC OCIクライアントのSDUサイズの設定について

JDBC OCIクライアントでは、Oracle Netレイヤーが使用されます。そのため、sqlnet.oraファイルでDEFAULT_SDU_SIZEパラメータを構成すると、JDBC OCIクライアントのSDUサイズを設定できます。

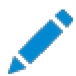
C.2.3.3 JDBC ThinクライアントのSDUサイズの設定について

特定の接続記述子のDESCRIPTIONパラメータに指定すると、JDBC ThinクライアントのSDUサイズを設定できます。

```
sales.example.com=
(DESCRIPTION=
     (SDU=11280)
        (ADDRESS=(PROTOCOL=tcp)(HOST=sales-server)(PORT=5221))
     (CONNECT_DATA=
       (SERVICE_NAME=sales.example.com))
)
```

C.2.4 JDBCバッチ更新機能

Oracle JDBCドライバでは、プリペアド文のINSERT、DELETEおよびUPDATEの各操作をクライアントに蓄積し、サーバーに一括送信できます。この機能により、サーバーへのラウンドトリップの回数が低下します。

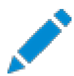


ノート:

バッチ・サイズを 100 以下の範囲に保つことをお薦めします。バッチが大きくなると、パフォーマンスはほとんど、またはまったく向上せず、実際には、大きなバッチを処理するために必要なクライアント・リソースのために、パフォーマンスが低下する可能性があります。

C.2.5 文キャッシュ

文キャッシュにより、繰り返しコールされるループやメソッドなどで何度も使用する実行文がキャッシュされるため、パフォーマンスが向上します。アプリケーションでは、特定の物理接続に関連付けられている文をキャッシュするために文キャッシュを使用します。文キャッシュを有効にすると、closeメソッドをコールするときに文オブジェクトがキャッシュされます。物理接続ごとに独自のキャッシュがあるため、複数の物理接続に対して文キャッシュを有効にすると複数のキャッシュが存在することになります。



ノート:

Oracle JDBC ドライバは、Oracle 文キャッシュで使用するために最適化されています。Oracle 文キャッシュ(暗黙的または明示的)を使用することを強くお薦めします。

接続キャッシュで文キャッシュを有効にすると、基礎となる物理接続で有効な文キャッシュが論理接続で利用されます。接続キャッシュによって保持されている論理接続で文キャッシュを有効にしようとすると、例外が発生します。

関連トピック

- 文キャッシュと結果セット・キャッシュ

C.2.6 組込みSQL型とJava型間のマッピング

SQL組込み型とはNUMBER、CHARなど、システム定義の名前を持つ型のことで、ユーザー定義の名前を持つ、Oracleオブジェクト、varrayおよびネストされた表型とは異なります。組込みSQL型のデータにアクセスするJDBCプログラムでは、プログラム・コン

テキストによりSQLデータの変換先Java型が決定されるため、型変換はすべて一意です。

表C-1 SQLデータ型とSQLデータ型を表すJavaクラスのマッピング

SQLデータ型	ORACLEマッピング - SQLデータ型を表すJavaクラス
CHAR	oracle.sql.CHAR
VARCHAR2	oracle.sql.CHAR
DATE	oracle.sql.DATE
DECIMAL	oracle.sql.NUMBER
DOUBLE PRECISION	oracle.sql.NUMBER
FLOAT	oracle.sql.NUMBER
INTEGER	oracle.sql.NUMBER
REAL	oracle.sql.NUMBER
RAW	oracle.sql.RAW
LONG RAW	oracle.sql.RAW
REF CURSOR	java.sql.ResultSet
CLOB LOCATOR	oracle.jdbc.OracleClob 脚注 1
BLOB LOCATOR	oracle.jdbc.OracleBlob 脚注 2
BFILE	oracle.sql.BFILE
ネストした表	oracle.jdbc.OracleArray 脚注 3
varray	oracle.jdbc.OracleArray
SQL オブジェクト値	型マップにオブジェクト値のエントリがない場合: ● oracle.jdbc.OracleStruct 脚注 4 型マップにオブジェクト値のエントリがある場合:

SQLデータ型	ORACLEマッピング - SQLデータ型を表すJavaクラス
	● カスタマイズされた Java クラス
SQL オブジェクト型への REF	通常、oracle.jdbc.OracleRef を実装することにより、oracle.sql.SQLRef を実装するクラス脚注 5

脚注1

Oracle Database 12cリリース1 (12.1)以降、oracle.sql.CLOBクラスは非推奨となり、oracle.jdbc.OracleClobインタフェースに置き換えられています。

脚注2

Oracle Database 12cリリース1 (12.1)以降、oracle.sql.BLOBクラスは非推奨となり、oracle.jdbc.OracleBlobインタフェースに置き換えられています。

脚注3

Oracle Database 12cリリース1 (12.1)以降、oracle.sql.ARRAYクラスは非推奨となり、oracle.jdbc.OracleArrayインタフェースに置き換えられています。

脚注4

Oracle Database 12cリリース1 (12.1)以降、oracle.sql.STRUCTクラスは非推奨となり、oracle.jdbc.OracleStructインタフェースに置き換えられています。

脚注5

Oracle Database 12cリリース1 (12.1)以降、oracle.sql.REFクラスは非推奨となり、oracle.jdbc.OracleRefインタフェースに置き換えられています。

数値データにアクセスする最も効率的な方法は、int、float、longおよびdoubleなどのプリミティブJava型としてアクセスする方法です。ただし、これらの型の範囲は、SQL NUMBERデータ型の値の範囲と正確には一致しません。このため、一部の情報が失われる可能性があります。値の範囲全体で絶対的な精度が必要である場合は、BigDecimal型を使用します。

文字データはすべて、JavaのUCS2文字セットに変換されます。文字データにアクセスする最も効率的な方法は、java.lang.Stringとしてアクセスする方法です。最も問題となるのは、データベース文字セットの2つ以上の文字が1つのUCS2キャラクタにマップされる場合に、情報が失われる可能性があることです。Oracle Database 11g以降、文字セットのすべての文字がUCS2文字セットの文字にマップされます。ただし、一部の文字はサロゲート・ペアにマップされます。

C.3 JDBCのトランザクション分離レベルとアクセス・モード

読取り専用接続は、Oracle JDBCドライバではサポートされていますが、Oracleサーバーではサポートされていません。

トランザクションの場合、Oracleサーバーでサポートしているトランザクション分離レベルはTRANSACTION_READ_COMMITTEDおよびTRANSACTION_SERIALIZABLEのみです。デフォルトはTRANSACTION_READ_COMMITTEDです。レベルの取出しおよび設定を

行うには、oracle.jdbc.OracleConnectionクラスの次のメソッドを使用します。

- getTransactionIsolation: 現在のトランザクション分離レベルの接続を取得します。
- setTransactionIsolation: TRANSACTION_READ_COMMITTED値またはTRANSACTION_SERIALIZABLE値のいずれかを使用して、トランザクション分離レベルを変更します。

D JDBCエラー・メッセージ

この付録では、Java Database Connectivity(JDBC)エラー・メッセージの一般構造について説明し、Oracle JDBCドライバによって戻されることがある一般JDBCエラー・メッセージとTTCエラー・メッセージのリストを示します。この付録の構成は次のとおりです。

- JDBCエラー・メッセージの一般構造
- 一般JDBCメッセージ
- ネイティブXAメッセージ
- TTCメッセージ

各メッセージ・リストは、まずORA番号順、次に五十音順にソートされています。

D.1 JDBCエラー・メッセージの一般構造

一般的なJDBCエラー・メッセージ構造では、次のようにメッセージの末尾にコロンを付けて、ランタイム情報を追加できます。

```
<error_message>:<extra_info>
```

たとえば、「closed statement」エラーは次のように出力されることがあります。

```
Closed Statement:next
```

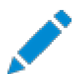
これは(結果セット・オブジェクトの)nextメソッドのコール中に例外がスローされたことを示します。

場合によっては、スタック・トレースに同様の情報が見つかることがあります。

D.2 一般JDBCメッセージ

この項では、まずORA番号順、次に五十音順にソートした一般JDBCエラー・メッセージのリストを示しています。

- ORA番号でソートしたJDBCメッセージ
- 五十音順にソートしたJDBCメッセージ



ノート:

エラー・メッセージ ORA-17033 および ORA-17034 では、SQL92 という用語が使用されています。JDBC エスケープ構文は、以前は SQL92 構文または SQL92 エスケープ構文と呼ばれていました。

D.2.1 ORA番号でソートしたJDBCメッセージ

次の表では、ORA番号順にソートしたJDBCエラー・メッセージを示しています。

表D-1 ORA番号でソートしたJDBCメッセージ

ORA番号	メッセージ
ORA-17001	内部エラー
ORA-17002	I/O 例外です
ORA-17003	列索引が無効です
ORA-17004	列の型が無効です
ORA-17005	サポートされない列の型です
ORA-17006	列名が無効です
ORA-17007	動的列が無効です
ORA-17008	クローズされた接続です
ORA-17009	クローズされた文です
ORA-17010	クローズされた結果セットです
ORA-17011	空の結果セットです
ORA-17012	パラメータの型が競合します
ORA-17014	ResultSet.next はコールされませんでした。
ORA-17015	文は取り消されました
ORA-17016	文は時間切れになりました
ORA-17017	カーソルはすでに初期化済です
ORA-17018	無効なカーソルです
ORA-17019	1 つの問合せのみ記述できます。
ORA-17020	行のプリフェッチが無効です

ORA番号	メッセージ
ORA-17021	定義がありません
ORA-17022	索引に定義がありません。
ORA-17023	サポートされない機能です
ORA-17024	読み込むデータがありません
ORA-17025	defines.isNull()でエラーが発生しました
ORA-17026	数値のオーバーフローです
ORA-17027	ストリームはすでにクローズ済です。
ORA-17028	現行の結果セットがクローズされるまで、新規結果セットを実行できません。
ORA-17029	setReadOnly：読取り専用の接続はサポートされません。
ORA-17030	READ_COMMITTED および SERIALIZABLE のみが有効なトランザクション・レベルです。
ORA-17031	setAutoClose：自動クローズ・モードが「オン」の場合のみサポートされます。
ORA-17032	行のプリフェッチをゼロに設定できません。
ORA-17033	不完全な SQL92 文字列です - 位置
ORA-17034	サポートされない SQL92 トークンです - 位置
ORA-17035	サポートされない文字セットです。
ORA-17036	OracleNumber で例外が発生しました
ORA-17037	UTF8 と UCS2 との間の変換に失敗しました。
ORA-17038	バイト配列の長さが不十分です。
ORA-17039	文字配列の長さが不十分です。

