

Oracle9i

SQLJ 開発者ガイドおよびリファレンス

リリース 2 (9.2)

2002 年 7 月

部品番号 : J06300-01

ORACLE®

Oracle9i SQLJ 開発者ガイドおよびリファレンス, リリース 2 (9.2)

部品番号 : J06300-01

原本名 : Oracle9i SQLJ Developer's Guide and Reference, Release 2 (9.2)

原本部品番号 : A96655-01

原本著者 : Brian Wright

原本協力者 : Janice Nygard, Ekkehard Rohwedder, Brian Becker, Alan Thiesen, Lei Tang, Julie Basu, Pierre Dufour, Quan Wang, Jerry Schwarz, Risto Lankinen, Cheuk Chau, Vishu Krishnamurthy, Rafiul Ahad, Jack Melnick, Tim Smith, Thomas Pfaeffle, Tom Portfolio, Ellen Barnes, Susan Kraft, Sheryl Maring, Angie Long

Copyright © 1999, 2002 Oracle Corporation. All rights reserved.

Printed in Japan

制限付権利の説明

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

当プログラムのリバース・エンジニアリング等は禁止されております。

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

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

危険な用途への使用について

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

Restricted Rights Notice

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

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

目次

はじめに	xiii
対象読者	xiv
このマニュアルの構成	xiv
関連文書	xvi
表記規則	xix
 1 概要	
SQLJ について	1-2
基本的な概念	1-2
Oracle 固有コード生成と ISO 標準コード生成との違い	1-3
Java および SQLJ と、PL/SQL との違い	1-3
SQLJ コンポーネントの概要	1-4
SQLJ トランスレータと SQLJ ランタイム	1-4
SQLJ プロファイル (ISO 標準コードの場合)	1-5
SQLJ 規格に準拠した Oracle 拡張型の概要	1-7
基本的な変換処理とランタイム処理	1-9
SQLJ 変換処理	1-9
トランスレータ入出力の概要	1-12
SQLJ のランタイム処理	1-15
JDBC と SQLJ のサンプル・コード	1-16
JDBC バージョンのサンプル・コード	1-17
SQLJ バージョンのサンプル・コード	1-20

他の配置例	1-22
アプレットでの SQLJ の実行	1-22
サーバー側 SQLJ について	1-26
Oracle9i Lite での SQLJ の使用	1-27
他の開発環境	1-29
SQLJ グローバリゼーション・サポート	1-29
Oracle9i JDeveloper などの IDE での SQLJ の使用	1-29
Windows に関する注意事項	1-30

2 SQLJ の基本事項

前提と要件	2-2
環境に対する前提	2-2
Oracle SQLJ を使用するための要件	2-3
Oracle SQLJ 環境 : 主な使用例とガイドライン	2-4
環境の問題点と制限事項	2-6
Oracle SQLJ の下位互換性	2-9
インストールおよび設定の確認	2-10
インストール先のディレクトリとファイルの確認	2-10
PATH と CLASSPATH の設定	2-11
sqljutil パッケージのインストールの確認	2-12
設定のテスト	2-13
ランタイム接続の設定	2-13
データベース検証用の表の作成	2-14
JDBC ドライバの検証	2-15
SQLJ トランスレータとランタイムの検証	2-15
SQLJ トランスレータからデータベースへの接続の検証	2-16

3 基本的な言語機能

SQLJ 宣言の概要	3-2
SQLJ 宣言の規則	3-2
イテレータ宣言	3-3
接続コンテキスト宣言	3-4
IMPLEMENTS 句宣言	3-5
WITH 句宣言	3-6

SQLJ 実行文の概要	3-8
SQLJ 実行文の規則	3-8
SQLJ 句	3-9
接続コンテキスト・インスタンスと実行コンテキスト・インスタンスの指定	3-11
実行文の例	3-12
実行文の PL/SQL ブロック	3-13
Java ホスト式、コンテキスト式および結果式	3-14
ホスト式の概要	3-15
基本的なホスト式の構文	3-15
ホスト式の例	3-18
結果式とコンテキスト式の概要	3-19
Java 式の実行時評価	3-20
Java 式の実行時評価の例（ISO コード生成）	3-21
ホスト式の制限事項	3-29
単一行の間合せ結果：SELECT INTO 文	3-30
SELECT INTO 構文	3-30
SELECT INTO 文の例	3-31
SELECT リストにホスト式を使用する例	3-31
SELECT INTO エラー状態	3-32
複数行の間合せ結果：SQLJ イテレータ	3-33
イテレータの概念	3-33
イテレータの使用手順	3-37
名前指定イテレータ、位置指定イテレータ、結果セット・イテレータの比較	3-37
名前指定イテレータの使用手法	3-38
位置指定イテレータの使用	3-43
ホスト変数としてのイテレータおよび結果セットの使用	3-47
イテレータ列としてのイテレータおよび結果セットの使用	3-50
代入文（SET）	3-53
ストアド・プロシージャおよびストアド・ファンクションのコール	3-54
ストアド・プロシージャのコール	3-54
ストアド・ファンクションのコール	3-55
ストアド・ファンクションの戻り値としてのイテレータおよび結果セットの使用	3-57

4 プログラミング上の主な考慮事項

JDBC ドライバの選択	4-2
Oracle JDBC ドライバの概要	4-2
変換用ドライバの選択	4-4
ランタイムに使用するドライバの選択および登録	4-5
接続の際の考慮事項	4-6
DefaultContext を使用した単一接続または複数接続	4-6
接続の終了	4-10
宣言済みの接続コンテキスト・クラスを使用した複数接続	4-12
Oracle クラスについて	4-12
DefaultContext クラスについて	4-14
変換時の接続	4-17
カスタマイズ時の接続	4-17
NULL の処理	4-18
NULL 処理用のラッパー・クラス	4-18
NULL 処理の例	4-19
例外処理の基本	4-20
SQLJ および JDBC の例外処理要件	4-20
例外の処理	4-21
SQLException サブクラスの使用方法	4-23
基本的なトランザクション制御	4-24
トランザクションの概要	4-24
自動コミットと手動コミットとの違い	4-24
接続を定義する際の自動コミットの指定	4-25
既存の接続の自動コミットに対する変更	4-26
手動 COMMIT または ROLLBACK の使用方法	4-26
イテレータおよび結果セットに対するコミットおよびロールバックの影響	4-27
セーブポイントの使用	4-28
要約: 簡単な SQLJ コード	4-28
必要なクラスのインポート	4-29
JDBC ドライバの登録およびデフォルト接続の設定	4-29
例外処理の設定	4-30
ホスト変数のセットアップ、SQLJ 句の実行、結果の処理	4-31
SELECT INTO を使用した単一行問合せの例	4-32

名前指定イテレータの設定	4-33
名前指定イテレータを使用した複数行問合せの例	4-34
Oracle 固有コード生成（プロファイルなし）	4-36
Oracle 固有コード生成のメリットとデメリット	4-36
Oracle 固有コード生成用環境の要件	4-37
Oracle 固有コード生成でのコード上の考慮事項と制限事項	4-38
Oracle 固有コード生成時の SQLJ 使用方法の変更	4-39
Oracle 固有コード生成でのサーバー側の考慮事項	4-41
ネーミングの要件および制限事項	4-42
Java のネームスペース：ローカル変数およびクラスのネーミングに関する制限事項	4-42
SQLJ のネームスペース	4-43
SQL のネームスペース	4-44
ファイル名の要件と制限	4-44
中間層の SQLJ に関する考慮事項	4-45

5 型のサポート

ホスト式での型サポート	5-2
サポートされている型の概要	5-2
JDBC 2.0 でサポートされている型と要件	5-8
PL/SQL の BOOLEAN 型、RECORD 型および TABLE 型の使用	5-9
以前の Oracle JDBC リリースとの下位互換性	5-11
ストリームのサポート	5-13
SQLJ ストリームの一般的な使用方法	5-13
ストリームをサポートするクラスの主な特長	5-14
データ送信時の SQLJ ストリームの使用	5-15
ストリームへのデータの取出し：注意	5-18
データ送信時の SQLJ ストリームの使用	5-19
ストリーム・クラスの方法	5-21
ストリーム・データの取出しおよび処理例	5-23
出力パラメータおよび関数の戻り値としての SQLJ ストリーム・オブジェクト	5-25
JDBC 2.0 LOB 型と Oracle 拡張型のサポート	5-27
パッケージ oracle.sql	5-28
BLOB、CLOB および BFILE のサポート	5-29
Oracle ROWID のサポート	5-36
Oracle の REF CURSOR 型のサポート	5-39

その他の Oracle9i データ型のサポート	5-41
BigDecimal のサポート強化	5-41

6 オブジェクト、コレクションおよび OPAQUE 型

Oracle のオブジェクトとコレクションについて	6-2
オブジェクトとコレクションの概要	6-2
Oracle オブジェクトの基本概念	6-4
Oracle コレクションの基本概念	6-4
オブジェクトとコレクションのデータ型	6-5
カスタム Java クラス	6-5
カスタム Java クラスのインタフェース指定	6-6
オブジェクト・メソッドでのカスタム Java クラスのサポート	6-9
カスタム Java クラスの要件	6-10
カスタム Java クラスのコンパイル	6-15
カスタム・データの読み込みと書き込み	6-16
ORADATA 実装のその他の使用例	6-16
ユーザー定義データ型	6-21
オブジェクト型の作成	6-21
コレクション型の作成	6-23
JPublisher とカスタム Java クラスの作成	6-26
JPublisher で生成される内容	6-26
カスタム Java クラスの生成	6-30
JPublisher の INPUT ファイルおよびプロパティ・ファイル	6-39
カスタム Java クラスの作成およびメンバー名の指定	6-42
JPublisher で生成されるラッパー・メソッド	6-43
JPublisher のカスタム Java クラスの例	6-44
JPublisher 生成クラスの拡張	6-48
SQLJ 実行文の強い型指定のオブジェクトと参照	6-51
オブジェクトとオブジェクト参照のイテレータ列への取出し	6-52
オブジェクトの更新	6-53
各オブジェクト属性から作成したオブジェクトの挿入	6-55
オブジェクト参照の更新	6-56
SQLJ 実行文の強い型指定のコレクション	6-57
ネストした表へのアクセス: TABLE 構文と CURSOR 構文	6-57
ネストした表を含む行の挿入	6-58

ホスト式へのネストした表の取出し	6-59
TABLE 構文によるネストした表の操作	6-60
ネスト・イテレータによるネストした表からのデータの選択	6-61
ホスト式への VARRAY の取出し	6-63
VARRAY 行への挿入	6-65
シリアル化された Java オブジェクト	6-65
RAW および BLOB 列に対する Java クラスのシリアル化	6-66
SerializableDatum: ORADData の実装	6-68
SQLJ アプリケーションの SerializableDatum	6-71
SerializableDatum (クラス全体)	6-72
弱い型指定のオブジェクト、参照およびコレクション	6-74
弱い型指定のオブジェクト、参照およびコレクションのサポート	6-74
弱い型指定のオブジェクト、参照およびコレクションの使用制限	6-75
Oracle OPAQUE 型	6-75

7 拡張言語機能

接続コンテキスト	7-2
接続コンテキストの概要	7-2
接続コンテキストのロジスティクス	7-3
接続コンテキスト・クラスの宣言と使用方法の補足	7-4
複数の接続コンテキストの例	7-7
接続コンテキスト・クラスの実装と機能	7-8
接続コンテキスト宣言での IMPLEMENTS 句の使用	7-10
接続コンテキストのセマンティクス・チェック	7-11
標準データ・ソースのサポート	7-12
SQLJ 固有のデータ・ソース	7-14
JavaServer Pages 用の SQLJ 固有の接続 JavaBeans	7-18
実行コンテキスト	7-22
実行コンテキストと接続コンテキストとの関係	7-23
実行コンテキスト・インスタンスの作成と指定	7-24
実行コンテキストの同期	7-25
実行コンテキストのメソッド	7-25
実行コンテキストとマルチスレッドの関係	7-31
SQLJ でのマルチスレッド	7-31

イテレータ・クラスの実装と拡張機能	7-34
イテレータ・クラスの実装と機能	7-35
イテレータ宣言での IMPLEMENTS 句の使用	7-36
イテレータ・クラスのサブクラス化のサポート	7-37
結果セット・イテレータ	7-37
スクロール可能なイテレータ	7-38
詳細なトランザクション制御	7-44
SET TRANSACTION 構文	7-45
アクセス・モードの設定	7-46
分離レベルの設定	7-46
JDBC 接続クラスのメソッドの使用	7-47
SQLJ と JDBC の関係動作	7-48
SQLJ 接続コンテキストと JDBC 接続の関係動作	7-48
SQLJ イテレータと JDBC 結果セットの関係動作	7-53
動的 SQL のサポート	7-57
メタ・バインド式	7-57
SQLJ 動的 SQL の例	7-59

8 トランスレータのコマンドラインとオプション

トランスレータのコマンドラインとプロパティ・ファイル	8-2
SQLJ のオプション、フラグおよび接頭辞	8-3
コマンドラインの構文と処理	8-12
プロパティ・ファイルによるオプション設定	8-16
環境変数 SQLJ_OPTIONS によるオプションの設定	8-19
オプション設定の優先順位	8-20
基本的なトランスレータ・オプション	8-21
基本的なコマンドライン専用オプション	8-21
出力ファイルとディレクトリのオプション	8-28
接続オプション	8-32
レポートと行マッピングのオプション	8-43
コード生成、最適化および CHAR 比較のオプション	8-49
拡張トランスレータ・オプション	8-57
オプション設定を他の実行可能プログラムに渡す接頭辞	8-57
特殊処理のフラグ	8-62
セマンティクス・チェックとオフライン解析のオプション	8-67

トランスレータによる代替環境のサポートとオプション	8-76
Java およびコンパイラのオプション	8-77
カスタマイズのオプション	8-84

9 トランスレータとランタイムの機能

トランスレータの内部操作	9-2
Java と SQLJ のコード解析と構文チェック	9-2
SQL のセマンティクス・チェックとオフライン解析	9-2
コードの生成	9-5
Java コンパイル	9-9
プロファイルのカスタマイズ (ISO コード生成の場合)	9-10
トランスレータのエラー、メッセージおよび終了コード	9-12
トランスレータのエラー、警告および情報メッセージ	9-12
トランスレータのステータス・メッセージ	9-14
トランスレータの終了コード	9-15
SQLJ ランタイム	9-15
SQLJ ランタイム・パッケージ	9-16
ランタイム・エラーの分類	9-18
トランスレータおよびランタイムでのグローバリゼーション・サポート	9-18
文字エンコーディングと言語サポート	9-19
SQLJ および Java の文字エンコーディングと言語サポートの設定	9-22
Oracle SQLJ 拡張グローバリゼーション・サポート	9-25
SQLJ 外部でのグローバリゼーション・サポート用の操作	9-28

10 パフォーマンスとデバッグ

パフォーマンス強化の機能	10-2
行ブリフェッチ	10-3
文のキャッシング	10-4
バッチ更新機能	10-11
列の定義	10-22
パラメータ・サイズの定義	10-23
SQLJ のデバッグ機能	10-26
デバッグ用の SQLJ -linemap フラグ	10-26
サーバー側の debug オプション	10-27

AuditorInstaller 専用カスタマイザの概要	10-27
Oracle9i JDeveloper での開発とデバッグの概要	10-27

11 サーバー側 SQLJ

サーバー側 SQLJ の概要	11-2
サーバーで使用する SQLJ コードの作成	11-3
サーバー側でのデータベース接続	11-4
サーバー側でのコーディングの注意事項	11-4
サーバーのデフォルトの出力デバイス	11-6
サーバー側での名前解決	11-6
SQL 名と Java 名	11-7
クライアント側での SQLJ ソースの変換とコンポーネントのロード	11-8
クラスおよびリソースのサーバーへのロード	11-8
ロードされたクラスとリソース・スキーマ・オブジェクトの名前付け	11-10
クラス・ファイルとリソース・ファイルをロードした後のアプリケーションの公開	11-12
要約:サーバー側でのクライアント・アプリケーションの実行	11-13
SQLJ ソースのロードとサーバーでの変換	11-14
SQLJ ソース・コードのサーバーへのロード	11-14
サーバー側の埋込みトランスレータでサポートされるオプション	11-16
ロードされたソース、生成されたクラスおよびリソース・スキーマ・オブジェクトの 名前付け	11-20
サーバー側埋込みトランスレータからのエラー出力	11-22
ソース・ファイルをロードした後の、アプリケーションの公開	11-22
Java スキーマ・オブジェクトの削除	11-22
その他の考慮事項	11-23
サーバーでの Java マルチスレッド	11-23
サーバーでの再帰的 SQLJ コール	11-23
サーバーでのコードの実行状況の確認	11-25

A カスタマイズおよび専用カスタマイザ

プロファイルについて	A-2
コード生成時のプロファイルの生成	A-2
プロファイル・エントリの例	A-3

プロファイルのカスタマイズについて	A-4
カスタマイザ・ハーネスおよびカスタマイザの概要	A-5
カスタマイズ処理の手順	A-5
プロファイルのカスタマイズと登録	A-7
カスタマイズ時のエラー・メッセージとステータス・メッセージ	A-8
カスタマイズされたプロファイルの実行時の機能	A-9
カスタマイズ・オプションとカスタマイザの選択	A-10
カスタマイザ・ハーネスのオプションの概要	A-11
カスタマイザ・ハーネスの汎用オプション	A-13
カスタマイザ・ハーネスの接続用オプション	A-17
専用のカスタマイザの起動に使用するカスタマイザ・ハーネスのオプション	A-20
カスタマイザ固有のオプションの概要	A-23
Oracle カスタマイザのオプション	A-23
他のカスタマイザ用のオプション	A-35
プロファイルのカスタマイズ用の SQLJ トランスレータのオプション	A-36
プロファイルの JAR ファイル	A-36
JAR ファイルの要件	A-37
JAR ファイルのロジスティクス	A-38
プロファイルのセマンティクス・チェック用の SQLCheckerCustomizer	A-38
カスタマイザ・ハーネスの verify オプションによる SQLCheckerCustomizer の起動	A-39
SQLCheckerCustomizer オプション	A-40
デバッグ用の AuditorInstaller カスタマイザ	A-42
オーディタとコード・レイヤーの概要	A-42
カスタマイザ・ハーネスの -debug オプションによる AuditorInstaller の起動	A-43
AuditorInstaller のランタイム出力	A-44
AuditorInstaller オプション	A-45
完全なコマンドラインの例	A-49

B SQLJ エラー・メッセージ

SQLJ 変換時のメッセージ	B-2
SQLJ ランタイム・メッセージ	B-44

索引

はじめに

ここでは、このマニュアルの対象読者、マニュアル構成および表記規則について説明します。また、オラクル社の関連文書も示します。

項目は次のとおりです。

- [対象読者](#)
- [このマニュアルの構成](#)
- [関連文書](#)
- [表記規則](#)

注意： スペースの都合により、このマニュアルから「サンプル・アプリケーション」の章は削除されました。このマニュアルで説明する多くのプログラミング機能の例は、**Oracle SQLJ** に同梱されているデモ・アプリケーションを実行してください。「[関連文書](#)」の最後に OTN-J へのリンクも記載してあります。

対象読者

このマニュアルは、SQLJ プログラミングに関心があり、次の項目についてある程度の知識のある方を対象としています。

- Java
- SQL
- Oracle PL/SQL
- JDBC
- Oracle データベース

基本的に SQL および JDBC を理解していれば十分ですが、Oracle 固有の SQL および JDBC の機能についての知識があれば、このマニュアルをより深く理解できます。

オラクル社の SQL および JDBC 関連の Oracle マニュアルは、xvi ページの [「関連文書」](#) を参照してください。

このマニュアルの構成

次の 2 つは、SQLJ を使用して行う主な作業です。

- SQLJ ソース・コードの作成
- SQLJ トランスレータの実行

第 3 章から第 7 章では、コードを作成するために必要な内容を説明します。第 3 章および第 4 章は特に重要です。

トランスレータのオプションおよび機能は、第 8 章で説明します。

このマニュアルは、次の章と付録で構成されています。

第 1 章「概要」

SQLJ の概念、コンポーネントおよびプロセスを紹介します。配布方法や開発環境の例も示します。

第 2 章「SQLJ の基本事項」

Oracle データベース、Oracle JDBC ドライバおよび Oracle SQLJ インストールのテスト方法および確認方法を手順に従って説明します。

第 3 章「基本的な言語機能」

基本的なアプリケーションの作成に使用する SQLJ のプログラミング機能について説明します。ここでは、Oracle 拡張機能ではなく、標準 SQLJ 構文に重点を置いて説明します。

第4章「プログラミング上の主な考慮事項」

接続、NULL 値の処理、例外処理、Oracle 固有コード生成など、ソース・コードを作成する際に考慮する事項について説明します。

第5章「型のサポート」

Oracle SQLJ でサポートされている Java 型の一覧を示すとともに、ストリーム型の使用方法について説明します。また、Oracle データ型の拡張とそれに対応する Java 型についても説明します。

第6章「オブジェクト、コレクションおよび OPAQUE 型」

Oracle SQLJ におけるユーザー定義のオブジェクト型およびコレクション型のサポートについて説明します。たとえば、これらのユーザー定義型に対応する Java 型を生成する際の、Oracle JPublisher ユーティリティの使用法などを取り上げます。Oracle OPAQUE 型のサポートについても簡単に説明します。

第7章「拡張言語機能」

高度なアプリケーションを作成するための、SQLJ のプログラミング機能について説明します。

第8章「トランスレータのコマンドラインとオプション」

Oracle SQLJ トランスレータのコマンドラインの構文、プロパティ・ファイルおよびオプションについて説明します。

第9章「トランスレータとランタイムの機能」

トランスレータ操作の機能、トランスレータとランタイムのエラー・メッセージ、グローバル化・サポートについて説明します。

第10章「パフォーマンスとデバッグ」

標準および Oracle 固有のパフォーマンス・チューニング機能について説明し、SQLJ のデバッグ機能に関する一般的な考慮事項を示します。

第11章「サーバー側 SQLJ」

サーバーで実行する SQLJ アプリケーションの作成方法およびロード方法について説明します。これらのアプリケーションは通常、ストアード・プロシージャまたはストアード・ファンクションとして実行されます。また、サーバー側の埋込みトランスレータ（オプション）についても説明します。

付録 A 「カスタマイズおよび専用カスタマイザ」

ISO 標準コード生成に必要な SQL 操作の実装に使用する SQLJ プロファイルについて説明します。特に、特定の環境用のプロファイルのカスタマイズに関連し、変換時に指定できる各種オプションについて説明します。また、プロファイルのセマンティクス・チェック用カスタマイザ、デバッグに使用する「オーディタ」のインストール用カスタマイザなど、専用カスタマイザについて説明します。

付録 B 「SQLJ エラー・メッセージ」

Oracle SQLJ トランスレータおよびランタイムのエラー・メッセージと各エラーの原因および処置の一覧を示します。

関連文書

Oracle Java Platform グループからは、Oracle9i の次のマニュアルも入手できます。

- 『Oracle9i Java Developer's Guide』

Oracle9i における Java の基本概念を紹介し、サーバー側の構成および機能性に関する概要を述べたマニュアルです。Oracle データベースの Java 環境について、特定の製品 (JDBC、SQLJ など) に限定せず一般的な知識を得るには、このマニュアルを参照してください。

- 『Oracle9i JDBC 開発者ガイドおよびリファレンス』

JDBC 規格 (Java Database Connectivity 対応) の Oracle 製品の特長およびプログラミング構文について解説したマニュアルです。Oracle JDBC ドライバの概要、JDBC 1.22、2.0 および 3.0 の機能に準拠した Oracle 実装の解説、Oracle JDBC 型の拡張およびパフォーマンス向上に関する説明などが収録してあります。

- 『Oracle9i JPublisher ユーザーズ・ガイド』

オブジェクト型などのユーザー定義型を Java クラスに変換する際の、Oracle JPublisher ユーティリティの使用方法を説明したマニュアルです。オブジェクト型、VARRAY 型、NESTED TABLE 型またはオブジェクト参照型が使用可能な SQLJ または JDBC アプリケーションを開発する際は、JPublisher を使用することをお勧めします。ユーザー定義 Java クラスの生成や、それらの型へのマッピングが実行できます。

- 『Oracle9i JavaServer Pages サポート・リファレンス』

Java コードおよび JavaBean の起動を HTML ページ内部に埋め込むための JavaServer Pages テクノロジの使用について解説したマニュアルです。標準 JSP 機能と Oracle 固有の機能の両方について説明しています。Oracle9i リリース 2 (9.2) の Apache JServ 環境に関する考慮事項とともに、Servlet 2.2 環境における機能および JServ 用の Oracle JSP コンテナによるこれらの機能の実行方法を取り上げています。

- 『Oracle9i Java Stored Procedures Developer's Guide』

Java ストアド・プロシージャと呼ばれる、Oracle9i データベースで直接実行されるプログラムについて説明するマニュアルです。ストアド・プロシージャ（ファンクション、プロシージャ、トリガーおよび SQL メソッド）を使用すると、Java 開発におけるビジネス・ロジックがサーバー・レベルで実装されるため、アプリケーションのパフォーマンス、拡張性およびセキュリティが改善できます。

Oracle Java Platform グループからは、Oracle9i Application Server の次の OC4J マニュアルが入手できます。

- 『Oracle9iAS Containers for J2EE ユーザーズ・ガイド』

OC4J の概要および一般情報を記載したマニュアルです。サーブレット、JSP ページおよび EJB に関する入門的な章が含まれ、一般的な構成とデプロイについて説明しています。

- 『Oracle9iAS Containers for J2EE JavaServer Pages サポート・リファレンス』

OC4J で独自のページを実行する JSP 開発者向けの情報を記載したマニュアルです。JSP 規格の概要とプログラミングに関する考慮事項も含まれています。また、OC4J 環境を初めて使用する方のために、Oracle に追加された機能および手順についても説明しています。

- 『Oracle9iAS Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』

タグ・ライブラリ、JavaBeans および OC4J が提供する他の Java ユーティリティに関する概念的な情報、詳細な構文および使用に関する情報を記載したマニュアルです。

- 『Oracle9iAS Containers for J2EE Servlet 開発者ガイド』

サーブレット開発者向けに、OC4J でのサーブレットおよびサーブレット・コンテナの使用方法に関する情報を記載したマニュアルです。関連する OC4J 構成ファイルについても説明しています。

- 『Oracle9iAS Containers for J2EE サービス・ガイド』

JTA、JNDI および Oracle9i Application Server の Java Object Cache など、OC4J が提供する基本的な Java サービスに関する情報を記載したマニュアルです。

- 『Oracle9iAS Containers for J2EE Enterprise JavaBeans 開発者ガイドおよびリファレンス』

OC4J での EJB 実装および EJB コンテナに関する情報を記載したマニュアルです。

Oracle Server Technologies グループからは、次のマニュアルが入手できます。

- 『Oracle9i XML データベース開発者ガイド - Oracle XML DB』
- 『Oracle9i XML Developer's Kit ガイド - XDK』
- 『Oracle9i アプリケーション開発者ガイド - 基礎編』
- 『Oracle9i アプリケーション開発者ガイド - ラージ・オブジェクト』
- 『Oracle9i アプリケーション開発者ガイド - オブジェクト・リレーショナル機能』
- 『Oracle9i Java パッケージ・プロシージャ・リファレンス』
- 『Oracle9i 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』
- 『PL/SQL ユーザーズ・ガイドおよびリファレンス』
- 『Oracle9i SQL リファレンス』
- 『Oracle9i Net Services 管理者ガイド』
- 『Oracle Advanced Security 管理者ガイド』
- 『Oracle9i Database グローバリゼーション・サポート・ガイド』
- 『Oracle9i データベース・リファレンス』
- 『Oracle9i データベース・エラー・メッセージ』
- 『Oracle9i サンプル・スキーマ』

Oracle9i Application Server グループの次のマニュアルにも関連情報が含まれています。

- 『Oracle9i Application Server 管理者ガイド』
- 『Oracle Enterprise Manager 管理者ガイド』
- 『Oracle9i Application Server Oracle HTTP Server 管理ガイド』
- 『Oracle9i Application Server パフォーマンス・ガイド』
- 『Oracle9i Application Server グローバリゼーション・サポート・ガイド』
- 『Oracle9i AS Web Cache 管理および配置ガイド』
- 『Oracle9i Application Server Oracle Application Server からの移行』

Oracle9i JDeveloper グループからは、次のドキュメントが入手できます。

- JDeveloper オンライン・ヘルプ
- OTN-J (Oracle Technology Network Japan) 上の JDeveloper マニュアル:

<http://otn.oracle.co.jp/>

リリース・ノート、インストレーション・マニュアル、ホワイト・ペーパーまたはその他の関連文書は、OTN-J (Oracle Technology Network Japan) に接続すれば、無償でダウンロードできます。OTN-J を使用するには、オンラインでの登録が必要です。次の URL で登録できます。

<http://otn.oracle.co.jp/membership/>

OTN-J のユーザー名とパスワードを取得済みの場合は、次の OTN-J Web サイトの文書セクションに直接接続できます。

<http://otn.oracle.co.jp/document/>

SQLJ 規格機能および構文の詳細は、ANSI 仕様 X3.135.10-1998 を参照してください。

- Information Technology - Database Languages - SQL - Part 10: Object Language Bindings (SQL/OLB)

このドキュメントは、次の ANSI の Web サイトで閲覧できます。

<http://www.ansi.org/>

(「Electronic Standards Store」をクリックして、仕様番号を検索してください。)

SQLJ のサンプル・アプリケーションは、次の場所にあります。

<http://otn.oracle.co.jp/>

表記規則

このマニュアル・セットの本文とコード例に使用されている表記規則について説明します。

- [本文の表記規則](#)
- [コード例の表記規則](#)

本文の表記規則

本文中には、特別な用語が一目でわかるように様々な表記規則が使用されています。次の表は、本文の表記規則と使用例を示しています。

規則	意味	例
固定幅フォントの大文字	固定幅フォントの大文字は、システムにより指定される要素を示します。この要素には、パラメータ、権限、データ型、Recovery Manager 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、パッケージとメソッドの他、システム指定の列名、データベース・オブジェクトと構造体、ユーザー名、およびロールがあります。	この句は、NUMBER 列に対してのみ指定できます。 BACKUP コマンドを使用すると、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・ビューの TABLE_NAME 列を問い合わせます。 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サンプルのユーザー指定要素を示します。この要素には、コンピュータ名とデータベース名、ネット・サービス名、接続識別子の他、ユーザー指定のデータベース・オブジェクトと構造体、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があります。 注意： 一部のプログラム要素には、大文字と小文字の両方が使用されます。この場合は、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 ファイルに指定されています。 データ・ファイルと制御ファイルのバックアップを /disk1/oracle/dbs ディレクトリに作成します。 department_id、department_name および location_id の各列は、hr.departments 表にあります。 初期化パラメータ QUERY_REWRITE_ENABLED を true に設定します。 oe ユーザーで接続します。 これらのメソッドは JRepUtil クラスに実装されます。
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。 old_release は、アップグレード前にインストールしたリリースです。

コード例の表記規則

コード例は、SQL、PL/SQL、SQL*Plus またはその他のコマンドラインを示します。次のように、固定幅フォントで、通常の本文とは区別して記載されています。

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

次の表は、コード例の記載上の表記規則とその使用例を示しています。

規則	意味	例
<>	このマニュアルでは、1つ以上のオプション項目は、通常の大カッコや中カッコではなく、山カッコで囲まれています。山カッコ自体は入力しないでください。(通常の大カッコおよび中カッコは、SQLJ 構文との混同を避けるために使用していません。)	DECIMAL (<i>digits</i> < , <i>precision</i> >)
	縦線は、山カッコ内の複数の選択肢を区切るために使用します。オプションのうち1つを入力します。縦線自体は入力しないでください。	<ENABLE DISABLE> <COMPRESS NOCOMPRESS>
...	水平の省略記号は、次のどちらかを示します。 ■ 例に直接関係のないコード部分が省略されていること。 ■ コードの一部が繰り返し可能であること。	CREATE TABLE ... AS <i>subquery</i> ; SELECT <i>col1</i> , <i>col2</i> , ... , <i>coln</i> FROM <i>employees</i> ;
その他の表記	山カッコ、縦線および省略記号以外の記号は、示されているとおりに入力してください。	<i>acctbal</i> NUMBER(11,2); <i>acct</i> CONSTANT NUMBER(4) := 3;
イタリック	イタリックの文字は、特定の値を指定する必要があるプレースホルダまたは変数を示します。	CONNECT SYSTEM/ <i>system_password</i> DB_NAME = <i>database_name</i>
大文字	大文字は、システムにより指定される要素を示します。これらの用語は、ユーザー定義用語と区別するために大文字で記載されています。山カッコで囲まれている場合を除き、記載されているとおりの順序とスペルで入力してください。ただし、この種の用語は大 / 小文字区別がないため、小文字でも入力できます。	SELECT <i>last_name</i> , <i>employee_id</i> FROM <i>employees</i> ; SELECT * FROM USER_TABLES; DROP TABLE hr.employees;
小文字	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、列名またはファイル名を示します。 注意： 一部のプログラム要素には、大文字と小文字の両方が使用されます。この場合は、記載されているとおりに入力してください。	SELECT <i>last_name</i> , <i>employee_id</i> FROM <i>employees</i> ; sqlplus hr/hr CREATE USER <i>mjones</i> IDENTIFIED BY <i>ty3MU9</i> ;

この章では、SQLJ の機能と使用例について概要を説明します。次の項目について説明します。

- [SQLJ について](#)
- [SQLJ コンポーネントの概要](#)
- [SQLJ 規格に準拠した Oracle 拡張型の概要](#)
- [基本的な変換処理とランタイム処理](#)
- [JDBC と SQLJ のサンプル・コード](#)
- [他の配置例](#)
- [他の開発環境](#)

SQLJ について

SQLJ の基本概念を紹介し、Oracle アプリケーションにおける Java と PL/SQL との間の相補的な関係を説明します。

基本的な概念

SQLJ を使用したアプリケーション・プログラミングでは、Java の設計思想に基づいて、Java コード中に SQL 操作の埋込みが可能です。SQLJ プログラムとは、SQL 文が埋め込まれた Java プログラムのことで、ISO 規格の SQLJ 言語リファレンス構文に準拠しています。Oracle SQLJ では、ISO SQLJ 標準仕様をサポートします。標準仕様には、静的 SQL 操作のみが含まれます。静的 SQL 操作は事前定義され、ユーザーがアプリケーションを起動する際、リアルタイムで変更されることはありません（転送されたデータ値は動的に変更されず）。Oracle SQLJ には、動的 SQL 操作をサポートする拡張機能もあります。動的 SQL 操作は事前定義されていないため、操作自体をリアルタイムに変更できます。（SQLJ アプリケーション内の JDBC コードまたは PL/SQL コードによって、動的 SQL 操作を使用することも可能です。）たいいていのアプリケーションでは、静的 SQL の方が動的 SQL よりもはるかに多く埋め込んであります。

SQLJ は、トランスレータとランタイム・コンポーネントで構成され、開発環境にスムーズに統合できます。sqlj フロントエンド・ユーティリティを使用すると、開発段階でトランスレータを実行しながら、変換、コンパイルおよび（ISO 標準コードの）カスタマイズが 1 つのステップで実行されます。変換処理では、埋込み SQL を SQL 操作を実装した SQLJ ランタイムのコールに置き換えられます。ISO 標準 SQLJ では、このための手段として、通常、JDBC ドライバのコールが使用されます（必須ではありません）。Oracle データベースにアクセスするには、通常、Oracle JDBC ドライバを使用します。エンド・ユーザーが SQLJ アプリケーションを実行すると、SQL 操作を処理できるように SQLJ ランタイムが起動します。

Oracle SQLJ トランスレータは、Oracle の他のプリコンパイラと概念的に似ています。トランスレータを使用すると、開発者は、SQL 構文のチェック、SQL 操作がスキーマで使用可能かどうかの検証および Java 型と対応するデータベース型との互換性チェックを行えます。これらのチェックを開発段階で行うことにより、実行時のエラーの発生を防ぐことができます。トランスレータは次のチェックを行います。

- 埋込み SQL の構文をチェックします。
- SQL 構文と所定のデータベース・スキーマとを照合し、特定の SQL エンティティ内の整合性を（必要に応じて）確認します。
具体的には、表や列の名前などを検証します。
- データ型をチェックします。Java と SQL 間で互換性のあるデータ型が交換され、型変換が適切に行われることが確認されます。

SQL 操作を Java コードに直接埋め込む SQLJ 方式は、JDBC 方式より扱いやすく簡潔です。SQLJ を使用すると、データベース接続を必要とする Java プログラムの開発費とメンテナンス費を節約できます。

Oracle 固有コード生成と ISO 標準コード生成との違い

Oracle SQLJ は ISO の SQLJ 標準をサポートしていますが、Oracle 固有コード生成のオプションも提供します。この場合は、コード内に Oracle JDBC コールが直接生成されます。Oracle9i リリース 2 (9.2) では、Oracle 固有コード生成はデフォルトの動作です。Oracle 固有コード生成の場合は、次の点に注意してください。

- プロファイル・ファイルがないため、変換時のカスタマイズ処理也没有ありません。
- 実行時は、変換されたコード内に JDBC コール (SQLJ ランタイム・コールではなく) が直接存在するため、SQL 操作は、SQL ランタイム・レイヤーを経由する必要がありません。

この章で SQLJ を紹介している説明の大部分は、ISO 標準コードの機能について述べています。したがって、Oracle 固有コードの場合には、これらの主な相違点に注意してください。

詳細は、4-36 ページの「[Oracle 固有コード生成 \(プロファイルなし\)](#)」を参照してください。

Java および SQLJ と、PL/SQL との違い

Oracle アプリケーションでは、Java (SQLJ を含む) を PL/SQL で代用することは不可能です。Java と PL/SQL は、互いに補いあって機能しているためです。

PL/SQL と Java は、データベース・アプリケーション作成時に併用することも可能ですが、本来の目的が異なるため、それぞれに適したアプリケーションの種類は異なります。

- PL/SQL は、SQL 集中型のアプリケーションに適したソリューションです。PL/SQL は SQL 用に最適化されているため、Java でなく PL/SQL で SQL 操作を記述した方が処理時間を短縮できます。また、Java アプリケーションを使用した場合は SQL データ型と Java 型との変換が必要になりますが、PL/SQL を使用すると SQL データ型を直接使用できます。
- Java は、高度なプログラミング・モデルを使用しているため、論理集中型のアプリケーションに適したソリューションとなっています。また、Java のシステムは汎用性に優れているため、PL/SQL よりも Java のほうが、コンポーネント指向のアプリケーションに適しています。

Oracle では PL/SQL と Java を容易に連係できるので、この両方の言語の長所を利用できます。PL/SQL プログラムでは、Java ストアド・プロシージャの透過的なコールによって、コンポーネントベースの Enterprise JavaBeans アプリケーションの作成を実現しています。PL/SQL コール指定によって、PL/SQL プログラムから既存の各種 Java クラス・ライブラリへの透過的なアクセスが可能となります。

一方、Java プログラムの場合、JDBC または SQLJ を使用することにより、PL/SQL ストアド・プロシージャおよび無名ブロックのコールが可能となります。特に SQLJ では、SQLJ 文の内部からストアド・プロシージャおよびファンクションをコールしたり、SQLJ 文中に無名 PL/SQL ブロック埋込みも可能です。

注意： SQLJ 文中での無名 PL/SQL ブロックの使用は、SQLJ アプリケーション内で動的 SQL をサポートする 1 つの方法です。ただし、Oracle SQLJ には、動的 SQL 操作を直接サポートする拡張機能があります。(7-57 ページの「[動的 SQL のサポート](#)」を参照してください。)

SQLJ コンポーネントの概要

SQLJ の主要コンポーネントとプロファイルの概念について説明します。(プロファイルは、ISO コード生成の場合にのみ使用します。)

SQLJ トランスレータと SQLJ ランタイム

Oracle SQLJ は、次の 2 つの主要コンポーネントで構成されています。

- Oracle SQLJ トランスレータ (以降、トランスレータと呼びます) : このコンポーネントは、開発者が SQLJ ソース・コード作成後に実行するプリコンパイラです。

トランスレータは Pure Java で書かれており、SQL 操作を SQLJ 実行文に埋め込むためのプログラミング構文をサポートします。SQLJ 実行文は、SQLJ 宣言と同様に #sql トークンで開始し、SQLJ ソース・コード・ファイル内で Java 文と混在できます。SQLJ ソース・コード・ファイルの拡張子は、必ず .sqlj にします。次に、SQLJ 文の例を示します。

```
#sql { INSERT INTO emp (ename, sal) VALUES ('Joe', 43000) };
```

トランスレータから、.java ファイルと、ISO 標準 SQLJ コード生成の場合は、1 つ以上の SQLJ プロファイルが作成されます。このプロファイルには、SQL 操作に関する情報が格納されます。SQLJ では、その後 Java コンパイラが自動的に起動され、.java ファイルから .class ファイルが生成されます。

注意： Oracle9i リリース 2 (9.2) のデフォルト設定では、Oracle 固有コード生成によって、Oracle JDBC コードに直接変換されます。この場合、プロファイルは生成されません。4-36 ページの「[Oracle 固有コード生成 \(プロファイルなし\)](#)」を参照してください。

- **Oracle SQLJ ランタイム**（以降、ランタイムと呼びます）：このコンポーネントも Pure Java で書かれており、エンド・ユーザーが SQLJ アプリケーションを実行するたびに自動的に起動されます。

ISO 標準コード生成の場合、SQLJ ランタイムは SQL 操作の処理を実装し、JDBC ドライバを介してデータベースにアクセスします。SQLJ ランタイムからデータベースへのアクセスに使用する JDBC ドライバは、汎用の ISO SQLJ 標準では不要ですが、Oracle SQLJ には必要です。ただし、実際に Oracle JDBC ドライバが必要になるのは、デフォルトの Oracle カスタマイザ（後述）を使用してアプリケーションをカスタマイズした場合です。

Oracle 固有コード生成の場合（これがデフォルトです）は、Oracle JDBC コールが変換されたコードに直接生成されるため、SQLJ ランタイムの役割は非常に小さくなります。

ランタイムの詳細は、9-15 ページの「[SQLJ ランタイム](#)」を参照してください。

トランスレータとランタイムに加え、ISO 標準コード生成を使用する場合は、さらに**カスタマイザ**と呼ばれるコンポーネントも役割を担います。カスタマイザを使用すると、SQLJ プロファイルが、特定のデータベースの実装およびベンダー固有の機能およびデータ型に合うように調整されます。デフォルトの設定では、ISO 標準コードの場合は、Oracle SQLJ フロントエンドによって Oracle カスタマイザが起動され、Oracle データベースと Oracle 固有の機能およびデータ型に合うように、プロファイルが調整されます。

Oracle カスタマイザを変換時に使用する場合、アプリケーションの実行時に Oracle SQLJ ランタイムおよび Oracle JDBC ドライバが必要になります。

SQLJ プロファイル（ISO 標準コードの場合）

ISO 標準 SQLJ コード生成の場合、SQLJ プロファイルとは、シリアル化された Java リソース（またはクラス）のことで、SQLJ トランスレータによって生成されます。このプロファイルには、SQLJ ソース・コード中の埋込み SQL 操作に関する詳細情報が格納されます。トランスレータはプロファイルを生成してから、シリアル化し、バイナリ・リソース・ファイルまたは .class ファイルに出力します。出力先は、トランスレータのオプション設定で指定できます。

注意： Oracle9i リリース 2 (9.2) のデフォルトでは、Oracle 固有コード生成が使用されます。この場合、トランスレータによって Oracle JDBC コールが直接生成され、埋込み SQL 操作の細部が JDBC コールに組み入れられます。プロファイルはありません。4-36 ページの「[Oracle 固有コード生成（プロファイルなし）](#)」を参照してください。

プロファイルの概要

ISO 標準コードでは、SQLJ 実行文に埋込み SQL 操作を実装する場合に SQLJ プロファイルを使用します。プロファイルの内容は、SQL 操作およびアクセスするデータの型とモードに関する情報です。プロファイルはエントリの集合であり、各エントリが 1 つの SQL 操作に

対応します。各エントリの内容は、該当する SQL 操作の詳細であり、この指示の実行時に使用する各パラメータが記述されています。

ISO コード生成の場合、SQLJ では、アプリケーション内の接続コンテキスト・クラスごとにプロファイルが生成されます。通常、この接続コンテキスト・クラスは、データベースの操作に使用する特定の SQL エンティティ別に 1 対 1 で対応しています。(デフォルトの接続コンテキスト・クラスが 1 つあり、必要に応じて、さらにコンテキスト・クラスを宣言できます。) ISO SQLJ 標準の規定では、プロファイルを標準の形式および内容にする必要があります。そのため、アプリケーションでベンダー固有の拡張機能を使用する場合は、プロファイルのカスタマイズが必要になります。デフォルトの設定では、このカスタマイズが自動的に行われます。Oracle 固有の拡張機能を使用するように、プロファイルがカスタマイズされます。

プロファイルをカスタマイズすると、ベンダーは次の 2 つの付加価値が得られます。

- ベンダー固有のデータ型と SQL 構文が使用できます。たとえば、Oracle カスタマイズを使用して、変換済み SQLJ コードの JDBC 標準 `PreparedStatement` メソッド・コールを `OraclePreparedStatement` メソッド・コールにマッピングすることにより、Oracle 拡張型がサポートされます。
- ベンダー固有の最適化によってパフォーマンスを向上できます。

プロファイルのカスタマイズは、Oracle オブジェクトを SQLJ アプリケーションで使用する場合などに必要となります。

注意：

- デフォルトの設定では、SQLJ プロファイルの拡張子が `.ser` になりますが、すべての `.ser` ファイルがプロファイルであるとは限りません。その他のシリアル化されたオブジェクトにもこの拡張子を使用できるため、SQLJ プログラム・ユニットによってプロファイル以外のシリアル化されたオブジェクトが使用される可能性があります。(プロファイルは、`.ser` ファイルではなく `.class` ファイルにも変換できます。)
 - ソース・コード中に SQLJ 実行文を記述しないかぎり、SQLJ プロファイルは生成されません。
-

バイナリ移植性

SQLJ で生成したプロファイル・ファイルは、バイナリ移植性をサポートしています。つまり、プロファイル・ファイルをそのまま異種データベースまたは他の環境に移植して使用できます。ただし、ベンダー固有のデータ型または機能を使用していない場合に限りです。SQLJ で生成した `.class` ファイルについても同様です。

SQLJ 規格に準拠した Oracle 拡張型の概要

Oracle SQLJ では、ISO SQLJ 仕様をサポートします。ISO SQLJ 規格は ANSI SQLJ 規格のスーパーセットであるため、J2EE でコンパイルする JDK 1.2 以上の環境を必要とします。ANSI SQLJ 規格は JDK 1.1.x のみを必須とします。Oracle SQLJ トランスレータは、ANSI SQLJ 規格の指定よりもより幅の広い SQL 構文を受け入れます。

ANSI SQLJ 規格は、SQL の SQL92 言語のみを規定していますが、拡張も許容しています。Oracle SQLJ では、Oracle の SQL 言語、つまり SQL92 のスーパーセットを使用できます。したがって、他の DBMS ベンダーと連携する SQLJ プログラムを作成する場合は、他の環境でサポートされない場合があるため、ANSI SQLJ 規格以外の SQL 構文と SQL 型の使用を避ける必要があります。（製品のディレクトリ \$Oracle_Home/sqlj/demo/components には、セマンティクス・チェックが収録されているので、SQLJ 文が標準 SQL のみに統一されていることの検証に使用してください。）

Oracle SQLJ 拡張機能の概要は、[第 5 章「型のサポート」](#) および [第 6 章「オブジェクト、コレクションおよび OPAQUE 型」](#) を参照してください。

Oracle SQLJ の拡張型

Oracle SQLJ では、次の Java 型が SQLJ 規格の拡張としてサポートされます。自作コードが他の環境で使用される場合は、拡張型を使用しないでください。アプリケーションの移植性を保証するには、Oracle SQLJ の `-warn=portable` フラグを使用します。8-44 ページの「[トランスレータからの警告 \(-warn\)](#)」を参照してください。

次に示す拡張要素のいずれかを使用した場合、変換時に Oracle 固有コード生成または Oracle のカスタマイズが必要になる他、アプリケーション実行時に Oracle SQLJ ランタイムおよび Oracle JDBC ドライバも必要になります。

- `oracle.sql.*` クラスのインスタンス。SQL データのラッパーとして使用します。
5-27 ページの「[JDBC 2.0 LOB 型と Oracle 拡張型のサポート](#)」を参照してください。
- カスタム Java クラス (`oracle.sql.ORADData` インタフェースまたは JDBC 標準 `java.sql.SQLData` インタフェースを実装したクラス)。通常、Oracle9i JPublisher ユーティリティで生成し、SQL オブジェクト、オブジェクト参照およびコレクションに対応付けます。
6-5 ページの「[カスタム Java クラス](#)」を参照してください。ただし、`SQLData` インタフェースが標準です。このインタフェースを実装するクラスは、他のベンダーの JDBC ドライバおよびデータベースでもサポートされている場合があります。
- ストリームのインスタンス (`BinaryStream` および `CharacterStream`)。後者は現在使用されていない `AsciiStream` および `UnicodeStream` にかわるものです。出力パラメータとして使用します (5-13 ページの「[ストリームのサポート](#)」を参照してください)。

- イテレータと結果セットのインスタンス。入力または出力パラメータとして様々な場所で使用されます。

SQLJ 規格では、結果式またはキャスト文での使用のみを規定しています。3-47 ページの「[ホスト変数としてのイテレータおよび結果セットの使用](#)」および 3-57 ページの「[ストアド・ファンクションの戻り値としてのイテレータおよび結果セットの使用](#)」を参照してください。

- Unicode キャラクタ型 (NString、NCHAR、NCLOB および NcharCharacterStream)。NcharCharacterStream は、現在使用されていない NcharAsciiStream および NcharUnicodeStream にかわるものです (9-25 ページの「[Oracle SQLJ 拡張グローバル化・サポート](#)」を参照してください)。

Oracle SQLJ の拡張機能

Oracle SQLJ では、次の拡張機能もサポートしています。

- Oracle 固有コード生成

これによって、JDBC コードを直接生成します。プロファイルは生成されず、SQLJ ランタイム機能の多くはプログラム実行中に無視されます。4-36 ページの「[Oracle 固有コード生成 \(プロファイルなし\)](#)」を参照してください。

- SQLJ 文中の動的 SQL

7-57 ページの「[動的 SQL のサポート](#)」を参照してください。

- 付加的なナビゲーション・メソッドによるスクロール可能な結果セット・イテレータ、および結果セット・イテレータやスクロール可能な結果セット・イテレータからの FETCH 構文

7-38 ページの「[スクロール可能なイテレータ](#)」を参照してください。

- 列定義とパラメータ・サイズ定義の最適化フラグ

10-22 ページの「[列の定義](#)」、10-23 ページの「[パラメータ・サイズの定義](#)」および 8-49 ページの「[コード生成、最適化および CHAR 比較のオプション](#)」を参照してください。

- 変更された変換動作のフラグ。識別子によってホスト式をバインドし、WHERE 句の CHAR 比較で空白埋めを考慮します。

8-66 ページの「[識別子によるホスト式のバインド \(-bind-by-identifier\)](#)」および 8-56 ページの「[空白埋めを考慮した CHAR 比較 \(-fixedchar\)](#)」を参照してください。

- 接続コンテキストでの SQLJ 文のキャッシング

10-4 ページの「[文のキャッシング](#)」を参照してください。

基本的な変換処理とランタイム処理

ここでは、次の内容について説明します。

- Oracle SQLJ トランスレータによる SQLJ ソース・コードの基本的な変換処理
- トランスレータ入出力の概要
- ユーザーがアプリケーションを実行するときのランタイム処理

変換処理の詳細は、9-2 ページの「[トランスレータの内部操作](#)」を参照してください。

SQLJ ソース・コードの内容は、標準 Java ソース・コードと SQLJ クラス宣言および SQLJ 実行文です。SQLJ 実行文には SQL 操作が埋め込まれています。

SQLJ ソース・ファイルの拡張子は `.sqlj` です。ファイル名は、必ず該当の Java 識別子にします。ソース・ファイルで `Public` クラス（最大 1 クラス）を宣言している場合は、このクラスの名前をファイル名にする必要があります。ソース・ファイルに `Public` クラスが宣言されていない場合は、1 番目に定義されているクラスと同じ名前をファイル名に付けてください。

SQLJ 変換処理

`.sqlj` ファイルを記述した後、SQLJ を実行してファイル进行处理します。（`.sqlj` ファイルのコーディングについては、基本的な SQLJ プログラミング機能を第 3 章および第 4 章で説明します。）次の例では、最初の `Public` クラスが `Foo` であるソース・ファイル `Foo.sqlj` に対して、SQLJ を実行します。ここでは、コマンドライン・オプションを指定しない最も単純な方法で実行します。

```
sqlj Foo.sqlj
```

このコマンドが実際に実行するものは、フロントエンドのスクリプトまたはユーティリティです。どちらが実行されるかは、プラットフォームによります。このスクリプトまたはユーティリティは、コマンドラインを読み込み、Java Virtual Machine (JVM) を起動し、引数をこの JVM に渡します。JVM は SQLJ トランスレータを起動し、フロントエンドとして機能します。

このドキュメントでは、フロントエンドの実行を「SQLJ の実行」、そのコマンドラインを「SQLJ コマンドライン」と呼びます。コマンドライン構文の詳細は、8-12 ページの「[コマンドラインの構文と処理](#)」を参照してください。

以降の処理は、次の順に展開されます（各手順で致命的エラーが発生しなかった場合を想定しています）。詳細は、9-2 ページの「[トランスレータの内部操作](#)」を参照してください。

1. JVM で SQLJ トランスレータを起動します。
2. トランスレータが .sqlj ファイル内の SQLJ および Java コードを解析します。SQLJ 構文が検証され、宣言した SQL データ型とその Java ホスト変数の型の不一致が検出されます。（ホスト変数はローカルな Java 変数であり、SQL 操作の入力パラメータまたは出力パラメータとして使用します。これらは、3-14 ページの「[Java ホスト式、コンテキスト式および結果式](#)」で説明します。）

3. SQLJ のオプション設定に従って、トランスレータがオンライン・セマンティクス・チェッカまたはオフライン・パーサー（あるいはその両方）を起動します。これによって、埋込み SQL 文や PL/SQL 文の構文が検証され、オンライン・チェックの場合はコード内のデータベース要素の使用が該当データベース・スキーマと照合してチェックされます。いずれのオプションも指定されていない場合でも、基本的なレベルのチェックは実行されます。

オンライン・チェックを指定すると、SQLJ では指定したデータベース・スキーマへの接続後に、アプリケーションで使用しているデータベース表、ストアド・プロシージャおよび SQL 構文がすべてサポートされているかどうかを検証されます。また、SQLJ アプリケーション内のホスト変数の型と対応するデータベース列のデータ型の互換性も検証されます。

4. Oracle 固有の SQLJ コード生成（デフォルトの `-codegen=oracle`）の場合は、SQL 操作が Oracle JDBC コールに直接変換され、プロファイルは生成されません。4-36 ページの「[Oracle 固有コード生成（プロファイルなし）](#)」を参照してください。

ISO 標準コード生成（`-codegen=iso`）の場合は、トランスレータが SQLJ ソース・コードを処理し、SQL 操作を SQLJ ランタイム・コールに変換し、Java 出力コードと 1 つ以上の SQLJ プロファイルを生成します。プロファイルは、ソース・コード中の接続コンテキスト・クラスごとに生成されます。接続コンテキスト・クラスは、操作で使用する SQL エンティティの種類ごとに異なるためです。

生成された Java コードは .java 出力ファイルに保存されます。次の内容がこのファイルに出力されます。

- .sqlj ソース・ファイル内のクラス定義と Java コード。
- SQLJ イテレータおよび接続コンテキストの宣言に基づいて生成されたクラス定義。
3-2 ページの「[SQLJ 宣言の概要](#)」を参照してください。
- 専用クラス（プロファイルキー・クラス）のクラス定義。SQLJ では、このクラスは、プロファイルと一緒に生成され、使用されます（ISO 標準 SQLJ コード生成のみ）。
- Oracle JDBC（Oracle 固有コード生成の場合）へのコール、または SQLJ ランタイム（ISO 標準コード生成の場合）へのコール。埋込み SQL 操作のアクションを実装します。

生成されたプロファイル（ISO 標準コード生成の場合のみ）の中に、SQLJ ソース・コード中のすべての埋込み SQL 文に関する情報が格納されます。実行するアクション、操作するデータ型、アクセスする表などが指定されています。アプリケーションを実行すると、SQLJ ランタイムはプロファイルにアクセスし、SQL 操作を取り出し、JDBC ドライバに渡します。

プロファイルがある場合は、デフォルトでシリアル化リソース・ファイル（.ser）に出力されますが、オプションで、変換時に .ser ファイルを .class ファイルに変換できます。

5. JVM が Java コンパイラを起動します。通常、このコンパイラは Sun 社の JDK で提供される標準 javac です。
6. 手順 4 で生成された Java ソース・ファイルがコンパイルされ、Java の .class ファイルが生成されます。定義した各クラスの .class ファイル、各 SQLJ 宣言の .class ファイル、プロファイルキー・クラスの .class ファイル（ISO 標準コード生成の場合）などが生成されます。
7. ISO 標準 SQLJ コード生成の場合は、JVM で Oracle SQLJ カスタマイザまたは他に指定したカスタマイザを起動して、手順 4 で生成したプロファイルのカスタマイズします。

SQLJ の一般的な注意 SQLJ アプリケーションを変換および実行する場合は、次の点を考慮してください。

- 前述は汎用的な例です。既存の .java ファイルをコンパイル（および型解決）するとき、コマンドラインで指定できます。また、カスタマイズする既存のプロファイルや、カスタマイズするプロファイルが格納されている .jar ファイルも指定できます。詳細は、8-2 ページの「[トランスレータのコマンドラインとプロパティ・ファイル](#)」を参照してください。
- Oracle 固有コード生成の場合、アプリケーションの実行時に Oracle JDBC ドライバが必要になります。これは、コードで Oracle 固有の機能を使用しない場合にも必要です。
- ISO 標準コード生成では、ソース・コードに SQLJ 実行文がある場合のみ、SQLJ によって、プロファイルおよびプロファイルキー・クラスが生成されます。
- また、ISO 標準コードで、Oracle カスタマイザを変換時に使用する場合、アプリケーションでは、実行時に Oracle SQLJ ランタイムおよび Oracle JDBC ドライバが必要になります。これらは、コードで Oracle 固有の機能を使用しない場合にも必要です。これを回避するには、変換時に `-profile=false` を指定することによって、Oracle 固有のカスタマイズをバイパスできます。

トランスレータ入出力の概要

ここでは、SQLJ トランスレータへの入力として扱われるもの、出力となるものおよびその出力先について、概要を説明します。

注意： この説明では、イテレータ・クラスと接続コンテキスト・クラスの宣言を扱います。イテレータは JDBC の結果セットに似ています。接続コンテキストはデータベース接続で使用します。これらのクラス宣言の詳細は、3-2 ページの「[SQLJ 宣言の概要](#)」を参照してください。

トランスレータへの入力

SQLJ トランスレータの最も基本的な操作では、入力として1つ以上の .sqlj ソース・ファイルをコマンドラインで指定します。メインの .sqlj ファイルの名前は、定義されている Public クラスまたは定義されている最初のクラス（ある場合）に基づきます。各 Public クラスは、.sqlj ファイル中に定義する必要があります。

メインの .sqlj ファイルでクラス MyClass を定義した場合は、ソース・ファイル名を次の名前にします。

MyClass.sqlj

Public クラスが未定義でも、最初に定義したクラスの名前が MyClass であれば、ソース・ファイル名をこの名前にする必要があります。

SQLJ を実行するときは、複数の SQLJ オプションをコマンドラインでもプロパティ・ファイルでも指定できます。

コマンドラインで指定できる追加ファイル・タイプなど、SQLJ 入力の詳細は、8-2 ページの「[トランスレータのコマンドラインとプロパティ・ファイル](#)」を参照してください。

トランスレータの出力

変換処理では、アプリケーション内の .sqlj ファイルごとに Java ソース・ファイルが生成されます。また、ISO 標準コード生成の場合、アプリケーションのプロファイルは、1つ以上生成されます（ソース・コードに SQLJ 実行文があるとき）。

SQLJ では、次のソース・ファイルとプロファイルが生成されます。

- Java ソース・ファイル。 .sqlj ファイルと同じベース名の .java ファイルです。

たとえば、MyClass.sqlj でクラス MyClass を定義すると、トランスレータによって MyClass.java が生成されます。出力 .java ファイルには、宣言したイテレータまたは接続コンテキスト・クラスのクラス定義も含まれています。

- アプリケーションのプロファイル・ファイル（ある場合）には、SQLJ アプリケーションの SQL 操作に関する情報が入ります。プロファイルは、アプリケーションで使用する接続クラスごとに1つ生成されます。プロファイルの名前は、メインの .sqlj ファイルと同じベース名になり、次の拡張子が付きます。

```
_SJProfile0.ser  
_SJProfile1.ser  
_SJProfile2.ser  
...
```

たとえば、MyClass.sqlj の場合は、トランスレータによって次のプロファイル・ファイルが生成されます。

```
MyClass_SJProfile0.ser
```

.ser 拡張子は、プロファイルがシリアル化されていることを示します。.ser ファイルはバイナリ・ファイルです。

注意： トランスレータ・オプション `-ser2class` を指定すると、トランスレータによって生成されるプロファイル・ファイルが .ser ファイルではなく .class ファイルになります。ファイル名のベース名は同じになりますが、拡張子が異なります。

コンパイル処理によって、Java ソース・ファイルが複数のクラス・ファイルにコンパイルされます。.sqlj ソース・ファイル中に定義した各クラス（少なくとも1つ）ごとに、また、ISO 標準コードの場合は、トランスレータが生成して SQL 操作を実装するために（ソース・コード中に SQLJ 実行文がある場合）プロファイルと一緒に使用されるプロファイルキー・クラスとも呼ばれるクラスごとに、1つの .class ファイルがあります。SQLJ のイテレータまたは接続コンテキストを宣言した場合は、さらに .class ファイルが生成されます。（3-2 ページの「[SQLJ 宣言の概要](#)」を参照してください。）また、コード中に内部クラスまたは無名クラスが宣言されていれば、それらの各クラスごとに .class ファイルが個別に生成されます。

Oracle 固有コード生成の場合（これがデフォルトです）、プロファイルまたはプロファイルキー・クラスは生成されません。Oracle 固有コード生成の詳細は、4-36 ページの「[Oracle 固有コード生成（プロファイルなし）](#)」を参照してください。

これらの .class ファイルには、次の名前が付けられます。

- 定義する各クラスのクラス・ファイル名は、クラス名および .class 拡張子から構成されます。

たとえば、トランスレータ出力ファイル MyClass.java は、MyClass.class クラス・ファイルにコンパイルされます。

- トランスレータで生成されたプロファイルキー・クラスがある場合は、メインの .sqlj ファイルのベース名と次の文字列から構成される名前が付けられます。

```
_SJProfileKeys
```

つまり、クラス・ファイルの拡張子は、次のような名前になります。

```
_SJProfileKeys.class
```

たとえば、MyClass.sqlj の場合は、トランスレータとコンパイラによって次のクラス・ファイルが生成されます。

```
MyClass_SJProfileKeys.class
```

- トランスレータを実行すると、イテレータ・クラスと接続コンテキスト・クラスに、宣言内容に応じた名前が付けられます。たとえば、イテレータ `MyIter` を宣言すると、`MyIter.class` クラス・ファイルが生成されます。

カスタマイズ処理によってプロファイルが変更されますが、追加出力はありません。

注意： SQLJ プロファイルまたはプロファイルキー・クラスは、必ずしも直接参照する必要はありません。これらは、すべて自動的に処理されます。

ファイルの出力先

デフォルトでは、生成された .java ファイルが .sqlj ファイルと同じディレクトリに出力されます。.java ファイルの出力先を変更するには、SQLJ の `-dir` オプションを使用します。

SQLJ のデフォルトの設定では、生成済み .class ファイルと .ser ファイル（ある場合は、生成済み .java ファイルと同じディレクトリに出力されます。.class ファイルと .ser ファイルの出力先を変更するには、SQLJ の `-d` オプションを使用します。このオプションの設定値が Java コンパイラに渡され、.class ファイルと .ser ファイルが同じディレクトリに出力されます。

`-d` または `-dir` オプションを使用するときは、既存のディレクトリを指定する必要があります。これらのオプションの詳細は、8-28 ページの「[出力ファイルとディレクトリのオプション](#)」を参照してください。

SQLJ のランタイム処理

ここでは、Oracle 固有コード生成および ISO 標準 SQLJ コード生成の両方について、プログラム実行中のランタイム処理を説明します。

Oracle 固有生成コードの処理

デフォルト設定 `-codegen=oracle` で変換する場合、プログラムでは実行時に次の内容が実行されます。

- SQLJ ランタイム内の Oracle 固有 API によって、Oracle JDBC 文のバッチ処理サポート、作成および終了を確実に実行します。
- Oracle JDBC API を直接コールして、パラメータおよび結果セットの登録、受渡しを実行します。

Oracle 固有コード生成の詳細は、4-36 ページの「[Oracle 固有コード生成（プロファイルなし）](#)」を参照してください。

ISO 標準生成コードの処理

ISO 標準 SQLJ アプリケーションの場合、SQLJ ランタイムがプロファイルを読み込み、接続プロファイルを作成します。このプロファイルには、データベース接続情報が組み込まれています。アプリケーションでデータベースへのアクセスが必要になると、そのつど次に示す処理が展開されます。

1. SQLJ で生成したプロファイルキー・クラスのメソッドを使用して、SQLJ で生成したアプリケーション・コードから接続プロファイルにアクセスし、該当する SQL 操作を読み込みます。アプリケーション内の SQLJ 実行文とプロファイル内の SQL 操作はマッピングされています。
2. SQLJ で生成したアプリケーション・コードから SQLJ ランタイムをコールします。SQL 操作がプロファイルから読み込まれます。
3. SQLJ ランタイムが JDBC ドライバをコールし、SQL 操作をドライバに渡します。
4. SQLJ ランタイムが入力パラメータを JDBC ドライバに渡します。
5. JDBC ドライバが SQL 操作を実行します。
6. 戻り値がある場合は、データベースがデータを JDBC ドライバに送ります。このデータはドライバから SQLJ ランタイムに送られ、アプリケーションで使用できるようになります。

注意： 入力パラメータを渡す（手順 4）ことは、「入力パラメータをバインドする」または「ホスト式をバインドする」とも言います。ホスト変数、ホスト式、バインド変数およびバインド式は、SQL 操作の入出力として使用する Java 変数や式の記述に使用します。

JDBC と SQLJ のサンプル・コード

ここでは、同じ内容のサンプルについて 2 つのバージョン（それぞれ JDBC、SQLJ で記述されたバージョン）を用意し、比較対照します。ここでの目的は、SQLJ と JDBC のコード化での要件の違いを示すことです。

ここで使用する SQLJ 文と機能の詳細は、このマニュアルの後の方で説明します。しかし、ここにサンプルを掲載することは、SQLJ と JDBC を比較対照する上での一般概念を示すために有用です。SQLJ の概念と機能をよく理解したときに、再度このサンプルを確認してもかまいません。

このサンプルでは、次のように 2 つのメソッドを定義します。getEmployeeAddress() は、従業員番号に基づいて表データを取り出し、その番号を持つ従業員の住所を戻します。updateAddress() は、取り出されたアドレスを受け取ってストアド・プロシージャをコールし、更新された住所をデータベースに戻します。

どちらのバージョンのコードも、次の事項を前提としています。

- SQL スクリプト（ここには示していません）が実行され、データベース内にスキーマが作成され、表にデータが取り込まれていること。いずれのバージョンのサンプル・コードも、このスクリプトによって生成されたオブジェクトおよび表を参照しています。
- 指定された住所を更新する PL/SQL ストアド・ファンクション UPDATE_ADDRESS() が存在すること。
- Connection オブジェクト（JDBC 用）およびデフォルトの接続コンテキスト（SQLJ 用）が、コール元によって生成されていること。
- コール元によって、例外処理が行われること。
- updateAddress() メソッドに渡される住所の引数（addr）に、NULL 値が許容されていること。

注意： サンプル・コードの JDBC と SQLJ のバージョンは、単なる部分的なサンプルで、単独では実行できません。いずれのサンプルにも、main() メソッドはありません。

JDBC バージョンのサンプル・コード

次は、JDBC バージョンのサンプル・コードです。このコードには、データベースから従業員の住所を取り出し、それを更新し、更新後の住所をデータベースに戻すメソッドが定義されています。必要に応じて、TO DO で始まるコメント行にコードを追加すると、このサンプル・コードの有用性が高くなります。

```
import java.sql.*;
import oracle.jdbc.*;

/**
 * This is what we have to do in JDBC
 */
public class SimpleDemoJDBC                                // line 7
{

    //TO DO: make a main that calls this

    public Address getEmployeeAddress(int empno, Connection conn)
        throws SQLException                                // line 13
    {
        Address addr;
        PreparedStatement pstmt =                            // line 16
            conn.prepareStatement("SELECT office_addr FROM employees" +
                " WHERE empnumber = ?");
        pstmt.setInt(1, empno);
        OracleResultSet rs = (OracleResultSet)pstmt.executeQuery();
        rs.next();                                           // line 21
        //TO DO: what if false (result set contains no data)?
        addr = (Address)rs.getORADData(1, Address.getORADDataFactory());
        //TO DO: what if additional rows?
        rs.close();                                          // line 25
        pstmt.close();
        return addr;                                         // line 27
    }

    public Address updateAddress(Address addr, Connection conn)
        throws SQLException                                // line 30
    {
        OracleCallableStatement cstmt = (OracleCallableStatement)
            conn.prepareCall("{ ? = call UPDATE_ADDRESS(?) }"); //line 34
        cstmt.registerOutParameter(1, Address._SQL_TYPECODE, Address._SQL_NAME);
                                                                    // line 36

        if (addr == null) {
            cstmt.setNull(2, Address._SQL_TYPECODE, Address._SQL_NAME);
        } else {
            cstmt.setORADData(2, addr);
        }
    }
}
```

```
    }  
  
    pstmt.executeUpdate(); // line 43  
    addr = (Address)pstmt.getORADData(1, Address.getORADDataFactory());  
    pstmt.close(); // line 45  
    return addr;  
  }  
}
```

12 行目： `getEmployeeAddress()` メソッドの定義では、メソッド定義に `Connection` オブジェクトを明示的に渡す必要があります。

16 ～ 20 行目： `EMPLOYEES` 表から従業員番号に基づいて従業員の住所を取り出す文を作成します。従業員番号は、`setInt()` メソッドで設定したマーカー変数で示されます。この `PreparedStatement` では、`INTO` 構文が認識されないの、住所 (`addr`) 変数に値を設定するコードを自分で作成する必要があります。この `PreparedStatement` はカスタム・オブジェクトを戻り値とするので、この出力を `OracleResultSet` にキャストしてください。

21 ～ 23 行目： `OracleResultSet` には、`Address` 型のカスタム・オブジェクトがあるため、`getORADData()` メソッドを使用して、そのオブジェクトを取得します。`Address` クラスは、`JPublisher` で作成できます。`getORADData()` メソッドには、「ファクトリ」オブジェクトが必要です。このファクトリ・オブジェクトは、カスタム・オブジェクト（この場合は `Address` オブジェクト）の追加作成や移入に使用されます。`getORADData()` メソッドで使用する `Address` ファクトリ・オブジェクトを作成するには、`static` ファクトリ・メソッド `Address.getORADDataFactory()` を使用します。

`getORADData()` は `Datum` を戻り値とするため、その出力を `Address` オブジェクトにキャストする必要があります。

このルーチンでは、結果セットが 1 行であることを前提としています。結果セットに行が 1 つも含まれないか、複数の行が含まれる場合は、`TO DO` で始まるコメント行にコードを追加する必要があります。

25 ～ 27 行目： 結果セットおよび `PreparedStatement` オブジェクトを終了し、その後で `addr` 変数を戻します。

29 行目： `updateAddress()` の定義で、`Connection` オブジェクトと `Address` オブジェクトを明示的に渡す必要があります。

`updateAddress()` メソッドは、住所のオブジェクト (`Address`) をデータベースに渡し、住所を更新した後、同じオブジェクトをフェッチします。実際の住所更新処理は、`UPDATE_ADDRESS()` ストアド・ファンクションによって実行されます。（このファンクションのコードは、このサンプルには含まれていません。）

33 ～ 43 行目： OracleCallableStatement オブジェクトを生成し、住所のオブジェクト (Address) を受け取り、UPDATE_ADDRESS() ストアド・プロシージャに渡します。オブジェクトを出力パラメータとして登録するには、そのオブジェクトの SQL 型コードと SQL 型名を知っている必要があります。

プログラムは、住所のオブジェクト (addr) の値が NULL かどうかを確認した後で、addr を入力パラメータとして渡します。プログラムは、addr の値によって、異なる設定メソッドをコールします。addr の値が NULL の場合、setNull() をコールします。addr の値が NULL 以外の場合、setORAData() をコールします。

44 行目： 戻り値 addr をフェッチします。Oracle コール可能文は Address 型のカスタム・オブジェクトを戻すため、getORAData() メソッドを使用して、そのオブジェクトを取得します。Address クラスは、JPublisher で作成できます。getORAData() メソッドでは、ファクトリ・メソッド Address.getORADataFactory を使用して、Address オブジェクトのインスタンスを生成する必要があります。getORAData() は Datum オブジェクトを戻り値とするため、その出力を Address オブジェクトにキャストする必要があります。

45、46 行目： OracleCallableStatement オブジェクトを終了した後で、addr 変数が戻されます。

JDBC バージョンのコード化での要件

次は、JDBC バージョンのサンプル・コードのコード化での要件です。

- getEmployeeAddress() および updateAddress() の定義に、接続オブジェクトを明示的に含めること。
- 長い SQL 文字列は、SQL 連結文字 (「+」) で連結すること。
- 明示的にリソースを管理すること。たとえば、ResultSet オブジェクトや Statement オブジェクトを終了する必要があります。
- 必要に応じて、データ型をキャストすること。
- 出力パラメータとして登録するオブジェクトとファクトリ・オブジェクトの _SQL_ TYPECODE 値および _SQL_ NAME 値に関する知識があること。
- NULL データを明示的に処理すること。
- CallableStatement および PreparedStatement のホスト変数に、パラメータ・マーカを付けること。
- Statement オブジェクトを再利用する場合 (たとえば、getEmployeeAddress() や updateAddress() を幾度もコールする場合) には、このコードを正しく記述すること。Oracle SQLJ および Oracle JDBC は、両方とも文キャッシングをサポートしています。

JDBC プログラムの管理

JDBC プログラムの場合、管理に手間がかかることがあります。たとえば、前述のサンプル・コードに別の WHERE 句を追加した場合は、SELECT 文字列を変更する必要があります。新しいホスト変数を追加した場合は、他のホスト変数の索引を 1 ずつ増やす必要があります。JDBC プログラムの変更箇所が 1 行の場合でも、プログラム内で他のいくつかの箇所を変更する必要が生じることもあります。

SQLJ バージョンのサンプル・コード

次は、SQLJ バージョンのサンプル・コードです。このコードには、データベースから従業員の住所を取り出し、それを更新し、更新後の住所をデータベースに戻すメソッドが定義されています。

```
import java.sql.*;

/**
 * This is what we have to do in SQLJ
 */
public class SimpleDemoSQLJ                                // line 6
{
    //TO DO: make a main that calls this

    public Address getEmployeeAddress(int empno)            // line 10
        throws SQLException
    {
        Address addr;                                       // line 13
        #sql { SELECT office_addr INTO :addr FROM employees
              WHERE empnumber = :empno };
        return addr;
    }                                                         // line 18

    public Address updateAddress(Address addr)
        throws SQLException
    {
        #sql addr = { VALUES(UPDATE_ADDRESS(:addr)) };    // line 22
        return addr;
    }
}
```

10 行目：getEmployeeAddress() メソッドは、明示的な Connection オブジェクトを必要としません。SQLJ では、すでにアプリケーションで初期化されているデフォルトの接続コンテキスト・インスタンスを使用できます。

13 ～ 15 行目： `getEmployeeAddress()` メソッドは、従業員番号に基づいて従業員の住所を取り出します。標準の SQLJ `SELECT INTO` 構文を使用して、`employee` 表から従業員のアドレスを選択します。その場合、`getEmployeeAddress()` に渡された従業員番号 (`empno`) と一致した従業員番号を持つ従業員の住所が選択されます。この処理には、データを受け取る `Address` オブジェクト (`addr`) を宣言する必要があります。`empno` および `addr` の各変数は、入力ホスト変数として使用されます。

16 行目： `getEmployeeAddress()` メソッドは、`addr` オブジェクトを戻り値とします。

19 行目： `updateAddress()` メソッドでも、デフォルトの接続コンテキスト・インスタンスが使用されます。

19 ～ 22 行目： 取り出された住所は `updateAddress()` メソッドに渡され、このメソッドからデータベースに渡されます。データベースでその住所が更新され、更新されたデータが戻されます。実際の住所更新処理は、`UPDATE_ADDRESS()` ストアド・ファンクションによって実行されます。(このファンクションのコードは、このサンプルには示してありません。) 標準の SQLJ ファンクション・コール構文を使用して、`UPDATE_ADDRESS()` から出力されたオブジェクト (`addr`) を取り出します。

23 行目： `updateAddress()` メソッドは、`addr` オブジェクトを戻り値とします。

SQLJ パージョンのコード化での要件

SQLJ パージョンのサンプル・コードのコード化に関して、要件の有無を次に示します。

- 明示的な接続が必要ないこと。SQLJ では、アプリケーションですでに初期化されているデフォルトの接続コンテキストを使用できます。
- データ型のキャストは不要です。
- SQLJ では、`_SQL_TYPECODE`、`_SQL_NAME` やファクトリ・オブジェクトの知識は不要です。
- `NULL` データの暗黙的な処理は必要です。
- リソースの明示的な管理 (たとえば、`Statement` オブジェクトや `ResultSet` オブジェクトの終了) は不要です。
- SQLJ ではホスト変数の埋込みが必要です。これに対し、JDBC ではパラメータ・マークが使用されます。
- 長い SQL 文での文字列の連結は不要です。
- 出力パラメータの登録は不要です。
- SQLJ 構文はさらに簡略化されています。具体的には、SQLJ では `SELECT INTO` 文はサポートされていますが、OBDC 形式のエスケープは使用されません。

- 文のキャッシュは、自分で実装する必要はありません。デフォルトで、SQLJ では自動的に #sql 文がキャッシュされます。これによって、getEmployeeAddress() および updateAddress() を繰り返してコールする場合などのパフォーマンスが改善します。

他の配置例

このマニュアルでは、主にクライアント側 SQLJ アプリケーションについて説明しています。しかし、次のような例においても SQLJ コードを実行すると便利な場合があります。

- アプレットから
- サーバー内で (SQLJ トランスレータもサーバーで実行できます)
- Oracle9i Lite に対して

アプレットでの SQLJ の実行

SQLJ ランタイムは Pure Java であるため、アプレットでもアプリケーションでも SQLJ ソース・コードを使用できます。ただし、次に示すような条件があります。

より一般的に Oracle JDBC ドライバに適用するアプレットに関しては、ファイアウォールとセキュリティについても説明している『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。

開発および配布に関する一般的な考慮事項

Oracle SQLJ アプレットを使用する際の一般的な考慮事項を、次に示します。

- 次の SQLJ ランタイム・パッケージはすべて、アプレットと一緒にパッケージ化する必要があります。

```
sqlj.runtime
sqlj.runtime.ref
sqlj.runtime.profile
sqlj.runtime.profile.ref
sqlj.runtime.error
```

(ISO コード生成に対して) Oracle 独自のカスタマイズを行った場合は、次のパッケージも含める必要があります。

```
oracle.sqlj.runtime
oracle.sqlj.runtime.error
```

これらのクラスは、Oracle インストールによって、\$Oracle_Home/sqlj/lib ディレクトリのランタイム・ライブラリの中の 1 つに組み込まれています。(2-3 ページの「Oracle SQLJ を使用するための要件」を参照してください。)

- データベース接続用のドライバとして、Oracle JDBC Thin ドライバなどの Pure Java JDBC ドライバを指定する必要があります。
- アプレット内の SQLJ 実行文ごとに、接続コンテキストのインスタンスを明示的に指定する必要があります。2つの SQLJ アプレットを1つのブラウザで実行した場合、同じ JVM で実行されるのを防ぐために、必ず明示的に指定してください。（接続の詳細は、4-6 ページの「[接続の際の考慮事項](#)」を参照してください。）
- デフォルトのトランスレータ設定 `-codegen=oracle` によって、Oracle 固有コードが生成されます。これによって、実行時の Java リフレクションの使用が削減されるので、異なるブラウザ環境間での移植性が向上します。`-codegen` オプションの詳細は、8-50 ページの「[コードの生成 \(-codegen\)](#)」を参照してください。Oracle 固有コード生成の詳細は、4-36 ページの「[Oracle 固有コード生成 \(プロファイルなし\)](#)」を参照してください。

エンド・ユーザーのための一般的な考慮事項

エンド・ユーザーが SQLJ アプレットを実行したときに、CLASSPATH 内のクラスが、アプレットと一緒にダウンロードされたクラスと競合する可能性があります。

これを回避するためには、アプレットを実行する前に、エンド・ユーザー側で CLASSPATH を削除することをお勧めします。

Java 環境および Java プラグイン

Java 環境および Oracle 固有の機能の使用に関しては、追加で考慮する事項を次に示します。

- SQLJ のランタイム環境として、JDK 1.1.x 以上が必要です。Netscape Navigator 3.0 や Microsoft Internet Explorer 3.x など、JDK 1.0.x を採用するブラウザで SQLJ を実行するには、ブラウザのデフォルト JRE でなく JRE 1.1.x を（プラグインを使用するといった方法で）使用する必要があります。

この対処としては、Sun 社のプラグインを使用する方法があります。詳細は次の Web サイトを参照してください。

<http://www.javasoft.com/products/plugin>

- Netscape Navigator 4.x などの一部のブラウザでは、プロファイル用の SQLJ シリアル化オブジェクト・ファイルで採用される、`.ser` 拡張子の付いたリソース・ファイルがサポートされていません（ISO 標準コードの場合にのみ関係します）。この `.ser` ファイルは、Sun 社の Java プラグインではサポートされています。

Oracle SQLJ では、このプラグインを使用する以外の方法として、`.ser` ファイルを `.class` ファイルに変換するための `-ser2class` オプションを使用する方法があります。詳細は、8-64 ページの「[.ser ファイルから .class ファイルへの変換 \(-ser2class\)](#)」を参照してください。

注意： これらの考慮事項は、プロファイルが生成されない Oracle 固有コード生成（これがデフォルトです）には適用されません。

- Oracle 固有の機能を使用するアプレットでは、Oracle SQLJ ランタイムを起動する必要があります。Oracle ランタイムは、`oracle.sqlj.*` 下の SQLJ ランタイム・ライブラリ・ファイル内のクラスで構成されます。Oracle SQLJ ランタイム・ライブラリでは、Java Reflection API (`java.lang.reflect.*`) が必要です。`runtime11`、`runtime12` および `runtime12ee` ランタイム・ライブラリでは、次のような環境でのみ Reflection API を使用する必要があります。大半のブラウザでは、Reflection API をサポートしていなかったりセキュリティ上の制約があるのに対し、Sun 社の Java プラグインの場合は Reflection API に対するサポートが提供されています。

注意： Oracle 固有の機能とは、Oracle 拡張型（第 5 章「型のサポート」を参照）を使用すること、および Oracle 固有コード生成を必要とする（または ISO 標準コード生成の場合は Oracle データベースと連携できるようにカスタマイズされたアプリケーションを必要とする）SQLJ 機能を使用することを意味します。（たとえば、第 3 章「基本的な言語機能」で説明されている SET 文がこれに該当します。）

ISO 標準 SQLJ コード生成の場合、次の SQLJ 言語の機能では、使用している SQLJ ランタイムのバージョンに関係なく、常に Java Reflection API (`java.lang.reflect.*`) が必要となります。

- CAST 文
- SQLJ イテレータのインスタンスとしてのデータベースから取得される REF CURSOR パラメータまたは REF CURSOR 列
- `java.sql.Ref`、`Struct`、`Blob` または `Clob` オブジェクトの取得
- `oracle.sql.ORAData` または `java.sql.SQLData` インタフェースを実装する Java クラスのインスタンスとしての SQL オブジェクトの取得

注意：

- ISO と完全に互換性のあるモードで SQLJ を使用する場合、前述の例外があります。それは、SQLJ を J2EE でコンパイルする環境で使用する場合、SQLJ runtime12ee ライブラリでプログラムを変換および実行する場合、および ISO で指定したように接続コンテキスト型を採用する場合です。この場合、`java.sql.Ref`、`Struct`、`Blob`、`Clob` および `SQLData` のインスタンスは、リフレクションを使用せずに取得されます。
 - Oracle 固有コード生成（デフォルトのトランスレータ設定 `-codegen=oracle`）を使用すると、前述のすべてのインスタンスでのリフレクションの使用を削減します。
-

- アプレットに対して runtime11 ライブラリまたは runtime12/runtime12ee（ブラウザが JDK 1.2 をサポートしている場合）を使用するとします。そうすると、Oracle 固有の機能および Oracle 固有のカスタマイズを使用できます。
- アプレットが Oracle 固有の機能を何も使用していない場合は、このアプレットを汎用 SQLJ ランタイム・ライブラリ `runtime-nonoracle` とともに配布できます。これをサポートするには、Oracle 固有コード生成を使用せず、変換時にアプレットをカスタマイズしないでください。コードを変換するときの設定は、`-codegen=iso` および `-profile=false` です。この `-profile=false` という設定を省略した場合は、デフォルトの Oracle カスタマイズによって Oracle 固有のランタイム・クラスがロードされます。その結果、Oracle 固有の機能を使用していない場合でも、アプレットでは Oracle ランタイムが必要となります。

これまで、Internet Explorer や Netscape のブラウザに重点を当てて説明してきました。その要約を次に示します。

- アプレットは、runtime11 および classes111 ライブラリと一緒に配布します。この場合、SQLJ と JDBC のバージョンは一致している必要があります。たとえば、リリース 9.0.1 の SQLJ ランタイムを使用するには、リリース 9.0.1 の JDBC ドライバが必要です。
 - SQLJ 文中にオブジェクト型、JDBC 2.0 型、REF CURSOR または CAST 文を使用する場合は、次のいずれかに適合している必要があります。
 - アプレットを変換するときに、デフォルトの `-codegen=oracle` 設定を使用します。
- または
- 起動するブラウザでは、JDK 1.1 以上をサポートしていて、リフレクションが使用できることを確認します。

または

- ブラウザの Java プラグインを介してアプレットを実行します。
- アプレットが Oracle 固有の機能を使用していない場合は、カスタマイズをせずに (-profile=false)、ISO 標準コード生成を使用 (-codegen=iso) してコンパイルし、汎用 SQLJ ランタイム runtime-nonoracle とともに配布できます。

サーバー側 SQLJ について

SQLJ コードは、クライアント・アプリケーションで使用できるばかりでなく、ターゲットの Oracle9i データベースのストアド・プロシージャ、ストアド・ファンクションまたはトリガーでも実行できます。サーバー側のアクセスは、サーバー内部で実行される Oracle JDBC ドライバを介して発生します。また、Oracle9i データベースには埋込みの SQLJ トランスレータがあるため、サーバー側で使用する SQLJ ソース・ファイルをサーバーで直接変換できます。

次の 2 点については考慮が必要です。詳細は、[第 11 章「サーバー側 SQLJ」](#)を参照してください。

- サーバー側で使用する SQLJ コードの作成

ターゲットの Oracle9i データベース側で使用する SQLJ アプリケーションのコードの記述方法は、クライアント側で使用するコードの記述方法と類似しています。主な相違点は、JDBC の全般的な特性によるものであり、SQLJ 固有の特性によるものではありません。大きな相違点は、次のように接続に関するものです。

- 接続数が 1 つに限定されること。
- コードが実行されるデータベースに接続すること。
- 接続が暗黙的に行われること（クライアント側とは異なり、明示的に初期化する必要がありません）。
- 接続をクローズできないこと。クローズを試みても無視されます。

この他、サーバー内で接続に使用する JDBC サーバー側ドライバは、自動コミット・モードをサポートしていないことも挙げられます。

注意： サーバー側 Thin ドライバの一部には、あるサーバー内のコードから別のサーバーへ接続できるものがあります。この場合、クライアントの Thin ドライバを使用した場合と結果は同様に、しかもコードの記述方法も同様です。4-2 ページの「[Oracle JDBC ドライバの概要](#)」を参照してください。

■ サーバー側 SQLJ コードの変換とロード

コードの変換とコンパイルは、クライアントでもサーバーでも行えます。クライアント側で行った場合は、クラス・ファイルとリソース・ファイルをクライアント・マシンからサーバーにロードできます。Oracle の loadjava ユーティリティを使用してクライアントからファイルをロードするか、または SQL コマンドを使用してサーバーの管理するファイルをロードします。(最初にすべてのファイルを 1 つの .jar ファイルにまとめておくくと便利です。)

また、サーバー側の埋込み SQLJ トランスレータを使用して変換とロードを 1 ステップで行う方法もあります。クラス・ファイルまたはリソース・ファイルのかわりに SQLJ ソース・ファイルをロードすると、変換とコンパイルは自動的に実行されます。通常、loadjava または SQL コマンドは、クラス・ファイルとリソース・ファイルに対して使用することも、ソース・ファイルに対して使用することも可能です。ユーザーの観点からは、.sqlj ファイルは .java ファイルと同様に扱われ、変換は暗黙的に実行されます。

サーバー側の埋込みトランスレータの使用方法は、11-14 ページの「[SQLJ ソースのロードとサーバーでの変換](#)」を参照してください。

注意： サーバー側トランスレータでは、Oracle SQLJ -codegen オプションおよび ISO 標準コード生成をサポートしていません。ISO 標準コードをサーバーで使用するには、クライアント上で変換して、個々のコンポーネントをサーバーにロードする必要があります。また、異なる設定で生成されたコードを実行するときには、相互運用性に関する制限事項に留意してください。詳細は、11-8 ページの「[クライアント側での SQLJ ソースの変換とコンポーネントのロード](#)」および 4-36 ページの「[Oracle 固有コード生成 \(プロファイルなし\)](#)」を参照してください。

Oracle9i Lite での SQLJ の使用

Oracle9i Lite に対しても SQLJ を使用できます。ここでは、この機能の概要を説明します。詳細は、『Oracle9i Lite Java 開発者ガイド』を参照してください。

Oracle9i Lite と Java サポートの概要

Oracle9i Lite は軽量データベースであり、大型データベースにはない融通性と多様性が特長です。必要なメモリー容量は全機能で 350 ～ 750KB であり、Palm Computing プラットフォームとネイティブに同期し、Windows NT4.0 (SP3 以上)、Windows2000、Windows 95 および Windows 98 で実行できます。埋込み環境が用意されているため、バックグラウンド・プロセスもサーバー・プロセスも不要です。

Oracle9i Lite は、Oracle9i、Oracle8i、Oracle8 および Oracle7 と互換性があります。JDBC、SQLJ、Java のストアド・プロシージャなど、Java を包括的にサポートします。次の 2 通りの方法で、Java プログラムから Oracle9i Lite にアクセスできます。

- ネイティブ JDBC ドライバ

JDBC ドライバは、リレーショナル・データ・モデルを使用する Java アプリケーション用のドライバです。アプリケーションでは、このドライバを介して、オブジェクト・リレーショナル・データベース・エンジンと直接通信できます。

プログラムで SQL 形式のデータにアクセスする場合、他のリレーショナル・データベース・システムに対してプログラムを実行する場合、またはプログラムで複雑な問合せを使用する場合は、リレーショナル・データ・モデルを使用します。

- Java Access Class (JAC)

Java オブジェクト・モデルまたは Oracle9i Lite オブジェクト・モデルを使用する Java アプリケーション用のクラスです。アプリケーションでこのクラスを使用すると、オブジェクト・モデルとリレーショナル・モデル間をマッピングしなくても、Oracle9i Lite に格納されている永続情報にアクセスできます。JAC を使用するときは、Java の永続プロキシ・クラスを使用して、Oracle9i Lite スキーマをモデル化する必要があります。このモデルは Oracle9i Lite ツールで生成できます。

プログラムのフットプリントを小型化して高速で実行する場合で、SQL 言語の全機能を使用する必要がない場合は、オブジェクト・モデルを使用します。

Oracle9i Lite JDBC と JAC は関係します。JAC は、JDBC がサポートするすべての型をサポートしますが、JDBC は特定の要件を満たす JAC の型のみをサポートします。

Oracle9i Lite で Java を実行するときの条件

Oracle9i Lite 上で Java プログラムを実行する場合は、次の環境が必要です。

- Windows NT4.0 SP 3 以上、Windows2000、Windows 95 または Windows 98
- Oracle9i Lite リリース 3.0 以上
- JDK 1.2 以上
- Java Native Interface (JNI) をサポートする Java Runtime Environment (JRE)

JDK 1.2 以上、Oracle JDeveloper で提供される JRE では、JNI がサポートされています。

Oracle 拡張型のサポート

リリース 4.0.x 以上の Oracle9i Lite には、Oracle 固有の JDBC ドライバと SQLJ ランタイム・クラス (Oracle のセマンティクス・チェックやカスタマイズなど) があるため、Oracle 固有の機能と拡張型を使用できます。

他の開発環境

このマニュアルの説明は、UNIX 環境で英語版コードを手動で作成する場合を想定しています。ただし、SQLJ は他のプラットフォームおよび IDE でも使用できます。他言語版のグローバリゼーション・サポートもあります。ここでは次の内容について説明します。

- グローバリゼーション・サポート
- IDE での SQLJ
- Windows に関する注意事項

SQLJ グローバリゼーション・サポート

Oracle SQLJ は、ネイティブ言語をサポートします。文字のエンコードは Java の組み込みグローバリゼーション・サポート機能に基づいて行われます。

トランスレータ・メッセージおよびランタイム・メッセージ用の言語とエンコーディング方法は、JVM 標準の `user.language` および `file.encoding` プロパティによって、特定されます。変換時にソース・ファイルを解析および生成するためのエンコーディングは、SQLJ の `-encoding` オプションで指定します。

詳細は、9-18 ページの「[トランスレータおよびランタイムでのグローバリゼーション・サポート](#)」を参照してください。

Oracle9i JDeveloper などの IDE での SQLJ の使用

Oracle SQLJ は、プログラム API、Oracle JDeveloper などの統合開発環境（IDE）への埋込みができます。IDE は、フロントエンドの `sqlj` スクリプトと同じように機能し、トランスレータ、セマンティクス・チェッカ、コンパイラおよびカスタマイザ（使用する場合）を起動します。

JDeveloper は、Java プログラミング用のビジュアル開発環境です。開発者は JDeveloper で、スケーラブルな複数層インターネット・アプリケーションを構築し、Oracle Internet Platform 全体で Java を使用できます。この製品（JDeveloper Integrated Development Environment）では、コンポーネントベースのアプリケーションを作成、デバッグおよび実行できます。

Oracle JDBC OCI ドライバと Thin ドライバは、Oracle9i Lite へのアクセス用ドライバとともに、JDeveloper に同梱されています。

JDeveloper のコンパイル機能には、Oracle SQLJ トランスレータが組み込まれています。SQLJ アプリケーションは、コンパイル時に自動的に変換されます。

JDeveloper に関する情報は、次の URL から閲覧できます。

<http://otn.oracle.co.jp/>

Windows に関する注意事項

UNIX 環境のかわりに Windows プラットフォームを使用する場合は、次の点に注意してください。

- このマニュアルでは、UNIX 構文を使用しています。プラットフォーム固有のファイル名とディレクトリ・セパレータ（Windows の「¥」など）を使用してください。JVM では、プラットフォーム固有の形式でファイル名とパスを指定する必要があるためです。このことは、別のファイル名構文を使用できるシェル（Windows NT の ksh など）を使用している場合にも当てはまります。
- UNIX の場合は、Oracle SQLJ のフロントエンド・スクリプト `sqlj` を使用して、SQLJ トランスレータを起動できます。Windows では、Oracle SQLJ の実行可能ファイル `sqlj.exe` をかわりに使用します。Windows プラットフォームの `.bat` ファイルには、引数への等号（=）の埋込み、引数の文字列操作およびファイル名引数のワイルドカード文字がサポートされず、スクリプトを使用できないためです。
- 環境変数の設定方法は、オペレーティング・システムにより異なります。オペレーティング・システム固有の制限が伴う場合もあります。Windows 95 では、「コントロールパネル」の「システム」の「ハードウェア環境」タブを使用します。また、Windows 95 では変数の設定に「=」文字を使用できないので、SQLJ でサポートされている「#」を「=」のかわりに使用して SQLJ_OPTIONS を設定します。この環境変数は、SQLJ のオプション設定に使用できます。環境変数の設定および構文については、オペレーティング・システムのドキュメントで確認してください。なお、サイズ上の制約には注意してください。
- どのようなオペレーティング・システムおよび環境を使用している場合も同様ですが、特定の制限事項には留意してください。特に、展開された SQLJ コマンドラインのサイズが、許容されるコマンドラインのサイズの最大値を超えないようにしてください。たとえば、Windows 95 の場合は 250 文字以内、Windows NT の場合は 4000 文字以内にします。詳細は、使用するオペレーティング・システムのドキュメントを参照してください。

詳細は、Windows プラットフォームの README ファイルを参照してください。

SQLJ の基本事項

この章では、Oracle SQLJ インストールと設定をテストし、簡単なアプリケーションを実行します。

Oracle データベースと Oracle JDBC ドライバを使用している場合は、『Oracle9i JDBC 開発者ガイドおよびリファレンス』の説明に従って、JDBC インストールも検証します。

この章では、次の項目について説明します。

- [前提と要件](#)
- [インストールおよび設定の確認](#)
- [設定のテスト](#)

前提と要件

ここでは、Oracle SQLJ を実行するための環境の前提とシステムの要件を次の項目に従って説明します。

- [環境に対する前提](#)
- [Oracle SQLJ を使用するための要件](#)
- [Oracle SQLJ 環境: 主な使用例とガイドライン](#)
- [環境の問題点と制限事項](#)
- [Oracle SQLJ の下位互換性](#)

環境に対する前提

Oracle SQLJ を実行するシステムでは、次の内容を前提とします。

- 使用しているシステム上で実行できる標準 Java 環境があること。通常は、Sun 社の JDK を使用しますが、他の Java 製品も使用できます。Java（通常は java）と Java コンパイラ（通常は javac）を実行できることを確認してください。

Sun JDK 上で Oracle SQLJ アプリケーションを変換して実行するには、JDK 1.2.x 以上または JDK 1.1.x のバージョンおよび対応する JDBC ドライバを使用する必要があります。これらの JDK のいずれにも対応する Oracle JDBC Thin ドライバおよび OCI ドライバのバージョンがあります。

詳細は、2-4 ページの「[Oracle SQLJ 環境: 主な使用例とガイドライン](#)」を参照してください。

注意： Java Runtime Environment (JRE) は、Oracle9i でインストールされるものも含めて、単独では SQLJ プログラムを変換できません。ただし JRE は、すでに変換されコンパイルされている SQLJ プログラムの実行であれば実行できます。

- 使用している環境で、JDBC アプリケーションを実行できること。

Oracle データベースおよび Oracle JDBC ドライバを使用している場合、『Oracle9i JDBC 開発者ガイドおよびリファレンス』の第 2 章「スタート・ガイド」の手順を完了する必要があります。Oracle JDBC ドライバについて、またユーザーの環境に適したドライバを確認する方法は、同じマニュアルの第 1 章「概要」を参照してください。

注意： Oracle 以外の JDBC ドライバを使用する場合は、次の操作が必要です。

- `connect.properties` の変更。詳細は、2-13 ページの「[ランタイム接続の設定](#)」を参照してください。
 - 4-5 ページの「[ランタイムに使用するドライバの選択および登録](#)」のデモ・アプリケーションの変更。ドライバの登録が完了した後に `Oracle.connect()` メソッドがコールされるように変更してください。
-

Oracle SQLJ を使用するための要件

Oracle SQLJ を使用するには、次のものがが必要です。

- Sun 社の標準 `java.sql` JDBC インタフェースを実装した JDBC ドライバ。
Oracle SQLJ は、標準 JDBC ドライバで動作します。
- JDBC ドライバを使用してアクセスできるデータベース・システム。
- SQLJ トランスレータ用クラス・ファイル

トランスレータ関連クラスは、次のファイル内で使用可能です。

`$Oracle_Home/sqlj/lib/translator.jar` (または `.zip`)

- SQLJ ランタイム用クラス・ファイル

各 SQLJ ランタイムのバージョンが使用可能です。Java 環境および JDBC ドライバと互換性のあるランタイムのバージョンを選択する必要があります (これらはすべて `$Oracle_Home/sqlj/lib` にあります)。

- `runtime12.jar` (または `.zip`) : JDK 1.2.x 以上のバージョン環境で Oracle9i JDBC ドライバを使用する場合。ISO SQLJ の完全機能を提供します。
- `runtime12ee.jar` (または `.zip`) : J2EE 環境 (JDK 1.2.x 以上を使用) で Oracle9i JDBC ドライバを使用する場合。ISO SQLJ の完全機能を提供します。
- `runtime11.jar` (または `.zip`) : JDK 1.1.x 環境で Oracle9i JDBC ドライバを使用する場合。
- `runtime.jar` (または `.zip`) : 現行より古い Oracle JDBC ドライバと JDK 環境 (Oracle JDBC リリース 8.1.7 以下) で使用する場合。
- `runtime-nonoracle.jar` (または `.zip`) : Oracle 以外の JDBC ドライバと JDK 環境で使用する場合。

注意： 次の点にも注意してください。

- SQLJ リリース 8.1.6 以下では、ランタイム・ライブラリは1つのみで、runtime の JAR または ZIP ファイルは translator の JAR または ZIP ファイルのサブセットでした。Oracle9i で、これは変更されています。ランタイム・ファイルとトランスレータ・ファイルを両方とも CLASSPATH に指定する必要があります。
 - runtime-nonoracle ライブラリは、様々な Java および JDBC 環境間で移植性を実現していますが、Oracle 固有機能はサポートしていません。
 - runtime ライブラリは、様々な Java および JDBC 環境間で柔軟性を実現していますが、すべての ISO SQLJ 機能はサポートしていません。
 - java.sql.Ref、Clob、Blob、Struct および SQLData といった JDBC 2.0 型用 ISO SQLJ 互換サポートが必要な場合は、JDK 1.2 または J2EE、および Oracle9i JDBC ドライバとともに runtime12 または runtime12ee ライブラリを使用する必要があります。
 - すでに変換、コンパイルおよびカスタマイズが完了した SQLJ アプリケーションのみを実行している場合は、translator の JAR または ZIP は必要ありません。
 - translator およびランタイム JAR ファイルは圧縮されています。ただし、ZIP ファイルは、最大限の移植性を持たせるために圧縮されていません。
-
-

Oracle SQLJ 環境 : 主な使用例とガイドライン

完全な動作環境を確保するために、環境を様々な角度から (SQLJ とそのコード生成モード、JDBC および JDK など) 考慮する必要があります。この項では、最初に、サポートされる組合せで Oracle の主な 2 つの使用例を説明し、次に、一般的で重要なガイドラインについて説明します。

関連情報は、2-6 ページの「[環境の問題点と制限事項](#)」を参照してください。

注意： コード生成は、Oracle SQLJ の `-codegen` オプションによって決まります。詳細は、8-50 ページの「[コードの生成 \(-codegen\)](#)」を参照してください。

使用例 1: Oracle 固有コード

ここでは、Oracle 固有コード生成の一般的な環境設定を示します。この場合、SQLJ 汎用 runtime ライブラリがオプションではないことに注意してください。

- SQLJ コード生成: `-codegen=oracle` (デフォルト)
 - 変換時の SQLJ ライブラリ:
 - Oracle SQLJ runtime11 ライブラリおよび JDK 1.1または
 - Oracle SQLJ runtime12 または runtime12ee ライブラリおよび JDK 1.2 以上
 - JDBC および JDK: Oracle JDBC リリース 9.0.1 以上および JDK 1.1 以上
- 9.0.1 または 9.2 JDBC ドライバに対して実行することがある場合は、9.0.1 に対して変換します。一般に、使用する可能性のある最も古いドライバに対してコンパイルします。
- 実行時の SQLJ ライブラリ: 変換時と同じライブラリ

使用例 2: ISO 標準コード

ここでは、ISO 標準コード生成の一般的な環境設定を示します。

1. SQLJ コード生成: `-codegen=iso`
2. 変換時の SQLJ ライブラリ:
 - SQLJ runtime11 ライブラリおよび JDK 1.1または
 - SQLJ runtime12 または runtime12ee ライブラリおよび JDK 1.2 以上または、Oracle JDBC リリース 8.1.7 以下の場合は
 - SQLJ 汎用 runtime ライブラリおよび JDK 1.1 以上
3. JDBC および JDK: Oracle JDBC の任意のバージョンおよび JDK 1.1 以上
4. 実行時の SQLJ ライブラリ: 変換時と同じライブラリ

環境の例: 主なガイドライン

Oracle SQLJ を実行する環境について、次の重要なガイドラインと考慮事項に留意してください。

- 一般に、アプリケーションの全コンポーネントの変換時およびコンパイル時は、同じバージョンの SQLJ ライブラリ、JDBC ライブラリおよび JDK を使用します。

- 次の相互の互換性を常に考慮してください。
 - JDK 1.1 の環境または JDK 1.2 以上の環境で、コンパイルされた同じコードを実行できるようにする場合は、JDK 1.1 環境の `runtime11` ライブラリに対して変換します。これによって、実行時に JDK 1.2 環境の `runtime12` ライブラリまたは `runtime12ee` ライブラリを使用するという選択が許可されます。JDK 1.2 に対して変換すると、JDK 1.1 ではサポートされないいくつかの JDBC 2.0 API がクラス・ファイルにコンパイルされます。
 - リリース 9.0.1 またはリリース 9.2 の Oracle JDBC ドライバに対して実行できるようにするには、9.0.1 ドライバに対して変換します。これによって、実行時にいずれのドライバ・バージョンも使用できるようになります。

生成されたコードは、変換時に CLASSPATH の JDBC ドライバに対応して最適化されます。
 - 相互の互換性を最大限にするため、宣言済みの接続コンテキスト・クラスは使用しないでください。JPublisher を使用する場合は、`-compatible` オプションおよび `-context` オプションのデフォルト設定を使用します。これらのオプションの詳細は、『Oracle9i JPublisher ユーザーズ・ガイド』を参照してください。

相互の互換性のために `runtime` ライブラリを使用することも考えられますが、このライブラリにはデメリットがあります（Oracle9i `oracle.jdbc.OracleXXX` インタフェースをサポートしないことが、中間層で起きる問題の原因となるなど）。

環境の問題点と制限事項

ここでは、環境に関する主な問題点（JDK 移行の型マップに関する問題点）について説明し、その結果生じる制限事項と、他の一部の環境に関する問題点と制限事項をリストします。これらは、2-4 ページの「[Oracle SQLJ 環境：主な使用例とガイドライン](#)」に説明されている一般的な使用例を計画する際の考慮事項の一部です。

注意：

- 汎用 `runtime` ライブラリの記述については、Oracle JDBC 環境（リリース 8.1.7 以下）を前提としています。Oracle JDBC 以外の環境では、`runtime-nonoracle` ライブラリで置き換えてください。
 - `runtime` ライブラリおよび `runtime-nonoracle` ライブラリは、下位互換性を主目的としています。これらは Oracle 固有の機能をサポートしていません。
-
-

JDK 移行の型マップに関する問題点

JDBC 型マップの型は、JDK 1.1.x と JDK 1.2.x 間で `java.util.Dictionary` から `java.util.Map` に変更されました。

すべての SQLJ 接続コンテキスト・クラスの `getTypeMap()` メソッドは、マップ・インスタンスの型を戻します。Java 型システムでは、実装されたメソッドが、基になるインタフェース（この場合は、JDBC Connection インタフェース）で指定されている型を正確に戻す必要があります。次のメソッド・シグネチャを例にとります。

```
java.util.Hashtable getTypeMap() { ... }
```

これは、`Hashtable` が `Dictionary` を拡張して `Map` を実装しているのです。JDK 1.1 と JDK 1.2 の両方の仕様を実装するように思われますが、Java 型システムでは許容されません。

SQLJ コードで接続コンテキスト型を宣言している場合、JDK 1.1 インタフェースと JDK 1.2 インタフェース間のこの非互換性によって次の結果が生じます。

- SQLJ runtime11 ライブラリに対して変換する場合や、SQLJ 汎用 runtime ライブラリに対する変換時に任意の JDK 環境でコンパイルする場合も含め、JDK 1.1.x 環境でコンパイルすると、生成されたコードで `Dictionary` インスタンスが戻ります。JDK 1.1.x および JDK 1.2.x 以上の両方がある環境で、コードを実行する必要があります。（JDK 1.2 環境では、SQLJ ランタイムが必要に応じて Java リフレクションを使用します。）
- SQLJ runtime12 ライブラリまたは runtime12ee ライブラリに対して変換する場合も含め、JDK 1.2.x 以上の環境でコンパイルすると、JDK 1.1.x 環境で実行できなくなります（特に、JDK 1.1.x では `java.util.Map` が定義されていません）。
- JDK 1.1.x 環境で `.sqlj` ファイルを変換する場合、生成された `.java` ファイルは JDK 1.2.x 以上ではコンパイルできません。（これは、変換とコンパイルを別々に行うために SQLJ を `-compile=false` 設定で実行する場合に該当します。）同様に、JDK 1.2.x 以上の環境で `.sqlj` ファイルを変換する場合も、生成された `.java` ファイルは JDK 1.1.x でコンパイルできません。

注意： JPublisher を使用する場合、デフォルトの設定では、接続に `DefaultContext` インスタンスを使用します。これによって、前述の問題は発生しません。しかし、JPublisher のデフォルトでない `-context` オプション設定や、`-compatible` オプション設定の `both8i` または `8` を使用すると、JPublisher で生成される `.sqlj` ソース・コードでは、接続コンテキスト型が宣言されます。詳細は、『Oracle9i JPublisher ユーザーズ・ガイド』を参照してください。

その他の環境の問題点と制限事項

次のリストは、その他の環境での問題点と制限事項です。大部分は前述の型マップの問題点に関連します。

- デフォルトの `-codegen=oracle` 設定を使用すると、汎用 runtime ライブラリは使用できません。runtime11 ライブラリ (JDK 1.1 の場合) や、runtime12 ライブラリまたは runtime12ee ライブラリ (JDK 1.2 以上の場合) を使用する必要があります。
- Oracle JDBC リリース 8.1.7 以下の場合、`-codegen=iso` および汎用 runtime ライブラリ (runtime11、runtime12 または runtime12ee でなく) を使用する必要があります。
- 宣言されている接続コンテキスト・クラスを使用する SQLJ アプリケーションの場合は、すべてのモジュールを (runtime ライブラリまたは runtime11 ライブラリを使用して) JDK 1.1 に対して変換、またはすべてのモジュールを (runtime12 ライブラリまたは runtime12ee ライブラリを使用して) JDK 1.2 以上に対して変換する必要があります。いずれかに統一されていることが必要です。これには、デフォルトでない `-context` オプションが設定されている JPublisher を使用する場合も含まれます。
- 少なくとも変換したバージョンと同じバージョンの JDK に対しては実行できます。JDK 1.1.x 環境では、JDK 1.1.x または JDK 1.2 以上の環境でアプリケーションを実行できます。これは、Oracle SQLJ でサポートされない `oracle.jdbc2` パッケージを使用した JDBC コードがないことが前提です。Oracle JDBC では、このパッケージを使用し、JDK 1.1.x 環境での JDBC 2.0 型をサポートしました。JDK 1.2 環境での変換では、JDK 1.2 以上の環境でアプリケーションを実行できます。

注意：

- Oracle JDBC リリース 8.1.5 以下では、JDK 1.2.x はサポートされません。
 - Oracle9i では、Oracle SQLJ および Oracle JDBC は JDK 1.0.2 をサポートしません。(リリース 8.1.6 が JDK 1.0.2 をサポートする最後の Oracle JDBC のリリースで、Oracle SQLJ では JDK 1.0.2 をサポートしません。) これは、JDK 1.0.2 を使用するブラウザで実行するアプレットも特別な対応をしていなければ対象となります。(この章では、アプレットについては説明していません。1-22 ページの「アプレットでの SQLJ の実行」を参照してください。)
-
-

Oracle SQLJ の下位互換性

Oracle SQLJ の下位互換性について、次の点に注意してください。

- SQLJ トランスレータの現行より前のリリースで生成されたコードは、次のようになります（相互の互換性の制限事項は、2-6 ページの「[環境の問題点と制限事項](#)」を参照してください）。
 - 現行のランタイム（.jar または .zip）ライブラリに対して実行できます。
 - 現行のランタイム（.jar または .zip）ライブラリに対してコンパイル可能です。
- Oracle 固有のトランスレータ出力（デフォルトの `-codegen=oracle` 設定で生成されたコード）は、`runtime11`、`runtime12` または `runtime12ee` ライブラリを使用して、作成および実行する必要があります。

追記：

- Oracle JDBC および SQLJ の将来のリリースで、このようなコードを実行できるようになります。
- ただし、Oracle JDBC および Oracle SQLJ ランタイムの以前のリリースでは、このようなコードを実行できるようになりません。この状況では、コードを再変換する必要があります。

また、Oracle 固有コードは移植できません。

- ISO 標準生成コード（`-codegen=iso`）は、現行の `runtime`（.jar または .zip）ライブラリを使用して、以前の Oracle JDBC リリースに対して作成および実行できます。

注意： Oracle JDBC の以前のデータベース・リリースへの下位互換性に関しては、Oracle JDBC ドライバのどのリリースでも、7.3.4 から JDBC ドライバのリリース番号までの Oracle データベース・リリースと互換性があります。詳細は、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。

インストールおよび設定の確認

前述の前提と要件が満たされていることを確認した後、Oracle SQLJ インストールを確認します。

インストール先のディレクトリとファイルの確認

次のディレクトリがインストールされ、その中にファイルが格納されていることを確認します。

Oracle JDBC のディレクトリ

Oracle JDBC ドライバのいずれかを使用している場合は、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照して、システムにインストールする必要のある JDBC ファイルを確認してください。

Oracle SQLJ のディレクトリ

Oracle9i Java 環境をインストールすると、sqlj ディレクトリが \$Oracle_Home ディレクトリの下に作成されます。sqlj ディレクトリには次のサブディレクトリがあります。

- demo (この章で説明するものを含むデモ用アプリケーション)
- doc
- lib (SQLJ のクラス・ファイルを格納する .jar または .zip ファイル)

また、\$Oracle_Home には、次のディレクトリがあります。このディレクトリには、すべての Java 製品で利用できるユーティリティが格納されています。

- bin

これらのディレクトリがすべて作成され、ファイルが格納されたことを確認してください。特に、lib と bin は重要なディレクトリです。

この構造は Web サイト (Oracle Technology Network の <http://otn.oracle.com> など) から SQLJ をダウンロードした場合もほぼ同じです。SQLJ および JPublisher 実行可能ファイルの両方がある bin ディレクトリは、sqlj ディレクトリに直接できます。

PATH と CLASSPATH の設定

環境変数 PATH と環境変数 CLASSPATH が、Oracle SQLJ（および使用する場合は Oracle JDBC）用に設定されていることを確認します。

Oracle JDBC 用の PATH と CLASSPATH

Oracle JDBC ドライバのうちいずれか 1 つを使用する場合は、使用する環境に対応した Oracle JDBC クラスの JAR または ZIP ファイルが必要です。

JDK 1.1 互換クラスは classes111.jar または .zip に、JDK 1.2（以上）の互換クラスは classes12.jar または .zip に、JDK 1.4 互換クラスは ojdbc14.jar または .zip に格納されています。Sun 社の JDK を使用する場合、対応する JAR または ZIP ファイル名が CLASSPATH に指定されていることを確認してください。ドライバ・デバッグ情報を出力できる classes12_g.jar など、代替 JDBC ドライバ・ライブラリも使用可能です。

Oracle JDBC のライブラリ、必要な PATH および CLASSPATH 設定については、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。

Oracle SQLJ 用の PATH と CLASSPATH

Oracle SQLJ の PATH 変数と CLASSPATH 変数は、次のように設定します。

PATH 設定 絶対パスを指定せずに sqlj スクリプト（SQLJ トランスレータを起動するスクリプト）を実行する場合には、環境変数 PATH に次のディレクトリが指定してあることを確認してください。

```
$Oracle_Home/bin
```

Windows では円記号を使用します。\$Oracle_Home には、実際の Oracle ホーム・ディレクトリを指定します。

CLASSPATH 設定 環境変数 CLASSPATH を変更して、カレント・ディレクトリと次の設定（.jar または .zip）を指定します。

```
$Oracle_Home/sqlj/lib/translator.jar
```

Windows では円記号を使用します。\$Oracle_Home には、実際の Oracle ホーム・ディレクトリを指定します。

また、CLASSPATH の設定に、次のランタイム・ライブラリのうちのいずれか 1 つ（.jar または .zip）を含める必要があります。

```
$Oracle_Home/sqlj/lib/runtime12.jar  
$Oracle_Home/sqlj/lib/runtime12ee.jar  
$Oracle_Home/sqlj/lib/runtime11.jar  
$Oracle_Home/sqlj/lib/runtime.jar  
$Oracle_Home/sqlj/lib/runtime-nonoracle.jar
```

JDBC ドライバと Java 環境用に使用するランタイム・ライブラリについては、2-3 ページの「[Oracle SQLJ を使用するための要件](#)」を参照してください。

重要： ランタイム・ライブラリを追加しない場合は、SQLJ トランスレータを実行できません。CLASSPATH にランタイム・ライブラリとトランスレータ・ライブラリを指定する必要があります。

SQLJ が適切にインストールされていることと、SQLJ、JDBC および Java のバージョン情報を確認するには、次のコマンドを実行します。

```
sqlj -version-long
```

sqljutl パッケージのインストールの確認

注意： この手順は、変換時のオンライン・チェックのみに該当し、SQLJ スタード・プロシージャまたはファンクションを 8.1.5 以下の Oracle データベース（またはサーバー側 JVM なしにインストールされたリリース 8.1.5 以上のデータベース）で使用していれば適用できます。

sqljutl パッケージは、Oracle データベースのスタード・プロシージャとスタード・ファンクションのオンライン・チェックを行うために使用します。リリース 8.1.5 以上の Oracle データベースでは、データベースのサーバー側 JVM をインストール中に、SYS スキーマの下に自動的にインストールされます。sqljutl のインストールを確認するには、たとえば、SQL*Plus から SQL コマンドを実行します。

```
describe sys.sqljutl
```

このコマンドを実行すると、パッケージの簡単な説明が出力されます。パッケージが見つからないというメッセージが表示された場合は、手動でインストールする必要があります。その場合は、SQL*Plus を使用して sqljutl.sql スクリプトを実行します。このスクリプトは、次のディレクトリにあります。

```
$Oracle_Home/sqlj/lib/sqljutl.sql
```

(sqljutl パッケージが SYS スキーマにインストールされます。)

必要に応じて、インストールに関するドキュメントを参照してください。

設定のテスト

データベース、JDBC および SQLJ セットアップをテストするには、次のソース・ファイルに定義されたデモ用アプリケーションを使用します。

- TestInstallCreateTable.java
- TestInstallJDBC.java
- TestInstallSQLJ.sqlj
- TestInstallSQLJChecker.sqlj

また、Java プロパティ・ファイルである `connect.properties` も提供されています。このファイルは、データベース接続の設定に使用します。このファイルを変更し、使用するユーザー名、パスワードおよび URL を指定します。

これらのデモ用アプリケーションは、SQLJ インストールにより `demo` ディレクトリ中に格納されます。

```
$Oracle_Home/sqlj/demo
```

必要に応じて、一部のソース・ファイルを変更し、変換やコンパイルを行います（後述の説明を参照）。

Oracle SQLJ インストールで提供されるデモ用アプリケーションでは、ユーザー名 `scott` およびパスワード `tiger` の Oracle アカウントで表を参照します。このアカウントは、ほとんどの Oracle インストールで使用されています。必要に応じて、`scott` と `tiger` に別の値を使用します。

注意： デモ用アプリケーションを実行するには、前述のように `demo` ディレクトリをカレント・ディレクトリにし、このカレント・ディレクトリ（「.」）を `CLASSPATH` で指定する必要があります。

ランタイム接続の設定

ここでは、`connect.properties` ファイルを変更し、Oracle にランタイム接続を設定します。このファイルは `demo` ディレクトリにあり、次のように記述されています。

```
# Users should uncomment one of the following URLs or add their own.
# (If using Thin, edit as appropriate.)
#sqlj.url=jdbc:oracle:thin:@localhost:1521:ORCL
#sqlj.url=jdbc:oracle:oci:@
#
# User name and password here
sqlj.user=scott
sqlj.password=tiger
```

（`scott` と `tiger` は、デモ用アプリケーションで使用されるユーザー名とパスワードです。）

Oracle JDBC ドライバを使用した接続

Oracle9i の場合、Oracle JDBC OCI ドライバ用接続文字列に `oci` を使用します。ただし、下位互換性を保つために `oci8` も許可されているので、既存のコードを変更する必要はありません。（Oracle JDBC リリース 7.3.4 では `oci7` も使用できます。）

JDBC Thin ドライバを使用する場合は、`connect.properties` の `thin URL` 行のコメント設定を外し、Oracle 接続にあわせて変更します。URL は、JDBC ドライバの設定時に指定した URL を使用してください。

Oracle 以外の JDBC ドライバを使用した接続

Oracle JDBC 以外のドライバを使用している場合は、`connect.properties` に行を追加して、次のように適切な URL を指定します。

```
sqlj.url=your_URL_here
```

URL は、JDBC ドライバの設定時に指定した URL を使用してください。

ドライバはコード中に明示的に登録する必要があります。Oracle JDBC ドライバを使用すると、この登録はデモ用およびテスト用プログラムによって自動的に行われます。4-5 ページの「ランタイムに使用するドライバの選択および登録」を参照してください。

また、SQLJ デモ用プログラムでは、次のコードを置換する必要があります。

```
Oracle.connect(url, user, password);
```

次のように変更します。

```
DriverManager.registerDriver(new yourdriver());
Connection conn = DriverManager.getConnection(url, user, password);
conn.setAutoCommit(false);
DefaultContext.setDefaultContext(new DefaultContext(conn));
```

データベース検証用の表の作成

ここでのテストでは、SALES という表を使用します。TestInstallCreateTable をコンパイルして実行すると、次のような表が作成されます。ただし、データベースと JDBC ドライバが連携していて、しかも `connect.properties` ファイル中の接続設定が適切である必要があります。

```
javac TestInstallCreateTable.java
java TestInstallCreateTable
```

注意： SALES 表がスキーマにすでにあり、それを変更しない場合は、TestInstallCreateTable.java を編集して表の名前を変更します。または、元の表を削除して、新しい表に置き換えます。

TestInstallCreateTable を使用しない場合は、かわりに SALES 表を作成します。その場合は、コマンドライン・プロセッサ (SQL*Plus など) で、次のコマンドを実行します。

```
CREATE TABLE SALES (  
    ITEM_NUMBER NUMBER,  
    ITEM_NAME CHAR(30),  
    SALES_DATE DATE,  
    COST NUMBER,  
    SALES_REP_NUMBER NUMBER,  
    SALES_REP_NAME CHAR(20));
```

JDBC ドライバの検証

Oracle JDBC ドライバをさらに詳しく調べる場合は、TestInstallJDBC デモを使用します。

前の項で説明したとおりに接続が connect.properties ファイルに正しく設定されていることを確認し、その後で、次に示したように TestInstallJDBC をコンパイルして実行します。

```
javac TestInstallJDBC.java  
java TestInstallJDBC
```

次のように出力されます。

```
Hello, JDBC!
```

SQLJ トランスレータとランタイムの検証

TestInstallSQLJ デモを変換して実行します。このデモは、TestInstallJDBC と同じような機能を持つ SQLJ アプリケーションです。次のコマンドを実行して、ソース・コードを変換します。

```
sqlj TestInstallSQLJ.sqlj
```

エラーがなければ、短時間でシステム・プロンプトが再度表示されます。このコマンドでは、Oracle データベース用にアプリケーションのコンパイルやカスタマイズもできます。

UNIX 環境では、sqlj スクリプトは \$Oracle_Home/bin にあります。このディレクトリは、前述のように PATH 内に指定されていることが必要です。(Windows では、bin ディレクトリの sqlj.exe 実行可能ファイルを使用します。) SQLJ translator の JAR または ZIP ファイルには、SQLJ トランスレータおよびランタイム用のクラス・ファイルがあります。\$Oracle_Home/sqlj/lib にあり、前述のように CLASSPATH に存在する必要があります。

アプリケーションを実行するには、次のようにします。

```
java TestInstallSQLJ
```

次のように出力されます。

```
Hello, SQLJ!
```

SQLJ トランスレータからデータベースへの接続の検証

SQLJ トランスレータをデータベースに接続できる場合、変換時に SQL 操作のオンラインのセマンティクス・チェックを実行できます。SQLJ トランスレータは Java で記述されており、JDBC を使用して必要な情報を指定されたデータベース接続から取得します。オンラインのセマンティクス・チェックの接続パラメータを指定するには、sqlj スクリプト・コマンドラインまたは SQLJ プロパティ・ファイル（デフォルトでは、sqlj.properties）を使用します。

demo ディレクトリで、sqlj.properties ファイルを編集し、必要に応じて sqlj.password 行、sqlj.url 行および sqlj.driver 行に対して更新、コメントまたはコメント削除を行い、connect.properties ファイルへの編集内容をデータベース接続情報に反映します。詳細は、sqlj.properties ファイルのコメントを参照してください。

Oracle JDBC OCI ドライバを使用する場合の、ドライバ、URL およびパスワードの設定例を次に示します。ユーザー名については後で説明します。

```
sqlj.url=jdbc:oracle:oci:@
sqlj.driver=oracle.jdbc.OracleDriver
sqlj.password=tiger
```

オンラインのセマンティクス・チェックは、変換時の接続に対してユーザー名を指定した時点で有効になります。ユーザー名を指定するには、sqlj.properties ファイルの sqlj.user 行のコメント設定を外すか、-user コマンドライン・オプションを使用します。ユーザー名、パスワード、URL およびドライバ名は、コマンドラインまたはプロパティ・ファイルで設定できます。詳細は、8-32 ページの「[接続オプション](#)」を参照してください。

オンラインのセマンティクス・チェックをテストするには、次のように TestInstallSQLJChecker.sqlj ファイル（demo ディレクトリ）を変換します。（または、別のユーザー名を使用してテストする方法もあります。）

```
sqlj -user=scott TestInstallSQLJChecker.sqlj
```

Oracle JDBC ドライバを使用している場合は、次のエラー・メッセージが表示されます。

```
TestInstallSQLJChecker.sqlj:41: Warning: Unable to check SQL query. Error returned
by database is: ORA-00904: 列名が無効です
```

TestInstallSQLJChecker.sqlj を編集し、41 行目で検出されたエラーを修正します。列名には、ITEM_NAMAE ではなく、ITEM_NAME を使用します。このように列名を変更すると、エラーが発生しなくなります。次のコマンドでアプリケーションを変換して実行します。

```
sqlj -user=scott TestInstallSQLJChecker.sqlj
java TestInstallSQLJChecker
```

正常に処理されると、次のように出力されます。

```
Hello, SQLJ Checker!
```

基本的な言語機能

この章では、アプリケーションのコーディングに使用する、SQLJ の基本的な言語機能と構文について説明します。

SQLJ 文は常に `#sql` トークンで開始し、2 つのメイン・カテゴリに分割できます。1 つは宣言で、イテレータ（JDBC の結果セットに類似）や接続コンテキスト（使用する SQL エンティティの集合に従って接続を厳密に分類するためのコンテキスト）の Java クラスの作成に使用します。もう 1 つは実行文で、埋込みの SQL 操作の実行に使用します。

詳細は、[第 7 章「拡張言語機能」](#)を参照してください。

この章では、次の項目について説明します。

- [SQLJ 宣言の概要](#)
- [SQLJ 実行文の概要](#)
- [Java ホスト式、コンテキスト式および結果式](#)
- [単一行の問合せ結果 : SELECT INTO 文](#)
- [複数行の問合せ結果 : SQLJ イテレータ](#)
- [代入文 \(SET\)](#)
- [ストアド・プロシージャおよびストアド・ファンクションのコール](#)

SQLJ 宣言の概要

SQLJ 宣言は `#sql` トークンで開始され、その後にクラスの宣言が続きます。SQLJ 宣言により、アプリケーションに専用 **Java** 型が導入されます。現行では、SQLJ 宣言にはイテレータ宣言と接続コンテキスト宣言の 2 種類があり、次のように **Java** クラスを定義します。

- イテレータ宣言はイテレータ・クラスを定義します。イテレータは、概念的に JDBC 結果セットと似ていて、複数行の間合せデータの受取りに使用されます。イテレータは、イテレータ・クラスのインスタンスとして実装されます。
- 接続コンテキスト宣言は、接続コンテキスト・クラスを定義します。通常、各接続コンテキスト・クラスは、操作の際に特定の SQL エンティティ（表、ビュー、ストアド・プロシージャなど）を使用する接続に使用されます。すなわち、名前と特性が同じ SQL エンティティを持つスキーマへの接続には、特定の接続コンテキスト・クラスのインスタンスが使用されています。SQLJ では、各データベース接続が、接続コンテキスト・クラスのインスタンスとして実装されます。

SQLJ には、事前定義された `sqlj.runtime.DefaultContext` 接続コンテキスト・クラスがあります。必要な接続コンテキスト・クラスが 1 つのみである場合は、接続コンテキスト宣言の必要がない `DefaultContext` を使用できます。

イテレータ宣言や接続コンテキスト宣言には、必要に応じて次の句を挿入できます。

- `implements` 句: 生成されたクラスによって実装されるインタフェースを 1 つ以上指定します。
- `with` 句: 生成されたクラスに含める初期化済み定数を 1 つ以上指定します。

これらの句については、3-5 ページの「[IMPLEMENTS 句宣言](#)」および 3-6 ページの「[WITH 句宣言](#)」で説明します。

SQLJ 宣言の規則

SQLJ ソース・コード中には、標準 **Java** でクラスを定義できる場所であればどこにでも SQLJ 宣言を記述できます。制限事項は、JDK 1.1.x ではメソッド・ブロック内部に宣言を記述できないことのみです。次に、例を示します。

```
SQLJ declaration;    // OK (top level scope)

class Outer
{
    SQLJ declaration; // OK (class level scope)

    class Inner
    {
        SQLJ declaration; // OK (nested class scope)
    }
}
```

```

void func()
{
    SQLJ declaration; // OK in JDK 1.2.x; ILLEGAL in JDK 1.1.x (method block)
}

```

注意： 標準 Java の場合と同様に、Public クラスは、次の方法のどちらかの方法で宣言する必要があります (Sun 社の JDK の標準 javac コンパイラを使用する場合は、これが要件となります)。

- 別個のソース・ファイル中に宣言する方法。ファイルのベース名はクラス名と同じにします。

または

- クラスレベルの有効範囲またはネストされたクラスレベルの有効範囲に宣言する方法。この方法の場合、`public static` 修飾子を使用することをお勧めします。
-

イテレータ宣言

イテレータ宣言により、問合せデータの受取り用に、イテレータの種類を定義するクラスが作成されます。宣言にはイテレータ・インスタンスの列型を指定します。データベース表から選択できる列型と同じ列型を指定してください。

基本的なイテレータ宣言では、次の構文が使用されます。

```
#sql <modifiers> iterator iterator_classname (type declarations);
```

修飾子の使用は任意で、`public` や `static` などの標準 Java クラス修飾子を使用できます。型宣言はカンマで区切ります。

イテレータには、名前指定イテレータと位置指定イテレータの 2 種類があります。名前指定イテレータには列名と型を指定し、位置指定イテレータには型のみを指定します。

次に、名前指定イテレータ宣言の例を示します。

```
#sql public iterator EmpIter (String ename, double sal);
```

この文を実行すると、SQLJ トランスレータによって、String 属性の `ename` および `double` 属性の `sal` で、`public EmpIter` クラスが作成されます。このイテレータを使用して、データベース表の従業員名と給料の各列から、名前 (`ENAME` および `SAL`) とデータ型 (`CHAR` および `NUMBER`) が一致するデータを選択できます。

EmpIter を名前指定イテレータではなく位置指定イテレータとして宣言すると、次のようになります。

```
#sql public iterator EmpIter (String, double);
```

イテレータの詳細は、3-33 ページの「[複数行の間合せ結果:SQLJ イテレータ](#)」を参照してください。

接続コンテキスト宣言

接続コンテキスト宣言により、接続コンテキスト・クラスが作成されます。このクラスのインスタンスは、通常、特定の SQL エンティティを使用するデータベース接続に使用されます。

基本的な接続コンテキスト宣言では、次の構文が使用されます。

```
#sql <modifiers> context context_classname;
```

イテレータ宣言の場合、修飾子は任意で、標準 Java のどのクラス修飾子でも使用できます。たとえば、次のように指定できます。

```
#sql public context MyContext;
```

この文を実行すると、SQLJ トランスレータによって `public MyContext` クラスが作成されます。SQLJ コード中にこのクラスのインスタンスを使用すると、任意のエンティティ（表、ビューまたはストアド・プロシージャなど）を持つスキーマへのデータベース接続を作成できます。接続先のスキーマが複数の場合は、MyContext の様々なインスタンスを使用することも可能です。ただし、各スキーマには、EMP 表、DEPT 表または TRANSFER_EMPLOYEE ストアド・プロシージャなどを指定する必要があります。

接続コンテキストの宣言は高度なトピックであるため、1 種類の SQL エンティティしか使用しない基本的な SQLJ アプリケーションでは、この宣言は不要です。また、基本的には、`sqlj.runtime.ref.DefaultContext` クラスのインスタンスをいくつか作成して複数の接続を使用できます。この方法では、接続コンテキストの宣言が不要です。

接続および接続コンテキストの概要は、4-6 ページの「[接続の際の考慮事項](#)」を参照してください。接続コンテキストの追加作成の詳細は、7-2 ページの「[接続コンテキスト](#)」を参照してください。

IMPLEMENTS 句宣言

イテレータ・クラスや接続コンテキスト・クラスを宣言する場合、生成されたクラスが実装するインタフェースを1つ以上指定できます。

イテレータ・クラスには次の構文を使用します。

```
#sql <modifiers> iterator iterator_classname implements intfcl,..., intfcn
      (type declarations);
```

`implements intfcl,..., intfcn`の部分は、`implements` 句と呼ばれます。イテレータ宣言では、`implements` 句に続いてイテレータ型宣言をします。

接続コンテキスト宣言には次の構文を使用します。

```
#sql <modifiers> context context_classname implements intfcl,..., intfcn;
```

`implements` 句は、イテレータ宣言または接続コンテキスト宣言のいずれかで有効な場合がありますが、特に `sqlj.runtime.Scrollable` または `sqlj.runtime.ForUpdate` インタフェースの実装時のイテレータ宣言で有効な場合が多くあります。スクロール可能なイテレータは Oracle SQLJ でサポートされています (7-38 ページの「[スクロール可能なイテレータ](#)」を参照してください)。位置指定更新および位置指定削除は現在サポートされていません。

`implements` 句の詳細は、7-36 ページの「[イテレータ宣言での IMPLEMENTS 句の使用](#)」および 7-10 ページの「[接続コンテキスト宣言での IMPLEMENTS 句の使用](#)」を参照してください。

注意： SQLJ の `implements` 句は、Java の `implements` 句に相当します。

次の例では、名前指定イテレータ・クラスの宣言に `implements` 句を使用しています。この例では、イテレータ・インタフェースの `MyIterIntfc` を含む `mypackage` パッケージが作成されていることを前提とします。

```
#sql public iterator MyIter implements mypackage.MyIterIntfc
      (String ename, int empno);
```

クラス `MyIter` の宣言によって、`mypackage.MyIterIntfc` インタフェースが実装されます。

次の例では、`MyConnCtxtIntfc` という名前のインタフェースを実装した接続コンテキスト・クラスを宣言しています。ここでも、`mypackage` パッケージを前提とします。

```
#sql public context MyContext implements mypackage.MyConnCtxtIntfc;
```

WITH 句宣言

接続コンテキスト・クラスやイテレータ・クラスの宣言時には、with 句を使用して、生成されたクラスの定義に含める定数を 1 つ以上指定し初期化できます。Oracle では 1 つの拡張機能をイテレータ宣言に追加していますが、この使用例の大部分は標準です。

標準 WITH 句の使用例

with 句を使用する場合、生成される定数は必ず `public static final` です。イテレータ・クラスには次の構文を使用します。

```
#sql <modifiers> iterator iterator_classname with (var1=value1,..., varN=valueN)
      (type declarations);
```

`with(var1=value1,..., varN=valueN)` の部分が with 句と呼ばれます。イテレータ宣言では、with 句に続けてイテレータ型宣言をします。

with 句と implements 句の両方がある場合は、implements 句を最初にする必要があります。with リストはカッコで囲みますが、implements リストはカッコで囲みません。

次に、with 句を使用する接続コンテキスト宣言の構文を示します。

```
#sql <modifiers> context context_classname with (var1=value1,..., varN=valueN);
```

その例を示します。

```
#sql public context MyContext with (typeMap="MyPack.MyClass");
```

MyContext クラスを宣言すると、typeMap 属性が定義されます。この属性は、String 型の `public static final` で、「MyPack.MyClass」の値に初期化されます。この値は、MyContext クラスのインスタンス上で実行した文に対する SQL と Java 型のマッピングを提供する ListResourceBundle 実装の完全修飾クラス名です。

もう 1 つ例を示します (sensitivity については、すぐ後に説明があります)。

```
#sql public iterator MyAsensitiveIter with (sensitivity=ASENSITIVE)
      (String ename, int empno);
```

この宣言では、カーソル sensitivity に ASENSITIVE が名前指定イテレータ・クラスとして設定されます。

次に、implements 句と with 句の両方を使用した例を示します。(holdability については、すぐ後に説明があります。)

```
#sql public iterator MyScrollableIterator implements sqlj.runtime.Scrollable
      with (holdability=true) (String ename, int empno);
```

implements 句は with 句の前に置く必要があります。

このように宣言すると、インタフェース `sqlj.runtime.Scrollable` が実装され、カーソル `holdability` が名前指定イテレータ・クラスに対して使用可能になります。（ただし、現行の `Oracle9i` では、`holdability` はサポートされていません。）

次のイテレータ宣言の標準定数は、`Oracle SQLJ` ではサポートされません。一般的にはカーソル状態を指定します。次のような特定の値のみを指定できます。

- `holdability` (`true/false`)
- `updateColumns` (列名のカンマ区切りリストを含む `String` リテラル)

`updateColumns` を指定する `with` 句を持つイテレータ宣言には、`sqlj.runtime.ForUpdate` インタフェースを指定する `implements` 句も含める必要があります。

`Oracle SQLJ` では、接続コンテキスト宣言について、次の標準定数をサポートしています。

- `sensitivity` (`SENSITIVE/ASENSITIVE/INSENSITIVE` で、スクロール可能なイテレータの機密性を定義する)
- `returnability` (`true/false` で、イテレータが `Java` ストアド・プロシージャまたはファンクションから戻されるかどうかを定義する)
- `typeMap` (型マップ・プロパティ・リソース名を定義する `String` リテラル)
- `dataSource` (`InitialContext` でデータ・ソースを検索する名前を定義する `String` リテラル)

データ・ソースへの `SQLJ` サポートは、7-12 ページの「[標準データ・ソースのサポート](#)」を参照してください。

次の接続コンテキスト宣言の標準定数は、`Oracle SQLJ` では現在サポートされていません。

- `path` (`Java` ストアド・プロシージャおよびファンクションを解決するために先頭に付加する `PATH` の名前を定義する `String` リテラル)
- `transformGroup` (`SQL` 型に適用する `SQL` 変換グループ名を定義する `String` リテラル)

注意： 事前に定義された標準 `SQLJ` 定数は、`with` 句に定義できますが、この定数すべてが現行の `Oracle9i` または `Oracle SQLJ` ランタイムでサポートされるわけではありません。`Oracle9i` では、前述の例のような標準定数以外の定数を定義できますが、他の `SQLJ` 実装に移行できない可能性があります。また、`-warn=portable` フラグが使用可能になっている場合は、警告が生成されます。このフラグの詳細は、8-44 ページの「[トランスレータからの警告 \(-warn\)](#)」を参照してください。

Oracle 固有の WITH 句の使用例

型マップを接続コンテキスト・クラスに対応付けるために、標準 with 句を接続コンテキスト宣言で使用方法の他に、Oracle では、型マップをイテレータ宣言内でイテレータ・クラスに対応付けるために、with 句を使用できます。次にその例を示します。

```
#sql iterator MyIterator with (typeMap="MyTypeMap") (Person pers, Address addr);
```

Oracle 固有コード生成を使用して（デフォルトのトランスレータ設定 `-codegen=oracle` による）、型マップをアプリケーションで使用する場合は、イテレータと接続コンテキスト宣言で同じ型マップを使用する必要があります。詳細は、4-38 ページの「[Oracle 固有コード生成でのコード上の考慮事項と制限事項](#)」を参照してください。

SQLJ 実行文の概要

SQLJ 実行文は `#sql` トークンで開始し、その後に SQLJ 句を続けます。SQLJ 句には、Java コードで SQL 実行文を記述します。その場合、指定された標準に従った構文を使用します。SQLJ 実行文の埋込み SQL 操作は、DML、DDL、トランザクション制御など、使用する JDBC ドライバでサポートされている SQL 操作になります。

SQLJ 実行文の規則

SQLJ 実行文には次の規則があります。

- Java コードでは、Java ブロック文が許可される場合は常に、SQLJ 実行文も許可されます。つまり、メソッド定義や静的な初期化ブロック内では許可されています。
- 埋込み SQL 操作は、中カッコ `{...}` で囲む必要があります。
- 最後にセミコロン `(;)` を指定する必要があります。

注意：

- SQL 操作（カッコ内）の終了を示す符号としてはセミコロンを使用しないでください。パーサーが操作の終了を認識するのは、SQLJ 句の内部に右中カッコが現れた場合です。
 - SQLJ 実行文の中カッコに囲まれた部分はすべて SQL 構文として扱われ、SQL 規則に従います。ただし、Java ホスト式の場合は例外です。（ホスト式については、3-14 ページの「[Java ホスト式、コンテキスト式および結果式](#)」で説明します。）
 - SQL 操作のオフライン解析では、すべての SQL 構文がチェックされます。しかし、（データベース接続を介して行われる）オンラインのセマンティクス・チェックでは、DML 操作（SELECT、UPDATE、INSERT および DELETE など）のみを解析およびチェックできます。CREATE...、ALTER... などの DDL 操作、COMMIT、ROLLBACK などのトランザクション制御操作、または SQL 操作については解析されません。
-

SQLJ 句

#sql トークンの後に続く SQLJ 句は、文の実行可能な部分です。この句は、中カッコで囲まれた埋込み SQL で構成されます。必要に応じて、この例の result のように、Java 結果式が前に置かれます。

```
#sql { SQL operation };    // For a statement with no output, like INSERT
...
#sql result = { SQL operation };    // For a statement with output, like SELECT
```

1 番目の例のように結果式のないものは、ステートメント句と呼ばれます。2 番目の例のように結果式のあるものは、代入句と呼ばれます。

結果式には、ストアド・ファンクションの戻り値が代入される単純な変数から、複数行 SELECT からの複数列のデータが代入されるイテレータまで、任意のものを使用できます（イテレータは、イテレータ・クラスまたはイテレータ・サブクラスのインスタンスになります）。

SQLJ 文の SQL 操作には、標準 SQL 構文のみを使用するか、SQLJ 固有の構文を持つ句を使用できます（[表 3-1](#) および [表 3-2](#) を参照してください）。

サポートされている SQLJ ステートメント句を [表 3-1](#) に、サポートされている SQLJ 代入句を [表 3-2](#) に示します。それぞれの句の使用の詳細は、「参照先」に記載されているページまたはマニュアルを参照してください。[表 3-1](#) の最後にあげた 2 つの項目は、SQLJ 固有の構文ではなく、標準 SQL 構文または Oracle PL/SQL 構文を使用するステートメント句の一般カテゴリの項目です。

表 3-1 SQLJ ステートメント句

カテゴリ	機能	参照先
SELECT INTO 句	Java ホスト式にデータを取り込みます。	「単一行の問合せ結果 : SELECT INTO 文」 3-30 ページ
FETCH 句	位置指定イテレータからデータをフェッチします。	「位置指定イテレータの使用」 3-43 ページ
COMMIT 句	データへの変更をコミットします。	「手動 COMMIT または ROLLBACK の使用方法」 4-26 ページ
ROLLBACK 句	データへの変更を取り消します。	「手動 COMMIT または ROLLBACK の使用方法」 4-26 ページ
SET SAVEPOINT RELEASE SAVEPOINT ROLLBACK TO 句	以後のロールバックのためのセーブポイントの設定、指定したセーブポイントの解放、セーブポイントのロールバックを行います。	「セーブポイントの使用」 4-28 ページ
SET TRANSACTION 句	アクセス・モードや分離レベルなど、詳細トランザクション制御を使用します。	「詳細なトランザクション制御」 7-44 ページ
プロシージャ句	ストアド・プロシージャをコールします。	「ストアド・プロシージャのコール」 3-54 ページ
代入句	Java ホスト式に値を代入します。	「代入文 (SET)」 3-53 ページ
SQL 句	次に示した標準 SQL 構文および機能を使用します。 (UPDATE、INSERT、DELETE)	『Oracle9i SQL リファレンス』
PL/SQL ブロック	SQLJ 文内部の BEGIN...END または DECLARE...BEGIN...END 無名ブロックを使用します。	「実行文の PL/SQL ブロック」 3-13 ページ 『PL/SQL ユーザーズ・ガイドおよびリファレンス』

表 3-2 SQLJ 代入句

カテゴリ	機能	参照先
問合せ句	SQLJ イテレータにデータを取り込みます。	「複数行の問合せ結果:SQLJ イテレータ」 3-33 ページ
ファンクション句	ストアド・ファンクションをコールします。	「ストアド・ファンクションのコール」 3-55 ページ
イテレータ変換句	JDBC 結果セットを SQLJ イテレータに変換します。	「結果セットから名前指定イテレータまたは位置指定イテレータへの変換」 7-53 ページ

注意： SQLJ 文は、その文の本体を構成する句と同じ名前になります。たとえば、`#sql` の後に `SELECT INTO` 句が続く実行文は、`SELECT INTO` 文になります。

接続コンテキスト・インスタンスと実行コンテキスト・インスタンスの指定

複数のデータベース接続が定義されているときに、実行文に特定の接続コンテキスト・インスタンスを指定する場合は、次の構文を使用します。

```
#sql [conn_context_instance] { SQL operation };
```

接続コンテキスト・インスタンスの詳細は、4-6 ページの「[接続の際の考慮事項](#)」を参照してください。

実行コンテキスト・インスタンス (`sqlj.runtime.ExecutionContext` クラス) を 1 つ以上定義していて、その 1 つを実行文用に指定する場合、次の構文 (接続コンテキスト・インスタンス用と類似) を使用します。

```
#sql [exec_context_instance] { SQL operation };
```

実行コンテキスト・インスタンスを使用して、SQLJ 実行文の SQL 操作の実行状態や実行制御を定義できます。(ただし、この定義は高度な手順です。) たとえば、同一の接続で複数の操作が行われるマルチスレッド状況で、実行コンテキスト・インスタンスを使用できます。詳細は、7-22 ページの「[実行コンテキスト](#)」を参照してください。

また、接続コンテキスト・インスタンスと実行コンテキスト・インスタンスの両方を指定することも可能です。

```
#sql [conn_context_instance, exec_context_instance] { SQL operation };
```

注意：

- 接続コンテキスト・インスタンスと実行コンテキスト・インスタンスは太カッコで囲みます。これらは構文の一部になります。
 - 接続コンテキスト・インスタンスと実行コンテキスト・インスタンスの両方を指定する場合は、接続コンテキスト・インスタンスを先にする必要があります。
-

実行文の例

ここでは、基本的な SQLJ 実行文の例を示します。より複雑な文については、後で説明します。

基本的な INSERT

次の例は、基本的な INSERT です。ステートメント句は、SQLJ 固有の構文を必要としません。

次の行を持つ従業員表 EMP を想定します。

```
CREATE TABLE EMP (  
    ENAME VARCHAR2(10),  
    SAL NUMBER(7,2) );
```

標準 SQL 構文の SQLJ 実行文を使用し、EMP 表に新しい従業員 Joe を挿入し、名前と給料を指定します。

```
#sql { INSERT INTO emp (ename, sal) VALUES ('Joe', 43000) };
```

接続コンテキスト・インスタンスまたは実行コンテキスト・インスタンスを指定する基本的な INSERT

次の例では、接続コンテキスト・インスタンスとして、デフォルトの `sqlj.runtime.ref.DefaultContext` のインスタンスまたは接続コンテキスト宣言で宣言したクラスのインスタンスである `ctx` を使用しています。また、実行コンテキスト・インスタンスとしては、`execctx` を使用しています。

```
#sql [ctx] { INSERT INTO emp (ename, sal) VALUES ('Joe', 43000) };
```

```
#sql [execctx] { INSERT INTO emp (ename, sal) VALUES ('Joe', 43000) };
```

```
#sql [ctx, execctx] { INSERT INTO emp (ename, sal) VALUES ('Joe', 43000) };
```

単純な SQLJ メソッド

この例は、SQLJ コードを使用した単純なメソッドを示し、SQLJ 文と Java 文の関係およびコード中での場所を表しています。この SQLJ 文は、Oracle SQL でサポートされている標準の `INSERT INTO table VALUES` 構文を使用しています。また、この文では、コロン (:) でマークされた Java ホスト式を使用して値を定義しています。ホスト式は、Java コードと SQL 命令の間でのデータの受渡しに使用されます。詳細は、3-14 ページの「[Java ホスト式、コンテキスト式および結果式](#)」を参照してください。

```
public static void writeSalesData (int[] itemNums, String[] itemNames)
    throws SQLException
{
    for (int i =0; i < itemNums.length; i++)
        #sql { INSERT INTO sales VALUES(:(itemNums[i]), :(itemNames[i]), SYSDATE) };
}
```

注意：

- `throws SQLException` を省略しないでください。例外処理の詳細は、4-20 ページの「[例外処理の基本](#)」を参照してください。
- SQLJ ファンクション・コールでも `VALUES` トークンが使用されますが、使用目的が異なります。

実行文の PL/SQL ブロック

PL/SQL ブロックは、SQL 操作と同様に、SQLJ 実行文の中カッコ内で使用できます。次にその例を示します。

```
#sql {
    DECLARE
        n NUMBER;
    BEGIN
        n := 1;
        WHILE n <= 100 LOOP
            INSERT INTO emp (empno) VALUES(2000 + n);
            n := n + 1;
        END LOOP;
    END
};
```

この例では、新しい従業員を EMP 表に挿入するループに入り、従業員番号の 2001 ～ 2100 が作成されます。（この例では、従業員番号以外のデータが後で埋められると想定しています。）

単純な PL/SQL ブロックは、次のように 1 行のコードでも完結します。

```
#sql { <DECLARE ...> BEGIN ... END };
```

SQLJ 文中での無名 PL/SQL ブロックの使用は、アプリケーション内で動的 SQL を使用する 1 つの方法です。動的 SQL を Oracle の拡張機能によって直接使用することもできます。また、SQLJ アプリケーション内部の JDBC コードによって、直接使用することもできます。(7-57 ページの「[動的 SQL のサポート](#)」および 7-48 ページの「[SQLJ と JDBC の関係動作](#)」を参照してください。)

注意：

- セミコロンは、PL/SQL 操作の終了を示す符号としては使用できないので、END の後に付けないようにしてください。パーサーがブロックの終了を認識するのは、SQLJ 句の内部に右中カッコが現れた場合です。
 - PL/SQL は Oracle 固有なので、PL/SQL を SQLJ コードで使用すると、他のプラットフォームへの移植性が失われます。
-
-

Java ホスト式、コンテキスト式および結果式

ここでは、SQLJ コードで使用される Java 式の 3 つのカテゴリについて説明します。ホスト式、コンテキスト式、結果式の 3 種類があります。ここでは、最も頻繁に使用されるホスト式に重点を置いて説明します。(メタ・バインド式と呼ばれる別の式カテゴリは、特に動的 SQL 操作で使用されます。構文は、ホスト式とほぼ同じです。詳細は、7-57 ページの「[動的 SQL のサポート](#)」を参照してください。)

SQLJ は、Java コードと SQL 操作の間の引数の受渡しにホスト式を使用します。これが、Java と SQL の間で情報を通信する方法です。ホスト式は、SQLJ ソース・コードの埋込み SQL 操作内に間隔をおいて配置されます。

Java 識別子のみで構成される最も基本的なホスト式は、ホスト変数と呼ばれます。

コンテキスト式には、SQLJ 文で使用される接続コンテキスト・インスタンスまたは実行コンテキスト・インスタンスを指定します。

結果式には、問合せ結果やファンクションの戻り値の出力変数を指定します。

(結果式および接続コンテキスト・インスタンスと実行コンテキスト・インスタンスの指定については、3-8 ページの「[SQLJ 実行文の概要](#)」でも触れています。)

ホスト式の概要

有効な Java 式はすべて、ホスト式として使用できます。最も単純なホスト式は、単一の Java 変数のみで構成されます。それ以外のホスト式には、次の式があります。

- 算術式
- Java メソッドのコールと戻り値
- Java クラス・フィールド値
- 配列の要素
- 条件式 (a ? b : c)
- 論理式
- ビット単位式

ホスト変数として使用された Java 識別子、またはホスト式で使用された Java 識別子は、次のいずれかを表します。

- ローカル変数
- 宣言したパラメータ
- クラス・フィールド (myclass.myfield など)
- static メソッド・コールまたはインスタンス・メソッド・コール

ホスト式で使用されたローカル変数は、他の Java 変数を宣言できる場所で宣言できます。フィールドは、スーパークラスから継承できます。

SQLJ 実行文がある Java スコープ内で有効な Java 変数は、SQL 文のホスト式で使用できます。ただし、その Java 変数の種類が SQL データ型と変換可能であることが前提になります。

ホスト式は、入力、出力または入出力のいずれにもなります。

入力および出力操作中の Java と SQL 間のデータ変換の詳細は、5-2 ページの「[ホスト式での型サポート](#)」を参照してください。

基本的なホスト式の構文

ホスト式の前にはコロン (:) が付きます。使用するホスト式のモード（入力、出力または入出力）がデフォルトでない場合は、コロンに続けて IN、OUT または INOUT を付け、その後ホスト式を置きます。これらはモード指定子と呼ばれます。ホスト式が INTO リストである場合、またはホスト式が SET 文中の代入式である場合は、デフォルトのモード指定子が OUT になります。それ以外の場合は、デフォルトが IN になります。（デフォルトを使用している場合、必要に応じてモード指定子を設定できます。）

OUT または INOUT ホスト式はすべて、代入可能（左辺値）であることが必要です。（つまり、論理的に等号の左辺に出現できるものにします。）

ホスト式を取り巻く SQL コードには、任意のベンダー固有 SQL 構文を使用できます。このため、SQL 操作の解析およびホスト式の判断を行うときに、構文を想定できません。不明確にならないよう、単純なホスト変数ではないホスト式、つまりドットで区切られていない Java 識別子より複雑なホスト式は、カッコで囲んでください。基本的な構文を要約すると次のようになります。

- モード指定子のない単純なホスト変数の場合は、次の例のように、コロンの後にホスト変数を置きます。

```
:hostvar
```

- モード指定子付きの単純なホスト変数の場合は、次の例のように、コロンの後にモード指定子を付け、続けて空白（スペース、タブ、改行またはコメント）を挿入してからホスト変数を指定します。

```
:INOUT hostvar
```

モード指定子と変数名を区別するために、空白が必要です。

- その他のホスト式の場合は、次の例のように、式をカッコで囲んでモード指定子の後に置きます。モード指定子がない場合はコロンの後に置きます。

```
:IN(hostvar1+hostvar2)
:(hostvar3*hostvar4)
:(index--)
```

この例では、カッコをセパレータとしているので、モード指定子の後に空白を置く必要はありませんが、置くことも可能です。

次の例のように、式が左カッコで始まる場合でも、全体を囲む左カッコがもう 1 つ必要です。

```
:((x+y).z)
:(((y)x).myOutput())
```

構文上の注意

- コロンやモード指定子の後には、どんな場合にも空白を挿入できます。空白を挿入できる場所には、コメントを記入することもできます。次のいずれかの SQL ネームスペースに SQL コメントを記入できます。
 - コロンの後とモード指定子の前
 - モード指定子がない場合は、コロンの後とホスト式の前
 - モード指定子の後とホスト式の前または Java ネームスペースに Java コメントを記入できます。
 - ホスト式内部（小カッコの中）

- ホスト変数およびホスト式に使用される IN、OUT および INOUT 構文には、大文字と小文字の区別がありません。これらのトークンには、大文字、小文字またはその両方使用できます。
- SQLJ ホスト式の IN、OUT および INOUT 構文は、PL/SQL 宣言に使用する構文と似ていますが、混同しないようにしてください。この PL/SQL 宣言の構文は、PL/SQL ストアド・ファンクションおよびプロシージャに渡されるパラメータのモードの指定に使用します。

使用上の注意

- 同じ SQLJ 文中に単純なホスト変数が複数回現れる場合を、次に示します（出力とは、OUT または INOUT 変数のことです）。
 - － ホスト変数を入力変数としてのみ使用する場合は、制約や煩雑な操作は伴いません。
 - － PL/SQL ブロック内にホスト変数が少なくとも 1 回現れた場合、トランスレータの `-warn=portability` フラグが設定されていると、移植性に関する警告が出されます。こうした状況では、各ベンダーの SQLJ ランタイムは、それぞれ固有の動作をします。Oracle の SQLJ ランタイムの場合、ホスト変数が出現するたびに値構文が使用されます（参照セマンティクスと対照）。`-warn=portability` フラグの詳細は、8-44 ページの「[トランスレータからの警告 \(-warn\)](#)」を参照してください。
 - － ホスト変数が、ストアド・プロシージャ・コール、ストアド・ファンクション・コール、SET 文または INTO リスト中の出力変数として少なくとも 1 回出現した場合、警告は出力されません。こうした場合に SQLJ ランタイムでは値構文を使用し、それぞれに対して共通の処理をします。
- 単純なホスト変数のホスト式が SQLJ 文中に複数回出現した場合は、（デフォルトで）値構文を使用し、それぞれを完全に別個のものとして処理します。ただし、SQLJ トランスレータで `-bind-by-identifier=true` 設定を使用している場合は、処理が異なります。`true` に設定されている場合、指定された SQLJ 文または PL/SQL ブロック内に同じホスト変数が複数回出現しても、単一のバインドとして処理されます。8-66 ページの「[識別子によるホスト式のバインド \(-bind-by-identifier\)](#)」を参照してください。
- CHAR 型データと比較するために文字列のホスト式を WHERE 句にバインドする場合は、SQLJ に `-fixedchar` オプションがあることに注意してください。このオプションは、比較時に CHAR 列の空白埋めを考慮します。8-56 ページの「[空白埋めを考慮した CHAR 比較 \(-fixedchar\)](#)」を参照してください。

Oracle SQLJ ランタイムによるホスト式の評価例は、3-21 ページの「[Java 式の実行時評価の例 \(ISO コード生成\)](#)」を参照してください。

ホスト式の例

次に、構文の説明を明確にするために、いくつかの例をあげます。（ここにあげる例のいくつかでは `SELECT INTO` 文を使用しています。この文については、3-30 ページの「[単一行の問合せ結果:SELECT INTO 文](#)」で詳しく説明します。）

1. この例では、2 つの入力ホスト変数を、一方は `WHERE` 句のテスト値として、もう一方はデータベースに送られる新しいデータを含むものとして使用しています。

データベースの従業員表 `EMP` に、従業員名の列 `ENAME` および従業員の給料の列 `SAL` があることを想定します。

この例では、ホスト変数を定義する関連 Java コードも示します。

```
String empname = "SMITH";
double salary = 25000.0;
...
#sql { UPDATE emp SET sal = :salary WHERE ename = :empname };
```

`IN` はデフォルトですが、明示的に示すことも可能です。

```
#sql { UPDATE emp SET sal = :IN salary WHERE ename = :IN empname };
```

見てわかるように、`IN` トークンを使用しない場合は、「:」のすぐ後ろに変数を置けるのに対し、「:IN」の場合はすぐ後ろに空白を置き、その後にホスト変数を置く必要があります。

2. この例では、`SELECT INTO` 文で出力ホスト変数を使用し、従業員番号が 28959 の従業員名を検索します。

```
String empname;
...
#sql { SELECT ename INTO :empname FROM emp WHERE empno = 28959 };
```

`OUT` は `INTO` リストのデフォルトですが、明示的に示すことも可能です。

```
#sql { SELECT ename INTO :OUT empname FROM emp WHERE empno = 28959 };
```

この例では、`EMP` 表の `EMPNO` 列で従業員番号 28959 を検索し、その行の `ename` 列から名前を選択し、それを `empname` ホスト変数に出力します。この場合、Java 文字列で出力されます。

3. この例では、入力ホスト式として算術式を使用しています。Java 変数の `balance` と `minPmtRatio` が乗算されます。その結果は、口座番号 537845 について `creditacct` 表の `minPayment` 列を更新する際に使用されます。

```
float balance = 12500.0;
float minPmtRatio = 0.05;
...
#sql { UPDATE creditacct SET minPayment = :(balance * minPmtRatio)
      WHERE acctnum = 537845 };
```

IN トークンを使用すると次のようになります。

```
#sql { UPDATE creditacct SET minPayment = :IN (balance * minPmtRatio)
      WHERE acctnum = 537845 };
```

4. この例では、入力ホスト式としてメソッドのコールの出力を使用し、さらに入力ホスト変数も使用しています。この文では、`getNewSal()` からの戻り値を使用して、Java 変数 `ename` で指定された従業員（ENAME 列）について、EMP 表の SAL 列を更新します。ホスト変数に初期値を設定する Java コードも示されています。

```
String empname = "SMITH";
double raise = 0.1;
...
#sql {UPDATE emp SET sal = :(getNewSal(raise, empname))
      WHERE ename = :empname};
```

結果式とコンテキスト式の概要

コンテキスト式は、SQLJ 実行文で使用される接続コンテキスト・インスタンスまたは実行コンテキスト・インスタンスの名前を指定する入力式です。式を評価し、そのような名前を生成する Java 式を使用できます。

結果式は、問合せ結果やファンクションの戻り値に使用する出力式です。代入可能、つまり論理的に等号の左辺に置くことが可能な（左辺値ともいう）正当な Java 式を使用できます。

次に示す例は、結果式とコンテキスト式のいずれにも使用できます。

- ローカル変数
- 宣言したパラメータ
- クラス・フィールド（`myclass.myfield` など）
- 配列の要素

構文上、ホスト式は SQL 領域（SQLJ 実行文の中カッコ内）に置きますが、結果式とコンテキスト式は SQLJ 領域に置きます。このため、結果式やコンテキストの前には、コロンを付けないようにする必要があります。

Java 式の実行時評価

ここでは、アプリケーションの実行時の、Java ホスト式、接続コンテキスト式、実行コンテキスト式および結果式の評価について説明します。

これらの式をすべて使用する、単純化された SQLJ 実行文を次に示します。

```
#sql [connctx_exp, execctx_exp] result_exp = { SQL with host expression };
```

Java 式には、必要に応じて次の用途があります。

- 接続コンテキスト式: 使用する接続コンテキスト・インスタンスの指定のために評価されます。
- 実行コンテキスト式: 使用する実行コンテキスト・インスタンスの指定のために評価されます。
- 結果式: 結果を受け取ります（たとえば、ストアド・ファンクションから）。
- ホスト式

ISO 標準コード生成（`-codegen=iso` 設定）では、式が評価される順序に依存する副作用を使用しても Java 式の評価は明確です。このような副作用の例は、3-21 ページの「[Java 式の実行時評価の例（ISO コード生成）](#)」を参照してください。

Oracle 固有コード生成（デフォルトの `-codegen=oracle` 設定）では、副作用がない場合、Java 式の評価は ISO 標準に従います（`-bind-by-identifier` オプションが使用可能な場合を除く）。副作用がある場合、Java 式の評価は実装ごとに固有で、変更される可能性があります。

重要： 次の説明および関連する例は、Oracle 固有コード生成には適用されません。ここに説明されているような副作用を使用する場合は、変換で ISO コードを生成する必要があります。

次に、ISO コードの場合の実行時に実行される文に対する Java 式の評価、実行および代入の順序の概要を示します。

1. 接続コンテキスト式がある場合は、他の Java 式が評価される前に、即座に評価されます。
2. 実行コンテキスト式がある場合は、接続コンテキスト式の後で、結果式の前に評価されます。
3. 結果式がある場合は、コンテキスト式の後で、ホスト式の前に評価されます。
4. コンテキスト式や結果式の評価後に、ホスト式は、SQL 操作での出現順に従って左から右へ評価されます。各ホスト式が出現してそれが評価されると、その値が保存され SQL に渡されます。

各ホスト式の評価は 1 度のみです。

5. IN および INOUT パラメータが SQL に渡され、SQL 操作が実行されます。
6. SQL 操作の実行後、出力パラメータ、つまり Java の OUT および INOUT ホスト式に、SQL 操作の結果の出力内容が代入されます。このとき、出力パラメータが SQL 操作で出現する順序に従って、左から右に代入が行われます。
各出力ホスト式は 1 度のみ代入されます。
7. 結果式がある場合は、最後に出力内容が代入されます。

評価順序の例、要点および特別な考慮点については、3-21 ページの「[Java 式の実行時評価の例 \(ISO コード生成\)](#)」を参照してください。

注意： PL/SQL ブロック内のホスト式はすべて、ブロック内の文が実行される前に、一度に評価されます。ホスト式は、ブロック内のコントロール・フローにかかわらず、出現順に評価されます。

文中の式が評価された後、入力および入出力ホスト式が SQL に渡され、SQL 操作が実行されます。SQL 操作の実行後、Java の出力ホスト式、入出力ホスト式および結果式に対して、次のように代入が行われます。

1. OUT および INOUT ホスト式は、左から右の順に出力内容が代入されます。
2. 結果式がある場合は、最後に出力内容が代入されます。

実行時には、同じ名前を共有するかどうか、または同じオブジェクトを参照するかどうかにかかわらず、すべてのホスト式が別々の値として扱われます。それぞれの文の実行はリモート・メソッドとして扱われ、各ホスト式は別々のパラメータとして使用されます。各入力または入出力パラメータは、その文に対して代入が行われる前に、最初の出現時に評価され渡されます。また、各出力パラメータも別々のものとして扱われ、1 度のみ代入が行われます。

各ホスト式の評価が 1 度のみであることにも注意してください。INOUT 式は最初に出現したときに評価されます。出力の代入時に、式そのものは再評価されず、副作用も繰り返されません。

Java 式の実行時評価の例 (ISO コード生成)

ここでは、ISO コード生成 (-codegen=iso) でのアプリケーション実行時の Java 式の評価方法について、いくつかの例を示して説明します。Oracle 固有コード生成 (デフォルトの -codegen=oracle 設定) を使用する場合は、これらの作用は適用されません。これらの作用を利用する場合は、変換時に ISO コード生成を要求してください。

ここには多数の例が含まれています。例で使用している SELECT INTO 文は 3-30 ページの「[単一行の問合せ結果: SELECT INTO 文](#)」で、代入文は 3-53 ページの「[代入文 \(SET\)](#)」で、ストアド・プロシージャおよびストアド・ファンクションのコールは 3-54 ページの「[ストアド・プロシージャおよびストアド・ファンクションのコール](#)」で説明しています。

評価の前に実行される前置演算子と評価の後に実行される後置演算子

Java 式に Java の後置の増分演算子または減分演算子が含まれる場合、式が評価された後に、増分または減分が行われます。同様に、Java 式に Java の前置の増分演算子または減分演算子が含まれる場合、式が評価される前に、増分または減分が行われます。

この動作は、標準の Java コードでのこれらの演算子の処理方法と同じです。

次に示す例で確認してください。

例 1: 後置演算子

```
int indx = 1;
...
#sql { ... :OUT (array[indx]) ... :IN (indx++) ... };
```

この例では、式が次のように評価されます。

```
#sql { ... :OUT (array[1]) ... :IN (1) ... };
```

変数 `indx` は 2 に増分され、`:IN (indx++)` が評価された後で次に出現するときはこの値になります。

例 2: 後置演算子

```
int indx = 1;
...
#sql { ... :OUT (array[indx++]) ... :IN (indx++) ... };
```

この例では、式が次のように評価されます。

```
#sql { ... :OUT (array[1]) ... :IN (2) ... };
```

変数 `indx` は、最初の式が評価された後、2 番目の式が評価される前に、2 に増分されます。この変数は、2 番目の式が評価されると 3 に増分され、次に出現するときはこの値になります。

例 3: 前置演算子および後置演算子

```
int indx = 1;
...
#sql { ... :OUT (array[++indx]) ... :IN (indx++) ... };
```