ORA番号	メッセージ
ORA-17040	接続 URL にサブ・プロトコルの指定が必要です。
ORA-17041	IN または OUT パラメータがありません - 索引
ORA-17042	バッチの値が無効です
ORA-17043	ストリームの最大サイズが無効です。
ORA-17044	内部エラー: データ配列を割当てできません。
ORA-17045	内部エラー: バッチの値範囲を超えてバインド変数にアクセスしようとしました。
ORA-17046	内部エラー: データ・アクセスに対する索引が無効です。
ORA-17047	型記述子の解析でエラーが発生しました。
ORA-17048	型が定義されていません
ORA-17049	JAVA と SQL のオブジェクト型が適合しません。
ORA-17050	ベクトルにそのような要素はありません。
ORA-17051	この API は、UDT 以外の型に使用できません。
ORA-17052	この参照は有効ではありません。
ORA-17053	サイズが無効です。
ORA-17054	LOB ロケータが無効です。
ORA-17055	無効なキャラクタが見つかりました -
ORA-17056	サポートされない文字セットです(orai18n.jar をクラスパスに追加してください)
ORA-17057	クローズされた LOB です
ORA-17058	内部エラー: NLS 変換率が無効です。

ORA番号	メッセージ
ORA-17059	内部表現の変換に失敗しました。
ORA-17060	記述子の構成に失敗しました。
ORA-17061	記述子がありません
ORA-17062	参照カーソルが無効です。
ORA-17063	トランザクション中ではありません。
ORA-17064	構文が無効、またはデータベース名が NULL です。
ORA-17065	変換クラスが NULL です。
ORA-17066	アクセス・レイヤーには固有の実装が必要です。
ORA-17067	無効な Oracle URL が指定されました。
ORA-17068	コールに無効な引数があります。
ORA-17069	明示的な XA コールを使用してください。
ORA-17070	データ・サイズがこの型の最大サイズを超えています。
ORA-17071	最大 VARRAY 制限を超えました。
ORA-17072	列に対して挿入値が大きすぎます。
ORA-17074	名前パターンが無効です
ORA-17075	転送専用の結果セットに対する操作が無効です。
ORA-17076	読取り専用の結果セットに対する操作が無効です。
ORA-17077	REF 値の設定に失敗しました。
ORA-17078	接続はすでにオープンしているため、操作できません。

ORA番号	メッセージ
ORA-17079	既存のユーザー資格証明と一致しません。
ORA-17080	無効なバッチ・コマンドです
ORA-17081	バッチ処理でエラーが発生しました。
ORA-17082	現行の行がありません
ORA-17083	挿入行ではありません。
ORA-17084	挿入行でコールされました。
ORA-17085	値の競合が発生しました
ORA-17086	挿入行の列値が未定義です。
ORA-17087	パフォーマンス・ヒントが無視されました: setFetchDirection()
ORA-17088	リクエストした結果セットの型と同時実行レベルの構文はサポートされていません。
ORA-17089	内部エラー
ORA-17090	操作できません
ORA-17091	リクエストした型および/または同時実行レベルで結果セットを作成できません。
ORA-17092	コール処理の終了時に JDBC 文を作成または実行できません。
ORA-17093	OCI 操作で OCI_SUCCESS_WITH_INFO が戻りました。
ORA-17094	オブジェクト・タイプのバージョンが適合しません。
ORA-17095	文のキャッシュ・サイズが設定されていません。
ORA-17096	文のキャッシュはこの論理接続に対して使用可能にできません。
ORA-17097	PL/SQL の索引表の要素タイプが無効です。

ORA番号	メッセージ
ORA-17098	空の LOB 操作は無効です。
ORA-17099	PL/SQL の索引表の配列の長さが無効です。
ORA-17100	データベースの Java オブジェクトが無効です。
ORA-17101	OCI 接続プール・オブジェクトに無効なプロパティがあります。
ORA-17102	Bfile は読取り専用です
ORA-17103	getConnection 経由で戻る接続タイプは無効です。かわりに、getJavaSqlConnection を使用してください
ORA-17104	実行する SQL 文は空または NULL にできません
ORA-17105	接続セッションのタイムゾーンが設定されていません
ORA-17106	指定された JDBC-OCI ドライバの接続プール構成が無効です
ORA-17107	無効なプロキシ・タイプが指定されました
ORA-17108	defineColumnType に最大長が指定されていません
ORA-17109	標準の Java 文字エンコーディングが見つかりません
ORA-17110	実行が警告で完了しました
ORA-17111	無効な接続キャッシュ TTL のタイムアウトが指定されました。
ORA-17112	指定されたスレッド間隔が無効です
ORA-17113	スレッド間隔値が、キャッシュのタイムアウト値より大きいです
ORA-17114	グローバル・トランザクションでローカル・トランザクション・コミットを使用できませんでした。
ORA-17115	グローバル・トランザクションでローカル・トランザクション・ロールバックを使用できませんでした。

ORA番号	メッセージ
ORA-17116	アクティブなグローバル・トランザクションで自動コミットをオンにできませんでした
ORA-17117	アクティブなグローバル・トランザクションでセーブポイントを設定できませんでした
ORA-17118	名前付きセーブポイントの ID を取得できませんでした。
ORA-17119	名前付きでないセーブポイントの名前を取得できませんでした。
ORA-17120	自動コミットがオンの状態でセーブポイントを設定できませんでした。
ORA-17121	自動コミットがオンの状態でセーブポイントにロールバックできませんでした。
ORA-17122	グローバル・トランザクションでローカル・トランザクション・セーブポイントにロールバックできません
ORA-17123	無効な文キャッシュ・サイズが指定されました。
ORA-17124	無効な接続キャッシュの Inactivity タイムアウトが指定されました
ORA-17125	明示的なキャッシュにより不適切な文の種類が戻されました。
ORA-17126	指定待機タイムアウトを経過しました
ORA-17127	指定された固定待機タイムアウトが無効です
ORA-17128	SQL 文字列が問合せではありません
ORA-17129	SQL 文字列が DML 文ではありません
ORA-17132	無効な変換がリクエストされました
ORA-17133	未使用
ORA-17134	SQL の名前指定パラメータの長さが 32 文字を超えています
ORA-17135	setXXXStream で使用されたパラメータ名が、SQL で複数回使用されています
ORA-17136	DATALINK URL の形式に誤りがあります。かわりに getString()を試行してください

ORA番号	メッセージ
ORA-17137	接続キャッシュが使用可能ではないか、またはキャッシュが使用可能な有効な DataSource ではありません
ORA-17138	接続キャッシュ名が無効です。有効な文字列で一意である必要があります
ORA-17139	接続キャッシュ・プロパティが無効です
ORA-17140	このキャッシュ名の接続キャッシュはすでに存在します
ORA-17141	このキャッシュ名の接続キャッシュは存在しません
ORA-17142	このキャッシュ名の接続キャッシュは無効です
ORA-17143	無効または失効している接続が接続キャッシュ内に見つかりました
ORA-17144	文の処理が実行されません
ORA-17145	無効な ONS イベントを受信しました
ORA-17146	無効な ONS イベントのバージョンを受信しました
ORA-17147	SQL に存在しないパラメータ名の設定を試行しました
ORA-17148	Thin でのみ実装されるメソッド
ORA-17149	これはすでにプロキシ・セッションです
ORA-17150	プロキシ・セッションの不正な引数です
ORA-17151	CLOB は大きすぎて Java 文字列に格納できません
ORA-17152	このメソッドは論理接続でのみ実装されます
ORA-17153	このメソッドは物理接続でのみ実装されます
ORA-17154	Oracle キャラクタを Unicode にマップできません
ORA-17155	Unicode を Oracle キャラクタにマップできません

ORA番号	メッセージ
ORA-17156	エンドツーエンドのメトリック値の配列サイズが無効です
ORA-17157	setString では、32766 文字未満の文字列のみ処理できます
ORA-17158	この関数では継続時間が無効です
ORA-17159	エンドツーエンドのトレースのメトリック値が長すぎます
ORA-17160	実行コンテキスト ID の順序番号が範囲外です
ORA-17161	使用されたトランザクション・モードが無効です
ORA-17162	サポートされていない holdability 値です
ORA-17163	接続キャッシュが使用可能な場合は、getXAConnection()を使用できません
ORA-17164	キャッシュが有効な物理接続からは getXAResource()をコールできません
ORA-17165	この接続のサーバーには DBMS_JDBC パッケージが事前に設定されていません
ORA-17166	PLSQL 文で fetch を実行できません
ORA-17167	PKI クラスが見つかりません。'connect /'機能を使用するには、oraclepki.jar を CLASSPATH に指定する必要があります
ORA-17168	Secret Store で問題が発生しました。ウォレット・ロケーションを調べ、オープン・ウォレット(cwallet.sso)の有無を確認してください。また、mkstore ユーティリティを使用して、このウォレットに適切な資格証明が含まれていることを確認してください
ORA-17169	ストリームを ScrollableResultSet または UpdatableResultSet にバインドできません
ORA-17170	ネームスペースは空にできません
ORA-17171	属性の長さは 30 文字を超えてはいけません
ORA-17172	属性の値は 400 文字を超えてはいけません

ORA番号	メッセージ
ORA-17173	すべてのリターン・パラメータが登録されていません
ORA-17174	サポートされているネームスペースは CLIENTCONTEXT だけです
ORA-17175	リモート ONS 構成中のエラー
ORA-17259	SQLXML は、クラスパスに XML サポート jar ファイルを見つけられません
ORA-17260	空の SQLXML を読み取ろうとしています
ORA-17261	読取り不可能な SQLXML を読み取ろうとしています
ORA-17262	書込み不可能な SQLXML に書き込もうとしています
ORA-17263	SQLXML は、そのタイプの結果を作成できません
ORA_17264	SQLXML は、そのタイプのソースを作成できません

D.2.2 五十音順にソートしたJDBCメッセージ

次の表では、五十音順にソートしたJDBCエラー・メッセージを示しています。

表D-2 五十音順にソートしたJDBCメッセージ

ORA番号	メッセージ
ORA-17066	アクセス・レイヤーには固有の実装が必要です。
ORA-17261	読取り不可能な SQLXML を読み取ろうとしています
ORA-17260	空の SQLXML を読み取ろうとしています
ORA-17147	SQL に存在しないパラメータ名の設定を試行しました
ORA-17262	書込み不可能な SQLXML に書き込もうとしています
ORA-17102	Bfile は読取り専用です
ORA-17038	バイト配列の長さが不十分です。

ORA番号	メッセージ
ORA-17084	挿入行でコールされました。
ORA-17164	キャッシュが有効な物理接続からは getXAResource()をコールできません
ORA-17028	現行の結果セットがクローズされるまで、新規結果セットを実行できません。
ORA-17163	接続キャッシュが使用可能な場合は、getXAConnection()を使用できません
ORA-17019	1 つの問合せのみ記述できます。
ORA-17169	ストリームを ScrollableResultSet または UpdatableResultSet にバインドできません
ORA-17078	接続はすでにオープンしているため、操作できません。
ORA-17154	Oracle キャラクタを Unicode にマップできません
ORA-17155	Unicode を Oracle キャラクタにマップできません
ORA-17166	PLSQL 文で fetch を実行できません
ORA-17032	行のプリフェッチをゼロに設定できません。
ORA-17039	文字配列の長さが不十分です。
ORA-17035	サポートされない文字セットです。
ORA-17151	CLOB は大きすぎて Java 文字列に格納できません
ORA-17008	クローズされた接続です
ORA-17057	クローズされた LOB です
ORA-17010	クローズされた結果セットです
ORA-17009	クローズされた文です
ORA-17140	このキャッシュ名の接続キャッシュはすでに存在します