この例では、式が次のように評価されます。

```
#sql { ... :OUT (array[2]) ... :IN (2) ... };
```

変数 `indx` は、最初の式が評価される前に、2 に増分されます。この変数は、2 番目の式が評価されると 3 に増分され、次に出現するときはこの値になります。

例 4: 後置演算子

```
int grade = 0;
int count1 = 0;
...
#sql { SELECT count INTO :count1 FROM staff
      WHERE grade = :(grade++) OR grade = :grade };
```

この例では、式が次のように評価されます。

```
#sql { SELECT count INTO :count1 FROM staff
      WHERE grade = 0 OR grade = 1 };
```

変数 `grade` は、`:(grade++)` が評価された後、1 に増分され、`:grade` が評価される時にはこの値になります。

例 5: 後置演算子

```
int count = 1;
int[] x = new int[10];
int[] y = new int[10];
int[] z = new int[10];
...
#sql { SET :(z[count++]) = :(x[count++]) + :(y[count++]) };
```

この例では、式が次のように評価されます。

```
#sql { SET :(z[1]) = :(x[2]) + :(y[3]) };
```

変数 `count` は、最初の式が評価された後、2 番目の式が評価される前に、2 に増分されます。また、2 番目の式が評価された後、3 番目の式が評価される前に 3 に増分されます。3 番目の式が評価されると 4 に増分され、次に出現するときにはこの値になります。

例 6: 後置演算子

```
int[] arr = {3, 4, 5};
int i = 0;
...
#sql { BEGIN
      :OUT (arr[i++]) := :(arr[i]);
      END };
```

この例では、式が次のように評価されます。

```
#sql { BEGIN
      :OUT (a[0]) := :(a[1]);
      END };
```

変数 `i` は、最初の式が評価された後、2 番目の式が評価される前に、1 に増分されます。このため、出力が `arr[0]` に代入されます。つまり、`arr[0]` に `arr[1]` の値、つまり 4 が代入されます。この文の実行後、配列 `arr` の値は {4, 4, 5} になります。

評価の順序に影響を与えない IN、INOUT、OUT

ホスト式は左から右に評価されます。式が IN、INOUT または OUT のいずれであるかは、評価の順序に影響しません。評価の順序を決めるのは、式が置かれる位置です。

例 7: IN、INOUT、OUT の比較

```
int[5] array;
int n = 0;
...
#sql { SET :OUT (array[n]) = :(++n) };
```

この例では、式が次のように評価されます。

```
#sql { SET :OUT (array[0]) = 1 };
```

入力式が出力式の前に評価されることはありません。式 `:OUT (array[n])` は、最左端にあるため最初に評価されます。次に、`n` は前置演算子で操作されるため、`++n` の評価の前に増分されます。次に、`++n` が 1 と評価されます。結果は `array[1]` ではなく `array[0]` に代入されます。これは、`n` が最初に出現したときの値が 0 (ゼロ) であったためです。

文の実行前に評価される PL/SQL ブロック内の式

PL/SQL ブロック内のホスト式はすべて、いずれかが実行される前に、一度に評価されます。

例 8: PL/SQL ブロック内の式の評価

```
int x=3;
int z=5;
...
#sql { BEGIN :OUT x := 10; :OUT z := :x; END };
System.out.println("x=" + x + ", z=" + z);
```

この例では、式が次のように評価されます。

```
#sql { BEGIN :OUT x := 10; :OUT z := 3; END };
```

このため、出力は「x=10,z=3」となります。

PL/SQL ブロック内の式はすべて、いずれかが実行される前に評価されます。この例では、2 番目の文のホスト式、つまり `:OUT z` および `:x` が、最初の文が実行される前に評価されます。2 番目の文が評価されるのは、`x` が元の値の 3 である場合で、式が評価された後にこの `x` に値 10 が代入されます。

例 9: PL/SQL ブロック内の式の評価（後置演算子がある場合）

ここでは、PL/SQL ブロック内で式が評価される方法について、例を挙げて説明します。

```
int x=1, y=4, z=3;
...
#sql { BEGIN
      :OUT x := :(y++) + 1;
      :OUT z := :x;
    END };
```

この例では、式が次のように評価されます。

```
#sql { BEGIN
      :OUT x := 4 + 1;
      :OUT z := 1;
    END };
```

後置増分演算子は、`:(y++)` が評価された後に実行されます。このため、式は `y` の初期値である 4 と評価されます。2 番目の文である `:OUT z := :x` は、最初の文が実行される前に評価されます。このため、`x` は初期化された値 1 のままになります。このブロックの実行後、`x` の値が 5 に、`z` の値が 1 になります。

例 10: 1 つのブロック内の文と個別の SQLJ 実行文

ここでは、1 つの SQLJ 実行文の PL/SQL ブロックに出現する 2 つの文と、個別の（連続した）SQLJ 実行文に出現する同じ 2 つの文との違いについて説明します。

まず、PL/SQL ブロック内に 2 つの文がある場合を考えます。

```
int y=1;
...
#sql { BEGIN :OUT y := :y + 1; :OUT x := :y + 1; END };
```

この例では、式が次のように評価されます。

```
#sql { BEGIN :OUT y := 1 + 1; :OUT x := 1 + 1; END };
```

2 番目の文の `:y` は、いずれかの文が実行される前に評価されます。このため、`y` は最初の文からその出力をまだ受け取っていません。このブロックの実行後、`x` と `y` の両方の値が 2 になります。

次に、個別の SQLJ 実行文の PL/SQL ブロック内に、前の例と同じ 2 つの文がある場合を考えます。

```
int y=1;
#sql { BEGIN :OUT y := :y + 1; END };
#sql { BEGIN :OUT x := :y + 1; END };
```

最初の文は次のように評価されます。

```
#sql { BEGIN :OUT y := 1 + 1; END };
```

この式が実行された後で、**y** に値 **2** が代入されます。

最初の文の実行後、2 番目の文が次のように評価されます。

```
#sql { BEGIN :OUT x := 2 + 1; END };
```

この場合、前述の PL/SQL ブロックの例とは異なり、前の文の実行により **y** はすでに値 **2** を受け取っています。このため、2 番目の文の実行後、**x** に値 **3** が代入されます。

常に 1 度のみ評価される PL/SQL ブロック内の文

ホスト式の評価は、プログラム・フローやプログラム・ロジックにかかわらず、1 度のみです。

例 11: ループでのホスト式の評価

```
int count = 0;
...
#sql {
  DECLARE
    n NUMBER
  BEGIN
    n := 1;
    WHILE n <= 100 LOOP
      :IN (count++);
      n := n + 1;
    END LOOP;
  END
};
```

Java 変数 **count** が SQL に渡されるときは値は **0** (ゼロ) になります。これは、この変数が前置演算子ではなく後置演算子で操作されるためです。その後、値は **1** に増分され、この PL/SQL ブロックの実行中はその値を維持します。この変数は、SQLJ 実行文の解析時に 1 度のみ評価され、SQL の実行前に値 **1** で置き換えられます。

例 12: 条件ブロック内のホスト式の評価

この例では、プログラム・フローに左右されずに式が評価される方法について説明します。ブロックの実行では、IF...THEN...ELSE 構文の 1 つの分岐のみが実行可能です。ただし、ブロックの実行前に、そのブロック内のすべての式が文の出現順序に従って評価されます。

```
int x;  
...  
(operations on x)  
...  
#sql {  
    DECLARE  
        n NUMBER  
    BEGIN  
        n := :x;  
        IF n < 10 THEN  
            n := : (x++);  
        ELSE  
            n := :x * :x;  
        END LOOP;  
    END  
};
```

x に対して実行された操作の結果、x の値が 15 になったとします。PL/SQL ブロックの実行時には、ELSE 分岐は実行されますが IF ブランチは実行されません。ただし、PL/SQL ブロック内の式はすべて、プログラム・ロジックやプログラム・フローにかかわらず、実行前に評価されます。このため、x++ が評価されてから x が増分され、次にそれぞれの x が (x * x) 式で評価されます。このため、IF...THEN...ELSE ブロックは次のように評価されます。

```
IF n < 10 THEN  
    n := 15;  
ELSE  
    n := :16 * :16;  
END LOOP;
```

x の初期値が 15 として、このブロックの実行後 n の値が 256 になります。

結果式の前に左から右へ代入が行われる出力ホスト式

前述のように、OUT および INOUT ホスト式がその出現順に従って左から右に代入され、結果式がある場合は最後に代入されます。同じ変数に対して複数回代入が行われた場合は、この順序に従って、最後に代入された内容で上書きされます。

注意： 一部の例では、ストアド・プロシージャおよびストアド・ファンクションのコールを使用しています。これらの構文については、3-54 ページの「[ストアド・プロシージャおよびストアド・ファンクションのコール](#)」で説明します。

例 13: 同じ変数を参照する複数の出力ホスト式

```
#sql { CALL foo(:OUT x, :OUT x) };
```

foo() が、値 2 および 3 を出力する場合、SQLJ 実行文の実行が終了すると、x の値は 3 になります。右端の代入が最後に行われるため、その値が優先されます。

例 14: 同じオブジェクトを参照する複数の出力ホスト式

```
MyClass x = new MyClass();
MyClass y = x;
...
#sql { ... :OUT (x.field):=1 ... :OUT (y.field):=2 ... };
```

SQLJ 実行文の実行後、x.field の値は、1 ではなく 2 になります。これは、x が y と同じオブジェクトであり、field にまず値 1 が代入された後、値 2 が代入されたためです。

例 15: ホスト式の代入に優先される結果の代入

この例は、ファンクションの出力結果が結果式に代入される場合と、結果が OUT ホスト式に代入される場合の相違を示します。

次に、入力パラメータ invar、出力パラメータ outvar および戻り値を使用するファンクションを想定します。

```
CREATE FUNCTION fn(invar NUMBER, outvar OUT NUMBER)
RETURN NUMBER AS BEGIN
    outvar := invar + invar;
    return (invar * invar);
END fn;
```

次に、ファンクションの出力結果が結果式に代入される例を考えてみます。

```
int x = 3;
#sql x = { VALUES(fn(:x, :OUT x)) };
```

このファンクションは、入力パラメータとして 3 をとり、出力パラメータとして 6 を代入し、また戻り値として 9 を戻します。実行後、まず `:OUT x` に対して代入が行われ、`x` の値が 6 になります。しかし、最後には結果式に代入が行われ、`x` に代入されていた値 6 が、戻り値の 9 で上書きされます。このため、`x` が次に出現するとき、その値は 9 になります。

次の例では、ファンクションの出力結果が、結果式ではなく OUT ホスト変数に代入されます。

```
int x = 3;
#sql { BEGIN :OUT x := fn(:x, :OUT x); END };
```

この場合は結果式がなく、OUT 変数は単純に左から右へ代入されます。実行後、数式の左側にある最初の `:OUT x` に代入が行われ、`x` の値がファンクション戻り値の 9 になります。ただし、左から右へ代入が進むと、数式の右側にある 2 番目の `:OUT x` に最後の代入が行われ、`x` が出力値の 6 になります。`x` にすでに代入されていた値 9 は、値 6 で上書きされます。このため、`x` が次に出現するとき、その値は 6 になります。

注意： ここでは、ホスト式の評価方法の概念を説明するために、通常では使用しない例をいくつか示しています。実際のコードでは、1 つの文または PL/SQL ブロック内で、OUT と INOUT または IN ホスト式に同じ変数を使用することはお勧めしません。このように記述した場合の動作は、Oracle SQLJ では正しく定義されていますが、SQLJ 仕様では扱われていません。このため、この方法で記述されたコードは移植できません。セマンティクス・チェックの際に `portable` フラグに設定されていると、このようなコードに対して Oracle SQLJ トランスレータからの警告が生成されます。

ホスト式の制限事項

ホスト式で `in`、`out` および `inout` を識別子として使用する場合は、必ずカッコで囲みます。カッコで囲まないと、モード指定子と間違われる可能性があります。ホスト式の識別子には、大文字と小文字との区別がありません。

たとえば、`in` という入力ホスト変数を次のように指定したとします。

```
:(in)
```

または

```
:IN(in)
```

単一行の問合せ結果 : SELECT INTO 文

戻り値がデータ 1 行のみの場合、SQLJ では、選択項目を SQL 構文内の Java ホスト式に直接代入できます。これは、SELECT INTO 文を使用して行います。

SELECT INTO 構文

SELECT INTO 文の構文を次に示します。

```
#sql { SELECT expression1,..., expressionN INTO :host_exp1,..., :host_expN
        FROM table <optional_clauses> };
```

上の構文を詳しく説明すると次のようになります。

- *expression1* から *expressionN* までは、データベースから選択するデータを指定する式です。すべての SELECT 文に有効な式を使用します。この式のリストは、SELECT リストと呼ばれます。

単純な場合は、データベース表の列名になります。

SELECT リストにはホスト式も記述できます。後述の「[SELECT INTO 文の例](#)」を参照してください。

- *host_exp1* から *host_expN* までは、変数や配列要素などのターゲット・ホスト式です。このホスト式のリストは、INTO リストと呼ばれます。
- *table* は、データを選択するデータベースの表、ビューまたはスナップショットの名前です。
- *optional_clauses* は、WHERE 句など、SELECT 文に追加する任意の句です。

SELECT INTO 文の戻り値は、必ず 1 行のデータのみとします。それ以外の場合は、実行時にエラーが発生します。

デフォルトでは、INTO リストのホスト式は OUT ですが、これを明示的に指定することも可能です。

```
#sql { SELECT column_name1, column_name2 INTO :OUT host_exp1, :OUT host_exp2
        FROM table WHERE condition };
```

INTO リストで IN または INOUT トークンを使用すると、変換時にエラーが発生します。

注意：

- *expression1* から *expressionN*、*table* およびオプションの句で
使用できる構文は、SQL SELECT 文の場合と同じです。Oracle SQL で
使用可能な構文などの詳細は、『Oracle9i SQL リファレンス』を参照
してください。
- 任意の数の SELECT リスト項目および INTO リスト項目を使用できま
す。ただし、SELECT リスト項目ごとに、互換性のある型の INTO リ
スト項目を使用するようにします。

SELECT INTO 文の例

この例では、次の行を持つ従業員表 EMP を使用します。

```
CREATE TABLE EMP (
  EMPNO NUMBER(4),
  ENAME VARCHAR2(10),
  HIREDATE DATE );
```

最初に、INTO リストにホスト式 1 つを使用した SELECT INTO 文の例を示します。

```
String empname;
#sql { SELECT ename INTO :empname FROM emp WHERE empno=28959 };
```

次に、INTO リストに複数のホスト式を使用した SELECT INTO 文の例を示します。

```
String empname;
Date hdate;
#sql { SELECT ename, hiredate INTO :empname, :hdate FROM emp
      WHERE empno=28959 };
```

SELECT リストにホスト式を使用する例

INTO リストと同様に、SELECT リストにも Java ホスト式を使用できます。

たとえば、1 つのホスト式から他のホスト式に直接データを取込むことも可能です。(ただし、実際にこの処理を行うことはあまりありません。)

```
...
#sql { SELECT :name1 INTO :name2 FROM emp WHERE empno=28959 };
...
```

次の例のように、選択されたデータに対して操作または連結を行う方がより現実的です。ここでは、必要に応じて Java 変数がすでに宣言および代入されていることを前提としています。

```
...
#sql { SELECT sal + :raise INTO :newsal FROM emp WHERE empno=28959 };
...

...
#sql { SELECT :(firstname + " ") || emp_last_name INTO :name FROM myemp
        WHERE empno=28959 };
...
```

2 番目の例では、MYEMP という表を想定します。この表は、EMP 表と類似していますが、列名には ENAME 列でなく EMP_LAST_NAME 列を使用しています。この SELECT 文では、Java ホスト式と Java の文字列連結 (+ 演算子) を使用して、firstname を「」(空白 1 つ) と連結しています。この結果は SQL エンジンに渡されます。SQL エン진은、SQL の文字列連結 (|| 演算子) を使用して、最後の名前に追加します。

SELECT INTO エラー状態

SELECT INTO 文は、1 行のデータのみを戻す問合せ用です。

0 (ゼロ) 行または複数の行を検索すると、SELECT INTO 問合せは例外を戻します。

- 0 (ゼロ) 行を検索している SELECT INTO は、「no data」状態を表す 2000 という SQL 状態で例外を戻します。
- 複数行を検索している SELECT INTO は、制約違反を表す 21000 という SQL 状態で例外を戻します。

これらの例外は、B-44 ページの「[SQLJ ランタイム・メッセージ](#)」に表示されています。java.sql.SQLException クラスの getSQLState() メソッドによって、SQL 状態を取得できます。4-22 ページの「[SQL の状態およびエラー・コードの取出し](#)」を参照してください。

これは、ISO SQLJ 規格で指定されたベンダーに依存しない動作です。この場合、ベンダー固有エラー・コードはありません。エラー・コードは常に 0 (ゼロ) になります。

複数行の問合せ結果 : SQLJ イテレータ

多数の SQL 操作は複数行の問合せです。SQLJ で複数行の問合せ結果を処理するには、SQLJ イテレータが必要です。これは、強い型指定の JDBC 結果セットであり、基になるデータベース・カーソルに対応付けられています。最初に SQLJ イテレータが使用され、SELECT 文からの問合せ結果を取得します。

Oracle SQLJ では、次の目的で SQLJ イテレータと結果セットを使用できます。

- SQL 実行文の OUT ホスト変数として使用
- SELECT INTO 文などの INTO リストのターゲットとして使用
- ストアド・ファンクション・コールからの戻り型として使用
- イテレータ宣言の列型として使用（基本的にはネストされたイテレータ）

注意： 前述の目的で SQLJ イテレータを使用するには、そのクラスが `public` として宣言されていることが必要です。クラスレベルまたはネストされたクラスレベルで宣言したイテレータは、`public static` として宣言することをお勧めします。

ストアド・ファンクション戻り値の使用の詳細は、ストアド・プロシージャおよびストアド・ファンクションの説明の後の 3-57 ページの「[ストアド・ファンクションの戻り値としてのイテレータおよび結果セットの使用](#)」を参照してください。その他の使用方法については、この項で説明します。

イテレータの詳細は、7-34 ページの「[イテレータ・クラスの実装と拡張機能](#)」を参照してください。ここでは、イテレータ・クラスの実装方法、および JDBC との連係動作やイテレータのサブクラス化などの高度な機能について説明します。

イテレータの概念

SQLJ イテレータ宣言を使用すると、強い型指定のイテレータになります。詳細は、3-2 ページの「[SQLJ 宣言の概要](#)」を参照してください。これは、一般的なイテレータの使用方法で、SQLJ セマンティクス・チェック機能のメリットを変換時に活用します。

また、弱い型指定のイテレータを使用することも可能であり、利便性が高いこともあります。弱い型指定のイテレータを使用するために、インスタンス化できる汎用クラスがあります。

ここでは、強い型指定のイテレータ機能について主に説明しますが、弱い型指定のイテレータも簡単に説明します。

強い型指定のイテレータの概要

強い型指定のイテレータ・オブジェクトを使用する前に、イテレータ・クラスを宣言する必要があります。イテレータ宣言には、SQLJ で生成される Java クラスを指定し、このクラス属性にはデータ列の型（および必要に応じて名前）を定義します。

SQLJ イテレータ・オブジェクトは、そのように具体的に宣言されたイテレータ・クラスがインスタンス化されたもので、事前定義された型の固定数の列を持ちます。これに対して、JDBC 結果セット・オブジェクトは、標準の `java.sql.ResultSet` のインスタンスで、原則的には任意の型の列をいくつも持つことが可能です。

イテレータを宣言する際には、選択された列のデータ型のみを指定するか、選択された列のデータ型と名前の両方を指定します。

- 名前とデータ型を指定すると、名前指定イテレータ・クラスが定義されます。
- データ型のみを指定すると、位置指定イテレータ・クラスが定義されます。

宣言するデータ型（および名前）により、そのクラスからインスタンス化するイテレータ・オブジェクトへの問合せ結果の格納方法が決まります。イテレータ・オブジェクトに取り込まれた SQL データは、イテレータ宣言で指定された Java 型に変換されます。

問合せを行い、名前指定イテレータ・オブジェクトにその結果を取り込む場合は、SELECT フィールドの名前やデータ型が、イテレータの列名や列型と一致する必要があります（ただし、大文字と小文字の区別はありません）。SELECT フィールドはどんな順序でもかまいませんが、SELECT フィールドとイテレータ列とを同じ名前にすることは重要です。最も単純な例では、データベース列とイテレータ列とを同じ名前にすることです。たとえば、データベース表の `ENAME` 列からのデータの選択や、イテレータ `ename` 列への取込みができるようにしてください。また、データベースの列名とイテレータの列名とが異なる場合には、別名を使用して列名どうしをマッピングできるようにする必要があります。より複雑な問合せでは、2 つの列の間で操作を実行し、その結果取得された列名が、対応するイテレータの列名と同じになるように変更できることが重要です。（イテレータ列名に関するこれら 2 つの事例は、3-40 ページの「[名前指定イテレータのインスタンス化と移入](#)」を参照してください。）

SQLJ イテレータは厳密に分類されているため、SQLJ のセマンティクス・チェック・フェーズで、Java の型チェックを実行できます。

ここで示す例は、次のように定義された表を前提とします。

```
CREATE TABLE EMP_SAL (
    EMPNO NUMBER(4),
    ENAME VARCHAR2(10),
    OLDSAL NUMBER(10),
    RAISE NUMBER(10) );
```

この表を使用すると、名前指定イテレータを宣言して使用できます。

次のように宣言します。

```
#sql iterator SalNamedIter (int empno, String ename, float raise);
```

実行可能コードを示します。

```
class MyClass {
    void func() throws SQLException {
        ...
        SalNamedIter niter;
        #sql niter = { SELECT ename, empno, raise FROM empсал };

        ... process niter ...
    }
}
```

前述の例は、イテレータの列名と表の列名を同じにした場合の簡単な例です。名前指定イテレータを使用した場合は、データは位置でなく名前で照合されるので、SELECT 文中にある項目はどのような順序でもかまいません。

問合せを行い、位置指定イテレータ・オブジェクトにその結果を取り込む場合は、列の選択順に従ってデータが取り込まれます。データベース表から最初に選択された列のデータはイテレータの最初の列に、2 番目に選択された列のデータはイテレータの 2 番目の列に、順を追って配置されます。表の列のデータ型は、イテレータ列のデータ型に変換可能である必要がありますが、イテレータ列には名前がないため、データベース列はどのような名前でもかまいません。

前述の EMPСАL 表では、次のように位置指定イテレータを宣言し、使用できます。

次のように宣言します。

```
#sql iterator SalPosIter (int, String, float);
```

実行可能コードを示します。

```
class MyClass {
    void func() throws SQLException {
        ...
        SalPosIter piter;
        #sql piter = { SELECT empno, ename, raise FROM empсал };

        ... process piter ...
    }
}
```

SELECT 文でのデータ項目の順序は、イテレータでの順序と同じです。

名前指定イテレータと位置指定イテレータとでは処理が異なります。詳細は、3-41 ページの「[名前指定イテレータへのアクセス](#)」および 3-44 ページの「[位置指定イテレータへのアクセス](#)」を参照してください。

イテレータの注意事項 前のコンセプトで述べたことの他に、イテレータ全般に関しては次のような注意事項に留意してください。

- イテレータにデータを取り込む場合に、`SELECT *` 構文を使用することはお薦めしません。位置指定イテレータにこの構文を使用する場合、表の列数とイテレータの列数が一致し、列型も一致する必要があります。また、名前指定イテレータにこの構文を使用する場合は、表の列数をイテレータの列数と同じかそれより多くなるようにし、各イテレータ列の名前と型をデータベース表と同じにする必要があります。トランスレータの `-warn=nostrict` フラグが設定されていないと、表の列数がイテレータの列数より多い場合には、警告が出されます。このフラグの詳細は、8-44 ページの「[トランスレータからの警告 \(-warn\)](#)」を参照してください。
- 位置指定イテレータと名前指定イテレータは、互換性のない別々の Java クラスです。ある種類のイテレータ・オブジェクトから別の種類のイテレータ・オブジェクトへのキャストはできません。
- SQL カーソルとは異なり、イテレータ・インスタンスは（たとえばメソッド・パラメータとして受け渡すことが可能な）ファーストクラスの Java オブジェクトで、`public` や `private` などの Java クラス修飾子を使用して宣言できます。
- SQLJ では、SQLJ イテレータと JDBC 結果セットとの係性および変換がサポートされています。詳細は、7-53 ページの「[SQLJ イテレータと JDBC 結果セットの係動作](#)」を参照してください。
- 一般に、イテレータを設定した `SELECT` 文が実行された時点でのデータベースの状態のみによって、そのイテレータの内容が決まります。それ以降に `UPDATE`、`INSERT`、`DELETE`、`COMMIT`、`ROLLBACK` などの操作を行っても、イテレータやその内容には反映されません。詳細は、4-27 ページの「[イテレータおよび結果セットに対するコミットおよびロールバックの影響](#)」を参照してください。

これに対する例外は、イテレータがスクロール可能で、データの変更に対して「sensitive」とであると宣言する場合です。7-38 ページの「[スクロール可能なイテレータの宣言](#)」および 7-38 ページの「[スクロール可能なイテレータの更新検出](#)」を参照してください。

弱い型指定のイテレータの概要

イテレータ・クラスを宣言したくないような場合、Oracle SQLJ では弱い型指定のイテレータを使用できます。このようなイテレータを結果セット・イテレータと呼びます。簡単な（スクロール不可能）結果セット・イテレータを使用するには、

`sqlj.runtime.ResultSetIterator` クラスをインスタンス化します。スクロール可能な結果セット・イテレータを使用するには、

`sqlj.runtime.ScrollableResultSetIterator` クラスをインスタンス化します。（スクロール可能なイテレータの詳細は、7-38 ページの「[スクロール可能なイテレータ](#)」を参照してください。）

結果セット・イテレータを使用する場合のデメリットは、強い型指定のイテレータと比較すると、SQLJ では問合せへのセマンティクス・チェックを同じように実行できないことです。

詳細は、7-37 ページの「[結果セット・イテレータ](#)」を参照してください。

イテレータの使用手順

SQLJ 名前指定または位置指定イテレータを使用する場合にも、次の 5 つの手順に従ってください。

1. SQLJ 宣言を使用して、イテレータ・クラス、つまりイテレータの種類を定義します。
2. イテレータ・クラスの変数を宣言します。
3. SELECT 文を使用した後、イテレータ変数に SQL 問合せ結果を取り込みます。
4. イテレータの問合せ列にアクセスします。後述のように、名前指定イテレータと位置指定イテレータではアクセス方法が異なります。
5. 問合せ結果の処理が終了した後、イテレータを閉じてそのリソースを解放します。

名前指定イテレータ、位置指定イテレータ、結果セット・イテレータの比較

名前指定イテレータと位置指定イテレータには、それぞれのメリットがあり、使用目的も異なります。

名前指定イテレータは、より柔軟に使用できます。選択されたデータを名前指定イテレータに取り込むときは、SELECT フィールドとイテレータ列との整合チェックは名前で判定されるため、問合せの順序は判定基準にはなりません。この場合、データが誤った列に取り込まれることがないため、エラーが少なくなります。名前が一致しない場合は、データベースに対して SQL 文をチェックするときに、SQLJ トランスレータがエラーを生成します。

位置指定イテレータの場合は、他の埋込み SQL 言語と同様のパラダイムや構文を使用できます。たとえば、名前指定イテレータではデータの取込みに `next()` メソッドを使用しますが、位置指定イテレータでは Pro*C の場合と似た `FETCH INTO` 構文を使用します。各フェッチが暗黙的にイテレータを次の行へ進め、その後で次の値が取り出されます。

また、位置指定イテレータの場合は、イテレータ列に選択するデータを名前ではなく位置によって識別するため、柔軟性に欠けます。SELECT 文では、項目の順序に注意する必要がありますからです。また、イテレータのすべての列にデータを選択する必要もあります。このとき、表で選択されている列のデータ型に偶然一致した他のイテレータ列に、データが誤って書き込まれる可能性もあります。

位置指定イテレータでは、個々のデータ要素へのアクセスも不便です。名前指定イテレータではデータを名前に基づいて格納するため、各列に対し、便利なアクセッサ・メソッドを使用できます。たとえば、`ename` イテレータ列からのデータの取出しには、`ename()` メソッドを使用できます。これに対して、位置指定イテレータでは、`FETCH INTO` 文でデータを直接 Java ホスト式にフェッチします。また、ホスト式を正しい順序で記述することが必要です。

最後に、問合せに対して強い型指定のイテレータ・クラスを宣言したくない場合は、弱い型指定の結果セット・イテレータを使用する方法を選択できます。結果セット・イテレータは、JDBC コードから SQLJ コードへの変換時に最も便利です。この考慮事項は、結果セット・イテレータ (ResultSetIterator インスタンスまたは ScrollableResultSetIterator インスタンス) では、変換時に完全な SQLJ セマンティクス・チェックができないという点と比較する必要があります。名前指定イテレータまたは位置指定イテレータの場合、SQLJ によって、SELECT リスト型が生成されたデータの Java 型と一致していることが検証されます。結果セット・イテレータの場合は、このことは検証できません。詳細は、7-37 ページの「[結果セット・イテレータ](#)」を参照してください。

イテレータの比較についての注意 SQLJ イテレータについて、次の点を注意してください。

- 位置指定イテレータを使用する場合、データベースから選択する列の数とイテレータの列の数を同じにすることが必要です。名前指定イテレータを使用した場合、通常はデータベースから選択できる列数が、イテレータの列数より少なくなることはありません。ただし、トランスレータ `-warn=nostrict` フラグに設定すると、イテレータの列数と同じかそれ未満にすることが可能になります。この場合、一致しない列は無視されます。このフラグの詳細は、8-44 ページの「[トランスレータからの警告 \(-warn\)](#)」を参照してください。
- フェッチの一般的な意味は、データベースからのデータのフェッチのことですが、位置指定イテレータに対して `FETCH INTO` 文を実行した場合は、サーバーへのラウンドトリップの要否が、行プリフェッチの値次第となります。これは、データベースではなくイテレータからデータをフェッチするためです。行プリフェッチの値が 1 であるときは、各フェッチのたびにデータベースへのトリップが 1 回行われます。(データベースへの 1 回のトリップで取り出される行の数は、行プリフェッチの値によって判断されます。詳細は、10-3 ページの「[行プリフェッチ](#)」を参照してください。)
- 結果セット・イテレータでは、位置指定イテレータと同じ `FETCH INTO` 構文が使用されています。また同じように、実行時に SELECT リストのサイズ (データ項目の数) と `FETCH` 文の代入されたデータである変数の数が一致する必要があるという制限を受けています。

名前指定イテレータの使用方法

名前指定イテレータ・クラスを宣言する場合は、イテレータの各列のデータ型と名前を宣言します。

データを名前指定イテレータに取り出す場合、SELECT フィールドとイテレータ列とを次の 2 つの面で同じにしておく必要があります。

- 各 SELECT フィールドの名前 (表の列名または別名) が、イテレータ列と同じ名前である必要があります (大文字と小文字の区別はないため、`ename` と `ENAME` の場合は同じ名前であると認識されます)。
- 標準の JDBC 型マッピングで、各イテレータ列の型が、対応する SELECT フィールドのデータ型との互換性を持つ必要があります。

名前指定イテレータ・クラスの宣言では、属性の宣言はどのような順序で行ってもかまいません。データは、名前に基づいてイテレータに選択されます。

名前指定イテレータの場合は、`next()` メソッドでデータを行ごとに取り出し、それぞれの列のアクセッサ・メソッドで個々のデータ項目を取り込みます。アクセッサ・メソッド名は列名と同じです。（ほとんどの Java のアクセッサ・メソッド名とは異なり、名前指定イテレータの場合、アクセッサ・メソッド名が「`get`」で始まることはありません。）たとえば、`sal` 列を持つ名前指定イテレータ・オブジェクトの場合は、使用するアクセッサ・メソッドが `sal()` という名前になります。

注意： 名前指定イテレータの列のネーミングには、次の制約事項が適用されます。

- 列名に Java の予約語は使用できません。
 - 列名には、`next()`、`close()`、`getResultSet()` および `isClosed()` など、名前指定イテレータ・クラスのユーティリティ・メソッドと同じ名前は使用できません。スクロール可能な名前指定イテレータの場合、`previous()`、`first()` および `last()` などのメソッドも使用できません。（7-39 ページの「[スクロール可能なインタフェース](#)」および 7-40 ページの「[スクロール可能な名前指定イテレータ](#)」を参照してください。）
-
-

名前指定イテレータ・クラスの宣言

名前指定イテレータ・クラスの宣言には、次の構文を使用します。

```
#sql <modifiers> iterator classname <implements clause> <with clause>
      ( type-name-list );
```

この構文の `modifiers` は正当な Java クラス修飾子の順序で、その使用は任意です。`classname` はイテレータのクラス名です。また、`type-name-list` は、Java の型と名前のリストです。これは、データベース表の列型と列名に相当します。

`implements` 句と `with` 句の使用は任意です。前者は実装するインタフェースを、後者は定義および初期化する変数を指定します。これらについては、3-5 ページの「[IMPLEMENTS 句宣言](#)」および 3-6 ページの「[WITH 句宣言](#)」を参照してください。

ここでは、次のような表について考えてみます。

```
CREATE TABLE PROJECTS (
  ID NUMBER(4),
  PROJNAME VARCHAR(30),
  START_DATE DATE,
  DURATION NUMBER(3) );
```

この表で使用する名前指定イテレータは、次のように宣言します。

```
#sql public iterator ProjIter (String projname, int id, Date deadline);
```

この結果、アクセッサ・メソッド `projname()`、`id()` および `deadline()` でアクセス可能なデータ列を持つ、イテレータ・クラスが生成されます。

注意： 標準 Java の場合と同様に、Public クラスは、次のいずれかの方法で宣言する必要があります。Sun 社の JDK の標準 `javac` コンパイラを使用する場合は、これが要件となります。

- 別個のソース・ファイル中に宣言する方法。ファイルのベース名はクラス名と同じにします。

または

- `public static` 修飾子を使用して、クラスレベルの有効範囲またはネストされたクラスレベルの有効範囲に宣言する方法。
-

名前指定イテレータのインスタンス化と移入

前の項で例示した `ProjIter` イテレータ型の変数を宣言し、`SELECT` 文を使用して移入します。

前の項で定義した `PROJECTS` 表と `ProjIter` イテレータを引き続き使用します。この表には、イテレータの `id` 列と `projname` 列と同じ名前とデータ型の列がありますが、イテレータの `deadline` 列への移入操作を実行するためには別名を使用する必要があることに注意してください。次にその例を示します。

```
ProjIter projIter;
```

```
#sql projIter = { SELECT start_date + duration AS deadline, projname, id
                  FROM projects WHERE start_date + duration >= sysdate };
```

この例では、開始日に期間を加算してプロジェクトの最終期限を算出し、`deadline` イテレータ列と同じ `deadline` という名前をその結果に別名として付けます。また、`WHERE` 句を使用して、現行のシステムの日付より先の最終期限のみが処理されるようにしています。

ファンクション・コールを使用する場合も、別名を作成する必要があります。ファンクション `MAXIMUM()` があるとします。このファンクションでは、入力として `DURATION` エントリおよび整数を取り、その 2 つのうちどちらか大きい方の最大値を戻り値とします。たとえば、使用しているアプリケーションで 3 を入力した場合、プロジェクトが少なくとも 3 か月間は存続することになります。

ここでは、イテレータを次のように宣言したとします。

```
#sql public iterator ProjIter2 (String projname, int id, float duration);
```


問合せに使用する `MAXIMUM()` ファンクションには、次のように問合せ結果に対して別名を指定できます。

```
ProjIter2 projIter2;
```

```
#sql projIter2 = { SELECT id, projname, maximum(duration, 3) AS duration
                  FROM projects };
```

通常、正当な Java 識別子ではない名前、またはイテレータの列名とは異なる名前の付いた `SELECT` フィールドの問合せには、別名を使用してください。

前述のように、名前指定イテレータを使用した場合、通常はデータベースから選択できる列数が、イテレータの列数より少なくなることはありません。イテレータの列数を上回る数の列を選択することも不可能ではありません（一致しない列は無視されます）が、SQLJ の `-warn=nostrict` オプションが設定されていないと、警告が出されます。

名前指定イテレータへのアクセス

名前指定イテレータ・オブジェクトの `next()` メソッドを使用すると、イテレータに選択されたデータを 1 行ずつ取得できます。各列の各行にアクセスするには、SQLJ で生成されたアクセッサ・メソッドを使用します。通常、このメソッドは `while` ループ内で使用します。

`next()` がコールされると、次の処理が行われます。

- イテレータから取り出す行がまだ他にも存在するときは、`next()` のコールでその行を取り出し、`true` を戻り値とします。
- イテレータから取り出す行がもう存在しないときは、`next()` のコールで `false` を戻り値とします。

次に、名前指定イテレータのデータにアクセスする方法の例を示します。この例では、3-40 ページの「[名前指定イテレータのインスタンス化と移入](#)」の項で使用した宣言、インスタンス化および移入を再び使用します。

注意： 各イテレータには `close()` メソッドがあります。イテレータからのデータの取出しが終了した後、必ずこのメソッドをコールしてください。イテレータを閉じてリソースを解放するには、このコールが必要です。

次に示すようにイテレータ・クラスを宣言するとします。

```
#sql public iterator ProjIter (String projname, int id, Date deadline);
```

次に示すように、このイテレータ・クラスのインスタンスを定義してからアクセスします。

```
// Declare the iterator variable
ProjIter projIter;

// Instantiate and populate iterator; order of SELECT doesn't matter
#sql projIter = { SELECT start_date + duration AS deadline, projname, id
                  FROM projects WHERE start_date + duration >= sysdate };

// Process the results
while (projIter.next()) {
    System.out.println("Project name is " + projIter.projname());
    System.out.println("Project ID is " + projIter.id());
    System.out.println("Project deadline is " + projIter.deadline());
}

// Close the iterator
projIter.close();
...
```

前述に示したデータの取出しでは、`projname()`、`id()` および `deadline()` アクセッサ・メソッドが簡単に使用できます。`SELECT` 項目の順序やアクセッサ・メソッドを使用する順序は、あまり関係ありません。

ただし、アクセッサ・メソッド名は、イテレータ・クラスの宣言で大文字を使用した場合は大文字で、小文字を使用した場合は小文字で生成されます。コンパイル・エラーが発生する例を、次に示します。

次のように宣言します。

```
#sql iterator Cursor1 (String NAME);
```

実行可能コードを示します。

```
...
Cursor1 c1;
#sql c1 = { SELECT NAME FROM TABLE };
while (c1.next()) {
    System.out.println("The name is " + c1.name());
}
...
```

`Cursor1` クラスのメソッドは `NAME()` であり、`name()` ではありません。`System.out.println` 文では、`c1.NAME()` を使用する必要があります。

位置指定イテレータの使用

位置指定イテレータの宣言では、各列のデータ型は宣言しますが、列名は定義しません。SQL 問合せ結果の列が取り出される Java 型は、SQL データのデータ型との互換性があることが必要です。データベースの列や SELECT フィールドの名前は、あまり関係がありません。

名前が使用されないため、位置指定イテレータの Java 型を宣言する順序と、データを選択する順序が完全に一致する必要があります。

位置指定イテレータにデータが選択された後、位置指定イテレータからデータを取り出すには、FETCH INTO 文を使用します。さらに、endFetch() メソッドをコールして、データの終わりに到達したかどうかを確認します。(詳細は、3-44 ページの「[位置指定イテレータへのアクセス](#)」を参照してください。)

位置指定イテレータ・クラスの宣言

位置指定イテレータ・クラスの宣言には、次の構文を使用します。

```
#sql <modifiers> iterator classname <implements clause> <with clause>
    ( position-list );
```

この構文の *modifiers* は正当な Java クラス修飾子の順序で、その使用は任意です。また、*position-list* は、データベース表の列型に変換可能な Java 型のリストです。

implements 句と with 句の使用は任意です。前者は実装するインタフェースを、後者は定義および初期化する変数を指定します。これらは、3-5 ページの「[IMPLEMENTS 句宣言](#)」および 3-6 ページの「[WITH 句宣言](#)」を参照してください。

ここでは、次の行を持つ従業員表 EMP を想定します。

```
CREATE TABLE EMP (
    EMPNO NUMBER(4),
    ENAME VARCHAR2(10),
    SAL NUMBER(7,2) );
```

次のような位置指定イテレータを宣言します。

```
#sql public iterator EmpIter (String, int, float);
```

前述の例では、Java クラス EmpIter の定義に、匿名の String、int および float が使用されています。実際には、String 型が ENAME に対応し、int 型が EMPNO に対応しますが、この表とイテレータとでは、それぞれの列の順序が対応していないことに注意してください。イテレータ列の順序によって、データを選択する順序が決まります。詳細は、後述の「[位置指定イテレータのインスタンス化と移入](#)」を参照してください。

注意： 標準 Java の場合と同様に、Public クラスは、次のいずれかの方法で宣言する必要があります。Sun 社の JDK の標準 javac コンパイラを使用する場合は、これが要件となります。

- 別個のソース・ファイル中に宣言する方法。ファイルのベース名はクラス名と同じにします。

または

- `public static` 修飾子を使用して、クラスレベルの有効範囲またはネストされたクラスレベルの有効範囲に宣言する方法。
-

位置指定イテレータのインスタンス化と移入

前の項で例示した `EmpIter` 位置指定イテレータ型の変数を宣言し、`SELECT` 文を使用して移入します。

位置指定イテレータのインスタンス化と移入は、名前指定イテレータの場合と同じです。ただし、`SELECT` フィールドの順序が適切であるかどうかを確認する必要があります。

`EmpIter` イテレータ・クラスの 3 つのデータ型は、`EMP` 表の型と互換性があります。ただし、それぞれの順序は対応していないため、データの選択には注意が必要です。次の例は、「[位置指定イテレータ・クラスの宣言](#)」に記述されているように、`SELECT` フィールドがイテレータ列と同じ順序なので、正常に実行できます。

```
EmpIter empsIter;
```

```
#sql empsIter = { SELECT ename, empno, sal FROM emp };
```

前述のように、位置指定イテレータを使用する場合は、データベースから選択する列の数と、イテレータの列の数を同じにする必要があります。

位置指定イテレータへのアクセス

位置指定イテレータで定義された列にアクセスするには、SQL の `FETCH INTO` 構文を使用します。

コマンドの `INTO` 部分には、結果列を受け取る Java ホスト変数を指定します。ホスト変数の順序は、対応するイテレータ列の順序と同じにすることが必要です。`endFetch()` メソッドを使用して、最後のフェッチがデータの終わりに達したかどうかを確認します。このメソッドは、すべての位置指定イテレータ・クラスで使用できます。

注意：

- `endFetch()` メソッドの行のフェッチ開始前の初期戻り値は `true` です。行のフェッチが正常に行なわれると戻り値が `false` になり、すべての行のフェッチが終了した時点で戻り値が再び `true` になります。このため、`FETCH INTO` 文の後に、`endFetch()` テストを行う必要があります。`FETCH INTO` 文の前に `endFetch()` テストを行うと、行を取り出せなくなります。これは、最初の `FETCH` の前に `endFetch()` が `TRUE` になり、すぐに `while` ループから抜けてしまうためです。
- 結果が処理される前に `endFetch()` テストをしておく必要があります。これは、`FETCH` がデータの最後に到達しても、`SQL` 例外が送り返されないためです。この場合、単に次の `endFetch()` コールがトリガーされた後で戻り値が `true` になります。結果が処理される前に `endFetch()` テストをしておかないと、データの最後に到達した後に、コード中の最初の `FETCH` での戻り値 (`NULL` または無効なデータ) に対して処理が試みられます。
- 各イテレータには `close()` メソッドがあります。イテレータからのデータの取得が終了した後、必ずこのメソッドをコールしてください。イテレータを閉じてリソースを解放するには、このコールが必要です。

次の例では、前述の「[位置指定イテレータのインスタンス化と移入](#)」で使用した宣言、インスタンス化および移入を再び使用します。

`SELECT` 文中の `Java` ホスト変数の順序は、位置指定イテレータの列の順序と同じです。これは必須条件です。

最初に、次のようにイテレータ・クラスを宣言するとします。

```
#sql public iterator EmpIter (String, int, float);
```

次に示すように、このイテレータ・クラスのインスタンスを定義してからアクセスします。

```
// Declare and initialize host variables
int empnum=0;
String empname=null;
float salary=0.0f;

// Declare an iterator instance
EmpIter empIter;

#sql empIter = { SELECT ename, empno, sal FROM emp };
```

```
while (true) {
    #sql { FETCH :empsIter INTO :empnum, :empname, :salary };
    if (empsIter.endFetch()) break; // This test must be AFTER fetch,
                                   // but before results are processed.
    System.out.println("Name is " + empname);
    System.out.println("Employee number is " + empnum);
    System.out.println("Salary is " + salary);
}

// Close the iterator
empsIter.close();
...
```

empname、empnum および salary 変数は、Java ホスト変数です。これらの変数の型は、イテレータ列の型と対応させる必要があります。

位置指定イテレータでは、next() メソッドを使用しないでください。このメソッドは、FETCH 操作が次の行に進むときに暗黙的にコールされます。

注意： FETCH INTO 文のホスト変数は、条件文の 1 つの分岐で代入されるため、必ず初期化する必要があります。初期化しないと、ホスト変数の代入が行われないことを通知するコンパイル・エラーが表示されます。フェッチ対象の行がある場合にのみ、FETCH で変数が代入されます。

next() メソッドによる位置指定イテレータ・ナビゲーション

前項で説明した位置指定 FETCH 句では、ホスト変数があればそれを移入する前に、暗黙的な next() コールによって移動が実行されます。この方法のかわりに、Oracle SQLJ では、JDBC 結果セットや SQLJ 名前指定イテレータと同じ移動ロジックを使用するために、特別な FETCH 構文を明示的な next() コールと併用する方法をサポートしています。この特別な FETCH 構文を使用すると、セマンティクスが変化して、INTO リストが移入される前に暗黙的な next() コールがなくなります。

詳細は、7-42 ページの「[FETCH CURRENT 構文 : JDBC 結果セットから SQLJ イテレータへ](#)」を参照してください。

ホスト変数としてのイテレータおよび結果セットの使用

SQLJ では、ホスト変数としての SQLJ イテレータおよび JDBC 結果セットの使用がサポートされています。

注意：

- また、SQLJ では、ストアド・ファンクションの戻り値としてイテレータおよび結果セットを使用できます。詳細は、3-57 ページの「[ストアド・ファンクションの戻り値としてのイテレータおよび結果セットの使用](#)」を参照してください。
 - Oracle JDBC ドライバでは、現在、入力ホスト変数としての結果セットの使用がサポートされていません。OraclePreparedStatement クラスには `setCursor()` メソッドがありますが、これをコールすると、実行時に例外が発生します。
-
-

次に示す例のように、イテレータと結果セットの使用方法は基本的には同じです。ただし、宣言やデータを取り出すアクセサ・メソッドが異なります。

ここでの例は、次の部門表および従業員表を想定します。

```
CREATE TABLE DEPT (  
    DEPTNO NUMBER(2),  
    DNAME VARCHAR2(14) );
```

```
CREATE TABLE EMP (  
    EMPNO NUMBER(4),  
    ENAME VARCHAR2(10),  
    SAL NUMBER(7,2),  
    DEPTNO NUMBER(2) );
```

例：OUT ホスト変数としての結果セットの使用 この例では、JDBC 結果セットを出力ホスト変数として使用しています。

```
...  
ResultSet rs;  
...  
#sql { BEGIN  
    OPEN :OUT rs FOR SELECT ename, empno FROM emp;  
    END };
```

```
while (rs.next())
{
    String empname = rs.getString(1);
    int empnum = rs.getInt(2);
}
rs.close();
...
```

この例では、結果セット `rs` を PL/SQL ブロックでオープンして `SELECT` 文からのデータを受け取り、EMP 表の `ENAME` 列と `EMPNO` 列のデータを選択し、結果セットをループしてデータをローカル変数に取り出します。

例 : OUT ホスト変数としてのイテレータの使用 この例では、名前指定イテレータを出力ホスト変数として使用しています。

次のように宣言します。

```
#sql public <static> iterator EmpIter (String ename, int empno);
```

`public` 修飾子は必須です。クラスレベルまたはネストされたクラスレベルでの宣言では `static` も使用することをお勧めします。

実行可能コードを示します。

```
...
EmpIter iter;
...
#sql { BEGIN
        OPEN :OUT iter FOR SELECT ename, empno FROM emp;
    END };

while (iter.next())
{
    String empname = iter.ename();
    int empnum = iter.empno();

    ...process/output empname and empnum...
}
iter.close();
...
```

この例では、イテレータ `iter` を PL/SQL ブロックでオープンして `SELECT` 文からのデータを受け取り、EMP 表の `ENAME` 列と `EMPNO` 列のデータを選択し、イテレータをループさせてデータをローカル変数に取り出します。

例 : SELECT INTO の OUT ホスト変数としてのイテレータの使用 この例では、名前指定イテレータを出力ホスト変数として使用し、SELECT INTO 文を介してデータを取得しています。INTO リスト内のホスト変数のデフォルトは、OUT です。SELECT INTO 文と構文の詳細は、3-30 ページの「[単一行の問合せ結果 : SELECT INTO 文](#)」を参照してください。

次のように宣言します。

```
#sql public <static> iterator ENameIter (String ename);
```

public 修飾子は必須です。クラスレベルまたはネストされたクラスレベルでの宣言では static も使用することをお勧めします。

実行可能コードを示します。

```
...
ENameIter enamesIter;
String deptname;
...

#sql { SELECT dname, cursor
      (SELECT ename FROM emp WHERE deptno = dept.deptno)
      INTO :deptname, :enamesIter FROM dept WHERE deptno = 20 };

System.out.println(deptname);
while (enamesIter.next())
{
    System.out.println(enamesIter.ename());
}
enamesIter.close();
...
```

この例では、ネストされた SELECT 文を使用して次の処理を行います。

- DEPT 表から従業員番号 20 の名前を選択し、出力ホスト変数 deptname に取り込みます。
- EMP 表を問い合わせ、部門番号が 20 の従業員をすべて選択した後で、カーソルに格納された結果を、出力ホスト変数 enamesIter（名前指定イテレータ）に取り込みます。
- 部門名を出力します。
- 従業員名を出力する名前指定イテレータを、ループさせます。こうして、この部門の全従業員の名前が出力されます。

大抵の場合、外部の SELECT 内にある 1 行を取り出すときは SELECT INTO を使用した方が、ネストされたイテレータよりも便利です。このネストされたイテレータについては、3-52 ページの「[例 : 位置指定イテレータ内の名前指定イテレータ列](#)」などで後述します。また、ネストされたイテレータを使用した場合は、データを処理して外部の SELECT 中にある行数を求める必要があります。これに対し、SELECT INTO を使用すると、1 行のみで済みます。

イテレータ列としてのイテレータおよび結果セットの使用

Oracle SQLJ では、イテレータ宣言で、ResultSet 型の列、または現在のスコープで宣言されたその他のイテレータ型の列を指定することが可能です。つまり、Oracle SQLJ では、イテレータ内で他のイテレータや結果セットを使用できます。これらの列型は、カーソルで列を取得するのに使用されます。この機能は、ネストした表の情報を戻り値とするネストされた SELECT 文に便利です。

次に、機能的には同じ例をいくつか示します。これらの例では、ネストされた結果セットまたはイテレータ（他のイテレータ内の列にある結果セットまたはイテレータ）を使用して、DEPT 表から各部門に所属する従業員をすべて出力します。最初の例では名前指定イテレータ内の結果セットを、2 番目の例では名前指定イテレータ内の名前指定イテレータを、3 番目の例では位置指定イテレータ内の名前指定イテレータを使用します。

手順を次に示します。

1. DEPT 表から、各 DNAME（従業員名）を選択します。
2. ネストされた SELECT を実行し、各部門の EMP 表から取得した全従業員数をカーソルに取り込みます。
3. 取得した部門名と部門別従業員数を、名前列とイテレータ列を持つ外部イテレータ（iter）に取り込みます。特定部門の従業員情報を持つカーソルは、外部イテレータの当該部門の行に対応するイテレータ列に移動します。
4. ネステッド・ループを巡回することによって、部門別に部門名を出力し、その後、内部イテレータ内で部門別の全従業員の名前を出力します。

例：名前指定イテレータ内の結果セット列 この例では、名前指定イテレータで ResultSet 型の列を使用します。

次のように宣言します。

```
#sql iterator DeptIter (String dname, ResultSet emps);
```

実行可能コードを示します。

```
...
DeptIter iter;
...
#sql iter = { SELECT dname, cursor
              (SELECT ename FROM emp WHERE deptno = dept.deptno)
              AS emps FROM dept };
while (iter.next())
{
    System.out.println(iter.dname());
    ResultSet enamesRs = iter.emps();
    while (enamesRs.next())
    {
```

```

        String empname = enamesRs.getString(1);
        System.out.println(empname);
    }
    enamesRs.close();
}
iter.close();
...

```

例: 名前指定イテレータ内の名前指定イテレータ列 この例では、前に定義された名前指定イテレータ（ネストされたイテレータ）と同じ型の列を持つ、名前指定イテレータを使用します。

次のように宣言します。

```

#sql iterator ENameIter (String ename);
#sql iterator DeptIter (String dname, ENameIter emps);

```

実行可能コードを示します。

```

...
DeptIter iter;
...
#sql iter = { SELECT dname, cursor
              (SELECT ename FROM emp WHERE deptno = dept.deptno)
              AS emps FROM dept };

while (iter.next())
{
    System.out.println(iter.dname());
    ENameIter enamesIter = iter.emps();
    while (enamesIter.next())
    {
        System.out.println(enamesIter.ename());
    }
    enamesIter.close();
}
iter.close();
...

```

例 : 位置指定イテレータ内の名前指定イテレータ列 この例では、前に定義された名前指定イテレータ（ネストされたイテレータ）と同じ型の列を持つ、位置指定イテレータを使用します。ここでは、位置指定イテレータの `FETCH INTO` 構文を使用します。この例は、機能的には前述の 2 つの例と同じです。

外部イテレータは位置指定イテレータです。このため、外部指定イテレータが名前指定イテレータである前の例とは異なり、列名を一致される場合に別名は必要ありません。

次のように宣言します。

```
#sql iterator ENameIter (String ename);
#sql iterator DeptIter (String, ENameIter);
```

実行可能コードを示します。

```
...
DeptIter iter;
...
#sql iter = { SELECT dname, cursor
              (SELECT ename FROM emp WHERE deptno = dept.deptno)
              FROM dept };
while (true)
{
    String dname = null;
    ENameIter enamesIter = null;
    #sql { FETCH :iter INTO :dname, :enamesIter };
    if (iter.endFetch()) break;
    System.out.println(dname);
    while (enamesIter.next())
    {
        System.out.println(enamesIter.ename());
    }
    enamesIter.close();
}
iter.close();
...
```

代入文 (SET)

SQLJ では、SQL 操作内の Java ホスト式に値を代入できます。これは代入文と呼ばれ、次の構文で記述されます。

```
#sql { SET :host_exp = expression };
```

host_exp は、変数や配列索引などのターゲット・ホスト式です。*expression* には、番号、ホスト式、算術式、ファンクション・コールまたはターゲット・ホスト式に有効な結果を生成するその他の構文を使用できます。

代入文のターゲット・ホスト式のデフォルトは OUT ですが、これを明示的に記述することも可能です。

```
#sql { SET :OUT host_exp = expression };
```

代入文で IN または INOUT トークンを使用すると、変換時にエラーが発生します。

前述の 2 つの文は、機能的には次の PL/SQL コードと同じです。

```
#sql { BEGIN :OUT host_exp := expression; END };
```

次に、代入文の単純な例を示します。

```
#sql { SET :x = foo1() + foo2() };
```

この文では、*foo1()* と *foo2()* の戻り値の合計を *x* に代入し、*x* の型が、ファンクションの出力合計の型と互換性がある場合を想定しています。

次の例についても考えてみます。

```
int i2;
java.sql.Date dat;
...
#sql { SET :i2 = TO_NUMBER(substr('750 etc.', 1, 3)) +
        TO_NUMBER(substr('250 etc.', 1, 3)) };
...
#sql { SET :dat = sysdate };
...
```

最初の文では、値 1000 (750 + 250) を *i2* に代入します。*substr()* コールでは、文字列の最初の 3 文字 (750 と 250) を使用します。*TO_NUMBER()* コールでは、文字列が数値 750 と 250 に変換されます。

2 番目の文では、データベース・システムの日付が読み取られ、それが *dat* に代入されます。

データベースに格納されているファンクションからの戻り値に対して操作を実行する場合、代入文を使用すると便利です。単にファンクションの結果を変数に代入する場合、代入文は必要ありません。この場合は、3-54 ページの「[ストアド・プロシージャおよびストアド・](#)

[ファンクションのコール](#)」で説明するように、通常ファンクション・コールの構文で処理できます。Java ファンクションの出力を操作する場合も、代入文は必要ありません。この場合は、通常の Java 文で処理できます。このため、前述の例の `foo1()` および `foo2()` は、Java ファンクションではなく、データベース内のストアド・ファンクションであることがわかります。

ストアド・プロシージャおよびストアド・ファンクションのコール

SQLJ には、データベース内のストアド・プロシージャやストアド・ファンクションのコールに便利な構文があります。ストアド・プロシージャやストアド・ファンクションは、Java、PL/SQL またはデータベースでサポートされるその他の言語のいずれでも記述できます。

ストアド・ファンクションの場合、戻り値を受け取るために、SQLJ 実行文内に結果式が必要です。また、必要に応じて入力パラメータ、出力パラメータおよび入出力パラメータも使用できます。

ストアド・プロシージャの場合、戻り値はありませんが、入力パラメータ、出力パラメータおよび入出力パラメータを使用できます。ストアド・プロシージャでは、任意の出力または入出力パラメータを介して、出力が戻されます。

注意： ここで説明するプロシージャ・コールやファンクション・コールの構文を使用するかわりに、JPublisher で PL/SQL ストアド・プロシージャおよびストアド・ファンクション用の Java ラッパーを作成し、他の Java メソッドの場合と同様に、Java ラッパーをコールすることも可能です。JPublisher の詳細は、6-26 ページの「[JPublisher とカスタム Java クラスの作成](#)」で説明します。詳細は、『Oracle9i JPublisher ユーザーズ・ガイド』を参照してください。

ストアド・プロシージャのコール

ストアド・プロシージャには戻り値はありませんが、入力パラメータ、出力パラメータおよび入出力パラメータとしてリストを使用できます。ストアド・プロシージャのコールでは、次に示すように、CALL トークンを使用します。「CALL」のすぐ後に空白を 1 つ挿入した後に、プロシージャ名を記述します。プロシージャ名と区別するため、CALL トークンの後には空白が必要です。プロシージャ・コールの外側にカッコを付けることはできません。これは、3-55 ページの「[ストアド・ファンクションのコール](#)」で説明するファンクション・コールの構文とは異なります。

```
#sql { CALL PROC(<PARAM_LIST>) };
```

PROC はストアド・プロシージャ名で、入力パラメータ、出力パラメータおよび入出力パラメータを使用できます。*PROC* には、スキーマ名またはパッケージ名を含めることも可能で、たとえば、*SCOTT.MYPROC()* となります。

次の *PL/SQL* ストアド・プロシージャを定義するとします。

```
CREATE OR REPLACE PROCEDURE MAX_DEADLINE (deadline OUT DATE) IS
BEGIN
    SELECT MAX(start_date + duration) INTO deadline FROM projects;
END;
```

この例では、*PROJECTS* 表を読み込み、*START_DATE* 列と *DURATION* 列を検索し、各行の *start_date + duration* を計算した後で、*start_date + duration* の合計の最大値が *DEADLINE* に選択されます。これは *DATE* 型の出力パラメータです。

SQLJ では、この *MAX_DEADLINE* プロシージャを次のようにコールできます。

```
java.sql.Date maxDeadline;
...
#sql { CALL MAX_DEADLINE(:out maxDeadline) };
```

どのパラメータに対しても、ホスト式のトークン *IN* (オプション / デフォルト)、*OUT* および *INOUT* を使用して、ストアド・プロシージャの入力先、出力先および入出力先のパラメータと対応させてください。また、パラメータ・リストで使用するホスト変数の型も、ストアド・プロシージャのパラメータ型に対応させる必要があります。

注意： アプリケーションに *Oracle7* との互換性を持たせる場合、プロシージャがパラメータを使用しないときは、パラメータ・リストに空のカッコを含めないでください。次にその例を示します。

```
#sql { CALL MAX_DEADLINE };
```

次のようには定義できません。

```
#sql { CALL MAX_DEADLINE() };
```

ストアド・ファンクションのコール

ストアド・ファンクションには戻り値があります。また、入力パラメータ、出力パラメータおよび入出力パラメータのリストを使用できます。ストアド・ファンクションのコールでは、次に示すように、*VALUES* トークンを使用します。この構文では、「*VALUES*」の後にファンクション・コールを記述します。標準の *SQLJ* では、ファンクション・コールをカッコで囲むことが必要です。*Oracle SQLJ* の場合、カッコの使用は任意です。外部のカッコを使用した場合、*VALUES* トークンと左カッコの間に空白があってもかまいません。(Oracle *SQL* でサポートされている *INSERT INTO table VALUES* 構文でも *VALUES* トークンが使用されますが、使用目的が異なります。)

```
#sql result = { VALUES(FUNC(<PARAM_LIST>)) };
```

この構文で *result* は結果式で、ファンクション戻り値を使用します。*FUNC* はストアド・ファンクション名で、必要に応じて入力パラメータ、出力パラメータおよび入出力パラメータの並びを取ることが可能です。*FUNC* には、スキーマ名またはパッケージ名も含めることも可能で、たとえば、`SCOTT.MYFUNC()` となります。

次に、3-54 ページの「ストアド・プロシージャのコール」の例を再び使用します。ここでは、ストアド・プロシージャのかわりにストアド・ファンクションを定義します。

```
CREATE OR REPLACE FUNCTION GET_MAX_DEADLINE RETURN DATE IS
    deadline DATE;
BEGIN
    SELECT MAX(start_date + duration) INTO deadline FROM projects;
    RETURN deadline;
END;
```

SQLJ では、この `GET_MAX_DEADLINE` ファンクションを次のようにコールできます。

```
java.sql.Date maxDeadline;
...
#sql maxDeadline = { VALUES (GET_MAX_DEADLINE) };
```

結果式の型は、ファンクションの戻り値の型と同じであることが必要です。

Oracle SQLJ では、外部のカッコを省略した、次のような構文を使用することも可能です。

```
#sql maxDeadline = { VALUES GET_MAX_DEADLINE };
```

ストアド・ファンクションのコールでは、ストアド・プロシージャの場合と同様に、ホスト式のトークン `IN` (オプション/デフォルト)、`OUT` および `INOUT` を使用して、ストアド・ファンクションの入力先、出力先および入出力先のパラメータと対応させる必要があります。また、パラメータ・リストで使用するホスト変数の型を、ストアド・プロシージャのパラメータ型に対応させる必要もあります。

注意： ストアド・ファンクションを Oracle 以外の環境にも移植可能にするには、コールで入力パラメータのみを使用し、出力または入出力パラメータは使用しないでください。

アプリケーションに Oracle7 との互換性を持たせる場合、ファンクションがパラメータを使用しないときは、パラメータ・リストに空のカッコを含めないでください。次にその例を示します。

```
#sql maxDeadline = { VALUES (GET_MAX_DEADLINE) };
```

次のようには定義できません。

```
#sql maxDeadline = { VALUES (GET_MAX_DEADLINE()) };
```

ストアド・ファンクションの戻り値としてのイテレータおよび結果セットの使用

SQLJ では、ファンクションの戻り値が REF CURSOR 型の場合は、ストアド・ファンクションの戻り値をイテレータまたは結果セット変数に代入できます。

次の例では、イテレータを使用してストアド・ファンクションの戻り値を取得します。結果セットを使用する場合も同じです。

例: ストアド・ファンクションの戻り値としてのイテレータ この例では、ストアド・ファンクションの戻り型として、イテレータを使用します。この処理では REF CURSOR 型を使用します。REF CURSOR 型の詳細は、5-39 ページの「[Oracle の REF CURSOR 型のサポート](#)」を参照してください。

ファンクションが次のように定義されているとします。

```
CREATE OR REPLACE PACKAGE sqlj_refcursor AS
    TYPE EMP_CURTYPE IS REF CURSOR;
    FUNCTION job_listing (j varchar2) RETURN EMP_CURTYPE;
END sqlj_refcursor;

CREATE OR REPLACE PACKAGE BODY sqlj_refcursor AS
    FUNCTION job_listing (j varchar) RETURN EMP_CURTYPE IS
    DECLARE
        rc EMP_CURTYPE;
    BEGIN
        OPEN rc FOR SELECT ename, empno FROM emp WHERE job = j;
        RETURN rc;
    END;
END sqlj_refcursor;
```

このファンクションは、次のように使用します。

次のように宣言します。

```
#sql public <static> iterator EmpIter (String ename, int empno);
```

public 修飾子は必須です。クラスレベルまたはネストされたクラスレベルでの宣言では static も使用することをお勧めします。

実行可能コードを示します。

```
EmpIter iter;
...
#sql iter = { VALUES(sqlj_refcursor.job_listing('SALES')) };

while (iter.next())
{
    String empname = iter.ename();
    int empnum = iter.empno();

    ... process empname and empnum ...
}
iter.close();
...
```

この例では、`job_listing()` ファンクションのコールにより、ジョブ・タイトルが「SALES」であるすべての従業員の名前と従業員番号を含んだイテレータが戻り値になります。その後、イテレータからこのデータが取り出されます。

プログラミング上の主な考慮事項

この章では、SQLJ アプリケーションの開発および実行前に検討する必要がある重要な問題について説明します。また、要約とサンプル・アプリケーションも示します。次の項目について説明します。

- [JDBC ドライバの選択](#)
- [接続の際の考慮事項](#)
- [NULL の処理](#)
- [例外処理の基本](#)
- [基本的なトランザクション制御](#)
- [要約 : 簡単な SQLJ コード](#)
- [Oracle 固有コード生成（プロファイルなし）](#)
- [ネーミングの要件および制限事項](#)
- [中間層の SQLJ に関する考慮事項](#)

JDBC ドライバの選択

JDBC ドライバの選択にあたっては、変換時とランタイムにそれぞれ別のドライバを使用するかどうかを検討する必要があります。変換用とランタイムの各ドライバ・クラスを選択または登録し、そのドライバを接続 URL で指定してください。

注意： Oracle 固有コード生成を使用する場合、または ISO コード生成と Oracle カスタマイザを使用する場合は、Oracle JDBC ドライバが必要です。Oracle 固有の機能を実際には使用しない場合にも必要です。

Oracle JDBC ドライバの概要

Oracle JDBC ドライバを次に示します。

- **OCI ドライバ:** クライアント側で使用するドライバで、Oracle クライアントのインストールが必要です。
- **Thin ドライバ:** 100% Java で記述されたドライバで、クライアント側で特にアプレットから使用します (Oracle のインストールは不要です)。
- **サーバー側 Thin ドライバ:** クライアント側 Thin ドライバと同様の機能を備えていますが、Oracle サーバー内で実行するコード用のドライバでありリモート・サーバーにアクセスするためのものです。
- **サーバー側内部ドライバ:** ターゲット・サーバー内 (つまり、アクセスする Oracle サーバー内) で実行するコード用のドライバ。

Oracle では、JDK 1.1、JDK 1.2 (以上) および JDK 1.4 と互換性があるクライアント側のドライバが用意されています。Oracle9i データベースは、JDK 1.2 以上と互換性があります。(Oracle9i データベース リリース 2 (9.2) には、JDK 1.3 J2SE Java 環境が含まれています。)

ここからは、各ドライバの概要を簡単に説明します。これらのドライバの詳細とドライバの選択方法は、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。

変換時とランタイムとで、別々のドライバを選ぶ場合もあります。具体的には、変換時のセマンティクス・チェックには Oracle JDBC OCI ドライバを使用し、ランタイムには Oracle JDBC Thin ドライバを使用します。

JDBC の中核となる機能

これまでに紹介したドライバは、いずれも同様の機能性を備えています。各ドライバでサポートされている機能セット、構文、プログラミング・インタフェースおよび Oracle 拡張型は、すべて同じです。Oracle JDBC ドライバはすべて、`oracle.jdbc.OracleDriver` クラスでサポートされています。

JDBC OCI ドライバ

Oracle JDBC OCI ドライバでは、Oracle Call Interface (OCI) を直接 Java からコールしてデータベースにアクセスできるため、Oracle 7、Oracle 8、Oracle 8i および Oracle 9i の各バージョンとの高度な互換性が確保されています。これらのドライバでは、あらゆるインストール済み Oracle 9i Net アダプタ (IPC、Named Pipes、TCP/IP および IPX/SPX など) がサポートされています。

この OCI ドライバはシステム固有のメソッドを使用して C エントリ・ポイントをコールするため、Oracle プラットフォームに依存します。したがって、Oracle 9i Net をはじめとする Oracle クライアントのインストールが必要になります。つまり、この OCI ドライバはアプレットには適していません。

OCI ドライバ用の接続文字列は次の形式になります (*tns* は、オプションで TNS の別名または TNS 完全指定です)。

```
jdbc:oracle:oci:@<tns>
```

(下位互換性のために、oci のかわりに oci8 も許可されています。Oracle JDBC リリース 7.3.4 では oci7 も使用できます。)

JDBC Thin ドライバ

Oracle JDBC Thin ドライバは、プラットフォームに依存しない 100% Pure Java の実装であり、Java ソケットを使用して Oracle または Oracle 以外のクライアントから直接 Oracle サーバーに接続できます。アプレットの実行と同時にこのドライバをブラウザにダウンロードすることも可能です。

この Thin ドライバでは、TCP/IP プロトコルしかサポートされていないため、データベース・サーバーの TCP/IP ソケットを受け付ける TNS リスナーが必要です。この Thin ドライバとアプレットとを併用する場合、Java ソケットをサポートしているクライアント・ブラウザを使用してください。

Thin ドライバ用の接続文字列は、通常次の形式です (さらに長い形式もあります)。

```
jdbc:oracle:thin:@host:port:sid
```

JDBC サーバー側 Thin ドライバ

Oracle JDBC サーバー側の Thin ドライバは、クライアント側の Thin ドライバと同様の機能性を備えています。さらに Oracle 9i 内で動作し、リモート・サーバーにアクセスできます。このドライバは、ある Oracle サーバー内部 (Java ストアド・プロシージャなど) から別の Oracle サーバーへのアクセスに利用できます。

サーバー側 Thin ドライバ用の接続文字列は、クライアント側 Thin ドライバ用と同じです。

注意： サーバー側 Thin ドライバの使用時に、元のデータベースをそのままにしておくためには、ユーザー・アカウントに `SocketPermission` を割り当てる必要があります。詳細は、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。また、`SocketPermission` および他のアクセス権の概要は、『Oracle9i Java Developer's Guide』を参照してください。

JDBC サーバー側内部ドライバ

Oracle JDBC サーバー側内部ドライバは、SQL 操作を行うターゲットの Oracle9i インスタンスで実行される、あらゆる Java コードをサポートしています。サーバー側内部ドライバを使用すると、Oracle JVM から直接 SQL エンジンに通信できるようになります。このドライバは、Oracle9i のストアド・プロシージャ、ストアド・ファンクションまたはトリガーとして SQLJ コードを実行するためのデフォルトの JDBC ドライバです。

サーバー側内部ドライバ用の接続文字列は、次の形式です。

```
jdbc:oracle:kprb:
```

SQLJ コードでデフォルトの接続コンテキストを使用する場合、SQLJ では、Oracle JVM で実行するコード用に自動的にこのドライバが使用されます。

変換用ドライバの選択

ドライバ・マネージャ・クラスを選択し、変換用のドライバを指定するには、コマンドラインまたはプロパティ・ファイルで SQLJ オプションを設定します。

OracleDriver (デフォルト) 以外のドライバ・マネージャ・クラスを選択するには、SQLJ の `-driver` オプションを使用します。

SQLJ の `-url` オプションで接続 URL を指定するときは、選択した特定の JDBC ドライバ (Oracle 用の Thin や OCI など) も一緒に指定します。

これらのオプションの詳細は、8-32 ページの「[接続オプション](#)」を参照してください。

通常は、ソース・コードでランタイム接続用に指定したドライバを使用します。

注意： 前述の `-driver` オプションは、特定のドライバの選択には使用できません。このオプションを使用すると、ドライバ・マネージャにあるドライバ・クラスを登録します。そのドライバ・クラスは、複数のドライバ・プロトコル (すべての Oracle JDBC プロトコルで使用される OracleDriver など) に使用できます。

ランタイムに使用するドライバの選択および登録

実行時にデータベースに接続するには、接続インスタンス (`sqlj.runtime.ref.DefaultContext` クラスまたはそれ以外の宣言済みの接続コンテキスト・クラスのインスタンス) に対して指定した URL を認識するドライバを 1 つ以上登録する必要があります。

SQLJ ではこの登録が自動的に行えます。ただし、そのためには Oracle JDBC ドライバを使用し、`Oracle.connect()` メソッド (後述の 4-6 ページの「[DefaultContext を使用した単一接続または複数接続](#)」を参照) でデフォルトの接続を作成してください。この `Oracle.connect()` メソッドによって `oracle.jdbc.OracleDriver` クラスが自動的に登録されます。

Oracle JDBC ドライバを使用し、`Oracle.connect()` を使用しない場合は、次のように `OracleDriver` クラスを手動で登録する必要があります。

```
DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
```

Oracle JDBC ドライバを使用しない場合は、次のように適切なドライバ・クラスを登録する必要があります。

```
DriverManager.registerDriver(new mydriver.jdbc.driver.MyDriver());
```

どの場合でも、接続 URL、ユーザー名およびパスワードの設定が必要です。詳細は、4-6 ページの「[DefaultContext を使用した単一接続または複数接続](#)」を参照してください。その項で、`Oracle.connect()` メソッドの詳細も説明しています。

注意： JDBC 接続の確立時に JDBC ドライバ・マネージャを使用する別の方法として、データ・ソースも使用できます。データ・ソースを `with` 句に指定できます。詳細は、3-6 ページの「[WITH 句宣言](#)」を参照してください。データ・ソースの概要は、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。

接続の際の考慮事項

SQLJ アプリケーションで使用するデータベース接続の選択にあたっては、次の点を考慮してください。

- 必要なデータベース接続が単一であるか複数であるか。
- 複数の接続（また、必要に応じて複数のスキーマ）を使用する場合、各接続で同じ名前の SQL エンティティ、つまり、表の名前、列の名前とデータ型、ストアド・プロシージャの名前とシグネチャなどは、各接続のたびに同じ名前のものを使用するか。
- 変換時とランタイムにそれぞれ別の接続を使用するか、同じ接続を使用するか。

データベース接続用の接続コンテキストのインスタンス（DefaultContext または宣言した接続コンテキスト・クラスのインスタンス）は、SQLJ 実行文で指定します。接続コンテキスト仕様を指定せずに、デフォルトの接続（デフォルトとしてあらかじめ設定した DefaultContext のインスタンス）を使用することも可能です。

注意： 操作の際に数種類の SQL エンティティを使用する場合は、通常、新たに接続コンテキスト・クラスを宣言し、使用します。詳細は、7-2 ページの「[接続コンテキスト](#)」を参照してください。

DefaultContext を使用した単一接続または複数接続

ここでは、DefaultContext クラスの接続インスタンスのみを使用する場合について説明します。

単一接続や、名前とデータ型が同じ SQL エンティティを使用する複数接続では、代表的な方法です。

単一接続

単一接続の場合は、通常、DefaultContext クラスのインスタンスを 1 つ使用します。DefaultContext オブジェクトの作成時に、データベース URL、ユーザー名およびパスワードを指定します。

この作業は、oracle.sqlj.runtime.Oracle クラスの connect () メソッドで行えます。このメソッドをコールすると、デフォルトの接続コンテキスト・インスタンスが自動的に初期化されます。

このメソッドには複数のシグネチャがあり、たとえば、ユーザー名、パスワードおよび URL を直接指定できるものや、プロパティ・ファイルに指定するものがあります。次に、プロパティ・ファイル connect.properties を使用した例を示します。

```
Oracle.connect(MyClass.class, "connect.properties");
```

MyClass はクラス名です。connect.properties の例は \$Oracle_Home/sqlj/demo および 2-13 ページの「[ランタイム接続の設定](#)」にあります。

注意： `connect.properties` ファイルは、指定したクラスに基づいて検索されます。たとえば `MyClass` が `my-package` にある場合、`connect.properties` は `MyClass.class` と同じパッケージの場所である `my-package` で検索されます。

`connect.properties` を使用する場合は、必要な編集とアプリケーションでのパッケージ化を行ってください。この例では、`oracle.sqlj.runtime.Oracle` クラスのインポートも必要です。

次のようにすると、ユーザー名、パスワードおよび URL を直接指定できます。

```
Oracle.connect("jdbc:oracle:thin:@localhost:1521:orcl", "scott", "tiger");
```

この例では、JDBC Thin ドライバを使用してユーザー `scott`（パスワード `tiger`）をマシン `localhost` 上のデータベースに、ポート `1521` 経由で接続します。`orcl` は、マシン上の接続先データベースの SID（Oracle セッション ID）です。

どちらの場合も、`DefaultContext` クラスの特別な静的インスタンスが生成され、デフォルトの接続としてインストールされます。`DefaultContext` のインスタンスを直接操作する必要はありません。

ここまでの手順を終えた後、アプリケーションの SQLJ 実行文に対していっさい接続を指定しないことも可能です。ただし、その場合は常にデフォルトの接続が使用されます。

Thin ドライバを使用する場合は、前の例で示したように、ホスト名、ポート番号および SID を URL に指定してください。また、データベースには指定したポートで実行するリスナーが必要です。OCI ドライバを使用する場合、SID を指定できますが、クライアントのデフォルト・アカウントを使用するときは SID の指定を省略できます。かわりに、名前と値の対を使用することも可能です（詳細は、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください）。

次に示した例のうち 1 番目は SID `orcl` を使用してデータベースに接続する例で、2 番目はクライアントのデフォルト・アカウントに接続する例です。

```
jdbc:oracle:oci:@orcl
jdbc:oracle:oci:@
```

注意：

- デフォルトの接続がすでに設定されていれば、`Oracle.connect()` を指定した場合でもデフォルトの接続が再設定されることはありません。この場合は、`NULL` が戻されます。（この機能により、クライアントとサーバーで同じコードを使用できます。）デフォルトの接続を指定変更するには、`DefaultContext` クラスの `static setDefaultContext()` メソッドを使用します。詳細は、次の項を参照してください。
- `Oracle.connect()` メソッドの自動コミット・フラグのデフォルト値は `false` ですが、このメソッドのシグネチャは明示的に設定できます。4-12 ページの「[Oracle クラスについて](#)」を参照してください。自動コミットの機能の概要は、4-24 ページの「[基本的なトランザクション制御](#)」を参照してください。（Oracle JDBC の自動コミット・フラグのデフォルト値は `true` です。）
- `Oracle.connect()` をコールする方法としては、`MyClass.class` のかわりに `getClass()` を指定する方法もあります。ただし、この指定は `getClass()` を `static` メソッドからコールしない場合にのみ有効です。SQLJ デモ・アプリケーションには、`getClass()` メソッドの例がいくつかあります。
- 次のように、`static DefaultContext` インスタンスに接続できます。これは、デフォルトの接続に対応しています。

```
DefaultContext.getDefaultContext();
```

複数接続

複数接続の場合は、`DefaultContext` クラスのインスタンスをさらに作成して使用します。前項の「単一接続」で作成したデフォルト接続もそのまま使用できます。

`DefaultContext` のインスタンス化するには、次の例のように `Oracle.getConnection()` メソッドを使用します。

最初に、これまでに作成したデフォルト接続を大半の文で使用し、残りの文では別の接続を使用するとします。`DefaultContext` のインスタンスを新たに 1 つ作成する必要があります。

```
DefaultContext ctx = Oracle.getConnection (
    "jdbc:oracle:thin:@localhost2:1521:orcl2", "bill", "lion");
```

（同一スキーマに対して複数の操作を行う場合は、`ctx` で `scott/tiger` スキーマを使用することも可能です。）

デフォルトの接続を使用する場合は、接続コンテキストを指定する必要はありません。

```
#sql { SQL operation };
```

これは実際には、次の形式を省略しています。

```
#sql [DefaultContext.getDefaultContext()] { SQL operation };
```

追加した接続を使用する場合は、接続として `ctx` を指定します。

```
#sql [ctx] { SQL operation };
```

次の例では、複数の接続を使用します。各接続は、`DefaultContext` の名前付きインスタンスです。これによって、たとえば接続を交互に切替えることが可能です。

次の文は、同一スキーマに対して複数の接続を確立します（複数の `Oracle` セッションまたはトランザクションを使用する場合など）。接続ごとに `DefaultContext` クラスをインスタンス化します。

```
DefaultContext ctx1 = Oracle.getConnection  
    ("jdbc:oracle:thin:@localhost1:1521:orcl1", "scott", "tiger");  
DefaultContext ctx2 = Oracle.getConnection  
    ("jdbc:oracle:thin:@localhost1:1521:orcl1", "scott", "tiger");
```

接続コンテキストのインスタンスが2つ作成されます。どちらも、`Oracle JDBC Thin` ドライバを介して、マシン `localhost1` 上の `SID orcl1` の `scott/tiger` に接続します。

次に、スキーマごとにそれぞれ別の接続を使用する場合を想定します。この場合も、接続ごとに `DefaultContext` クラスをインスタンス化します。

```
DefaultContext ctx1 = Oracle.getConnection  
    ("jdbc:oracle:thin:@localhost1:1521:orcl1", "scott", "tiger");  
DefaultContext ctx2 = Oracle.getConnection  
    ("jdbc:oracle:thin:@localhost2:1521:orcl2", "bill", "lion");
```

接続コンテキストのインスタンスが2つ作成されます。この2つのインスタンスは、同じ `Oracle JDBC Thin` ドライバを介して、それぞれ別のスキーマを使用します。`ctx1` オブジェクトは、マシン `localhost1` 上の `SID orcl1` にある `scott/tiger` に接続します。一方、`ctx2` オブジェクトは、マシン `localhost2` 上の `SID orcl2` にある `bill/lion` に接続します。

アプリケーションの `SQLJ` 実行文でこれらの接続を交互に切り替える方法には、次の2通りがあります。

- 接続の切替え頻度が高い場合に、アプリケーションの各文に対して、次のように接続を指定する方法。

```
#sql [ctx1] { SQL operation };  
...  
#sql [ctx2] { SQL operation };
```

注意： 接続コンテキストのインスタンスの名前は、必ず大カッコで囲みます。大カッコも構文の構成要素です。

または

- コード・フロー内の行で片方の接続を複数回連続して使用する場合、デフォルト接続をリセットするために、DefaultContext クラスの static setDefaultContext() メソッドを定期的に使用する方法。このメソッドによって、デフォルトの接続コンテキスト・インスタンスが初期化されます。この方法では、SQLJ 文で接続を指定する必要がありません。

```
DefaultContext.setDefaultContext(ctx1);
#sql { SQL operation };    // These three statements all use ctx1
#sql { SQL operation };
#sql { SQL operation };
...
DefaultContext.setDefaultContext(ctx2);
#sql { SQL operation };    // These three statements all use ctx2
#sql { SQL operation };
#sql { SQL operation };
```

注意： 前述の文では接続コンテキストを指定していないため、変換時にデフォルトの接続コンテキストかどうか文ごとにチェックされます。

接続の終了

接続が完了した後、接続コンテキスト・インスタンスを終了してください。try ブロックの finally 句（アプリケーションが例外時に終了した場合）で終了を指定することをお勧めします。

DefaultContext クラス、および宣言したその他のあらゆる接続コンテキスト・クラスには、close() メソッドが用意されています。この close() メソッドをコールすると、SQLJ 接続コンテキスト・インスタンスが終了し、デフォルトでは、基になる JDBC 接続インスタンスおよび物理的な接続も終了します。

また、oracle.sqlj.runtime.Oracle クラスに用意されている static close() メソッドを使用すると、デフォルトの接続のみが終了します。

次の例では、あらゆる接続コンテキスト・クラスのインスタンスを `ctx` として示した場合を想定しています。

```
...
finally
{
    ctx.close();
}
...
```

SQL 例外が発生して、`finally` 句が `try` ブロック内にはない場合は、次のようになります。

```
...
finally
{
    try { ctx.close(); } catch(SQLException ex) {...}
}
...
```

デフォルトの接続を終了する方法としては、`Oracle` クラスに用意されている `close()` メソッドを使用する方法もあります。

```
...
finally
{
    Oracle.close();
}
...
```

接続を終了する際は、必ず、保留になっている変更をコミットまたはロールバックしてください。接続終了時に暗黙的な `COMMIT` 操作が行われるかどうかは、`JDBC` 規格には明示されておらず、またベンダーによっても異なります。`Oracle` の場合、接続終了時に暗黙的な `COMMIT` が行われるようになっています。また、ガベージ・コレクションが実行中のために接続が未終了のままになったときには、暗黙的な `ROLLBACK` が行われるようになっていますが、こうしたメカニズムに頼ることはなるべく避けてください。

注意： 基になる接続（共有接続の場合）を終了せずに、接続コンテキスト・インスタンスを終了することも可能です。7-52 ページの「[共有接続の終了](#)」を参照してください。

宣言済みの接続コンテキスト・クラスを使用した複数接続

数種類の SQL エンティティを使用した接続の場合は、接続コンテキストの宣言によって、接続コンテキスト・クラスをさらに定義すると利便性が高まります。使用する SQL エンティティごとに個別の接続コンテキスト・クラスを作成すると、SQLJ ではコードのセマンティクス・チェックがより厳密に行われます。

これについては、さらに細かい説明があります。詳細は、7-2 ページの「[接続コンテキスト](#)」を参照してください。

Oracle クラスについて

Oracle SQLJ の `oracle.sqlj.runtime.Oracle` クラスでは、`DefaultContext` クラスのインスタンスを簡単に作成して使用できます。

`static connect()` メソッドは、デフォルトの接続コンテキスト・インスタンスを初期化します（`DefaultContext` オブジェクトのインスタンスを作成し、デフォルトの接続としてインストールします）。`connect()` から戻された `DefaultContext` のインスタンスの代入や使用は、任意に行ってください。デフォルトの接続がすでに確立されていると、`connect()` から `NULL` が戻されます。

`static getConnection()` メソッドでは、単に `DefaultContext` オブジェクトのインスタンスが生成され、そのインスタンスが戻されます。必要に応じて、戻されたインスタンスを使用できます。

`oracle.jdbc.OracleDriver` クラスを `CLASSPATH` に指定すると、どちらのメソッドを使用しても Oracle JDBC ドライバ・マネージャが自動的に登録されるようになります。

`static close()` メソッドは、デフォルトの接続を終了するメソッドです。

Oracle.connect() メソッドおよび Oracle.getConnection() メソッドのシグネチャ

各メソッドのシグネチャは、次のパラメータを入力としてとります。

- URL (String)、ユーザー名 (String)、パスワード (String)
- URL (String)、ユーザー名 (String)、パスワード (String)、自動コミット・フラグ (boolean)
- URL (String)、`java.util.Properties` オブジェクト（接続のプロパティが格納されているオブジェクト）
- URL (String)、`java.util.Properties` オブジェクト、自動コミット・フラグ (boolean)

- ユーザー名やパスワードなど、接続を詳細に指定する URL (String)

次に示す URL 文字列のフォーマットの例では、ユーザー名 (scott) およびパスワード (tiger) が指定されており、Oracle JDBC ドライバとしては Thin ドライバが使用されています。

```
"jdbc:oracle:thin:scott/tiger@localhost:1521:orcl"
```

- URL (String)、自動コミット・フラグ (boolean)
- クラスの `java.lang.Class` オブジェクト (そのクラスに基づいてプロパティ・ファイルがロードされる)、プロパティ・ファイル名 (String)
- `java.lang.Class` オブジェクト、プロパティ・ファイル名 (String)、自動コミット・フラグ (boolean)
- `java.lang.Class` オブジェクト、プロパティ・ファイル名 (String)、ユーザー名 (String)、パスワード (String)
- `java.lang.Class` オブジェクト、プロパティ・ファイル名 (String)、ユーザー名 (String)、パスワード (String)、自動コミット・フラグ (boolean)
- JDBC 接続オブジェクト (Connection)
- SQLJ 接続コンテキスト・オブジェクト

最後の 2 つのシグネチャは既存のデータベース接続を継承します。接続を継承すると、その接続に対する自動コミットの設定も継承されます。

自動コミット・フラグは、SQL 操作を自動的にコミットするかしないかを指定するフラグです。`Oracle.connect()` メソッドおよび `Oracle.getConnection()` メソッドの場合にのみ、デフォルトは `false` になっています。デフォルト値を使用すると、入力として自動コミットをとらないシグネチャを使用できます。ただし、コンストラクタを使用して、`DefaultContext` などの接続コンテキスト・クラスのインスタンスを作成する場合は、自動コミットの設定を指定する必要があります。`Oracle JDBC` の自動コミット・フラグは、デフォルトとして `true` を取ります。

自動コミット・フラグの詳細は、4-24 ページの「[基本的なトランザクション制御](#)」を参照してください。

`connect()` コールおよび `getConnection()` コールのいくつかの例は、4-6 ページの「[DefaultContext を使用した単一接続または複数接続](#)」の項を参照してください。

Oracle.close() メソッドのオプションのパラメータ

デフォルトの接続をクローズするのに `Oracle.close()` メソッドを使用すると、基になる物理的なデータベース接続をクローズするかしないかを指定できます。デフォルトでは、接続のクローズが指定されます。複数の接続オブジェクト間において物理的な接続を共有している場合には、この指定は重要です。そのオブジェクトが、SQLJ 接続コンテキスト・インスタンスまたは JDBC 接続インスタンスのどちらの場合にも、当てはまります。

基になる物理的な接続を、オープンした状態に保つには、次のようにします。

```
Oracle.close(ConnectionContext.KEEP_CONNECTION);
```

基になる物理的な接続をクローズするには、次のようにします（デフォルトの動作）。

```
Oracle.close(ConnectionContext.CLOSE_CONNECTION);
```

KEEP_CONNECTION および CLOSE_CONNECTION は、ConnectionContext インタフェースの static 定数です。

これらのパラメータの使用方法および共有接続の詳細は、7-52 ページの「[共有接続の終了](#)」を参照してください。

DefaultContext クラスについて

sqlj.runtime.ref.DefaultContext クラスは、接続コンテキスト・クラスの完全なデフォルト実装を提供します。接続コンテキストを宣言して作成したクラスと同じように、DefaultContext クラスも sqlj.runtime.ConnectionContext インタフェースを実装します。（このインタフェースの詳細は、7-8 ページの「[接続コンテキスト・クラスの実装と機能](#)」を参照してください。）

DefaultContext クラスのクラス定義は、次のように宣言したときに SQLJ トランスレータで生成される定義と同じです。

```
#sql public context DefaultContext;
```

DefaultContext メソッド

DefaultContext クラスには、重要なメソッドが 4 つ用意されています。

- `getConnection()`: 基になる JDBC 接続オブジェクトを取得します。これは、アプリケーションに JDBC コードを使用する場合に役立ちます（たとえば、動的 SQL 操作を使用する 1 つの方法です）。基になる JDBC 接続オブジェクトの `setAutoCommit()` メソッドを使用すると、接続の自動コミット・フラグを設定できます。
- `setDefaultContext()`: アプリケーションがデフォルトで使用する接続を設定する static メソッドです。入力として DefaultContext インスタンスを取ります。SQLJ 実行文で接続コンテキストのインスタンスを指定しないと、このメソッド（または `Oracle.connect()` メソッドで定義した）デフォルト接続が使用されます。
- `getDefaultContext()`: static メソッドの一種です。アプリケーションのデフォルト接続として現在定義されている DefaultContext のインスタンスを戻します（デフォルトの接続は、`setDefaultContext()` メソッドで定義します）。
- `close()`: 接続コンテキスト・クラスと同様、DefaultContext クラスには、接続コンテキスト・インスタンスを終了する `close()` メソッドが用意されています。

`getConnection()` メソッドおよび `close()` メソッドは、`sqlj.runtime.ConnectionContext` インタフェースで指定します。

注意： クライアント側では、`setDefaultContext()` をあらかじめコールしないと、`getDefaultContext()` からの戻り値が `NULL` になります。ただし、データ・ソース・オブジェクトが JNDI の `jdbc/defaultDataSource` 下にバインドされている場合は、クライアントでデフォルトの接続としてこのデータ・ソース・オブジェクトが使用されます。（データ・ソースと JNDI への Oracle SQLJ のサポートの詳細は、7-12 ページの「[標準データ・ソースのサポート](#)」を参照してください。）

サーバーでは、`getDefaultContext()` によってデフォルトの接続（サーバー自体への接続）が戻されます。

DefaultContext コンストラクタ

通常は、`Oracle.connect()` または `Oracle.getConnection()` メソッドを使用して `DefaultContext` をインスタンス化します。ただし、`DefaultContext` の場合、5 つのコンストラクタのうちのいずれかを使用すると、インスタンスを直接作成できます。これらのコンストラクタは、次の入力パラメータを使用します。

- URL (String)、ユーザー名 (String)、パスワード (String)、自動コミット (boolean)
- URL (String)、`java.util.Properties` オブジェクト、自動コミット (boolean)
- URL (ユーザー名やパスワードなど、接続の詳細を指定する String)、自動コミットの設定 (boolean)

次に示す URL 文字列のフォーマットの例では、ユーザー名 (scott) およびパスワード (tiger) が指定されており、Oracle JDBC ドライバとしては Thin ドライバが使用されています。

```
"jdbc:oracle:thin:scott/tiger@localhost:1521:orcl"
```

- JDBC 接続オブジェクト (Connection)
- SQLJ 接続コンテキスト・オブジェクト

最後の 2 つは、既存のデータベース接続を継承するためのパラメータです。接続を継承すると、その接続に対する自動コミットの設定も継承されます。

`DefaultContext` インスタンスを作成する例を、次に示します。

```
DefaultContext defctx = new DefaultContext  
    ("jdbc:oracle:thin:@localhost:1521:orcl", "scott", "tiger", false);
```

接続コンテキスト・コンストラクタに関する注意：

- 接続コンテキスト・クラスは、`Oracle.connect()` メソッドとは異なり、自動コミットの設定が必要です。
- 前述のコンストラクタのうち、最初の 3 つを使用するには、あらかじめ JDBC ドライバを登録する必要があります。ドライバを自動的に登録するには、Oracle JDBC ドライバを使用し、`Oracle.connect()` をコールします。手動で登録する場合には、4-5 ページの「ランタイムに使用するドライバの選択および登録」を参照してください。
- 通常宣言する接続コンテキスト・クラスのコンストラクタ・シグネチャは、`DefaultContext` クラスと同じです。ただし、接続コンテキスト・クラスをデータ・ソースに対応付けるように宣言する場合、別のコンストラクタが用意されています。（詳細は、7-12 ページの「標準データ・ソースのサポート」を参照してください。）
- JDBC 接続オブジェクトを引数とするコンストラクタを使用する場合は、接続コンテキスト・インスタンスを `NULL` JDBC で初期化しないでください。
- 自動コミットの設定では、SQL 操作が自動的にコミットされるかどうか定義されます。詳細は、4-24 ページの「基本的なトランザクション制御」を参照してください。

DefaultContext close() メソッドのオプションのパラメータ

(`DefaultContext` クラスをはじめとする、あらゆるクラスの) 接続コンテキスト・インスタンスをクローズするときは、基になる物理的な接続をクローズするかしないかを指定できます。デフォルトでは、接続のクローズが指定されます。複数の接続オブジェクト間において物理的な接続を共有している場合には、この指定は重要です。そのオブジェクトが、SQLJ 接続コンテキスト・インスタンスまたは JDBC 接続インスタンスのどちらの場合にも、当てはまります。`DefaultContext` のインスタンス `defctx` を想定した例を、いくつか次に示します。

基になる物理的な接続を、オープンした状態に保つには、次のようにします。

```
defctx.close(ConnectionContext.KEEP_CONNECTION);
```

基になる物理的な接続をクローズするには、次のようにします（デフォルトの動作）。

```
defctx.close(ConnectionContext.CLOSE_CONNECTION);
```

`KEEP_CONNECTION` および `CLOSE_CONNECTION` は、`ConnectionContext` インタフェースの `static` 定数です。

これらのパラメータの使用方法および共有接続の詳細は、7-52 ページの「共有接続の終了」を参照してください。

変換時の接続

変換時にオンライン・セマンティクス・チェックを使用する場合は、SQLJ 用のデータベース接続（基本スキーマと総称されます）を使用するように指定します。詳細は、7-2 ページの「[接続コンテキストの概要](#)」を参照してください。

変換時とランタイムで別々の接続を使用できます。実際に別々の接続を使用する必要がある場合が多く、またそうすることをお勧めします。現在開発を行っている環境と将来アプリケーションを実行する環境とが異なる場合は、別々の接続を使用する必要がある場合がありますが、変換時にランタイム接続を使用できる場合にも、オンライン・チェックの精度を高めるためにはリソースを絞り込んでアカウントを作成する方が望ましいといえます。実際にこの精度を高めるには、ランタイム接続時にアプリケーションで使用される SQL エンティティのサブセットを少量にする方法があります。アプリケーションで実際に使用する SQL エンティティのみに絞り込んだ基本スキーマを作成すると、オンライン・チェックの精度と信頼度が高まります。

変換時の接続を指定するには、コマンドラインまたはプロパティ・ファイルで SQLJ トランスレータの接続オプション（-url、-user および -password）を使用します。

これらのオプションの詳細は、8-32 ページの「[接続オプション](#)」を参照してください。

カスタマイズ時の接続

一般に、Oracle のカスタマイズではデータベース接続は不要ですが、Oracle SQLJ ではカスタマイズ接続がサポートされています。これは、2 つの環境で利用できます。

- 使用する Oracle カスタマイズの `optcols` オプションを使用可能にすると、接続が必須になります。このオプションを使用すると、イテレータ列型およびサイズは、パフォーマンスが最適化されるように定義されます。
- `SQLCheckerCustomizer` と呼ばれる、プロファイルに対してセマンティクス・チェックを実行する特別なカスタマイズを使用した場合、オンライン・チェッカ（デフォルトでは使用）を使用するためには接続が必要になります。

Oracle カスタマイズの `optcols` オプション（ISO 標準コード生成用）の詳細は、A-26 ページの「[Oracle カスタマイズの列定義オプション（optcols）](#)」を参照してください。（Oracle 固有コード生成の場合は、SQLJ トランスレータに同じ機能で `-optcols` オプションがあります。）

`SQLCheckerCustomizer` は、Oracle カスタマイズ・ハーネスの `verify` オプションを指定すると起動されるようになります。A-38 ページの「[プロファイルのセマンティクス・チェック用の SQLCheckerCustomizer](#)」を参照してください。

どのようなカスタマイズを使用する場合にも、カスタマイズ用の接続パラメータを指定します。この指定には、カスタマイズ・ハーネスの `user`、`password`、`url` および `driver` オプションを使用します。A-17 ページの「[カスタマイズ・ハーネスの接続用オプション](#)」を参照してください。

NULL の処理

Java の基本型 (int、double、float など) は、NULL 値をとりません。結果式およびホスト式の型を選択するときは、このことに注意してください。

NULL 処理用のラッパー・クラス

JDBC では NULL を 0 (つまり特定のデータ型の false) として取り出しますが、これとは対照的に、SQLJ では SQL の NULL を Java の NULL として取り出すことが一貫して実施されています。そのため、SQL の NULL が入力されたとしても、SQLJ では Java の基本型を出力変数としては使用しないでください。Java の基本型は、NULL 値をとらないためです。

このことは、結果式、出力ホスト式、入出力ホスト式およびイテレータ列の型について当てはまるので、注意が必要です。Java の基本型を受取り用にして SQL の NULL を取り出そうとすると、`sqlj.runtime.SQLNullException` が発生し、代入が行われなくなります。

Java の基本型に NULL 値が代入されるのを避けるには、基本型のかわりに次のラッパー・クラスを使用します。

- `java.lang.Boolean`
- `java.lang.Byte`
- `java.lang.Short`
- `java.lang.Integer`
- `java.lang.Long`
- `java.lang.Double`
- `java.lang.Float`

基本型の値に再変換する必要がある場合は、ラッパー・クラスの `xxxValue()` メソッドを使用してください。たとえば、`intValue()` を使用すると、int 値が `Integer` オブジェクトから戻され、`floatValue()` を使用すると、float 値が `Float` オブジェクトから戻されます。このメソッドは、次の例に示したように使用します。この例では、`intobj` を `Integer` オブジェクトとして想定しています。

```
int j = intobj.intValue();
```

注意：

- SQLException は、標準 java.sql.SQLException クラスのサブクラスです。4-23 ページの「[SQLException サブクラスの使用方](#)法」を参照してください。
 - Java のオブジェクトは NULL 値を格納できます。したがって、SQLJ では、他のホスト言語（C、C++、COBOL など）と異なり、インジケータ変数を必要としません。
-

NULL 処理の例

次に、java.lang ラッパ・クラスを使用して NULL データを処理する例を示します。

例：NULL 入力ホスト変数 次の例では、Float オブジェクトを使用して、NULL 値をデータベースに渡します。この場合は、Java の基本型 float を使用できません。

例：

```
int empno = 7499;
Float commission = null;

#sql { UPDATE emp SET comm = :commission WHERE empno = :empno };
```

例：NULL イテレータ行 次の例では、NULL データに対応できるように、イテレータに Double 列型を使用します。

EMP 表の従業員のうち、給与が \$50,000 以上の従業員の名前 (ENAME) と歩合 (COMM) をイテレータに取り出します。次に各行をテストして、COMM フィールドが実際に NULL になっているかどうかを調べます。NULL の場合は、NULL の処理が行われます。

次のような宣言があるとします。

```
#sql iterator EmployeeIter (String ename, Double comm);

例：

EmployeeIter ei;
#sql ei = { SELECT ename, comm FROM emp WHERE sal >= 50000 };

while (ei.next())
{
    if (ei.comm() == null)
        System.out.println(ei.ename() + " is not on commission.");
}
ei.close();
...
```

注意： WHERE 句の NULL 値との比較を実行するには、次の SQL 構文を使用します。

```
...WHERE :x IS NULL
```

例外処理の基本

ここでは、SQLJ アプリケーションにおける例外処理の基本として、エラー・チェック要件などについて説明します。

SQLJ および JDBC の例外処理要件

SQLJ 実行文は、`sqlj.runtime` からの JDBC コールになります。JDBC では、SQL の例外をキャッチまたは発生させる必要があります。したがって、SQLJ でも、SQLJ 実行文を含むブロックで、SQL の例外をキャッチまたは発生させる必要があります。適切な例外処理を組み込んでおかないと、ソース・コードのコンパイル時にエラーが発生することがあります。

SQL 例外処理には、`SQLException` クラスが必要です。これは標準 JDBC の `java.sql.*` パッケージにあります。

例：例外処理 SQLJ アプリケーションに必要な基本的な例外処理の例です。main メソッドでは `try/catch` ブロックを使用します。もう 1 つのメソッドは、main からコールされ、必要な場合にコール側の main に対して例外を発生させます。

```
/* Import SQLExceptions class. The SQLException comes from
   JDBC. Executable #sql clauses result in calls to JDBC, so methods
   containing executable #sql clauses must either catch or throw
   SQLException.
*/
import java.sql.* ;
import oracle.sqlj.runtime.Oracle;

// iterator for the select

#sql iterator MyIter (String ITEM_NAME);

public class TestInstallSQLJ
{
    //Main method
    public static void main (String args[])
    {
        try {
            /* if you're using a non-Oracle JDBC Driver, add a call here to
               DriverManager.registerDriver() to register your Driver
            */
        }
    }
}
```

```
// set the default connection to the URL, user, and password
// specified in your connect.properties file
Oracle.connect(TestInstallSQLJ.class, "connect.properties");

TestInstallSQLJ ti = new TestInstallSQLJ();
ti.runExample();
} catch (SQLException e) {
    System.err.println("Error running the example: " + e);
}

} //End of method main

//Method that runs the example
void runExample() throws SQLException
{
    //Issue SQL command to clear the SALES table
    #sql { DELETE FROM SALES };
    #sql { INSERT INTO SALES(ITEM_NAME) VALUES ('Hello, SQLJ!')};

    MyIter iter;
    #sql iter = { SELECT ITEM_NAME FROM SALES };

    while (iter.next()) {
        System.out.println(iter.ITEM_NAME());
    }
}
}
```

例外の処理

ここでは、SQLJ アプリケーションにおける例外の処理および解析方法について説明します。実行時の例外の発生元は次のいずれかです。

- SQLJ ランタイム
- JDBC ドライバ
- RDBMS

SQLJ ランタイムで発生するエラーの一覧は、B-44 ページの「[SQLJ ランタイム・メッセージ](#)」を参照してください。

Oracle JDBC ドライバで発生するエラーの一覧は、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。Oracle RDBMS で発生するエラーの一覧は、『Oracle9i データベース・エラー・メッセージ』を参照してください。

エラー・テキストの出力

前項の例では、SQL 例外の捕捉方法およびエラー・メッセージの出力方法を示しました。この項でも同じ例を使用します。

```
...
try {
...
} catch (SQLException e) {
    System.err.println("Error running the example: " + e);
}
...
```

この結果、SQLException オブジェクトからのエラー・テキストが出力されます。

エラー情報を取り出すには、SQLException クラスの getMessage()、getErrorCode() および getSQLState() メソッドを使用します。詳細は、次の項で説明します。

この例に示したようにエラー・テキストを出力すると、エラー・メッセージの他に「SQLException」などのテキストも出力されます。

SQL の状態およびエラー・コードの取出し

java.sql.SQLException クラスとサブクラスには、メソッド getMessage()、getErrorCode() および getSQLState() があります。これらのメソッドを使用すると、例外の発生元およびエラー例外の実装方法に応じて、次のような補足情報が得られます。

- String getMessage()

SQLJ ランタイムまたは JDBC ドライバで発生したエラーに対してこのメソッドを使用すると、接頭辞なしのエラー・メッセージが戻されます。RDBMS で発生したエラーに対してこのメソッドを使用すると、接頭辞として ORA 番号の付いたエラー・メッセージが戻されます。

- int getErrorCode()

SQLJ ランタイムで発生したエラーに対しては、このメソッドを使用しても役立つ情報は戻されません。JDBC ドライバまたは RDBMS で発生したエラーに対してこのメソッドを使用すると、ORA 番号が 5 桁の整数として戻されます。

- String getSQLState()

SQLJ ランタイムで発生したエラーに対してこのメソッドを使用すると、SQL の状態を示す 5 桁のコードと文字列が戻されます。JDBC ドライバで発生したエラーに対しては、このメソッドを使用しても役立つ情報は戻されません。RDBMS で発生したエラーに対してこのメソッドを使用すると、SQL の状態を示す 5 桁のコードが戻されます。コードは、NULL の戻り値を扱えるように作成してください。

次に、前述の例と同様にエラー・メッセージを出力し、さらに SQL の状態をチェックする例を示します。

```
...
try {
    ...
} catch (SQLException e) {
    System.err.println("Error running the example: " + e);
    String sqlState = e.getSQLState();
    System.err.println("SQL state = " + sqlState);
}
...
```

SQLException サブクラスの使用法

さらに細かいエラー・チェックを実行するには、`java.sql.SQLException` クラスのサブクラスを使用します。

SQLJ には、`sqlj.runtime.NullException` クラスというエラー・チェック用のサブクラスが 1 つ用意されています。Java の基本型変数に NULL 値が戻された場合は、このクラスでキャッチしてください。(Java の基本型は NULL を処理できないため。)

バッチ処理が有効な環境では、標準 `java.sql.BatchUpdateException` サブクラスを使用できます。詳細は、10-21 ページの「[バッチ実行中のエラー状態](#)」を参照してください。

SQLException サブクラスを使用する際は、事前にサブクラスの例外をキャッチしてから、SQLException をキャッチしてください。次に例を示します。

```
...
try {
    ...
} catch (SQLNullException ne) {
    System.err.println("Null value encountered: " + ne);
} catch (SQLException e) {
    System.err.println("Error running the example: " + e);
}
...
```

このようにするのは、サブクラスの例外も SQLException としてキャッチできるからです。SQLException を先にキャッチすると、サブクラスの例外に対しては特別な処理が行われなくなります。

基本的なトランザクション制御

ここでは、データ更新を管理する方法について説明します。

より詳細なトランザクション制御の機能であるアクセス・モードと分離レベルに対する SQLJ サポートは、7-44 ページの「[詳細なトランザクション制御](#)」を参照してください。

トランザクションの概要

トランザクションは、Oracle で 1 つの単位として順次処理される SQL 操作です。次の処理を行った後の SQL 実行文でトランザクションが開始されます。

- データベースへの接続
- COMMIT（データの更新を自動的にまたは手動でデータベースにコミットします）
- ROLLBACK（変更内容を取り消します）

トランザクションは、COMMIT 操作または ROLLBACK 操作で終了します。

注意： Oracle9i では、どの DDL コマンド（CREATE や ALTER など）を実行したときにも、暗黙的な COMMIT が実行されます。この場合、DDL コマンドの他、前に実行した DML コマンド（INSERT、DELETE、UPDATE など）のうちコミットやロールバックが未実行のものは、すべてコミットの対象になります。

自動コミットと手動コミットとの違い

SQLJ または JDBC を使用すると、自動でも手動でも、データの更新をコミットできます。どちらの場合も、新規のトランザクションは COMMIT 操作で開始されます。変更内容の自動コミットを指定するには、SQLJ 接続を定義するときの自動コミット・フラグを使用可能にするか、または既存の接続を表す JDBC の **Connection** オブジェクトに用意されている `setAutoCommit()` メソッドを使用します。手動で制御するには、自動コミット・フラグを使用禁止にし、SQLJ の COMMIT 文および ROLLBACK 文を使用します。

自動コミットを使用可能にすると、手間が省けますが、細かい制御ができません。たとえば、変更内容のロールバックができません。また、自動コミット・モードでは、SQLJ や JDBC の一部の機能を使用できません。そのため、バッチ更新や SELECT FOR UPDATE 構文を使用する場合などは、自動コミット・フラグを使用禁止にする必要があります。

接続を定義する際の自動コミットの指定

`Oracle.connect()` または `Oracle.getConnection()` メソッドで `DefaultContext` のインスタンスを作成し、接続を定義すると、自動コミット・フラグがデフォルトで `false` に設定されます。ただし、これらのメソッドのシグネチャでは、このフラグを明示的に設定できます。自動コミット・フラグは、必ず最後のパラメータで指定します。

次に、`DefaultContext` のインスタンスを作成し、自動コミット・モードのデフォルト値 `false` を使用する例を示します。

```
Oracle.getConnection  
("jdbc:oracle:thin:@localhost:1521:orcl", "scott", "tiger");
```

`true` に設定するには、次のように指定します。

```
Oracle.getConnection  
("jdbc:oracle:thin:@localhost:1521:orcl", "scott", "tiger", true);
```

`Oracle.connect()` および `Oracle.getConnection()` のシグネチャの一覧は、4-12 ページの「[Oracle クラスについて](#)」を参照してください。

コンストラクタで接続コンテキストのインスタンスを作成する場合は、`DefaultContext` クラスの場合でも、宣言済みの接続コンテキスト・クラスの場合でも、自動コミットの設定を指定する必要があります。この設定のフラグも、次のように最後のパラメータで指定します。

```
DefaultContext ctx = new DefaultContext  
("jdbc:oracle:thin:@localhost:1521:orcl", "scott", "tiger", false);
```

`DefaultContext` コンストラクタのシグネチャの一覧は、4-14 ページの「[DefaultContext クラスについて](#)」を参照してください。

クライアント側で実行するプログラムの場合は、JDBC の `Connection` インスタンスを直接生成すると、自動コミット・フラグがデフォルトで `true` に設定されます。サーバー側で実行するプログラムの場合は、デフォルトで `false` に設定されます。JDBC の `Connection` インスタンスを直接生成するときは、自動コミットの設定値を指定できません。ただし、`setAutoCommit()` メソッドを使用すると、この設定値を変更できます。詳細は、後述の「[既存の接続の自動コミットに対する変更](#)」を参照してください。

注意： 自動コミット機能は、サーバー側の JDBC 内部ドライバではサポートされていません。

既存の接続の自動コミットに対する変更

既存の接続の場合は、自動コミット・フラグの設定変更は通常不要ですが、必要な場合は変更できます。変更するには、JDBC の基底 Connection オブジェクトの `setAutoCommit()` メソッドを使用します。

JDBC の基底 Connection オブジェクトを取得するには、SQLJ の接続コンテキスト・インスタンスの `getConnection()` メソッドを使用します。つまり、`DefaultContext` クラスのインスタンスのメソッドを使用するか、または宣言済みの接続コンテキスト・クラスのインスタンスのメソッドを使用します。

この 2 つの操作を一度に行うには、次のように指定します。この例では、`ctx` を SQLJ の接続コンテキストのインスタンスとしています。

```
ctx.getConnection().setAutoCommit(false);
```

または

```
ctx.getConnection().setAutoCommit(true);
```

重要： トランザクションの処理中には、自動コミットの設定を変更できません。

手動 COMMIT または ROLLBACK の使用方法

自動コミット・フラグを無効化した場合は、データの更新を手動でコミットする必要があります。

最終の COMMIT 操作以降に加えられた変更内容（更新、挿入または削除など）をコミットするには、次のように SQLJ の COMMIT 文を使用します。

```
#sql { COMMIT };
```

最終の COMMIT 操作以降に加えられた変更内容のロールバック（取消し）を行うには、次のように SQLJ の ROLLBACK 文を使用します。

```
#sql { ROLLBACK };
```

自動コミットを使用可能にしたときは、COMMIT コマンドや ROLLBACK コマンドを使用しないでください。使用した場合は、意図しない動作または SQL 例外が発生します。

注意：

- 指定したセーブポイントをロールバックすることもできます。4-28 ページの「[セーブポイントの使用](#)」を参照してください。
 - Oracle SQL のすべての DDL 文には、暗黙的な COMMIT 操作が組み込まれています。SQLJ には、この機能はありません。DDL 文は標準の Oracle SQL 規則に従います。
 - 自動コミット・モードがオフになっていると、クライアント・アプリケーションから接続コンテキスト・インスタンスを終了したときに、最終の COMMIT 操作以降の変更内容がコミットされます。ただし、接続コンテキストのインスタンスを `KEEP_CONNECTION` で終了した場合は除きます (7-52 ページの「[共有接続の終了](#)」を参照してください)。
-

イテレータおよび結果セットに対するコミットおよびロールバックの影響

COMMIT 操作 (自動または手動) および ROLLBACK 操作は、オープンしている結果セットとイテレータには反映されません。結果セットとイテレータはオープンしたままであり、通常それぞれの内容は SELECT 文を実行したときのデータベースの状態が引き続き反映されています。

注意： この例外は、イテレータ・クラスを `sensitivity=SENSITIVE` と宣言している場合です。この場合、イテレータをウィンドウ・サイズの外部でスクロールすると、基になる結果セットへの変更が表示されます。スクロール可能なイテレータの詳細は、7-38 ページの「[スクロール可能なイテレータ](#)」を参照してください。基になるスクロール可能な結果セットの詳細は、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。

UPDATE、INSERT および DELETE 文を SELECT 文の後に実行した場合も同じです。これらの文を実行しても、オープンしている結果セットとイテレータの内容は変わりません。

SELECT、UPDATE および COMMIT の各操作をこの順に実行した場合を考えてください。UPDATE および COMMIT を実行しても、`sensitive` ではない結果セットやイテレータの内容は、SELECT 文で設定されたときの状態のまま変更されません。

さらに、UPDATE、SELECT および ROLLBACK の操作をこの順に実行したとします。SELECT で移入された `sensitive` でない結果セットやイテレータには、依然として更新データが格納されており、その後に実行された ROLLBACK の影響は受けません。

セーブポイントの使用

JDBC 3.0 の仕様では、セーブポイントのサポートが追加されました。セーブポイントとは、トランザクション全体をロールバックするのではなく、必要に応じてトランザクション内でロールバックできるように定義された位置です。Oracle SQLJ および JDBC では、1.1 以上のいずれの JDK バージョンでも使用できるように、Oracle9i リリース 2 (9.2) からセーブポイントをサポートしています。

SQLJ では、セーブポイントの次の文をサポートしています。

```
#sql { SET SAVEPOINT :savepoint };  
...  
#sql { ROLLBACK TO :savepoint };  
...  
#sql { RELEASE :savepoint };
```

セーブポイントとは、SET SAVEPOINT 文が出現するトランザクション内の位置です。savepoint ホスト式では、セーブポイントの名前を Java 文字列で指定します。その後、指定したセーブポイントまでロールバックするか、セーブポイントを解放（削除）できます。

セーブポイントは、SQLJ 実行コンテキストに保存されます。SQLJ 実行コンテキストには、前述の 3 つの文の機能に相当するメソッドがあります。7-29 ページの「セーブポイント・メソッド」を参照してください。

COMMIT 操作によってトランザクションが終了するため、トランザクション内のセーブポイントもすべて解放されます。トランザクションの終了には、手動 COMMIT 操作、自動 COMMIT 操作および DDL 文（結果として、自動 COMMIT になります）が含まれます。

注意： Oracle9i リリース 2 (9.2) の時点で、Oracle9i および Oracle9i JDBC は、セーブポイントの解放機能をサポートしていません。

要約：簡単な SQLJ コード

これまでに説明した SQLJ 実行文の特長と機能について要点を把握するには、簡潔で包括的なプログラムを試してみるのが一番効果的な方法です。ここでは、2 つの例を示します。

最初の例では、まず 1 操作ずつ例を示し、その後で各操作を包括した例を示します。この例では、SELECT INTO 文を使用して、従業員表の 2 つの列に対して単一行問合せを実行します。この例を実行する場合は、connect.properties ファイル中のパラメータを、目的のデータベースに接続するための設定に変更する必要があります。

2 番目の例は多少複雑です。SQLJ のイテレータを使用して複数行問合せを実行します。

必要なクラスのインポート

必要な JDBC または SQLJ パッケージをすべてインポートします。少なくとも、`java.sql` パッケージ内のクラスがいくつか必要です。

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

ただし、`java.sql` パッケージ内のクラスを、必ずしもすべて使用するわけではありません。主に使用するクラスとしては、`java.sql.SQLException` と、明示的に参照するクラス (`java.sql.Date` や `java.sql.ResultSet` など) があります。

Oracle クラスには、次のパッケージが必要です。通常、`DefaultContext` オブジェクトをインスタンス化し、デフォルト接続を確立するときに、このクラスを使用します。

```
import oracle.sqlj.runtime.*;
```

SQLJ のランタイム・クラスをコード中に直接使用する場合は、次のパッケージをインポートします。

```
import sqlj.runtime.*;  
import sqlj.runtime.ref.*;
```

これに対し、コード中に SQLJ のランタイム・クラスを直接使用しない場合は、ランタイム・クラスを `CLASSPATH` に指定するのみで十分です。2-11 ページの「[PATH と CLASSPATH の設定](#)」を参照してください。

重要なランタイム・クラスとしては、`sqlj.runtime` パッケージの `ResultSetIterator` および `ExecutionContext` と、`sqlj.runtime.ref` パッケージの `DefaultContext` などがあります。

JDBC ドライバの登録およびデフォルト接続の設定

デフォルトの接続を設定するには、`static Oracle.connect()` メソッドを使用するコンストラクタで、`SimpleExample` クラスを宣言します。この結果、Oracle JDBC ドライバの登録も行われます。Oracle 以外の JDBC ドライバを使用する場合は、ドライバ登録用のコードを追加する必要があります (次に示したコードのコメントを参照してください)。

ここでは、URL、ユーザー名およびパスワードを `connect.properties` ファイルから取得する `connect()` のシグネチャを使用します。このファイルの例は、`$Oracle_Home/sqlj/demo` ディレクトリおよび 2-13 ページの「[ランタイム接続の設定](#)」にあります。

```
public class SimpleExample {

    public SimpleExample() throws SQLException {
        /* If you are using a non-Oracle JDBC driver, add a call here to
           DriverManager.registerDriver() to register your driver.  */
        // Set default connection (as defined in connect.properties).
        Oracle.connect(getClass(), "connect.properties");
    }
}
```

main() メソッドは、後述の「[例外処理の設定](#)」に定義されています。

例外処理の設定

main() を作成します。このメソッドから SimpleExample コンストラクタをコールし、try/catch ブロックをセットアップします。このブロックでは、runExample() メソッドにより発生した SQL 例外を処理します。このアプリケーションの実際の処理は、このメソッドが行います。

```
...
public static void main (String [] args) {

    try {
        SimpleExample o1 = new SimpleExample();
        o1.runExample();
    }
    catch (SQLException ex) {
        System.err.println("Error running the example: " + ex);
    }
}
...
```

runExample() メソッドは、後述の「[ホスト変数のセットアップ、SQLJ 句の実行、結果の処理](#)」に定義されています。

接続をクローズするには、try/catch ブロックを finally 句の内部に使用します。SQL 例外の場合に finally 句が try/catch ブロック内部に使用されていないことが前提です。

```
finally
{
    try { Oracle.close(); } catch(SQLException ex) {...}
}
```


ホスト変数のセットアップ、SQLJ 句の実行、結果の処理

次の処理を行う `runExample()` メソッドを作成します。

1. `main()` メソッドに対して SQL 例外を発生させ、処理させます。
2. Java のホスト変数を宣言します。
3. SQLJ 句を実行します。Java のホスト変数が埋込み `SELECT` 文にバインドされ、データがホスト変数に取り込まれます。
4. 結果を出力します。

次にコードを示します。

```
void runExample() throws SQLException {

    System.out.println( "Running the example--" );

    // Declare two Java host variables--
    Float salary;
    String empname;

    // Use SELECT INTO statement to execute query and retrieve values.
    #sql { SELECT ename, sal INTO :empname, :salary FROM emp
          WHERE empno = 7499 };

    // Print the results--
    System.out.println("Name is " + empname + ", and Salary is " + salary);
}

// Closing brace of SimpleExample class
```

前述のコード例では、Java のホスト変数として `salary` と `ename` が宣言されています。その次の SQLJ 句の処理では、EMP 表の ENAME 列と SAL 列から選択されたデータが、ホスト変数に格納されます。最後に、`salary` と `empname` の値が出力されます。

この `SELECT` 文では、EMP 表から選択できる行は 1 行のみです。WHERE 句の EMPNO 列が表のプライマリになっているためです。

SELECT INTO を使用した単一行問合せの例

ここでは、前に1つずつ順を追って説明した内容を SimpleExample クラスを使用して総復習します。次の例は単一行問合せの例であるため、イテレータは使用しません。

```
// Import SQLJ classes:
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import oracle.sqlj.runtime.*;

// Import standard java.sql package:
import java.sql.*;

public class SimpleExample {

    public SimpleExample() throws SQLException {
        /* If you are using a non-Oracle JDBC driver, add a call here to
           DriverManager.registerDriver() to register your driver.  */
        // Set default connection (as defined in connect.properties).
        Oracle.connect(getClass(), "connect.properties");
    }

    public static void main (String [] args) throws SQLException {

        try {
            SimpleExample o1 = new SimpleExample();
            o1.runExample();
        }
        catch (SQLException ex) {
            System.err.println("Error running the example: " + ex);
        }
    }

    finally
    {
        try { Oracle.close(); } catch(SQLException ex) {...}
    }

    void runExample() throws SQLException {

        System.out.println( "Running the example--" );

        // Declare two Java host variables--
        Float salary;
        String empname;

        // Use SELECT INTO statement to execute query and retrieve values.
        #sql { SELECT ename, sal INTO :empname, :salary FROM emp
              WHERE empno = 7499 };
    }
}
```

```

        // Print the results--
        System.out.println("Name is " + empname + ", and Salary is " + salary);
    }
}

```

名前指定イテレータの設定

次の例は、前の例がベースになっていますが、それに加えて名前指定イテレータを使用し、複数行問合せを実行します。

最初に、イテレータ・クラスを宣言します。NULL 値が戻される可能性がある場合は、基本型 `int` と `float` ではなくて、オブジェクト型 `Integer` と `Float` を使用してください。

```

#sql iterator EmpRecs(
    int empno,          // This column cannot be null, so int is OK.
                        // (If null is possible, use Integer.)
    String ename,
    String job,
    Integer mgr,
    Date hiredate,
    Float sal,
    Float comm,
    int deptno);

```

次に `EmpRecs` クラスをインスタンス化し、問合せ結果を移入します。

```

EmpRecs employees;

#sql employees = { SELECT empno, ename, job, mgr, hiredate,
                        sal, comm, deptno FROM emp };

```

次に、イテレータの `next()` メソッドで結果を出力します。

```

while (employees.next()) {
    System.out.println( "Name:      " + employees.ename() );
    System.out.println( "EMPNO:     " + employees.empno() );
    System.out.println( "Job:       " + employees.job() );
    System.out.println( "Manager:   " + employees.mgr() );
    System.out.println( "Date hired: " + employees.hiredate() );
    System.out.println( "Salary:    " + employees.sal() );
    System.out.println( "Commission: " + employees.comm() );
    System.out.println( "Department: " + employees.deptno() );
    System.out.println();
}

```

ここまでの操作が完了した後、イテレータを終了します。

```

employees.close();

```

名前指定イテレータを使用した複数行問合せの例

この例では、名前指定イテレータを使用して複数行問合せを行い、従業員表から複数のデータ列を選択します。

この例は、名前指定イテレータを使用していることを除けば、前に示した単一行問合せの例と概念はほとんど同じです。

```
// Import SQLJ classes:
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import oracle.sqlj.runtime.*;

// Import standard java.sql package:
import java.sql.*;

// Declare a SQLJ iterator.
// Use object types (Integer, Float) for mgr, sal, And comm rather
// than primitive types to allow for possible null selection.

#sql iterator EmpRecs(
    int empno,          // This column cannot be null, so int is OK.
                        // (If null is possible, Integer is required.)
    String ename,
    String job,
    Integer mgr,
    Date hiredate,
    Float sal,
    Float comm,
    int deptno);

// This is the application class.
public class EmpDemo1App {

    public EmpDemo1App() throws SQLException {
        /* If you are using a non-Oracle JDBC driver, add a call here to
           DriverManager.registerDriver() to register your driver. */
        // Set default connection (as defined in connect.properties).
        Oracle.connect(getClass(), "connect.properties");
    }

    public static void main(String[] args) {

        try {
            EmpDemo1App app = new EmpDemo1App();
            app.runExample();
        }
        catch( SQLException exception ) {
            System.err.println( "Error running the example: " + exception );
        }
    }
}
```

```
    }
}

finally
{
    try { Oracle.close(); } catch(SQLException ex) {...}
}

void runExample() throws SQLException {
    System.out.println("\nRunning the example.\n" );

    // The query creates a new instance of the iterator and stores it in
    // the variable 'employees' of type 'EmpRecs'. SQLJ translator has
    // automatically declared the iterator so that it has methods for
    // accessing the rows and columns of the result set.

    EmpRecs employees;

    #sql employees = { SELECT empno, ename, job, mgr, hiredate,
                          sal, comm, deptno FROM emp };

    // Print the result using the iterator.

    // Note how the next row is accessed using method 'next()', and how
    // the columns can be accessed with methods that are named after the
    // actual database column names.

    while (employees.next()) {
        System.out.println( "Name:      " + employees.ename() );
        System.out.println( "EMPNO:    " + employees.empno() );
        System.out.println( "Job:      " + employees.job() );
        System.out.println( "Manager:  " + employees.mgr() );
        System.out.println( "Date hired: " + employees.hiredate() );
        System.out.println( "Salary:   " + employees.sal() );
        System.out.println( "Commission: " + employees.comm() );
        System.out.println( "Department: " + employees.deptno() );
        System.out.println();
    }

    // You must close the iterator when it's no longer needed.
    employees.close() ;
}
}
```

Oracle 固有コード生成（プロファイルなし）

このマニュアルでは、全体にわたって一般的かつ標準的な SQLJ ランタイム・レイヤーおよび SQLJ プロファイルについて説明しています。ただし、Oracle9i リリース 2 (9.2) のデフォルトでは、Oracle SQLJ は、SQL 操作の SQLJ ランタイムをコールし、次に Oracle JDBC をコールする ISO 標準コードを生成するのではなく、Oracle JDBC コードを直接コールする Oracle 固有コードを生成します。Oracle 固有コード生成の場合、プロファイルは生成されず、SQLJ ランタイム・レイヤーの役割はプログラム実行時に大幅に縮小されます。

Oracle 固有コードでは、すべての Oracle 固有拡張機能をサポートしています。

コード生成は、SQLJ トランスレータの `-codegen` オプションによって決まります。デフォルトの設定は、`-codegen=oracle`（Oracle 固有コード生成）です。ISO の標準に従ったコードを生成する場合は、`-codegen=iso` を設定できます。

このオプションの構文は、8-50 ページの「[コードの生成 \(-codegen\)](#)」を参照してください。

ここでは、次の項目について説明します。

- [Oracle 固有コード生成のメリットとデメリット](#)
- [Oracle 固有コード生成用環境の要件](#)
- [Oracle 固有コード生成でのコード上の考慮事項と制限事項](#)
- [Oracle 固有コード生成時の SQLJ 使用方法の変更](#)
- [Oracle 固有コード生成でのサーバー側の考慮事項](#)

Oracle 固有コード生成のメリットとデメリット

Oracle 固有コード生成を使用すると、ISO 標準コード生成よりも多くのメリットがあります。

- アプリケーションの実行がより効率的になります。コードでは JDBC API を直接コールするので、ランタイムのパフォーマンスが JDBC レベルになります。中間的な SQLJ ランタイム・レイヤーの役割は、プログラム実行時に大幅に縮小されます。
- アプリケーションのサイズが小さくなります。
- プロファイル・ファイル（.ser）は生成されません。変換されたアプリケーションをデータベースにロードするか、別のシステムに移植する場合、コンポーネントが少ないので、これは特に便利です。
- プロファイルのカスタマイズ処理がないので、変換が高速になります。
- 実行時に、Oracle SQLJ と Oracle JDBC は同じ文キャッシュ・リソースを使用するため、Oracle SQLJ と Oracle JDBC 間のリソースのパーティション化が不要です。
- プロファイル・ファイルではなく Java クラス・ファイルに SQL 固有情報があると、潜在的なセキュリティの問題を回避できます。

- 静的 SQL コードの実行のために考慮されている拡張機能など、将来サポートされる Oracle JDBC のパフォーマンス強化のメリットを活用するために、コードを再度記述する必要はありません。Oracle SQLJ トランスレータの将来のリリースではこれが自動的に処理できるようになります。
- ブラウザ環境に完全な移植性が備わるので、Java リフレクションが実行時に使用されることが少なくなります。

デメリットはあまり多くありません。

- Oracle 固有生成コードは、SQLJ 標準に準拠していません。また、汎用 JDBC プラットフォームに移植できません。
- プロファイル固有の機能は使用できません。たとえば、後から Oracle カスタマイザ・ハーネスの `-debug`、`-verify` および `-print` オプションを使用してカスタマイズを実行することができません。（これらのオプションについては、A-20 ページの「[専用のカスタマイズの起動に使用するカスタマイザ・ハーネスのオプション](#)」で説明しています。`-debug` オプションで発行される AuditorInstaller については、A-42 ページの「[デバッグ用の AuditorInstaller カスタマイザ](#)」を参照してください。）

Oracle 固有コード生成用環境の要件

Oracle 固有コード生成を使用する場合は、次の環境の要件に注意してください。

- Oracle 固有コード生成では JDBC の文キャッシング機能が必要なので、Oracle9i JDBC ドライバを使用する必要があります。Oracle8i 以下の JDBC リリースでは動作しません。
- 汎用 SQLJ ランタイム・ライブラリである runtime および runtime-nonoracle は、Oracle 固有コード生成用にサポートされていません。次の Oracle SQLJ ランタイム・ライブラリのうち 1 つが、CLASSPATH に指定されている必要があります。
 - runtime11.jar (または .zip)
 - runtime12.jar (または .zip)
 - runtime12ee.jar (または .zip)

これらのランタイム・ライブラリの詳細は、2-3 ページの「[Oracle SQLJ を使用するための要件](#)」に説明されています。

Oracle 固有コード生成でのコード上の考慮事項と制限事項

Oracle 固有コード生成が使用される SQLJ アプリケーションのコーディング時には、次のプログラミング上の考慮事項および制限に注意してください。

- デフォルト以外の文キャッシュ・サイズを使用するには、Oracle カスタマイザの `stmtcache` オプションが使用できないので、コードにメソッド・コールを記述する必要があります。4-39 ページの「[Oracle 固有コード生成時の SQLJ 使用方法の変更](#)」を参照してください。
- 同じアプリケーションに、Oracle 固有生成コードと ISO 標準生成コードを混在しないください。

ただし、Oracle 固有コードと ISO 標準コードが同じ接続を共有する必要がある場合には、次のいずれかを行ってください。

- Oracle 固有コードと ISO 標準コードで、必ず異なる SQLJ 実行コンテキスト・インスタンスを使用するようにします。（SQLJ 実行コンテキストについては、7-22 ページの「[実行コンテキスト](#)」を参照してください。）

または

- トランザクション境界（手動の COMMIT または ROLLBACK）を 2 種類のコードの間に配置します。

コードの混在に関するこの制限事項は、あるセッションで実行中のすべての Java コードが同じ JDBC 接続と SQLJ 接続コンテキストを使用するので、サーバー側のコードで特に重要になります。（4-41 ページの「[Oracle 固有コード生成でのサーバー側の考慮事項](#)」も参照してください。）

- データベースから値が戻られるとき、パラメータ表現での副作用に頼らないでください。Oracle 固有コード生成では、OUT パラメータ、INOUT パラメータ、SELECT INTO 変数または SQL 文の戻り引数を評価するための一時変数が作成されません。

たとえば、次のような文は使用しないでください。

```
#sql { SELECT * FROM EMP INTO :x[i++], :f_with_sideeffect()[i++],  
      :a.b[i] };
```

または

```
#sql x[i++] = { VALUES f(:INOUT (x[i++] ), :OUT (f_with_sideeffect())) };
```

引数の評価は、生成コードの適切な場所で実行されます。これによって、ISO SQLJ 標準に従って評価が実施される場合と、動作が異なることになります。

副作用および例は、3-20 ページの「[Java 式の実行時評価](#)」および 3-21 ページの「[Java 式の実行時評価の例（ISO コード生成）](#)」を参照してください。

- Oracle オブジェクト機能の型マップを使用する場合（JPublisher で生成された Java クラスで型マップが必要ない場合に、Java クラスで `java.sql.SQLData` インタフェースを実装することが前提です）、イテレータ宣言と接続コンテキスト宣言に同じ型マップを指定する必要があります。これは、`with` 句の宣言で指定します。

たとえば、接続コンテキスト・クラスを宣言する場合、次のようにします。

```
#sql context TypeMapContext with (typeMap="MyTypeMap");
```

また、この接続コンテキスト・クラスのインスタンスを使用する SQLJ 文からイテレータ・インスタンスを移入するには、次のようにします。

```
TypeMapContext tmc = new TypeMapContext(...);
...
MyIterator it;
#sql [tmc] it = ( SELECT pers, addr FROM tab WHERE ...);
```

イテレータ宣言では同じ型マップを指定する必要があります。次のようにします。

```
#sql iterator MyIterator with (typeMap="MyTypeMap")
    (Person pers, Address addr);
```

型マップについては、6-10 ページの「[カスタム Java クラスの要件](#)」を参照してください。with 句の概要は、3-6 ページの「[WITH 句宣言](#)」を参照してください。

注意： この制限事項の理由は、Oracle 固有コード生成の場合、イテレータを取得するすべてのメソッドが変換時に完全に Oracle JDBC コールとして生成されているためです。適切なコールを生成するには、イテレータがある特定の型マップで使用されるかどうかを SQLJ トランスレータで把握する必要があります。

Oracle 固有コード生成時の SQLJ 使用方法の変更

以前に Oracle カスタマイザのオプションとしてのみ使用していたオプションで、Oracle 固有コード生成でも便利なオプションがあります。プロファイルのカスタマイズは Oracle 固有コード生成には適用できないので、これらのオプションは他の方法で使えるようになりました。

文キャッシュのサイズを変更したり、Oracle 固有コードの生成時に文のキャッシングを無効化するには、カスタマイザ `stmtcache` オプションではなく、メソッド・コールをコードに使用します。`sqlj.runtime.ref.DefaultContext` クラスと同様に、宣言するすべての接続コンテキスト・クラスには、現在次の `static` メソッドがあります。

- `setDefaultStmtCacheSize(int)`
- `int getDefaultStmtCacheSize()`

また、次のインスタンス・メソッドがあります。

- `setStmtCacheSize(int)`
- `int getStmtCacheSize()`

デフォルトで、文のキャッシングを使用できます。

詳細は、10-5 ページの「[文キャッシングのための接続コンテキスト・メソッド（Oracle 固有コードの場合）](#)」を参照してください。（これは、文のキャッシングの概要について説明する 10-4 ページの「[文のキャッシング](#)」の一部分です。）

さらに、次のオプションは、Oracle カスタマイザのオプションと同様に、フロントエンドの Oracle SQLJ トランスレータとして使用できます。

- `-optcols`: パフォーマンスが最適化されるようにイテレータ列型およびサイズ定義を有効化するためのオプション。
- `-optparams`: JDBC リソース割当てが最適化されるようにパラメータ・サイズ定義を有効化するためのオプション（`optparamdefaults` と併用します）。
- `-optparamdefaults`: 特定のデータ型に対してパラメータ・サイズのデフォルトを設定するためのオプション（`optparams` と併用します）。
- `-fixedchar`: WHERE 句に関して、空白埋めを考慮した CHAR 比較を有効にするためのオプション。

これらのオプションの詳細は、8-49 ページの「[コード生成、最適化および CHAR 比較のオプション](#)」を参照してください。

次の点に留意してください。

- オンラインのセマンティクス・チェックを使用している場合のみ（SQLJ トランスレータの `-user`、`-password` および `-url` オプションを使用して、変換時にデータベース接続を要求している場合）、`-optcols` オプションを使用します。
- `-optcols`、`-optparams` および `-optparamdefaults` オプションの機能とデフォルト値は、カスタマイザのオプションと同じです。

Oracle 固有コード生成でのサーバー側の考慮事項

SQLJ コードをサーバーで実行する場合は、次の考慮事項に留意してください。

- サーバー側 SQLJ トランスレータでは、ISO 標準生成コードはサポートされません。サーバーにロードされコンパイルされた SQLJ ソース・コードは、常にデフォルトの `-codegen=oracle` 設定で変換されます。

したがって、ISO 標準生成コードをサーバーで使用するには、クライアント上で SQLJ コードを変換し、コンパイルしてから、個々のコンポーネントをサーバーにロードする必要があります。（11-8 ページの「[クライアント側での SQLJ ソースの変換とコンポーネントのロード](#)」を参照してください。）

- Oracle 固有生成コードと ISO 標準生成コードの混在に対する注意（4-38 ページの「[Oracle 固有コード生成でのコード上の考慮事項と制限事項](#)」を参照）は、Java ストアド・プロシージャ（またはストアド・ファンクション）をコールするサーバー側 Java コードに適用されます。このストアド・プロシージャが PL/SQL ラッパーから発行されていても同様です。これは再帰的コールインにも当てはまります。デフォルトで、`ExecutionContext` オブジェクトは、コール側のモジュールとコールされる側のモジュールで共有されています。そのため、2 つのモジュールは同じ `-codegen` を設定して変換する必要があります。

ISO 標準コード生成で変換したコードと連係して動作するには、次の例のように、実行コンテキスト・インスタンスを明示的にインスタンス化することをお勧めします。

```
public static method() throws SQLException
{
    Execution Context ec = new ExecutionContext();
    ...
    try {
        ...
        #sql [ec] { SQL operation };
        ...
    } finally { ec.close(); }
}
```

重要： 明示的な `ExecutionContext` インスタンスを使用する際にリソースのリークを回避するには、この例のように、`close()` メソッドを使用します。

ネーミングの要件および制限事項

ネーミング要件、ネーミング制限および予約語については、次の 4 点を考慮する必要があります。

- Java のネームスペース。ローカル変数とクラスの名前については、SQLJ 固有の制限もあります。
- SQLJ のネームスペース。
- SQL のネームスペース。
- ソース・ファイル名。

Java のネームスペース：ローカル変数およびクラスのネーミングに関する制限事項

Java のネームスペースによる制限は、標準 Java のすべての文および宣言（Java のクラスやローカル変数のネーミングなど）に適用されます。標準 Java のネーミングに関するあらゆる制限が適用されるので、Java の予約語は使用しないでください。

また、ローカル変数とクラスのネーミングには、SQLJ 固有の制限も若干伴います。

注意： ホスト変数名固有の制約については、3-29 ページの「[ホスト式の制限事項](#)」を参照してください。

ローカル変数のネーミングに関する制限

ローカル変数のネーミングの際は、SQLJ トランスレータの一部の機能に起因する制限が若干伴います。

SQLJ トランスレータは各 SQLJ 実行文を文ブロックに置き換えます。このブロックでは、次のように標準構文に基づく SQLJ 実行文が使用されます。

```
#sql { SQL operation };
```

SQLJ では、生成された文ブロックの中に一時変数を宣言できます。この一時変数の名前には、次の接頭辞が付きます。

```
__sJT_
```

（アンダースコアが先頭に 2 つ、末尾に 1 つ付きます。）

SQLJ で生成した文ブロックでは、次のような宣言が使用されます。

```
int __sJT_index;  
Object __sJT_key;  
java.sql.PreparedStatement __sJT_stmt;
```

文字列 `__sJT_` は、SQLJ で生成した変数名の接頭辞として予約されています。SQLJ のプログラミングの際は、この文字列を次の名前の接頭辞としては使用しないでください。

- SQL 実行文を含むブロックで宣言した変数名
- SQL 実行文を含むメソッドのパラメータ名
- SQL 実行文を含むクラスのフィールド名、あるいはサブクラスまたは被包含クラスに SQL 実行文があるクラスのフィールド名

クラスのネーミングに関する制限

SQLJ アプリケーションのクラスのネーミングには、次のような制限が伴います。

- SQLJ の内部クラスと同じクラス名は宣言できません。具体的には、SQLJ アプリケーション内の既存クラスの名前が `a` の場合、最上位クラスに次の形式の名前は付けられません。

`a_sJb` (`a` と `b` は正当な Java 識別子)

たとえば、ファイル `Foo.sqlj` のアプリケーション・クラスが `Foo` であると、SQLJ ではプロファイルキー・クラス `Foo_sJProfileKeys` が生成されます。宣言するクラス名は、この名前と同じ名前にしないでください。

- SQLJ 実行文が含まれているクラスの名前は、アプリケーションで使用している Java 型を格納したパッケージの名前の先頭部分と同じにならないようにしてください。たとえば、`java`、`sqlj`、`oracle` など（大 / 小文字区別）は、クラス名としては使用できません。また、SQLJ 文に使用するホスト変数の型が `abc.def.MyClass` である場合、このホスト変数を使用するクラスの名前には `abc` を使用できません。

こうした制限を回避するには、Java のネーミング規則に従い、パッケージ名の先頭を小文字にし、クラス名の先頭を大文字にすることをお薦めします。

SQLJ のネームスペース

SQLJ のネームスペースは、`#sql` クラス宣言と `#sql` 実行文の中カッコの外側にある部分です。

注意： イテレータ列のネーミングに伴う特別な制限については、3-38 ページの「[名前指定イテレータの使用方法](#)」を参照してください。

SQLJ の次の予約語は、宣言する接続コンテキスト・クラスまたはイテレータ・クラスのクラス名として、`with` 句または `implements` 句で使用できません。イテレータ列の型宣言リストでも使用できません。

- `iterator`
- `context`
- `with`

たとえば、イテレータ・クラスまたはインスタンスの名前を `iterator` にしたり、接続コンテキスト・クラスまたはインスタンスの名前を `context` にしないでください。

ただし、ストアド・ファンクションからの戻り値を格納する変数の名前には、これらのどのワードでも使用できます。

SQL のネームスペース

SQL のネームスペースは、SQLJ 実行文の中カッコ内の部分です。このネームスペースには、通常の SQL のネーミングに伴う制限が課されます。詳細は、『Oracle9i SQL リファレンス』を参照してください。

ただし、ホスト式は、SQL のネームスペースの規則ではなく、Java のネームスペースの規則に従います。この規則は、ホスト変数の名前にも、ホスト式の外部のカッコで囲まれたすべての部分にも適用されます。

ファイル名の要件と制限

SQLJ ソース・ファイルの拡張子は `.sqlj` です。ソース・ファイルに `Public` クラスが宣言してある場合、この `Public` クラス名をソース・ファイルのベース名に使用してください（大 / 小文字区別）。なお、1 つのソース・ファイルに宣言できる `Public` クラスは 1 つのみです。ソース・ファイルは、`Public` クラスが宣言されていない場合でも、Java 識別子として有効な名前を付ける必要があります。ファイル名は、先頭に定義されているクラスの名前と同じにすることをお勧めします。

たとえば、ソース・ファイルに `Public` クラス `MySource` が定義されている場合は、次に示すファイル名を付けてください。

`MySource.sqlj`

注意： これらのファイル・ネーミング要件は、Java 言語仕様であり、SQLJ の仕様ではありません。このネーミング要件は、Oracle9i には直接適用されませんが、それらに従うことをお勧めします。

中間層の SQLJ に関する考慮事項

Oracle9iAS Containers for J2EE (OC4J) 環境など、中間層で SQLJ を実行する場合は特別に考慮する事項があります。

リリース 1 (9.0.1) 以上では、Oracle JDBC ドライバが `oracle.jdbc` パッケージで Oracle 固有のインタフェースを提供します。Oracle SQLJ ライブラリ (`runtime11`、`runtime12` および `runtime12ee`) は、これらのインタフェースをフル活用します。これは、Oracle SQLJ ライブラリに Oracle JDBC リリース 8.1.7 以下との互換性がないためです。

Oracle9iAS 製品の場合、接続はデータ・ソースを介して確立され、通常は旧式の `oracle.jdbc.driver.OracleConnection` クラスではなく、`oracle.jdbc.OracleConnection` インタフェースのインスタンスを戻します。これは分散トランザクション (XA) などの接続機能に必要です。このような機能をサポートするには、接続オブジェクトが新規インタフェースを実装する必要があります。

この結果、Oracle9iAS 中間層環境またはデータ・ソースが使用される他の環境について、次のような処理が必要となります。

- コードの移植性と柔軟性を最大限にするには、`oracle.jdbc.driver.OracleXXX` 型ではなく `oracle.jdbc.OracleXXX` 型を使用します。
- カスタム Java 型 (通常は SQL オブジェクトとコレクション) については、現在使用されていない `oracle.sql.CustomDatum` インタフェースではなく `oracle.sql.ORAData` を実装します。
- SQLJ runtime ライブラリは使用しないでください。かわりに (JDK 環境に応じて) `runtime11`、`runtime12` または `runtime12ee` を使用します。ランタイム・ライブラリは、リリース 8.1.7 などの古い JDBC ドライバとの下位互換性があるため、`oracle.jdbc.OracleXXX` 型ではなく `oracle.jdbc.driver.OracleXXX` 型をサポートしています。

ただし、なんらかの理由で runtime ライブラリを使用する必要がある場合は、変換時に `-profile=false` オプションを設定します。この場合、プログラムで Oracle 固有のカスタマイズは使用されません。したがって、`oracle.jdbc.driver.OracleConnection` インスタンスのかわりに `oracle.jdbc.OracleConnection` インスタンスが渡された場合に障害が発生しません。この環境では、Oracle 固有の機能はサポートされません。

データ・ソースおよび接続 JavaBeans (SQLJ JavaServer Pages の) を介して取得される接続の管理を容易にするために、Oracle SQLJ は `runtime12ee` ライブラリで多数の API を提供しています。

データ・ソースおよび接続 JavaBeans への SQLJ サポートの概要は、次の項目を参照してください。

- 7-12 ページ [「標準データ・ソースのサポート」](#)
- 7-14 ページ [「SQLJ 固有のデータ・ソース」](#)
- 7-18 ページ [「JavaServer Pages 用の SQLJ 固有の接続 JavaBeans」](#)

型のサポート

この章では、Oracle SQLJ でサポートされているデータ型について説明します。サポートされている SQL 型とそれに対応する Java 型の一覧を示し、Oracle8 と Oracle7 との下位互換性について解説します。さらに、ストリームと Oracle 拡張型のサポートについても詳しく説明します。SQLJ でサポートされている Java 型は、ホスト式で使用する型です。

Oracle SQLJ によるユーザー定義型（SQL オブジェクト、オブジェクト参照およびコレクション）のサポートの詳細は、[第 6 章「オブジェクト、コレクションおよび OPAQUE 型」](#)を参照してください。

この章では、次の項目について説明します。

- [ホスト式での型サポート](#)
- [ストリームのサポート](#)
- [JDBC 2.0 LOB 型と Oracle 拡張型のサポート](#)

ホスト式での型サポート

ここでは、Oracle SQLJ でサポートされている型について概説します。具体的には、JDBC 2.0 の型に関して新規にサポートされた点や、Oracle のドライバ 8.0.x および 7.3.x との下位互換性などについて述べます。

各 Oracle SQL 型ごとの正当な Java マッピングに関する完全なリストは、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。

注意： SQLJ（および SQL）では、SQL 型と Java 型との変換が暗黙的に実行されるようになっていました。この型変換は、一般的には便利ですが、予想と異なる結果を招く場合もあります。コードが正確かどうかの確認は、変換時のタイプ・チェックのみに頼らないでください。

サポートされている型の概要

Oracle JDBC ドライバを使用した場合にホスト式で利用できる Java 型を、表 5-1 に示します。また、Java 型と、クラス `oracle.jdbc.OracleTypes` に型コードが定義されている SQL 型と、Oracle9i のデータ型との相関関係もこの表からわかります。

注意： `OracleTypes` クラスは、各 Oracle データ型の型コード（整数の定数）のみを定義します。標準 JDBC 型の `OracleTypes` の値は、標準 `java.sql.Types` の値と同じです。

Java 変数に出力される SQL データは、該当する Java 型に変換されます。SQL に入力される Java 変数は、該当する Oracle データ型に変換されます。

表 5-1 ホスト式でサポートされている型の対応

Java 型	Oracle 型定義	Oracle SQL データ型
標準 JDBC 1.x 型		
boolean	BIT	NUMBER
byte	TINYINT	NUMBER
short	SMALLINT	NUMBER
int	INTEGER	NUMBER
long	BIGINT	NUMBER
float	REAL	NUMBER
double	FLOAT、DOUBLE	NUMBER

表 5-1 ホスト式でサポートされている型の対応（続き）

Java 型	Oracle 型定義	Oracle SQL データ型
java.lang.String	CHAR VARCHAR LONGVARCHAR	CHAR VARCHAR2 LONG
byte[]	BINARY VARBINARY LONGVARBINARY	RAW RAW LONGRAW
java.sql.Date	DATE	DATE
java.sql.Time	TIME	DATE
java.sql.Timestamp	TIMESTAMP TIMESTAMP	DATE TIMESTAMP
java.math.BigDecimal	NUMERIC DECIMAL	NUMBER NUMBER
標準 JDBC 2.0 型		
java.sql.Blob	BLOB	BLOB
java.sql.Clob	CLOB	CLOB
java.sql.Struct	STRUCT	オブジェクト型
java.sql.Ref	REF	参照型
java.sql.Array	ARRAY	コレクション型
java.sql.SQLData を実装したカスタム・オブジェクト・クラス	STRUCT	オブジェクト型
JAVA ラッパー・クラス		
java.lang.Boolean	BIT	NUMBER
java.lang.Byte	TINYINT	NUMBER
java.lang.Short	SMALLINT	NUMBER
java.lang.Integer	INTEGER	NUMBER
java.lang.Long	BIGINT	NUMBER
java.lang.Float	REAL	NUMBER
java.lang.Double	FLOAT、DOUBLE	NUMBER
SQLJ ストリーム・クラス		
sqlj.runtime.BinaryStream	LONGVARBINARY	LONG RAW
sqlj.runtime.CharacterStream	LONGVARCHAR	LONG

表 5-1 ホスト式でサポートされている型の対応（続き）

Java 型	Oracle 型定義	Oracle SQL データ型
sqlj.runtime.AsciiStream (現在は使用されていない。 CharacterStream で使用)	LONGVARCHAR	LONG
sqlj.runtime.UnicodeStream (現在は使用されていない。 CharacterStream で使用)	LONGVARCHAR	LONG
ORACLE 拡張型		
oracle.sql.NUMBER	NUMBER	NUMBER
oracle.sql.CHAR	CHAR	CHAR
oracle.sql.RAW	RAW	RAW
oracle.sql.DATE	DATE	DATE
oracle.sql.TIMESTAMP	TIMESTAMP	TIMESTAMP
oracle.sql.TIMESTAMPTZ	TIMESTAMPTZ	TIMESTAMP-WITH-TIMEZONE
oracle.sql.TIMESTAMPLTZ	TIMESTAMPLTZ	TIMESTAMP-WITH-LOCAL-TIMEZONE
oracle.sql.ROWID	ROWID	ROWID
oracle.sql.BLOB	BLOB	BLOB
oracle.sql.CLOB	CLOB	CLOB
oracle.sql.BFILE	BFILE	BFILE
oracle.sql.STRUCT	STRUCT	オブジェクト型
oracle.sql.REF	REF	参照型
oracle.sql.ARRAY	ARRAY	コレクション型
oracle.sql.OPAQUE	OPAQUE	OPAQUE 型
oracle.sql.ORAData を実装した カスタム・オブジェクト・クラス	STRUCT	オブジェクト型
oracle.sql.ORAData を実装した カスタム参照クラス	REF	参照型
oracle.sql.ORAData を実装した カスタム・コレクション・クラス	ARRAY	コレクション型

表 5-1 ホスト式でサポートされている型の対応（続き）

Java 型	Oracle 型定義	Oracle SQL データ型
OPAQUE 型の oracle.sql.ORAData を実装したカスタム・クラス (例 : oracle.xdb.XMLType)	OPAQUE	OPAQUE 型
oracle.sql.ORAData を実装したカスタム Java クラス (任意の oracle.sql 型をラッピングするクラス)	任意	任意
SQLJ オブジェクト Java 型 (SQLData または ORAData を実装可能)	JAVA_STRUCT	SQLJ オブジェクト SQL 型 (背後で使用される JAVA_STRUCT。適切な Java クラスに自動変換)
JAVA 型 (PL/SQL 型用)		
スカラーの索引付き表 (JDBC OCI ドライバのみ) で、Java 数値配列または String、oracle.sql.CHAR または oracle.sql.NUMBER の配列表現	該当なし	該当なし 注意 : PLSQL_INDEX_TABLE 型がありますが、外部的には使用されません。
グローバル化セッション・サポート		
oracle.sql.NCHAR	CHAR	CHAR
oracle.sql.NString	CHAR VARCHAR LONGVARCHAR	CHAR VARCHAR2 LONG
oracle.sql.NCLOB	CLOB	CLOB
oracle.sqlj.runtime.NcharCharacterStream	LONGVARCHAR	LONG
oracle.sqlj.runtime.NcharAsciiStream (現在は使用されていない。 NcharCharacterStream で使用)	LONGVARCHAR	LONG
oracle.sqlj.runtime.NcharUnicodeStream (現在は使用されていない。 NcharCharacterStream で使用)	LONGVARCHAR	LONG
問合せ結果オブジェクト		
java.sql.ResultSet	CURSOR	CURSOR
SQLJ イテレータ・オブジェクト	CURSOR	CURSOR

Oracle 型のサポートの詳細は、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。

標準機能での型のサポートについては、次に要点を示します。

- Java char 型と Character 型は、JDBC および SQLJ ではサポートされていません。文字データを表現するときは、かわりに Java の String 型を使用してください。
- サポートされている java.sql.Date 型と、直接にはサポートされていない java.util.Date 型とを混同しないようにしてください。java.sql.Date クラス (java.util.Date のラッパー) を使用すると、JDBC の日付値に関して、SQL の DATE データを識別したり、JDBC エスケープ構文をサポートする書式設定操作および解析操作を加えたりできるようになります。
- Oracle9i のすべての数値型は、NUMBER として格納されます。表作成時に NUMBER を宣言すると、(総桁数と小数点以下の桁数を宣言することによって) さらに詳細に精度を指定できます。ただし、Oracle JDBC ドライバを介してデータを取得する場合、データの受取り側の Java 型によっては、この精度が損なわれることがあります。完全な情報は、oracle.sql.NUMBER インスタンスに保持されます。
- Java のラッパー・クラス (Integer や Float など) を使用すると、SQL 文から NULL 値が戻された場合に役立ちます。基本型 (int や float など) には、NULL 値を格納できないためです。詳細は、4-18 ページの「[NULL の処理](#)」を参照してください。
- 結果セットおよびイテレータのホスト変数に対する SQLJ のサポートの詳細は、3-47 ページの「[ホスト変数としてのイテレータおよび結果セットの使用](#)」を参照してください。
- ストリームをホスト変数として使用するときには、SQLJ ストリーム・クラスが必要です。詳細は、5-13 ページの「[ストリームのサポート](#)」を参照してください。
- 弱い型指定は、OUT パラメータまたは INOUT パラメータには使用できません。対応する Oracle 拡張型と同様に、これは Struct、Ref および Array の標準的な JDBC 2.0 型に適用されます。
- 一連の新しいインタフェースは、oracle.jdbc パッケージにあり、oracle.jdbc.driver パッケージに代わり、Oracle9i JDBC に追加されました。これらの新しいインタフェースには、ユーザーが Oracle JDBC ドライバによって Oracle 固有機能を利用するためのより汎用的な方法が用意されています。特に、中間層のプログラムを作成する場合は、新しい API を使用してください。Oracle8i の API も引き続きサポートされるので、Oracle8i から Oracle9i へのアップグレード時に、既存の JDBC コードを変更する必要はありません。(ただし、SQLJ プログラムは通常これらのインタフェースを直接使用しません。これらのインタフェースは、SQLJ ランタイムまたは Oracle 固有生成コードで透過的に使用されます。)

詳細は、6-6 ページの「[カスタム Java クラスのインタフェース指定](#)」を参照してください。

次に、Oracle の拡張型について要点を示します。詳細は、5-27 ページの「[JDBC 2.0 LOB 型と Oracle 拡張型のサポート](#)」および第 6 章「オブジェクト、コレクションおよび OPAQUE 型」を参照してください。

- Oracle SQLJ で `public static _SQL_TYPECODE` パラメータを設定する場合には、`oracle.sql.ORAData` を実装したクラスが必要になります。ただし、どのクラスが必要になるかは、`OracleTypes` クラスに定義されている値に応じて異なります。場合によっては、オブジェクト用の `_SQL_NAME` やオブジェクト参照用の `_SQL_BASETYPE` など、`_SQL_TYPECODE` 以外のパラメータを設定する必要があることもあります。Oracle JPublisher ユーティリティを使用すれば、必要なクラスを自動的に生成できます。

6-10 ページの「[ORAData を実装したクラスに対する Oracle の要件](#)」を参照してください。

- `oracle.sql` クラスは SQL データのラッパーです。Oracle の各データ型に対応しています。ARRAY、STRUCT、REF、BLOB および CLOB クラスは、標準 JDBC 2.0 インタフェースに対応します。これらのクラスと Oracle 拡張型の詳細は、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。
- カスタム Java クラスでマッピングの対象となるのは、Oracle オブジェクト（ORAData または SQLData を実装）、参照（ORAData のみを実装）、コレクション（ORAData のみを実装）、OPAQUE 型（ORAData のみを実装）またはその他のカスタマイズ処理に使用する SQL 型（ORAData のみを実装）です。6-5 ページの「[カスタム Java クラス](#)」を参照してください。

Oracle JPublisher ユーティリティを使用すれば、カスタム Java クラスを自動的に生成できます。6-26 ページの「[JPublisher とカスタム Java クラスの作成](#)」を参照してください。

- Oracle SQLJ では、文字列を WHERE 句の CHAR 列値と比較するときに、自動的に空白埋めを考慮する機能があります。この機能がない場合は、データベース列の文字数と一致させるために文字列に対する埋込みが必要です。この機能は、Oracle 固有コード生成では SQLJ トランスレータ・オプション、ISO 標準コード生成では Oracle カスタマイザ・オプションとして使用できます。8-56 ページの「[空白埋めを考慮した CHAR 比較 \(-fixedchar\)](#)」および A-30 ページの「[Oracle カスタマイザによる空白埋めを考慮した CHAR 比較 \(fixedchar\)](#)」を参照してください。
- 弱い型指定は、OUT パラメータまたは INOUT パラメータには使用できません。Oracle OPAQUE 型と同様に、これは STRUCT、REF および ARRAY の Oracle 拡張型および対応する標準的な JDBC 2.0 型に適用されます。
- 次に、Oracle 拡張機能の使用に必要な項目を示します。
 - Oracle JDBC ドライバ
 - 変換時の Oracle 固有コード生成または Oracle のカスタマイズ
 - アプリケーション実行時の Oracle SQLJ ランタイム

JDBC 2.0 でサポートされている型と要件

前述の表 5-1 に示したとおり、Oracle JDBC および SQLJ で標準 `java.sql` パッケージの JDBC 2.0 型がサポートされています。

ここでは、JDBC 2.0 でサポートされている型および関連する Oracle の拡張機能について説明します。

重要： Sun 社の JDK 環境で JDBC 2.0 型を使用するには、JDK バージョン 1.2.x 以上が必要です。Oracle JDBC を JDK 1.1.x 環境で使用すると、JDBC 2.0 型の機能と同等の `oracle.jdbc2` 拡張機能がサポートされますが、Oracle SQLJ ではどのような場合にも `oracle.jdbc2` パッケージがサポートされません。

Oracle SQLJ で JDBC 2.0 型または Oracle 拡張型を使用するには、JDK 1.2.x 以上をサポートする SQLJ `runtime12` または `runtime12ee` ライブラリを使用します。

Oracle SQLJ でサポートされている JDBC 2.0 型を、表 5-2 に示します。この JDBC 2.0 型とそれに対応する Oracle 拡張型については、この表を参照してください。

Oracle 拡張型は、以前のリリースから提供されていたもので、引き続き現行のリリースでも利用できます。この `oracle.sql.*` クラスには、SQL ロー・データをラッピングする機能が用意されています。『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。

表 5-2 Oracle 拡張型と JDBC 2.0 型との相関関係

JDBC 2.0 型	Oracle 拡張型
<code>java.sql.Blob</code>	<code>oracle.sql.BLOB</code>
<code>java.sql.Clob</code>	<code>oracle.sql.CLOB</code>
<code>java.sql.Struct</code>	<code>oracle.sql.STRUCT</code>
<code>java.sql.Ref</code>	<code>oracle.sql.REF</code>
<code>java.sql.Array</code>	<code>oracle.sql.ARRAY</code>
<code>java.sql.SQLData</code>	該当なし
該当なし	<code>oracle.sql.ORAData</code> (<code>_SQL_TYPECODE = OracleTypes.STRUCT</code> の場合)

ORADData 機能は、ユーザー定義型の Java サポートの標準 SQLData 機能に対する Oracle 固有の代替機能です。詳細は、6-5 ページの「[カスタム Java クラス](#)」を参照してください。

表 5-2 に示した他の型のサポートの詳細は、5-29 ページの「[BLOB、CLOB および BFILE のサポート](#)」および 6-74 ページの「[弱い型指定のオブジェクト、参照およびコレクションのサポート](#)」を参照してください。

次に示した JDBC 2.0 型については、現行の Oracle JDBC や SQLJ で未対応となっています。

- JAVA_OBJECT: SQL 列の Java 型のインスタンスを表す型。
- DISTINCT: 基本 SQL 型と区別して表現や取出しが可能な SQL 型（たとえば、SHOESIZE -> NUMBER など）。

PL/SQL の BOOLEAN 型、RECORD 型および TABLE 型の使用

Oracle SQLJ および JDBC では、コール用引数または戻り値として、PL/SQL の BOOLEAN 型または RECORD 型をサポートしていません。Thin ドライバを使用している場合は、コール用引数または戻り値として、PL/SQL の TABLE 型（索引付き表とも呼ばれます）もサポートされません。ただし、OCI ドライバを使用している場合は TABLE 型がサポートされます。

TABLE 型のサポート（OCI ドライバを使用している場合のみ）

Oracle JDBC OCI ドライバは、Oracle8i リリース 8.1.7 から PL/SQL のスカラー索引付き表をサポートしています。JDBC サポートの詳細は、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。

Oracle SQLJ は、スカラー索引付き表とのデータのやりとりを単純化します。次の配列型がサポートされます。

- 数値型: `int []`、`long []`、`float []`、`double []`、`short []`、`java.math.BigDecimal []`、`oracle.sql.NUMBER []`
- 文字型: `java.lang.String []`、`oracle.sql.CHAR []`

次に、索引付き表のデータをデータベースに書き込む例を示します。

```
int[] vals = {1,2,3};
#sql { call procin(:vals) };
```

次に、索引付き表のデータをデータベースから取り出す例を示します。

```
oracle.sql.CHAR[] outvals;
#sql { call procout(:OUT outvals/*[111](22)*/) };
```

このように `/*...*/` 内の `[xxx]` 構文を使用して、取得する出力配列の最大長を指定する必要があります。また、文字などのバインドの場合は、オプションで `(xx)` 構文を指定し、例のように配列要素の最大長（バイト単位）を指定できます。

注意： `oracle.sql.Datum` クラスは、直接サポートされません。
`oracle.sql.CHAR` または `oracle.sql.NUMBER` などの適切なサブクラスを使用する必要があります。

サポートされない型に対する次善策

サポートされない型の問題を回避するには、ラッパー・プロシージャを作成し、サポートされている型を使用してデータを処理します。たとえば、PL/SQL のブール値を使用するストアド・プロシージャをラッピングするには、JDBC から文字または数字をとり、これを元のプロシージャに `BOOLEAN` として渡すストアド・プロシージャを作成します。出力パラメータの場合は、元のプロシージャから `BOOLEAN` 引数を受け取り、これを `CHAR` または `NUMBER` として JDBC に渡すようにします。同様に、PL/SQL の `RECORD` 型を使用するストアド・プロシージャをラッピングするには、レコードの各コンポーネント（`CHAR` や `NUMBER` など）を処理するストアド・プロシージャを作成します。PL/SQL の `TABLE` 型を使用するストアド・プロシージャをラッピングするには、データを複数のコンポーネントに分解するか、または Oracle のコレクション型を使用します。

次に、PL/SQL のラッパー・プロシージャ `MY_PROC` の例を示します。このプロシージャは、入力として `BOOLEAN` をとるストアド・プロシージャ `PROC` をラッピングします。

```
PROCEDURE MY_PROC (n NUMBER) IS
BEGIN
    IF n=0
    THEN proc(false);
    ELSE proc(true);
    END IF;
END;

PROCEDURE PROC (b BOOLEAN) IS
BEGIN
    ...
END;
```

注意： PL/SQL パッケージまたは SQL オブジェクトで、メソッドとしてサポートされない PL/SQL 型のシグネチャを使用する場合は、Oracle9i JPublisher ユーティリティを使用することを考慮してください。このユーティリティによって、そのようなメソッドをコールする Java 型の作成が容易になります。JPublisher の概要は、6-26 ページの「[JPublisher とカスタム Java クラスの作成](#)」、JPublisher の詳細は、『Oracle9i JPublisher ユーザーズ・ガイド』を参照してください。

以前の Oracle JDBC リリースとの下位互換性

ここでは、以前の Oracle JDBC リリースで Oracle SQLJ を使用する場合の下位互換性について説明します。

注意： Oracle9i リリース 2 (9.2) では、OPAQUE 型および TIMESTAMP 型のサポートが追加されました。

Oracle8i との下位互換性

次に示す Oracle9i の機能は、Oracle8i の JDBC ドライバではサポートされていないか、異なる形でサポートされています。

- ユーザー定義 SQL 型の Java マッピング用の `oracle.sql.ORAData` および `ORADataFactory` インタフェース

Oracle8i `oracle.sql.CustomDatum` および `CustomDatumFactory` インタフェースを使用します。6-8 ページの「[ORAData と CustomDatum インタフェースの比較](#)」を参照してください。
- グローバリゼーション・サポートのためのキャラクタ・タイプへの Oracle 拡張型：
`NCHAR`、`NCLOB`、`NString` および `NcharCharacterStream`（または以前のリリースでの `NcharAsciiStream` および `NcharUnicodeStream`）

Oracle 8.0.x および 7.3.x との下位互換性

Oracle9i JDBC ドライバでサポートされている Oracle 拡張型の一部は、Oracle 8.0.x と Oracle 7.3.x の JDBC ドライバのどちらを使用するかによってサポートの可否が決まります。特に、次の点に注意してください。

- Oracle 8.0.x および 7.3.x ドライバには、`oracle.sql` パッケージがありません。つまり、SQL ロー・データをラッピングするためのラッパー型（`oracle.sql.NUMBER` や `oracle.sql.CHAR` など）がありません。
- Oracle 8.0.x および 7.3.x ドライバでは、Oracle のオブジェクト型とコレクション型については未対応となっています。
- Oracle 8.0.x および 7.3.x のドライバで、Oracle の ROWID データ型をサポートするには、`oracle.jdbc` パッケージの `OracleRowid` クラスを使用する必要があります。
- Oracle 8.0.x ドライバでは、Oracle BLOB、CLOB および BFILE データ型を `oracle.jdbc` パッケージ内の `OracleBlob`、`OracleClob` および `OracleBfile` クラスでサポートしています。これらのクラスには、5-29 ページの「[BLOB、CLOB および BFILE のサポート](#)」で述べられているような LOB および BFILE 操作メソッドがありません。そのため、このパッケージのかわりに、PL/SQL の DBMS_LOB パッケージを使用する必要があります（同項を参照）。
- Oracle 7.3.x ドライバは、BLOB、CLOB および BFILE については未対応です。