ORA番号	メッセージ
ORA-17141	このキャッシュ名の接続キャッシュは存在しません
ORA-17142	このキャッシュ名の接続キャッシュは無効です
ORA-17137	接続キャッシュが使用可能ではないか、またはキャッシュが使用可能な有効な DataSource ではありません
ORA-17105	接続セッションのタイムゾーンが設定されていません
ORA-17065	変換クラスが NULL です。
ORA-17118	名前付きセーブポイントの ID を取得できませんでした。
ORA-17119	名前付きでないセーブポイントの名前を取得できませんでした。
ORA-17122	グローバル・トランザクションでローカル・トランザクション・セーブポイントにロールバックできません
ORA-17121	自動コミットがオンの状態でセーブポイントにロールバックできませんでした。
ORA-17120	自動コミットがオンの状態でセーブポイントを設定できませんでした。
ORA-17117	アクティブなグローバル・トランザクションでセーブポイントを設定できませんでした
ORA-17116	アクティブなグローバル・トランザクションで自動コミットをオンにできませんでした
ORA-17114	グローバル・トランザクションでローカル・トランザクション・コミットを使用できませんでした。
ORA-17115	グローバル・トランザクションでローカル・トランザクション・ロールバックを使用できませんでした。
ORA-17017	カーソルはすでに初期化済です
ORA-17070	データ・サイズがこの型の最大サイズを超えています。
ORA-17165	この接続のサーバーには DBMS_JDBC パッケージが事前に設定されていません
ORA-17158	この関数では継続時間が無効です

ORA番号	メッセージ
ORA-17168	Secret Store で問題が発生しました。ウォレット・ロケーションを調べ、オープン・ウォレット(cwallet.sso)の有無を確認してください。また、mkstore ユーティリティを使用して、このウォレットに適切な資格証明が含まれていることを確認してください
ORA-17175	リモート ONS 構成中のエラー
ORA-17025	defines.isNull()でエラーが発生しました
ORA-17047	型記述子の解析でエラーが発生しました。
ORA-17081	バッチ処理でエラーが発生しました。
ORA-17071	最大 VARRAY 制限を超えました。
ORA-17036	OracleNumber で例外が発生しました
ORA-17110	実行が警告で完了しました
ORA-17160	実行コンテキスト ID の順序番号が範囲外です
ORA-17011	空の結果セットです
ORA-17060	記述子の構成に失敗しました。
ORA-17037	UTF8 と UCS2 との間の変換に失敗しました。
ORA-17059	内部表現の変換に失敗しました。
ORA-17077	REF 値の設定に失敗しました。
ORA-17126	指定待機タイムアウトを経過しました
ORA-17087	パフォーマンス・ヒントが無視されました: setFetchDirection()
ORA-17125	明示的なキャッシュにより不適切な文の種類が戻されました。
ORA-17049	JAVA と SQL のオブジェクト型が適合しません。

ORA番号	メッセージ
ORA-17072	列に対して挿入値が大きすぎます。
ORA-17001	内部エラー
ORA-17089	内部エラー
ORA-17045	内部エラー: バッチの値範囲を超えてバインド変数にアクセスしようとしました。
ORA-17044	内部エラー: データ配列を割当てできません。
ORA-17046	内部エラー: データ・アクセスに対する索引が無効です。
ORA-17058	内部エラー: NLS 変換率が無効です。
ORA-17068	コールに無効な引数があります。
ORA-17156	エンドツーエンドのメトリック値の配列サイズが無効です
ORA-17080	無効なバッチ・コマンドです
ORA-17042	バッチの値が無効です
ORA-17055	無効なキャラクタが見つかりました -
ORA-17003	列索引が無効です
ORA-17006	列名が無効です
ORA-17004	列の型が無効です
ORA-17124	無効な接続キャッシュの Inactivity タイムアウトが指定されました
ORA-17138	接続キャッシュ名が無効です。有効な文字列で一意である必要があります
ORA-17139	接続キャッシュ・プロパティが無効です
ORA-17111	無効な接続キャッシュ TTL のタイムアウトが指定されました。

ORA番号	メッセージ
ORA-17103	getConnection 経由で戻る接続タイプは無効です。かわりに、getJavaSqlConnection を使用してください
ORA-17132	無効な変換がリクエストされました
ORA-17018	無効なカーソルです
ORA-17100	データベースの Java オブジェクトが無効です。
ORA-17007	動的列が無効です
ORA-17098	空の LOB 操作は無効です。
ORA-17127	指定された固定待機タイムアウトが無効です
ORA-17106	指定された JDBC-OCI ドライバの接続プール構成が無効です
ORA-17074	名前パターンが無効です
ORA-17145	無効な ONS イベントを受信しました
ORA-17146	無効な ONS イベントのバージョンを受信しました
ORA-17075	転送専用の結果セットに対する操作が無効です。
ORA-17076	読取り専用の結果セットに対する操作が無効です。
ORA-17143	無効または失効している接続が接続キャッシュ内に見つかりました
ORA-17067	無効な Oracle URL が指定されました。
ORA-17099	PL/SQL の索引表の配列の長さが無効です。
ORA-17097	PL/SQL の索引表の要素タイプが無効です。
ORA-17101	OCI 接続プール・オブジェクトに無効なプロパティがあります。
ORA-17107	無効なプロキシ・タイプが指定されました

ORA番号	メッセージ
ORA-17020	行のプリフェッチが無効です
ORA-17123	無効な文キャッシュ・サイズが指定されました。
ORA-17043	ストリームの最大サイズが無効です。
ORA-17064	構文が無効、またはデータベース名が NULL です。
ORA-17112	指定されたスレッド間隔が無効です
ORA-17161	使用されたトランザクション・モードが無効です
ORA-17002	I/O 例外です
ORA-17092	コール処理の終了時に JDBC 文を作成または実行できません。
ORA-17134	SQL の名前指定パラメータの長さが 32 文字を超えています
ORA-17136	DATALINK URL の形式に誤りがあります。かわりに getString()を試行してください
ORA-17033	不完全な SQL92 文字列です - 位置
ORA-17148	Thin でのみ実装されるメソッド
ORA-17159	エンドツーエンドのトレースのメトリック値が長すぎます
ORA-17021	定義がありません
ORA-17022	索引に定義がありません。
ORA-17061	記述子がありません
ORA-17041	IN または OUT パラメータがありません - 索引
ORA-17082	現行の行がありません
ORA-17024	読み込むデータがありません

ORA番号	メッセージ
ORA-17108	defineColumnType に最大長が指定されていません
ORA-17050	ベクトルにそのような要素はありません。
ORA-17056	サポートされない文字セットです(orai18n.jar をクラスパスに追加してください)
ORA-17034	サポートされない SQL92 トークンです - 位置
ORA-17173	すべてのリターン・パラメータが登録されていません
ORA-17063	トランザクション中ではありません。
ORA-17083	挿入行ではありません。
ORA-17026	数値のオーバーフローです
ORA-17094	オブジェクト・タイプのバージョンが適合しません。
ORA-17093	OCI 操作で OCI_SUCCESS_WITH_INFO が戻りました。
ORA-17090	操作できません
ORA-17135	setXXXStream で使用されたパラメータ名が、SQL で複数回使用されています
ORA-17012	パラメータの型が競合します
ORA-17167	PKI クラスが見つかりません。'connect /'機能を使用するには、oraclepki.jar を CLASSPATH に指定する必要があります
ORA-17030	READ_COMMITTED および SERIALIZABLE のみが有効なトランザクション・レベルです。
ORA-17062	参照カーソルが無効です。
ORA-17014	ResultSet.next はコールされませんでした。
ORA-17031	setAutoClose: 自動クローズ・モードが「オン」の場合のみサポートされます。

ORA番号	メッセージ
ORA-17029	setReadOnly：読取り専用の接続はサポートされません。
ORA-17157	setString では、32766 文字未満の文字列のみ処理できます
ORA-17104	実行する SQL 文は空または NULL にできません
ORA-17129	SQL 文字列が DML 文ではありません
ORA-17128	SQL 文字列が問合せではありません
ORA-17263	SQLXML は、そのタイプの結果を作成できません
ORA_17264	SQLXML は、そのタイプのソースを作成できません
ORA-17259	SQLXML は、クラスパスに XML サポート jar ファイルを見つけられません
ORA-17109	標準の Java 文字エンコーディングが見つかりません
ORA-17095	文のキャッシュ・サイズが設定されていません。
ORA-17096	文のキャッシュはこの論理接続に対して使用可能にできません。
ORA-17144	文の処理が実行されません
ORA-17016	文は時間切れになりました
ORA-17015	文は取り消されました
ORA-17027	ストリームはすでにクローズ済です。
ORA-17040	接続 URL にサブ・プロトコルの指定が必要です。
ORA-17172	属性の値は 400 文字を超えてはいけません
ORA-17171	属性の長さは 30 文字を超えてはいけません
ORA-17054	LOB ロケータが無効です。

ORA番号	メッセージ
ORA-17170	ネームスペースは空にできません
ORA-17174	サポートされているネームスペースは CLIENTCONTEXT だけです
ORA-17053	サイズが無効です。
ORA-17051	この API は、UDT 以外の型に使用できません。
ORA-17149	これはすでにプロキシ・セッションです
ORA-17152	このメソッドは論理接続でのみ実装されます
ORA-17153	このメソッドは物理接続でのみ実装されます
ORA-17052	この参照は有効ではありません。
ORA-17113	スレッド間隔値が、キャッシュのタイムアウト値より大きいです
ORA-17091	リクエストした型および/または同時実行レベルで結果セットを作成できません。
ORA-17086	挿入行の列値が未定義です。
ORA-17048	型が定義されていません
ORA-17005	サポートされない列の型です
ORA-17023	サポートされない機能です
ORA-17162	サポートされていない holdability 値です
ORA-17088	リクエストした結果セットの型と同時実行レベルの構文はサポートされていません。
ORA-17133	未使用
ORA-17069	明示的な XA コールを使用してください。
ORA-17079	既存のユーザー資格証明と一致しません。

ORA番号	メッセージ
ORA-17085	値の競合が発生しました
ORA-17150	プロキシ・セッションの不正な引数です

D.3 ネイティブXAメッセージ

次の項では、ネイティブXA機能に固有のJDBCエラー・メッセージを示します。

- ORA番号でソートしたネイティブXAメッセージ
- 五十音順にソートしたネイティブXAメッセージ

D.3.1 ORA番号でソートしたネイティブXAメッセージ

次の表では、ORA番号順にソートしたネイティブXAメッセージを示しています。

表D-3 ORA番号でソートしたネイティブXAメッセージ

ORA番号	メッセージ
ORA-17200	XA open 文字列を Java から C へ正しく変換できません
ORA-17201	XA close 文字列を Java から C へ正しく変換できません
ORA-17202	RM 名を Java から C へ正しく変換できません
ORA-17203	ポインタ・タイプを jlong にキャストできません
ORA-17204	入力配列が短かすぎて OCI ハンドルを保持できません
ORA-17205	xaoSvcCtx を使用して C-XA から OCISvcCtx ハンドルを取得するのに失敗しました
ORA-17206	xaoEnv を使用して C-XA から OCIEnv を取得するのに失敗しました
ORA-17207	tnsEntry プロパティが DataSource に設定されていません
ORA-17213	xa_open で C-XA から XAER_RMERR が戻されました
ORA-17215	xa_open で C-XA から XAER_INVAL が戻されました