表 5-3 に、前述の相違点を要約します。

表 5-3 Oracle 8.0.x および 7.3.x の JDBC ドライバの型サポートの相違点

Java 型 (Oracle 拡張型)	Oracle 型定義	Oracle データ型
oracle.sql.NUMBER	未対応	該当なし
oracle.sql.CHAR	未対応	該当なし
oracle.sql.RAW	未対応	該当なし
oracle.sql.DATE	未対応	該当なし
oracle.jdbc.OracleRowid	ROWID	ROWID
oracle.jdbc.OracleBlob	8.0.x では BLOB	8.0.x では BLOB
	7.3.x では未対応	7.3.x では該当なし
oracle.jdbc.OracleClob	8.0.x では CLOB	8.0.x では CLOB
	7.3.x では未対応	7.3.x では該当なし
oracle.jdbc.OracleBfile	8.0.x では BFILE	8.0.x では BFILE
	7.3.x では未対応	7.3.x では該当なし
oracle.sql.STRUCT	未対応	該当なし
oracle.sql.REF	未対応	該当なし
oracle.sql.ARRAY	未対応	該当なし
JPub 生成のオブジェクト	未対応	該当なし
JPub 生成のオブジェクト参照	未対応	該当なし
JPub 生成の配列	未対応	該当なし
クライアント・カスタマイズ型 (オブジェクト、参照、コレクションな どの oracle.sql 型のカスタマイズ)	未対応	該当なし

ストリームのサポート

標準 SQLJ には、ロング・データをストリームとして処理する次の 2 つの専用クラスがあります。イテレータ列でデータベースからデータを取り出すとき、または入力ホスト変数を使用してデータをデータベースに送信するときに、これらのストリーム型を使用します。一般の Java ストリームと同じように、これらのクラスでは、大きなデータ項目を扱いやすい大きさの塊に分割して処理し、転送できます。

- `sqlj.runtime.BinaryStream`
- `sqlj.runtime.CharacterStream`

ここでは、これらのクラス、Oracle 固有の SQLJ 拡張機能およびストリーム・クラスのメソッドの一般的な使用方法について説明します。

注意： JDBC 2.0 では、`AsciiStream` クラスおよび `UnicodeStream` クラスは、`CharacterStream` クラスに置換されています。`CharacterStream` によって、ユーザーはエンコーディングに関する不要なロジスティクスから解放されます。`AsciiStream` クラスおよび `UnicodeStream` クラスは、下位互換性の目的でサポートされていますが、現在は使用されません。

SQLJ ストリームの一般的な使用方法

Oracle9i の場合は、通常、5-2 ページの表 5-1 のデータ型をこれらのストリーム・クラスで処理します。つまり、

- `BinaryStream` は通常、LONG RAW データ型 (`Types.LONGVARBINARY`) で使用されますが、RAW データ型 (`Types.BINARY` または `Types.VARBINARY`) でも使用されます。
- `CharacterStream` は通常、LONG データ型 (`java.sql.Types.LONGVARCHAR`) で使用されますが、VARCHAR2 データ型 (`Types.VARCHAR`) でも使用されます。

ストリームの使用方法は自由です。表 5-1 に示したように、LONG データと VARCHAR2 データは、Java 文字列でも使用できます。一方、RAW データと LONGRAW データは、Java バイト配列でも使用できます。また、データベースが BLOB (binary large object) や CLOB (character large object) などのラージ・オブジェクト型をサポートする場合は、LONG や LONG RAW などの型より BLOB や CLOB の方が便利な場合もあります (ラージ・オブジェクトからのデータの抽出には、ストリームも使用できます)。Oracle SQLJ および JDBC では、ラージ・オブジェクト型をサポートします。詳細は、5-29 ページの「BLOB、CLOB および BFILE のサポート」を参照してください。

SQLJ ストリーム・クラスは両方ともに標準 Java クラスのサブクラス（`BinaryStream` の `java.io.InputStream` および `CharacterStream` の `java.io.Reader`）で、SQLJ に必要な機能を提供するラッパーとして動作します。つまり、基になるデータが適切に処理および変換されるように、データのデータ型とデータ長を SQLJ に通知します。

ホスト変数に SQLJ ストリーム型を使用して、データの送信または取出しが実行できます。

ストリームをサポートするクラスの主な特長

次の略記されたコードは、`BinaryStream` クラスの主な特長（拡張される内容、コンストラクタのシグネチャ、および主要メソッドのシグネチャなど）を示しています。

```
public class sqlj.runtime.BinaryStream extends sqlj.runtime.StreamWrapper
{
    public sqlj.runtime.BinaryStream(java.io.InputStream);
    public sqlj.runtime.BinaryStream(java.io.InputStream,int);
    public java.io.InputStream getInputStream();
    public int getLength();
    public void setLength(int);
}
```

次の略記されたコードは、`CharacterStream` クラスの主な特長を示しています。

```
public class sqlj.runtime.CharacterStream extends java.io.FilterReader
{
    public sqlj.runtime.CharacterStream(java.io.Reader);
    public sqlj.runtime.CharacterStream(java.io.Reader,int);
    public int getLength();
    public java.io.Reader getReader();
    public void setLength(int);
}
```

コンストラクタの `int` パラメータは、データ長をバイト数または文字数の適切な値で指定します。

注意：

- `java.io.InputStream` オブジェクトを入力に使用するメソッドの場合は、かわりに `BinaryStream` オブジェクトを使用できます。同様に、`java.io.Reader` オブジェクトを入力に使用するメソッドの場合は、かわりに `CharacterStream` オブジェクトを使用できます。
 - 現在使用されていない `AsciiStream` クラスおよび `UnicodeStream` クラスにも、`BinaryStream` と同じ主な特長とシグネチャがあります。
-
-

データ送信時の SQLJ ストリームの使用

標準 SQLJ では、ストリームをホスト変数として使用して、データベースを更新できます。

SQLJ ストリームをデータベースに送信するときは、データの長さを特定し、この長さを SQLJ ストリームのコンストラクタに対して指定する必要があります。この方法については、後述します。

次の手順で、SQLJ ストリームを使用して、データをデータベースに送信します。

1. データの長さを特定します。
2. 入力として適切な標準 Java データ・オブジェクトを作成します。BinaryStream の場合は、入力ストリーム (java.io.InputStream またはいくつかのサブクラスのインスタンス) を作成します。CharacterStream の場合は、Reader オブジェクト (java.io.Reader またはいくつかのサブクラスのインスタンス) を作成します。
3. データ型に応じた SQLJ ストリーム・クラスのインスタンスを作成するには、データ・オブジェクトと長さ (int 値) をコンストラクタに渡します。
4. この SQLJ のストリーム・インスタンスは、SQLJ 実行文の SQL 操作のホスト変数として使用します。
5. ストリームを終了します。(終了しなくてもかまいませんが、終了することをお勧めします。)

次に、SQLJ ストリームをデータベースに送信する方法を、2 つの一般的な例を示して詳しく説明します。

- オペレーティング・システム・ファイルを使用した、LONG 列または LONG RAW 列の更新

LONG RAW 列の更新にはバイナリ・ファイルを使用し、LONG 列の更新にはテキスト・ファイルを使用します。
- バイト配列を使用した、LONG RAW 列の更新

ファイルからの LONG または LONG RAW の更新

ここでは、File オブジェクトから CharacterStream オブジェクトまたは BinaryStream オブジェクトを作成し、そのオブジェクトをデータベースの更新に使用する方法を示します。この項の最後にあるコード例では、LONG 列に CharacterStream を使用しています。

データベース列 (LONG または LONG RAW 列を想定) をファイルから更新するときは、長さを特定する必要があります。この作業を行うには、入力ストリームを作成する前に、java.io.File オブジェクトを作成します。

次の手順で、ファイルからデータベースを更新します。

1. ファイルから `java.io.File` オブジェクトを作成します。このためには、`File` クラスのコンストラクタへのファイル・パス名を指定してください。
2. `File` オブジェクトの `length()` メソッドで、データの長さを特定します。このメソッドから戻される値は `long` 値なので、この値を `SQLJ` ストリーム・クラスのコンストラクタに入力するには、`int` 値にキャストする必要があります。

注意： このキャストを行う前に、`long` 値が `int` 変数に収まることを確認してください。クラス `java.lang.Integer` の `static` 定数 `MAX_VALUE` は、Java の許容最大 `int` 値です。

3. 文字データの場合は、`File` オブジェクトから `java.io.FileReader` オブジェクトを作成します。このためには、`File` オブジェクトを `FileReader` コンストラクタに渡してください。

バイナリ・データの場合は、`File` オブジェクトから `java.io.FileInputStream` オブジェクトを作成します。このためには、`File` オブジェクトを `FileInputStream` コンストラクタに渡してください。
4. 該当する `SQLJ` ストリーム・オブジェクトを作成します。テキスト・ファイルの場合は `CharacterStream` オブジェクト、バイナリ・ファイルの場合は `BinaryStream` オブジェクトを作成します。`FileReader` または `FileInputStream` の該当するオブジェクトとデータ長 (`int` 値) を `SQLJ` のストリーム・クラスのコンストラクタに渡します。
5. `SQLJ` のストリーム・オブジェクトは、`SQLJ` 実行文中で `SQL` 操作のホスト変数として使用します

次に、ファイルからデータベースに `LONG` データを書き込む例を示します。ここでは、`/private/mydir/myfile.html` にある `HTML` ファイルの内容をデータベースの表 `filetable` の `LONG` 列 `chardata` に挿入します。

インポートは、次のように指定します。

```
import java.io.*;
import sqlj.runtime.*;
```

実行可能コードを示します。

```
File myfile = new File ("/private/mydir/myfile.html");
int length = (int)myfile.length();      // Must cast long output to int.
FileReader filereader = new FileReader(myfile);
CharacterStream charstream = new CharacterStream(filereader, length);
#sql { INSERT INTO filetable (chardata) VALUES (:charstream) };
charstream.close();
...
```


バイト配列からの LONG RAW の更新

ここでは、バイト配列から `BinaryStream` オブジェクトを作成し、そのオブジェクトをデータベースの更新に使用する方法を示します。この項の最後にあるコード例も参照してください。

バイト配列からデータベースを更新する場合は、あらかじめデータの長さを特定する必要があります。（ここでは、LONG RAW 列を更新します。）配列の場合は、ファイルの場合より簡単です。Java ではどの配列も、データ長を戻す機能を備えているためです。

次の手順で、バイト配列からデータベースを更新します。

1. 配列の `length` 機能を使用して、データの長さを特定します。この機能により、`int` 値が戻されます。SQLJ ストリーム・クラスのすべてのコンストラクタがこの値を必要とします。
2. 配列から `java.io.ByteArrayInputStream` オブジェクトを作成します。このためには、バイト配列を `ByteArrayInputStream` コンストラクタに渡してください。
3. `BinaryStream` オブジェクトを作成します。`ByteArrayInputStream` オブジェクトとデータ長（`int` 値）を `BinaryStream` クラスのコンストラクタに渡してください。

コンストラクタのシグネチャは、次のように指定します。

```
BinaryStream (InputStream in, int length)
```

このシグネチャには、`java.io.InputStream` クラスまたは任意のサブクラス（`ByteArrayInputStream` など）のインスタンスを指定してください。

4. SQLJ のストリーム・オブジェクトは、SQLJ 実行文中で SQL 操作作用のホスト変数として使用します

次に、バイト配列から LONG RAW データをデータベースに書き込む例を示します。ここでは、バイト配列 `bytearray[]` の内容をデータベース表 BINTABLE の中にある LONG RAW 型の列 BINDATA に書き込みます。

インポートは、次のように指定します。

```
import java.io.*;
import sqlj.runtime.*;
```

実行可能コードを示します。

```
byte[] bytearray = new byte[100];

(Populate bytearray somehow.)
...
int length = bytearray.length;
ByteArrayInputStream arraystream = new ByteArrayInputStream(bytearray);
BinaryStream binstream = new BinaryStream(arraystream, length);
#sql { INSERT INTO bintable (bindata) VALUES (:binstream) };
binstream.close();
...
```

注意： この例に示したように、ストリームは必ずしも必要ではありません。バイト配列から直接データベースを更新することも可能です。

ストリームへのデータの取出し：注意

SQLJ ストリーム・クラスでも、データを取り出せます。ただし、ストリームの使用時は、一部のデータベース製品で注意が必要です。

ストリームでのロング・データの読取り / 書込みに **Oracle9i** と **Oracle JDBC** ドライバを使用する場合は、ストリーム・データへのアクセス方法およびデータの処理方法に注意する必要があります。

Oracle JDBC ドライバはイテレータ行からデータにアクセスするため、通信パイプからストリーム項目をフラッシュした後で次のデータ項目にアクセスするようになっています。イテレータ行の処理時にストリーム・データがローカル・ストリームに書き込まれる場合でも、**JDBC** ドライバが次のデータ項目にアクセスする前にローカル・ストリームからデータを読み取らないと、ストリーム・データが失われます。このようなストリーム処理は、ストリームの特性である大型化と長さが不明な点を考慮したものです。

したがって、**Oracle JDBC** ドライバでストリーム項目にアクセスし、ローカル・ストリーム変数への書込みが完了した時点で、このローカル・ストリームの読取りと処理を行い、その後で、イテレータから別のアクセスを行う必要があります。

特に、位置指定イテレータの場合は必須の **FETCH INTO** 構文があるため、注意が必要です。各回のフェッチでは、すべての列の読取りが完了した後で、処理が開始されます。したがって、ストリーム項目は1つのみ、つまり最後にアクセスした項目のみとなります。

次に、注意点をまとめます。

- 位置指定イテレータを使用するときは、一度に1つのストリーム列、つまり最後の列のみを操作できます。イテレータの各行をフェッチし、ストリーム項目をプロセスのローカル入力ストリーム変数に書き込むたびに、ただちにローカル・ストリーム変数を読み込み、処理する必要があります。その後、イテレータの次の行に進みます。
- 名前指定イテレータの使用時は、複数のストリーム列を操作できます。ただし、各イテレータ行を処理するときに、ストリーム・フィールドにアクセスし、データをプロセスのローカル・ストリーム変数に書き込むたびに、ただちにローカル・ストリームを読み込み、処理する必要があります。その後、イテレータの他の部分を読み込みます。

また、名前指定イテレータの各行を処理するときは、イテレータ設定時の問合せでデータベース列を選択したときと同じ順番で、列のアクセッサ・メソッドをコールする必要があります。前述のように、問合せ後の通信パイプにストリーム・データが残っているためです。列へのアクセス順が異なると、ストリーム・データがスキップされ、他の列へのアクセス時に失われることがあります。

注意： Oracle9i および Oracle JDBC ドライバの場合は、`SELECT INTO` 文でストリームを使用できません。

データ取得時の SQLJ ストリームの使用

データをストリームとして取り出すとき、標準 SQLJ では、名前指定または位置指定イテレータの SQLJ ストリーム型列へのデータの読み込みができます。

ここでは、位置指定イテレータまたは名前指定イテレータを使用してデータを SQLJ ストリームに取り出す基本的な手順を示します。5-18 ページの「[ストリームへのデータの取出し](#) : 注意」の項で述べた注意事項に留意してください。

ここでは、手順を簡単に示します。詳細は、5-21 ページの「[ストリーム・クラスのメソッド](#)」および 5-23 ページの「[ストリーム・データの取出しおよび処理例](#)」を参照してください。

位置指定イテレータの SQLJ ストリーム列の使用法

次の手順で、位置指定イテレータを使用して、データを SQLJ ストリームに取り出します。

1. 位置指定イテレータ・クラスを宣言します。最終列を該当する SQLJ ストリーム型として指定します。
2. イテレータ型のローカル変数を宣言します。
3. 該当する SQLJ ストリーム型のローカル変数を宣言します。これをホスト変数として、イテレータの SQLJ ストリーム列の各行からデータを取得します。
4. 問合せを実行して、手順 2 で宣言したイテレータにデータを設定します。

5. 通常どおり、イテレータを処理します。(3-43 ページの「[位置指定イテレータの使用](#)」を参照してください。) `FETCH INTO` 文の `INTO` リスト内のホスト変数は、位置指定イテレータの列順に記述する必要があるので、ローカル入力ストリーム変数をリストの最後のホスト変数にします。
6. イテレータの処理ループでは、各イテレータ行にアクセスした時点でローカル入力ストリームの読取りと処理を行い、必要に応じてストリーム・データの格納や出力を行います。
7. イテレータ処理ループが一巡するたびに、ローカル入力ストリームを終了します。(終了しなくてもかまいませんが、終了することをお勧めします。)
8. イテレータを終了します。

名前指定イテレータの SQLJ ストリーム列の使用法

次の手順で、名前指定イテレータを使用して、データを 1 つ以上の SQLJ ストリームに取り出します。

1. 該当する SQLJ ストリーム型の列を 1 つ以上指定して、名前指定イテレータ・クラスを宣言します。
2. イテレータ型のローカル変数を宣言します。
3. イテレータの各 SQLJ ストリーム列に対して、入力ストリーム型またはリーダー型のローカル変数を宣言します。ストリーム列のアクセッサ・メソッドからデータを受け取るときに、この変数を使用します。これらのローカル・ストリーム変数は、SQLJ のストリーム型にする必要はありません。標準の `java.io.InputStream` または `java.io.Reader` の該当する型でもかまいません。イテレータ列を SQLJ のストリーム型にすると、データの型も正しく変換されます。
4. 問合せを実行して、手順 2 で宣言したイテレータにデータを設定します。
5. 通常どおり、イテレータを処理します。(3-38 ページの「[名前指定イテレータの使用法](#)」を参照してください。) イテレータの各行の処理では、各ストリーム列のアクセッサ・メソッドがストリーム・データを戻すたびに、手順 3 で宣言したローカル入力ストリーム変数にデータが書き込まれます。

ストリーム・データの欠損を防ぐには、手順 4 の問合せで列を選択したときと同じ順番で、列のアクセッサ・メソッドをコールする必要があります。

6. イテレータの処理ループで、ストリーム列のアクセッサ・メソッドをコールし、データをローカル入力ストリーム変数に書き込んでから、ただちにローカル入力ストリームを読み取って処理し、ストリーム・データを格納または出力します。
7. イテレータ処理ループが一巡するたびに、ローカル入力ストリームを終了します。(終了しなくてもかまいませんが、終了することをお勧めします。)
8. イテレータを終了します。

注意： SQLJ ストリーム・オブジェクトにデータを移入した場合は、ストリームのデータ長属性が有効になりません。データ長を明示的に設定すると、この属性が有効になります。つまり、各 SQLJ ストリーム・クラスの `setLength()` メソッドを使用するか、またはコンストラクタにデータ長を指定すると、有効になります（詳細は、5-15 ページの「[データ送信時の SQLJ ストリームの使用](#)」を参照してください）。

ストリーム・クラスのメソッド

名前指定または位置指定イテレータの SQLJ ストリーム列を処理するときは、ストリーム・データを受け取るローカル・ストリーム変数を SQLJ のストリーム型または標準の `java.io.InputStream` または `java.io.Reader` のいずれか該当する型にできます。いずれの場合も、入力データ・オブジェクトの標準のメソッドを使用できます。

ローカル・ストリーム変数を SQLJ のストリーム型、つまり `BinaryStream` または `CharacterStream` にすると、データを SQLJ のストリーム・オブジェクトから直接読み取れることも、基になる `InputStream` オブジェクトまたは `Reader` オブジェクトを取得し、このオブジェクトからデータを読み取ることも可能です。どちらの方法をとるかは、好みの問題ですが、最初の方法の方が簡単です。2 番目の方法は、より直接的にデータ・アクセスできるので効率的です。

バイナリ・ストリームのメソッド

`BinaryStream` クラスは、`sqlj.runtime.StreamWrapper` クラスのサブクラスです。`StreamWrapper` クラスには、次の主なメソッドがあります。

- `InputStream getInputStream()`: このメソッドを使用して、基になる `java.io.InputStream` オブジェクトを取得することもできます。SQLJ のストリーム・オブジェクトは、このメソッドを使用せずに、直接処理できます。
- `void setLength(int length)`: SQLJ ストリーム・オブジェクトの `length` 属性を設定できます。ストリーム・オブジェクトの作成時に設定した `length` の値をそのまま使用する場合は、このメソッドを使用する必要はありません。

SQLJ ストリームをデータベースに送信する場合は、あらかじめ `length` 属性の値を設定する必要があります。

- `int getLength()`: SQLJ ストリームの `length` 属性の値を戻します。ストリーム・オブジェクトのコンストラクタまたは `setLength()` メソッドで明示的にデータ長を設定すると、この属性値が有効になります。データをストリームに取り出すときは、`length` 属性が自動的に設定されません。

`sqlj.runtime.StreamWrapper` クラスは `java.io.FilterInputStream` クラスのサブクラスです。この `java.io.FilterInputStream` は `java.io.InputStream` クラスのサブクラスです。`InputStream` クラスの主要メソッド (`skip()` メソッド、`close()` メソッドおよび3種類の `read()` メソッド) は、SQLJ の `BinaryStream` クラスでもサポートされています。

- `int read():` 入力ストリームから次のバイト・データを読み取ります。このバイト・データは `int` 値 (0 ~ 255) で戻されます。ストリームの終端に達した場合の戻り値は、-1 です。次のいずれかの状態になるまで、このメソッドはプログラムを実行しません。1) 入力データが用意されたとき。2) ストリームの終端に達したとき。3) 例外が送出されたとき。
- `int read (byte b[]):` 入力ストリームから最大 `b.length` バイトのデータを読み取り、指定された `b[]` バイト配列に書き込みます。戻り値として、読み取ったバイト数を示す `int` 値が戻されます。ストリームの終端に達した場合は、-1 が戻されます。入力が用意されるまで、このメソッドはプログラムを実行しません。
- `int read (byte b[], int off, int len):` 入力ストリームの指定オフセット `off` のバイト位置から最大 `len` (長さ) バイトのデータを読み取り、指定された `b[]` バイト配列に書き込みます。戻り値として、読み取ったバイト数を示す `int` 値が戻されます。ストリームの終端に達した場合は、-1 が戻されます。入力が用意されるまで、このメソッドはブロックします。
- `long skip (long n):` 入力ストリームの `n` バイトのデータをスキップし、破棄します。実際にスキップするバイト数が、指定値より少ない場合もあります。実際にスキップしたバイト数は、`long` 値で戻されます。
- `void close():` ストリームを終了し、関連リソースを解放します。

文字ストリームのメソッド

`CharacterStream` クラスには、次の主なメソッドがあります。

- `Reader getReader():` このメソッドを使用して、基になる `java.io.Reader` オブジェクトを取得することもできます。SQLJ のストリーム・オブジェクトは、このメソッドを使用せずに、直接処理できます。
- `void setLength(int length)`
- `int getLength()`

これらを使用して、ストリーム・オブジェクトの長さを設定または取得します。これは、バイナリ・ストリームの場合と同じ機能です。前述の「[バイナリ・ストリームのメソッド](#)」を参照してください。

`sqlj.runtime.CharacterStream` クラスは `java.io.FilterReader` クラスのサブクラスです。この `java.io.FilterReader` は `java.io.Reader` クラスのサブクラスです。`Reader` クラスの主要メソッド (`skip()` メソッド、`close()` メソッドおよび3種類の `read()` メソッド) は、SQLJ の `CharacterStream` クラスでもサポートされています。

- `int read()`: リーダーから次の文字データを読み取ります。このデータは `int` 値 (0 ~ 65535) で戻されます。データの終端に達した場合の戻り値は、-1 です。次のいずれかの状態になるまで、このメソッドはプログラムを実行しません。1) 入力データが用意されたとき。2) データの終端に達したとき。3) 例外が送出されたとき。
- `int read (char cbuf [])`: 文字を配列に読み込み、データを指定された `char` 型配列の `cbuf []` に書き込みます。戻り値として、読み取った文字数を示す `int` 値が戻されます。データの終端に達した場合は、-1 が戻されます。入力が用意されるまで、このメソッドはプログラムを実行しません。
- `int read (char cbuf [], int off, int len)`: 入力の指定オフセット `off` の文字位置から最大 `len` (長さ) 文字のデータを読み取り、指定された `char` 型配列の `cbuf []` に書き込みます。戻り値として、読み取った文字数を示す `int` 値が戻されます。データの終端に達した場合は、-1 が戻されます。入力が用意されるまで、このメソッドはブロックします。
- `long skip (long n)`: 入力の `n` 文字のデータをスキップし、破棄します。実際にスキップする文字数が、指定値より少ない場合もあります。実際にスキップした文字数は、`long` 値で戻されます。
- `void close()`: ストリームを終了し、関連リソースを解放します。

ストリーム・データの取出しおよび処理例

ここでは、ストリーム・データを取り出す例を示します。

- 最初の例では、`SELECT` 文で `LONG` 列からデータを選択し、名前指定イテレータの `SQLJ CharacterStream` 列に設定します。
- もう 1 つの例では、`SELECT` 文で `LONG RAW` 列からデータを選択し、位置指定イテレータの `SQLJ BinaryStream` 列に設定します。

例: 名前指定イテレータの `CharacterStream` 列への `LONG` データの格納 この例では、データベースの `LONG` 列からデータを選択し、名前指定イテレータの `SQLJ CharacterStream` 列に格納します。

表 `FILETABLE` の `VARCHAR2` 型の列 `FILENAME` にファイル名が格納されており、`LONG` 型の列 `FILECONTENTS` にファイルの内容が文字形式で格納されているものとします。

インポートは次のように宣言します。

```
import sqlj.runtime.*;
import java.io.*;
...
#sql iterator MyNamedIter (String filename, CharacterStream filecontents);
```

実行可能コードを示します。

```
MyNamedIter namediter = null;
String fname;
CharacterStream charstream;
#sql namediter = { SELECT filename, filecontents FROM filetable };
while (namediter.next()) {
    fname = namediter.filename();
    charstream = namediter.filecontents();
    System.out.println("Contents for file " + fname + ":");
    printStream(charstream);
    charstream.close();
}

namediter.close();
...
public void printStream(Reader in) throws IOException
{
    int character;
    while ((character = in.read()) != -1) {
        System.out.print((char) character);
    }
}
```

すでに述べたように、入力パラメータとして標準 `java.io.Reader` をとるメソッドには、SQLJ 文字ストリームを渡すことが可能です。

例：位置指定イテレータの `BinaryStream` 列への `LONG RAW` の格納 この例では、`LONG RAW` 列からデータを選択し、位置指定イテレータの `SQLJ BinaryStream` 列に移入します。

5-18 ページの「[ストリームへのデータの取出し：注意](#)」で説明したように、位置指定イテレータには1つの入力ストリームのみ可能で、必ず最終列の必要があります。

表 `BINTABLE` には `NUMBER` 型の列 `IDENTIFIER` と `LONG RAW` 型の列 `BINDATA` があり、識別子に該当するバイナリ・データが `BINDATA` 列に格納されているものとします。

インポートは次のように宣言します。

```
import sqlj.runtime.*;
...
#sql iterator MyPosIter (int, BinaryStream);
```

実行可能コードを示します。

```
MyPosIter positer = null;
int id=0;
BinaryStream binstream=null;
#sql positer = { SELECT identifier, bindata FROM bintable };
while (true) {
```



```
#sql { FETCH :positer INTO :id, :binstream };
if (positer.endFetch()) break;

(...process data as desired...)

binstream.close();
}
positer.close();
...
```

出力パラメータおよびファンクションの戻り値としての SQLJ ストリーム・オブジェクト

前述のように、標準 SQLJ では、パッケージ `sqlj.runtime` の `BinaryStream` および `CharacterStream` クラスを使用して、ストリーム・データを取り出し、イテレータ列に格納できます。

また、Oracle SQLJ では、Oracle9i、Oracle JDBC ドライバ、Oracle 固有コード生成または Oracle カスタマイザ、および Oracle SQLJ ランタイムを使用すると、SQLJ のストリーム型を次のように使用できます。

- ストアド・プロシージャまたはストアド・ファンクションのコールからの OUT または INOUT ホスト変数として
- ストアド・ファンクションのコールの戻り値として

ストアド・プロシージャの出力パラメータとしてのストリーム

`BinaryStream` および `CharacterStream` 型は、ストアド・プロシージャまたはストアド・ファンクションの OUT または INOUT パラメータの代入型として使用できます。

次の表定義を想定します。

```
CREATE TABLE streamexample (name VARCHAR2 (256), data LONG);
INSERT INTO streamexample (data, name)
VALUES
('0000000000001111111111112222222222333333333344444444445555555555',
'StreamExample');
```

また、次のストアド・プロシージャ定義を想定します。この定義には、`STREAMEXAMPLE` 表が使用されています。

```
CREATE OR REPLACE PROCEDURE out_longdata
    (dataname VARCHAR2, longdata OUT LONG) IS
BEGIN
    SELECT data INTO longdata FROM streamexample WHERE name = dataname;
END out_longdata;
```

次のサンプル・コードでは、out_longdata ストアド・プロシージャをコールして、ロング・データを読み取ります。

インポートは、次のように指定します。

```
import sqlj.runtime.*;
```

実行可能コードを示します。

```
CharacterStream data;  
#sql { CALL out_longdata('StreamExample', :OUT data) };  
int c;  
while ((c = data.read ()) != -1)  
    System.out.print((char)c);  
System.out.flush();  
data.close();  
...
```

注意： ストリームを必ずしも終了する必要はありませんが、終了することをお勧めします。

ストアド・ファンクションの結果としてのストリーム

BinaryStream 型および CharacterStream 型は、ストアド・ファンクションの戻り値の代入型として使用できます。

前に示したストアド・プロシージャ例と同じ STREAMEXAMPLE 表定義を想定します。

また、次のストアド・ファンクション定義を想定します。この定義には、STREAMEXAMPLE 表が使用されています。

```
CREATE OR REPLACE FUNCTION get_longdata (dataname VARCHAR2) RETURN long  
    IS longdata LONG;  
BEGIN  
    SELECT data INTO longdata FROM streamexample WHERE name = dataname;  
    RETURN longdata;  
END get_longdata;
```

次のサンプル・コードでは、get_longdata ストアド・ファンクションをコールして、ロング・データを読み取ります。

インポートは、次のように指定します。

```
import sqlj.runtime.*;
```

実行可能コードを示します。

```
CharacterStream data;  
#sql data = { VALUES(get_longdata('StreamExample')) };  
int c;  
while ((c = data.read ()) != -1)  
    System.out.print((char)c);  
System.out.flush();  
data.close();  
...
```

注意： ストリームを必ずしも終了する必要はありませんが、終了することをお勧めします。

JDBC 2.0 LOB 型と Oracle 拡張型のサポート

Oracle SQLJ では、次に示したように JDBC 2.0 のデータ型と Oracle 固有のデータ型に対して拡張機能が用意されています。

- JDBC 2.0 LOB データ型 (BLOB および CLOB)
- Oracle BFILE データ型
- Oracle ROWID データ型
- Oracle REF CURSOR データ型
- その他の Oracle9i データ型 (NUMBER や RAW など)

ここであげたデータ型は、後述の `oracle.sql` パッケージ中のクラスでサポートされています。LOB と BFILE の使用方法はほとんど同じなので、一緒に説明します。

Oracle SQLJ では、次の標準 JDBC 型に対するサポートも強化されています。

- `BigDecimal`

ユーザー定義の SQL オブジェクト (弱い型指定と強い型指定)、オブジェクト参照およびコレクション (可変長配列とネストした表) については、JDBC 2.0 機能もサポートされています。これらについては、[第 6 章「オブジェクト、コレクションおよび OPAQUE 型」](#)で説明します。

コード中に Oracle 拡張型を使用する場合は、次の要件が伴うので注意してください。

- Oracle JDBC ドライバを使用する必要があります。
- Oracle 固有コード生成（デフォルトの `-codegen=oracle` 設定）を使用するか、または ISO コード生成（`-codegen=iso`）の場合は必要に応じてプロファイルをカスタマイズします。カスタマイズはデフォルトのカスタマイズ `oracle.sqlj.runtime.util.OraCustomizer` をお勧めします。
- アプリケーション実行時に Oracle SQLJ ランタイムを使用する必要があります。

Oracle SQLJ ランタイムと Oracle JDBC ドライバは、Oracle 拡張型をコード中に使用しない場合であっても、Oracle カスタマイザを使用する場合は常に必要です。

Oracle 固有のセマンティクス・チェックを行うには、適切なチェッカを使用する必要があります。デフォルトのチェッカ `oracle.sqlj.checker.OracleChecker` は、フロントエンドとして機能し、環境に応じたチェッカを起動します。JDBC ドライバを使用すると、Oracle 固有のチェッカが起動されます。

Oracle 固有型は `oracle.sql` パッケージに定義されます。詳細は、後述の「[パッケージ oracle.sql](#)」を参照してください。

パッケージ oracle.sql

`oracle.sql` パッケージは、SQLJ ユーザーと JDBC ユーザーにとって重要です。このパッケージ内のクラスによって、Oracle9i のすべてのデータ型（`oracle.sql.ROWID`、`oracle.sql.CLOB`、`oracle.sql.NUMBER` など）がサポートされます。`oracle.sql` クラスは、SQL ロー・データ用のラッパーであり、該当の Java 形式へのマッピングと変換メソッドを提供します。`oracle.sql.*` オブジェクトは、該当する SQL データのバイナリ表現をバイト配列の形式で保持しています。

各 `oracle.sql.*` データ型クラスは、`oracle.sql.Datum` クラスのサブクラスです。

Oracle 固有のセマンティクス・チェックを行うには、適切なチェッカを使用する必要があります。デフォルトのチェッカ `oracle.sqlj.checker.OracleChecker` は、フロントエンドとして機能し、環境に応じたチェッカを起動します。JDBC ドライバを使用すると、Oracle 固有のチェッカが起動されます。

セマンティクス・チェック用のトランスレータ・オプションの詳細は、8-32 ページの「[接続オプション](#)」および 8-67 ページの「[セマンティクス・チェックとオフライン解析のオプション](#)」を参照してください。

`oracle.sql` クラスの詳細は『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。

BLOB、CLOB および BFILE のサポート

Oracle JDBC および SQLJ では、JDBC 2.0 ラージ・オブジェクト (LOB) ・データ型 (BLOB (バイナリ LOB) および CLOB (キャラクタ LOB) の総称) がサポートされています。このサポート内容は、Oracle 固有の BFILE 型 (データベースの外部に格納されている読取り専用バイナリ・ファイル) とほとんど同じです。これらのデータ型は、次のクラスでサポートされています。

- `oracle.sql.BLOB`
- `oracle.sql.CLOB`
- `oracle.sql.BFILE`

LOB とファイルの詳細およびサポートされているストリーム API の使用方法は、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。

`oracle.sql.BLOB`、`oracle.sql.CLOB` および `oracle.sql.BFILE` クラスは、Oracle 固有の SQLJ アプリケーションで次のように使用できます。

- SQLJ 実行文中 (INTO リストでも使用可) の IN、OUT または INOUT ホスト変数として
- ストアド・ファンクション・コールの戻り値として
- イテレータ宣言での列型として (名前指定および位置指定イテレータ)

LOB の操作には、BLOB および CLOB クラスで定義したメソッドを使用することをお勧めします。PL/SQL パッケージ `DBMS_LOB` で定義したプロシージャやファンクションも使用できます。このパッケージで定義されているプロシージャとファンクションは、SQLJ プログラムからコールできます。

BFILE の操作には、BFILE クラスで定義したメソッドを使用することをお勧めします。`DBMS_LOB` パッケージのファイル処理ルーチンも使用できます。

Java アプリケーションでは、BLOB、CLOB および BFILE クラスのメソッドの方が `DBMS_LOB` パッケージより便利です。実行時間が短縮される場合もあります。

読取りまたは書込み対象のチャンクの型は、操作対象の LOB の種類によります。たとえば、CLOB の内容は文字データです。したがって、データ・チャンクを Java 文字列で保持します。BLOB の内容はバイナリ・データです。したがって、チャンク・データを Java バイト配列で保持します。

注意： `DBMS_LOB` は Oracle9i パッケージです。サーバーへのラウンドトリップが必要になります。BLOB、CLOB および BFILE クラスのメソッドも、サーバーへのラウンドトリップが必要になることがあります。

BFILF で BFILF クラスを使用した場合と DBMS_LOB 機能を使用した場合の比較

次の例では、BFILF で `oracle.sql` のメソッドを使用した場合と `DBMS_LOB` パッケージを使用した場合を比較します。

例：BFILF での `oracle.sql.BFILE` ファイル処理メソッドの使用 この例では、`oracle.sql.BFILE` クラスのファイル処理メソッドで BFILF を操作します。

```
BFILF openFile (BFILF file) throws SQLException
{
    String dirAlias, name;
    dirAlias = file.getDirAlias();
    name = file.getName();
    System.out.println("name: " + dirAlias + "/" + name);

    if (!file.isFileOpen())
    {
        file.openFile();
    }
    return file;
}
```

BFILF の `getDirAlias()` および `getName()` メソッドは、フルパスとファイル名を取得するためのメソッドです。 `openFile()` メソッドは、ファイルをオープンするためのメソッドです。操作する BFILF は、あらかじめオープンしておく必要があります。

例：BFILF での `DBMS_LOB` ファイル処理ルーチンの使用 この例では、`DBMS_LOB` パッケージのファイル処理ルーチンで BFILF を操作します。

```
BFILF openFile(BFILF file) throws SQLException
{
    String dirAlias, name;
    #sql { CALL dbms_lob.filegetname(:file, :out dirAlias, :out name) };
    System.out.println("name: " + dirAlias + "/" + name);

    boolean isOpen;
    #sql isOpen = { VALUES(dbms_lob.fileisopen(:file)) };
    if (!isOpen)
    {
        #sql { CALL dbms_lob.fileopen(:inout file) };
    }
    return file;
}
```

`openFile()` メソッドは、ファイル・オブジェクトの名前を出力し、その後で、そのファイルのオープンしているバージョンを戻すためのメソッドです。BFILE を操作するには、あらかじめ `DBMS_LOB.FILEOPEN` または `BFILE` クラスの該当メソッドをコールし、ファイルをオープンにしておく必要があります。

LOB で BLOB および CLOB クラスを使用した場合と DBMS_LOB 機能を使用した場合の比較

次の例では、BLOB および CLOB で `oracle.sql` のメソッドを使用した場合と `DBMS_LOB` パッケージを使用した場合を比較します。`oracle.sql` メソッドの各使用例の後に、同じ機能を `DBMS_LOB` で実現した例を示します。

例 : CLOB に対する `oracle.sql.CLOB` 読取りメソッドの使用 `oracle.sql.CLOB` クラスのメソッドを使用して、CLOB からデータを読み取ります。

```
void readFromClob(CLOB clob) throws SQLException
{
    long clobLen, readLen;
    String chunk;

    clobLen = clob.length();

    for (long i = 0; i < clobLen; i+= readLen) {
        chunk = clob.getSubString(i, 10);
        readLen = chunk.length();
        System.out.println("read " + readLen + " chars: " + chunk);
    }
}
```

このメソッドには、CLOB から Java 文字列を 10 文字ずつ読み取って戻すループがあります。CLOB 全体を読み取るまで、ループが繰り返されます。

例：CLOB に対する DBMS_LOB 読取りルーチンの使用 DBMS_LOB パッケージのルーチンを使用して CLOB からの読取りを行います。

```
void readFromClob(CLOB clob) throws SQLException
{
    long clobLen, readLen;
    String chunk;

    #sql clobLen = { VALUES(dbms_lob.getlength(:clob)) };

    for (long i = 1; i <= clobLen; i += readLen) {
        readLen = 10;
        #sql { CALL dbms_lob.read(:clob, :inout readLen, :i, :out chunk) };
        System.out.println("read " + readLen + " chars: " + chunk);
    }
}
```

このメソッドは CLOB の内容を 10 文字ずつ読み取ります。チャンクのホスト変数の型は、String です。

例：BLOB に対する oracle.sql.BLOB 書き込みルーチンの使用 oracle.sql.BLOB クラスのメソッドを使用して、BLOB ヘデータを書き込みます。BLOB と指定の長さを入力します。

```
void writeToBlob(BLOB blob, long blobLen) throws SQLException
{
    byte[] chunk = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    long chunkLen = (long)chunk.length;

    for (long i = 0; i < blobLen; i+= chunkLen) {
        if (blobLen < chunkLen) chunkLen = blobLen;
        chunk[0] = (byte) (i+1);
        chunkLen = blob.putBytes(i, chunk);
    }
}
```

指定された BLOB 長に達するまで、このメソッドは BLOB に 10 バイトずつ書き込むループを繰り返します。

例：BLOB に対する DBMS_LOB 書き込みルーチンの使用 DBMS_LOB パッケージのルーチンを使用して、BLOB への書き込みを行います。

```
void writeToBlob(BLOB blob, long blobLen) throws SQLException
{
    byte[] chunk = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    long chunkLen = (long)chunk.length;

    for (long i = 1; i <= blobLen; i += chunkLen) {
        if ((blobLen - i + 1) < chunkLen) chunkLen = blobLen - i + 1;
        chunk[0] = (byte)i;
        #sql { CALL dbms_lob.write(:INOUT blob, :chunkLen, :i, :chunk) };
    }
}
```

このメソッドは、BLOB に 10 バイトずつ書き込みます。チャンクのホスト変数の型は、byte[] です。

LOB および BFILE 型でのストアド・ファンクションの結果

型 BLOB、CLOB および BFILE のホスト変数は、ストアド・ファンクション・コールの結果に代入できます。次に示した例は、CLOB を対象としています。BLOB および BFILE のコードにも同じ機能があります。

最初に、次のファンクション定義を想定します。

```
CREATE OR REPLACE function longer_clob (c1 clob, c2 clob) return clob is
    result clob;
BEGIN
    if dbms_lob.getLength(c2) > dbms_lob.getLength(c1) then
        result := c2;
    else
        result := c1;
    end if;
    RETURN result;
END longer_clob;
```

次の例では、前の例で定義したファンクションからの戻り値の代入型として CLOB を使用します。

```
void readFromLongest(CLOB c1, CLOB c2) throws SQLException
{
    CLOB longest;
    #sql longest = { VALUES(longer_clob(:c1, :c2)) };
    readFromClob(longest);
}
```

`readFromLongest()` メソッドは、前に定義した `readFromClob()` メソッドを使用して、渡された CLOB の長い方の内容を出力します。

LOB および BFILE ホスト変数と SELECT INTO ターゲット

型 BLOB、CLOB および BFILE のホスト変数は、SELECT INTO 実行文の INTO リストに記述できます。次に示した例は、BLOB と CLOB を対象としていますが、BFILE のコードにも同じ機能があります

次の表定義を想定します。

```
CREATE TABLE basic_lob_table(x varchar2(30), b blob, c clob);
INSERT INTO basic_lob_table
    VALUES('one', '01010101010101010101010101010101', 'onetwothreefour');
INSERT INTO basic_lob_table
    VALUES('two', '02020202020202020202020202020202', 'twothreefourfivesix');
```

次の例では、BLOB と CLOB をホスト変数として使用し、前に定義した表のデータを SELECT INTO 文を使用して受け取ります。

```
...
BLOB blob;
CLOB clob;
#sql { SELECT one.b, two.c INTO :blob, :clob
        FROM basic_lob_table one, basic_lob_table two
        WHERE one.x='one' AND two.x='two' };
#sql { INSERT INTO basic_lob_table VALUES('three', :blob, :clob) };
...
```

BASIC_LOB_TABLE の先頭行から BLOB を取り出し、2 番目の行から CLOB を取り出します。その後で、前の操作で選択した BLOB と CLOB を使用して、3 番目の行を表に挿入します。

LOB と BFILE を使用したイテレータの宣言

型 BLOB、CLOB および BFILE は、SQLJ の位置指定および名前指定イテレータの列型として使用できます。これらのイテレータには、該当する実行可能な SQLJ 操作の結果を格納できます。

次に示したサンプルの宣言は、以後繰り返し使用します。

```
#sql iterator NamedLOBIter(CLOB c);
#sql iterator PositionedLOBIter(BLOB);
#sql iterator NamedFILEIter(BFILE bf);
```

LOB および BFILE 型ホスト変数と名前指定イテレータの結果

次の例では、表 BASIC_LOB_TABLE と前の例で定義したメソッド readFromLongest() を使用し、名前指定イテレータで CLOB を使用します。BLOB および BFILE のコードも、同様の記述になります。

次のように宣言します。

```
#sql iterator NamedLOBIter(CLOB c);
```

実行可能コードを示します。

```
...
NamedLOBIter iter;
#sql iter = { SELECT c FROM basic_lob_table };
if (iter.next())
    CLOB c1 = iter.c();
if (iter.next())
    CLOB c2 = iter.c();
iter.close();
readFromLongest(c1, c2);
...
```

この例では、イテレータを使用して、BASIC_LOB_TABLE の最初の 2 行から CLOB を 2 つ選択します。次に readFromLongest() メソッドを使用して、2 つのうちの大きい方を出力します。

LOB および BFILE 型ホスト変数と、位置指定イテレータの FETCH INTO ターゲット

型 BLOB、CLOB および BFILE のホスト変数は、位置指定イテレータで使用できます。イテレータの該当列の属性が同じ型の場合は、対応付けられている FETCH INTO 文の INTO リストに、これらのホスト変数を記述できます。

次の例では、表 BASIC_LOB_TABLE と、前の例で定義したメソッド writeToBlob() を使用します。CLOB および BFILE のコードも、同様の記述になります。

次のように宣言します。

```
#sql iterator PositionedLOBIter(BLOB);
```

実行可能コードを示します。

```
...
PositionedLOBIter iter;
BLOB blob = null;
#sql iter = { SELECT b FROM basic_lob_table };
for (long rowNum = 1; ; rowNum++)
{
    #sql { FETCH :iter INTO :blob };
    if (iter.endFetch()) break;
    writeToBlob(blob, 512*rowNum);
}
iter.close();
...
```

この例では、BASIC_LOB_TABLE 内の各 BLOB に対して、writeToBlob() をコールします。各行から、512 バイトずつデータが書き込まれます。

Oracle ROWID のサポート

Oracle 固有の型 ROWID は、データベース表の各行の一意のアドレスを格納するための型です。クラス oracle.sql.ROWID は、ROWID 情報をラッピングします。型 ROWID の変数をバインドおよび定義するときに、このクラスを使用します。

oracle.sql.ROWID 型の変数は、Oracle9i に接続する SQLJ アプリケーションで次のように使用できます。

- SQLJ 実行文中 (INTO リストでも使用可) の IN、OUT または INOUT ホスト変数として
- ストアド・ファンクション・コールの戻り値として
- イテレータ宣言での列型として (名前指定および位置指定イテレータ)

注意： 現行の Oracle では位置指定 UPDATE または位置指定 DELETE で、SQLJ 仕様どおりに WHERE CURRENT OF 句を使用できません。かわりに、この機能をシミュレートする ROWID の使用をお勧めします。

ROWID を使用したイテレータ宣言

型 `oracle.sql.ROWID` は、次の宣言のように、SQLJ の位置指定および名前指定イテレータの列型として使用できます。

```
#sql iterator NamedRowidIter (String ename, ROWID rowid);

#sql iterator PositionedRowidIter (String, ROWID);
```

ROWID ホスト変数と名前指定イテレータの SELECT 結果

ROWID オブジェクトは、SQLJ 実行文で IN、OUT および INOUT パラメータとして使用できます。また、イテレータの ROWID 型の列を設定できます。次のコード例では、前の例で示した宣言を使用します。

次のように宣言します。

```
#sql iterator NamedRowidIter (String ename, ROWID rowid);
```

実行可能コードを示します。

```
...
NamedRowidIter iter;
ROWID rowid;
#sql iter = { SELECT ename, rowid FROM emp };
while (iter.next())
{
    if (iter.ename().equals("CHUCK TURNER"))
    {
        rowid = iter.rowid();
        #sql { UPDATE emp SET sal = sal + 500 WHERE rowid = :rowid };
    }
}
iter.close();
...
```

前述の例では、ROWID に基づき、従業員 **Chuck Turner** の給与が \$500 増えます。この方法で `WHERE CURRENT OF` セマンティクスをコード化することをお勧めします。

ストアド・ファンクションの結果としての ROWID

Oracle9i に次のファンクションがあるものとします。

```
CREATE OR REPLACE function get_rowid (name varchar2) return rowid is
    rid rowid;
BEGIN
    SELECT rowid INTO rid FROM emp WHERE ename = name;
    RETURN rid;
END get_rowid;
```

このストアド・ファンクションの場合は、このファンクションの戻り値の代入型として、ROWID オブジェクトが次のように使用されます。

```
ROWID rowid;
#sql rowid = { values(get_rowid('AMY FEINER')) };
#sql { UPDATE emp SET sal = sal + 500 WHERE rowid = :rowid };
```

この例では、ROWID に基づき、従業員 Amy Feiner の給与が \$500 増えます。

SELECT INTO のターゲットとしての ROWID

型 ROWID のホスト変数は、SELECT INTO 文の INTO リストに記述できます。

```
ROWID rowid;
#sql { SELECT rowid INTO :rowid FROM emp WHERE ename='CHUCK TURNER' };
#sql { UPDATE emp SET sal = sal + 500 WHERE rowid = :rowid };
```

この例では、ROWID に基づき、従業員 Chuck Turner の給与が \$500 増えます。

ROWID ホスト変数と位置指定イテレータの FETCH INTO ターゲット

型 ROWID のホスト変数は、FETCH INTO 文の INTO リストに記述できます（イテレータ内の該当列の属性が同じ型の場合）。

次のように宣言します。

```
#sql iterator PositionedRowidIter (String, ROWID);
```

実行可能コードを示します。

```
...
PositionedRowidIter iter;
ROWID rowid = null;
String ename = null;
#sql iter = { SELECT ename, rowid FROM emp };
while (true)
{
    #sql { FETCH :iter INTO :ename, :rowid };
    if (iter.endFetch()) break;
    if (ename.equals("CHUCK TURNER"))
    {
        #sql { UPDATE emp SET sal = sal + 500 WHERE rowid = :rowid };
    }
}
iter.close();
...
```

この例は、前述の名前指定イテレータの例に似ていますが、位置指定イテレータの必須 `FETCH INTO` 構文を使用します。

Oracle の REF CURSOR 型のサポート

Oracle PL/SQL と Oracle SQLJ では、データベースのカーソルを表すカーソル変数を使用できます。

REF CURSOR 型の概要

カーソル変数は、機能的には JDBC の結果セットに相当し、問合せ結果をカプセル化します。カーソル変数は `REF CURSOR` と呼ばれますが、`REF CURSOR` 自体は型指定子であって、型名ではありません。指定子でなく、名前付き `REF CURSOR` 型を指定する必要があります。次に、`REF CURSOR` の型指定の例を示します。

```
TYPE EmpCurType IS REF CURSOR;
```

Oracle の `REF CURSOR` 型のパラメータは、ストアド・プロシージャとストアド・ファンクションで戻されます。`REF CURSOR` パラメータを戻すには、`PL/SQL` を使用する必要があります、`SQL` のみを使用した場合は戻りません。`PL/SQL` のストアド・プロシージャやストアド・ファンクションでは、任意の名前付き `REF CURSOR` 型の変数を宣言したり、`SELECT` 文を実行したり、`REF CURSOR` 変数に戻り値としての結果を代入したりできます。

カーソル変数の詳細は、『`PL/SQL ユーザーズ・ガイド`およびリファレンス』を参照してください。

SQLJ の REF CURSOR 型

Oracle SQLJ では、REF CURSOR 型を任意のイテレータ・クラス型または型 `java.sql.ResultSet` のイテレータ列またはホスト変数にマッピングできます。ただし、マッピングの対象となるホスト変数は OUT 型のみです。次に、REF CURSOR 型の用途をまとめます。

- ストアド・ファンクションの戻り値に使用する結果式として
- ストアド・プロシージャまたはストアド・ファンクションの出力パラメータの出力ホスト式として
- INTO リスト中の出力ホスト式として
- イテレータ列として

Oracle SQL の CURSOR 演算子は、外部の SELECT 文の内側にネストされている SELECT に対して使用できます。これによって、REF CURSOR オブジェクトをイテレータ列またはイテレータの `ResultSet` 列に書き込むか、REF CURSOR オブジェクトをイテレータ・ホスト変数または INTO リストの `ResultSet` ホスト変数に書き込むことが可能になります。

暗黙的な REF CURSOR 変数の使用例（たとえば、CURSOR 演算子の例など）は、3-47 ページの「[ホスト変数としてのイテレータおよび結果セットの使用](#)」を参照してください。

注意：

- REF CURSOR 型には、型コード `OracleTypes.CURSOR` を使用する必要があります。
 - REF CURSOR 型は、`oracle.sql` クラスではサポートされていません。`java.sql.ResultSet` クラスまたはイテレータ・クラスを使用する必要があります。リソースを解放するために、結果セットまたはイテレータは、処理完了時点でクローズしてください。
-

REF CURSOR の例

無名ブロックから REF CURSOR 型を取り出すサンプル・メソッドを次に示します。

```
private static EmpIter refCursInAnonBlock(String name, int no)
    throws java.sql.SQLException {
    EmpIter emps = null;
    System.out.println("Using anonymous block for ref cursor..");
    #sql { begin
        INSERT INTO emp (ename, empno) VALUES (:name, :no);
        OPEN :out emps FOR SELECT ename, empno FROM emp ORDER BY empno;
    end
    };
    return emps;
}
```


その他の Oracle9i データ型のサポート

`oracle.sql` のあらゆるクラスは、イテレータ列にも、入力、出力または入出力のホスト変数にも、標準 Java 型と同じように使用できます。たとえば、前述したクラスでは、`oracle.sql.NUMBER`、`oracle.sql.CHAR`、`oracle.sql.RAW` などのクラスがサポートされています。

`oracle.sql.*` クラスは Java 型の形式に変換する必要がないので、対応する Java 型よりも効率および精度面で優れています。ただし、標準 Java プログラムで使用する場合、またはエンド・ユーザーで表示する場合は、データを標準 Java 型に変換する必要があります。

BigDecimal のサポート強化

SQLJ では、`java.math.BigDecimal` を次の用途に使用できます。

- SQLJ 実行文中のホスト変数として
- ストアド・ファンクション・コールの戻り値として
- イテレータの列型として

標準 SQLJ で数値データおよび 10 進データの値を `BigDecimal` として取得するには、JDBC のデフォルト・マッピングを使用する必要があります。(JDBC のデフォルト・マッピングの詳細は、5-2 ページの表 5-1 を参照してください。)

標準 SQLJ に対し、Oracle SQLJ では、Oracle9i、Oracle JDBC ドライバ、Oracle 固有コード生成（または Oracle カスタマイザ）および Oracle SQLJ ランタイムを使用すると、数値から変換可能なデータ型をデフォルト以外の型にマッピングできます。変換可能なデータ型としては、`CHAR`、`VARCHAR2`、`LONG` および `NUMBER` があります。たとえば、`CHAR` 列のデータは、`BigDecimal` 変数への取込みができます。ただし、エラーを防ぐために、文字データの内容を数字のみにする必要があります。

注意： `BigDecimal` クラスは標準 `java.math` パッケージにあります。

オブジェクト、コレクションおよび OPAQUE 型

この章では、Oracle SQLJ におけるユーザー定義 SQL 型のサポート方法について説明します。ユーザー定義 SQL 型は、オブジェクト（および関連するオブジェクト参照）とコレクション（可変長配列およびネストした表）です。Oracle の JPublisher ユーティリティについても説明します。ユーザー定義 SQL 型に対応する Java クラスをこのユーティリティで生成できます。

この章の最後には Oracle OPAQUE 型に関する項目もあります。これらは、機能的にはオブジェクト型とほぼ同じですが、実装の種類が異なります。データは構造化オブジェクト形式ではなく、不透明なバイトのペイロードとして表されます。

次の項目について説明します。

- [Oracle のオブジェクトとコレクションについて](#)
- [カスタム Java クラス](#)
- [ユーザー定義データ型](#)
- [JPublisher とカスタム Java クラスの作成](#)
- [SQLJ 実行文の強い型指定のオブジェクトと参照](#)
- [SQLJ 実行文の強い型指定のコレクション](#)
- [シリアル化された Java オブジェクト](#)
- [弱い型指定のオブジェクト、参照およびコレクション](#)
- [Oracle OPAQUE 型](#)

Oracle のオブジェクトとコレクションについて

ここでは、Oracle9i のオブジェクトとコレクションの基本的な概念について説明します。

Oracle のオブジェクト、参照およびコレクションに関する概念および詳細は、『Oracle9i SQL リファレンス』および『Oracle9i アプリケーション開発者ガイド - 基礎編』を参照してください。

オブジェクトとコレクションの宣言方法の詳細は、6-21 ページの「[ユーザー定義データ型](#)」を参照してください。

オブジェクトとコレクションの概要

Oracle9i および Oracle SQLJ では、ユーザー定義の SQL オブジェクト型（複合データ構造体）、関連する SQL オブジェクト参照型およびユーザー定義の SQL コレクション型を使用できます。Oracle のオブジェクトとコレクションは、複合データ構造体であり、複数のデータ要素で構成されます。

Oracle SQLJ では、強い型指定または弱い型指定の、Java 表現のオブジェクト型、参照型およびコレクション型を、イテレータまたはホスト式で使用できます。強い型指定の表現では、オブジェクト型、参照型またはコレクション型が、JDBC2.0 標準の `java.sql.SQLData` インタフェース（オブジェクト型の場合のみ）または Oracle の `oracle.sql.ORAData` インタフェースを実装したカスタム Java クラスにマップされます。カスタム Java クラスは Oracle9i JPublisher ユーティリティを使用して自動的に生成することが可能です。このユーティリティは前述の 2 つのインタフェースをサポートしています。弱い型指定の表現では、クラス `oracle.sql.STRUCT`（オブジェクトの場合）、`oracle.sql.REF`（オブジェクト参照の場合）または `oracle.sql.ARRAY`（コレクションの場合）を使用します。弱い型指定の表現では、標準 `java.sql.Struct`、`Ref` または `Array` オブジェクトもかわりに使用できます。

「強い型指定」という用語は、特定の Java 型が SQL 名前指定（ユーザー定義）型に対応付けられている場合に使用されます。たとえば、PERSON 型と、対応する Person Java クラスがある場合です。

「弱い型指定」という用語は、Java 型が汎用的な方法で使用され、複数の SQL 名前指定型にマップできる場合に使用されます。Java クラス（インタフェース）には、SQL 型に特有の情報はありません。これは、`oracle.sql.STRUCT`、`REF` および `ARRAY` 型と `java.sql.Struct`、`Ref` および `Array` 型の場合です。

コード中に Oracle 拡張型を使用する場合は、次の要件が伴うので注意してください。

- Oracle JDBC ドライバを使用する必要があります。
- デフォルトの Oracle 固有コード生成を使用するか、または ISO コード生成の場合は必要に応じてプロファイルをカスタマイズします。（カスタマイザは、デフォルトのカスタマイザ `oracle.sqlj.runtime.util.OraCustomizer` をお勧めします。）

デフォルトの `-codegen=oracle` トランスレータ設定によって生成される Oracle 固有生成コードの場合、プロファイルが生成されないため、カスタマイズは該当しません。Oracle JDBC API は、生成された Java コードによって直接コールされます。

- アプリケーション実行時に Oracle SQLJ ランタイムを使用する必要があります。

Oracle SQLJ ランタイムと Oracle JDBC ドライバは、Oracle 拡張型をコード中に使用しない場合であっても、Oracle カスタマイザを使用する場合は常に必要です。

Oracle 固有のセマンティクス・チェックを行うには、適切なチェッカを使用する必要があります。デフォルトのチェッカ `oracle.sqlj.checker.OracleChecker` は、フロントエンドとして機能し、環境に応じたチェッカを起動します。JDBC ドライバを使用すると、Oracle 固有のチェッカが起動されます。

Oracle オブジェクトとコレクション用の Oracle 固有の型は、`oracle.sql` パッケージにあります。

セマンティクス・チェック用のトランスレータ・オプションの詳細は、8-32 ページの「[接続オプション](#)」および 8-67 ページの「[セマンティクス・チェックとオフライン解析のオプション](#)」を参照してください。

カスタム Java クラスの使用法

- この章では、主に、ユーザー定義型に対するカスタム Java クラスの使用法について説明しますが、`ORADATA` を実装したクラスは、他の Oracle SQL 型に対しても使用できます。`ORADATA` を実装したクラスを使用すると、SQL と Java 間のデータ転送時に任意の処理または変換を行えます。6-16 ページの「[ORADATA 実装のその他の使用例](#)」を参照してください。
- `SQLData` インタフェースは、カスタム・オブジェクト・クラス専用です。これに対し、`ORADATA` インタフェースは、どのカスタム Java クラスにも使用できます。

用語上の注意

- ユーザー定義の SQL オブジェクト型と SQL コレクション型は、ユーザー定義型（UDT）と呼ばれます。
- オブジェクト、参照およびコレクションのカスタム Java クラスは、それぞれカスタム・オブジェクト・クラス、カスタム参照クラスおよびカスタム・コレクション・クラスと呼ばれます。

Oracle オブジェクトの特長と機能の概要は、『Oracle9i アプリケーション開発者ガイド - オブジェクト・リレーショナル機能』を参照してください。

Oracle オブジェクトの基本概念

Oracle のオブジェクト（SQL オブジェクト）は複合データ構造体であり、関連するデータ項目（各従業員に関する事実など）を 1 つのデータ単位にまとめたものです。オブジェクト型は、機能的には Java のクラスに相当します。Java 型をインスタンス化してオブジェクトをいくつでも使用できるように、特定のオブジェクト型のオブジェクトをいくつでも設定して使用できます。

たとえば、属性 `name`（CHAR 型）、`address`（CHAR 型）、`phonenum`（CHAR 型）および `employeenumber`（NUMBER 型）を持つオブジェクト型 `EMPLOYEE` を定義できます。

Oracle のオブジェクトのメソッド（ストアド・プロシージャ）は、オブジェクト型への対応付けができます。これらのメソッドは、`static` メソッドまたはインスタンス・メソッドとして、PL/SQL または Java で実装できます。メソッドのシグネチャとして、任意の数の入力、出力または入出力パラメータを使用できます。すべては初期定義によって決まります。

Oracle コレクションの基本概念

Oracle のコレクション（SQL コレクション）は、次の 2 つのカテゴリに分類されます。

- 可変長配列（VARRAY 型）
- ネストした表（NESTED TABLE 型）

両カテゴリのコレクションは 1 次元ですが、複合オブジェクト型の要素を収容できます。VARRAY 型は、1 次元配列に対して使用します。一方、NESTED TABLE 型は、表の中に別の表を入れ子にする場合に使用します。VARRAY 型の変数を可変長配列と呼び、NESTED TABLE 型の変数をネストした表と呼びます。

VARRAY は、配列と同じように、データ要素の順序付けられた集合です。各要素に索引があり、すべての要素が同じデータ型です。VARRAY のサイズは、要素の最大数を示します。Oracle の VARRAY は、名前が示すように可変サイズです。各 VARRAY 型の宣言時に、VARRAY 型の最大サイズを指定する必要があります。

ネストした表は、順序付けされていない要素の集合です。表内のネストした表要素は、それ自体 SQL で問合せを実行します。表と同じように、ネストした表の行数は動的に算出されるので、作成時に指定する必要はありません。

注意： VARRAY の各要素またはネストした表の各行は、ユーザー定義のオブジェクト型にできます。したがって、ユーザー定義オブジェクト型の属性に対して、VARRAY 型および NESTED TABLE 型を使用できます。Oracle9i では、コレクション型のネストをサポートしています。VARRAY の要素またはネストした表の行は、別の VARRAY 型または NESTED TABLE 型にしたり、VARRAY 属性またはネストした表の属性を持つユーザー定義のオブジェクト型にできます。

オブジェクトとコレクションのデータ型

Oracle9i のユーザー指定のオブジェクト定義およびコレクション定義は、SQL のデータ型定義として機能します。これらのデータ型は、他のデータ型と同じように、表の列定義、SQL オブジェクトの属性定義およびストアド・プロシージャまたはストアド・ファンクションのパラメータの定義に使用できます。また、オブジェクト型を定義すると、そのオブジェクト参照型を他の SQL 参照型のように使用できます。

EMPLOYEE を Oracle のオブジェクトとして定義すると（6-4 ページの「[Oracle オブジェクトの基本概念](#)」を参照）、Oracle のデータ型になります。したがって、EMPLOYEE 型の列は、NUMBER 型の列と同じように、表への取込みができます。EMPLOYEE 列の各行に、EMPLOYEE オブジェクト全体を格納できます。REF EMPLOYEE 型の列も表に取り込めます。この列の内容は、EMPLOYEE オブジェクトへの参照です。

同様に、NUMBER の VARRAY(10) として VARRAY 型 MYVARR を定義したり、CHAR(20) の NESTED TABLE 型 NTBL を定義できます。コレクション型 MYVARR と NTBL は Oracle のデータ型になるので、これらの型の列を表に定義できます。MYVARR 列の各行は、最大 10 数字を収容する配列です。NTBL 列の各行は 20 文字で構成されます。

カスタム Java クラス

カスタム Java クラスの用途は、SQL と Java 間のデータの変換手段を提供し、データへのアクセスを可能にすることです。この機能は、特に、オブジェクトとコレクションのサポートにおいて重要です。データのカスタム変換を行う場合も、この機能が必要です。

SQLJ アプリケーションで使用する各ユーザー定義型（オブジェクトとコレクション）に対して、カスタム Java クラスを用意することをお勧めします。Oracle JDBC ドライバでのデータ変換には、これらのカスタム Java クラスのインスタンスを使用します。カスタム Java クラスを使用した方が、弱い型指定の oracle.sql.STRUCT、REF および ARRAY クラスを使用するより便利であり、エラーも発生しにくくなるためです。

カスタム Java クラスはファーストクラスの型です。ユーザー定義 SQL 型に対する読み込み / 書き込みは、ユーザーが意識することなく行われます。

SQLJ イテレータまたはホスト式で使用するために、カスタム Java クラスでは oracle.sql.ORAData（および ORADataFactory）インタフェースまたは標準 java.sql.SQLData インタフェースを実装する必要があります。ここでは、これらのインタフェースとカスタム Java クラス機能の概要を説明します。次の項目について説明します。

- [カスタム Java クラスのインタフェース指定](#)
- [オブジェクト・メソッドでのカスタム Java クラスのサポート](#)
- [カスタム Java クラスの要件](#)
- [カスタム Java クラスのコンパイル](#)
- [カスタム・データの読み込みと書き込み](#)
- [ORAData 実装のその他の使用例](#)

カスタム Java クラスのインタフェース指定

ここでは、ORADATA および ORADATAFACTORY インタフェースと標準 SQLDATA インタフェースの指定について説明します。

Oracle9i には、ユーザー定義型にマップされるカスタム Java クラスから、Oracle 固有の機能を使用するために一連の新しい API があります。新しい API は、`oracle.sql.ORADATA` および `oracle.sql.ORADATAFACTORY` です。

以前この機能のために使用されていた `oracle.sql.CustomDatum` および `oracle.sql.CustomDatumFactory` インタフェースは、Oracle9i では使用されていませんが、下位互換性を保つために現在もサポートされています。Oracle8i JDBC ドライバを使用している場合は、`CustomDatum` インタフェースを使用する必要があります。

ORADATA および ORADATAFACTORY の指定

Oracle には、インタフェース `oracle.sql.ORADATA` と関連インタフェース `oracle.sql.ORADATAFACTORY` が用意されています。これらのインタフェースを使用して、Oracle のオブジェクト型、参照型およびコレクション型をカスタム Java クラスにマップすることができます。

データの送信や取出しには、`oracle.sql.Datum` オブジェクトの形式を使用します。元のデータは、`oracle.sql.Datum` のサブクラス（`oracle.sql.STRUCT` など）の形式です。このデータは SQL 形式のままになっており、`oracle.sql.Datum` オブジェクトはラッパーとしてのみ機能します。Oracle の拡張型をサポートする `oracle.sql` パッケージ内のクラスの詳細は、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。

Java 形式から SQL 形式へのデータ変換に使用する `toDatum()` メソッドは、ORADATA インタフェースで指定します。このメソッドでは、接続オブジェクトを入力として見なし、データを `oracle.sql.*` 表現に変換します。JDBC ドライバは、実行時に接続オブジェクトを使用して、型チェックと型変換を行います。ORADATA と `toDatum()` は、次のように指定します。

```
interface oracle.sql.ORADATA
{
    oracle.sql.Datum toDatum(java.sql.Connection c) throws SQLException;
}
```

カスタム Java クラスのインスタンスを作成し、SQL 形式から Java 形式に変換する `create()` メソッドは、ORADATAFACTORY インタフェースで指定します。このメソッドは、入力として `Datum` オブジェクトを取ります。このオブジェクトには、データおよび型コード（基になるデータが SQL 型であることを示す `OracleTypes.RAW` など）があります。このメソッドで戻されるカスタム Java クラスのオブジェクトには、ORADATA インタフェースが実装されています。このオブジェクトは、入力された `Datum` オブジェクトからデータを取得します。ORADATAFACTORY と `create()` は、次のように指定します。


```
interface oracle.sql.ORADataFactory
{
    oracle.sql.ORAData create(oracle.sql.Datum d, int sqlType)
                        throws SQLException;
}
```

ORAData インタフェースと ORADataFactory インタフェース間の関係を確立するには、ORAData インタフェースを実装したカスタム Java クラスに、static `getORADataFactory()` メソッドを実装する必要があります。このメソッドで戻されるオブジェクトは、ORADataFactory インタフェースが実装されているので、カスタム Java クラスのインスタンスを作成できます。戻されたオブジェクト自体が、カスタム Java クラスのインスタンスの場合もあります。この場合、Oracle JDBC ドライバは、その `create()` メソッドを使用して、カスタム Java クラスのインスタンスを生成します。

注意： JPublisher の出力は、ORAData インタフェースおよびその `toDatum()` メソッドと、ORADataFactory インタフェースおよびその `create()` メソッドとを、1つのカスタム Java クラスに実装できます。ただし、`toDatum()` と `create()` を別々のインタフェースに指定した場合は、それぞれを別個のクラスに実装することも可能になります。ORAData およびその `toDatum()` メソッドと、`getORADataFactory()` メソッドを1つのカスタム Java クラスに実装する一方で、ORADataFactory およびその `create()` メソッドを別個のファクトリ・クラスに実装することも可能です。ここでは、両方のインタフェースを1つのクラスに実装する場合について説明します。

ORAData を実装したクラスに対する Oracle SQLJ での要件は、6-10 ページの「[ORAData を実装したクラスに対する Oracle の要件](#)」を参照してください。

ORAData インタフェースと ORADataFactory インタフェース、`oracle.sql` クラスおよび `OracleTypes` クラスの詳細は、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。

JPublisher を使用して `-usertypes=oracle` と指定すると、JPublisher のカスタム Java クラスが生成されます。このクラスには、ORAData インタフェース、ORADataFactory インタフェースおよび `getORADataFactory()` メソッドが実装されます。または、下位互換性を保つために、JPublisher の `-compatible` オプションとともに `-usertypes=oracle` をオプション指定して、`CustomDatum` および `CustomDatumFactory` インタフェースを使用します。詳細は、『Oracle9i JPublisher ユーザーズ・ガイド』を参照してください。

ORADData と CustomDatum インタフェースの比較

Oracle9i では、`oracle.jdbc.driver` クラスにかわって `oracle.jdbc` インタフェースが採用されているため、カスタマイズされたオブジェクトの利用に以前使用されていた `oracle.sql.CustomDatum` および `oracle.sql.CustomDatumFactory` インタフェースは使用されなくなりました。かわりに、新しいインタフェースの `oracle.sql.ORADData` および `oracle.sql.ORADDataFactory` が使用されます。`CustomDatum` インタフェースのように、これらを標準 `SQLData` インタフェースの Oracle 固有の代替インタフェースとして使用できます。`CustomDatum` インタフェースは、下位互換性を保つために現在もサポートされています。

`CustomDatum` および `CustomDatumFactory` は、次のように定義します。

```
public interface CustomDatum
{
    oracle.sql.Datum toDatum(
        oracle.jdbc.driver.OracleConnection conn
    ) throws SQLException;

    public interface CustomDatumFactory
    {
        oracle.sql.CustomDatum create(
            oracle.sql.Datum d, int sqlType
        ) throws SQLException;
    }
}
```

接続 `conn` および型コード `sqlType` は、6-6 ページの「[ORADData および ORADDataFactory の指定](#)」の `ORADData` および `ORADDataFactory` で説明されているように使用されます。ただし、`CustomDatum` では、標準 `Connection` 型ではなく、必ず Oracle 固有の `OracleConnection` 型を使用してください。

SQLData の指定

標準 JDBC 2.0 には、構造化オブジェクト型を Java クラスにマッピングし、変換するためのインタフェース `java.sql.SQLData` が提供されています。このインタフェースは、構造化オブジェクト型のみをマッピングの対象としているため、コレクションや配列などの SQL 型はマッピングできません。

`SQLData` インタフェースは、JDBC 2.0 標準であり、`readSQL()` メソッドを指定してデータを Java オブジェクトに読み込む一方で、`writeSQL()` メソッドを指定して Java オブジェクトからデータベースに書き込みます。

`SQLData` を実装したクラスに要求される機能の詳細は、6-12 ページの「[SQLData を実装したクラスに対する要件](#)」を参照してください。

標準 `SQLData` 機能の追加詳細は、Sun 社の JDBC 2.0 以上の API 仕様を参照してください。

`JPublisher` を使用して `-usertypes=jdbc` と指定すると、`JPublisher` のカスタム Java クラスが生成されます。このクラスには、`SQLData` インタフェースが実装されています。

オブジェクト・メソッドでのカスタム Java クラスのサポート

Oracle オブジェクト・メソッドは、カスタム Java クラスのラッパーから起動できます。基になるストアド・プロシージャの記述言語が PL/SQL、Java のどちらであるか、また SQL に公開されているかどうかは、ユーザーには表示されません。

Java のラッパー・メソッドでサーバー側メソッドを起動するには、サーバーと通信するための接続が必要です。接続オブジェクトは、パラメータとして明示的に指定することも、別の方法（カスタム Java クラスの属性など）で対応付けることも可能です。

ラッパー・メソッドで使用する接続オブジェクトを非静的属性とした場合、このラッパー・メソッドをカスタム Java クラスのインスタンス・メソッドとすると、この接続にアクセスできます。JPublisher で生成したカスタム Java クラスでは、この手法が採用されています。

Oracle オブジェクト・メソッドの出力および入出力パラメータに関しては、次のことに留意してください。ストアド・プロシージャ（SQL オブジェクト・メソッド）によって、ある引数の内部状態が変更された場合、ストアド・プロシージャに渡された実引数も変更されます。Java で記述した場合、こうした現象は起こりません。JDBC 出力パラメータがストアド・プロシージャ・コールから戻る場合、新規に作成されたオブジェクトに格納する必要があります。元のオブジェクト識別性は失われます。

出力または入出力パラメータをコール元に戻す方法としては、パラメータを配列要素として渡す方法もあります。入出力パラメータの場合は、ラッパー・メソッドが入力として配列要素をとります。処理後、ラッパーは出力を配列要素に代入します。JPublisher で生成したカスタム Java クラスでは、この手法が使用されます。つまり、出力または入出力パラメータをそれぞれ単一要素配列で渡します。

JPublisher を使用すると、デフォルトではラッパー・メソッドが実装されます。生成されたクラスには、SQLData インタフェースまたは ORADData インタフェースが実装されており、このことが当てはまります。このデフォルトの機能を無効化するには、JPublisher の `-methods` フラグを `false` に設定します。詳細は、『Oracle9i JPublisher ユーザーズ・ガイド』を参照してください。

注意： カスタム Java クラスを自分で実装する場合、ラッパー・メソッドを実装する方法としていくつかの選択肢があります。サーバーでのデータ処理には、SQL オブジェクト・メソッドによって直接実行する場合と、クライアントからサーバーへオブジェクト値を転送して、サーバー上でメソッドを実行する場合があります。JPublisher でのラッパー・メソッドの実装および実装による効果については、6-43 ページの「[JPublisher で生成されるラッパー・メソッド](#)」を参照してください。

カスタム Java クラスの要件

カスタム Java クラスに求められる要件としては、有効なホスト変数型として Oracle SQLJ トランスレータで認識できることおよびトランスレータでタイプ・チェックできることが挙げられます。

こうした機能をカスタム Java クラスでサポートするための Oracle 固有の要件については、この項で説明します。ORADATA および SQLDATA 実装の両方に対する要件についても解説します。

注意： ユーザー定義型用のカスタム Java クラスは、このマニュアルで「ラッパー・クラス」と呼ばれている場合があります。

ORADATA を実装したクラスに対する Oracle の要件

ORADATA を実装するための Oracle での要件は、どのカスタム Java クラスの場合にも基本的には同じです。ただし、クラスによるマッピング先が、オブジェクト、オブジェクト参照、コレクションまたはその他の SQL 型のどれに該当するかによって、この要件は若干異なってきます。

この要件を次に示します。

- oracle.sql.ORADATA インタフェースを実装したクラスであること。
- oracle.sql.ORADATAFactory オブジェクトを戻り値とするメソッド getORADATAFactory() を実装したクラスであること。

```
public static oracle.sql.ORADATAFactory getORADATAFactory();
```

現在推奨されていない CustomDatum インタフェースを使用する場合、クラスでメソッド getFactory() を実装します。このメソッドは次のように oracle.sql.CustomDatumFactory オブジェクトを戻します。

```
public static oracle.sql.CustomDatumFactory getFactory();
```

- クラスが文字列定数 _SQL_TYPECODE を持つこと。この定数は、toDatum() が戻す Datum サブクラス・インスタンスの oracle.jdbc.OracleTypes 型コードで初期化される必要がある。
 - カスタム・オブジェクト・クラスの場合

```
public static final int _SQL_TYPECODE = OracleTypes.STRUCT;
```
 - カスタム参照クラスの場合

```
public static final int _SQL_TYPECODE = OracleTypes.REF;
```

- カスタム・コレクション・クラスの場合

```
public static final int _SQL_TYPECODE = OracleTypes.ARRAY;
```

これ以外のクラスの場合は、それぞれ適切な型コードで初期化します。たとえば、カスタム Java クラスで、RAW フィールドに対して Java オブジェクトをシリアル化およびシリアル化解除するには、`_SQL_TYPECODE` を `OracleTypes.RAW` に初期化します。6-65 ページの「[シリアル化された Java オブジェクト](#)」を参照してください。

(`OracleTypes` クラスは、各 Oracle データ型の型コード (整数定数) のみを定義します。標準 SQL 型の `OracleTypes` エントリは、標準 `java.sql.Types` 型定義クラスのエントリと同じです。)

- `_SQL_TYPECODE` が `STRUCT`、`REF` または `ARRAY` のいずれかに該当する (つまり、オブジェクト、オブジェクト参照またはコレクションを表す) カスタム Java クラスの場合、該当のユーザー定義型の名前を示す定数を保持していること。

- カスタム・オブジェクト・クラスおよびカスタム・コレクション・クラスの定数 `_SQL_NAME` (文字列) は、ユーザー定義型に対して宣言した SQL 名に初期化する必要があります。次に例を示します。

```
public static final String _SQL_NAME = UDT name;
```

カスタム・オブジェクト・クラスの例 (ユーザー定義の `PERSON` オブジェクトの場合)

```
public static final String _SQL_NAME = "PERSON";
```

または (スキーマを指定する場合)

```
public static final String _SQL_NAME = "SCOTT.PERSON";
```

カスタム・コレクション・クラスの例 (`PERSON` オブジェクトのコレクションを `PERSON_ARRAY` と宣言した場合)

```
public static final String _SQL_NAME = "PERSON_ARRAY";
```

- カスタム参照クラスの定数 `_SQL_BASETYPE` (文字列) は、参照先のユーザー定義型に対して宣言した SQL 名に初期化する必要があります。

```
public static final String _SQL_BASETYPE = UDT name;
```

カスタム参照クラスの例 (`PERSON` 参照の場合)

```
public static final String _SQL_BASETYPE = "PERSON";
```

前述以外の `ORADData` で使用する場合、UDT 名の指定はありません。

使用上の注意

- コレクション型の名前は、ベース型ではなく、コレクション型を表す名前にします。たとえば、PERSON オブジェクトの VARRAY または NESTED TABLE 型の PERSON_ARRAY を宣言した場合は、_SQL_NAME エントリで指定するコレクション型の名前を、PERSON ではなく、PERSON_ARRAY にします。
- SQL 型を _SQL_NAME フィールドに指定するときに、SQL 型が大 / 小文字を区別する方法で（引用符内で）宣言されている場合、CaseSensitive または SCOTT.CaseSensitive など宣言されているとおりに SQL 名を指定する必要があります。（これは、引用符内で同様に名前の大 / 小文字を区別する JPublisher 入力ファイルでの使用方法と異なります。）SQL 型を大 / 小文字を区別して宣言していない場合（引用符なし）、ADDRESS や SCOTT.ADDRESS のように、すべて大文字で SQL 名を指定する必要があります。

JPublisher では、大 / 小文字の区別と JPublisher の -omit_schema_names 設定がある場合はそれに従って、自動的にこのフィールドの値を生成します。

SQLData を実装したクラスに対する要件

ISO SQLJ 規格では、SQLData インタフェースを実装するクラスへの型マップ定義の要件を概説しています。

一方、SQLData ラッパー・クラスは、public static final フィールドで SQL 関連オブジェクト型を識別できます。この非標準機能は、Oracle SQLJ リリース 8.1.6 から使用可能であり、サポートも継続されています。

次の点は重要なので、十分に注意してください。

- マッピングの指定に型マップを使用する場合でも、標準でない public static final フィールドを使用する場合でも、一貫して指定する必要があります。public static final フィールドを不要にする目的ですべての関連マッピングを指定する型マップを使用するか、または型マップをまったく使用せずにすべてのマッピングを public static final フィールドで指定します。
- SQLData は、ORADData とは異なり、構造化オブジェクト型のマッピング専用です。オブジェクト参照やコレクション / 配列などの SQL 型のマッピングには、SQLData を使用できません。ORADData を使用しない場合、オブジェクト参照とコレクションをマッピングする方法は、それぞれ弱い型指定の java.sql.Ref と java.sql.Array、または oracle.sql.REF と oracle.sql.ARRAY を使用する方法のみです。
- SQLData 実装には、JDK 1.2.x 以上の環境が必要です。JDK 1.1.x 環境では、oracle.jdbc2 パッケージを使用すると、JDBC 2.0 の拡張型が Oracle JDBC でサポートされますが、Oracle SQLJ ではサポートされません。

- SQL 型から Java 型へのマッピングを後述のように指定するとき、SQL 型が大 / 小文字を区別する方法で（引用符内で）宣言されている場合、CaseSensitive または SCOTT.CaseSensitive など宣言されているとおりに SQL 名を指定する必要があります。（これは、引用符内で同様に名前の大 / 小文字を区別する JPublisher 入力ファイルでの使用方法と異なります。）SQL 型を大 / 小文字を区別して宣言していない場合（引用符なし）、ADDRESS や SCOTT.ADDRESS のように、すべて大文字で SQL 名を指定する必要があります。

型マップ・リソースで指定したマッピング まず、ISO SQLJ 規格に従ったマッピング表現を想定します。Address、pack.Person および pack.Manager.InnerPM（InnerPM は Manager の内部クラスです）が java.sql.SQLData を実装している 3 つのラッパー・クラスであるとしします。

- これらのクラスは、宣言済みの接続コンテキスト型の明示的接続コンテキスト・インスタンスを使用する文でのみ採用する必要があります。たとえば、この型を SDContext と呼ぶとします。

例：

```
Address          a =...;
pack.Person      p =...;
pack.Manager.InnerPM pm =...;
SDContext ctx = new SDContext(url,user,pwd,false);
#sql [ctx] { ... :a ... :p ... :pm ... };
```

- 接続コンテキスト型は、java.util.PropertyResourceBundle を実装している関連クラスを指定する with 属性の typeMap で宣言されている必要があります。前述の例では、SDContext が次のように宣言されています。

```
#sql public static context SDContext with (typeMap="SDMap");
```

- 型マップのリソースは、SQL オブジェクト型から java.sql.SQLData インタフェースを実装する Java クラスにマッピングする必要があります。このマッピングは、次の形式のエントリで指定します。

```
class.<java_class_name>=STRUCT <sql_type_name>
```

キーワード STRUCT は省略可能です。例では、リソース・ファイル SDMap.properties には次のエントリが含まれます。

```
class.Address=STRUCT SCOTT.ADDRESS
class.pack.Person=PERSON
class.pack.Manager$InnerPM=STRUCT PRODUCT_MANAGER
```

パッケージとクラス名の分割には「.」を使用しますが、内部クラス名を分割するには文字「\$」を使用する必要があります。

重要： デフォルトの Oracle 固有コード生成を使用している場合、コンテキスト型が `SDContext` の文に使用されるイテレータにも同じ型マップ `SDMap` が宣言されていることが必要です。次に例を示します。

```
#sql public static iterator SDIter with (typeMap="SDMap");
...
SDContext sdctx = ...
SDIter sditer;
#sql [sdctx] sditer = { SELECT ...};
```

これにより、イテレータ・クラスに正しいコードが生成されたことを確認します。

マッピングを型マップ・リソースに指定するこのメカニズムは、次に説明する非標準の方法よりも複雑になっています。さらに、型マップ・リソースをデフォルトの接続コンテキストと関連付けることはできません。すべてのマッピング情報が 1 つの場所、型マップ・リソースに置かれるというメリットがあります。つまり、コンパイル済みアプリケーション上の型マッピングは後になっても簡単に調整できます。たとえば、新規 SQL 型および Java ラッパーを拡張 SQL-Java 型階層に含めることができます。

次の点に注意してください。

- この機能を使用するには、SQLJ runtime12 または runtime12ee ライブラリを採用する必要があります。型マップは `java.util.Map` オブジェクトで表されます。これらは、SQLJ ランタイム API で公開されていますが、JDK 1.1 または汎用ランタイム・ライブラリでサポートされていません。
- `SQLData` ラッパー・クラスが SQLJ 文の OUT または INOUT パラメータに現れた場合、Oracle SQLJ ランタイムおよび Oracle 固有コード生成またはプロファイルのカスタマイズを使用する必要があります。これは、Oracle JDBC がそのようなパラメータの SQL 型を `registerOutParameter()` で必要とするためです。さらに、OUT パラメータ型を登録するため、SQL 型は変換中に有力な型マップにより凍結されます。
- SQLJ 型マップは基礎となる接続で使用している JDBC 型マップとは独立しています。このため、`SQLData` ラッパーを使用している SQLJ コードおよび JDBC コードを混在させている場合は、注意が必要です。しかし、指定された SQLJ 接続コンテキスト上の有効な型マップは、簡単に抽出できます。

```
ctx.getTypeMap();
```


ラッパー・クラスの静的フィールドで指定されたマッピング 一方、SQLData を実装しているクラスは、次の非標準要件を満たすことができます。

- Java クラスは `public static final String` 値のフィールド `_SQL_NAME` を宣言します。このフィールドには、Java クラスによりラップされた SQL 型名を定義します。

例として、Address クラスには次のようなフィールド定義があります。

```
public static final String _SQL_NAME="SCOTT.ADDRESS";
```

次の宣言は `pack.Person` 内にあります。

```
public static final String _SQL_NAME="PERSON";
```

また、クラス `pack.Manager.InnerPM` には次の要素があります。

```
public static final String _SQL_NAME="PRODUCT_MANAGER";
```

JPublisher では、常に `_SQL_NAME` フィールドとともに SQLData ラッパー・クラスが生成されます。ただし、このフィールドは型マップを参照する SQLJ 文では無視されます。

注意：

- `_SQL_NAME` フィールドを実装するクラスが明示的接続コンテキスト型および関連型マップとともに SQLJ 文内で使用されており、かつ `_SQL_NAME` フィールドが無視されている場合、既存の SQLJ プログラムの新規 ISO SQLJ 規格への移行は簡素化されます。
 - `_SQL_NAME` フィールドで指定した静的 SQL-Java 型の対応は、基礎となる接続で使用している JDBC 型マップからは独立しています。このため、SQLData ラッパーを使用している SQLJ コードおよび JDBC コードを混在させている場合は、注意が必要です。
-
-

カスタム Java クラスのコンパイル

SQLJ のコマンドラインには、アプリケーションの `.sqlj` ファイルと一緒に、カスタム Java クラス (ORADData または SQLData を実装したクラス) の `.java` ファイルを指定できます。ただし、この指定が必要ではない場合もあります。それは、SQLJ の `-checksource` フラグを `true` (デフォルト) に設定し、CLASSPATH にカスタム Java ソースの保存ディレクトリを指定した場合です。(ここでは、カスタム・オブジェクトとカスタム・コレクション用に、`.sqlj` ファイルではなく `.java` ファイルを作成していることを想定しています。`.sqlj` ファイルは、SQLJ コマンドラインで指定される必要があります。)

たとえば、ObjectDemo.sqlj で Oracle のオブジェクト型 ADDRESS および PERSON を使用し、これらのオブジェクトのカスタム Java クラスが生成されている場合は、次のように SQLJ を実行できます。

- `-checksource=true` (デフォルト) を指定し、`CLASSPATH` にカスタム Java ソースの保存ディレクトリを指定した場合は、次のようにします。

```
% sqlj ObjectDemo.sqlj
```

- `-checksource=false` を指定した場合は、次のようにします (この行は、全体を 1 行内に入力します)。

```
% sqlj ObjectDemo.sqlj Address.java AddressRef.java Person.java PersonRef.java
```

また、Java コンパイラを使用してカスタム `.java` ソース・ファイルを直接コンパイルすることもできます。この場合は、`.sqlj` ファイルの変換前に、コンパイルする必要があります。

SQLJ トランスレータの実行方法については、[第 8 章「トランスレータのコマンドラインとオプション」](#)を参照してください。`-checksource` フラグの詳細は、8-65 ページの「[型解決を目的としたソース・チェック \(-checksource\)](#)」を参照してください。

注意： ORADATA 実装は Oracle 固有の機能を必要とするので、トランスレータの移植設定を `-warn=noportable` (デフォルト) に設定する必要があります。この設定にしない場合、移植性に関する警告がいくつか通知されます。`-warn` のフラグの詳細は、8-44 ページの「[トランスレータからの警告 \(-warn\)](#)」を参照してください。

カスタム・データの読み込みと書き込み

カスタム Java クラス・インスタンスを使用すると、Oracle SQLJ および JDBC では、ユーザー定義型の読み込みと書き込みが可能になります (ただし、これらは組込み型です)。この仕組みをユーザーが意識することはありません。

ORADATA 実装および SQLData 実装でのデータの読み込み / 書き込みの仕組みは、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。

ORADATA 実装のその他の使用例

ここまでは、カスタム Java クラスについて、次のような使用例を取り上げてきました。

- SQL オブジェクトのラッパー: このカスタム・オブジェクト・クラスは、`oracle.sql.STRUCT` インスタンスに対して使用します。
- SQL 参照のラッパー: このカスタム参照クラスは、`oracle.sql.REF` インスタンスに対して使用します。
- SQL コレクションのラッパー: このカスタム・コレクション・クラスは、`oracle.sql.ARRAY` インスタンスに対して使用します。

これ以外の `oracle.sql.*` 型をラッピングする場合でも、カスタム Java クラスを用意すると独自の変換または処理を行うのに便利です。このためには、`ORADData` を実装したクラスを使用してください（`SQLData` は不使用）。次に例を示します。

- データの暗号化と復号化、またはデータの妥当性チェックを行う場合
- 読み込み後または書き込み前の値をロギングする場合
- `CHAR` 列（URL 情報を格納した文字フィールドなど）をより小さいコンポーネントに解析する場合
- 文字列を数値定数にマッピングする場合
- データをより適切な Java 形式にマッピングする場合（`DATE` フィールドを `java.util.Date` 形式にマッピングする場合など）
- データ表現をカスタマイズする場合（選択した表のフィート単位データをメートル単位で表現する場合など）
- Java オブジェクトをシリアル化およびシリアル化解除する場合（`RAW` フィールドなどに対して）

最後に示した用途例の詳細は、6-65 ページの「[シリアル化された Java オブジェクト](#)」を参照してください。

ここからは、クラス `BetterDate` の例を取り上げています。このクラスの場合は、`ORADData` を実装し、これを `java.sql.Date` のかわりに使用して日付を表現しています。

注意： このような機能は、`SQLData` インタフェースを使用しても実現されません。`SQLData` 実装でラッピング対象となるのは構造化オブジェクト型のみに限られているためです。

ORADData の一般的な使用方法 : `BetterDate.java` ここでは、`ORADData` インタフェースを実装したクラスを使用して、Java の日付をカスタマイズする方法を示します。

注意： このコード例は `main()` メソッドが含まれていないので、完全なアプリケーションとしては機能しません。

```
import java.util.Date;
import oracle.sql.ORAData;
import oracle.sql.DATE;
import oracle.sql.ORADataFactory;
import oracle.jdbc.OracleTypes;

// a Date class customized for user's preferences:
//     - months are numbers 1..12, not 0..11
//     - years are referred to via four-digit numbers, not two.

public class BetterDate extends java.util.Date
    implements ORAData, ORADataFactory {
    public static final int _SQL_TYPECODE = OracleTypes.DATE;

    String[] monthNames={"JAN", "FEB", "MAR", "APR", "MAY", "JUN",
        "JUL", "AUG", "SEP", "OCT", "NOV", "DEC"};
    String[] toDigit={"0", "1", "2", "3", "4", "5", "6", "7", "8", "9"};

    static final BetterDate _BetterDateFactory = new BetterDate();

    public static ORADataFactory getORADataFactory() { return _BetterDateFactory;}

    // the current time...
    public BetterDate() {
        super();
    }

    public oracle.sql.Datum toDatum(java.sql.Connection conn) {
        return new DATE(toSQLDate());
    }

    public oracle.sql.ORAData create(oracle.sql.Datum dat, int intx) {
        if (dat==null) return null;
        DATE DAT = ((DATE)dat);
        java.sql.Date jsd = DAT.dateValue();
        return new BetterDate(jsd);
    }

    public java.sql.Date toSQLDate() {
        java.sql.Date retval;
        retval = new java.sql.Date(this.getYear()-1900, this.getMonth()-1,
            this.getDate());
        return retval;
    }

    public BetterDate(java.sql.Date d) {
        this(d.getYear()+1900, d.getMonth()+1, d.getDate());
    }
}
```

```

private static int [] deconstructString(String s) {
    int [] retval = new int[3];
    int y,m,d; char temp; int offset;
    StringBuffer sb = new StringBuffer(s);
    temp=sb.charAt(1);
    // figure the day of month
    if (temp < '0' || temp > '9') {
        m = sb.charAt(0)-'0';
        offset=2;
    } else {
        m = (sb.charAt(0)-'0')*10 + (temp-'0');
        offset=3;
    }

    // figure the month
    temp = sb.charAt(offset+1);
    if (temp < '0' || temp > '9') {
        d = sb.charAt(offset)-'0';
        offset+=2;
    } else {
        d = (sb.charAt(offset)-'0')*10 + (temp-'0');
        offset+=3;
    }

    // figure the year, which is either in the format "yy" or "yyyy"
    // (the former assumes the current century)
    if (sb.length() <= (offset+2)) {
        y = (((new BetterDate()).getYear())/100)*100 +
            (sb.charAt(offset)- '0') * 10 +
            (sb.charAt(offset+1)- '0');
    } else {
        y = (sb.charAt(offset)- '0') * 1000 +
            (sb.charAt(offset+1)- '0') * 100 +
            (sb.charAt(offset+2)- '0') * 10 +
            (sb.charAt(offset+3)- '0');
    }
    retval[0]=y;
    retval[1]=m;
    retval[2]=d;
    //    System.out.println("Constructing date from string as: "+d+"/"+m+"/"+y);
    return retval;
}

private BetterDate(int [] stuff) {
    this(stuff[0], stuff[1], stuff[2]);
}

// takes a string in the format: "mm-dd-yyyy" or "mm/dd/yyyy" or
// "mm-dd-yy" or "mm/dd/yy" (which assumes the current century)
public BetterDate(String s) {

```

```
        this(BetterDate.deconstructString(s));
    }

    // years are as '1990', months from 1..12 (unlike java.util.Date!), date
    // as '1' to '31'
    public BetterDate(int year, int months, int date) {
        super(year-1900,months-1,date);
    }
    // returns "Date: dd-mon-yyyy"
    public String toString() {
        int yr = getYear();
        return getDate()+"-"+monthNames[getMonth()-1]+"-"+
            toDigit[(yr/1000)%10] +
            toDigit[(yr/100)%10] +
            toDigit[(yr/10)%10] +
            toDigit[yr%10];
    }
    // return "Date: " + getDate() + "-"+getMonth()+"-"+(getYear()%100);
    }
    public BetterDate addDays(int i) {
        if (i==0) return this;
        return new BetterDate(getYear(), getMonth(), getDate()+i);
    }
    public BetterDate addMonths(int i) {
        if (i==0) return this;
        int yr=getYear();
        int mon=getMonth()+i;
        int dat=getDate();
        while(mon<1) {
            --yr;mon+=12;
        }
        return new BetterDate(yr, mon,dat);
    }
    // returns year as in 1996, 2007
    public int getYear() {
        return super.getYear()+1900;
    }
    // returns month as 1..12
    public int getMonth() {
        return super.getMonth()+1;
    }
    public boolean equals(BetterDate sd) {
        return (sd.getDate() == this.getDate() &&
            sd.getMonth() == this.getMonth() &&
            sd.getYear() == this.getYear());
    }
    // subtract the two dates; return the answer in whole years
    // uses the average length of a year, which is 365 days plus
```

```

// a leap year every 4, except 100, except 400 years =
// = 365 97/400 = 365.2425 days = 31,556,952 seconds
public double minusInYears(BetterDate sd) {
    // the year (as defined above) in milliseconds
    long yearInMillis = 31556952L;
    long diff = myUTC()-sd.myUTC();
    return (((double)diff/(double)yearInMillis)/1000.0);
}
public long myUTC() {
    return Date.UTC(getYear()-1900, getMonth()-1, getDate(),0,0,0);
}

// returns <0 if this is earlier than sd
// returns = if this == sd
// else returns >0
public int compare(BetterDate sd) {
    if (getYear()!=sd.getYear()) {return getYear()-sd.getYear();}
    if (getMonth()!=sd.getMonth()) {return getMonth()-sd.getMonth();}
    return getDate()-sd.getDate();
}
}

```

ユーザー定義データ型

ここでは、Oracle9i のユーザー定義オブジェクト型とユーザー定義コレクション型の作成例および使用例を示します。ここで使用している SQL コマンドの詳細は、『Oracle9i SQL リファレンス』を参照してください。

オブジェクト型の作成

オブジェクト型を作成する Oracle SQL コマンドは、次の形式で指定します。

```

CREATE TYPE typename AS OBJECT
(
    attrname1    datatype1,
    attrname2    datatype2,
    ...          ...
    attrnameN    datatypeN
);

```

typename で、オブジェクト型の名前を指定します。*attrname1* ～ *attrnameN* で、属性名を指定します。*datatype1* ～ *datatypeN* で、属性のデータ型を指定します。

これより、Oracle9i におけるユーザー定義オブジェクト型の作成例を示します。

次の項目を作成する SQL スクリプトを示します。

- 2つのオブジェクト型、PERSON と ADDRESS。
- PERSON オブジェクト用の型付けされた表。
- EMPLOYEES 表。ADDRESS 列が1列と PERSON 参照列が2列あります。

次にスクリプトを示します。

```
/** Using user-defined types (UDTs) in SQLJ */
/
/** Create ADDRESS UDT */
CREATE TYPE ADDRESS AS OBJECT
(
    street      VARCHAR(60),
    city        VARCHAR(30),
    state       CHAR(2),
    zip_code    CHAR(5)
)
/
/** Create PERSON UDT containing an embedded ADDRESS UDT */
CREATE TYPE PERSON AS OBJECT
(
    name      VARCHAR(30),
    ssn       NUMBER,
    addr      ADDRESS
)
/
/** Create a typed table for PERSON objects */
CREATE TABLE persons OF PERSON
/
/** Create a relational table with two columns that are REFs
    to PERSON objects, as well as a column which is an Address ADT. */
CREATE TABLE employees
(
    empnumber      INTEGER PRIMARY KEY,
    person_data    REF PERSON,
    manager        REF PERSON,
    office_addr    ADDRESS,
    salary         NUMBER
)
/
/** Insert some data--2 objects into the persons typed table */
INSERT INTO persons VALUES (
    PERSON('Wolfgang Amadeus Mozart', 123456,
        ADDRESS('Am Berg 100', 'Salzburg', 'AT', '10424')))
```



```

/
INSERT INTO persons VALUES (
    PERSON('Ludwig van Beethoven', 234567,
        ADDRESS('Rheinallee', 'Bonn', 'DE', '69234'))
/
/** Put a row in the employees table */
INSERT INTO employees (empnumber, office_addr, salary) VALUES (
    1001,
    ADDRESS('500 Oracle Parkway', 'Redwood Shores', 'CA', '94065'),
    50000)
/
/** Set the manager and PERSON REFs for the employee */
UPDATE employees
    SET manager =
        (SELECT REF(p) FROM persons p WHERE p.name = 'Wolfgang Amadeus Mozart')
/
UPDATE employees
    SET person_data =
        (SELECT REF(p) FROM persons p WHERE p.name = 'Ludwig van Beethoven')

```

注意： Oracle SQL では、特にユーザー定義型の表にアクセスする場合に、表の別名（この例の `p` など）の使用を習慣付けることをお勧めします。オブジェクトの属性にアクセスする場合は、この構文を使用する必要があります。別名を使用する必要のない場合でも、別名の使用によって明確化できます。表の別名の詳細は、『Oracle9i SQL リファレンス』を参照してください。

コレクション型の作成

コレクションは、次の2つの種類があります。

- 可変長配列 (VARRAY)
- ネストした表 (NESTED TABLE)

VARRAY 型を作成する Oracle SQL コマンドは、次の形式で指定します。

```
CREATE TYPE typename IS VARRAY(n) OF datatype;
```

`typename` で VARRAY 型の名前を、`n` で配列内の最大要素数を、また `datatype` で配列要素のデータ型を指定します。次にその例を示します。

```
CREATE TYPE myvarr IS VARRAY(10) OF INTEGER;
```

NESTED TABLE 型を作成する Oracle SQL コマンドは、次の形式で指定します。

```
CREATE TYPE typename AS TABLE OF datatype;
```

typename でネストした表の型の名前を、*datatype* で表要素のデータ型を指定します。データ型は標準データ型またはユーザー定義型を指定できます。ネストした表は1列のみに限られています。ただし、1つの列型を、複数の属性を持つ複合オブジェクトにすることも可能です。ネストした表には、データベース表と同様に、行数制限がありません。次にその例を示します。

```
CREATE TYPE person_array AS TABLE OF person;
```

この CREATE TYPE コマンドを指定すると、PERSON オブジェクトから構成された各行に NESTED TABLE 型が作成されます。

次に、Oracle9i におけるユーザー定義のコレクション型（およびオブジェクト型）の作成例を示します。

次の項目を作成および設定する SQL スクリプトを示します。

- 2つのオブジェクト型、PARTICIPANT_T と MODULE_T
- コレクション型、MODULE_T オブジェクトのネストした表である MODULETBL_T
- PARTICIPANT_T 参照列とネストした表の MODULETBL_T 列を含む PROJECTS 表
- コレクション型、VARCHAR2(30) の VARRAY である PHONE_ARRAY
- PERSON と ADDRESS オブジェクト（6-21 ページの「[オブジェクト型の作成](#)」に同じ定義が示されています。）
- PHONE_ARRAY 列を含む EMPLOYEES 表

次にスクリプトを示します。

```
Rem This is a SQL*Plus script used to create schema to demonstrate collection
Rem manipulation in SQLJ
```

```
CREATE TYPE PARTICIPANT_T AS OBJECT (
    empno    NUMBER(4),
    ename     VARCHAR2(20),
    job       VARCHAR2(12),
    mgr       NUMBER(4),
    hiredate  DATE,
    sal       NUMBER(7,2),
    deptno    NUMBER(2))
/
show errors
CREATE TYPE MODULE_T AS OBJECT (
    module_id    NUMBER(4),
    module_name  VARCHAR2(20),
    module_owner REF PARTICIPANT_T,
    module_start_date DATE,
    module_duration NUMBER )
```

```
/
show errors
create TYPE MODULETBL_T AS TABLE OF MODULE_T;
/
show errors
CREATE TABLE projects (
  id NUMBER(4),
  name VARCHAR(30),
  owner REF PARTICIPANT_T,
  start_date DATE,
  duration NUMBER(3),
  modules MODULETBL_T ) NESTED TABLE modules STORE AS modules_tab;

show errors
CREATE TYPE PHONE_ARRAY IS VARRAY (10) OF varchar2(30)
/

/** Create ADDRESS UDT */
CREATE TYPE ADDRESS AS OBJECT
(
  street      VARCHAR(60),
  city        VARCHAR(30),
  state       CHAR(2),
  zip_code    CHAR(5)
)
/
/** Create PERSON UDT containing an embedded ADDRESS UDT */
CREATE TYPE PERSON AS OBJECT
(
  name        VARCHAR(30),
  ssn         NUMBER,
  addr        ADDRESS
)
/
CREATE TABLE employees
( empnumber      INTEGER PRIMARY KEY,
  person_data    REF person,
  manager        REF person,
  office_addr    address,
  salary         NUMBER,
  phone_nums     phone_array
)
/
```

JPublisher とカスタム Java クラスの作成

Oracle では、Oracle のオブジェクト型、参照型およびコレクション型を強い型指定のパラダイムの Java クラスに柔軟にマッピングできます。開発者は、これらのカスタム Java クラスを次の方法で作成できます。

- Oracle の JPublisher でカスタム Java クラスを自動生成し、このクラスを変更せずにそのまま使用する方法
- JPublisher でカスタム Java クラスと対応するサブクラス（後でユーザーが機能を変更できる）を自動生成する方法
- JPublisher を使用せずに、手動でカスタム Java クラスをコード化する方法（6-10 ページの「[カスタム Java クラスの要件](#)」の要件を満たすクラスの場合のみ）

カスタム Java クラスは手動でもコード化できますが、JPublisher で生成されたクラスを直接使用すること、または JPublisher で生成されたサブクラスを変更することをお勧めします。

JPublisher では、Oracle `oracle.sql.ORAData` または標準 `java.sql.SQLData` インタフェースを実装したカスタム・オブジェクト・クラスを生成できます。このうちの `ORAData` 実装を JPublisher で生成した場合は、カスタム参照クラスも一緒に生成されます。現行より古い JDBC バージョンとの互換性のために、JPublisher で、現在は使用されていない `oracle.sql.CustomDatum` インタフェースを実装するクラスを生成することもできます。

`SQLData` インタフェースは、カスタム参照クラスやカスタム・コレクション・クラスには実装できません。これから開発するコードを移植可能にする方法は、弱い型指定の標準 `java.sql.Ref` オブジェクトと `java.sql.Array` オブジェクトを、それぞれ参照とコレクションにマッピングする方法のみです。

このマニュアルの内容は、JPublisher ユーティリティに関する必要最小限の概説です。詳細は、『Oracle9i JPublisher ユーザーズ・ガイド』を参照してください。

`ORAData` および `SQLData` インタフェースの詳細と `ORAData` インタフェースのメリットについては、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。

JPublisher で生成される内容

JPublisher を使用してカスタム Java クラスを生成する場合は、`ORAData`（カスタム・オブジェクト・クラス、カスタム参照クラスまたはカスタム・コレクション・クラス用）または `SQLData`（カスタム・オブジェクト・クラス専用）実装を使用できます。`ORAData` 実装には、`ORADataFactory` インタフェース（カスタム Java クラスのインスタンス生成に必要なインタフェース）も実装できます。

JPublisher の `-usertypes` オプションの設定内容に応じて、実装の対象が決まります。`-usertypes=oracle` 設定では `ORAData` 実装が指定され、`-usertypes=jdbc` 設定では `SQLData` 実装が指定されます。

ORADATA 実装

ユーザー定義オブジェクト型を処理対象として JPublisher を実行し、カスタム・オブジェクト・クラスに対して ORADATA を実装するようにデフォルトの `-usertypes=oracle` 設定を使用すると、次のようなクラスが自動的に生成されます。

- カスタム・オブジェクト・クラス（通常は `.sqlj` ソース・ファイル内）。Oracle のオブジェクト型に対応する型定義です。

このクラスには、各属性の取得および設定用のメソッドがあります。属性が `foo` の場合、このメソッド名の形式は `getFoo()` および `setFoo()` です。

JPublisher のデフォルトでは、サーバー側 Oracle オブジェクトのメソッドを起動するラッパー・メソッドもクラスに生成されます。ただし、`-methods=false` を設定すると、このラッパー・メソッドを生成しないように指定できます。この場合、JPublisher は、ラッパー・メソッドを生成せずに、カスタム・オブジェクト用に `.sqlj` ファイルではなく、`.java` ファイルを作成します。`-methods` オプションの詳細は、後で詳しく説明します。

- Oracle オブジェクト型へのオブジェクト参照にアクセスするためのカスタム参照クラス。

このカスタム参照クラスの `getValue()` メソッドは、カスタム・オブジェクト・クラスのインスタンスを戻り値としており、もう 1 つの `setValue()` メソッドは、カスタム・オブジェクト・クラスのインスタンスを入力として取り、データベース内のオブジェクトの値を更新します。

SQL オブジェクト型で参照を使用しているかどうかにかかわらず、強い型指定の参照クラスが常に生成されます。

弱い型指定ではなく、強い型指定の参照を使用するメリットについては、6-28 ページの「[ORADATA 実装のための強い型指定のオブジェクト参照](#)」を参照してください。

- 最上位オブジェクトのオブジェクト属性またはコレクション属性にアクセスするためのカスタム・クラス。

Java では、最上位クラスのインスタンスを作成するとき、このクラスを使用して属性を作成します。

ユーザー定義コレクション型を処理対象として JPublisher を実行し、ORADATA を実装するように指定すると、次のようなクラスが自動的に生成されます。

- カスタム・コレクション・クラス。指定した Oracle コレクション型に対応する型定義として機能します。

このカスタム・コレクション・クラスのオーバーロード・メソッド `getArray()` および `setArray()` は、コレクション単位で取得や更新を行うメソッドであり、`getElement()` メソッドおよび `setElement()` メソッドは、コレクションの要素単位で取得や更新を行うメソッドです。このクラスには、これ以外のユーティリティ・メソッドもあります。

- 要素のカスタム・オブジェクト・クラス（コレクションの要素がオブジェクトの場合）。

Java では、コレクションのインスタンスを作成するときに、このクラスを使用してオブジェクト要素を作成します。

JPublisher で生成したこの 2 種類のカスタム Java クラスには、ORADATA インタフェース、ORADATAFactory インタフェースおよび `getORADATAFactory()` メソッドが実装されます。

注意：

- ORADATA 実装を指定した場合、生成されたクラスは Oracle 固有の機能を使用するため、移植できなくなります。
 - JPublisher では、CustomDatum インタフェースの実装を `-compatible` オプションによってサポートしています。このインタフェースは ORADATA に引き継がれ、Oracle9i では使用されていません。詳細は、6-30 ページの「[生成されたクラスへの実装の選択](#)」を参照してください。
-

ORADATA 実装のための強い型指定のオブジェクト参照

Oracle の ORADATA を実装する場合、JPublisher では、弱い型指定の `oracle.sql.REF` クラスを使用するのではなく、常に強い型指定のオブジェクト参照クラスを生成します。これによって、オブジェクト参照が強い型指定されている場合、型の安全性が向上し、SQL の動作がミラー化されます。強い型指定のクラス（PERSON オブジェクトを参照する `PersonRef` などの名前とともに使用）は、基本的には REF クラス用のラッパーです。

これらの強い型指定の REF ラッパーには、参照される SQL オブジェクトのインスタンスを該当する Java クラスのインスタンスの形式で生成する `getValue()` メソッドがあります。（継承する場合は、該当する Java クラスのサブクラスのインスタンスとして生成します。）たとえば、PERSON SQL オブジェクト型と `Person` Java クラスがある場合、`PersonRef` Java クラスもあります。`PersonRef` クラスの `getValue()` メソッドによって、データベース内に PERSON オブジェクト用のデータを含む `Person` インスタンスが戻されます。

SQL オブジェクト型にオブジェクト参照の属性がある場合、そのオブジェクト型に対応する Java クラスには、該当の参照型に対応する Java クラスのインスタンスの属性があります。たとえば、MANAGER REF 属性の PERSON オブジェクトがある場合、対応する `Person` Java クラスは `ManagerRef` 属性を持ちます。

SQLData 実装

ユーザー定義型のオブジェクト型を処理対象として JPublisher を実行し、
-usertypes=jdbc 設定でカスタム・オブジェクト・クラスに対して SQLData 実装を選択する場合、Oracle オブジェクト型に対応する型定義として機能するカスタム・オブジェクト・クラスが生成されます。このカスタム・コレクション・クラスに属するものを次に示します。

- 各属性を取得するメソッドと設定するメソッド
- readSQL() メソッドと writeSQL() メソッドおよび標準 SQLData インタフェースの実装
- サーバー側で実行される Oracle オブジェクト・メソッドを起動するためのラッパー・メソッド (JPublisher 実行時の設定が -methods=false 以外の場合)

ただし、SQLData インタフェースはオブジェクトのみを対象とし、参照やコレクションには使用できないため、Oracle オブジェクト型を参照するためのカスタム参照クラスについては、JPublisher では生成の対象とはしていません。弱い型指定の標準 `java.sql.Ref` インスタンスまたは `oracle.sql.REF` インスタンスを使用する場合、移植性を気にする必要はありません。REF インスタンスは、カスタム参照クラスのインスタンスと同様に、Oracle 拡張メソッド `getValue()` および `setValue()` を使用して、参照するオブジェクトのインスタンスに対して読み込みと書き込みを行います。標準 Ref インスタンスには、こうした読み込みと書き込みの機能はありません。

カスタム・コレクション・クラスに対しては SQLData を実装できないため、弱い型指定の標準 `java.sql.Array` インスタンスを使用するか、または移植性がなくてなければ `oracle.sql.ARRAY` インスタンスを使用してください。Array インスタンスと ARRAY インスタンスは、カスタム・コレクション・クラスのインスタンスと同様に、`getArray()` 機能を備えているため、コレクション単位での読み込みはできますが、要素レベルでのアクセスや書き込みはできません。このアクセスや書き込みには、カスタム・コレクション・クラス `getElement()` メソッドと `setElement()` メソッドを使用する必要があります。

注意： JDBC の仕様では、SQLData インタフェースが移植可能であると定義されています。ただし、JPublisher で生成される SQLData 実装を移植可能とするには、Oracle 固有の機能と Oracle 型のマッピングの使用を避ける必要があります（これらを使用した場合は Oracle 固有の `oracle.sql.*` クラスが使用されるためです）。

カスタム Java クラスの生成

ここでは、JPublisher のコマンドラインの主要機能について説明します。主要機能として、Java にマッピングするユーザー定義型を指定する機能、およびオブジェクト・クラス名、コレクション・クラス名、属性型のマッピングおよびラッパー・メソッドを指定する機能があります。次に、要点をまとめます。

- JPublisher の `-usertypes` オプションを介して、使用する実装（ORADData または SQLData）を指定します。
- Java にマッピングするユーザー定義型の指定。使用する JPublisher のカスタム・オブジェクト・クラスとカスタム・コレクション・クラスの名前を指定することも、デフォルトの名前を使用することも可能です。必要に応じて JPublisher の `-sql`、`-user` および `-case` オプションを使用します。
- オプションとして、`-numbertypes`、`-builtintypes` および `-lobtypes` などの JPublisher の `-XXXtypes` オプションを介して属性型マッピングを指定します。
- ラッパー・メソッド（Oracle オブジェクト・メソッドなど）の作成要否も指定できます。デフォルトでは作成されますが、JPublisher の `-methods` フラグでの指定も可能です。

注意： これ以降は、カスタム参照クラスやカスタム・コレクション・クラスを簡潔に説明するために、ORADData 実装についてのみ記述します。

生成されたクラスへの実装の選択

JPublisher を実行する前に、Oracle の ORADData インタフェースと標準 SQLData インタフェースのどちらを生成されたクラスに実装するかを検討します。SQLData を使用すると、コードの移植性が高くなりますが、ORADData を使用すると、型マップが不要になるなどいくつかのメリットがあります。

それぞれを使用する場合の実装の詳細は、6-26 ページの「[JPublisher で生成される内容](#)」に説明してあります。

次の点に注意してください。

- カスタム・コレクション・クラスには、ORADData 実装を使用する必要があります。SQLData インタフェースではコレクション（配列）をサポートしていません。
- 強い型指定の参照クラスでは、常に ORADData カスタム・オブジェクト・クラスの実装が生成され、SQLData カスタム・オブジェクト・クラスの実装は生成されません。SQLData インタフェースは、強い型指定のオブジェクト参照をサポートしていません。かわりに、弱い型指定の `java.sql.Ref` 型または `oracle.sql.REF` 型を使用してください。

ORADData および SQLData インタフェースの詳細と ORADData インタフェースのメリットについては、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。

JPublisher -usertypes オプションを使用して、クラスに実装するインタフェースを指定します。-usertypes=oracle 設定（デフォルト）では、ORADATA インタフェースが指定され、-usertypes=jdbc 設定では、SQLData インタフェースが指定されます。

注意： CustomDatum インタフェースは、ORADATA に引き継がれ、Oracle9i では使用されていませんが、このインタフェースの実装が必要な場合は、JPublisher の customdatum の -compatible 設定を使用して実装が可能です。これに -usertypes=oracle 設定を組み合わせることによって、生成されたクラスに CustomDatum インタフェースが実装されます。デフォルトは -compatible=oradata です。

-compatible=8i または -compatible=both8i を設定しても、JPublisher で CustomDatum が使用され、同様に Oracle8i の JPublisher のバージョンと互換性のあるコード生成が実行されます。詳細は、『Oracle9i JPublisher ユーザーズ・ガイド』を参照してください。

次に示す JPublisher のコマンドラインは、それぞれ ORADATA、CustomDatum および SQLData を実装する例です（% はシステム・プロンプトの例です）。

```
% jpub -usertypes=oracle ... <other option settings>
```

```
% jpub -usertypes=oracle -compatible=customdatum ... <other option settings>
```

```
% jpub -usertypes=jdbc ... <other option settings>
```

-usertypes=jdbc の場合、-compatible=customdatum または -compatible=oradata 設定は JPublisher で無効になります。

Java にマッピングするユーザー定義型の指定

JPublisher でカスタム Java クラスを作成する場合は、Java にマッピングするユーザー定義 SQL 型を -sql オプションで指定します。カスタム・オブジェクト・クラスとカスタム・コレクション・クラスの名前を指定することも、デフォルト名を使用することも可能です。

最上位カスタム・クラス（-sql オプションで名前を指定したユーザー定義型に対応するクラス）のデフォルト名には、JPublisher コマンドラインで入力したユーザー定義型の名前が使用されます。データベース内の SQL 名はデフォルトで大 / 小文字が区別されるので、SQL 名を大文字で入力すると、Java 規則に従ってクラス名が大文字になります。たとえば、employee オブジェクトのカスタム・クラスを作成するには、次のように JPublisher を実行します。

```
% jpub -sql=Employee ...
```

employee オブジェクトの属性である home_address オブジェクトなど、他のクラスのデフォルト名は、JPublisher の `-case` オプションで指定します。`-case` オプションを指定しないと、mixed に設定されます。つまり、カスタム・クラスのデフォルト名は、ユーザー定義型名の頭文字と以降の各ワードの頭文字が大文字になります。JPublisher では、アンダースコア (`_`)、ドル記号 (`$`) および Java の識別子以外の文字がワード間のセパレータと見なされ、処理時に破棄されます。

たとえば、Oracle のオブジェクト型 home_address の場合は、クラス HomeAddress が HomeAddress.sqlj または .java のソース・ファイルに生成されます。

重要： JPublisher コマンドラインでは、大 / 小文字を区別していない SQL 名のみサポートされます。大 / 小文字を区別する方法で（引用符内で）ユーザー定義型を SQL に定義した場合、コマンドラインや引用符ではなく JPublisher の INPUT ファイルに名前を指定する必要があります。6-39 ページの「[JPublisher の INPUT ファイルの使用方法](#)」を参照してください。

注意： 旧バージョンの JPublisher との下位互換用として、`-sql` のかわりに `-types` オプションを従来どおり使用できます。

JPublisher のコマンドラインでは、`-sql` オプションに次の構文を使用します。単一のオプション設定で複数のアクションを指定できます。

```
-sql=udt1<:mapclass1><,udt2<:mapclass2>>,...,<udtN<:mapclassN>> ...
```

データベース・スキーマを指定するには、`-user` オプションを使用します。次に例を示します。

```
% jpub -sql=Myobj,mycoll:MyCollClass -user=scott/tiger
```

（カンマの前後にはスペースを挿入できません。）

このように入力すると、Oracle のオブジェクトは MYOBJ という名前になり、Myobj クラスを定義したソース・ファイル Myobj.sqlj が生成されます。また、Oracle コレクション MYCOLL に対して、MyCollClass クラスを定義したソース・ファイル MyCollClass.java が生成されます。

`-sql` にはスキーマ名（scott スキーマなど）も指定できます。

```
% jpub -sql=scott.Myobj,scott.mycoll:MyCollClass -user=scott/tiger
```

カスタム参照クラスの名前は指定できません。JPublisher では、カスタム・オブジェクト・クラス名の後ろに **Ref** が追加されて、自動的に生成されます (ORADATA 実装の場合のみ)。たとえば、カスタム・オブジェクト・クラス **Myobj** を定義する Java ソース・ファイル **Myobj.sqlj** を JPublisher で作成すると、**MyobjRef** カスタム参照クラスを定義する Java ソース・ファイル **MyobjRef.java** も生成されます。

注意： スキーマ (例のように **scott** など) を指定しても、カスタム Java クラスの名前には取り込まれません。

6-21 ページの「**ユーザー定義データ型**」で定義したオブジェクト型とコレクション型のカスタム Java クラスを作成するには、次のように JPublisher を実行します。

```
%jpub -user=scott/tiger -sql=Address,Person,Phone_array,Participant_t,
Module_t,Moduletbl_t
```

または、カスタム・オブジェクト・クラスとカスタム・コレクション・クラスの名前を明示的に指定します。

```
%jpub -user=scott/tiger -sql=Address,Person,phone_array:PhoneArray,
participant_t:ParticipantT,module_t:ModuleT,moduletbl_t:ModuletblT
```

(前述の 2 つの例はそれぞれ折り返されて表示されていますが、1 行で入力されたコマンドラインです。)

2 番目の例では、Java ソース・ファイル **Address.sqlj**、**AddressRef.java**、**Person.sqlj**、**PersonRef.java**、**PhoneArray.java**、**ParticipantT.sqlj**、**ParticipantTRef.java**、**ModuleT.sqlj**、**ModuleTRef.java** および **ModuletblT.java** が生成されます。ソース・ファイルの例は、6-44 ページの「**JPublisher のカスタム Java クラスの例**」を参照してください。

カスタム Java クラスの設定方法を特定するために、JPublisher は指定されたスキーマ (この例では **scott/tiger**) に接続し、指定されたオブジェクト型の属性または指定されたコレクション型の要素を調べます。

注意： Oracle9i リリース 2 (9.2) では、単一の **-sql** 設定に複数のマッピングを指定するかわりに、同一のコマンドラインに複数の **-sql** オプションを使用できます。複数の **-sql** オプションの効果は累積されます。

JPublisher で生成するメソッドおよび属性のデフォルト名の大 / 小文字の使用方法を変更するには、**-case** オプションを使用します。オブジェクトまたはコレクションが属性の場合、つまり下位カスタム Java クラスの場合も、このオプションを使用します。次の 4 通りの設定があります。

- `-case=mixed` (デフォルト) : 大文字になる文字は、クラス名を構成する各ワードの頭文字、属性名を構成する各ワードの頭文字およびメソッド名の 2 番目以降のワードの頭文字です。これ以外の文字はすべて小文字になります。JPublisher では、アンダースコア (`_`)、ドル記号 (`$`) および Java の識別子以外の文字がワード間のセパレータと見なされ、処理時に破棄されます。
- `-case=same`: 大 / 小文字区別は、データベース内で表現された場合と同じになります。アンダースコアとドル記号は保持されます。Java の識別子以外の文字は破棄されます。
- `-case=upper`: 小文字が大文字に変換されます。アンダースコアとドル記号は保持されます。Java の識別子以外の文字は破棄されます。
- `-case=lower`: 大文字が小文字に変換されます。アンダースコアとドル記号は保持されます。Java の識別子以外の文字は破棄されます。

注意： JPublisher の実行時に Java にマッピングするユーザー定義型を指定しないと、スキーマ内のすべてのユーザー定義型が処理されます。最上位カスタム・クラスとその他のクラス（オブジェクトの属性またはコレクションの要素）のクラス名は、`-case` オプションに基づき生成されます。

型のマッピング指定

JPublisher には、ユーザー定義型およびその属性型や要素型を SQL と Java との間でマッピングする方法が数通りあります。カテゴリ別 SQL 型とマッピング・オプションの分類については、後で一覧の形式で示します。

(Oracle データ型から Java 型へのマッピング方法の概要は、5-2 ページの「[ホスト式での型サポート](#)」を参照してください。)

JPublisher の機能やオプションの詳細は、『Oracle9i JPublisher ユーザーズ・ガイド』を参照してください。

SQL 型の分類 JPublisher では SQL 型が次のようなグループに分類されています。各グループに対応する JPublisher オプションは、各項に示したとおりです。

- **数値型**: SQL 型 `NUMBER` として格納されるすべての型
JPublisher の `-numbertypes` オプションは、数値型のマッピング指定に使用します。
- **LOB 型**: SQL 型 `BLOB` および `CLOB`
JPublisher の `-lobtypes` オプションは、LOB 型のマッピング指定に使用します。
- **組込み型**: データベース内に SQL 型として格納されている型のうち、前に示した分類のどれにも該当しない型 (たとえば、`CHAR`、`VARCHAR2`、`LONG`、`RAW` など)
JPublisher の `-builtintypes` オプションは、組込み型のマッピング指定に使用します。

型のマッピング・モード JPublisher では次の型マッピング・モードを定義します。

- JDBC マッピング（設定は `jdbc`）: SQL 型とネイティブな Java 型とのマッピングに標準デフォルト・マッピング型を使用します。この設定は、`-numbertypes`、`-lobtypes` および `-builtintypes` オプションに有効です。
- Oracle マッピング（設定は `oracle`）: SQL 型へのマッピングの際に、対応する `oracle.sql` 型を使用します。この設定は、`-numbertypes`、`-lobtypes` および `-builtintypes` オプションに有効です。
- Object JDBC マッピング（設定は `objectjdbc`）: JDBC マッピングをさらに拡張したマッピングです。必要に応じて、Object JDBC マッピングでは、標準 `java.lang` パッケージの数値オブジェクト型を使用します（具体的には `java.lang.Integer`、`Float` または `Double` など）。この数値オブジェクト型は、Java の基本型（具体的には `int`、`float` または `double` など）のかわりに使用されます。`java.lang` 型では NULL 値が許容されていますが、基本型では許容されていません。この設定は、`-numbertypes` オプションにのみ有効です。
- BigDecimal マッピング（設定は `bigdecimal`）: 数値属性へのマッピングの際に `java.math.BigDecimal` を使用します。桁数の多い数値を使用しても `oracle.sql.NUMBER` 型は使用しない場合に適したマッピング方法です。この設定は、`-numbertypes` オプションにのみ有効です。

注意: BigDecimal マッピングを使用すると、パフォーマンスが著しく劣化する場合があります。

オブジェクト属性やコレクション要素に使用可能な型のマッピング・オプションについては、次項で説明します。

属性や要素から Java へのマッピング SQL オブジェクト型や要素型の属性型に対してマッピング指定を省略すると、JPublisher では次のデフォルトが使用されます。

- 数値型のデフォルトは、Object JDBC マッピングです。
- LOB 型のデフォルトは、Oracle マッピングです。
- 組込みタイプ型のデフォルトは、JDBC マッピングです。

これ以外に代替のマッピングを使用する場合は、`-numbertypes`、`-lobtypes` および `-builtintypes` オプションを使用します。どのオプションを使用するかは、処理対象とする属性型およびマッピングに応じて決めてください。

属性型自体が SQL オブジェクト型である場合には、`-usertypes` 設定値に基づいてマッピングされます。

重要： カスタム・オブジェクト・クラスに対して `SQLData` が実装されるように指定し、コードが移植可能であるようにする場合は、属性型に対して移植可能なマッピングを使用する必要があります。数値型または組み込み型をマッピングの対象とする場合はデフォルトで移植可能になりますが、LOB 型の場合は `-lobtypes=jdbc` を指定する必要があります。

SQL 型の分類とマッピング設定のまとめ 表 6-1 に、SQL 型、マッピング設定およびデフォルト設定に関して JPublisher での分類をカテゴリ別にまとめます。

表 6-1 JPublisher での SQL 型の分類、サポートされている設定およびデフォルト

SQL 型の分類	JPublisher マッピング・オプション	マッピング設定	デフォルト
UDT 型	-usertypes	oracle、jdbc	oracle
数値型	-numbertypes	oracle、jdbc、objectjdbc、bigdecimal	objectjdbc
LOB 型	-lobtypes	oracle、jdbc	oracle
組み込み型	-builtinotypes	oracle、jdbc	jdbc

注意： 以前のリリースで使用された `-mapping` オプションは、現在の JPublisher では使用されていませんが、サポートはされています。JPublisher の `-mapping` オプション設定値を新しいマッピング・オプション設定値に変換する方法の詳細は、『Oracle9i JPublisher ユーザーズ・ガイド』を参照してください。

ラッパー・メソッドの生成

Oracle オブジェクトを Java にマッピングするためのカスタム・オブジェクト・クラスを作成する際に、Oracle オブジェクトのメソッド（メンバー・ファンクション）で使用する Java ラッパーについて作成要否を指定するには、JPublisher で `-methods` オプションを使用します。デフォルトの `-methods=true` 設定では、ラッパーが生成されます。また、基となる SQL オブジェクトに実際にメソッドがないかぎり、JPublisher では、カスタム・オブジェクト・クラス用に `.sqlj` ファイル（`.java` ファイルではなく）が生成されます。

JPublisher でラッパー・メソッドを生成すると、元のオブジェクト・メソッドが `static` でも、必ずインスタンス・メソッドになります。詳細は、6-9 ページの「[オブジェクト・メソッドでのカスタム Java クラスのサポート](#)」を参照してください。

次に、`-methods` オプションの設定例を示します。

```
% jpub -sql=Myobj,mycoll:MyCollClass -user=scott/tiger -methods=true
```

この例では、デフォルトのネーミング・メソッドが使用されます。Java のメソッド名は、最初の文字が小文字になる他は、カスタム Java クラス名と同じ方法でネーミングされます (6-31 ページの「[Java にマッピングするユーザー定義型の指定](#)」を参照してください)。たとえば、オブジェクトのメソッド名 CALC_SAL は、デフォルトで Java のラッパー・メソッド名 calcSal() になります。

かわりに Java のメソッド名を指定することも可能ですが、この場合は、JPublisher の INPUT ファイルを使用する必要があります。詳細は、6-42 ページの「[カスタム Java クラスの作成およびメンバー名の指定](#)」を参照してください。

注意： -methods オプションには、他の用途もあります。たとえば、パッケージのラッパー・クラスやパッケージのメソッドのラッパー・メソッドの生成時に使用できます。この内容はこのマニュアルの対象外です。詳細は、『Oracle9i JPublisher ユーザーズ・ガイド』を参照してください。

オーバーロード・メソッドについて JPublisher の実行対象の Oracle オブジェクトにオーバーロード・メソッドがあり、複数のシグネチャが同一の Java シグネチャに対応する場合は、シグネチャごとに一意なメソッド名が生成されます。つまり、ファンクション名の後ろに `_n` が付けられます (`n` は番号)。生成されたカスタム Java クラスの 2 つのメソッドは、それぞれ一意な名前とシグネチャになります。一例として、MY_TYPE オブジェクト型の生成時に定義される SQL ファンクションについて考えます。

```
CREATE OR REPLACE TYPE my_type AS OBJECT
(
  ...

  MEMBER FUNCTION myfunc(x INTEGER)
    RETURN my_return IS
  BEGIN
    ...
  END;

  MEMBER FUNCTION myfunc(y SMALLINT)
    RETURN my_return IS
  BEGIN
    ...
  END;
  ...
);
```

このままでは、myfunc の定義は両方とも、Java 形式の名前とシグネチャになります。

```
myfunc(Integer)
```

その理由は、SQL 形式の INTEGER および SMALLINT が、Java Integer 型にマッピングされているためです。

そのかわりに、JPublisher である myfunc_1 ともう 1 つの myfunc_2 をコールします。(_n はそれぞれ一意です。簡単な例では、_1、_2 のようになりますが、各ファンクションの一意の値ではなく、任意の値でもかまいません。)

注意： オーバーロードしたラッパー・メソッドに対する JPublisher での処理は、オブジェクトやパッケージ内で生成された SQL ファンクションに対しても実行されます。ただし、最上位ファンクションに対しては（オーバーロードが許容されないため）JPublisher での処理が実行されません。

カスタム Java クラスの生成および代替クラスのマッピング

JPublisher では、カスタム Java クラスを生成できる他、オブジェクト型（またはコレクション型）のマッピング先として、生成したクラスでなく代替クラスが指定できます。

通常は、JPublisher で生成したクラスをスーパークラスにし、これを拡張して機能を追加し、オブジェクト型をサブクラスにマッピングします。たとえば、Oracle オブジェクト型 ADDRESS のカスタム Java クラスを作成し、JPublisher で生成されなかった機能を追加するとします。この場合は、JPublisher でカスタム Java クラス JAddress を生成してから、このサブクラス MyAddress を作成します。専用機能を MyAddress に追加してから、JPublisher で ADDRESS オブジェクトを、JAddress クラスではなく、MyAddress にマッピングします。JPublisher では、JAddress ではなく、MyAddress の参照クラスも生成できます。

JPublisher では、代替クラスへのマッピングを容易に行えます。-sql オプションを次の構文で指定します。

```
-sql=object_type:generated_class:map_class
```

前の例の場合は、次のように設定します。

```
-sql=ADDRESS:JAddress:MyAddress
```

この設定では、クラス JAddress がソース・ファイル JAddress.sqlj（または JAddress.java）に作成され、次の処理が行われます。

- オブジェクト型 ADDRESS のマッピング先が、JAddress クラスではなく、MyAddress クラスになります。したがって、データベースから取得したオブジェクトに ADDRESS 属性があると、Java でこの属性が MyAddress のインスタンスとして生成されます。ADDRESS オブジェクトを直接取得する場合は、MyAddress インスタンスに取り込まれます。
- JAddressRef クラスではなく、MyAddressRef クラスが MyAddressRef.java に生成されます。

- ファイルがすでに存在していないかぎり、MyAddress.sqlj（または MyAddress.java）ソース・ファイルに MyAddress クラスの初期バージョンが生成されます（存在している場合、ファイルは変更されません）。

MyAddress は JAddress にサブクラス化されます。MyAddress に対して拡張機能を実装するためには、JPublisher で生成された MyAddress ソース・ファイルを使用し、必要に応じてそのファイルを編集できます。

JPublisher 生成クラスのサブクラスを作成する方法（前例の続き）は、6-48 ページの「[JPublisher 生成クラスの拡張](#)」を参照してください。

JPublisher の INPUT ファイルおよびプロパティ・ファイル

JPublisher では、専用の INPUT ファイルおよび標準のプロパティ・ファイルを使用して、型のマッピングなどのオプション設定を指定できます。

JPublisher の INPUT ファイルの使用方法

JPublisher のコマンドライン・オプション `-input` で、型マッピングを追加するための INPUT ファイルを指定できます。

INPUT ファイルの SQL は、コマンドラインの `-sql` に相当します。AS または `GENERATE...AS` 構文は、コマンドラインのコロン区切り構文に相当します。次の構文でマッピングを指定します。SQL コマンドで一度に指定できるマッピングは 1 つのみです。

```
SQL udt1 <GENERATE GeneratedClass1> <AS MapClass1>
SQL udt2 <GENERATE GeneratedClass2> <AS MapClass2>
...
```

`GeneratedClass1` と `GeneratedClass2` が生成され、`udt1` が `MapClass1` に、`udt2` が `MapClass2` にマッピングされます。

重要： ユーザー定義型が大 / 小文字を区別する方法で（引用符内で）SQL に定義されている場合、引用符内に名前を指定する必要があります。次にその例を示します。

```
SQL "CaseSenstiveType" AS CaseSensitiveType
```

大 / 小文字を区別しないスキーマ名を指定する場合、次のようになります。

```
SQL SCOTT."CaseSensitiveType" AS CaseSensitiveType
```

大 / 小文字を区別するスキーマ名を指定する場合、次のようになります。

```
SQL "Scott"."CaseSensitiveType" AS CaseSensitiveType
```

AS 句はオプションです。

スキーマ名または型名自体の一部として、ドット (.) を使用しないでください。

INPUT ファイルの例 次の例では、JPublisher は、コマンドラインの `-user` オプションに基づき、INPUT ファイル `myinput.in` で型マッピングを調べます。

コマンドラインで次のように指定します。

```
% jpub -input=myinput.in -user=scott/tiger
```

INPUT ファイル `myinput.in` の内容は次のようになります。

```
SQL Myobj
SQL mycoll AS MyCollClass
SQL employee GENERATE Employee AS MyEmployee
```

次の処理が行われます。

- ユーザー定義型 `MYOBJ` の名前は、カスタム・オブジェクト・クラスの名前として入力した `Myobj` になります。JPublisher では、ソース・ファイル `Myobj.sqlj`（または、`Myobj` にメソッドがない場合は `Myobj.java`）および `MyobjRef.java` が生成されます。
- ユーザー定義型 `MYCOLL` が、`MyCollClass` にマッピングされます。JPublisher では、`MyCollClass.java` ソース・ファイルが生成されます。
- ユーザー定義型 `EMPLOYEE` が `MyEmployee` クラスにマッピングされます。JPublisher では、ファイルがすでに存在していないかぎり、初期バージョンの `MyEmployee.sqlj`（または `.java`）に加え、ソース・ファイル `Employee.sqlj`（または `Employee.java`）および `MyEmployeeRef.java` が作成されます。データベースから取得したオブジェクトに `EMPLOYEE` 属性があると、Java でこの属性が `MyEmployee` のインスタンスとして生成されます。`EMPLOYEE` オブジェクトを直接取得する場合は、

MyEmployee のインスタンスに取り込まれます。MyEmployee コードに関する責任は開発者にありますが、EMPLOYEE オブジェクトに対して特別な機能を Java で実装するためには、JPublisher で生成した MyEmployee ソース・ファイルを利用して、そのファイルを編集できます。MyEmployee は Employee クラスにサブクラス化されます。

JPublisher のプロパティ・ファイルの使用方法

JPublisher のコマンドライン・オプション `-props` で、型マッピングなどのオプション設定に使用するプロパティ・ファイルを指定できます。

プロパティ・ファイルの `jpub.`（ピリオドを含む）は、コマンドラインの `-`（シングル・ダッシュ）に相当します。その他の構文は同じです。1 行に 1 つのオプションを指定してください。

型マッピングの場合、`jpub.sql` は `-sql` に相当します。単一の `jpub.sql` 設定で複数のマッピングを指定できます。また、Oracle9i リリース 2 (9.2) では、複数の `jpub.sql` オプションを使用できます。（複数の `-sql` オプションが 1 つのコマンドラインにある場合）オプションの効果は累積されます。

注意： Oracle9i リリース 2 (9.2) では、「`jpub.`」または「`--jpub.`」（2 つのダッシュの後に「`jpub.`」が続く）で開始しないプロパティ・ファイルの行は無視されます。これによって、型を作成する SQL スクリプトと JPublisher 用のプロパティ・ファイルに同じファイルを使用できます。JPublisher の各文を「`--`」（SQL のコメントを示す）で開始すると、SQL*Plus では無視されます。また、SQL 文は JPublisher によって無視されます。

プロパティ・ファイルの例 次の例では、JPublisher はコマンドラインの `-user` オプションに基づき、プロパティ・ファイル `jpub.properties` で型マッピングと属性マッピング・オプションを調べます。

コマンドラインで次のように指定します。

```
% jpub -props=jpub.properties -user=scott/tiger
```

プロパティ・ファイル `jpub.properties` の内容は次のとおりです。

```
jpub.sql=Myobj,mycoll:MyCollClass,employee:Employee:MyEmployee
jpub.usertypes=oracle
```

このように指定すると、oracle マッピングの設定が明示的に指定され、前の入力ファイルの例と同じ結果になります。

注意： SQLJ と異なり、JPublisher にはデフォルトのプロパティ・ファイルがありません。プロパティ・ファイルを使用するには、`-props` オプションを使用します。

カスタム Java クラスの作成およびメンバー名の指定

カスタム Java クラスの作成時に、カスタム・クラスの属性またはメソッドの名前を指定できます。ただし、この属性やメソッド名は、JPublisher コマンドラインでの指定はできません。JPublisher の INPUT ファイルでのみ指定可能で、次のように TRANSLATE 構文を使用します。

```
SQL udt <GENERATE GeneratedClass> <AS MapClass> <TRANSLATE membername1 AS Javaname1>
<, membername2 AS Javaname2> ...
```

TRANSLATE ペア (`membernameNAS JavanameN`) はカンマで区切ります。

たとえば、Oracle オブジェクト型 EMPLOYEE の ADDRESS 属性の名前を HomeAddress にし、GIVE_RAISE メソッドの名前を giveRaise() にするとします。また、Employee クラスを作成し、EMPLOYEE オブジェクトのマッピング先をこれから作成する MyEmployee クラスにするとします。（この例は INPUT ファイル構文の全体を示すものであり、メンバー名の指定には関係ありません。）

```
SQL employee GENERATE Employee AS MyEmployee
TRANSLATE address AS HomeAddress, GIVE_RAISE AS giveRaise
```

注意：

- メンバー名を指定すると、それ以外の指定しなかったメンバーすべてにデフォルトの名前が与えられます。
 - 属性を大文字 (`homeAddress` ではなく `HomeAddress`) で指定する理由は、アクセッサ・メソッドと同じネーミング規則が適用されるためです。たとえば、`getHomeAddress()` はネーミング規則に従っていますが、`gethomeAddress()` は従っていません。
-

JPublisher で生成されるラッパー・メソッド

ここでは、JPublisher でのラッパー・メソッドの生成方法と、ランタイムのラッパー・メソッド・コールの処理方法を説明します。

ラッパー・メソッドの生成

次に、JPublisher でのラッパー・メソッドの生成方法について説明します。

- JPublisher で生成したラッパー・メソッドは、SQLJ で実装されます。したがって、`-methods=true` に設定すると、オブジェクト型でメソッドを定義している場合は、カスタム・オブジェクト・クラスが、`.java` ファイルではなく、`.sqlj` ファイルに定義されます。`.sqlj` ファイルを変換するには、SQLJ を実行します。

注意： オブジェクト型でメソッドを定義していない場合でも、`-methods=always` の設定で `.sqlj` ファイルが生成されることを確認できます。詳細は、『Oracle9i JPublisher ユーザーズ・ガイド』を参照してください。

- JPublisher で生成したラッパー・メソッドは、すべてインスタンス・メソッドとして実装されます。サーバー側メソッドの起動にデータベース接続を必要とするからです。JPublisher 生成のカスタム Java クラスの各インスタンスに接続が対応付けられます。

ラッパー・メソッド・コールのランタイム実行

JPublisher で生成した Java ラッパー・メソッドをランタイムに実行する方法を説明します。ここで取り上げる Java ラッパー・メソッドとは、カスタム Java オブジェクトのメソッドのことを指していますが、SQL メソッドのラッピングとは、SQL オブジェクトのメソッドをラッパー・メソッドでラッピングすることを指します。

- カスタム Java オブジェクトが SQL オブジェクトに変換され、データベースに転送されると、ラッピングされた SQL メソッドが起動されます。この SQL メソッドの起動後、SQL オブジェクトの新しい値が戻り値として新規のカスタム Java オブジェクトの形で Java に戻されます。この戻り値は、ラッピングされた SQL メソッドからのファンクション戻り値（SQL メソッドがストアード・プロシージャである場合）、またはファンクション戻り値がすでに存在している場合は、追加の出力パラメータの配列要素（SQL メソッドがストアード・ファンクションである場合）のどちらかになります。
- 出力または入出力パラメータが単一要素配列の要素として渡されます。（出力および入出力パラメータの受渡し上の問題を回避するためです。6-9 ページの「[オブジェクト・メソッドでのカスタム Java クラスのサポート](#)」を参照してください。）入出力パラメータの場合は、ラッパー・メソッドが入力として配列要素をとります。処理後、ラッパーによって出力が配列要素に代入されます。

JPublisher のカスタム Java クラスの例

ここでは、次のユーザー定義型（6-21 ページの「[ユーザー定義データ型](#)」で作成）に対して、JPublisher で生成される ORADData 実装の例を示します。

- カスタム・オブジェクト・クラス（Oracle のオブジェクト型 ADDRESS に対応する Address）とそのカスタム参照クラス（AddressRef）
- カスタム・コレクション・クラス（Oracle のコレクション型 MODULETBL_T に対応する ModuletblT）

-methods オプションがデフォルトの true に設定されており、ADDRESS 型にメソッドがあるとします。この結果、.sqlj ファイルが Address クラスに対して生成されます。

注意： JPublisher で生成される SQLData および ORADData 実装の各例については、『Oracle9i JPublisher ユーザーズ・ガイド』を参照してください。

カスタム・オブジェクト・クラス : Address.sqlj

次に、JPublisher で生成されるカスタム・オブジェクト・クラスのソース・コードの例を示します。実装の詳細は省略されています。

この例では、6-21 ページの「[オブジェクト型の作成](#)」と異なり、Oracle オブジェクト ADDRESS の属性は street と zip_code のみです。

```
package bar;

import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORADData;
import oracle.sql.ORADDataFactory;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.MutableStruct;
```

```
public class Address implements ORADATA, ORADATAFactory
{
    public static final String _SQL_NAME = "SCOTT.ADDRESS";
    public static final int _SQL_TYPECODE = OracleTypes.STRUCT;

    public static ORADATAFactory getORADATAFactory()
    { ... }

    /* constructors */
    public Address()
    { ... }

    public Address(String street, java.math.BigDecimal zip_code)
        throws SQLException
    { ... }

    /* ORADATA interface */
    public Datum toDatum(Connection c) throws SQLException
    { ... }

    /* ORADATAFactory interface */
    public ORADATA create(Datum d, int sqlType) throws SQLException
    { ... }

    /* accessor methods */
    public String getStreet() throws SQLException
    { ... }

    public void setStreet(String street) throws SQLException
    { ... }

    public java.math.BigDecimal getZipCode() throws SQLException
    { ... }

    public void setZipCode(java.math.BigDecimal zip_code) throws SQLException
    { ... }
}
```

カスタム参照クラス : AddressRef.java

次に、ADDRESS オブジェクトへの参照として JPublisher で生成されるカスタム参照クラスのソース・コードの例を示します。実装の詳細は省略されています。

```
package bar;

import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.REF;
import oracle.sql.STRUCT;

public class AddressRef implements ORAData, ORADataFactory
{
    public static final String _SQL_BASETYPE = "SCOTT.ADDRESS";
    public static final int _SQL_TYPECODE = OracleTypes.REF;

    public static ORADataFactory getORADataFactory()
    { ... }

    /* constructors */
    public AddressRef()
    { ... }

    public static AddressRef(ORAData o) throws SQLException
    { ... }

    /* ORAData interface */
    public Datum toDatum(Connection c) throws SQLException
    { ... }

    /* ORADataFactory interface */
    public ORAData create(Datum d, int sqlType) throws SQLException
    { ... }

    public static AddressRef cast(ORAData o) throws SQLException
    { ... }

    public Address getValue() throws SQLException
    { ... }

    public void setValue(Address c) throws SQLException
    { ... }
}
```


カスタム・コレクション・クラス : ModuletblT.java

次に, JPublisher で生成されるカスタム・コレクション・クラスのソース・コードの例を示します。実装の詳細は省略されています。

```
import java.sql.SQLException;
import java.sql.Connection;
import oracle.jdbc.OracleTypes;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.ARRAY;
import oracle.sql.ArrayDescriptor;
import oracle.jpub.runtime.MutableArray;

public class ModuletblT implements ORAData, ORADataFactory
{
    public static final String _SQL_NAME = "SCOTT.MODULETBL_T";
    public static final int _SQL_TYPECODE = OracleTypes.ARRAY;

    public static ORADataFactory getORADataFactory()
    { ... }

    /* constructors */
    public ModuletblT()
    { ... }

    public ModuletblT(ModuleT[] a)
    { ... }

    /* ORAData interface */
    public Datum toDatum(Connection c) throws SQLException
    { ... }

    /* ORADataFactory interface */
    public ORAData create(Datum d, int sqlType) throws SQLException
    { ... }

    public String getBaseTypeName() throws SQLException
    { ... }

    public int getBaseType() throws SQLException
    { ... }

    public ArrayDescriptor getDescriptor() throws SQLException
    { ... }
```

```
/* array accessor methods */
public ModuleT[] getArray() throws SQLException
{ ... }

public void setArray(ModuleT[] a) throws SQLException
{ ... }

public ModuleT[] getArray(long index, int count) throws SQLException
{ ... }

public void setArray(ModuleT[] a, long index) throws SQLException
{ ... }

public ModuleT getObjectElement(long index) throws SQLException
{ ... }

public void setElement(ModuleT a, long index) throws SQLException
{ ... }
}
```

JPublisher 生成クラスの拡張

JPublisher で生成したカスタム Java クラスは、メソッドと一時フィールドを追加して、機能を拡張できます。このためには、JPublisher 生成クラスを拡張します。

たとえば、JPublisher で、SQL のオブジェクト型 ADDRESS からクラス JAddress を生成する場合を想定します。また、ADDRESS オブジェクトを表すクラス MyAddress を使用し、専用機能を実装します。この MyAddress クラスは、JAddress の拡張クラスとする必要があります。

JPublisher 生成クラスの機能を拡張する場合は、単にメソッドを追加する方法もあります。ただし、クラスを JPublisher で再生成する予定がある場合は、JPublisher 生成クラスへのメソッドの追加は避けてください。この方法で変更したクラスを JPublisher で再生成する場合は、コピーを保存しておき、変更内容を手動でマージする必要があります。

生成クラスを拡張するための JPublisher の機能

次の JPublisher の構文で JAddress を生成し、MyAddress にマッピングできます（6-38 ページの「[カスタム Java クラスの生成および代替クラスのマッピング](#)」を参照してください）。

```
-sql=ADDRESS:JAddress:MyAddress
```

または、INPUT ファイルで次のように指定します。

```
SQL ADDRESS GENERATE JAddress AS MyAddress
```

このように指定した結果、JPublisher で生成される参照クラスは、MyAddressRef.java の MyAddressRef であって、JAddressRef が生成されることはありません。

また、JPublisher 生成コードが変更され、次の機能が実装されます。

- SQL 型 ADDRESS の属性は、JAddress クラスではなく、MyAddress クラスで表されます。
- ADDRESS 型のメソッド引数およびファンクション結果は、JAddress クラスではなく、MyAddress クラスで表されます。
- SQL 型 ADDRESS の Java オブジェクトは、JAddress ファクトリではなく、MyAddress ファクトリで構築されます。

追加コードを作成する場合も、おそらく MyAddress を使用することになります。

実行時に Oracle JDBC ドライバによって、データベース内の ADDRESS データのオカレンスが、JAddress のインスタンスではなく、MyAddress のインスタンスにマッピングされます。

拡張クラスの要件

JPublisher ではデフォルトの場合、作成されるファイルがすでに存在していないかぎり、MyAddress.sqlj ファイルまたは MyAddress.java ファイル（元のクラスがメソッドを使用し、これらのメソッドを公開している場合は MyAddress.sqlj ファイル）に、初期バージョンのユーザー・サブクラス MyAddress が作成されます（ファイルがすでに存在している場合、ファイルは変更されません）。このファイルは、必要な機能を追加するために編集できます。

MyAddress には、引数を持たないコンストラクタを用意する必要があります。適切に初期化したオブジェクトを容易に構築するには、スーパークラスのコンストラクタを明示的にまたは暗黙的に起動します。

JPublisher 生成クラスのサブクラスを生成した場合、_SQL_NAME フィールド（必須）の定義と _SQL_TYPECODE フィールドの定義がこのサブクラスに継承されます。

また、次のどちらかが当てはまります。

- JPublisher で生成したクラスには、ORADATA および ORADATAFACTORY インタフェースが実装されているため、必要とされる toDatum() および JPublisher で生成されたクラスの create() 機能がサブクラスに継承されます。ただし、サブクラスには、自分でマッピングしたクラスのインスタンス（MyAddress オブジェクトなど）を戻り値とする getORADATAFACTORY() メソッドを実装します。

または

- JPublisher で生成したクラスに SQLDATA インタフェースが実装されている場合、この実装と生成クラスの readSQL() および writeSQL() 機能がサブクラスに継承されます。

JPublisher 生成のカスタム・オブジェクト・クラス : JAddress.sqlj

JPublisher で生成された JAddress クラスのコード（ORADATA および ORADATAFACORY を実装する）は、Address が JAddress に置換されることを除き、前述の Address クラスのコードとほぼ同じです。

JPublisher 生成のその他の参照クラス : MyAddressRef.java

これまでの項の続きとして、JPublisher 生成の参照クラス MyAddressRef のコードを示します（ここで取り上げるクラスは、JAddressRef ではありません。MyAddress クラスが ADDRESS オブジェクトのマッピング先となるためです）。また、このクラスには ORADATA と ORADATAFACORY が実装されます。この実装は、クラス名が異なることと、設定メソッドおよび取得メソッドが Address インスタンスではなく MyAddress インスタンスを使用することを除いて、AddressRef.java の実装とほぼ同じです。

拡張カスタム・オブジェクト・クラス : MyAddress.sqlj

これまでの例の続きとして、JPublisher で生成される JAddress クラスのサブクラスである MyAddress のサンプル・コードを示します。JAddress から継承された内容は、コード中のコメントに記述してあります。実装の詳細は省略されています。

```
import java.sql.SQLException;
import oracle.sql.ORAData;
import oracle.sql.ORADataFactory;
import oracle.sql.Datum;
import oracle.sql.STRUCT;
import oracle.jpub.runtime.MutableStruct;

public class MyAddress extends JAddress
{
    /* _SQL_NAME inherited from MyAddress */
    /* _SQL_TYPECODE inherited from MyAddress */

    static _myAddressFactory = new MyAddress();

    public static ORADataFactory getORADataFactory()
    {
        return _myAddressFactory;
    }

    /* constructor */
    public MyAddress()
    { super(); }

    /* ORADATA interface */
    /* toDatum() inherited from JAddress */
```

```

/* ORADDataFactory interface */
public ORADData create(oracle.sql.Datum d, int sqlType) throws SQLException
{ ... }

/* accessor methods inherited from JAddress */

/* Additional methods go here. These additional methods (not shown)
   are the reason that JAddress was extended.
*/
}

```

SQLJ 実行文の強い型指定のオブジェクトと参照

Oracle SQLJ では、強い型指定のオブジェクトまたは参照をホスト式およびイテレータを使用して、オブジェクト・データの読み込みおよび書き込みを柔軟に行えます。

イテレータの場合は、カスタム・オブジェクト・クラスをイテレータの列型として使用できます。または、属性の SQL データ型にマップしている列型を使用して、イテレータ列をオブジェクトの各属性に対応付けることができます（エクステンント表と同様）。

ホスト式の場合は、カスタム・オブジェクト・クラス型またはカスタム参照クラス型のホスト変数を使用できます。または、属性の SQL データ型にマップしている変数型を使用して、ホスト変数をオブジェクトの属性に対応付けることができます。

次に、Oracle オブジェクトの操作例を示します。SQLJ 実行文のホスト変数とイテレータ列に対して、カスタム・オブジェクト・クラス、カスタム・オブジェクト・クラスの属性およびカスタム参照クラスを使用します。

最初の 2 つの例は、オブジェクト・レベルでの操作です。

1. [オブジェクトとオブジェクト参照のイテレータ列への取出し](#)
2. [オブジェクトの更新](#)

3 番目の例は、スカラー属性レベルでの操作です。

3. [各オブジェクト属性から作成したオブジェクトの挿入](#)

4 番目の例は、参照による操作です。

4. [オブジェクト参照の更新](#)

Oracle のオブジェクト型 ADDRESS および PERSON は、6-21 ページの「[オブジェクト型の作成](#)」を参照してください。

オブジェクトとオブジェクト参照のイテレータ列への取出し

この例では、カスタム Java クラスとカスタム参照クラス（ORADData 実装）をイテレータの列型として使用します。

Oracle のオブジェクト型 ADDRESS が次のように定義されているものとします。

```
CREATE TYPE ADDRESS AS OBJECT
( street VARCHAR(40),
  zip NUMBER );
```

また、表 EMPADDRS が次のように定義されているものとします。この表には、ADDRESS 列と ADDRESS 参照列があります。

```
CREATE TABLE empaddrs
( name VARCHAR(60),
  home ADDRESS,
  loc REF ADDRESS );
```

Oracle のオブジェクト型 ADDRESS に対応するカスタム Java クラス Address とカスタム参照クラス AddressRef を JPublisher で生成するか、または手動で作成すると、次のように Address と AddressRef とを名前指定イテレータで使用できます。

次のように宣言します。

```
#sql iterator EmpIter (String name, Address home, AddressRef loc);
```

実行可能コードを示します。

```
EmpIter ecur;
#sql ecur = { SELECT name, home, loc FROM empaddrs };
while (ecur.next()) {
    Address homeAddr = ecur.home();
    // Print out the home address.
    System.out.println ("Name: " + ecur.name() + "\n" +
                        "Home address: " + homeAddr.getStreet() + " " +
                        homeAddr.getZip());
    // Now update the loc address zip code through the address reference.
    AddressRef homeRef = ecur.loc();
    Address location = homeRef.getValue();
    location.setZip(new BigDecimal(98765));
    homeRef.setValue(location);
}
...
```

メソッドのコール `ecur.home()` は、イテレータの `home` 列から `Address` オブジェクトを抽出し、ローカル変数 `homeAddr` に代入します（効率化）。このオブジェクトの属性にアクセスするには、次に示した `Java` の標準ドット区切り構文を使用します。

```
homeAddr.getStreet()
```

ロケーション・アドレス（この例では `zip` コード）を操作するには、`getValue()` および `setValue()` メソッドを使用します。これらのメソッドは、`JPublisher` 生成のカスタム参照クラスの標準機能です。

注意： 次の例では、6-21 ページの「[オブジェクト型の作成](#)」の `SQL` スクリプトで定義した型と表を使用します。

オブジェクトの更新

この例では、`Java` の型 `Address` の入力ホスト変数を宣言し、設定します。この変数によって、`employees` 表の列にある `ADDRESS` オブジェクトを更新します。更新前と更新後のアドレスが `Address` 型の出力ホスト変数に取り出され、検証用に出力されます。

```
...
// Updating an object

static void updateObject()
{
    Address addr;
    Address new_addr;
    int empnum = 1001;

    try {
        #sql {
            SELECT office_addr
            INTO :addr
            FROM employees
            WHERE empnumber = :empnum };
        System.out.println("Current office address of employee 1001:");

        printAddressDetails(addr);

        /* Now update the street of address */

        String street = "100 Oracle Parkway";
        addr.setStreet(street);

        /* Put updated object back into the database */
```

```
try {
    #sql {
        UPDATE employees
        SET office_addr = :addr
        WHERE empnumber = :empnum };
    System.out.println
        ("Updated employee 1001 to new address at Oracle Parkway.");

    /* Select new address to verify update */

    try {
        #sql {
            SELECT office_addr
            INTO :new_addr
            FROM employees
            WHERE empnumber = :empnum };

        System.out.println("New office address of employee 1001:");
        printAddressDetails(new_addr);

        } catch (SQLException exn) {
            System.out.println("Verification SELECT failed with "+exn); }

        } catch (SQLException exn) {
            System.out.println("UPDATE failed with "+exn); }

        } catch (SQLException exn) {
            System.out.println("SELECT failed with "+exn); }
    }
    ...
}
```

Address オブジェクトの `setStreet()` アクセッサ・メソッドが使用されています。`JPublisher` で生成したカスタム Java クラスには、属性ごとにアクセッサ・メソッドが用意されています。

この例では、`printAddressDetails()` ユーティリティを使用しています。このメソッドのソース・コードを示します。

```
static void printAddressDetails(Address a) throws SQLException
{
    if (a == null) {
        System.out.println("No Address available.");
        return;
    }
}
```



```

String street = ((a.getStreet()==null) ? "NULL street" : a.getStreet());
String city = (a.getCity()==null) ? "NULL city" : a.getCity();
String state = (a.getState()==null) ? "NULL state" : a.getState();
String zip_code = (a.getZipCode()==null) ? "NULL zip" : a.getZipCode();

System.out.println("Street: '" + street + "'");
System.out.println("City:   '" + city   + "'");
System.out.println("State:  '" + state  + "'");
System.out.println("Zip:    '" + zip_code + "'");
}

```

各オブジェクト属性から作成したオブジェクトの挿入

この例では、PERSON オブジェクトとネストされている ADDRESS オブジェクトの属性に対応する入力ホスト変数を宣言し、設定します。これらの値を使用して、新しい PERSON オブジェクトをデータベース内の `persons` 表に挿入します。

```

...
// Inserting an object

static void insertObject()
{
    String new_name   = "NEW PERSON";
    int    new_ssn    = 987654;
    String new_street = "NEW STREET";
    String new_city   = "NEW CITY";
    String new_state  = "NS";
    String new_zip    = "NZIP";
    /*
     * Insert a new PERSON object into the persons table
     */
    try {
        #sql {
            INSERT INTO persons
            VALUES (PERSON(:new_name, :new_ssn,
                           ADDRESS(:new_street, :new_city, :new_state, :new_zip))) };

        System.out.println("Inserted PERSON object NEW PERSON.");

    } catch (SQLException exn) { System.out.println("INSERT failed with "+exn); }
}
...