ORA番号	メッセージ
ORA-17216	xa_open で C-XA から XAER_PROTO が戻されました
ORA-17233	xa_close で C-XA から XAER_RMERR が戻されました
ORA-17235	xa_close で C-XA から XAER_INVAL が戻されました
ORA-17236	xa_close で C-XA から XAER_PROTO が戻されました

D.3.2 五十音順にソートしたネイティブXAメッセージ

次の表では、五十音順にソートしたネイティブXAメッセージを示しています。

表D-4 五十音順にソートしたネイティブXAメッセージ

ORA番号	メッセージ
ORA-17203	ポインタ・タイプを jlong にキャストできません
ORA-17235	xa_close で C-XA から XAER_INVAL が戻されました
ORA-17215	xa_open で C-XA から XAER_INVAL が戻されました
ORA-17236	xa_close で C-XA から XAER_PROTO が戻されました
ORA-17216	xa_open で C-XA から XAER_PROTO が戻されました
ORA-17233	xa_close で C-XA から XAER_RMERR が戻されました
ORA-17213	xa_open で C-XA から XAER_RMERR が戻されました
ORA-17206	xaoEnv を使用して C-XA から OCIEnv を取得するのに失敗しました
ORA-17205	xaoSvcCtx を使用して C-XA から OCISvcCtx ハンドルを取得するのに失敗しました
ORA-17204	入力配列が短かすぎて OCI ハンドルを保持できません
ORA-17207	tnsEntry プロパティが DataSource に設定されていません

ORA番号	メッセージ
ORA-17202	RM 名を Java から C へ正しく変換できません
ORA-17201	XA close 文字列を Java から C へ正しく変換できません
ORA-17200	XA open 文字列を Java から C へ正しく変換できません

D.4 TTCメッセージ

この項では、まずORA番号順、次に五十音順にソートしたTTCエラー・メッセージを示します。

- ORA番号でソートしたTTCメッセージ
- 五十音順にソートしたTTCメッセージ

D.4.1 ORA番号でソートしたTTCメッセージ

次の表では、ORA番号順にソートしたTTCメッセージを示しています。

表D-5 ORA番号でソートしたTTCメッセージ

ORA番号	メッセージ
ORA-17401	プロトコル違反です
ORA-17402	RPA メッセージは 1 つのはずです。
ORA-17403	RXH メッセージは 1 つのはずです。
ORA-17404	予定より多い RXD を受け取りました。
ORA-17405	UAC の長さはゼロではありません。
ORA-17406	最大バッファ長を超えています。
ORA-17407	型表現が無効です(setRep)。
ORA-17408	型表現が無効です(getRep)。
ORA-17409	バッファ長が無効です
ORA-17410	ソケットから読み込むデータはこれ以上ありません。

ORA番号	メッセージ
ORA-17411	データ型の表現が適合しません。
ORA-17412	型の長さが最大を超えています。
ORA-17413	キー・サイズが大きすぎます
ORA-17414	列名を保存するにはバッファ・サイズが不十分です。
ORA-17415	この型は未処理です。
ORA-17416	FATAL
ORA-17417	NLS の問題で、列名のデコードに失敗しました。
ORA-17418	内部構造体のフィールド長エラーです。
ORA-17419	無効な列の数が戻りました。
ORA-17420	Oracle バージョンが定義されていません。
ORA-17421	型または接続が定義されていません。
ORA-17422	ファクトリに無効なクラスがあります。
ORA-17423	IOV の定義なしで PLSQL ブロックを使用しています。
ORA-17424	異なる配列操作を試みています。
ORA-17425	PLSQL ブロックでストリームを戻しています。
ORA-17426	IN バインド、OUT バインドともに NULL です。
ORA-17427	初期化されていない OAC を使用しています
ORA-17428	接続後にログオンのコールが必要です。
ORA-17429	少なくともサーバーに接続している必要があります。

ORA番号	メッセージ
ORA-17430	サーバーへのログオンが必要です。
ORA-17431	解析する SQL 文が NULL です。
ORA-17432	all7 でオプションが無効です。
ORA-17433	コールの引数が無効です。
ORA-17434	ストリーム・モードではありません。
ORA-17435	IOV で in_out_binds の数が無効です。
ORA-17436	アウトバインドの数が無効です。
ORA-17437	PLSQL ブロックの IN/OUT 引数にエラーがあります。
ORA-17438	内部 - 予期しない値です
ORA-17439	SQL の型が無効です
ORA-17440	DBItem/DBType が NULL です。
ORA-17441	この Oracle バージョンはサポートされません。7.2.3 以上はサポートされます。
ORA-17442	Refcursor 値が無効です。
ORA-17443	NULL のユーザーまたはパスワードは、THIN ドライバでサポートされません。
ORA-17444	サーバーから受け取った TTC プロトコル・バージョンはサポートされません。
ORA-17445	LOB は、同じトランザクションですでにオープンされています
ORA-17446	LOB は、同じトランザクションですでにクローズされています
ORA-17447	OALL8 矛盾した状態にあります

D.4.2 五十音順にソートしたTTCメッセージ

次の表では、五十音順にソートしたTTCメッセージを示しています。

表D-6 五十音順にソートしたTTCメッセージ

ORA番号	メッセージ
ORA-17424	異なる配列操作を試みています。
ORA-17412	型の長さが最大を超えています。
ORA-17426	IN バインド、OUT バインドともに NULL です。
ORA-17411	データ型の表現が適合しません。
ORA-17440	DBItem/DBType が NULL です。
ORA-17437	PLSQL ブロックの IN/OUT 引数にエラーがあります。
ORA-17413	キー・サイズが大きすぎます
ORA-17406	最大バッファ長を超えています。
ORA-17416	FATAL
ORA-17414	列名を保存するにはバッファ・サイズが不十分です。
ORA-17438	内部 - 予期しない値です
ORA-17418	内部構造体のフィールド長エラーです。
ORA-17433	コールの引数が無効です。
ORA-17409	バッファ長が無効です
ORA-17422	ファクトリに無効なクラスがあります。
ORA-17419	無効な列の数が戻りました。
ORA-17435	IOV で in_out_binds の数が無効です。
ORA-17436	アウトバインドの数が無効です。

ORA番号	メッセージ
ORA-17432	all7 でオプションが無効です。
ORA-17439	SQL の型が無効です
ORA-17408	型表現が無効です(getRep)。
ORA-17407	型表現が無効です(setRep)。
ORA-17446	LOB は、同じトランザクションですでにクローズされています
ORA-17445	LOB は、同じトランザクションですでにオープンされています
ORA-17428	接続後にログオンのコールが必要です。
ORA-17429	少なくともサーバーに接続している必要があります。
ORA-17430	サーバーへのログオンが必要です。
ORA-17417	NLS の問題で、列名のデコードに失敗しました。
ORA-17410	ソケットから読み込むデータはこれ以上ありません。
ORA-17434	ストリーム・モードではありません。
ORA-17443	NULL のユーザーまたはパスワードは、THIN ドライバでサポートされません。
ORA-17447	OALL8 矛盾した状態にあります
ORA-17402	RPA メッセージは 1 つのはずです。
ORA-17403	RXH メッセージは 1 つのはずです。
ORA-17420	Oracle バージョンが定義されていません。
ORA-17441	この Oracle バージョンはサポートされません。7.2.3 以上はサポートされます。
ORA-17401	プロトコル違反です

ORA番号	メッセージ
ORA-17404	予定より多い RXD を受け取りました。
ORA-17442	Refcursor 値が無効です。
ORA-17425	PLSQL ブロックでストリームを戻しています。
ORA-17431	解析する SQL 文が NULL です。
ORA-17415	この型は未処理です。
ORA-17444	サーバーから受け取った TTC プロトコル・バージョンはサポートされません。
ORA-17421	型または接続が定義されていません。
ORA-17405	UAC の長さはゼロではありません。
ORA-17423	IOV の定義なしで PLSQL ブロックを使用しています。
ORA-17427	初期化されていない OAC を使用しています

E トラブルシューティング

この付録では、Java Database Connectivity (JDBC)アプリケーションのトラブルシューティングについて説明します。内容は次のとおりです。

- 一般的な問題
- 基本的なデバッグ処理

E.1 一般的な問題

この項では、Oracle JDBCドライバの使用中に発生する可能性のある、一般的な問題について説明します。たとえば、次の問題があります。

- OUTまたはIN/OUT変数として定義されたCHAR列に対するメモリー消費
- メモリー・リークおよびカーソルの不足
- 1プロセスで17以上のOCI接続のオープン数について
- Statement.cancelの使用
- ファイアウォールとJDBCの使用方法
- 多数回にわたるサーバーからの突然の切断
- ネットワーク・アダプタで接続を確立できない

E.1.1 OUTまたはIN/OUT変数として定義されたCHAR列に対するメモリー消費

PL/SQLでは、CHARまたはVARCHAR2列がOUTまたはIN/OUT変数として定義される場合、ドライバは32512文字のCHAR配列を割り当てます。このため、メモリー消費に関する問題が生じます。JDBC Thinドライバは、VARCHAR2出力型を使用する場合、メモリーを割り当てません。ただし、JDBC OCIドライバは、CHARとVARCHAR2の両方の型に対してメモリーを割り当てます。このため、OCIドライバのCPU負荷はThinドライバより高くなります。

以前のリリースでは、この問題はStatement.setMaxFieldSizeメソッドをコールすることで解決できました。これよりもよい解決方法は、OracleCallableStatement.registerOutParameterを使用することです。すべてのCHARまたはVARCHAR2列で、必ずregisterOutParameter(int paramIndex, int sqlType, int scale, int maxLength)をコールすることをお薦めします。このメソッドは、oracle.jdbc.OracleCallableStatementに定義されています。4つ目の引数maxLengthを使用して、メモリー消費を制限します。このパラメータは、この列の格納に必要な文字数をドライバに指示します。文字配列に列データを保持できない場合、列は切り捨てられます。3つ目の引数scaleは、ドライバでは無視されます。

E.1.2 メモリー・リークおよびカーソルの不足

カーソルまたはメモリーが不足しているというメッセージを受け取った場合は、すべてのStatementおよびResultSetオブジェクトを明示的にクローズしてください。Oracle JDBCドライバには、ファイナライザ・メソッドがありません。クリーン・アップ・ルーチンは、ResultSetおよびStatementクラスのcloseメソッドで実行されます。結果セットおよび文オブジェクトを明示的にクローズしておかないと、重大なメモリー・リークが発生します。また、データベースのカーソルが不足します。文をクローズすると、データベース内

の対応するカーソルが解放されます。

同様に、サーバー側でのリークおよびカーソル不足を避けるには、Connectionオブジェクトも明示的にクローズしておく必要があります。接続をクローズすると、その接続に関連付けられたオープン中の文オブジェクトが、JDBCドライバによってクローズされ、サーバー側のカーソルが解放されます。