```

オブジェクト参照の更新

この例では、`persons` 表から `PERSON` 参照を選択し、この値を使用して `employees` 表内の `PERSON` 参照を更新します。属性値の基準チェックには、単純な (`int` および `String`) 入力ホスト変数を使用します。更新後の値で参照先 `PERSON` オブジェクトを選択し、情報を出力します。ユーザーは変更内容を検証できます。

```
...
// Updating a REF to an object

static void updateRef()
{
    int empnum = 1001;
    String new_manager = "NEW PERSON";

    System.out.println("Updating manager REF.");
    try {
        #sql {
            UPDATE employees
            SET manager =
              (SELECT REF(p) FROM persons p WHERE p.name = :new_manager)
            WHERE empnumber = :empnum };

        System.out.println("Updated manager of employee 1001. Selecting back");

    } catch (SQLException exn) {
        System.out.println("UPDATE REF failed with "+exn); }

    /* Select manager back to verify the update */
    Person manager;
    try {
        #sql {
            SELECT deref(manager)
            INTO :manager
            FROM employees e
            WHERE empnumber = :empnum };

        System.out.println("Current manager of "+empnum+":");
        printPersonDetails(manager);
    } catch (SQLException exn) {
        System.out.println("SELECT REF failed with "+exn); }
}
...
```

注意： この例では、前述の表の別名構文 (p) を使用しています。参照先オブジェクトから参照を選択するには、REF 構文を使用します。参照からオブジェクトを選択するには、DEREF 構文を使用します。表の別名、REF および DEREF の詳細は、『Oracle9i SQL リファレンス』を参照してください。

SQLJ 実行文の強い型指定のコレクション

Oracle SQLJ では、強い型指定のオブジェクトおよび参照と同じように、強い型指定のコレクションをイテレータまたはホスト式で使用する、データの読み込みおよび書き込みを行えます。

SQLJ 開発者にとっては、2 種類のコレクション (VARRAY とネストした表) の扱いは基本的に同じですが、実装とパフォーマンスの面で相違点が多少あります。

Oracle SQLJ および Oracle SQL 全般では、複数の構文を使用してネストした表にアクセスし、操作できます。ネストした表は、単独で操作することも、外部のネストした表と一緒に操作することも可能です。ここでは、ネストした表単独の操作を詳細レベルの操作と呼び、ネストした表と外部の表の同時操作をマスター・レベルの操作と呼びます。

ここでは、いくつかの構文について簡単に説明してから、ネストした表の操作例を示します。ネストした表の操作は、VARRAY の操作より複雑です。

Oracle のコレクション型 `MODULETBL_T` とその表およびオブジェクト型の定義は、6-23 ページの「[コレクション型の作成](#)」を参照してください。

ネストした表の説明の後に、簡単な VARRAY の例を示します。

注意： Oracle SQLJ では、VARRAY 型および NESTED TABLE 型を型単位でのみ取得できます。Oracle SQL では、ネストした表の間合せを選択的に実行できます。

ネストした表へのアクセス : TABLE 構文と CURSOR 構文

Oracle SQLJ では、ネストしたイテレータを使用して、ネストした表のデータにアクセスできます。外部の SELECT 文で CURSOR キーワードを使用し、内部の SELECT 文をカプセル化します。6-61 ページの「[ネスト・イテレータによるネストした表からのデータの選択](#)」を参照してください。

Oracle SQLJ では、TABLE キーワードを使用すると、ネストした表を行単位で操作できます。Oracle ではこのキーワードを使用すると、副問合せで戻される列値が、スカラー値ではなくネストした表であることが認識されます。単一系列値を戻す副問合せまたはネストした表に解決される式の前に、TABLE キーワードを指定する必要があります。

次に、TABLE 構文の使用例を示します。

```
UPDATE TABLE(SELECT a.modules FROM projects a WHERE a.id=555) b
SET module_owner=
  (SELECT ref(p) FROM employees p WHERE p.ename= 'Smith')
WHERE b.module_name = 'Zebra';
```

この例のように TABLE を使用すると、参照先の単一のネストした表が外部の表の列から選択されます。

注意： この例では、前述のように、表の別名構文（projects には a、ネストした表には b、また employees には p）が使用されています。表の別名の詳細は、『Oracle9i SQL リファレンス』を参照してください。

ネストした表を含む行の挿入

この例では、マスター・レベル（外部の表）と詳細レベル（ネストした表）を同時に明示的に操作します。ここでは、行を projects 表に挿入します。この表の各行に型 MODULE_TBL_T のネストした表が収容され、MODULE_T オブジェクトの行が格納されます。

最初に、スカラー値を設定します（id、name、start_date、duration）。次に、ネストした表の値を設定します。ネストした表の要素は複数の属性を持つオブジェクトなので、抽象化も行います。ネストした表の値を設定するとき、ネストした表の各 MODULE_T オブジェクトに対して、属性値を設定する必要があります。最後に、owner 値（初期値 NULL）を別の文で設定します。

```
// Insert Nested table details along with master details

public static void insertProject2(int id) throws Exception
{
    System.out.println("Inserting Project with Nested Table details..");
    try {
        #sql { INSERT INTO Projects(id,name,owner,start_date,duration, modules)
              VALUES ( 600, 'Ruby', null, '10-MAY-98', 300,
                moduletbl_t(module_t(6001, 'Setup ', null, '01-JAN-98', 100),
                  module_t(6002, 'BenchMark', null, '05-FEB-98',20) ,
                  module_t(6003, 'Purchase', null, '15-MAR-98', 50),
                  module_t(6004, 'Install', null, '15-MAR-98',44),
                  module_t(6005, 'Launch', null,'12-MAY-98',34))) };
    } catch ( Exception e) {
        System.out.println("Error:insertProject2");
        e.printStackTrace();
    }

    // Assign project owner to this project
```

```

try {
    #sql { UPDATE Projects pr
        SET owner=(SELECT ref(pa) FROM participants pa WHERE pa.empno = 7698)
        WHERE pr.id=600 };
} catch ( Exception e) {
    System.out.println("Error:insertProject2:update");
    e.printStackTrace();
}
}

```

ホスト式へのネストした表の取出し

この例では、ネストした表を詳細レベルで直接操作します。前述したとおり、ModuletblT は、ネストした表 MODULETBL_T に対する JPublisher 生成のカスタム・コレクション・クラス (ORADData 実装)、ModuleT は MODULE_T オブジェクトに対する JPublisher 生成のカスタム・オブジェクト・クラスです。ネストした表 MODULETBL_T には MODULE_T オブジェクトがあります。

MODULE_T オブジェクトのネストした表を projects 表の modules 列から選択し、ModuletblT ホスト変数に格納します。

次に、getArray() メソッドを介してネストした表の要素にアクセスするメソッドに、ネストした表を格納した ModuletblT 変数を渡します。getArray() メソッドはデータを ModuleT[] 配列に書き込みます。JPublisher で生成したカスタム・コレクション・クラスには、必ず getArray() メソッドがあります。この ModuleT[] 配列の各要素を ModuleT オブジェクトにコピーし、各属性をアクセッサ・メソッド (getModuleName() など) で取得し、出力します。JPublisher で生成したカスタム・オブジェクト・クラスには、必ずこのようなアクセッサ・メソッドがあります。

```

static ModuletblT mymodules=null;
...

public static void getModules2(int projId)
throws Exception
{
    System.out.println("Display modules for project " + projId );

    try {
        #sql {SELECT modules INTO :mymodules
            FROM projects WHERE id=:projId };
        showArray(mymodules);
    } catch(Exception e) {
        System.out.println("Error:getModules2");
        e.printStackTrace();
    }
}

```

```
public static void showArray(ModuleTblT a)
{
    try {
        if ( a == null )
            System.out.println( "The array is null" );
        else {
            System.out.println( "printing ModuleTable array object of size "
                                +a.length());
            ModuleT[] modules = a.getArray();

            for (int i=0;i<modules.length; i++) {
                ModuleT module = modules[i];
                System.out.println("module "+module.getModuleId()+
                                   ", "+module.getModuleName()+
                                   ", "+module.getModuleStartDate()+
                                   ", "+module.getModuleDuration());
            }
        }
    }
    catch( Exception e ) {
        System.out.println("Show Array");
        e.printStackTrace();
    }
}
```

TABLE 構文によるネストした表の操作

この例では、TABLE 構文を使用して、ネストした表を詳細レベルで操作します。マスター・レベルの基準に基づき、ネストした表の要素に直接アクセスして更新します。

`assignModule()` メソッドを使用すると、PROJECTS 表の MODULES 列から `MODULE_T` オブジェクトのネストした表が選択され、その後でこの NESTED TABLE 型の特定行の `MODULE_NAME` が更新されます。

同様に、`deleteUnownedModules()` メソッドを使用すると、`MODULE_T` オブジェクトのネストした表が選択され、その後でこのネストした表の非所有モジュール (`MODULE_OWNER` が `NULL` のモジュール) が削除されます。

これらのメソッドでは、前述のように表の別名構文を使用します。この例では、ネストした表に `m`、`participants` 表に `p` を使用しています。表の別名の詳細は、『Oracle9i SQL リファレンス』を参照してください。

```
/* assignModule
// Illustrates accessing the nested table using the TABLE construct
// and updating the nested table row
*/
public static void assignModule(int projId, String moduleName,
                                String modOwner) throws Exception
```

```

{
    System.out.println("Update:Assign '"+moduleName+"' to '"+ modOwner+"'");

    try {
        #sql {UPDATE TABLE(SELECT modules FROM projects WHERE id=:projId) m
            SET m.module_owner=
                (SELECT ref(p) FROM participants p WHERE p.ename= :modOwner)
            WHERE m.module_name = :moduleName };
    } catch(Exception e) {
        System.out.println("Error:insertModules");
        e.printStackTrace();
    }
}

/* deleteUnownedModules
// Demonstrates deletion of the Nested table element
*/

public static void deleteUnownedModules(int projId)
throws Exception
{
    System.out.println("Deleting Unowned Modules for Project " + projId);
    try {
        #sql { DELETE TABLE(SELECT modules FROM projects WHERE id=:projId) m
            WHERE m.module_owner IS NULL };
    } catch(Exception e) {
        System.out.println("Error:deleteUnownedModules");
        e.printStackTrace();
    }
}
}

```

ネスト・イテレータによるネストした表からのデータの選択

SQLJ では、ネストされているイテレータを使用して、ネストした表にアクセスできます。このためには、次の例に使用されている CURSOR 構文を使用する必要があります。

コードで名前指定イテレータ・クラス `ModuleIter` を定義します。このクラスを `modules` 列の型として、別の名前指定イテレータ・クラス `ProjIter` で使用します。設定した `ProjIter` インスタンスの中の各 `modules` 項目は、ネスト・イテレータとして表されたネストした表です。

CURSOR 構文は、ネストされている SELECT 文の要素であり、ネストされているイテレータを移入します。

選択されたデータは、イテレータのアクセッサ・メソッドによってユーザーに出力されます。

この例では、前述のように、表の別名構文を使用します。ここでは、`projects` 表には `a` を使用し、ネストした表には `b` を使用しています。表の別名の詳細は、『Oracle9i SQL リファレンス』を参照してください。

...

```
// The Nested Table is accessed using the ModuleIter
// The ModuleIter is defined as Named Iterator
```

```
#sql public static iterator ModuleIter(int moduleId ,
                                       String moduleName ,
                                       String moduleOwner);
```

```
// Get the Project Details using the ProjIter defined as
// Named Iterator. Notice the use of ModuleIter below:
```

```
#sql public static iterator ProjIter(int id,
                                       String name,
                                       String owner,
                                       Date start_date,
                                       ModuleIter modules);
```

...

```
public static void listAllProjects() throws SQLException
{
```

```
    System.out.println("Listing projects...");
```

```
    // Instantiate and initialize the iterators
```

```
    ProjIter projs = null;
```

```
    ModuleIter mods = null;
```

```
#sql projs = {SELECT a.id,
                   a.name,
                   initcap(a.owner.ename) as "owner",
                   a.start_date,
                   CURSOR (
                       SELECT b.module_id AS "moduleId",
                              b.module_name AS "moduleName",
                              initcap(b.module_owner.ename) AS "moduleOwner"
                       FROM TABLE(a.modules) b) AS "modules"
                   FROM projects a };
```



```

// Display Project Details

while (projs.next()) {
    System.out.println( "\n" + projs.name() + " Project Id:"
        + projs.id() + " is owned by " + projs.owner() + " "
        + " start on "
        + projs.start_date());

    // Notice below the modules from the ProjIter are assigned to the module
    // iterator variable

    mods = projs.modules();
    System.out.println ("Modules in this Project are : ");

    // Display Module details

    while(mods.next()) {
        System.out.println ( "  " + mods.moduleId() + " " +
            mods.moduleName() + " owner is " +
            mods.moduleOwner()+" " );
    }
    mods.close();
}
projs.close();
}

```

ホスト式への VARRAY の取出し

ここでは、VARRAY をホスト式に取り出す例を示します。次の SQL 定義を想定します。

```

CREATE TYPE PHONE_ARRAY IS VARRAY (10) OF varchar2(30)
/
/** Create ADDRESS UDT */
CREATE TYPE ADDRESS AS OBJECT
(
    street      VARCHAR(60),
    city        VARCHAR(30),
    state       CHAR(2),
    zip_code    CHAR(5)
)

```

```
)
/
/** Create PERSON UDT containing an embedded ADDRESS UDT */
CREATE TYPE PERSON AS OBJECT
(
    name      VARCHAR(30),
    ssn       NUMBER,
    addr      ADDRESS
)
/

CREATE TABLE employees
( empnumber          INTEGER PRIMARY KEY,
  person_data        REF person,
  manager            REF person,
  office_addr        address,
  salary             NUMBER,
  phone_nums         phone_array
)
/
```

SQL 型 PHONE_ARRAY にマッピングするカスタム・コレクション・クラス PhoneArray を、JPublisher で生成するものとします。

次のメソッドでこの表から行を選択し、データを PhoneArray 型のホスト変数に格納します。

```
private static void selectVarray() throws SQLException
{
    PhoneArray ph;
    #sql {select phone_nums into :ph from employees where empnumber=2001};
    System.out.println(
        "there are "+ph.length()+" phone numbers in the PhoneArray.  They are:");

    String [] pharr = ph.getArray();
    for (int i=0;i<pharr.length;++i)
        System.out.println(pharr[i]);
}
```

VARRAY 行への挿入

ここでは、ホスト式から VARRAY にデータを挿入する例を示します。前例と同じ SQL 定義とカスタム・コレクション・クラス (PhoneArray) を使用します。

次のメソッドは PhoneArray インスタンスを定義し、ホスト変数として使用して、データをデータベース内の VARRAY に挿入します。

```
// creates a varray object of PhoneArray and inserts it into a new row
private static void insertVarray() throws SQLException
{
    PhoneArray phForInsert = consUpPhoneArray();
    // clean up from previous demo runs
    #sql {delete from employees where empnumber=2001};
    // insert the PhoneArray object
    #sql {insert into employees (empnumber, phone_nums)
        values(2001, :phForInsert)};
}

private static PhoneArray consUpPhoneArray()
{
    String [] strarr = new String[3];
    strarr[0] = "(510) 555.1111";
    strarr[1] = "(617) 555.2222";
    strarr[2] = "(650) 555.3333";
    return new PhoneArray(strarr);
}
```

シリアル化された Java オブジェクト

Java オブジェクトのインスタンスをデータベースとの間で読み書きする場合、Java クラスに対応する SQL オブジェクト型を定義して、前述のカスタム Java クラスのマッピング機構を使用すると、利便性が高まる場合があります。これによって、Java オブジェクトへの完全な SQL 問合せが可能になります。

ただし、RAW または BLOB のデータベース列型を使用して、Java オブジェクトをそのまま格納した後で取り出す場合もあります。このためには、2 つの方法があります。

- 標準でない型マップの拡張機能を使用するか、型コード・フィールドをシリアル化可能なクラスに追加することによって、シリアル化可能な Java クラスを RAW または BLOB 列にマップできます。これにより、RAW または BLOB としてシリアル化可能なクラスのインスタンスを格納できます。
- ORADATA 機能を使用して、RAW または BLOB 列に格納できるインスタンスを持つ、シリアル化可能なラッパー・クラスを定義できます。

これらの方法で実行するシリアル化は、Oracle SQLJ ランタイム・ライブラリのみ対象です。

RAW および BLOB 列に対する Java クラスのシリアル化

RAW 列または BLOB 列内に直接 Java クラスのインスタンスを格納する場合は、SQL と Java のマッピングを指定する非標準要件を満たす必要があります。(SQLJ 文内では、組込み型のように透過的に、シリアル化可能な Java オブジェクトを読み書きできます。)

SQL と Java のマッピングを指定する場合、2 つのオプションがあります。

- 型マップを接続コンテキスト宣言で宣言して、この型マップでマッピングを指定します。
- `public static final` フィールド `_SQL_TYPECODE` を使用して、マッピングを指定します。

ここでは、各オプションについて説明します。

シリアル化可能なクラスに対する型マップの定義

SAddress、pack.SPerson および pack.Manager.InnerSPM (InnerSPM は Manager の内部クラスです) がシリアル化可能な Java クラスであるとしします。つまり、これらのクラスは `java.io.Serializable` インタフェースを実装します。

このクラスは、宣言済みの接続コンテキスト型の明示的接続コンテキスト・インスタンスを使用する文でのみ採用する必要があります。次の SerContext がその例です。

```
SAddress          a =...;
pack.SPerson      p =...;
pack.Manager.InnerSPM pm =...;
SerContext ctx = new SerContext(url,user,pwd,false);
#sql [ctx] { ... :a ... :OUT p ... :INOUT pm ... };
```

この場合の要件を次に示します。

- 接続コンテキスト型は、`java.util.PropertyResourceBundle` を実装している関連クラスを指定する `with` 句の `typeMap` 属性で宣言されている必要があります。前述の例では、SerContext が次のように宣言されています。

```
#sql public static context SerContext with (typeMap="SerMap");
```

- 型マップ・リソースは RAW 列または BLOB 列から、シリアル化可能な Java クラスまでに非標準マッピングを提供する必要があります。このマッピングは、次の形式のエントリで指定します。Java クラスが RAW 列または BLOB のいずれの列にマップされているかによって、形式が異なります。

```
oracle-class.<java_class_name>=JAVA_OBJECT RAW
oracle-class.<java_class_name>=JAVA_OBJECT BLOB
```

キーワード `oracle-class` は、これが Oracle 固有の拡張機能であることを示しています。この例では、リソース・ファイル `SerMap.properties` に次のエントリが含まれます。

```
oracle-class.SAddress=JAVA_OBJECT RAW
oracle-class.pack.SPerson=JAVA_OBJECT BLOB
oracle-class.packManager$InnerSPM=JAVA_OBJECT RAW
```

(パッケージとクラス名の分割には「`.`」を使用しますが、内部クラス名を分割するには文字「`$`」を使用する必要があります。)

この Oracle 固有の拡張機能は、標準 `SQLData` 型マップ・エントリと同じ型マップ・リソースに配置できます。

フィールド使用によるシリアル化可能クラスのマッピングの指定

シリアル化可能クラスに型マップを使用する別の方法として、シリアル化可能クラスの `static` フィールドを使用して、型マッピングを指定できます。

6-66 ページの「[シリアル化可能なクラスに対する型マップの定義](#)」にある例の `SAddress` および `SPerson` クラスなど、`java.io.Serializable` インタフェースを実装するクラスに次のフィールドのいずれかを追加できます。

```
public final static int _SQL_TYPECODE = oracle.jdbc.OracleTypes.RAW;
```

または

```
public final static int _SQL_TYPECODE = oracle.jdbc.OracleTypes.BLOB;
```

注意： 手動でのクラスへの `_SQL_TYPECODE` フィールド追加は、型マップ機能の使用に置き換えられました。

Java オブジェクトのシリアル化における制限

シリアル化の効果について、認識しておく必要があります。2つのオブジェクト A および B が、同じオブジェクト C を共有している場合、A および B のシリアル化と以降のシリアル化解除に際して、それぞれがオブジェクト C のクローンに焦点をあてるため、共有は解かれます。

さらに、与えられた Java クラスに対して宣言できるシリアル化は、RAW または BLOB の 1 種類のみです。SQLJ トランスレータは、実際の使用方法が RAW または BLOB のいずれかに適合することのみをチェックします。

RAW 列では、サイズが制限されています。シリアル化した Java オブジェクトが、列のサイズを超えると、ランタイム・エラーが発生します。

BLOB 列の列サイズ制限がより緩やかです。Oracle9i リリース 2 (9.2) では、シリアル化された Java オブジェクトの BLOB 列への書込みは、Oracle JDBC OCI および Thin ドライバでサポートされます。(Oracle9i リリース 1 (9.0.1) では、OCI ドライバでのみサポートされていました。) BLOB 列からのシリアル化されたオブジェクトの取得は、リリース 1 (9.0.1) およびリリース 2 (9.2) のすべての Oracle JDBC ドライバでサポートされています。

最後に、シリアル化された Java オブジェクトをこの方法で処理することは、Oracle 固有拡張機能であり、Oracle SQLJ ランタイムと、デフォルトの Oracle 固有コード生成 (変換時の `-codegen=oracle`) または ISO 標準コード生成 (`-codegen=iso`) の場合は Oracle 固有のプロファイルのカスタマイズが必要です。将来、Oracle ではシリアル化した Java オブジェクトを直接カプセル化する SQL 型をサポートする予定です。これらは、JDBC 2.0 で、`JAVA_OBJECT SQL` 型として説明されています。現在のところ、各 BLOB および RAW 指定を対応する `JAVA_OBJECT SQL` 型に置き換え、エントリ上で接頭辞 `oracle-` を削除できます。

注意： この特定のシリアル化メカニズムの実装では、JDBC 型マップは使用しません。BLOB または RAW に対するマップは、変換時にカスタマイズされた Oracle のプロファイルにハードコードされるか、Java コードに直接生成されます。

SerializableDatum: ORADData の実装

`oracle.sql.STRUCT`、`oracle.sql.REF` または `oracle.sql.ARRAY` 以外の `oracle.sql.*` 型にマッピングするカスタム Java クラスの定義例は、6-16 ページの「[ORADData 実装のその他の使用例](#)」を参照してください。

RAW フィールドに対して Java オブジェクトのシリアル化およびシリアル化解除を行う場合の例では、カスタム Java クラスを `oracle.sql.RAW` 型にマッピングしています。(これは、BLOB フィールドにも同様に当てはまり、カスタム Java クラスを `oracle.sql.BLOB` 型にマッピングします。)

ここでは、このようなアプリケーションの例として、ORADData インタフェースを実装する `SerializableDatum` クラスの作成を説明します。このクラスは、6-5 ページの「[カスタム Java クラス](#)」に記載されている汎用形式のカスタム Java クラスに準拠しています。

この例では、`SerializableDatum` の開発手順を詳細に示してから、完全なサンプル・コードを示します。

注意： このアプリケーションでは、`java.io`、`java.sql`、`oracle.sql` および `oracle.jdbc` パッケージのクラスを使用しています。インポート文はここには含まれていません。

1. まず、このクラスのスケルトンから始めます。

```
public class SerializableDatum implements ORADData
{
    // <Client methods for constructing and accessing the Java object>

    public Datum toDatum(java.sql.Connection c) throws SQLException
    {
        // <Implementation of toDatum()>
    }

    public static ORADDataFactory getORADDataFactory()
    {
        return FACTORY;
    }

    private static final ORADDataFactory FACTORY =
        // <Implementation of an ORADDataFactory for SerializableDatum>

    // <Construction of SerializableDatum from oracle.sql.RAW>

    public static final int _SQL_TYPECODE = OracleTypes.RAW;
}
```

SerializableDatum 自体では、ORADDataFactory インタフェースが実装されることはありませんが、このクラスの getORADDataFactory() メソッドを使用すると、このインタフェースを実装した static メンバーが戻されます。

_SQL_TYPECODE を OracleTypes.RAW に設定します。データベースに対してこのデータ型の読み込みおよび書き込みを行うためです。SQLJ トランスレータは、この型コード情報に基づき、オンラインで型チェックを行い、ユーザー定義の Java 型と SQL 型間の互換性を検証します。

2. 次の処理を行うクライアント・メソッドを定義します。
 - SerializableDatum オブジェクトの作成
 - SerializableDatum オブジェクトの設定
 - SerializableDatum オブジェクトからのデータの取得

```
// Client methods for constructing and accessing a SerializableDatum
```

```
private Object m_data;
public SerializableDatum()
{
    m_data = null;
}
public void setData(Object data)
{
    m_data = data;
}
public Object getData()
{
    return m_data;
}
```

3. toDatum() メソッドを実装して、SerializableDatum オブジェクトから oracle.sql.RAW オブジェクトへデータをシリアル化します。toDatum() を実装すると、oracle.sql.RAW インスタンスのように m_data フィールドのオブジェクトをシリアル化した表現に戻す必要があります。

```
// Implementation of toDatum()
try {
    ByteArrayOutputStream os = new ByteArrayOutputStream();
    ObjectOutputStream oos = new ObjectOutputStream(os);
    oos.writeObject(m_data);
    oos.close();
    return new RAW(os.toByteArray());
} catch (Exception e) {
    throw new SQLException("SerializableDatum.toDatum: "+e.toString()); }
}
```

4. oracle.sql.RAW オブジェクトから SerializableDatum オブジェクトへのデータ変換を実装します。この段階で、データをシリアル化解除します。

```
// Constructing SerializableDatum from oracle.sql.RAW
private SerializableDatum(RAW raw) throws SQLException
{
    try {
        InputStream rawStream = new ByteArrayInputStream(raw.getBytes());
        ObjectInputStream is = new ObjectInputStream(rawStream);
        m_data = is.readObject();
        is.close();
    } catch (Exception e) {
        throw new SQLException("SerializableDatum.create: "+e.toString()); }
}
```


5. ORADDataFactory を実装します。この例では、無名クラスとして実装します。

```
// Implementation of an ORADDataFactory for SerializableDatum
new ORADDataFactory()
{
    public ORADData create(Datum d, int sqlCode) throws SQLException
    {
        if (sqlCode != _SQL_TYPECODE)
        {
            throw new SQLException
                ("SerializableDatum: invalid SQL type "+sqlCode);
        }
        return (d==null) ? null : new SerializableDatum((RAW)d);
    }
};
```

SQLJ アプリケーションの SerializableDatum

ここでは、前項で作成した SerializableDatum クラスのインスタンスをホスト変数およびイテレータ列として、SQLJ アプリケーションで使用方法を示します。

次の表定義を想定します。

```
CREATE TABLE PERSONDATA (NAME VARCHAR2(20) NOT NULL, INFO RAW(2000));
```

ホスト変数としての SerializableDatum

次に、SerializableDatum インスタンスをホスト変数として使用します。

```
...
SerializableDatum pinfo = new SerializableDatum();
pinfo.setData (
    new Object[] { "Some objects", new Integer(51), new Double(1234.27) } );
String pname = "MILLER";
#sql { INSERT INTO persondata VALUES(:pname, :pinfo) };
...
```

イテレータ列としての SerializableDatum

次に、SerializableDatum を名前指定イテレータの列として使用する例を示します。

次のように宣言します。

```
#sql iterator PersonIter (SerializableDatum info, String name);
```

実行可能コードを示します。

```
PersonIter pcur;
#sql pcur = { SELECT * FROM persondata WHERE info IS NOT NULL };
while (pcur.next())
{
    System.out.println("Name:" + pcur.name() + " Info:" + pcur.info());
}
pcur.close();
...
```

SerializableDatum（クラス全体）

ここでは、詳細手順を示した SerializableDatum クラスの全体を示します。

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

public class SerializableDatum implements ORADData
{
    // Client methods for constructing and accessing a SerializableDatum

    private Object m_data;
    public SerializableDatum()
    {
        m_data = null;
    }
    public void setData(Object data)
    {
        m_data = data;
    }
    public Object getData()
    {
        return m_data;
    }
}

// Implementation of toDatum()

public Datum toDatum(Connection c) throws SQLException
{

```

```

        try {
            ByteArrayOutputStream os = new ByteArrayOutputStream();
            ObjectOutputStream oos = new ObjectOutputStream(os);
            oos.writeObject(m_data);
            oos.close();
            return new RAW(os.toByteArray());
        } catch (Exception e) {
            throw new SQLException("SerializableDatum.toDatum: " + e.toString()); }
    }

    public static ORADDataFactory getORADDataFactory()
    {
        return FACTORY;
    }

    // Implementation of an ORADDataFactory for SerializableDatum

    private static final ORADDataFactory FACTORY =

        new ORADDataFactory()
        {
            public ORADData create(Datum d, int sqlCode) throws SQLException
            {
                if (sqlCode != _SQL_TYPECODE)
                {
                    throw new SQLException(
                        "SerializableDatum: invalid SQL type " + sqlCode);
                }
                return (d==null) ? null : new SerializableDatum((RAW)d);
            }
        };

    // Constructing SerializableDatum from oracle.sql.RAW

    private SerializableDatum(RAW raw) throws SQLException
    {
        try {
            InputStream rawStream = new ByteArrayInputStream(raw.getBytes());
            ObjectInputStream is = new ObjectInputStream(rawStream);
            m_data = is.readObject();
            is.close();
        } catch (Exception e) {
            throw new SQLException("SerializableDatum.create: " + e.toString()); }
    }

    public static final int _SQL_TYPECODE = OracleTypes.RAW;
}

```

弱い型指定のオブジェクト、参照およびコレクション

SQLJ では、弱い型指定のオブジェクト、参照およびコレクションを使用できます。一般的には、これらのクラスの使用はお薦めしません。使用方法も制限されます。ただし、便利な場合もあります。たとえば、汎用コードを記述して、どのような STRUCT または REF でも使用できます。

弱い型指定のオブジェクト、参照およびコレクションのサポート

Oracle のオブジェクト、参照またはコレクションを SQLJ アプリケーションで使用する場合は、強い型指定のカスタム・オブジェクト・クラス、参照クラスまたはコレクション・クラス（ORADData インタフェースを実装したクラス）または強い型指定のカスタム・オブジェクト・クラス（SQLData インタフェースを実装したクラス）を使用するかわりに、弱い型指定の汎用 `java.sql` インスタンスまたは `oracle.sql` インスタンスを使用できます。（ただし、SQLData 実装をカスタム・オブジェクト・クラスとして使用した場合、弱い型指定のカスタム参照インスタンスしか使用できません。）

Oracle SQLJ では、イテレータ列またはホスト式に対して、次に示した弱い型指定を使用できます。

- `java.sql.Struct` または `oracle.sql.STRUCT`（オブジェクトの場合）
- `java.sql.Ref` または `oracle.sql.REF`（オブジェクト参照の場合）
- `java.sql.Array` または `oracle.sql.ARRAY`（コレクションの場合）

ホスト式の場合は、次の方法で使用できます。

- 入力ホスト式として
- INTO リストで出力ホスト式として

こうした弱い型指定は、通常は使用しないようにしてください。SQLJ の強い型指定のパラダイムのあらゆるメリットが損なわれるためです。

STRUCT オブジェクト内の各属性または ARRAY オブジェクト内の各要素が `oracle.sql.Datum` オブジェクトに格納されます。元のデータの形式は、Datum の該当する `oracle.sql.*` サブタイプ（`oracle.sql.NUMBER` や `oracle.sql.CHAR` など）です。STRUCT オブジェクト内の属性は匿名です。

STRUCT および ARRAY クラスは汎用クラスなので、これらのクラスのインスタンスに対してオブジェクトまたはコレクションの読み込みや書き込みを行う際は、SQLJ で型チェックを行いません。

一般的には、オブジェクト、参照およびコレクションに対してカスタム Java クラスの使用をお薦めします。可能な場合は、JPublisher で生成したクラスを使用してください。

弱い型指定のオブジェクト、参照およびコレクションの使用制限

弱い型指定のオブジェクト (Struct または STRUCT インスタンス)、参照 (Ref または REF のインスタンス) またはコレクション (Array または ARRAY のインスタンス) は、ホスト式の次のパラメータとしては使用できません。

- IN パラメータとして (NULL の場合)
- ストアド・プロシージャまたはストアド・ファンクションのコールで、OUT または INOUT パラメータとして
- ストアド・ファンクションの結果式で、OUT パラメータとして

これらのパラメータとして使用できない理由は、元の SQL 型の名前 (Person など) を特定できないからです。Oracle JDBC ドライバは、元の SQL 型の名前を使用して、Java のユーザー定義型のインスタンスを生成します。

Oracle OPAQUE 型

Oracle OPAQUE 型は抽象データ型です。単なる一連のバイトとして実装されたデータの場合、内部表現は公開されません。通常、OPAQUE 型は Oracle で提供され、カスタマによる実装はありません。

OPAQUE 型は、基本的な点でオブジェクト型に類似しています。つまり、static メソッド、インスタンスおよびインスタンス・メソッドの概念が類似しています。一般的に、状態や内部バイト表現を操作できるのは、OPAQUE 型を使用して提供されるメソッドのみです。Java では、oracle.sql.OPAQUE として、または oracle.sql.ORAData インタフェースを実装しているカスタム・クラスとして、OPAQUE 型を表現できます。クライアント側では、Java コードを実装してバイトを操作できます (バイト・パターンが判明していることが前提です)。Oracle9i JPublisher ユーティリティを使用すると、このように ORAData を実装するカスタム・クラスを作成し、データベースへのラウンドトリップを繰り返さずにデータを操作できます。詳細は、『Oracle9i JPublisher ユーザーズ・ガイド』を参照してください。

OPAQUE 型の主な例は、Oracle9i で提供している XMLType です。Oracle が提供する型によって、データベースの XML データをネイティブに処理することが容易になります。

SYS.XMLType によって次の機能が提供され、Java の oracle.xdb.XMLType クラスを介して公開されます。

- 表またはビューの列のデータ型として使用できます。XMLType にはあらゆるコンテンツを格納できますが、XML コンテンツを最適に格納するように設計されています。そのインスタンスは SQL で XML ドキュメントを表現できます。
- XML コンテンツの操作を行う組込みメンバー・ファンクションを含む SQL API があります。たとえば、XMLType ファンクションを使用すると、Oracle9i データベースに格納されている XML データの作成、問合せ、抽出および索引付けができます。

- ストアド・プロシージャで、パラメータ、戻り値および変数に使用できます。
- この機能は、PL/SQL、Java および C (OCI) で提供される API を介して使用することもできます。

XMLType の詳細は、『Oracle9i XML データベース開発者ガイド - Oracle XML DB』を参照してください。

拡張言語機能

この章では、アプリケーションのコーディングで利用できる SQLJ 言語の拡張機能について説明します。SQLJ の基本機能に関する詳細は、[第 3 章「基本的な言語機能」](#)を参照してください。

次の項目について説明します。

- [接続コンテキスト](#)
- [実行コンテキスト](#)
- [SQLJ でのマルチスレッド](#)
- [イテレータ・クラスの実装と拡張機能](#)
- [詳細なトランザクション制御](#)
- [SQLJ と JDBC の関係動作](#)
- [動的 SQL のサポート](#)

接続コンテキスト

SQLJ では接続コンテキストの概念がサポートされているため、各種 SQL エンティティを使用する際に、接続を厳密に分類できます。接続コンテキストは、特定の SQL エンティティ（具体的には、表、ビュー、ストアド・プロシージャなど）に関連付けられているといえます。SQLJ では、追加の接続コンテキスト・クラスを宣言することにより、特定の SQL エンティティを使用する接続で各クラスを使用できるようになります。接続コンテキスト・クラス 1 つに対してインスタンスがいくつか存在する場合、同じ物理エンティティを使用したり、同じスキーマに接続する必要はありません。ただし、少なくとも名前およびデータ型が同じエンティティを使用します。

注意： 1 種類の SQL エンティティと 1 つの接続コンテキスト・クラスのみを使用する場合に重点を置いた、接続の基本に関する概要は、4-6 ページの「[接続の際の考慮事項](#)」を参照してください。

接続コンテキストの概要

アプリケーションで数種類の SQL エンティティを使用する場合は、通常、新たに 1 つ以上の接続コンテキスト・クラスを宣言し、使用することをお勧めします。3-2 ページの「[SQLJ 宣言の概要](#)」を参照してください。各接続コンテキスト・クラスは、相互に関連しあう特定の SQL エンティティで使用できます。つまり、特定の接続コンテキスト・クラスを使用して接続を定義すると、各接続で使用する表、ビュー、ストアド・プロシージャなどは、名前やデータ型が同じになります。

たとえば、人事部で使用する表とストアド・プロシージャのセットを、SQL エンティティの一例として取り上げます。その場合、表として EMPLOYEES と DEPARTMENTS、ストアド・プロシージャとして CHANGE_DEPT と UPDATE_HEALTH_PLAN が使用されます。他の種類の SQL エンティティとしては、財務部で使用する表とプロシージャのセットを取り上げます。この場合、表として EMPS（人事部で使用する従業員表とは異なるもの）、ストアド・プロシージャとして GIVE_RAISE と CHANGE_WITHHOLDING を使用します。

SQL エンティティへの接続コンテキスト・クラスをカスタマイズすると、オンラインのセマンティクス・チェックの精度が上がります。オンライン・チェックでは、SQLJ 文中にある SQL エンティティのうち、指定した接続コンテキスト・クラスを使用する各エンティティが、基本スキーマにある変換用 SQL エンティティと同じかどうかを検証されます。基本スキーマは、SQLJ の接続先データベース・アカウントの 1 つで、特定の接続コンテキスト・クラスを使用した SQLJ 文に対するオンライン・チェックに使用されます。トランスレータへの基本スキーマを指定するには、SQLJ コマンドラインから `-user`、`-password` および `-url` オプションを使用します。（これらのオプションの詳細は、8-32 ページの「[接続オプション](#)」を参照してください。）基本スキーマは、実行時にアプリケーションで使用するアカウントと同じにしてもしなくてもかまいません。

関連のない広範囲の SQL エンティティ・グループを複数の SQLJ 文で使用し、これらの文に接続コンテキスト・クラスを 1 つしか使用しない場合は、汎用性の高い基本スキーマを作成する必要があります。この基本スキーマに含まれるあらゆる表、ビューおよびストアド・プ

ロシージャは、すべての文にわたって使用されます。この逆に、特定の接続コンテキスト・クラスを使用した各 SQLJ 文中で、関連する小範囲の SQL エンティティのセットを使用する場合には、特定の基本スキーマを作成します。特定のスキーマを使用すると、セマンティクス・チェックの精度が向上します。

注意：

- 接続コンテキスト・クラスの宣言では、宣言済みの接続コンテキスト・クラスで使用する SQL エンティティを定義しません。また、関連のない異種のエンティティを使用した接続に、それぞれ同じ接続コンテキスト・クラスを使用できます。接続コンテキスト・クラスの使用方法は任意です。特定の接続コンテキスト・クラスで使用可能な SQL エンティティが限られる場合、その原因となるのは、基本スキーマ用エンティティ（変換時にオンライン・セマンティクス・チェックを使用する場合）と、実行時に接続先のスキーマ用エンティティ（接続コンテキスト・クラスのインスタンスを使用する場合）です。
- アプリケーションで使用する SQL 修飾名（SCOTT.EMP など）に、エンティティの常駐場所を指定した場合、基本スキーマ（オンライン・チェックを使用する場合）およびランタイム・スキーマからリソースへのアクセスには、完全修飾名を指定する必要があります。
- ベンダー各社のデータベースへの接続にも、接続コンテキスト・クラスを使用可能です。ただし、同じ名前や互換性のあるデータ型を使用して、接続先スキーマのエンティティにアクセスできる場合に限られます。

接続コンテキストのロジスティクス

接続コンテキスト・クラスを宣言すると、SQLJ トランスレータによって生成されたコードにクラスが定義されます。宣言した接続コンテキスト・クラスの他に、デフォルトの接続コンテキスト・クラスが定義されます。

```
sqlj.runtime.ref.DefaultContext
```

接続コンテキスト・インスタンスを作成する場合は、特定のスキーマ（ユーザー名、パスワード、URL）および SQL 操作が実行される特定のセッションとトランザクションを指定します。その場合、接続コンテキスト・クラスのコンストラクタへの入力パラメータとして、ユーザー名、パスワードおよびデータベース URL を指定します。接続コンテキスト・インスタンスでは、そのセッションで実行された SQL 操作が管理されます。

SQLJ 文ごとに、使用する接続コンテキスト・インスタンスを指定します。7-6 ページの「SQLJ 句での接続コンテキスト・インスタンスの指定」を参照してください。

次の例では、2つの異種スキーマに接続するための接続コンテキスト・クラス `MyContext` の基本的な宣言と使用方法を示します。通常の使用では、この両スキーマの SQL エンティティは、それぞれ名前とデータ型が同じであることが前提になります。

次のように宣言します。

```
#sql context MyContext;
```

実行可能コードを示します。

```
MyContext mctx1 = new MyContext
    ("jdbc:oracle:thin@localhost:1521:ORCL", "scott", "tiger", false);
MyContext mctx2 = new MyContext
    ("jdbc:oracle:thin@localhost:1521:ORCL", "brian", "mypasswd", false);
```

接続コンテキスト・クラスのコンストラクタでは、ブール値の自動コミット・パラメータが定義されます。詳細は、7-4 ページの「[接続コンテキスト・クラスの宣言と使用方法の補足](#)」を参照してください。

また、異なる接続コンテキスト・インスタンスで、同じスキーマに接続できます。前述の例では、`mctx1` と `mctx2` の両方には、必要に応じて `scott/tiger` を指定できます。ただし、実行時に接続コンテキスト・インスタンスがデータベースに加えた変更は、コミットされるまで別の接続コンテキスト・インスタンスには認識されません。これには例外もあります。それは、この2つの接続コンテキスト・インスタンスが、同じ JDBC 接続インスタンスから生成された場合です。(接続コンテキスト・クラスのコンストラクタのいずれか1つは、入力として JDBC 接続インスタンスを取るからです。)

接続コンテキスト・クラスの宣言と使用方法の補足

ここでは、接続コンテキスト・クラスを宣言し、クラスのインスタンスを使用してデータベース接続を定義する方法について、例を示して説明します。

接続コンテキスト・クラスのコンストラクタは、次のいずれかが指定されている場合、データベース・スキーマへの接続を開きます (`DefaultContext` クラスの場合)。

- URL (String)、ユーザー名 (String)、パスワード (String)、自動コミット (boolean)
- URL (String)、`java.util.Properties` オブジェクト、自動コミット (boolean)
- URL (ユーザー名やパスワードなど、接続の詳細を指定する String)、自動コミットの設定 (boolean)
- JDBC 接続オブジェクト (`Connection`)
- SQLJ 接続コンテキスト・オブジェクト

注意：

- JDBC 接続オブジェクトを引数とするコンストラクタを使用する場合は、接続コンテキスト・インスタンスを NULL JDBC で初期化しないでください。
- 自動コミットの設定では、SQL 操作が自動的にコミットされるかどうかは定義されます。詳細は、4-24 ページの「[基本的なトランザクション制御](#)」を参照してください。
- 接続コンテキスト・クラスがデータ・ソース with 句で宣言されている場合は、別の種類のコンストラクタが実装されます。詳細は、7-12 ページの「[標準データ・ソースのサポート](#)」を参照してください。

接続コンテキスト・クラスの宣言

次に示す宣言では、接続コンテキスト・クラスが生成されます。

```
#sql context OrderEntryCtx <implements_clause> <with_clause>;
```

これを実行すると、SQLJ トランスレータで `sqlj.runtime.ConnectionContext` インタフェースを実装したクラスが生成されます。また、`ConnectionContext` インタフェースを実装した基本クラス（抽象クラス）が拡張されます。この基本クラスは、使用している特定の SQLJ 実装の一部となります。

`implements` 句と `with` 句の使用は任意です。前者は追加して実装するインタフェースを、後者は定義および初期化する変数を指定します。3-5 ページの「[IMPLEMENTS 句宣言](#)」および 3-6 ページの「[WITH 句宣言](#)」を参照してください。データ・ソース with 句の詳細は、7-12 ページの「[標準データ・ソースのサポート](#)」を参照してください。

次は、SQLJ トランスレータによって生成される内容です（メソッドの実装は省略してあります）。

```
class OrderEntryCtx implements sqlj.runtime.ConnectionContext
    extends ...
{
    public OrderEntryCtx(String url, Properties info, boolean autocommit)
        throws SQLException {...}
    public OrderEntryCtx(String url, boolean autocommit)
        throws SQLException {...}
    public OrderEntryCtx(String url, String user, String password,
        boolean autocommit) throws SQLException {...}
    public OrderEntryCtx(Connection conn) throws SQLException {...}
    public OrderEntryCtx(ConnectionContext other) throws SQLException {...}

    public static OrderEntryCtx getDefaultContext() {...}
    public static void setDefaultContext(OrderEntryCtx ctx) {...}
}
```

接続コンテキスト・インスタンスの作成

前述の例を使用して、次の構文で OrderEntryCtx クラスをインスタンス化します。

```
OrderEntryCtx myOrderConn = new OrderEntryCtx
    (url, username, password, autocommit);
```

次にその例を示します。

```
OrderEntryCtx myOrderConn = new OrderEntryCtx
    ("jdbc:oracle:thin:@localhost:1521:orcl", "scott", "tiger", true);
```

このためには、DefaultContext クラスのインスタンス化と同じ方法を使用します。DefaultContext を含むすべての接続コンテキスト・クラスでは、それぞれコンストラクタのシグネチャが同じです。

注意：

- 通常は、JDBC ドライバを登録した後で、接続コンテキスト・インスタンスを作成する必要があります。4-5 ページの「ランタイムに使用するドライバの選択および登録」を参照してください。
 - 接続コンテキスト・クラスがデータ・ソース with 句で宣言されている場合は、別の種類のコンストラクタが実装されます。詳細は、7-12 ページの「標準データ・ソースのサポート」を参照してください。
-
-

SQLJ 句での接続コンテキスト・インスタンスの指定

次は、基本的な SQLJ 文の構文です。

```
#sql <[<conn><, ><exec>]> { SQL operation };
```

接続コンテキスト・インスタンスは、#sql トークンの後ろの大かっこ内に指定します。次の SQLJ 文の例では、前述の例で使用した myOrderConn を接続コンテキスト・インスタンスとして使用します。

```
#sql [myOrderConn] { UPDATE TAB2 SET COL1 = :w WHERE :v < COL2 };
```

DefaultContext クラスや宣言済みの接続コンテキスト・クラスのインスタンスは、このようにして指定します。

接続コンテキスト・インスタンスの終了

接続コンテキスト・インスタンスは、処理完了後にすべて終了することをお薦めします。各接続コンテキスト・クラスには、`close()` メソッドが含まれます。具体例は、4-10 ページの「[接続の終了](#)」で取り上げた `DefaultContext` クラスの例を参照してください。

接続コンテキスト・インスタンスのうち、元の接続を他の接続インスタンスと共有するものについては、元の接続を開いたままにしておく必要があります。7-52 ページの「[共有接続の終了](#)」を参照してください。

複数の接続コンテキストの例

次は、複数の接続コンテキストを使用する SQLJ アプリケーションの例です。この例では、一方の SQL エンティティには `DefaultContext` クラスのインスタンスを暗黙的に使用し、もう一方の SQL エンティティには、宣言済みの接続コンテキスト・クラス `DeptContext` のインスタンスを使用しています。

この例では、静的な `Oracle.connect()` メソッドを使用し、デフォルトの接続を確立します。その後で、静的な `Oracle.getConnection()` メソッドを使用してさらに接続を生成し、もう一方の `DefaultContext` インスタンスを `DeptContext` コンストラクタに渡します。前述のように、これは SQLJ 接続コンテキスト・インスタンスの生成方法の 1 つにすぎません。

```
import java.sql.SQLException;
import oracle.sqlj.runtime.Oracle;

// declare a new context class for obtaining departments
#sql context DeptContext;

#sql iterator Employees (String ename, int deptno);

class MultiSchemaDemo
{
    public static void main(String[] args) throws SQLException
    {
        /* if you're using a non-Oracle JDBC Driver, add a call here to
           DriverManager.registerDriver() to register your Driver
        */

        // set the default connection to the URL, user, and password
        // specified in your connect.properties file
        Oracle.connect(MultiSchemaDemo.class, "connect.properties");
```

```
// create a context for querying department info using
// a second connection
DeptContext deptCtx =
    new DeptContext(Oracle.getConnection(MultiSchemaDemo.class,
        "connect.properties"));

new MultiSchemaDemo().printEmployees(deptCtx);
deptCtx.close();
}

// performs a join on deptno field of two tables accessed from
// different connections.
void printEmployees(DeptContext deptCtx) throws SQLException
{
    // obtain the employees from the default context
    Employees emps;
    #sql emps = { SELECT ename, deptno FROM emp };

    // for each employee, obtain the department name
    // using the dept table connection context
    while (emps.next()) {
        String dname;
        int deptno = emps.deptno();
        #sql [deptCtx] {
            SELECT dname INTO :dname FROM dept WHERE deptno = :deptno
        };
        System.out.println("employee: " + emps.ename() +
            ", department: " + dname);
    }
    emps.close();
}
}
```

接続コンテキスト・クラスの実装と機能

ここでは、DefaultContext クラスなどの接続コンテキスト・クラスを SQLJ で実装する方法について説明します。また、その主なメソッドについても取り上げます。

前述のように、DefaultContext クラスとすべての生成された接続コンテキスト・クラスには、ConnectionContext インタフェースが実装されます。

注意： 接続コンテキスト・クラスのサブクラス化は SQLJ 仕様で許可されていないので、Oracle SQLJ でのサブクラス化はサポートされていません。

ConnectionContext インタフェース

各接続コンテキスト・クラスには、`sqlj.runtime.ConnectionContext` インタフェースが実装されています。

このインタフェースで指定される基本メソッドは、次のとおりです。

- `close(boolean CLOSE_CONNECTION/KEEP_CONNECTION)`: その接続をメンテナンスするために使用されているすべてのリソースを解放し、開いている接続プロファイルをすべて閉じるメソッド。基になる JDBC 接続を閉じるかどうかは `CLOSE_CONNECTION` あるいは `KEEP_CONNECTION` が指定されているかによります。これらは、`ConnectionContext` インタフェース特有の `static` ブール定数です。

詳細は、7-52 ページの「共有接続の終了」を参照してください。

- `getConnection()`: その接続コンテキスト・インスタンスの基になる JDBC 接続オブジェクトを戻すメソッド。
- `getExecutionContext()`: その接続コンテキスト・インスタンスのデフォルトの `ExecutionContext` インスタンスを戻すメソッド。詳細は、7-22 ページの「実行コンテキスト」を参照してください。

接続コンテキスト・クラスの追加メソッド

`ConnectionContext` インスタンスで定義されているメソッドの他に、次のメソッドが各接続コンテキスト・クラスで定義されています。

- `YourCtxClass getDefaultContext()`: これは `static` メソッドで、特定の接続コンテキスト・クラスにデフォルトの接続コンテキスト・インスタンスを戻します。
- `setDefaultContext(YourCtxClass connctxinstance)`: これは `static` メソッドで、クラスのデフォルトの接続コンテキスト・インスタンスとして、任意の接続コンテキスト・インスタンスを定義します。

デフォルトの接続としては、`DefaultContext` クラスのインスタンスしか使用できません。ただし、宣言された接続コンテキスト・クラスのデフォルトのコンテキストとして、`setDefaultContext()` メソッドを使用して、そのクラスのインスタンスを指定しておきます。次に、特定のクラスの `getDefaultContext()` メソッドを使用して、そのインスタンスを取得します。このようにすると、たとえば、次のように SQLJ 実行文に接続コンテキスト・インスタンスを指定できます。

次のように宣言します。

```
#sql context MyContext;
```

実行可能コードを示します。

```
...
MyContext myctx1 = new MyContext(url, user, password, autocommit);
...
MyContext.setDefaultContext(myctx1);
...
#sql [MyContext.getDefaultContext()] { SQL operations };
...
```

さらに、各接続コンテキスト・クラスでは、SQLJ 文のキャッシングを制御するメソッドを定義します。次に示す **static** メソッドです。

- `setDefaultStmtCacheSize(int)`
- `int getDefaultStmtCacheSize()`

また、次のインスタンス・メソッドがあります。

- `setStmtCacheSize(int)`
- `int getStmtCacheSize()`

デフォルトで、文のキャッシングを使用できます。詳細は、10-5 ページの「[文キャッシングのための接続コンテキスト・メソッド \(Oracle 固有コードの場合\)](#)」を参照してください。(これは、文のキャッシングの概要について説明する 10-4 ページの「[文のキャッシング](#)」の一部分です。)

接続コンテキスト宣言での IMPLEMENTS 句の使用

接続コンテキスト宣言にインタフェースを実装する場合があります。その概要と構文は、3-5 ページの「[IMPLEMENTS 句宣言](#)」を参照してください。

たとえば、インタフェースで、接続コンテキスト・クラスに定義された機能のサブセットのみを公開する場合があります。または、クラスで、`getConnection()` の機能のみ必要で、接続コンテキスト・クラスの他の機能が不要な場合もあります。

たとえば、`HasConnection` という名前のインタフェースを作成し、`getConnection()` メソッドのみを指定して、接続コンテキスト・クラスにある他のメソッドを指定しないことも可能です。その後で、接続コンテキスト・クラスを宣言し、`getConnection()` の機能のみを公開できます。その場合、宣言された接続コンテキスト・クラスの型を持つ変数ではなく、`HasConnection` 型の変数に、接続コンテキスト・インスタンスを代入します。

次のように宣言します（ここでは、`HasConnection` が `mypackage` パッケージにあるとします）。

```
#sql public context MyContext implements mypackage.HasConnection;
```


次のように、接続インスタンスをインスタンス化します。

```
HasConnection myConn = new MyContext (url, username, password, autocommit);
```

次にその例を示します。

```
HasConnection myConn = new MyContext  
    ("jdbc:oracle:thin:@localhost:1521:orcl", "scott", "tiger", true);
```

接続コンテキストのセマンティクス・チェック

SQLJ の大きな特長としては、接続の厳密な分類が挙げられます。相関する特定の SQL エンティティに対する操作では、各接続コンテキスト・クラスを使用するのが一般的です。このことは、1 つのクラスの接続インスタンスがすべて同じ物理エンティティを使用するという意味ではありませんが、これらのインスタンスは同じプロパティ（表やビューに関連付けられた名前や権限、インスタンスの行のデータ型、ストアド・プロシージャの名前や定義など）を持つエンティティを使用します。この厳密な分類により、SQLJ のセマンティクス・チェックでは、SQL 操作の使用がデータベース接続に適切であるかどうかに変換時に検証できます。

変換時にオンラインのセマンティクス・チェックを利用するには、接続コンテキスト・クラスごとにサンプル・スキーマ（該当の SQL エンティティを含む）を指定します。これらのサンプル・スキーマは、基本スキーマと総称されています。基本スキーマを指定するには、SQLJ の `-user`、`-password` および `-url` オプションを組み合わせで使用します。次に 2 つの例を示します。1 つ目は `DefaultContext` クラスの例で、2 つ目が宣言された接続コンテキスト・クラスの例です。`-user` オプションでユーザー、パスワードおよび URL を指定しています。

```
-user=scott/tiger@jdbc:oracle:oci:@  
-user@MyContext=scott/tiger@jdbc:oracle:oci:@
```

（これらの SQLJ オプションについては、8-32 ページの「[接続オプション](#)」を参照してください。）

トランスレータは、セマンティクス・チェック時に、特定の接続コンテキスト・クラスを指定された基本スキーマに接続し、次の処理を行います。

- 接続コンテキスト・クラスのインスタンスを指定するコード中の各 SQLJ 文を検証し、その SQL 操作（どの表にアクセスし、どのストアド・プロシージャを使用するのか、など）をチェックします。
- SQL 操作におけるエンティティが基本スキーマにある既存のエンティティと一致することを検証します。

ユーザーは、適切なランタイム・スキーマを基本スキーマとして選択する必要があります。具体的には、基本スキーマの表、ビュー、ストアド・ファンクションおよびストアド・プロシージャの各名前とデータ型を、SQL 操作で使用されているものと同一にし、各権限を正しく設定する必要があります。

変換時にいずれかの接続コンテキスト・クラスに適切な基本スキーマを使用できない場合は、その接続コンテキスト・クラスに SQLJ トランスレータ・オプション（`-user`、`-password`、`-url`）を指定する必要はありません。その場合、その接続コンテキスト・クラスの接続オブジェクトを指定する SQLJ 文では、可能な範囲でセマンティクス・チェックが行われます。

注意： トランスレータ・オプション設定に指定する基本スキーマでは、実行時に使用されるスキーマは指定されません。基本スキーマからトランスレータに渡されるのは、SQLJ 実行文で使用するエンティティと照合するための一組の SQL エンティティのみです。

標準データ・ソースのサポート

JDBC 2.0 拡張 API では、`DriverManager` にかわって、データ・ソースおよび JNDI を使用した JDBC 接続の取得が仕様化されました。JNDI 名を検索することによって、データベース接続を確立できます。この名前は、通常、GUI JavaBeans 配布ツールを通してインストールされる `javax.sql.DataSource` オブジェクトを通して、プログラム・ランタイムより前に特定のデータベースおよびスキーマに対してバインドされます。この名前は、ディレクトリ・サービスに名前を再バインドするのみでソース・コードを変更せずに、別の物理的接続にバインドされる場合もあります。

SQLJ では、柔軟で移植可能な方法で接続コンテキスト・インスタンスを作成するため、同じメカニズムを使用します。データ・ソースは、JDBC 2.0 拡張 API で定義したように、接続プールまたは分散トランザクション・サービスを使用して実装されます。

データ・ソースの詳細は、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。

接続コンテキストとデータ・ソースの関連付け

SQLJ では、通常、データ・ソース名が JDBC 接続の象徴名となるのと同じように、接続コンテキスト・クラスを論理的スキーマに関連付けます。データ・ソース名を接続コンテキストの宣言に追加することにより、両方のコンセプトを結合します。

```
#sql context EmpCtx with (dataSource="jdbc/EmpDB");
```

`dataSource` プロパティで宣言した接続コンテキスト・クラスには、必ず追加のコンストラクタがあります。この `EmpCtx` の例では、次のコンストラクタがあります。

- `public EmpCtx():jdbc/EmpDB` のデータ・ソースを検索し、接続を取得するためにデータ・ソースで `getConnection()` メソッドをコールします。
- `public EmpCtx(String user, String password):jdbc/EmpDB` のデータ・ソースを検索し、接続を取得するためにデータ・ソースで `getConnection(user, password)` メソッドをコールします。

- `public EmpCtx(ConnectionContext ctx):` 接続を取得するために `ctx` へ委任します。

`dataSource` プロパティで宣言した接続コンテキスト・クラスでは、`DriverManager` ベースのコンストラクタがいくつか削除されます。この `EmpCtx` の例では、次のコンストラクタが削除されます。

- `public EmpCtx(Connection conn)`
- `public EmpCtx(String url, String user, String password, boolean autoCommit)`
- `public EmpCtx(String url, boolean autoCommit)`
- `public EmpCtx(String url, java.util.Properties info, boolean autoCommit)`
- `public EmpCtx(String url, boolean autoCommit)`

データ・ソース接続の自動コミット・モード

置き換える `DriverManager` ベースのコンストラクタとは異なり、新規のデータ・ソース・ベースのコンストラクタには、明示的な自動コミット・パラメータは含まれていません。常に、データ・ソースで定義された自動コミット・モードを使用します。

データ・ソースは、配置例によってはデフォルトの自動コミット・モードを持つように設定されています。たとえば、通常、サーバーのデータ・ソースでは自動コミットをオフにして、クライアント上のデータ・ソースは自動コミットをオンにしています。ただし、特定の自動コミット設定を使用してデータ・ソースを設定することも可能です。これで、データ・ソースを特定のアプリケーションや配置例にあわせて設定できます。これを単一のデータベース / ドライバ設定のみを指定する JDBC URL と比較してください。

プログラムでは、接続コンテキスト・インスタンスの基になる JDBC 接続を使用して、既存の自動コミット設定を確認および上書きできます。

注意： 確立する接続の自動コミット状態に留意してください。

- Oracle クラスを使用する場合、明示的に指定しないと、自動コミットはオフになります。
 - `DefaultContext` または `DriverManager` 形式のコンストラクタを指定した接続コンテキストを使用する場合、自動コミット設定は、常に明示的に指定する必要があります。
 - データ・ソースのメカニズムを使用する場合、自動コミット設定が基になるデータ・ソースから継承されます。ほとんどの環境では、データ・ソース・オブジェクトは JDBC に基づいており、自動コミット・オプションはオンになっています。予期せぬ動作を防ぐために、常に自動コミット設定を確認してください。
-
-

データ・ソースとデフォルト・コンテキストの関連付け

SQLJ プログラムがデフォルトの接続コンテキストにアクセスし、かつデフォルトのコンテキストが設定されていない場合、SQLJ ランタイムは接続を確立するために、デフォルトの SQLJ データ・ソースを使用します。デフォルトの SQLJ データ・ソースは、JNDI 名 `jdbc/defaultDataSource` にバインドされます。

このメカニズムでは、デフォルトの SQLJ 接続コンテキストに対するデフォルトの JDBC 接続を定義し、インストールするための移植性のある方法を提供します。

データ・ソースのサポートの要件

プログラムからデータ・ソースを使用するには、`javax.sql.*` パッケージ、`javax.naming.*` パッケージおよび Java 環境では `InitialContext` プロバイダを提供する必要があります。後者は、SQLJ ランタイムがデータ・ソース・オブジェクトを検索できる JNDI コンテキストを取得する際に必要となります。

通常、データ・ソースは、JDK 1.2x 環境で Java 拡張クラスとともに使用するか、または J2EE 環境で使用します。ただし、JDK 1.1.x でも Java 拡張クラスとともにデータ・ソースを使用することも可能です。

Oracle で提供する SQLJ ランタイムはすべて、データ・ソースをサポートしています。ただし、`runtime12ee` ライブラリを使用する場合、`CLASSPATH` 内に常に `javax.sql.*` および `javax.naming.*` がなければ、ランタイムはロードされません。反対に、他のランタイム・ライブラリでは、リフレクションを使用して `DataSource` オブジェクトが取得されません。

SQLJ 固有のデータ・ソース

Oracle9i リリース 2 (9.2) では、Oracle SQLJ による SQLJ 固有のデータ・ソースのサポートが `runtime12ee` ライブラリで提供されます。現在、SQLJ 固有のデータ・ソースは、サーバー内部ではなく、クライアント側または中間層アプリケーションで使用できます。

SQLJ 固有のデータ・ソースは、SQLJ 接続コンテキスト・インスタンスを戻すメソッドを使用して、JDBC データ・ソース機能を拡張します。これによって、JDBC 開発者が接続を管理するように、SQLJ 開発者が接続コンテキストを管理できるようになります。一般に、SQLJ 固有のデータ・ソースの各インタフェースまたは各クラスは、対応する標準 JDBC データ・ソースのインタフェースまたは Oracle JDBC のデータ・ソース・クラスに基づいています。

ここからは、SQLJ 固有のデータ・ソースのインタフェースおよびクラスについて説明し、まとめとして使用例も示します。

sqlj.runtime.ConnectionContextFactory インタフェースは、SQLJ データ・ソース機能の基本インタフェースとして動作します。このインタフェースは、Oracle データ・ソース専用の一連のインタフェースによって実装され、接続プーリング、接続キャッシングまたは分散トランザクションなどの機能に対するサポートを追加します。

getDefaultContext() および getContext() には、接続コンテキスト・インスタンスの基になる JDBC 接続に対する接続パラメータ（自動コミット設定、ユーザー設定とパスワード設定、または 3 つの設定すべて）を指定できるシグネチャがあります。ユーザーとパスワードが未指定の場合は、接続を生成する基となるデータ・ソースからユーザーとパスワードが取得されます。自動コミット設定が未指定の場合は、基になるデータ・ソースの設定が明示的に true に指定されていないかぎり、デフォルトは false です。

- ConnectionContextFactory を実装する各 Oracle データ・ソース・インタフェースは、基本的なデータ・ソース、接続プーリング・データ・ソースまたは分散トランザクション (XA) データ・ソースなどに適した機能のメソッドを指定するために、標準 JDBC データ・ソース・インタフェースも実装しています。Oracle では、SqljDataSource、SqljConnectionPoolDataSource および SqljXADataSource の 3 つのインタフェースが実装されています。これらは sqlj.runtime パッケージに格納されており、次のように指定されています。

- `public interface SqljDataSource extends
javax.sql.DataSource, ConnectionContextFactory { }`
- `public interface SqljDataSource extends
javax.sql.ConnectionPoolDataSource, ConnectionContextFactory { }`
- `public interface SqljXADataSource extends
javax.sql.XADataSource, ConnectionContextFactory { }`

SQLJ データ・ソースのクラス

Oracle では、`OracleDataSource`、`OracleConnectionPoolDataSource`、`OracleXADataSource`、`OracleConnectionCacheImpl`、`OracleXAConnectionCacheImpl` および `OracleOCIConnectionPool` の Oracle JDBC データ・ソースの各クラスに対応する SQLJ 固有のクラスを提供しています。これらのクラスの詳細は、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。

Oracle SQLJ 固有のデータ・ソース・クラスは、`oracle.sqlj.runtime` および `oracle.sqlj.runtime.client` の 2 つのパッケージに格納されています。

`oracle.sqlj.runtime` パッケージには、次のクラスが含まれています。

- `public class OracleSqljDataSource
extends oracle.jdbc.pool.OracleDataSource
implements ConnectionContextFactory`

注意： `OracleSqljDataSource` クラスは `java.io.Serializable` インタフェースを実装します。したがって、このクラスはシリアル化可能で、`Oracle9iAS Containers for J2EE (OC4J)` などのクラスタ環境で使用できます。

- `public class OracleSqljConnectionPoolDataSource
extends oracle.jdbc.pool.OracleConnectionPoolDataSource
implements ConnectionContextFactory`
- `public abstract class OracleSqljXADataSource
extends oracle.jdbc.xa.OracleXADataSource
implements ConnectionContextFactory;`
- `public class OracleSqljConnectionCacheImpl
extends oracle.jdbc.pool.OracleConnectionCacheImpl
implements ConnectionContextFactory`

- `public class OracleSqljXAConnectionCacheImpl`
`extends oracle.jdbc.pool.OracleXAConnectonCacheImpl`
`implements ConnectionContextFactory`
- `public class OracleSqljOCIConnectionPool`
`extends oracle.jdbc.pool.OracleOCIConnectonPool`
`implements ConnectionContextFactory`

`oracle.sqlj.runtime.client` パッケージには、次のクラスが含まれています。

- `public class OracleSqljXADataSource`
`extends oracle.jdbc.xa.client.OracleXADataSource`
`implements ConnectionContextFactory`

対応する JDBC クラスのかわりに、拡張されたこれらのクラスを使用できます。これらのクラスには、`getDefaultContext()` および `getContext()` メソッドが含まれています (7-15 ページの「[SQLJ データ・ソースのインタフェース](#)」を参照してください)。これらのメソッドをコールする場合は、次の手順が発生します。

1. 現行のデータ・ソースから新しい論理 JDBC 接続が取得されます。
2. 論理接続から接続コンテキスト・インスタンスが作成され、戻されます。

例 : SQLJ データ・ソースの使用

中間層の環境で使用される場合、SQLJ 固有のデータ・ソースは、JDBC データ・ソースのように JNDI の位置にバインドされます。次の例のように、明示的にバインドできます。

```
//Initialize datasource
SqljXADataSource sqljDS = new OracleSqljXADataSource();
sqljDS.setUser("scott");
sqljDS.setPassword("tiger");
sqljDS.setServerName("myserver");
sqljDS.setDatabaseName("ORCL");
sqljDS.setDataSourceName("jdbc/OracleSqljXADS");

//Bind the datasource to JNDI
Context ctx = new InitialContext();
ctx.bind("jdbc/OracleSqljXADS");
```

中間層の Oracle9iAS Containers for J2EE (OC4J) 環境の場合、もう 1 つの方法として、`j2ee/home/config/data-sources.xml` ファイルの設定によってデータ・ソースをインスタンス化し、JNDI にバインドする方法もあります。たとえば、ファイル内の次の `<data-source>` 要素は、`OracleSqljXADataSource` インスタンスを作成し、それを JNDI の位置 `jdbc/OracleSqljXADS` にバインドします。

```
<data-source
  class="oracle.sqlj.runtime.OracleSqljXADatasource"
  name="jdbc/OracleSqljXADS"
  location="jdbc/OracleSqljXADS"
  xa-location="jdbc/OracleSqljXADS/xa"
  username="scott"
  password="tiger"
  url="jdbc:oracle:thin:@dlsun960:1521:ORCL"
/>
```

OC4J のデータ・ソースについては、『Oracle9iAS Containers for J2EE サービス・ガイド』を参照してください。

JNDI の位置にバインドされた SQLJ 固有のデータ・ソースは、接続コンテキスト・インスタンスの作成時に検索および使用されます。次のコードは、前述の <data-source> 要素の情報を使用して、接続コンテキスト・インスタンス（それぞれ DefaultContext インスタンスとユーザー宣言クラス MyCtx のインスタンス）を作成します。

```
sqlj.runtime.SqljDataSource sqljDS;
InitialContext initCtx = new InitialContext();
sqljDS = (sqlj.runtime.SqljDataSource) initCtx.lookup("jdbc/OracleSqljXADS");
// getDefaultContext
DefaultContext ctx = sqljDS.getDefaultContext();
// getContext
/* Declare MyCtx connection context class. You could optionally use a "with"
   clause to specify any desired connection parameters not available
   through the underlying data source.
*/
#sql public static context MyCtx;
MyCtx ctx = (MyCtx) sqljDS.getContext(MyCtx.class);
```

JavaServer Pages 用の SQLJ 固有の接続 JavaBeans

Oracle では、JSP ページ内部からのデータベース接続用に一連の JavaBeans が実装されました。元の Bean（oracle.jsp.dbutil パッケージの ConnBean および ConnCacheBean）については、『Oracle9iAS Containers for J2EE JSP タグ・ライブラリおよびユーティリティ・リファレンス』を参照してください。

Oracle9i リリース 2 (9.2) の Oracle SQLJ には、SQLJ JSP ページで使用する runtime12ee ライブラリのこれらの JavaBeans について、次の拡張機能が用意されています。

- oracle.sqlj.runtime.SqljConnBean
- oracle.sqlj.runtime.SqljConnCacheBean

ConnBean および ConnCacheBean には、JDBC 接続オブジェクトを戻すメソッドが含まれています。SqljConnBean および SqljConnCacheBean は、String 型の ContextClass と呼ばれる Bean プロパティをサポートし、SQLJ 接続コンテキスト・インスタンスを戻すように機能が拡張されています。

注意： `SqljConnBean` クラスは `java.io.Serializable` インタフェースを実装します。したがって、このクラスはシリアル化可能で、Oracle9iAS Containers for J2EE (OC4J) などのクラスタ環境で使用できます。

`SqljConnBean` および `SqljConnCacheBean` には、次のメソッドが用意されています。

- `void setContextClass(String contextClassName)`
- `String getContextClass()`
- `DefaultContext getDefaultContext()`
- `ConnectionContext getContext()`

`ContextClass` プロパティは、`DefaultContext` を使用していない場合に、ユーザー宣言の接続コンテキスト・クラスの名前を指定します。`setContextClass()` メソッドを介して、このプロパティを設定できます。

接続コンテキスト・インスタンスを取得するには、状況に応じて `getDefaultContext()` または `getContext()` を使用します。`getDefaultContext()` は `sqlj.runtime.ref.DefaultContext` インスタンスを返し、`getContext()` は `sqlj.runtime.ConnectionContext` インスタンスを返します。具体的には、`ContextClass` プロパティで指定されたクラスのインスタンスを返します（デフォルトでは `DefaultContext` です）。

ただし、`getDefaultContext()` メソッドと `getContext()` メソッドは、次の項に説明されているように `SqljConnBean` と `SqljConnCacheBean` では実装が異なることに注意してください。`SqljConnCacheBean` を使用したサンプル SQLJ JSP ページも示します。

SqljConnBean の動作（単純な接続の場合）

`SqljConnBean` インスタンスは、常に 1 つの論理 JDBC 接続と 1 つの SQLJ 接続コンテキスト・インスタンスのみをラッピングできます。

最初の `getDefaultContext()` または `getContext()` メソッドのコールによって、基礎となる JDBC 接続に基づいて接続コンテキスト・インスタンスが作成され、戻されます。この接続コンテキスト・インスタンスも `SqljConnBean` インスタンスに格納されます。

接続コンテキスト・インスタンスが作成および格納されると、後続の `getDefaultContext()` または `getContext()` コールの動作は、格納された接続コンテキストの型によって異なります。また、`getContext()` の場合は、次のように `ContextClass` プロパティに指定されている接続コンテキスト型にも依存します。

- 後続の `DefaultContext` コールで、格納された接続コンテキスト・インスタンスが `getDefaultContext()` インスタンスの場合、メソッドはそのインスタンスを返し続けます。

- 後続の `getDefaultContext()` コールで、格納された接続コンテキスト・インスタンスが `DefaultContext` インスタンスでない場合、メソッドは、格納されている接続コンテキスト・インスタンスをクローズし、基になる JDBC 接続を再利用して新規接続コンテキストを（前の接続コンテキスト型に関係なく）`DefaultContext` インスタンスとして作成し、戻します。これによって、新規接続コンテキスト・インスタンスが `SqljConnBean` インスタンスに格納されます。
- 後続の `getContext()` で、格納された接続コンテキスト・インスタンスが、`ContextClass` プロパティの指定と同じ型である場合、メソッドはそのインスタンスを戻し続けます。
- 後続の `getContext()` コールで、格納された接続コンテキスト・インスタンスが `ContextClass` の指定と同じ型でない場合、メソッドは、格納されている接続コンテキスト・インスタンスをクローズし、基になる JDBC 接続を再利用して新規接続コンテキスト・インスタンス（`ContextClass` プロパティに指定されているインスタンス）を作成し、戻します。これによって、新規接続コンテキスト・インスタンスが `SqljConnBean` インスタンスに格納されます。

注意： `SqljConnBean` で接続コンテキスト・インスタンスをクローズする場合、`KEEP_CONNECTION` 設定を使用してクローズを行うと、基になる JDBC 接続はそのままとなります。詳細は、7-52 ページの「[共有接続の終了](#)」を参照してください。

SqljConnCacheBean の動作（接続キャッシング）

`SqljConnBean` とは異なり、`SqljConnCacheBean` JavaBean は、`getDefaultContext()` または `getContext()` の各呼出しに対して、新規論理 JDBC 接続に基づいて新規接続コンテキスト・インスタンスを作成し、戻します。接続コンテキスト型は、`getDefaultContext()` コールの場合は `DefaultContext`、`getContext()` コールの場合は `ContextClass` プロパティに指定されている型になります。

`SqljConnCacheBean` では、作成した接続コンテキスト・インスタンスを格納しません。

例：SqljConnCacheBean を使用する SQLJ JSP ページ

次のプログラム `SQLJSelectInto.sqljsp` は、`SqljConnCacheBean` と、その `ContextClass` Bean プロパティおよびその `getContext()` メソッドを使用します。

注意： この例では、説明のために `ContextClass` プロパティを使用しています。ただし、`DefaultContext` がデフォルト値です。`DefaultContext` を使用し、`getContext()` ではなく `getDefaultContext()` を使用する場合、`ContextClass` は不適切な値となります。

```

<%@ page language="sqlj"
    import="java.sql.*, oracle.sqlj.runtime.SqljConnCacheBean" %>
<jsp:useBean id="cbean" class="oracle.sqlj.runtime.SqljConnCacheBean"
    scope="session">
    <jsp:setProperty name="cbean" property="User" value="scott"/>
    <jsp:setProperty name="cbean" property="Password" value="tiger"/>
    <jsp:setProperty name="cbean" property="URL"
        value="jdbc:oracle:thin:@pdcsun-dev3:1521:view13"/>
    <jsp:setProperty name="cbean" property="ContextClass"
        value="sqlj.runtime.ref.DefaultContext"/>
</jsp:useBean>
<HTML>
<HEAD> <TITLE> The SQLJSelectInto JSP </TITLE> </HEAD>
<BODY BGCOLOR=white>
<% String empno = request.getParameter("empno");
    if (empno != null) { %>
        <H3> Employee # <%=empno %> Details: </H3>
        <% String ename = null; double sal = 0.0; String hireDate = null;
            StringBuffer sb = new StringBuffer();
            sqlj.runtime.ref.DefaultContext ctx=null;
            try {
                // Make the Connection
                ctx = (sqlj.runtime.ref.DefaultContext) cbean.getContext();
            } catch (SQLException e) {
            }
            try {
                #sql [ctx] { SELECT ename, sal, TO_CHAR(hiredate, 'DD-MON-YYYY')
                            INTO :ename, :sal, :hireDate
                            FROM scott.emp WHERE UPPER(empno) = UPPER(:empno)
                };
                sb.append("<BLOCKQUOTE><BIG><B><PRE>\n");
                sb.append("Name      : " + ename + "\n");
                sb.append("Salary    : " + sal + "\n");
                sb.append("Date hired : " + hireDate);
                sb.append("</PRE></B></BIG></BLOCKQUOTE>");
            } catch (java.sql.SQLException e) {
                sb.append("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
            } finally {
                if (ctx!= null) ctx.close();
            }
            %>
        <H3><%=sb.toString()%></H3>
    <%}
    %>
    <B>Enter an employee number:</B>
    <FORM METHOD=get>
    <INPUT TYPE="text" NAME="empno" SIZE=10>

```

```
<INPUT TYPE="submit" VALUE="Ask Oracle");  
</FORM>  
</BODY>  
</HTML>
```

実行コンテキスト

実行コンテキストは `sqlj.runtime.ExecutionContext` クラスのインスタンスで、SQL 操作が実行されるコンテキストを提供します。実行コンテキスト・インスタンスは、SQLJ アプリケーションの各 SQL 操作に暗黙的にまたは明示的に対応付けられています。

`ExecutionContext` クラスには、次の機能に対するメソッドが含まれています。

- 実行制御操作は、後続の SQL 操作のセマンティクスを変更します。
- 実行ステータス操作は、最新の SQL 操作の結果を示します。
- 実行取消し操作は、現在実行している SQL 操作を終了します。
- バッチ更新操作は、バッチの有効化と無効化、バッチ制限の設定、および更新カウンターの取得を行います。（バッチ更新の詳細は、10-11 ページの「[バッチ更新機能](#)」を参照してください。）
- セーブポイント操作は、セーブポイントの設定、セーブポイントのロールバック、およびセーブポイントの解放を行います。
- 閉包操作は、実行コンテキスト・インスタンスをクローズしてリソースのリークを回避します。

注意： 実行コンテキスト・クラスは1つのみです。この点が、必要に応じて追加クラスを宣言する接続コンテキスト・クラスと異なります。すべての実行コンテキストは、`ExecutionContext` クラスのインスタンスです。接続コンテキストという用語は、通常、宣言されたクラスを示し、実行コンテキストという用語は、常に `ExecutionContext` クラスのインスタンスを示します。このマニュアルでは、接続コンテキスト・クラス、接続コンテキスト・インスタンスおよび実行コンテキスト・インスタンスを使い分けています。

実行コンテキストと接続コンテキストとの関係

各接続コンテキスト・インスタンスには、暗黙的にデフォルトの実行コンテキスト・インスタンスが対応付けられています。このインスタンスを取得するには、接続コンテキスト・インスタンスの `getExecutionContext()` メソッドを使用します。

1 つの接続コンテキスト・インスタンスに必要な実行コンテキスト・インスタンスは、1 つのみです。ただし、次の場合を除きます。

- 1 つの接続コンテキスト・インスタンスで複数のスレッドを使用している場合
マルチスレッドを使用している場合は、各スレッドにその実行コンテキスト・インスタンスが必要です。
- 同じコンテキスト・インスタンスを使用する複数の SQLJ 文で、SQL 実行制御の異なる操作を指定する場合
- 同じ接続コンテキスト・インスタンスを使用する複数の SQL 操作から、異なる SQL のステータス情報を保持する場合
同じ実行コンテキスト・インスタンスを使用する一連の SQL 操作を実行すると、各操作からのステータス情報によって前の操作からのステータス情報が上書きされます。

実行コンテキスト・インスタンスは、接続コンテキスト・インスタンスに関連付けされているように見える場合（各接続コンテキスト・インスタンスごとにデフォルトの実行コンテキスト・インスタンスがあり、特定の SQLJ 文に対して、接続コンテキスト・インスタンスと実行コンテキスト・インスタンスをともに指定できる場合）がありますが、実際には独立して操作しています。同じ接続コンテキスト・インスタンスを使用する文に、異なる実行コンテキスト・インスタンスを使用できます。また、この逆も可能です。

たとえば、マルチスレッドによって、スレッドごとに別の実行コンテキスト・インスタンスを使用している場合、1 つの接続コンテキスト・インスタンスで複数の実行コンテキスト・インスタンスを使用すると便利です。また、シングル・スレッド方式のプログラムを使用し、同じ SQL 制御パラメータをすべての接続コンテキスト・インスタンスに適用する場合は、1 つの明示的な実行コンテキスト・インスタンスで複数の接続コンテキスト・インスタンスを使用できます。（SQL 制御の設定は、7-25 ページの「[実行コンテキストのメソッド](#)」を参照してください。）

複数の実行コンテキスト・インスタンスを 1 つの設定コンテキスト・インスタンスで使用するには、`ExecutionContext` クラスのインスタンスを追加し、SQLJ 文で正しく指定する必要があります。

実行コンテキスト・インスタンスの作成と指定

デフォルト以外の実行コンテキスト・インスタンスを特定の接続コンテキスト・インスタンスで使用するには、別の実行コンテキスト・インスタンスを作成する必要があります。`ExecutionContext` コンストラクタには、入力パラメータがありません。

```
ExecutionContext myExecCtx = new ExecutionContext();
```

次に、この実行コンテキスト・インスタンスを特定の `SQLJ` 文で指定します。これは、接続コンテキスト・インスタンスを指定する方法と同じです。一般的な構文は、次のとおりです。

```
#sql [<conn_context><, ><exec_context>] { SQL operation };
```

たとえば、接続コンテキスト・クラスの `MyConnCtxClass` を宣言およびインスタンス化して、インスタンス `myConnCtx` を作成した場合は、次の文を使用することもできます。

```
#sql [myConnCtx, myExecCtx] { DELETE FROM emp WHERE sal > 30000 };
```

`myConnCtx` に別の接続コンテキスト・インスタンスを使用したり、`myExecCtx` に別の実行コンテキスト・インスタンスを使用できます。

次のように、デフォルトの接続コンテキスト・インスタンスを使用して、実行コンテキスト・インスタンスを指定することも可能です。

```
#sql [myExecCtx] { DELETE FROM emp WHERE sal > 30000 };
```

注意：

- 実行コンテキスト・インスタンスを指定しないで接続コンテキスト・インスタンスを指定した場合は、その接続コンテキスト・インスタンスのデフォルトの実行コンテキスト・インスタンスが使用されます。
 - 接続コンテキスト・インスタンスを指定しないで実行コンテキスト・インスタンスを指定した場合は、その実行コンテキスト・インスタンスには、アプリケーションのデフォルトの接続コンテキスト・インスタンスが使用されます。
 - 接続コンテキスト・インスタンスと実行コンテキスト・インスタンスの両方を指定しなかった場合は、`SQLJ` ではデフォルトの接続コンテキスト・インスタンスとデフォルトの実行コンテキスト・インスタンスが使用されます。
-
-

実行コンテキストの同期

`ExecutionContext` メソッド (7-25 ページの「[実行コンテキストのメソッド](#)」を参照) は、すべて `synchronized` メソッドです。そのため、ISO 標準コード生成の場合は、ある文がすでに使用されている実行コンテキスト・インスタンスの使用を試みると (実際に使用するの **は** 実行コンテキスト・インスタンスのメソッド)、最初の文の処理が終了するまで次の文がブロックされます。

これについては、クライアント・アプリケーションでのマルチスレッド化の状況などが関係します。別のスレッドで使用されている実行コンテキスト・インスタンスの使用を試みたスレッドは、ブロックされます。

このようなブロックを避けるには、7-31 ページの「[SQLJ でのマルチスレッド](#)」で説明しているように、使用する各スレッドに対して、別の実行コンテキスト・インスタンスを指定する必要があります。

ただし、前述の説明は、デフォルトの Oracle 固有コード生成 (`-codegen=oracle`) には適用されません。パフォーマンス上の理由から、SQLJ では、Oracle 固有生成コードの `ExecutionContext` インスタンスに対する同期が再度実行されません。そのため、必ず、1 つの実行コンテキスト・インスタンスを複数のスレッドで使用しないようにする必要があります。複数のスレッドで同じ実行コンテキストを使用すると、ブロック化ではなくてアプリケーションで、不正な結果や `NullPointerException` 例外などのエラーが発生します。

もう 1 つの例外はサーバーでのみ発生する再帰型の場合です。再帰的コールの場合、1 つのスレッド内の複数の SQLJ 文は、同時に同じ実行コンテキスト・インスタンスを使用できます。たとえば、SQLJ ストアド・プロシージャまたはストアド・ファンクションに、別の SQLJ ストアド・プロシージャまたはストアド・ファンクションへのコールが含まれている場合などです。この両方のストアド・プロシージャでデフォルトの実行コンテキスト・インスタンスが使用されている場合、最初のプロシージャでコールされる SQLJ 文でその実行コンテキストが使用されている間も、2 番目のプロシージャの SQLJ 文でその実行コンテキストが使用されます。このように、同時に同じ実行コンテキストを使用することは可能です。詳細は、11-23 ページの「[サーバーでの再帰的 SQLJ コール](#)」を参照してください。

実行コンテキストのメソッド

ここでは、`ExecutionContext` クラスのメソッドについて説明します。これらのメソッドは、状態メソッド、制御メソッド、取消しメソッド、バッチ更新メソッド、セーブポイント・メソッドおよび閉包メソッドに分類できます。

状態メソッド

次に示す実行コンテキスト・インスタンスのメソッドを使用すると、そのインスタンスを使用して完了した最新の SQL 操作に関するステータス情報を取得できます。

- `SQLWarning getWarnings(): java.sql.SQLWarning` オブジェクトを戻すメソッド。このオブジェクトには、この実行コンテキスト・インスタンスを使用した最後の SQL 操作から通知された最初の警告が含まれています。警告は連鎖形式で戻されるの

で、最初の警告を得るには実行コンテキスト・インスタンスの `getWarnings()` メソッドを使用し、次の警告を得るには各 `SQLWarning` オブジェクトの `getNextWarning()` メソッドを使用します。この警告の連鎖には、SQL 操作の実行中に生成されたあらゆる警告が含まれています。

- `int getUpdateCount()`: バッチ更新が有効化されている場合を除き、この実行コンテキスト・インスタンスを使用した最後の SQL 操作での更新行数を示す `int` 値を返すメソッド。最後の SQL 操作が DML 文以外なら、戻り値は 0 (ゼロ) です。最後の SQL 操作でイテレータまたは結果セットが生成された場合は、定数 `QUERY_COUNT` を返します。実行が終了する前に最後の SQL 操作が終了した場合は、定数 `EXCEPTION_COUNT` を返します。または、この実行コンテキスト・インスタンスを使用した操作が試みられていない場合も、この定数を返します。

バッチ可能なアプリケーションの場合、`getUpdateCount()` からの戻り値は、バッチ関連の定数値 (`NEW_BATCH_COUNT`、`ADD_BATCH_COUNT` または `EXEC_BATCH_COUNT`) のいずれかに該当します。詳細は、10-17 ページの「[実行コンテキストの更新カウント](#)」を参照してください。

制御メソッド

次に示す実行コンテキスト・インスタンスのメソッドを使用すると、そのインスタンスを使用して実行される以降の SQL 操作（まだ開始されていない操作）を制御できます。

- `int getMaxFieldSize()`: この実行コンテキスト・インスタンスを使用した SQL 操作から戻される最大データ量（バイト単位）を示す `int` 値を返すメソッド。このメソッドが適用されるのは、`BINARY`、`VARBINARY`、`LONGVARBINARY`、`CHAR`、`VARCHAR` または `LONGVARCHAR` 型の列のみです。

デフォルトでは、このパラメータは 0 (ゼロ) に設定されます。これは、サイズに上限がないことを示します。

- `setMaxFieldSize(int)`: `int` 値を入力パラメータとして取り、フィールド・サイズの上限を変更するメソッド。
- `int getMaxRows()`: この実行コンテキスト・インスタンスを使用して作成された JDBC 結果セットまたは SQLJ イテレータに格納できる最大行数を示す `int` 値を返すメソッド。上限を超えた場合、エラー・メッセージや警告は出力されずに、超えた分の行が削除されます。

デフォルトでは、このパラメータは 0 (ゼロ) に設定されます。これは、行数に上限がないことを示します。

- `setMaxRows(int)`: `int` 値を入力パラメータとして取り、最大行数を変更するメソッド。

- `int getQueryTimeout()`: この実行コンテキスト・インスタンスを使用した SQL 操作のタイムアウトの期限 (秒単位) を示す `int` 値を返すメソッド。SQL 操作がこの期限を過ぎた場合、SQL 例外が発生します。

デフォルトでは、このパラメータは 0 (ゼロ) に設定されます。これは、問合せにタイムアウトの期限がないことを示します。

- `setQueryTimeout(int)`: `int` 値を入力パラメータとして取り、問合せのタイムアウトの期限を変更するメソッド。
- `int getFetchSize()`: この `ExecutionContext` オブジェクトから生成されたイテレータ・オブジェクトに対して既存のフェッチ・サイズである行数を取得するメソッド。この `ExecutionContext` オブジェクトに、`setFetchSize()` をコールしてフェッチ・サイズを設定する作業が行われていない場合、戻り値は 0 (ゼロ) となります。`setFetchSize()` メソッドをコールすることによって、この `ExecutionContext` オブジェクトにより正のフェッチ・サイズが設定された場合、戻り値は `setFetchSize()` で指定したフェッチ・サイズとなります。
- `setFetchSize(int)`: より多くの行が必要な場合にフェッチされる行数に関して、SQLJ ランタイムにヒントを与えるメソッド。指定した行数は、この `ExecutionContext` オブジェクトを使用して作成したイテレータにのみ影響します。0 (ゼロ) を指定すると、実装に依存したデフォルト値がフェッチ・サイズとして使用されます。
- `int getFetchDirection()`: フェッチしているデータのデフォルトの方向と、この `ExecutionContext` オブジェクトから生成されるスクロール可能なイテレータ・オブジェクトのデフォルトの方向を取得するメソッド。この `ExecutionContext` オブジェクトに `setFetchDirection()` をコールすることによってフェッチ方向が設定されていない場合、戻り値は `FETCH_FORWARD` となります。
- `setFetchDirection(int)`: スクロール可能なイテレータ・オブジェクトの行が処理される方向に関して、SQLJ ランタイムにヒントを与えるメソッド。ヒントは、この `ExecutionContext` オブジェクトを使用して作成されるスクロール可能なイテレータ・オブジェクトにのみ適用されます。デフォルト値は次のとおりです。