E.1.3 1プロセスで17以上のOCI接続のオープン数について

必ずしも1プロセスで約17以上のJDBC-OCI接続をオープンできないことがあります。サーバー上のプロセス数が初期化ファイルに指定されている制限を超えたためか、またはプロセスごとのファイル記述子の制限を超えた可能性があります。1つのJDBC-OCI接続で、複数のファイル記述子が使用される(3から4個のファイル記述子が使用されます)ことがあります。

サーバーで17以上のプロセスを使用可能にしている場合は、プロセスごとのファイル記述子の制限が原因である可能性があります。この制限数を増やすと、解決する可能性があります。

E.1.4 Statement.cancelの使用

JDBC標準メソッドStatement.cancelは、データベースにメッセージを送信することにより、SQL文の実行を正常に停止させます。データベースは、これに応答して実行を停止し、エラー・メッセージを返します。Statement.executeをコールしたJavaスレッドはサーバーで待機し、他のスレッドからのStatement.cancelメソッドのコールにより起動された返答のエラー・メッセージを受信した場合にのみ実行を続けます。

このため、Statement.cancelメソッドは、ネットワークとデータベースが正しく機能していることに依存します。ネットワーク接続が切断するか、データベース・サーバーがダウンした場合、クライアントは取消しメッセージに対するエラー応答を受け取りません。サーバー・プロセスに障害が発生すると、多くの場合、JDBCはStatement.executeをコールしたスレッドを解放するIOExceptionを受け取ります。ただし、サーバーがダウンしてもJDBCがIOExceptionを受け取らない場合があります。Statement.executeメソッドを開始したスレッドは、Statement.cancelメソッドによって解放されません。

ノート:

Statement.cancel メソッドの使用時には次の点に注意してください。

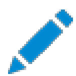
- 接続レベルの取消しと文レベルの取消しが区別されます。文の実行以外の場合、たとえば ROLLBACK が statement.cancel メソッドによって取り消された場合、コマンドがリプレイされます(ROLLBACK, COMMIT、autoCommit ON、autoCommit OFF、VERSION の場合のみ)。データの整合性を保証するために、文の実行をリプレイしません。
- 取消しのコールがデータベースに送信されるまで実行が戻されないように、文の実行と文の取消しを同期します。これによって、文の実行を取り消せるようになります。
- 取消しコールを同期します。これにより、処理中の取消しが処理を完了するまで(データベースが中断、実行および JDBC への通知を受け取った後)、新しい取消しリクエストが無視されます。

JDBCがIOExceptionを受け取らない場合は、Oracle Netがいずれはタイムアウトし、接続をクローズすることになります。これ

により、IOExceptionが発生し、スレッドが解放されます。ただし、このプロセスにはかなりの時間がかかることがあります。このタイムアウトを制御する方法は、OracleDatasource.setConnectionProperties()のreadTimeoutプロパティの説明を参照してください。このタイムアウトは、Oracle Netの設定を使用してチューニングすることもできます。

JDBC標準メソッドStatement.setQueryTimeoutは、Statement.cancelメソッドに依存します。指定されたタイムアウト間隔よりも実行時間が長くなった場合、モニター・スレッドがStatement.cancelメソッドをコールします。これは、前述と同じすべての制限を受けます。この結果、Statement.executeメソッドをコールしたスレッドがタイムアウトにより解放されない場合があります。

実行と取消しの間の時間の長さは、それほど正確ではありません。この間隔は、指定されたタイムアウト間隔より短くはなりませんが、数秒間長くなることがあります。アプリケーションに、高い優先順位で実行されているアクティブ・スレッドがある場合、この間隔は随意に長くなることがあります。モニター・スレッドは高い優先順位で実行されますが、他にも高い優先順位のスレッドがあり、それが実行し続けられる場合があります。モニター・スレッドが開始されるのは、ゼロ以外のタイムアウトで実行される文がある場合のみです。すべてのOracle JDBC文の実行を監視するモニター・スレッドは、1つのみあります。



ノート:

Statement.cancel メソッドと Statement.setQueryTimeout メソッドは、サーバー側内部ドライバでサポートされません。サーバー側内部ドライバは単一スレッドのサーバー・プロセスで動作します。Oracle JVM はこの単一スレッド・プロセス内に Java スレッドを実装します。サーバー側内部ドライバが SQL 文を実行している場合、Java スレッドは Statement.cancel メソッドをコールできません。これは、Oracle JDBC 監視スレッドにも適用されます。

E.1.5 ファイアウォールとJDBCの使用

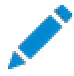
アイドル接続に対するファイアウォール・タイムアウトによって、接続が切断される場合があります。このために、JDBCアプリケーションが接続の待機中に停止する可能性があります。ファイアウォール・タイムアウトによって接続が切断されないようにするには、次の処理を1つ以上実行します。

- 接続キャッシュまたは接続プーリングを使用している場合は、接続キャッシュのInactivityタイムアウト値を、常にファイアウォールのアイドル・タイムアウト値より短い値に設定します。
- 接続プロパティとしてoracle.jdbc.ReadTimeoutを渡してソケット上で読込みタイムアウトを有効にします。タイムアウト値はミリ秒で指定します。
- JDBC OCIドライバとJDBC Thinドライバの両方について、ネット記述子を使用してデータベースに接続し、接続記述子のDESCRIPTION句にENABLE=BROKENパラメータを指定します。また、TCP_KEEPALIVE_INTERVALの下限値も設定します。
- サーバー側のsqlnet.oraファイルにSQLNET.EXPIRE_TIME=1を設定して、Oracle Net DCDを使用可能にします。

E.1.6 多数回にわたるサーバーからの突然の切断

ネットワークの信頼性が低い場合、サーバーの接続が突然切断されても、クライアントでは頻繁に発生する切断を検出するのが困難です。デフォルトでは、Linux上で稼働するクライアントは、突然の切断を検出するのに7200秒(2時間)かかります。この値は、tcp_keepalive_timeプロパティの値と同じです。アプリケーションが切断をより迅速に検出するようにするには、tcp_keepalive_time、tcp_keepalive_intervalおよびtcp_keepalive_probesプロパティの値をオペレーティング・システ

ム・レベルでより小さい値に設定する必要があります。



ノート:

tcp_keepalive_interval プロパティに小さい値を設定すると、ネットワーク上にプローブ・パケットが頻繁に送出され、システム速度が低下する可能性があります。そのため、このプロパティの値は、システム要件に基づいて適切に設定してください。

また、接続記述子のDESCRIPTION句にENABLE=BROKENパラメータを指定する必要があります。たとえば:

```
jdbc:oracle:thin:@(DESCRIPTION=(ENABLE=BROKEN)(ADDRESS=(PROTOCOL=tcp)(PORT=5221)(HOST=myhost))(CONNECT
_DATA=(SERVICE_NAME=orcl)))
```

E.1.7 ネットワーク・アダプタで接続を確立できない

JDBCアプリケーションからOracleインスタンスへの接続を確立しようとしているときに、次のエラーが発生する場合があります。

```
java.sql.SQLException: Io exception:
  The Network Adapter could not establish connection

SQLException: SQLState (null) vendor code (17002)
```

このエラーは、次の状況のいずれかまたは全部に当てはまる場合でも発生することがあります。

- 同じクライアントから同じOracleインスタンスへのSQL*Plus接続を確立できる。
- JDBC OCI接続は確立できるが、同じクライアントから同じOracleインスタンスへのJDBC Thin接続は確立できない。
- 同じJDBCアプリケーションが別のクライアントから同じOracleインスタンスに接続できる。
- 初期JDBC接続文字列がホスト名を指定していようとIPアドレスを指定していようと、同じ動作が適用される。

このエラーの原因として、次の理由の1つ以上が考えられます。

- 接続を確立しようとしているホスト名が正しくない。
- 接続の確立に使用しているポート番号が間違っている。
- NICカードでIPv4とIPv6の両方がサポートされている。
- OracleインスタンスがMTS用に構成されているのに、JDBC接続で専用サーバーではなく共有サーバーを使用している。

SQL*Plusを使用すると、前述の原因を短時間で診断できます。ただし、NICカードに問題がある場合を除きます。次の各項では、このエラーの解決方法とサンプル・アプリケーションを示します。

- MTSサーバーで構成したOracleインスタンスが共有サーバーを使用
- IPv4とIPv6の両方をサポートするNICカードを保有するJDBC Thinドライバ
- サンプル・アプリケーション

E.1.7.1 MTSサーバーで構成したOracleインスタンスが共有サーバーを使用

このエラーを解決するには、Oracleインスタンスがマルチスレッド・サーバー(MTS)用に構成されているかどうかを確認する必要が

あります。OracleインスタンスがMTS用に構成されていない場合は、構成する必要があります。

OracleインスタンスがMTS用に構成されている場合は、JDBC接続が共有サーバーではなく専用サーバーを使用するように強制する必要があります。これを行うには、専用接続のみを使用するようにサーバーを再構成します。専用接続のみを使用するようにサーバーを構成することが実現可能ではない場合は、次のステップを実行してクライアント側から設定します。

JDBC OCIクライアントの場合

1. クライアントのtnsnames.oraファイルに格納されているTNS接続文字列に(SERVER=DEDICATED)プロパティを追加します。
2. クライアントのsqlnet.oraファイルでUSER_DEDICATED_SERVER=ONを設定します。

JDBC Thinの場合:

短縮JDBC Thin構文ではなく、フルネーム/値ペアの接続文字列(tnsnames.oraファイルに指定されているのと同じ)を指定する必要があります。たとえば、"jdbc:oracle:thin:@host:port:sid"接続文字列ではなく、次の形式の接続文字列を使用する必要があります。

```
"jdbc:oracle:thin:@(DESCRIPTION="                      +
                   "(ADDRESS_LIST="                    +
                       "(ADDRESS=(PROTOCOL=TCP)"       +
                               "(HOST=host)"           +
                               "(PORT=port)"           +
                       ")"                             +
                   ")"                                 +
                   "(CONNECT_DATA="                    +
                       "(SERVICE_NAME=sid)"            +
                       "(SERVER=DEDICATED)"            +
                   ")"                                 +
                 ")"
```

E.1.7.2 IPv4とIPv6の両方をサポートするNICカードを保有するJDBC Thinドライバ

サーバーのネットワーク・インタフェース・コントローラ(NIC)カードがIPv4とIPv6の両方をサポートするように構成されている場合、サービスによってはIPv6で起動することがあります。IPv6で実行中のサービスにIPv4を使用して(またはその逆)接続しようとするクライアント・アプリケーションはいずれも接続拒否エラーを受け取ります。JDBC thinクライアント・アプリケーションがデータベース・サーバーに接続しようとすると、そのアプリケーションは応答を停止するか、次のエラーで失敗する可能性があります。

```
java.sql.SQLException: Io exception: The Network Adapter could not establish the connection Error
Code: 17002
```

次の解決策のいずれかを使用してこのエラーを解消してください。

- Java仮想マシン(JVM)に、IPプロトコル・バージョン4を使用するように指定します。-Djava.net.preferIPv4Stackパラメータをtrueに設定して、JDBCアプリケーションが実行されているJVMを起動します。たとえば、jdbcTestという名前のJDBCアプリケーションを実行しているとします。その場合、アプリケーションを次の方法で実行します。

  ```
  java -Djava.net.preferIPv4Stack=true jdbcTest
  ```

- OCI JDBCドライバを使用します。

E.1.7.3 サンプル・アプリケーション

例E-1に、データベースに接続し、接続のテストに使用できる基本的なJDBCプログラムを示します。Oracle JDBCドライバを使用する、あらゆる形式の接続を試すことができます。

例E-1 5つの異なる方法でデータベースに接続する基本的なJDBCプログラム

```
import java.sql.*;
public class Jdbctest
{
   public static void main (String args[])
   {
         try
         {
                   /* Uncomment the next line for more connection information */
                   // DriverManager.setLogStream(System.out);
                   /* Set the host, port, and sid below to match the entries in the listener.ora */
                   String host = "myhost.oracle.com";
                   String port = "5221";
                   String sid  = "orcl";
                   // or pass on command line arguments for all three items
                   if ( args.length >= 3 )
                   {
                              host = args[0];
                              port = args[1];
                              sid  = args[2];
                   }

                   String s1 = "jdbc:oracle:thin:@" + host + ":" + port + ":" + sid ;
                   if ( args.length == 1 )
                   {
                              s1 =  "jdbc:oracle:oci8:@" + args[0];
                   }
                   if ( args.length == 4 )
                   {
                               s1 = "jdbc:oracle:" + args[3] + ":@" +
                                    "(description=(address=(host=" + host+ ")(protocol=tcp)(port=" +
port+ "))(connect_data=(sid="+ sid + ")))";
                   }
                   System.out.println( "Connecting with: " );
                   System.out.println( s1 );
                   DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
                   Connection conn = DriverManager.getConnection( s1,"hr","hr");
                   DatabaseMetaData dmd = conn.getMetaData();
                   System.out.println("DriverVersion:["+dmd.getDriverVersion()+"]");
                   System.out.println("DriverMajorVersion: ["+dmd.getDriverMajorVersion()+"]");
                   System.out.println("DriverMinorVersion: ["+dmd.getDriverMinorVersion()+"]");
                   System.out.println("DriverName:["+dmd.getDriverName()+"]");
                   if ( conn!=null )
                      conn.close();
                   System.out.println("Done.");
           }
           catch ( SQLException e )
           {
                   System.out.println ("¥n*** Java Stack Trace ***¥n");
                   e.printStackTrace();
                   System.out.println ("¥n*** SQLException caught ***¥n");
                   while ( e != null )
```

```
                    {
                                    System.out.println ("SQLState: " + e.getSQLState ());
                                    System.out.println ("Message:  " + e.getMessage ());
                                    System.out.println ("Error Code:   " + e.getErrorCode ());
                                    e = e.getNextException ();
                                    System.out.println ("");
                     }
            }
  }
}
```

E.2 基本的なデバッグ処理

JDBCプログラムのデバッグ方法について説明します。

- ネットワーク・イベントをトラップするためのOracle Netトレース
- サード・パーティのデバッグ・ツール

関連トピック

- SQL例外の処理について

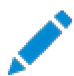
E.2.1 ネットワーク・イベントをトラップするためのOracle Netトレース

クライアントおよびサーバーのOracle-Netトレースが、Oracle Net経由で送信されるパケットをトラップできるようにできます。クライアント側トレースはJDBC OCIドライバについてのみ使用できます。JDBC Thinドライバについてはサポートされません。

トレース機能を使用すると、ネットワーク・イベントが実行されるたびにそのイベントについて記述される、一連の詳細な文が生成されます。操作をトレースすることにより、イベントの内部操作に関する詳細な情報を取り出すことができます。この情報は読込み可能ファイルに出力され、エラーの原因となったイベントを特定できます。トレース情報の収集は、SQLNET.ORAファイルにあるいくつかのOracle Netパラメータによって制御されます。SQLNET.ORAのパラメータを設定した後に、トレースを実行するために新しい接続を作成する必要があります。

トレース・レベルを高くすると、より詳細な情報がトレース・ファイルに取得されます。トレース・ファイルの理解が難しくなるため、トレースを有効にする場合はトレース・レベル4から開始します。トレース・ファイルの最初の部分は接続ハンドシェイク情報です。JDBCプログラムに関連するSQL文とエラー・メッセージの情報は、その先を参照してください。

ノート:



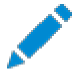
トレース機能ではディスク領域が大量に使用されるため、システムのパフォーマンスが大幅に低下する可能性があります。トレースは、必要なときにのみ使用可能にしてください。

関連トピック

- 『Oracle Call Interfaceプログラマーズ・ガイド』

E.2.1.1 クライアント側でのトレース

クライアント・システムのSQLNET.ORAファイルに、次のパラメータを設定します。



ノート:

Oracle Database 12c リリース 1 (12.1)以降、sqlnet.ora ファイルにある OCI 固有の構成パラメータのかわりに、新しい XML 構成ファイル oraaccess.xml にある構成パラメータを使用することをお薦めします。ただし、sqlnet.ora ファイル内の構成パラメータもまだサポートされています。

E.2.1.1.1 TRACE_LEVEL_CLIENT

目的:

トレースを一定の指定レベルでonまたはoffにします。

デフォルト値:

0またはOFF

有効な値

- 0またはOFF- トレースの出力なし
- 4またはUSER- ユーザー・トレース情報
- 10またはADMIN- 管理トレース情報
- 16またはSUPPORT- カスタマ・サポート・トレース情報

例:

```
TRACE_LEVEL_CLIENT=10
```

E.2.1.1.2 TRACE_DIRECTORY_CLIENT

目的:

トレース・ファイルの書込み先ディレクトリを指定します。

デフォルト値:

ORACLE_HOME/network/trace

例:

UNIX: TRACE_DIRECTORY_CLIENT=/oracle/traces

Windows: TRACE_DIRECTORY_CLIENT=C:¥ORACLE¥TRACES

E.2.1.1.3 TRACE_FILE_CLIENT

目的:

クライアント・トレース・ファイルの名前を指定します。

デフォルト値:

SQLNET.TRC

例:

TRACE_FILE_CLIENT=cli_Connection1.trc

ノート:

TRACE_FILE_CLIENT ファイルのために選択する名前は、TRACE_FILE_SERVER ファイルのために選択する名前と異なっている必要があります。

E.2.1.1.4 TRACE_UNIQUE_CLIENT

目的:

クライアント側の各トレースに一意の名前を付け、各トレース・ファイルが次に発生したクライアント・トレースによって上書きされないようにします。ファイル名の最後にPIDが付加されます。

デフォルト値:

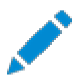
OFF

例:

TRACE_UNIQUE_CLIENT = ON

E.2.1.2 サーバー側でのトレース

サーバー・システムのSQLNET.ORAファイルに次のパラメータを設定します。接続ごとに、一意のファイル名を持つ個別のファイルが生成されます。



ノート:

Oracle Database 12c リリース 1 (12.1)以降、sqlnet.ora ファイルにある OCI 固有の構成パラメータのかわりに、新しい XML 構成ファイル oraaccess.xml にある構成パラメータを使用することをお薦めします。ただし、sqlnet.ora ファイル内の構成パラメータもまだサポートされています。

E.2.1.2.1 TRACE_LEVEL_SERVER

目的:

トレースを一定の指定レベルでonまたはoffにします。

デフォルト値:

0またはOFF

有効な値

- 0またはOFF- トレースの出力なし
- 4またはUSER- ユーザー・トレース情報

- 10またはADMIN- 管理トレース情報
- 16またはSUPPORT- カスタマ・サポート・トレース情報

例:

```
TRACE_LEVEL_SERVER=10
```

E.2.1.2.2 TRACE_DIRECTORY_SERVER

目的:

トレース・ファイルの書込み先ディレクトリを指定します。

デフォルト値:

ORACLE_HOME/network/trace

例:

TRACE_DIRECTORY_SERVER=/oracle/traces

E.2.1.2.3 TRACE_FILE_SERVER

目的:

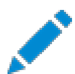
サーバー・トレース・ファイルの名前を指定します。

デフォルト値:

SERVER.TRC

例:

TRACE_FILE_SERVER= svr_Connection1.trc



ノート:

TRACE_FILE_SERVER ファイルのために選択する名前は、TRACE_FILE_CLIENT ファイルのために選択する名前と異なっている必要があります。

E.2.2 サード・パーティのデバッグ・ツール

Intersolv社のJDBCSpyおよびJDBCTestなどのツールを使用して、JDBC APIレベルで問題に対処できます。これらのツールはODBC SpyおよびODBC Testツールと類似しています。

索引

A B C D E F G I J K L M N O P R S T U W X

A

- PL/SQL連想配列へのアクセス 4.2.6
- ANYDATA 4.5.4
- ANYTYPE 4.5.4
- アプリケーション・コンティニュイティ
  - Oracle Databaseの構成 28.2
  - Oracle JDBCの構成 28.1
  - 再接続の遅延 28.7
  - リプレイの無効化 28.10
  - リクエスト境界の識別 28.4
  - 論理トランザクションID 27.1
  - 接続のコールバックの登録 28.6
  - 可変値の保持 28.8
  - トランザクション・ガード 28
- ARRAY
  - オブジェクト, 作成 16.4.1
- 配列
  - 定義済 16.1.1
  - 取得 16.4.2.6
  - 名前付き 16.1.1
  - コール可能文への引渡し 16.4.3
  - 結果セットからの取出し 16.4.2.1
  - 配列の一部の取出し 16.4.2.5
  - 型マップの使用 16.5
  - 操作 16.1.1
- 最大1回の実行 27.1
- 認証(セキュリティ) 9.4
- 自動コミット 2.3.8
- 自動コミット・モード
  - 無効化 C.2.1
  - 結果セットの動作 C.2.1

B

- バッチ・ジョブ、ユーザーの認証 9.10
- バッチの更新, 「バッチ更新」を参照 21.1

* BFILE
    * クラス 4.3.1
    * 定義済 12.4
* BFILEロケータ，選択 4.3.1
* BLOB
    * クラス 4.3.1
    * ロケータ
        * 選択 4.3.1
* ブランチ修飾子(分散トランザクション) 32.2.5

C

* CachedRowSet 18.2
* キャッシュ，クライアント側
    * スクロール可能な結果セットを実現するOracleの使用 17.1
* コール可能文
    * getOracleObject()メソッドの使用 11.4.2
* 取消し
    * SQL文 E.1.4
* 戻り値のキャスト 11.4.5
* catalog引数(DatabaseMetaData) A.5.4
* 文字セット 4.4.3
* CHAR列
    * setFixedCHAR()の使用によるWHEREでの合致 11.4.7.2
* CLOB
    * クラス 4.3.1
    * ロケータ，選択 4.3.1
* close()メソッド E.1.2
* closeメソッド 20.3.2
* コレクション
    * 定義済 16.1.1
* コレクション(ネストした表と配列) 16.4.1
* 列型
    * 定義 21.2.3
    * 再定義 21.2
* 分散トランザクション・ブランチのコミット 32.2.4
* データベースの変更のコミット 2.3.8
* 接続
    * クローズ 2.3.9
    * オープン 2.3.2
* 接続プロパティ 8.1.5
    * put()メソッド 8.1.9
* 接続

    - 読取り専用 C.3
- SQL型の定数 4.5.5.7
- CursorName
    - 制限 A.5.1
- カーソル E.1.2
- カスタム・コレクション・クラス
    - 定義済 16.1.2
- カスタムJavaクラス 4.2.3
    - 定義済 13.1
- カスタム・オブジェクト・クラス
    - 作成 13.3.1
    - 定義済 13.1
- カスタム参照クラス
    - 定義済 15.1

D

- データベース
    - 接続
        - サーバー側内部ドライバ 7.2
    - 接続テスト 2.2.5
- DatabaseMetaDataコール A.5.4
- データベース常駐接続プーリング 23.1
    - API 23.6
    - DRCP 23
    - 有効化
        - クライアント側 23.2.2
        - サーバー側 23.2.1
    - 複数の接続間での共有 23.3
    - タグ付け 23.4
- データベース指定子 8.2
- データベースURL
    - ユーザーIDとパスワードを含む 2.3.2
- データベースURL, 指定 2.3.2
- データベースURL
    - データベース指定子 8.2
- データ変換 11.2
    - LONG 12.2.3
    - LONG RAW 12.2.2
- データソース 8.1
    - JNDI 8.1.4
- データ・ソース
    - 作成と接続(JNDIを使用) 8.1.4

    - 作成と接続(JNDIなし) 8.1.3
    - Oracleによる実装 8.1.2
    - プロパティ 8.1.2
    - 標準インタフェース 8.1.2
- データ・ストリーム
    - 回避 12.2.5
- データ型マッピング 11.1
- データ型
    - Java 11.1
    - Javaネイティブ 11.1
    - JDBC 11.1
    - Oracle SQL 11.1
- DATEクラス 4.3.1
- DBOPタグ 3.4.1
- JDBCプログラムのデバッグ E.2
- defaultConnection()メソッド 7.2
- detachServerConnection 23.6
- 分散トランザクションIDコンポーネント 32.2.5
- 分散トランザクション
    - ブランチ修飾子 32.2.5
    - 同じリソース・マネージャのチェック 32.2.4
    - トランザクション・ブランチのコミット 32.2.4
    - コンポーネントおよびシナリオ 32.1.2
    - 概要 32.1.3
    - 分散トランザクションIDコンポーネント 32.2.5
    - トランザクション・ブランチの終了 32.2.4
    - 実装例 32.4
    - forget 32.2.4
    - グローバル・トランザクション識別子 32.2.5
    - IDフォーマット識別子 32.2.5
    - リカバリ中のトランザクション・ブランチのリストの取得 32.2.4
    - OracleのXA接続実装 32.2.2
    - OracleのXAデータソース実装 32.2.1
    - OracleのXA ID実装 32.2.5
    - OracleのXA最適化 32.3.4
    - OracleのXAリソース実装 32.2.3
    - 概要 32.1
    - トランザクション・ブランチの準備 32.2.4
    - トランザクション・ブランチのロールバック 32.2.4
    - トランザクション・ブランチの起動 32.2.4
    - トランザクション・ブランチIDコンポーネント 32.2.5
    - XA接続インタフェース 32.2.2
    - XAデータソース・インタフェース 32.2.1

    - XAエラー処理 32.3.3
    - XA例外クラス 32.3.1
    - XA IDインタフェース 32.2.5
    - XAリソース機能 32.2.4
    - XAリソース・インタフェース 32.2.3
- DML RETURNING 4.2.5, 4.6
    - 例 4.6.3
    - 制限 4.6.4
    - Oracle固有のAPI 4.6.1
    - 文の実行 4.6.2
- Double.NaN
    - 使用上の制限事項 4.3.1
- DRCP 23.1

E

- 分散トランザクション・ブランチの終了 32.2.4
- Enterprise Java Beans(EJB) 18.2
- エラー
    - 一般JDBCメッセージ, リスト D.2
    - 一般JDBCメッセージ構造 D.1
    - 例外の処理 2.13
    - TTCメッセージ, リスト D.4
- 明示的文キャッシュ
    - 定義 20.1.3
- JDBCに対する拡張機能, Oracle 4, 11, 13, 15, 16, 21
- 外部変更(結果セット)
    - 定義済 17.6
    - 可視性と検出 17.6.1
- 外部ファイル
    - 定義済 12.4

F

- 結果セットのフェッチ方向 17.4.2
- フェッチ・サイズ, 結果セット 17.4
- FilteredRowSet 18.5
- ファイナライザ・メソッド E.1.2
- ファイアウォール, JDBCとともに使用 E.1.5
- Float.NaN
    - 使用上の制限事項 4.3.1
- 浮動小数点との互換性 A.5.3

- フォーマット識別子，トランザクションID 32.2.5
- ファンクション・コール構文，JDBCエスケープ構文 A.4.6

G

- getBinaryStream()メソッド 12.2.4
- getBytes()メソッド 12.2.4
- getColumns 2.5
- getConnection()メソッド 7.2
- getCursorName()メソッド
    - 制限 A.5.1
- getLogicalTransactionIdメソッド 27.4.1
- getMoreResultSet(int) 2.7
- getObject()メソッド
    - ORADataオブジェクト 13.3.6
    - 戻り型 11.4.3
- getOracleObject()メソッド
    - 戻り型 11.4.2, 11.4.3
    - コール可能文での使用 11.4.2
    - 結果セットでの使用 11.4.2
- getStatementCacheSize()メソッド
    - コード例 20.2.1
- getXXX()メソッド
    - 戻り値のキャスト 11.4.5
    - 特定のデータ型 11.4.4
- グローバリゼーション 19
    - 使用 19
- グローバル・トランザクション識別子(分散トランザクション) 32.2.5
- グローバル・トランザクション 32.1

I

- IEEE 754浮動小数点との互換性 A.5.3
- 暗黙的文キャッシュ
    - 定義 20.1.2
    - 最低使用頻度(LRU)アルゴリズム 20.1.2
- インストール
    - クライアントでの検証 2.2
- Instant Client機能 6.4.1
- 内部変更(結果セット)
    - 定義済 17.6
- isColumnInvisible 2.5

- isSameRM() (分散トランザクション) 32.2.4

J

- Java
  - コンパイルと実行 2.2.3
  - データ型 11.1
  - ネイティブなデータ型 11.1
  - ストアド・プロシージャ 2.12.2
  - ストリーム・データ 12
- java.sql.Connectionインタフェース
  - closeメソッド 20.3.2
- java.sql.Statementインタフェース
  - closeメソッド 20.3.2
- java.util.Properties 22.3.4
- Java Naming and Directory Interface(JNDI) 8.1.1
- Javaソケット 1.1
- Java仮想マシン(JVM) 7.1
- JDBC
  - IDE 1.4.3
  - 基本プログラム 2.3
  - データ型 11.1
  - 定義済 1
  - パッケージのインポート 2.3.1
  - Oracle拡張機能の制限事項 A.5
  - サンプル・ファイル 2.2.3
  - テスト 2.2.5
  - バージョンのサポート 3
- JDBC 2.0サポート
  - データ型のサポート 3.1.1
  - 拡張機能のサポート 3.1.3
  - 概要 3.1
  - JDK 1.2.xとJDK 1.1.x 3.1, 3.2
  - 標準機能のサポート 3.1.2
- JdbcCheckupプログラム 2.2.5
- JDBCドライバ
  - ニーズに合ったドライバの選択 1.2
  - 一般的な問題 E.1
  - 概要 1.1
  - JDBCエスケープ構文 A.4
- JDBCエスケープ構文 A.4
  - ファンクション・コール構文 A.4.6
  - LIKEエスケープ文字 A.4.3

    - 外部結合 A.4.5
    - スカラー関数 A.4.2
    - 時刻および日付リテラル A.4.1
    - SQLへの変換例 A.4.7
- JDBCRowSet 18.3
- JDBCSpy E.2.2
- JDBCTest E.2.2
- JDeveloper 1.4.3
- JNDI
    - データソース 8.1.4
    - データソースの参照 8.1.4
    - Oracleによるサポートの概要 8.1.1
    - データソースの登録 8.1.4
- JoinRowSet 18.6
- JVM 7.1

K

- KPRBドライバ
    - 概要 1.1
    - SQLエンジンとの関係 7.1
    - セッション・コンテキスト 7.3
    - テスト 7.4
    - トランザクション・コンテキスト 7.3
    - URL 7.2

L

- 最低使用頻度(LRU)アルゴリズム 20.1.2, 22.3.5
- LIKEエスケープ文字, JDBCエスケープ構文 A.4.3
- setBytes()とsetString()の制限, 回避するためのストリームの使用 12.7.2
- LOB
    - 定義済 12.4
- 論理トランザクションID
    - LTXID 27.1
- LONG
    - データ変換 12.2.3
- LONG RAW
    - データ変換 12.2.2
- LRUアルゴリズム 20.1.2
- LTXID 27.1

M

- メモリー・リーク E.1.2
- データベースの動作の監視
    - DBOP 3.4.1
    - setClientInfo 3.4.1

N

- 名前付き配列 16.1.1
    - 定義済 16.4.1
- nativeXA 8.1.2
- ネットワーク・イベント, トラップ E.2.1
- NLS「グローバリゼーション」を参照 19
- NULL
    - テスト 11.2.3
- NUMBERクラス 4.3.1

O

- オブジェクト参照
    - オブジェクト値のアクセス 15.2.1, 15.3
    - 説明 15.1
    - プリペアド文への引渡し 15.2.3
    - 取得 15.2.1
    - コール可能文からの取出し 15.2.2
    - オブジェクト値の更新 15.2.1, 15.3
- OCIドライバ
    - 説明 1.1
- ODBCSpy E.2.2
- ODBCTest E.2.2
- 最適化, パフォーマンス C.2
- oracle.jdbc, OracleのJDBC対応拡張機能 2.3.1
- oracle.jdbc.LogicalTransactionIdEventListenerインタフェース 27.4.2
- oracle.jdbc.OracleCallableStatementインタフェース 4.5.5.4
- oracle.jdbc.OracleConnectionインタフェース 4.5.5.1
- oracle.jdbc.OraclePreparedStatementインタフェース 4.5.5.3
- oracle.jdbc.OracleResultSetインタフェース 4.5.5.5
- oracle.jdbc.OracleResultSetMetaDataインタフェース 4.5.5.6
- oracle.jdbc.OracleSqlクラス A.4.7
- oracle.jdbc.OracleStatementインタフェース 4.5.5.2
- oracle.jdbc.OracleTypesクラス 4.5.5.7

- oracle.jdbc.xaパッケージおよびサブパッケージ 32.1.5
- oracle.sql.ARRAYクラス
    - Javaプリミティブ型用のメソッド 16.3.1
- oracle.sql.BFILEクラス 4.3.1
- oracle.sql.BLOBクラス 4.3.1
- oracle.sql.CLOBクラス 4.3.1
- oracle.sql.dataタイプ
    - サポート 4.3.1
- oracle.sql.DATEクラス 4.3.1
- oracle.sql.NUMBERクラス 4.3.1
- oracle.sql.RAWクラス 4.3.1
- oracle.sql.STRUCTクラス 4.3.1
- Oracle Advanced Security
    - JDBCによるサポート 9.3
- OracleCallableStatementインタフェース 4.5.5.4
- OracleCallableStatementオブジェクト 20.1.2
- OracleConnectionクラス 4.5.5.1
- OracleDataインタフェース
    - 利点 13.3.2
- OracleDataSourceクラス 8.1.2
- Oracleデータ型
    - 使用 11
- Oracleの拡張機能 4.2
    - データ型のサポート 4.2.2
    - 制限 A.5
        - DatabaseMetaDataコールへのcatalog引数 A.5.4
        - CursorName A.5.1
        - IEEE 754浮動小数点との互換性 A.5.3
        - JDBC外部結合エスケープ A.5.2
        - 読取り専用接続 C.3
        - SQLWarningクラス A.5.5
    - オブジェクトのサポート 4.2.3
    - 結果セット 11.3
    - 文 11.3
    - JDBC 4, 11, 13, 15, 16, 21
- Oracleオブジェクト
    - JDBC 13.1
    - Javaクラスによるサポート 13.2.1
    - カスタム・オブジェクト・クラスへのマッピング 13.3.1
    - SQLDataインタフェースを使用したデータの読込み 13.3.5
    - 操作 13.1
    - SQLDataインタフェースを使用したデータの書込み 13.3.5
- OraclePreparedStatementインタフェース 4.5.5.3

- OraclePreparedStatementオブジェクト 20.1.2
- OracleResultSetインタフェース 4.5.5.5
- OracleResultSetMetaDataインタフェース 4.5.5.6
- Oracle SQLデータ型 11.1
- OracleStatementインタフェース 4.5.5.2
- OracleTypesクラス 4.5.5.7
- OracleXAConnectionクラス 32.2.2
- OracleXADataSourceクラス 32.2.1
- OracleXAResourceクラス 32.2.3
- OracleXidクラス 32.2.5
- ORADataインタフェース
    - その他の使用方法 13.3.8
    - データの読込み 13.3.7
    - データの書込み 13.3.7
- orai18n.jarファイル 19.1
- 外部結合, JDBCエスケープ構文 A.4.5

P

- パスワード, 指定 2.3.2
- PDA 18.2
- パフォーマンス拡張, 標準準拠とOracle 3.1.4
- パフォーマンス拡張機能
    - 列型の定義 21.2.3
- パフォーマンスの最適化 C.2
- 携帯情報端末(PDA) 18.2
- PL/SQL
    - ストアド・プロシージャ 2.12.1
- PL/SQL連想配列 4.7
- 行のプリフェッチ 21.2
    - 推奨デフォルト値 21.2.2
- 分散トランザクション・ブランチの準備 32.2.4
- put()メソッド
    - プロパティ・オブジェクト 8.1.9

R

- RAWクラス 4.3.1
- リカバリ(分散トランザクション) 32.2.4
- REF CURSOR 4.5.2
- 結果セットへの行の再フェッチ 17.5
- registerConnectionInitializationCallback 28.6.3.2

- Remote Method Invocation(RMI) 18.2
- リソース・マネージャ 32.1.2
- 結果セット
  - 自動コミット・モード C.2.1
  - メタデータ 4.5.5.6
  - Oracle拡張機能 11.3
  - getOracleObject()メソッドの使用 11.4.2
- 結果セット, 処理 2.3.5
- 結果セット拡張
  - ダウングレード・ルール 17.2
  - フェッチ・サイズ 17.4
  - 制限 17.2
  - Oracleスクロール可能性の要件 17.1
  - Oracle更新可能性の要件 17.1
  - 行の再フェッチ 17.5
  - 変更の可視性のサマリー 17.6.2
  - 外部変更の可視性と検出 17.6.1
- 結果セット・フェッチ・サイズ 17.4
- 結果セットの保持機能 3.2.4
- 結果セット・オブジェクト
  - クローズ 2.3.6
- 自動生成キーの取出し 3.2.2
- 戻り型
  - getXXX()メソッド 11.4.4.1
  - getObject()メソッド 11.4.3
  - getOracleObject()メソッド 11.4.3
- 戻り値
  - キャスト 11.4.5
- RMI 18.2
- 分散トランザクション・ブランチのロールバック 32.2.4
- データベースの変更のロールバック 2.3.8
- ROWID, 結果セット更新での使用 17.1
- ROWIDクラス
  - 定義済 4.5.1
- 行のプリフェッチ
  - データ・ストリーム 12.7.3
- RowSet
  - イベントおよびイベント・リスナー 18.1.2
  - 概要 18.1
  - プロパティ 18.1.1
  - 横断 18.1.4

S

- セーブポイント
    - トランザクション 3.2.1
- スカラー関数，JDBCエスケープ構文 A.4.2
- SCAN
    - 下位互換性 31.4
    - データベースの構成 31.2
    - 接続ロード・バランシング 31.3
    - 最大可用性アーキテクチャ環境 31.5
    - Oracle Connection Manager 31.6
    - 概要 31.1
    - バージョン 31.4
- スキーマの命名 4.2.4
- スクリプト、ユーザーの認証 9.10
- スクロール可能な結果セット
    - フェッチ方向 17.4.2
    - scroll-sensitivityの実装 17.6.3
    - 行の再フェッチ 17.5
    - 外部変更の可視性と検出 17.6.1
- scroll-sensitive結果セット
    - 制限 17.2
- セキュリティ
    - 認証 9.4
    - Oracle Advanced Securityのサポート 9.3
- サーバー側内部ドライバ
    - データベースへの接続 7.2
- サーバー側Thinドライバ，概要 1.1
- セッション・コンテキスト
    - KPRBドライバ 7.3
- setBytes()の制限，回避するためのストリームの使用 12.7.2
- setCursorName()メソッド A.5.1
- setDisableStmtCaching()メソッド 20.2.3
- setEscapeProcessing()メソッド A.4
- setFixedCHAR()メソッド 11.4.7.2
- setNull() 11.2.3
- setObejct()メソッド 11.4.6
- setObject()メソッド
    - STRUCTオブジェクト 13.2.4
- setOracleObject()メソッド 11.4.6
- setString()の制限，回避するためのストリームの使用 12.7.2
- setXXX()メソッド，特定のデータ型 11.4.7
- Solaris

    - 共有ライブラリ 32.5.1
- 指定子
    - データベース 8.2
- SQL
    - Javaデータ型へのデータ変換 11.2
    - 型，定数 4.5.5.7
- SQLDataインタフェース
    - 利点 13.3.2
    - Oracleオブジェクトからのデータの読込み 13.3.5
    - Oracleオブジェクトからのデータの書込み 13.3.5
- SQLエンジン
    - KPRBドライバとの関係 7.1
- SQL構文（Oracle) A.4
- SQLWarningクラス，制限事項 A.5.5
- 分散トランザクション・ブランチの起動 32.2.4
- statement.cancel() E.1.4
- 文キャッシュ
    - 明示的
        - 定義 20.1.3
    - 暗黙的
        - 定義 20.1.2
        - 最低使用頻度(LRU)アルゴリズム 20.1.2
- Statementオブジェクト
    - クローズ 2.3.6
- 文
    - Oracle拡張機能 11.3
- 停止
    - 文の実行 E.1.4
- ストアド・プロシージャ
    - Java 2.12.2
    - PL/SQL 2.12.1
- ストリーム・データ 12
    - CHAR列 12.3
    - クローズ 12.6
    - 例 12.2.4
    - 外部ファイル 12.4
    - LOB 12.4
    - LONG列 12.2.1
    - LONG RAW列 12.2.1
    - 複数列 12.5
    - 注意 12.7.1
    - RAW列 12.3
    - 行のプリフェッチ 12.7.3

    - setBytes()とsetString()の制限を回避するための使用 12.7.2
    - VARCHAR列 12.3
- ストリーム・データ列
    - バイパス 12.5
- STRUCTクラス 4.3.1
- STRUCTオブジェクト
    - 取得 13.2.2
    - oracle.sql型としての属性の取出し 13.2.2
- SYS.ANYDATA 4.5.4
- SYS.ANYTYPE 4.5.4

T

- TAF, 定義 30.1
- TCP/IPプロトコル 8.2.4
- テスト
    - NULL値 11.2.3
- Thinドライバ
    - 概要 1.1
    - サーバー側, 概要 1.1
- 時刻および日付リテラル, JDBCエスケープ構文 A.4.1
- トレース機能 E.2.1
- トレース・パラメータ
    - クライアント側 E.2.1.1
    - サーバー側 E.2.1.2
- トランザクション・ブランチ 32.1.1
- トランザクション・ブランチIDコンポーネント 32.2.5
- トランザクション・コンテキスト
    - KPRBドライバ 7.3
- トランザクション・ガード 27.1, 28
    - 最大1回の実行 27.1
    - 論理トランザクションID 27.1
- トランザクションID(分散トランザクション) 32.1.3
- トランザクション・マネージャ 32.1.2
- トランザクション
    - ローカルとグローバルの切替え 32.1.4
- トランザクション・セーブポイント
- 透過的アプリケーション・フェイルオーバー(TAF), 定義 30.1
- TTCエラー・メッセージ, リスト D.4
- 型マップ 11.4.1
    - エントリの追加 13.3.4.2
    - STRUCT 13.3.4.4
    - 新しいマップの作成 13.3.4.3

    - 配列の使用 16.4.2.4
    - 配列の使用 16.5
- 型マップ(SQLからJavaへ) 13.3.1
- 型マップ
    - データベース接続との関係 7.2

U

- Unicodeデータ 4.4.2
- unregisterConnectionInitializationCallbackメソッド 28.6.3.3
- 更新可能な結果セット
    - 制限 17.2
    - 行の再フェッチ 17.5
    - 更新の競合 17.3
- バッチ更新(標準モデル)
    - バッチに対する追加 21.1.2.2
    - バッチのクリア 21.1.2.6
    - 変更のコミット 21.1.2.5
    - エラー処理 21.1.2.8
    - 例 21.1.2.7
    - バッチの実行 21.1.2.3
    - バッチと非バッチの混在 21.1.2.9
    - 概要 21.1.2
    - 更新件数 21.1.2.7
    - エラー時の更新件数 21.1.2.8
- 結果セットでの更新の競合 17.3
- 更新件数
    - 標準バッチ更新 21.1.2.7
    - エラー時(標準バッチ処理) 21.1.2.8
- URL
    - KPRBドライバ 7.2
- ユーザーID，指定 2.3.2

W

- WebRowSet 18.4
- ウィンドウ，scroll-sensitiveな結果セット 17.6.3

X

- XA

- 接続実装 32.2.2
- 接続(定義) 32.1.3
- データソース実装 32.2.1
- データソース(定義) 32.1.3
- 定義 32.1.1
- エラー処理 32.3.3
- 実装例 32.4
- 例外クラス 32.3.1
- Oracle最適化 32.3.4
- OracleトランザクションID実装 32.2.5
- リソース実装 32.2.3
- リソース(定義) 32.1.3
- トランザクションIDインタフェース 32.2.5