```
sqlj.runtime.ResultSetIterator.FETCH_FORWARD.
```

このメソッドは、与えられた方向が `FETCH_FORWARD`、`FETCH_REVERSE` または `FETCH_UNKNOWN` のいずれでもない場合、`SQLException` を発生させます (`int` 定数)。

取消しメソッド

マルチスレッド環境で SQL 操作を取り消す場合や、バッチ更新が有効化されているときに保留文を一括で取り消す場合には、次のメソッドを使用します。

- `cancel()`: マルチスレッド環境において、あるスレッドで実行中の SQL 操作を取り消すには、このメソッドを別のスレッドで操作します。このメソッドを使用すると、この実行コンテキスト・インスタンスを使用して開始した未完了の最終操作が取り消されま

す。この実行コンテキスト・インスタンスを使用した実行中の文が存在しない場合、このメソッドは機能しません。

バッチ可能な環境で保留文を一括で取り消すには、このメソッドを使用します。バッチが空にされて、バッチ内の文は実行されません。既存のバッチを取り消した場合、次に出現したバッチ可能な文は、新規のバッチに追加されます。（詳細は、10-16 ページの「[バッチの取消し](#)」で説明しています。）

バッチ更新メソッド

アプリケーションでパフォーマンス強化の機能を使用する場合は、次のメソッドを使用してバッチ更新を制御します。これらのメソッドおよび更新バッチ機能全般の詳細は、10-11 ページの「[バッチ更新機能](#)」を参照してください。

- `setBatching(boolean)`: バッチ更新を有効化するためのブール値を取るメソッド。詳細は、10-13 ページの「[バッチ更新機能の有効化 / 無効化](#)」を参照してください。

バッチ更新機能は、デフォルトでは無効になります。

- `boolean isBatching()`: バッチ更新の実行可否を示すブール値を戻すメソッド。

これは、現在未完了のバッチがあるかどうかを示しているものではありません。バッチが新規に作成、追加または実行されたかどうかを確認するには、7-25 ページの「[状態メソッド](#)」で説明されている `getUpdateCount()` メソッドを使用してください。

- `int getBatchLimit()`: その時点でのバッチ制限を示す `int` 値を戻すメソッド。バッチ制限がある場合、一括されている文の数が上限に達したバッチは、暗黙的に実行されます。詳細は、10-18 ページの「[バッチ制限の設定](#)」を参照してください。

デフォルトのバッチ制限は、`ExecutionContext` `static` 定数値 `UNLIMITED_BATCH` (バッチ制限なし) に設定されます。

- `setBatchLimit(int)`: 現行のバッチ制限を設定するための入力として、0 (ゼロ) でない正の `int` 値を取るメソッド。入力可能な値としては、`UNLIMITED_BATCH` (制限なし) と、`AUTO_BATCH` (バッチ制限は `SQLJ` ランタイムで動的に判定可能) の 2 つがあります。
- `int[] executeBatch()`: 保留文を一括実行し、`int` 型の更新カウン트의配列を戻すメソッド。この更新カウン트의意味は、10-17 ページの「[実行コンテキストの更新カウント](#)」で解説します。詳細は、10-14 ページの「[明示的および暗黙的なバッチ実行](#)」を参照してください。エラー状態は、10-21 ページの「[バッチ実行中のエラー状態](#)」を参照してください。
- `int[] getBatchUpdateCounts()`: 最後に実行されたバッチの更新カウンートを示す `int` 型の配列を戻すメソッド。この更新カウンートの意味は、10-17 ページの「[実行コンテキストの更新カウント](#)」で解説します。このメソッドは、バッチを暗黙的に実行する場合に有効です。

セーブポイント・メソッド

Oracle9i リリース 2 (9.2) から、Oracle SQLJ は JDBC 3.0 セーブポイントをサポートします。SQLJ のセーブポイント文については、4-28 ページの「[セーブポイントの使用](#)」を参照してください。セーブポイントは `ExecutionContext` インスタンスに格納され、次のメソッドが用意されています。

- `setSavepoint (String)`
トランザクション内の現在の位置にセーブポイント名を指定します。
- `rollbackToSavepoint (String)`
指定されているセーブポイントまでの変更をロールバックします。
- `releaseSavepoint (String)`
指定されているセーブポイントを解放します（実行コンテキストからセーブポイントを削除します）。

ただし、コードを記述する場合、一般的にはこれらのメソッドではなくセーブポイント文が使用されます。

注意： Oracle9i リリース 2 (9.2) の時点で、Oracle9i および Oracle9i JDBC は、セーブポイントの解放機能をサポートしていません。

閉包メソッド

Oracle SQLJ では、`ExecutionContext` クラスの `close ()` メソッドに拡張機能が用意されています。

- `close ()`: リソースのリークを回避するには、次の一連の状況でこのメソッドを使用します。
 - Oracle 固有コード生成（デフォルトの `-codegen=oracle` 設定）を使用している場合。

および

- 接続コンテキスト・インスタンスを介して使用可能なデフォルトのインスタンスを使用するのではなく、`ExecutionContext` インスタンスを明示的に作成して使用している場合。

および

- `ExecutionContext` インスタンスを使用して SQLJ のロールバック文またはコミット文を明示的に発行していない場合。

```
#sql [ec] { COMMIT };  
#sql [ec] { ROLLBACK };
```

および

- ExecutionContext インスタンスで `executeBatch()` をコールしていない場合。

このような状況下では、ExecutionContext インスタンス上でバッチ可能文がオープンしたままとなる可能性があり、そのうちにデータベース・カーソルを使い果たすることがあります。これを回避するには、次の例のように `close()` メソッドを使用します。

```
ExecutionContext ec = new ExecutionContext();
...
try {
    ...
    #sql [ec] { SQL operation };
    ...
} finally { ec.close(); }
```

注意：（前述のように明示的に宣言されるかわりに）実行コンテキスト・インスタンスが接続コンテキスト・インスタンスに関連付けられている場合は、（基になる JDBC 接続のクローズと一緒に、または別に）接続コンテキスト・インスタンスをクローズすると、実行コンテキスト・インスタンス上に残っているすべての文が自動的にクローズされます。

例：ExecutionContext メソッドの使用

次のコードは、ExecutionContext メソッドの使用方法の一例です。

```
ExecutionContext execCtx =
    DefaultContext.getDefaultContext().getExecutionContext();

// Wait only 3 seconds for operations to complete
execCtx.setQueryTimeout(3);

// delete using execution context of default connection context
#sql { DELETE FROM emp WHERE sal > 10000 };

System.out.println
    ("removed " + execCtx.getUpdateCount() + " employees");
```

実行コンテキストとマルチスレッドの関係

1つの実行コンテキストで複数のスレッドを使用しないでください。複数のスレッドを使用した場合に、2つのSQLJ文で同じ実行コンテキストが同時に使用されると、最初の文の処理が完了するまで2番目の文がブロックされます。また、最初の操作からのステータス情報が、その情報が取り出される前に上書きされることがあります。

そのため、1つの接続コンテキスト・インスタンスで複数のスレッドを使用している場合は、次の手順で操作を行ってください。

1. スレッドごとに一意の実行コンテキスト・インスタンスをインスタンス化します。
2. #sql 文で実行コンテキストを指定し、各スレッドで該当の実行コンテキストを使用するようにします。(7-24 ページの「[実行コンテキスト・インスタンスの作成と指定](#)」を参照してください。)

スレッドごとに別の接続コンテキスト・インスタンスを使用している場合、実行コンテキスト・インスタンスをインスタンス化して、指定する必要はありません。接続コンテキスト・インスタンスごとに、そのインスタンスに対するデフォルトの実行コンテキスト・インスタンスが暗黙的に対応付けられているためです。

マルチスレッドの詳細は、7-31 ページの「[SQLJ でのマルチスレッド](#)」を参照してください。

注意： パフォーマンス上の理由から、SQLJ では、Oracle 固有生成コードの `ExecutionContext` インスタンスに対する同期が再度実行されません。そのため、必ず、1つの実行コンテキスト・インスタンスを複数のスレッドで使用しないようにする必要があります。複数のスレッドで同じ実行コンテキストを使用すると、ブロック化ではなくてアプリケーションで、不正な結果や `NullPointerException` 例外などのエラーが発生します。

SQLJ でのマルチスレッド

ここでは、SQLJ のサポート、マルチスレッドの要件、マルチスレッドと実行コンテキスト・インスタンスとの関係について説明します。

SQLJ を使用して、マルチスレッド・アプリケーションを記述できます。ただし、SQLJ アプリケーションでマルチスレッドを使用すると、JDBC ドライバまたは専用のデータベース・アクセス用のデバイスになんらかの制限が発生します。この制限には、同期に関する制限も含まれています。

スレッドごとに異なる実行コンテキスト・インスタンスを使用してください。それには、次の2通りの方法があります。

- SQLJ 文に接続コンテキスト・インスタンスを指定し、スレッドごとに異なる接続コンテキスト・インスタンスが使用されるようにします。接続コンテキスト・インスタンスごとに、デフォルトの実行コンテキスト・インスタンスが自動的に対応付けられます。

- 同じ接続コンテキスト・インスタンスで複数のスレッドを使用している場合は、実行コンテキスト・インスタンスを追加して宣言します。次に、スレッドごとに異なる実行コンテキスト・インスタンスが使用されるように、SQLJ 文で実行コンテキスト・インスタンスを指定します。

SQLJ 文で接続コンテキスト・インスタンスと実行コンテキスト・インスタンスを指定する方法は、3-11 ページの「[接続コンテキスト・インスタンスと実行コンテキスト・インスタンスの指定](#)」を参照してください。

いずれかの Oracle JDBC ドライバを使用している場合は、必要に応じて、同じ接続コンテキスト・インスタンスを複数のスレッドで使用できます（ただし、異なる実行コンテキスト・インスタンスが指定されていることが必要です）。また、ユーザーが直接確認できるような同期は必要ありません。ただし、データ・アクセスは逐次行われます。データには一度に 1 つのスレッドしかアクセスできません。同期とは、スレッドを通して実行される SQL 操作の各実行段階の制御の流れを示します。たとえば、各文では入力パラメータがバインドされて、文が実行されます。その後で、出力パラメータがバインドされます。一部の JDBC ドライバでは、これらの処理段階が混在されないように注意する必要があります。

ISO 標準コード生成の場合、あるスレッドが、別の操作で使用されている実行コンテキストを使用する SQL 操作を実行すると、現行の操作が完了するまでスレッドがブロックされます。実行コンテキストが 2 つのスレッド間で共有されている場合、一方のスレッドで実行された SQL 操作の結果は、もう一方のスレッドで認識できます。両方のスレッドが SQL 操作を実行している場合、競合状態が発生する可能性があります。つまり、一方のスレッドが元の結果を処理する前に、そのスレッドの実行結果がもう一方のスレッドの実行結果で上書きされる場合があります。このため、複数のスレッドが同じ実行コンテキスト・インスタンスを共有することは不可能です。

重要： 前述の説明は、デフォルトの Oracle 固有コード生成（`-codegen=oracle`）には適用されません。パフォーマンス上の理由から、SQLJ では、Oracle 固有生成コードの `ExecutionContext` インスタンスに対する同期が追加実行されません。そのため、必ず、1 つの実行コンテキスト・インスタンスを複数のスレッドで使用しないようにする必要があります。複数のスレッドで同じ実行コンテキストを使用すると、ブロック化ではなくてアプリケーションで、不正な結果や `NullPointerException` 例外などのエラーが発生します。

マルチスレッド : MultiThreadDemo.sqlj 次は、マルチスレッドを使用した SQLJ アプリケーションの例です。

データが永続的に変更されるのを防ぐために、ROLLBACK 操作の実行後に接続を終了します。

```
import java.sql.SQLException;
import java.util.Random;
import sqlj.runtime.ExecutionContext;
import oracle.sqlj.runtime.Oracle;
/**
 * Each instance of MultiThreadDemo is a thread that gives all employees
 * a raise of some ammount when run. The main program creates two such
 * instances and computes the net raise after both threads have completed.
 */
class MultiThreadDemo extends Thread
{
    double raise;
    static Random randomizer = new Random();

    public static void main (String args[])
    {
        try {
            /* if you're using a non-Oracle JDBC Driver, add a call here to
             DriverManager.registerDriver() to register your Driver
            */
            // set the default connection to the URL, user, and password
            // specified in your connect.properties file
            Oracle.connect(MultiThreadDemo.class, "connect.properties");
            double avgStart = calcAvgSal();
            MultiThreadDemo t1 = new MultiThreadDemo(250.50);
            MultiThreadDemo t2 = new MultiThreadDemo(150.50);
            t1.start();
            t2.start();
            t1.join();
            t2.join();
            double avgEnd = calcAvgSal();
            System.out.println("average salary change: " + (avgEnd - avgStart));
        } catch (Exception e) {
            System.err.println("Error running the example: " + e);
        }
        try { #sql { ROLLBACK }; Oracle.close(); } catch (SQLException e) { }
    }
    static double calcAvgSal() throws SQLException
    {
        double avg;
        #sql { SELECT AVG(sal) INTO :avg FROM emp };
        return avg;
    }
}
```

```
}
MultiThreadDemo(double raise)
{
    this.raise = raise;
}
public void run()
{
    // Since all threads will be using the same default connection
    // context, each run uses an explicit execution context instance to
    // avoid conflict during execution
    try {
        delay();
        ExecutionContext execCtx = new ExecutionContext();
        #sql [execCtx] { UPDATE EMP SET sal = sal + :raise };
        int updateCount = execCtx.getUpdateCount();
        System.out.println("Gave raise of " + raise + " to " +
                           updateCount + " employees");
    } catch (SQLException e) {
        System.err.println("error updating employees: " + e);
    }
}
// delay is used to introduce some randomness into the execution order
private void delay()
{
    try {
        sleep((long)Math.abs(randomizer.nextInt())/10000000);
    } catch (InterruptedException e) {}
}
}
```

イテレータ・クラスの実装と拡張機能

ここでは、イテレータ・クラスの実装方法について説明します。また、3-38 ページの「[名前指定イテレータの使用](#)方法」と 3-43 ページの「[位置指定イテレータの使用](#)」で説明した基本メソッドの他に、これらのクラスで提供される追加機能について取り上げます。この章では、次の項目について説明します。

- [イテレータ・クラスの実装と機能](#)
- [イテレータ宣言での IMPLEMENTS 句の使用](#)
- [イテレータ・クラスのサブクラス化のサポート](#)
- [結果セット・イテレータ](#)
- [スクロール可能なイテレータ](#)

イテレータ・クラスの実装と機能

宣言した名前指定イテレータ・クラスは、SQLJ トランスレータによって生成され、`sqlj.runtime.NamedIterator` インタフェースが実装されます。`NamedIterator` インタフェースを実装しているクラスには、(位置ではなく) 名前で、イテレータ列をデータベース列にマッピングする機能があります。

宣言した位置指定イテレータ・クラスは、SQLJ トランスレータによって生成され、`sqlj.runtime.PositionedIterator` インタフェースが実装されます。`PositionedIterator` インタフェースを実装しているクラスには、(名前ではなく) 位置で、イテレータ列をデータベース列にマッピングする機能があります。

`NamedIterator` インタフェースと `PositionedIterator` インタフェースの両方、そしてすべての生成された SQLJ イテレータ・クラスは、`sqlj.runtime.ResultSetIterator` インタフェースを実装または拡張します。

すべての SQLJ イテレータ (名前指定イテレータと位置指定イテレータの両方) では、`ResultSetIterator` インタフェースで次のメソッドが指定されます。

- `close()`: イテレータを終了するメソッド
- `ResultSet getResultSet()`: 基になる JDBC 結果セットをイテレータから抽出するメソッド
- `boolean isClosed()`: イテレータが終了しているかどうかを判定するメソッド
- `boolean next()`: イテレータの次の行に移動するメソッド (移動できる有効な行がある場合、`true` が戻り値)

`PositionedIterator` インタフェースは、次のメソッド仕様部を位置指定イテレータに追加します。

- `boolean endFetch()`: 位置指定イテレータの最後の行に到達したかどうかを判定するメソッド

3-38 ページの「[名前指定イテレータの使用](#)方法」で説明したように、`next()` メソッドを使用して名前指定イテレータの行を移動し、アクセッサ・メソッドでデータを取得します。SQLJ 生成の名前指定イテレータ・クラスは、各イテレータ列のアクセッサ・メソッドを定義します。その場合、各メソッドの名前は対応する列名と同じになります。たとえば、`name` 列を宣言すると、`name()` メソッドが生成されます。

3-43 ページの「[位置指定イテレータの使用](#)」で説明したように、`FETCH INTO` 文と `endFetch()` メソッドを使用して、位置指定イテレータの行を移動し、データを取得します。`FETCH INTO` 文では、暗黙的に `next()` メソッドがコールされます。特別な `FETCH CURRENT` 構文 (7-42 ページの「[FETCH CURRENT 構文: JDBC 結果セットから SQLJ イテレータへ](#)」を参照) を使用していない場合は、`next()` メソッドを位置指定イテレータで明示的に使用しないでください。この `FETCH INTO` 文では、イテレータ列番号に従って名前付けされたアクセッサ・メソッドも暗黙的にコールされます。SQLJ 生成の位置指定イテレータ・クラスは、各イテレータ列のアクセッサ・メソッドを定義します。その場合、各メソッドの名前は列の位置を示します。

`close()` メソッドを使用して、操作の終了したイテレータを終了します。

`getResultSet()` メソッドは、SQLJ と JDBC の連係動作の中心となるメソッドです。詳細は、7-53 ページの「[SQLJ イテレータと JDBC 結果セットの連係動作](#)」を参照してください。

注意： かわりに、`ResultSetIterator` インスタンス（または `ScrollableResultSetIterator` インスタンス）を弱い型指定のイテレータとして直接使用できます。（`ResultSetIterator` は、`ScrollableResultSetIterator` によって拡張されます。）これは、JDBC 結果セットに変換することのみが目的で、名前指定または位置指定イテレータの機能が不要である場合、便利な方法です。また、Oracle SQLJ リリース 8.1.7 で取り入れられた SQLJ の `FETCH CURRENT` 構文でもアクセスできます。7-37 ページの「[結果セット・イテレータ](#)」を参照してください。

イテレータ宣言での IMPLEMENTS 句の使用

イテレータ宣言でインタフェースを実装すると役立つ場合があります。その概要と構文は、3-5 ページの「[IMPLEMENTS 句宣言](#)」を参照してください。

たとえば、イテレータ・クラスがあり、そのクラスで 1 つ以上の列へのアクセスを制限するとします。3-38 ページの「[名前指定イテレータの使用法](#)」で説明したように、SQLJ で生成された名前指定イテレータ・クラスには、イテレータの各列にアクセッサ・メソッドがあります。特定の列へのアクセスを制限する場合は、アクセッサ・メソッドのサブセットのみでインタフェースを作成します。そして、イテレータ・クラス型のインスタンスを公開せずに、インタフェース型のインスタンスをユーザーに公開します。

たとえば、従業員データの名前指定イテレータを作成し、それに `ENAME`（従業員名）、`EMPNO`（従業員番号）および `SAL`（給与）の各列を定義するとします。その場合、次のように指定します。

```
#sql iterator EmpIter (String ename, int empno, float sal);
```

この文を実行すると、`ename()`、`empno()` および `sal()` の各アクセッサ・メソッドが定義されたクラス `EmpIter` が生成されます。

`SAL` 列にアクセスできないようにするとします。`ename()` メソッドおよび `empno()` メソッドはあるが、`sal()` メソッドはないインタフェース `EmpIterIntfc` を作成できます。前述の宣言のかわりに、次のイテレータ宣言の使用も可能です（`EmpIterIntfc` がパッケージ `mypackage` 内にあると想定します）。

```
#sql iterator EmpIter implements mypackage.EmpIterIntfc
    (String ename, int empno, float sal);
```

`EmpIterIntfc` インスタンスからしかユーザーがデータにアクセスできないようにアプリケーションをコーティングすると、`SAL` 列にはアクセスできなくなります。

イテレータ・クラスのサブクラス化のサポート

SQLJ は、イテレータ・クラスのサブクラス化機能をサポートしています。このサブクラス化は、問合せおよび問合せの結果へ追加できる機能なので、非常に便利です。

イテレータ・サブクラスに関する主要な要件の 1 つは、`sqlj.runtime.ResultSet` インスタンスを入力として取る `public` コンストラクタを宣言することです。SQLJ ランタイムでは、問合せ結果をサブクラスのインスタンスに割り当てる際にこのコンストラクタがコールされるためです。しかしもっと重要なのは、必要な機能を指定することです。

元のイテレータ・クラス（使用するサブクラスのスーパークラス）の機能を使用することも可能です。たとえば、各問合せ結果の中で次に進むときは、`super.next()` メソッドをコールすることも可能です。

結果セット・イテレータ

SQLJ イテレータの強い型指定の機能が必要ない場合があります。

そのような場合は、`sqlj.runtime.ResultSetIterator` 型のインスタンスを直接使用して、問合せデータを取得できるため、名前指定イテレータ・クラスや位置指定イテレータ・クラスを宣言する必要はありません。かわりに、`ResultSetIterator` を拡張する `sqlj.runtime.ScrollableResultSetIterator` 型を使用できます。これにより、SQLJ のスクロール可能なイテレータ機能を使用できます。詳細は、7-43 ページの「[スクロール可能な結果セット・イテレータ](#)」を参照してください。

強い型指定のイテレータではなく、結果セット・イテレータを使用すると、イテレータ・クラスを宣言しなくて済むかわりに、SQLJ SELECT 操作に対する厳密な分類のチェックを実行できなくなります。

7-34 ページの「[イテレータ・クラスの実装と拡張機能](#)」で説明したように、すべての名前指定イテレータと位置指定イテレータは、`ResultSetIterator` インタフェースに基づいています。このインタフェースでは、`getResultSet()` と `close()` の各メソッドが定義されています。

SQLJ を使用して結果セット・イテレータ・インスタンスを処理する場合、`ScrollableResultSetIterator` インスタンスを使用して、`FETCH CURRENT` 構文を作成します。7-42 ページの「[FETCH CURRENT 構文: JDBC 結果セットから SQLJ イテレータへ](#)」を参照してください。

JDBC を使用して結果セット・イテレータ・インスタンスを処理する場合、その `getResultSet()` メソッドを使用できます（7-56 ページの「[弱い型指定のイテレータ \(ResultSetIterator\) の使用と変換](#)」を参照してください）。次に、基になっている取得した結果セットを処理します。

結果セット・イテレータをその基になる結果セットで処理する場合、操作の終了後に、結果セットではなく結果セット・イテレータを終了する必要があります。結果セット・イテレータを終了すると、結果セットも終了します。ただし、結果セットを終了しても、結果セット・イテレータは終了しません。

注意： Oracle SQLJ では、結果セット・イテレータのホスト式としての使用と、FETCH 文でのカーソル表現をサポートしています。この機能は、Oracle8i リリースではサポートされていません。

スクロール可能なイテレータ

SQLJ の ISO 規格では、スクロール可能な JDBC 結果セットのための JDBC 2.0 仕様に合せて作成された機能によって、スクロール可能なイテレータをサポートしています。Oracle SQLJ で、この機能のサポートを追加しています。

スクロール可能な結果セットの概要は、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。

スクロール可能なイテレータの宣言

イテレータをスクロール可能として設定するには、イテレータの宣言に次の句を追加します。

```
implements sqlj.runtime.Scrollable
```

これにより、SQLJ トランスレータでスクロール可能なインタフェースを実装したイテレータが生成されます。次に、スクロール可能な名前指定イテレータ宣言の例を示します。

```
#sql public static MyScrIter implements sqlj.runtime.Scrollable
    (String ename, int empno);
```

SQLJ トランスレータにより MyScrIter クラスに対して生成されたコードは、スクロール可能なインタフェースのすべてのメソッドを自動的にサポートします。後述の「[スクロール可能なインタフェース](#)」を参照してください。

スクロール可能なイテレータの更新検出

スクロール可能な結果セットのように、基になるデータへの変更にスクロール可能なイテレータが反応するように宣言できます。Oracle SQLJ のスクロール可能なイテレータは、デフォルトで sensitivity 設定が INSENSITIVE にセットされています。この設定では、基になるデータにどのような変更があっても検出されません。ただし、with 句の宣言を使用して、この設定を変更できます。次の例では、前述の例にさらに更新検出を指定しています。

```
#sql public static MyScrIter implements sqlj.runtime.Scrollable
    with (sensitivity=SENSITIVE)
    (String ename, int empno);
```

重要： implements 句は with 句の前に置く必要があります。

SQLJ 標準では `ASENSITIVE` も設定できますが、Oracle SQLJ ではこれは定義されていません。`sensitivity` を `ASENSITIVE` に設定すると、デフォルト設定の `INSENSITIVE` が使用されます。

前述の宣言がある場合、`MyScrIter` インスタンスはデータ変更に反応し、フェッチ・サイズ・ウィンドウなどの要因によって変わります。更新検出性のあるスクロール可能な JDBC 結果セットの動作（更新検出性のあるスクロール可能なイテレータに基づいています）の概要は、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。

スクロール可能なインタフェース

ここでは、`sqlj.runtime.Scrollable` インタフェースの重要なメソッドを説明します。

スクロール可能なインタフェースのフェッチ方向についてのヒントを提供します。次のメソッドは、実行コンテキスト上のみでなくスクロール可能なイテレータ上でも定義されます。`ExecutionContext` インスタンスを使用して、デフォルトの方向がスクロール可能なイテレータの生成で使用されるようにします。

- `setFetchDirection(int)`: 行が処理される方向に関して、SQLJ ランタイムにヒントを与えるメソッド。方向は、`sqlj.runtime.ResultSetIterator.FETCH_FORWARD`、`FETCH_REVERSE` または `FETCH_UNKNOWN` のいずれかである必要があります。

`ExecutionContext` で方向に対する値を指定しない場合、`FETCH_FORWARD` がデフォルトとして使用されます。

- `int getFetchDirection()`: その時点でのデータ行をフェッチする方向（直前に説明されている整数の定数のいずれか）を取り出すメソッド。

また、基になる結果セットにイテレータ・オブジェクトの現在の位置情報を戻す、スクロール可能なイテレータのメソッドもいくつかあります。これらのメソッドはすべて、イテレータの基となる結果セットに行が含まれない場合は常に `false` を戻します。

- `boolean isBeforeFirst()`: イテレータ・オブジェクトが結果セットの最初の行の前にあるかどうかを識別するメソッド。
- `boolean isFirst()`: イテレータ・オブジェクトが結果セットの最初の行にあるかどうかを識別するメソッド。
- `boolean isLast()`: イテレータ・オブジェクトが結果セットの最後の行にあるかどうかを識別するメソッド。メソッド `isLast()` をコールすると、負荷がかかる場合があります。これは、JDBC ドライバで現在の行が結果セットの最終行であるかどうかを判別するために、1 つ先の行をフェッチする必要があるためです。
- `boolean isAfterLast()`: イテレータ・オブジェクトが結果セットの最後の行より後にあるかどうかを識別するメソッド。

注意： ナビゲーションの追加のメソッドも、スクロール可能なインタフェースで定義されていれば、同様に使用できます。これについては、後述の「[スクロール可能な名前指定イテレータ](#)」で説明します。

スクロール可能な名前指定イテレータ

名前指定イテレータでは、スクロール可能なインタフェースが定義されたナビゲーション・メソッドを使用して、結果セットの行を移動できます。前述のように、スクロールしないイテレータには、次のナビゲーション・メソッドのみがあります。

- `boolean next()`: 結果セットの最終行にイテレータ・オブジェクトを移動するメソッド。

(詳細は、3-38 ページの「[名前指定イテレータの使用法](#)」を参照してください。)

追加のナビゲーション・メソッドは、スクロール可能な名前指定イテレータで使用できます。これらのメソッド・ファンクションは、結果セットの実際の行上にイテレータを配置しようとする際に、`next()` と似た働きをします。イテレータが有効な行に配置された場合は `true` を、そうでない場合は `false` を戻します。さらに、結果セット内の最初の行の前または最終行の後にイテレータ・オブジェクトを配置すると、イテレータ・オブジェクトがそれぞれ最初の前、最後の後の位置に配置されます。

次のメソッドがサポートされます。

- `boolean previous()`: 結果セットの以前の行にイテレータ・オブジェクトを移動するメソッド。
- `boolean first()`: 結果セットの最初の行にイテレータ・オブジェクトを移動するメソッド。
- `boolean last()`: 結果セットの最終行にイテレータ・オブジェクトを移動するメソッド。
- `boolean absolute(int)`: 結果セットの指定された行番号にイテレータ・オブジェクトを移動するメソッド。最初の行は 1、2 番目の行は 2 というように数えられます。行番号がマイナスの場合、イテレータ・オブジェクトは、結果セットの最終行に対する相対的な行位置に移動します。たとえば、`absolute(-1)` をコールすると、イテレータ・オブジェクトは最終行に配置され、`absolute(-2)` をコールすると最後から 2 番目の行に配置されることになります。
- `boolean relative(int)`: 現在の位置からプラスまたはマイナスのいずれかの相対行番号へイテレータ・オブジェクトを移動するメソッド。`relative(0)` のコールは有効ですが、イテレータの位置は変わりません。

メソッド `beforeFirst()` およびメソッド `afterLast()` は、イテレータ・オブジェクトを結果セット上の実際の行に配置しないため、`void` を返します。

- `void beforeFirst()`: 結果セットの冒頭、つまり最初の行の前にイテレータ・オブジェクトを移動するメソッド。結果セットに行がない場合は、機能しません。
- `void afterLast()`: 結果セットの最後、つまり最終行の後にイテレータ・オブジェクトを移動するメソッド。結果セットに行がない場合は、機能しません。

スクロール可能な位置指定イテレータ

位置指定イテレータの汎用的な `FETCH` 構文は、3-43 ページの「[位置指定イテレータの使用](#)」を参照してください。次にその例を示します。

```
#sql { FETCH :iter INTO :x, :y, :z };
```

これは、実際には次の構文の省略バージョンです。

```
#sql { FETCH NEXT FROM :iter INTO :x, :y, :z };
```

結果セットの以前の行、最初の行または最後の行に移動する別のパターンです。(ただし、移動メソッドがモデル化された後の JDBC 2.0 では `previous()` が使用される一方で、SQL を基本とした `FETCH` では `PRIOR` が採用されています。この矛盾を忘れた場合でも、Oracle SQLJ トランスレータでは `FETCH PREVIOUS` も受け入れられます。)

```
#sql { FETCH PRIOR FROM :iter INTO :x, :y, :z };  
#sql { FETCH FIRST FROM :iter INTO :x, :y, :z };  
#sql { FETCH LAST FROM :iter INTO :x, :y, :z };
```

絶対的または相対的な移動に対する数値を渡す構文もあります。この構文で、特定の絶対行に移動したり、その時点での位置から前や後ろに移動できます。

```
#sql { FETCH ABSOLUTE :n FROM :iter INTO :x, :y, :z };  
#sql { FETCH RELATIVE :n FROM :iter INTO :x, :y, :z };
```

注意： このような構文を作成した場合、`FETCH` が有効な行へ移動できずに値を取得することに失敗した場合でも、イテレータの `endFetch()` メソッドは常に `true` を返します。

必ずホスト式を使用して、移動を指定してください。数値の定数のみを使用することはできません。次の構文ではなく、

```
#sql { FETCH RELATIVE 0 FROM :iter INTO :x, :y, :z };
```

このように記述する必要があります。

```
#sql { FETCH RELATIVE :(0) FROM :iter INTO :x, :y, :z };
```

イテレータが有効な行にある場合、このコマンドはイテレータの位置を変更しません。変数に代入します。

注意： 7-40 ページの「スクロール可能な名前指定イテレータ」に説明されているナビゲーション・メソッドと、後述の「**FETCH CURRENT 構文：**[JDBC 結果セットから SQLJ イテレータへ](#)」に説明されている **FETCH CURRENT** 構文を組み合わせることによって、スクロール可能な位置指定イテレータで移動できます。

FETCH CURRENT 構文：JDBC 結果セットから SQLJ イテレータへ

既存の JDBC プログラムをできるだけ修正せずに SQLJ で書き換えたい場合を想定します。

JDBC 結果セットでは、`next()`、`previous()`、`absolute()` などの移動メソッドのみを使用します。名前指定イテレータを使用して SQLJ 内でこれを迅速にモデル化できます。ただし、このためには、SQL 結果セットのすべての列に適切な名前が必要です。実際は、結果セットの多くの（すべてではないにしても）列でエイリアス名が必要となります。この方法は、問合せテキストを加工しないでおく場合は、使用できません。

別の方法は、問合せソースを変更しないために、結果セットの位置指定イテレータ型を定義する方法です。ただし、このアプローチでは、プログラムのコントロールフロー・ロジックを変更する必要があります。次の JDBC コードのサンプルで確認します。

```
ResultSet rs = ... // execute ...query...;
while (rs.next()) {
    x := rs.getXxx(1); y:=rs.getXxx(2);
    ...process...
}
```

これで、次の行を SQLJ に変換します。

```
MyIter iter;
#sql iter = { ...query... };
while(true) {
    #sql { FETCH :iter INTO :x, :y };
    if (iter.endFetch()) break;
    ...process...
}
```


スクロール可能なイテレータ上での任意の移動を想定した場合、プログラム・ロジックへの変換はより困難になります。位置指定イテレータは名前指定イテレータの移動コマンドをすべて実装しているため、これを利用し、イテレータから変数を代入させるために `RELATIVE : (0)` を使用できます。

```
MyIter iter;
#sql iter = { ...query... };
while (iter.next()) {
    #sql { FETCH RELATIVE : (0) FROM :iter INTO :x, :y };
    ...process...
}
```

これで、元の間合せおよび元のプログラム・ロジックをそのまま利用できます。ただし、このアプローチにはまだデメリットが1つあります。つまり、`MyIter` イテレータ型には、スクロール可能なインタフェースを実装する必要があります。このプロパティが実際に必要でない場合も実装する必要があります。これに対処するため、`Oracle SQLJ` には次の構文拡張機能があります。

```
#sql { FETCH CURRENT FROM :iter INTO :x, :y, :z };
```

この構文の場合、スクロール可能なイテレータと同様にスクロールしないイテレータに対しても `JDBC` の例を `SQLJ` に書き換えられます。

```
AnyIterator ai;
#sql ai = { ...query... };
while (ai.next()) {
    #sql { FETCH CURRENT FROM :ai INTO :x, :y };
    ...process...
}
```

スクロール可能な結果セット・イテレータ

`Oracle SQLJ` では、弱い型指定の結果セットへのサポートが拡張されて、スクロール可能な結果セット・イテレータ型が追加されました。定義は次のとおりです。

```
package sqlj.runtime;
public interface ScrollableResultSetIterator
    extends ResultSetIterator
    implements Scrollable
{ }
```

この型は `sqlj.runtime.ResultSetIterator` を拡張しているため、7-37 ページの「[結果セット・イテレータ](#)」のメソッドがサポートされています。

また、`sqlj.runtime.Scrollable` インタフェースも実装されたため、7-39 ページの「[スクロール可能なインタフェース](#)」および 7-40 ページの「[スクロール可能な名前指定イテレータ](#)」に説明されているメソッドがサポートされています。

さらに、スクロール可能な結果セット・イテレータでは、7-42 ページの「[FETCH CURRENT 構文: JDBC 結果セットから SQLJ イテレータへ](#)」に説明されている FETCH CURRENT 構文がサポートされています。

次の JDBC コードを考えます。

```
Statement st = conn.createStatement("SELECT ename, empid FROM emp");
ResultSet rs = st.executeQuery();
while (rs.next()) {
    x = rs.getString(1);
    y = rs.getInt(2);
}
rs.close();
```

SQLJ 結果セット・イテレータを使用して、同じ内容を次のコードで記述できます。

```
sqlj.runtime.ResultSetIterator rsi;
#sql rsi = { SELECT ename, empid FROM emp };
while (rsi.next()) {
    #sql { FETCH CURRENT FROM :rsi INTO :x, :y };
}
rsi.close();
```

スクロール機能を活用して、次のコードを記述することもできます。

```
sqlj.runtime.ScrollableResultSetIterator srsi;
#sql srsi = { SELECT ename, empid FROM emp };
srsi.afterLast();
while (srsi.previous()) {
    #sql { FETCH CURRENT FROM :srsi INTO :x, :y };
}
srsi.close();
```

詳細なトランザクション制御

SQLJ では、SQL SET TRANSACTION 文がサポートされています。この文を使用すると、トランザクションのアクセス・モードと分離レベルを指定できます。標準 SQLJ では、READ ONLY および READ WRITE アクセス・モード設定がサポートされています。ただし、READ ONLY は、Oracle JDBC ではサポートされません。（アクセス権を設定すると、アクセス・モードと同じ効果が得られます。）分離レベルとしては、SERIALIZABLE、READ COMMITTED、READ UNCOMMITTED または REPEATABLE READ を設定できます。ただし、Oracle SQL では、READ UNCOMMITTED または REPEATABLE READ はサポートされていません。

READ WRITE は、標準 SQL と Oracle SQL のデフォルトのアクセス・モードです。

READ COMMITTED は Oracle SQL でのデフォルトの分離レベルであり、SERIALIZABLE は標準 SQL でのデフォルトの分離レベルです。

次に、アクセス・モードと分離レベルについて簡単に説明します。詳細は、『Oracle9i SQL リファレンス』を参照してください。また、標準 SQL のマニュアルも参照してください。

COMMIT や ROLLBACK などの基本的なトランザクション制御操作に対する SQLJ サポートも含めたトランザクションの概要は、4-24 ページの「[基本的なトランザクション制御](#)」を参照してください。

SET TRANSACTION 構文

SQLJ では、SET TRANSACTION 文の構文は次のようになります。

```
#sql { SET TRANSACTION <access_mode>, <ISOLATION LEVEL isolation_level> };
```

この文に接続コンテキスト・インスタンスを指定しなかった場合は、デフォルトの接続が適用されます。

SET TRANSACTION を使用する場合、それが DML 文の前に定義され、トランザクション内で最初の文であることが必要です。つまり、データベースへの接続、あるいは最新の COMMIT または ROLLBACK 以降、最初の文であることが必要です。

標準 SQLJ で設定したアクセス・モードや分離レベルは、以降のトランザクション開始時に明示的にリセットしないかぎり、トランザクション全体にわたって有効のままになっています。

標準 SQLJ SET TRANSACTION 文では、最初に分離レベルを指定できます。または、アクセス・モードのみ、あるいは分離レベルのみの指定も可能です。たとえば、次のように指定します。

```
#sql { SET TRANSACTION READ WRITE };
```

```
#sql { SET TRANSACTION ISOLATION LEVEL SERIALIZABLE };
```

```
#sql { SET TRANSACTION READ WRITE, ISOLATION LEVEL SERIALIZABLE };
```

```
#sql { SET TRANSACTION ISOLATION LEVEL READ COMMITTED, READ WRITE };
```

SET TRANSACTION 文には、デフォルト接続を適用することも、特定の接続コンテキスト・インスタンスも指定できます。

```
#sql [myCtxt] { SET TRANSACTION ISOLATION LEVEL SERIALIZABLE };
```

SQLJ では、1 つの SET TRANSACTION 文に、アクセス・モードと分離レベルの両方を設定できます。これは、SQL*Plus などの他の Oracle SQL Tools には適用されません。Oracle の SQL 製品では、1 つの文にアクセス・モードと分離レベルのいずれかしか設定できません。

アクセス・モードの設定

READ WRITE と READ ONLY の各アクセス・モード設定（サポートされている場合）の機能は次のとおりです。

- READ WRITE（デフォルト）：READ WRITE トランザクションでは、ユーザーがデータベースを更新できます。SELECT、INSERT、UPDATE および DELETE のすべてを実行できます。
- READ ONLY（Oracle9i リリース 2 (9.2) から Oracle JDBC でサポートされています）：READ ONLY トランザクションの場合、データベースの更新はユーザーに許可されていません。SELECT は実行できますが、INSERT、UPDATE、DELETE および SELECT FOR UPDATE は実行できません。

分離レベルの設定

READ COMMITTED、SERIALIZABLE、READ UNCOMMITTED および REPEATABLE READ の各分離レベルには（サポートされている場合）、次の機能があります。

- READ UNCOMMITTED（Oracle9i ではサポートされていません）：dirty read、non-repeatable read、phantom read のすべてが許可されます。dirty read、non-repeatable read、phantom read については後述します。
- READ COMMITTED（Oracle9i ではデフォルト）：dirty read は許可されませんが、non-repeatable read と phantom read は許可されます。トランザクションに、他のトランザクションで行をロックする DML 文が含まれている場合、そのトランザクションで必要な行のロックが他のトランザクションによって解除されるまで、文がブロックされます。
- REPEATABLE READ（Oracle9i ではサポートされていません）：dirty read と non-repeatable read は許可されませんが、phantom read は許可されます。
- SERIALIZABLE: dirty read、non-repeatable read、phantom read のすべてが許可されません。トランザクション内の DML 文では、トランザクション開始後に変更がコミットされたリソースを更新できません。そのような DML 文は、失敗します。

dirty read は、トランザクション B がトランザクション A によって更新された行にアクセスし、その後でトランザクション A がその更新をロールバックした場合に起こります。その結果、トランザクション B は、実際にはデータベースにコミットされていないデータを扱っていることになります。

non-repeatable read は、トランザクション A が行を取得し、次にトランザクション B がその行を更新し、その後でトランザクション A が同じ行を再度取得した場合に起こります。トランザクション A は同じ行を 2 回取得しますが、取得した内容が異なります。

phantom read は、トランザクション A が特定の条件を満たす行のセットを取得し、次にトランザクション A の条件を満たす行をトランザクション B が挿入または更新し、その後でトランザクション A が条件に基づいて行を取得した場合に起こります。トランザクション A は追加された行を扱っていることになります。その行を phantom と呼びます。

4つの分離レベルは、次の順で移行します。

SERIALIZABLE > REPEATABLE READ > READ COMMITTED > READ UNCOMMITTED

たとえば、Oracle9iを使用しているために、REPEATABLE READやREAD UNCOMMITTEDを設定できない場合、上位の（1つ右の）分離レベルを設定して、必要な分離レベルがカバーされるようにします。

JDBC 接続クラスのメソッドの使用

必要に応じて、接続コンテキスト・インスタンスの基になる JDBC 接続インスタンスのメソッドを使用して、トランザクションのアクセス・モードと分離レベルにアクセスし、それらを設定できます。ただし、これらの JDBC メソッドを使用した SQLJ コードは、移植性が高くないので、お薦めしません。

次は、アクセス・モードと分離レベルを設定する Connection クラスのメソッドです。

- `public abstract int getTransactionIsolation():` カレント・トランザクションの分離レベルを示す、次のいずれかの定数の値を返すメソッド
TRANSACTION_NONE
TRANSACTION_READ_COMMITTED
TRANSACTION_SERIALIZABLE
TRANSACTION_READ_UNCOMMITTED
TRANSACTION_REPEATABLE_READ
- `public abstract void setTransactionIsolation(int):` 前述のいずれかの定数を入力パラメータとして取り、トランザクションの分離レベルを設定するメソッド
- `public abstract boolean isReadOnly():` トランザクションが READ ONLY の場合は true を返し、READ WRITE の場合は false を返すメソッド
- `public abstract void setReadOnly(boolean):` true が入力された場合は、トランザクションのアクセス・モードを READ ONLY に設定し、false が入力された場合は READ WRITE に設定するメソッド

SQLJ と JDBC の関係動作

1-2 ページの「[SQLJ について](#)」に説明されているように、SQLJ 文は静的 SQL 操作で通常使用されます。Oracle9i には動的 SQL もサポートする拡張機能がありますが、別の方法として、動的操作のために JDBC コードを SQLJ アプリケーション内で使用することもできます（この方がさらに移植性が高くなります）。この他にも、JDBC コードを SQLJ アプリケーションを使用すると便利になるのみでなく、使用が必須である場合があります。

この理由で、SQLJ では、SQLJ 文と JDBC 文を同時に使用すると、SQLJ 構文と JDBC 構文を関係できます。

SQLJ と JDBC 間のトランザクションでは、次の 2 つが特に有用です。

- SQLJ 接続コンテキストと JDBC 接続間のトランザクション
- SQLJ イテレータと JDBC 結果セット間のトランザクション

JDBC の機能の概要は、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。

SQLJ 接続コンテキストと JDBC 接続の関係動作

SQLJ では、SQLJ 接続コンテキスト・インスタンスから JDBC 接続インスタンスに変換することも、その逆に変換することも可能です。

注意： SQLJ 接続コンテキストと JDBC 接続間で変換する場合、この 2 つのオブジェクトは基になる同じ物理接続を共有していることに注意してください。7-51 ページの「[共有接続について](#)」を参照してください。

接続コンテキストから JDBC 接続への変換

JDBC 操作を SQLJ で確立したデータベース接続によって実行する場合（たとえば、アプリケーションで JDBC 接続オブジェクトを戻すライブラリ・ルーチンをコールする場合）、SQLJ 接続コンテキスト・インスタンスを JDBC 接続インスタンスに変換する必要があります。

SQLJ アプリケーションの接続コンテキスト・インスタンスには、`sqlj.runtime.ref.DefaultContext` クラスのインスタンスの場合も、宣言済みの接続コンテキスト・クラスのインスタンスの場合も、基になる JDBC 接続インスタンスとその JDBC 接続インスタンスを戻す `getConnection()` メソッドが含まれています。JDBC 操作を使用する場合は、JDBC 接続インスタンスを使用して JDBC 文オブジェクトを作成します。

次は、`getConnection()` メソッドの使用法の例です。

インポートは、次のように指定します。

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

実行可能コードを示します。

```
DefaultContext ctx = new DefaultContext
    ("jdbc:oracle:thin:@localhost:1521:orcl", "scott", "tiger", true);
...
(SQLJ operations through SQLJ ctx connection context instance)
...
Connection conn = ctx.getConnection();
...
(JDBC operations through JDBC conn connection instance)
...
```

接続コンテキスト・インスタンスは、DefaultContext クラスのインスタンスでも、宣言した任意の接続コンテキスト・クラスのインスタンスでもかまいません。

デフォルトの SQLJ 接続の基になる JDBC 接続を取得する場合は、getConnection() を DefaultContext.getDefaultContext() コールから直接使用します。その場合、getDefaultContext() は、デフォルトの接続として初期化した DefaultContext インスタンスを戻します。一方、この getConnection() は、その基になる JDBC 接続インスタンスを戻しています。その場合、DefaultContext インスタンスを明示的に使用する必要はないので、Oracle.connect() メソッドも使用できます。このメソッドは、暗黙的にインスタンスを生成し、それをデフォルトの接続にします。

(接続コンテキスト・インスタンスとデフォルトの接続の基本事項は、4-6 ページの「[接続の際の考慮事項](#)」を参照してください。Oracle.connect() の詳細は、4-12 ページの「[Oracle クラスについて](#)」を参照してください。)

次に、その例を示します。

インポートは、次のように指定します。

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

実行可能コードを示します。

```
...
Connection conn = Oracle.connect
    ("jdbc:oracle:thin:@localhost:1521:orcl", "scott", "tiger").getConnection();
...
(JDBC operations through JDBC conn connection instance)
...
```

例：動的 SQL に対する JDBC と SQLJ 接続の連携動作 次は、デフォルトの SQLJ 接続コンテキスト・インスタンスの基になる JDBC 接続インスタンスを使用して、JDBC 内で動的 SQL 操作を実行するメソッドの例です。JDBC の `java.sql.Connection`、`java.sql.PreparedStatement` および `java.sql.ResultSet` オブジェクトによって、動的操作が実行されます。(JDBC プログラミングの基本機能については、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。)

かわりに、動的 SQL 操作の Oracle SQLJ 拡張機能を使用できます。詳細は、7-57 ページの「動的 SQL のサポート」を参照してください。この例を SQLJ の動的 SQL 機能と結果セット・イテレータからの FETCH 機能を使用するように修正する場合は、7-61 ページの「例 5: 動的 SQL と結果セット・イテレータからの FETCH」を参照してください。

```
import java.sql.*;

public static void projectsDue(boolean dueThisMonth) throws SQLException {
    // Get JDBC connection from previously initialized SQLJ DefaultContext.
    Connection conn = DefaultContext.getDefaultContext().getConnection();

    String query = "SELECT name, start_date + duration " +
        "FROM projects WHERE start_date + duration >= sysdate";
    if (dueThisMonth)
        query += " AND to_char(start_date + duration, 'fmMonth') " +
            " = to_char(sysdate, 'fmMonth') ";
    PreparedStatement pstmt = conn.prepareStatement(query);
    ResultSet rs = pstmt.executeQuery();
    while (rs.next()) {
        System.out.println("Project: " + rs.getString(1) + " Deadline: " +
            rs.getDate(2));
    }
    rs.close();
    pstmt.close();
}
```

JDBC 接続から接続コンテキストへの変換

JDBC `Connection` インスタンスとして接続を開始した後で、それを SQLJ 接続コンテキスト・インスタンスとして使用する場合（たとえば、コンテキスト式でこのインスタンスを使用して、SQLJ 実行文で使用する接続を指定する場合）は、JDBC 接続インスタンスを SQLJ 接続コンテキスト・インスタンスに変換します。

`DefaultContext` クラスとすべての宣言済みの接続コンテキスト・クラスに定義されたコンストラクタは、JDBC 接続インスタンスを入力パラメータとして取り、SQLJ 接続コンテキスト・インスタンスを生成します。

たとえば、JDBC 接続インスタンス `conn` をインスタンス化し、定義したとします。そして、宣言済みの SQLJ 接続コンテキスト・クラスである `MyContext` のインスタンスに、同じ接続を使用するとします。その場合は、次のように指定します。


```
...
#sql context MyContext;
...
MyContext myctx = new MyContext (conn);
...
```

共有接続について

SQLJ 接続コンテキスト・インスタンスとそれに関連付けされた JDBC 接続インスタンスは、基になる同じ物理接続を共有します。そのため、次の状況が発生します。

- 接続コンテキスト `getConnection()` メソッドを使用して、SQLJ 接続コンテキスト・インスタンスから JDBC 接続インスタンスを取得すると、この接続コンテキスト・インスタンスの状態が `Connection` インスタンスに継承されます。特に、`Connection` インスタンスは、接続コンテキスト・インスタンスの自動コミットの設定を保持します。
- (Connection インスタンスを入力パラメータとして取る接続コンテキスト・コンストラクタを使用して) JDBC 接続インスタンスから SQLJ 接続コンテキスト・インスタンスを作成すると、`Connection` インスタンスの状態がこの接続コンテキスト・インスタンスに継承されます。特に、接続コンテキスト・インスタンスは、`Connection` インスタンスの自動コミットの設定を継承します。デフォルトでは、JDBC 接続インスタンスの自動コミットは `true` に設定されます。このデフォルトの設定を変更するには、`Connection` インスタンスの `setAutoCommit()` メソッドを使用します。
- SQLJ 接続コンテキスト・インスタンスとそれに関連付けされた JDBC 接続インスタンスがある場合、一方のインスタンスのセッション状態を変更するメソッドをコールすると、もう一方のインスタンスにも影響します。実際に変更されるのは、基になる共有セッションのためです。
- 物理接続は 1 つのみであるため、その基になるトランザクションのセットも 1 つのみとなっています。ある接続インスタンスで `COMMIT` または `ROLLBACK` 操作を行うと、基になる接続を共有している他の接続インスタンスにも影響します。

注意： 同一の JDBC 接続インスタンスから複数の SQLJ 接続コンテキスト・インスタンスを作成すると、各インスタンスで基になる物理接続を共有することも可能となります。このようにすると、プログラム・モジュール間で同じ種類のトランザクションを共有する場合などに便利ことがあります。ただし、この場合にも前に述べた注意があてはまるので注意してください。

共有接続の終了

(`getConnection()` メソッドを使用して) SQLJ 接続コンテキスト・インスタンスから JDBC 接続インスタンスを取得する場合も、(接続コンテキスト・コンストラクタを使用して) JDBC 接続インスタンスから SQLJ 接続コンテキスト・インスタンスを作成する場合も、接続コンテキスト・インスタンスを終了する必要があります。デフォルトでは、接続コンテキスト・インスタンスの `close()` メソッドをコールすると、対応する JDBC 接続インスタンスと、基になる物理接続が終了します。そして、その接続にかかわるすべてのリソースが解放されます。

対応する JDBC 接続インスタンスを終了せずに SQLJ 接続コンテキスト・インスタンスを終了する場合（たとえば、`Connection` インスタンスが、直接または他の接続コンテキスト・インスタンスによって、別の場所で使用されている場合）、次のようにブール値の `KEEP_CONNECTION` を `close()` メソッドに指定します（ここでは、接続コンテキスト・インスタンスとして `ctx` を使用します）。

```
ctx.close(ConnectionContext.KEEP_CONNECTION);
```

`KEEP_CONNECTION` を指定しないと、デフォルトで、それに関連付けされた JDBC 接続インスタンスが終了します。一方、JDBC 接続インスタンスの終了を明示的に指定するには、次のようにします。

```
ctx.close(ConnectionContext.CLOSE_CONNECTION);
```

`KEEP_CONNECTION` および `CLOSE_CONNECTION` は、`sqlj.runtime.ConnectionContext` インタフェースの `static` 定数です。

明示的に終了しなかった接続コンテキスト・インスタンスは、ガベージ・コレクションの際にファイナライザによって終了しますが、`KEEP_CONNECTION` の状態のままとなります。つまり、JDBC 接続インスタンスのリソースを解放するには、明示的な解放またはガベージ・コレクションが必要となります。

JDBC 接続インスタンスのみを終了しても、それに関連付けされた SQLJ 接続コンテキスト・インスタンスは終了しません。基になる物理接続は終了します。ただし、接続コンテキスト・インスタンスのリソースは、ガベージ・コレクションが行われるまで解放されません。

注意：

- 基になる 1 つの JDBC 接続が複数の接続コンテキスト・インスタンスで共有される場合、最後のオープン接続コンテキスト・インスタンスを残してそれ以外すべての接続を終了するには、KEEP_CONNECTION を使用します。
 - 接続コンテキスト・インスタンスを終了しようとするときに、基になる JDBC 接続がすでに終了している場合、エラー・メッセージが発行されます。また基になる接続を終了しようとするときに、その接続がすでに終了している場合も同様です。エラー・メッセージが表示された場合は、JDBC 接続が JDBC コードとは無関係に終了していないことと、基になる接続を使用する SQLJ 接続コンテキスト・インスタンスに発行されたすべての close() コールで KEEP_CONNECTION パラメータを使用していることを確認します。
-

SQLJ イテレータと JDBC 結果セットの関係動作

SQLJ では、SQLJ イテレータから JDBC 結果セットに、またはその逆に変換できます。SQLJ 文のデータを選択する場合に、強い型指定のイテレータが特に必要のないときは、SQLJ で JDBC 結果セットに変換可能な、弱い型指定のイテレータを使用してもかまいません。

結果セットから名前指定イテレータまたは位置指定イテレータへの変換

JDBC 結果セットの操作は、様々な状況で活用できます。たとえば、JDBC に別のパッケージが実装されており、結果セットを介してしかデータにアクセスできない場合、または ResultSetMetaData 情報がどの結果セットにも使用できるように記述されたルーチンであるために、その情報が必要な場合などです。これ以外に、SQLJ アプリケーションから、JDBC 結果セットを戻り値とするストアード・プロシージャを起動する場合も挙げられます。

動的結果セットに構造がある場合は、それをイテレータとして操作して、そのイテレータの提供する強い型指定のパラダイムを使用します。

SQLJ では、既存の JDBC 結果セット・オブジェクトを変換して、名前指定イテレータ・オブジェクトまたは位置指定イテレータ・オブジェクトを設定できます。この変換は、結果セットをイテレータにキャストする処理であるといえます。この処理のための構文は次のようになります。

```
#sql iter = { CAST :rs };
```

これにより、結果セット・オブジェクト rs が SQLJ 実行文にバインドされ、結果セットが変換され、イテレータ iter に結果セット・データが設定されます。

次に、その例を示します。myEmpQuery() は、RSClass クラスの static Java ファンクションで、JDBC 結果セット・オブジェクトを戻り値とする問合せが定義されているとします。

インポートは次のように宣言します。

```
import java.sql.*;
...
#sql public iterator MyIterator (String ename, float sal);
...
```

実行可能コードを示します。

```
ResultSet rs;
MyIterator iter;
...
rs = RSClass.myEmpQuery();
#sql iter = { CAST :rs };
...
(process iterator)
...
iter.close();
...
```

この例で名前指定イテレータのかわりに、位置指定イテレータを使用したとしても、同じ機能が得られます。

次の規則は、JDBC 結果セットを SQLJ イテレータに変換し、データを処理する場合に適用されます。

- 位置指定イテレータを変換する場合、結果セットとイテレータの列数が同じであることが必要です。また、型が正しくマッピングする必要があります。
- 名前指定イテレータに変換する場合、結果セットに少なくともイテレータと同じ数の列が必要です。そして、イテレータのすべての列が名前と型で一致する必要があります。結果セットとイテレータの列の数が同じではない場合、SQLJ トランスレータによって警告が出力されます。ただし、`-warn=nostrict` オプションが設定されている場合は除きます。
- キャストされている結果セットでは、`java.sql.ResultSet` インタフェースを実装する必要があります。クラス `oracle.jdbc.OracleResultSet` では、標準結果セット・クラスと同様に、このインタフェースが実装されています。
- キャストを受け取るイテレータは、`public` と宣言されたイテレータ・クラスのインスタンスであることが必要です。
- 変換前でも変換後でも、結果セットからデータにアクセスしないでください。データには、イテレータのみからアクセスします。
- 操作が終了した後、結果セットではなくイテレータを終了します。イテレータを終了すると、結果セットも終了します。ただし、結果セットを終了しても、イテレータは終了しません。JDBC と関係している場合は、必ず SQLJ エンティティを終了してください。

名前指定イテレータまたは位置指定イテレータから結果セットへの変換

当初は SQLJ で定義した問合せが必要だったときでも、最終的には結果セットの方が必要になる場合があります。(SQLJ では、わかりやすく簡単な構文が使用されています。ただし、結果を動的に処理する場合、または結果セットを入力パラメータとして取る既存の Java メソッドを使用する場合もあります。)

イテレータを結果セットに変換するために、名前指定イテレータ・クラスでも位置指定イテレータ・クラスでも、すべての SQLJ イテレータ・クラスは `getResultSet()` メソッドで生成されます。このメソッドを使用すると、イテレータ・オブジェクトの基になる JDBC 結果セット・オブジェクトが戻り値になります。

次は、`getResultSet()` の使用方法の例です。

インポートは次のように宣言します。

```
import java.sql.*;

#sql public iterator MyIterator (String ename, float sal);
...
```

実行可能コードを示します。

```
MyIterator iter;
...
#sql iter = { SELECT * FROM emp };
ResultSet rs = iter.getResultSet();
...
(process result set)
...
iter.close();
...
```

次の規則は、SQLJ イテレータを JDBC 結果セットに変換し、データを処理する場合に適用されます。

- イテレータ・データを結果セットに書き込む場合、結果セットからのみデータにアクセスする必要があります。変換前でも変換後でも、イテレータに直接アクセスしないでください。
- 操作が終了した後、結果セットではなく元のイテレータを終了します。イテレータを終了すると、結果セットも終了します。ただし、結果セットを終了しても、イテレータは終了しません。JDBC と関係している場合は、必ず SQLJ エンティティを終了してください。

弱い型指定のイテレータ (ResultSetIterator) の使用と変換

7-55 ページの「名前指定イテレータまたは位置指定イテレータから結果セットへの変換」で説明した状況が発生しても、強い型指定のイテレータが必要ない場合があります。問合せに SQLJ 構文を使用し、結果セットからデータを動的に処理することが必要です。

その場合は、`sqlj.runtime.ResultSetIterator` 型を直接使用して問合せデータを受け取ることができます。結果セット・イテレータ型の概要は、7-37 ページの「結果セット・イテレータ」を参照してください。

JDBC 文と標準の結果セットではなく、SQLJ 文と `ResultSetIterator` を使用すると、SQLJ の `SELECT` 構文を簡略化できます。

次は、弱い型指定の結果セット・イテレータの使用方法と変換方法の例です。

インポートは、次のように指定します。

```
import sqlj.runtime.*;
import java.sql.*;
...
```

実行可能コードを示します。

```
ResultSetIterator rsiter;
...
#sql rsiter = { SELECT * FROM table };
ResultSet rs = rsiter.getResultSet();
...
(process result set)
...
rsiter.close();
...
```

注意： Oracle SQLJ では、`next()` メソッドと `FETCH CURRENT` 構文を使用して、結果セット・イテレータで移動できます。7-42 ページの「[FETCH CURRENT 構文: JDBC 結果セットから SQLJ イテレータへ](#)」を参照してください。さらに、スクロール可能な結果セット・イテレータの場合、ナビゲーション・メソッドがサポートされています。これらのオプションについては、7-40 ページの「[スクロール可能な名前指定イテレータ](#)」で説明します。

動的 SQL のサポート

Oracle SQLJ には、動的 SQL（事前定義されずリアルタイムに変更できる操作）をサポートする拡張機能があります。SQLJ 文に埋め込まれた動的 SQL 式は、メタ・バインド式と呼ばれ、後述で説明されています。

注意： JDBC コードを使用することは、Oracle9i の動的 SQL の選択肢であり、コードの移植性がかかわる場合は好ましい選択ですが、Oracle SQLJ でサポートする動的 SQL では、SQLJ を簡略化された 1 つのデータ・アクセス用 API として使用できます。（SQLJ と JDBC の相互作用については、7-48 ページの「SQLJ と JDBC の関係動作」を参照してください。）

メタ・バインド式

メタ・バインド式は、SQLJ 文で動的 SQL 用に使用されますが、静的 SQL 句が使用されています。メタ・バインド式には、String 型の Java 識別子、または実行時に変換される文字列の値の Java 式が含まれます。さらに、SQLJ ではオンラインでセマンティクス・チェックを実行するので、変換時のチェックに使用するために、必要に応じてメタ・バインド式に静的 SQL 置換コードを指定します。

ここでは、メタ・バインド式の使用方法、制限事項、構文および動作について説明します。

メタ・バインド式：使用方法と制限事項の概要

次のいずれかの項目のかわりにメタ・バインド式を使用できます。

- 表名
- SELECT リストの列名（列の別名が指定されている場合は、別名を除く）
- WHERE 句条件の一部またはすべて
- DDL や DML 文のロール名、スキーマ名、カタログ名またはパッケージ名
- SQL リテラル値または SQL 式

メタ・バインド式では、次の制限事項に注意してください。この制限事項は、SQLJ トランスレータで SQL 操作の種類を確認して、SQLJ 文全体の構文分析を実行するためのものです。

- メタ・バインド式は、SQLJ 文の中で、コメント以外の最初の SQL 操作として使用できません。
- メタ・バインド式には、SQLJ SELECT INTO 文の INTO トークンを使用できません。また、メタ・バインド式を拡張して、SELECT INTO 式の INTO リストにすることもできません。

- メタ・バインド式は、次のいずれの種類の SQL または SQLJ の命令および句では使用できません。使用できない命令および句は、CALL、VALUES、PSM SET、COMMIT、ROLLBACK、FETCH INTO および CAST です。

メタ・バインド式：構文と動作

次にメタ・バインド式の一般的な構文を示します。

```
:{ Java_bind_expression }
```

または

```
:{ Java_bind_expression :: SQL_replacement_code }
```

空白はオプションです。

SQLJ 文の SQL 命令内には複数のメタ・バインド式を使用できます。

Java バインド式 Java バインド式は次のいずれかになります。

- String 型の Java 識別子
- 文字列を評価する Java 式

メタ・バインド式内の Java バインド式は、標準 Java 規則に準拠しています。また、SQLJ ホスト式と同様の構文です（3-14 ページの「[Java ホスト式、コンテキスト式および結果式](#)」を参照してください）。ただし、ホスト式と異なり、メタ・バインド式内の Java バインド式は小カッコで囲まれていません。これは、SQL 置換コードがある場合に、「::」トークンが Java バインド式と SQL コードのセパレータとして機能して、SQL 置換コードがない場合に閉じカッコ「}」が終了記号として機能するためです。いずれの場合も、明確になります。

注意： Java バインド式内またはメタ・バインド式の「:」と「{」の間には、モード指定子（IN、OUT または INOUT）を指定できません。

SQL 置換コード SQL 置換コード句は、0（ゼロ）以上の SQL トークンで構成されています。次の要件と制限事項を参照してください。

- SQL 規則に準拠していることが必要です。
- 中カッコの「{」および「}」を 1 組で使用する必要があります（SQL のコメント、定数または識別子の一部に使用される場合は例外です）。
- SQL 命令内に SQLJ ホスト式やネストしたメタ・バインド式を使用できません。

注意： SQL 置換コードを空白にすることは可能です。

変換時の動作 SQL 置換コード（空白文字列のみの場合も）がメタ・バインド式にある場合は、変換時に SQL コードによってメタ・バインド式が置換されます。SQL 置換コードの目的は、SQLJ トランスレータでオンライン・セマンティクス・チェックを実行できるようにすることです。

SQLJ 文内のメタ・バインド式に SQL 置換コード句がない場合は、SQLJ トランスレータでその文に対するオンライン・セマンティクス・チェックを実行できません。その文には構文チェックのみが実行されます。

実行時の動作 実行時に各メタ・バインド式は、Java バインド式の評価によって置換されます。

Java バインド式の評価が null になる場合、動的 SQL 文全体が未定義になります。

SQLJ 動的 SQL の例

ここでは、SQLJ コードでの動的 SQL の使用例を示します。

例 1

```
...
int x = 10;
int y = x + 10;
int z = y + 10;
String table = "new_Emp";
#sql { INSERT INTO :{table} VALUES (:x, :y, :z) };
...
```

変換時の SQL 操作は次のようになります。

```
INSERT INTO emp VALUES (10, 20, 30);
```

SQLJ によって、emp 表があるスキーマに対してオンライン・セマンティクス・チェックが実行されます。（ランタイム・スキーマに new_Emp のみが存在し、アプリケーション実行まで作成されません。）

実行時の SQL 操作は次のようになります。

```
INSERT INTO new_Emp VALUES (10, 20, 30);
```

例 2

```
...
String table = "new_Emp";
String query = "ename LIKE 'S%' AND sal>1000";
#sql myIter = { SELECT * FROM :{table} :: emp2}
                WHERE :{query} :: ename='SCOTT' };
...
```

変換時の SQL 操作は次のようになります。

```
SELECT * FROM emp2 WHERE ename='SCOTT';
```

SQLJ によって、emp2 表があるスキーマに対してオンライン・セマンティクス・チェックが実行されます。

実行時の SQL 操作は次のようになります。

```
SELECT * FROM new_Emp WHERE ename LIKE 'S%' AND sal>1000;
```

例 3

```
...
double raise = 1.12;
String col = "comm";
String whereQuery = "WHERE "+col+" IS NOT null";
for (int i=0; i<5; i++)
{
    #sql { UPDATE :{"emp"+i} :: emp}
          SET :{col} :: sal = :{col} :: sal * :raise :{whereQuery} :: };
}
...
```

変換時の SQL 操作は次のようになります。

```
UPDATE emp SET sal = sal * 1.12;
```

SQLJ によって、emp 表があるスキーマに対してオンライン・セマンティクス・チェックが実行されます。SQL 置換コードは空白なので、変換時に WHERE 句はありません。

実行時に SQL 操作が次のように 5 回実行されます。

```
UPDATE emp0 SET comm = comm * 1.12 WHERE comm IS NOT null;
UPDATE emp1 SET comm = comm * 1.12 WHERE comm IS NOT null;
UPDATE emp2 SET comm = comm * 1.12 WHERE comm IS NOT null;
UPDATE emp3 SET comm = comm * 1.12 WHERE comm IS NOT null;
UPDATE emp4 SET comm = comm * 1.12 WHERE comm IS NOT null;
```

例 4

```

...
double raise = 1.12;
String col = "comm";
String whereQuery = "WHERE "+col+" IS NOT null";
for (int i=0; i<10; i++)
{
    #sql { UPDATE :{"emp"+i}
          SET :{col :: sal} = :{col :: sal} * :raise :{whereQuery ::} };
}
...

```

例 4 と例 3 の実行時の動作は同じです。ただし、変換時には違いがあります。例 4 では、最初のメタ・バインド式への SQL 置換コード :{"emp"+i} がいないため、SQLJ でオンライン・セマンティクス・チェックが実行できません。

例 5: 動的 SQL と結果セット・イテレータからの FETCH この例は、JDBC 文のかわりに SQLJ 文を使用して、7-50 ページの「例: 動的 SQL に対する JDBC と SQLJ 接続の連係動作」を修正しています。またこの例では、結果セット・イテレータから FETCH CURRENT 機能を使用しています。7-42 ページの「FETCH CURRENT 構文: JDBC 結果セットから SQLJ イテレータへ」を参照してください。

```

import java.sql.*;

public static void projectsDue(boolean dueThisMonth) throws SQLException {

    ResultSetIterator rsi;
    String andClause = (dueThisMonth) ?
        " AND to_char(start_date + duration, 'fmMonth' ) "
        + " = to_char(sysdate, 'fmMonth') "
        : "";
    #sql rsi = { SELECT name, start_date + duration FROM projects
                WHERE start_date + duration >= sysdate :{andClause ::} };
    while (rsi.next())
    {
        String name = null;
        java.sql.Date deadline = null;
        #sql { FETCH CURRENT FROM :rsi INTO :name, :deadline };
        System.out.println("Project: " + name + "Deadline: " + deadline);
    }
    rsi.close();
}

```

トランスレータのコマンドラインとオプション

ソース・コードの作成後、SQLJ トランスレータでこのコードを変換する必要があります。この章では、SQLJ トランスレータのコマンドライン、オプションおよびプロパティ・ファイルについて説明します。

次の項目について説明します。

- トランスレータのコマンドラインとプロパティ・ファイル
- 基本的なトランスレータ・オプション
- 拡張トランスレータ・オプション
- トランスレータによる代替環境のサポートとオプション

トランスレータのコマンドラインとプロパティ・ファイル

ここでは、SQLJ トランスレータを実行するためのスクリプト `sqlj` の一般的なコマンドライン構文について説明し、使用可能なすべてのオプション一覧を示します。また、SQLJ プロパティ・ファイルと環境変数 `SQLJ_OPTIONS` について説明します。コマンドラインのかわりに、SQLJ のプロパティ・ファイルを使用して、大半のコマンドを設定できます。環境変数 `SQLJ_OPTIONS` は、コマンドラインと一緒にまたは単独で使用して、オプションを設定できます。

基本的なオプションの設定方法の詳細は、8-21 ページの「[基本的なトランスレータ・オプション](#)」を参照してください。

より高度なオプションの詳細は、8-57 ページの「[拡張トランスレータ・オプション](#)」および 8-76 ページの「[トランスレータによる代替環境のサポートとオプション](#)」を参照してください。

`sqlj` スクリプトによって Java Virtual Machine (JVM) を起動し、SQLJ トランスレータのクラス名 (`sqlj.tools.Sqlj`) を JVM に渡します。トランスレータの起動や、コマンドラインとプロパティ・ファイルの解析などは、JVM によって実行されます。ここでは、スクリプトの実行を SQLJ の実行と呼び、スクリプトのコマンドラインを SQLJ コマンドラインと呼びます。

コマンドラインの代表的な構文は、次のとおりです。

```
sqlj <optionlist> filelist
```

optionlist は、SQLJ オプションの設定値をスペースで区切って並べたものです。Java インタプリタ、コンパイラおよびカスタマイザに渡すオプションには、(必要に応じて) 接頭辞を付けます。

filelist は、SQLJ トランスレータで処理するファイルをスペースで区切って並べたものです。8-12 ページの「[コマンドラインの構文と処理](#)」で説明されているように、`.sqlj`、`.java`、`.ser` または `.jar` を指定できます。ファイル名には、* ワイルドカードを使用できます。たとえば、`Foo*.sqlj` では、`Foo1.sqlj`、`Foo2.sqlj` および `FooBar.sqlj` が検索されます。

注意：

- オプションを指定する位置は、必ずしもファイル・リストの前でなくてもかまいません。オプションは、コマンドラインのどの箇所に指定した場合にも正常に処理されます。
 - すべてのコマンドラインのオプションは、変換されるすべてのファイルに適用されます。ファイル固有のオプションは設定できません。
-

ファイル・リストに `.class` ファイルは指定しないでください。SQLJ トランスレータが、SQLJ ソース・ファイル中の変数の型解決をするためのクラスを認識できるように、`CLASSPATH` を設定してください。

`-checksource` フラグが使用できる場合（デフォルト設定）、SQLJ トランスレータでは、`CLASSPATH` 内のコンパイルされていない `.java` ファイルで必要なクラスも検索できます。8-65 ページの「[型解決を目的としたソース・チェック（-checksource）](#)」を参照してください。

注意：

- SQLJ コマンドラインの説明は、サーバー側の変換ではなく、クライアント側の変換を対象としています。サーバー側の SQLJ オプションの指定はメカニズムが異なり、オプションの小規模なサブセットのみがサポートされます。詳細は、11-16 ページの「[サーバー側の埋込みトランスレータでサポートされるオプション](#)」を参照してください。
 - ただ単に `sqlj` と入力してスクリプトを実行すると、使用頻度の最も高い SQLJ オプション一覧が表示できます。処理対象ファイルを指定しない場合も、スクリプト実行のたびにこのオプション一覧が表示されます。つまり `-help` フラグを設定した場合と同じ結果になります。
-

SQLJ のオプション、フラグおよび接頭辞

ここでは、SQLJ トランスレータに対して指定できるオプションについて説明します。グループ型オプションは、フラグと呼ばれます。JVM（SQLJ スクリプトから起動）、および Java コンパイラと SQLJ プロファイル・カスタマイザ（JVM から起動）に渡すオプションには、接頭辞を付けます。

SQLJ オプションの概説

次の表 8-1 は、SQLJ トランスレータでサポートされているオプションの一覧です。次のカテゴリに分類されています。

- 「コマンドラインのみ」のフラグ、オプションおよび接頭辞。これらは、プロパティ・ファイル中には設定できないオプションです。
- 基本カテゴリのフラグとオプション。詳細は、8-21 ページの「[基本的なトランスレータ・オプション](#)」を参照してください。
- 「高度」カテゴリのフラグ、オプションおよび接頭辞。詳細は、8-57 ページの「[拡張トランスレータ・オプション](#)」を参照してください。
- 「環境」カテゴリのフラグとオプション。詳細は、8-76 ページの「[トランスレータによる代替環境のサポートとオプション](#)」を参照してください。このカテゴリのフラグとオプションは、非標準の JVM、コンパイラまたはカスタマイザを使用するためのものです。

- 「javac」カテゴリのオプションは、SQLJ がコンパイラ接頭辞なしで直接認識する javac オプションです。javac オプションは Java コンパイラ（通常は javac）に渡され、その一部は SQLJ トランスレータ設定にも影響します。詳細は、8-10 ページの「[javac のオプションのサポート](#)」を参照してください。

オプション、フラグおよび接頭辞に関する注意

- コマンドラインのオプション名は、大文字 / 小文字が区別されます。他の実行可能プログラムに渡すオプションも同様です。通常、すべて小文字にします。通常、オプション値も大文字 / 小文字が区別されます。
- いくつかのオプションは、次の表 8-1 に示すとおり、Oracle の loadjava ユーティリティとの互換性をサポートするために、コマンドラインで代替構文を使用できます。
- SQLJ の大半のオプションは、プロパティ・ファイルでも設定できます。8-16 ページの「[プロパティ・ファイルによるオプション設定](#)」を参照してください。
- 環境変数 SQLJ_OPTIONS をコマンドラインのかわりに使用するか、またはコマンドラインと併用して、オプションを設定できます。8-19 ページの「[環境変数 SQLJ_OPTIONS によるオプションの設定](#)」を参照してください。
- このマニュアルの説明では、ほとんどの場合、ブール・フラグの設定値を true または false で示していますが、yes/no、on/off または 1/0 でも有効化 / 無効化を行えます。

コマンドラインの構文と処理については、8-12 ページの「[コマンドラインの構文と処理](#)」を参照してください。

表 8-1 SQLJ トランスレータのオプション

オプション	説明	デフォルト	カテゴリ
-bind-by-identifier	指定された SQLJ 文の中で複数回出現する同一のホスト変数を、単一のバインドとして処理するフラグです。	FALSE	高度
-C	Java コンパイラに渡すオプションに付ける接頭辞です。	該当なし	高度
-cache	オンライン・セマンティクス・チェックの結果のキャッシングを有効化します（キャッシングすると、データベースへのラウンドトリップが削減されます）。	FALSE	高度
-checkfilename	ソース・ファイル名とそのソース・ファイル中の Public クラス（定義済みの場合）の名前とが対応していない場合に、変換時に警告を出すかどうかを指定します。	TRUE	環境

表 8-1 SQLJ トランスレータのオプション（続き）

オプション	説明	デフォルト	カテゴリ
-checksource	状況によってはクラス・ファイルに加えてソース・ファイルも、SQLJ の型解決の検査対象に指定します。	TRUE	高度
-classpath (コマンドラインのみ)	JVM と Java コンパイラに対して CLASSPATH を指定します (javac にも渡されます)。	なし	基本
-codegen	コード生成のモードを指定します。oracle で直接 Oracle JDBC コールによる Oracle 固有コード生成を指定し、iso で ISO 標準 SQLJ コード生成を指定します。	oracle	基本
-compile	Java のコンパイル処理を有効化 / 無効化します。対象は、SQLJ の現在の実行で生成されたまたは以前に生成された .java ファイル、あるいはコマンドラインで指定された .java ファイルです。	TRUE	高度
-compiler-executable	使用する Java コンパイラを指定します。	javac	環境
-compiler-encoding-flag	SQLJ で -encoding 値を Java コンパイラに渡すかどうかを設定します (encoding 設定時)。	TRUE	環境
-compiler-output-file	Java コンパイラの出力を書き込むファイルを指定します。このオプションを設定しないと、コンパイラの出力は標準出力に送られます。	なし	環境
-compiler-pipe-output-flag	javac.pipe.output システム・プロパティを設定するかどうかを SQLJ に指示します。このフラグを設定すると、Java コンパイラがエラーとメッセージを、STDERR ではなく、STDOUT に出力します。	TRUE	環境
-d	SQLJ で生成される .ser プロファイル・ファイル (生成される場合) およびコンパイラで生成される .class ファイルの出力先ディレクトリを指定します (javac にも渡されます)。	空 (.class ファイルの場合は .java ファイルのディレクトリを使用し、.ser ファイルの場合は .sqlj ファイルのディレクトリを使用します。)	基本
-default-customizer	使用するプロファイル・カスタマイザを判断します。クラス名を指定します。	oracle.sqlj.runtime.util.OraCustomizer	環境
-default-url-prefix	URL 設定のデフォルト接頭辞を設定します。	jdbc:oracle:thin:	基本

表 8-1 SQLJ トランスレータのオプション（続き）

オプション	説明	デフォルト	カテゴリ
-depend (プロパティ・ファイルに設定されている場合はコンパイラ接頭辞が必要)	javac に渡されます。-checksource が有効になります。	該当なし	javac
-deprecation (プロパティ・ファイルに設定されている場合はコンパイラ接頭辞が必要)	javac にのみ渡されます。	該当なし	javac
-dir	SQLJ により生成される .java ファイルの出力ディレクトリを設定します。	空 (.sqlj 入力ファイルのディレクトリが使用されます。)	基本
-driver	登録する JDBC ドライバ・クラスを判断します。クラス名またはクラス名のカンマ区切りのリストを指定します。	oracle.jdbc. OracleDriver	基本
-encoding (コマンドラインでは、-e としても認識)	SQLJ とコンパイラでグローバル化・サポートに使用するエンコーディングを指定します。javac にも渡されます。	JVM の file.encoding の設定	基本
-explain	トランスレータのエラー・メッセージに「原因」と「処置」情報の表示を要求するフラグです。	FALSE	基本
-fixedchar	CHAR データと比較するために文字列を WHERE 句にバインドするときに、空白埋めを考慮するフラグです。	FALSE	基本
-g (プロパティ・ファイルに設定されている場合はコンパイラ接頭辞が必要)	javac に渡されます。-linemap が有効になります。	該当なし	javac
-help (-h としても認識) -help-long -help-alias (すべてコマンドラインのみ)	SQLJ オプションの名前、説明および現行値に関する様々なレベルの情報を表示するフラグです。	無効	基本
-jdblinemap	-linemap オプションの改良型です。Sun 社の jdbc デバッガで使用されます。	FALSE	基本
-J (コマンドラインのみ)	JVM に渡すオプションに付ける接頭辞です。	該当なし	高度
-linemap	生成された Java クラス・ファイルと元の SQLJ コード間での行番号のマッピングを有効化します。	FALSE	基本

表 8-1 SQLJ トランスレータのオプション（続き）

オプション	説明	デフォルト	カテゴリ
-n (コマンドラインのみ。代替 -vm=echo)	SQLJ トランスレータに渡すコマンドライン全体（SQLJ_OPTIONS の設定も含む）のエコーを sqlj スクリプトに指示します。SQLJ トランスレータはコマンドラインを実行しません。	無効	基本
-nowarn (プロパティ・ファイルに設定されている場合はコンパイラ接頭辞が必要)	javac に渡され、-warn=none が設定されます。	該当なし	javac
-O (プロパティ・ファイルに設定されている場合はコンパイラ接頭辞が必要)	javac に渡されると、-linemap は無効化されます。	該当なし	javac
-offline	セマンティクス・チェックに使用するオフライン・チェッカを判断します。クラスの完全修飾名のリストを指定します。	oracle.sqlj.checker. OracleChecker	高度
-online	セマンティクス・チェックに使用するオンライン・チェッカを判断します。クラスの完全修飾名を指定します。（オンライン・チェックを有効にするには、-user の設定も必要です。）	oracle.sqlj.checker. OracleChecker	高度
-optcols	パフォーマンスが最適化されるようにイテレータ列型とサイズ定義を有効化します。Oracle 固有コード生成ではトランスレータが直接使用します。また、ISO コード生成ではユーザー、パスワード、URL の設定とともに Oracle カスタマイザに転送されます。	FALSE	基本
-optparams	JDBC リソース割当てが最適化されるようにパラメータ・サイズ定義を有効化します（-optparamdefaults と併用します）。Oracle 固有コード生成ではトランスレータが直接使用します。また、ISO コード生成では Oracle カスタマイザに転送されます。	FALSE	基本
-optparamdefaults	特定のデータ型にパラメータ・サイズのデフォルトを設定します（-optparams と併用します）。Oracle 固有コード生成ではトランスレータが直接使用します。また、ISO コード生成では Oracle カスタマイザに転送されます。	FALSE	基本
-P	SQLJ プロファイル・カスタマイザに渡すオプションに付ける接頭辞です。	該当なし	高度

表 8-1 SQLJ トランスレータのオプション（続き）

オプション	説明	デフォルト	カテゴリ
-parse	オフライン SQL パーサーを有効にするオプションです。使用可能な設定は、both、online-only、offline-only、none または代替パーサーを実装する Java クラス名です。 注意： このオプションの設定によっては、-user オプションの効果が無視され、オンライン・セマンティクス・チェックも無効になります。	both	高度
-passes (コマンドラインのみ)	コンパイルを挟んで、SQLJ を 2 回実行することを sqlj スクリプトに指示します。	FALSE	環境
-password (コマンドラインでは、-p としても認識)	オンライン・セマンティクス・チェック用のデータベース接続のユーザー・パスワードを設定します。	なし	基本
-profile	ISO コード生成の場合に、現在の SQLJ の実行で生成されるプロファイル・ファイルのプロファイル・カスタマイズ処理を有効化 / 無効化します。	TRUE	高度
-props (コマンドラインのみ)	プロパティ・ファイルを指定します。コマンドラインのかわりに、このプロパティ・ファイルに基づいてオプションが設定されます。(sqlj.properties も読み込まれます。)	なし	基本
-ser2class	ISO コード生成の場合に、生成した .ser プロファイルを .class ファイルに変換するように SQLJ に指示します。	FALSE	高度
-status (コマンドラインでは、-v としても認識)	実行中のステータス・メッセージの表示を SQLJ に要求します。	FALSE	基本
-url	オンライン・セマンティクス・チェック用のデータベース接続の URL を設定します。	jdbc:oracle:oci:@	基本
-user (コマンドラインでは、-u としても認識)	オンライン・セマンティクス・チェックを有効にし、データベース接続のユーザー名（およびパスワードと URL）を設定します。	なし（オンライン・セマンティクス・チェックはありません。)	基本
-verbose (プロパティ・ファイルに設定されている場合はコンパイラ接頭辞が必要)	javac に渡されると、-status が有効になります。	該当なし	javac

表 8-1 SQLJ トランスレータのオプション (続き)

オプション	説明	デフォルト	カテゴリ
-version -version-long (どちらもコマンドラインのみ)	SQLJ および JDBC ドライバのバージョン情報に関する様々なレベルの情報を表示するフラグです。	無効	基本
-vm (コマンドラインのみ)	JVM を指定して SQLJ トランスレータの実行に使用します。	java	環境
-warn	各種の SQLJ 警告を有効化または無効化するフラグのカンマ区切りリストです。個別フラグは、cast/nocast、precision/noprecision、nulls/nonulls、portable/noportable、strict/nostRICT および verbose/noverbose です。グローバル・フラグは all/none です。	cast precision nulls noportable strict noverbose	基本

loadjava 互換オプション

Java および SQLJ アプリケーションを Oracle9i にロードするには、loadjava ユーティリティを使用します。このユーティリティとの互換のために、次のコマンドライン・オプションの代替構文が認識されます。(前述の表 8-1 を参照してください。)

- -e (-encoding)
- -h (-help)
- -p (-password)
- -u (-user)
- -v (詳細メッセージの出力。-status に相当。)

loadjava 構文と完全に一貫させるには、次のように「=」のかわりにスペースを使用してこれらのオプションを設定します。

```
-u scott/tiger -v -e SJIS
```

loadjava ユーティリティの概要は、『Oracle9i Java Developer's Guide』を参照してください。

注意： この代替オプション構文はコマンドラインまたは環境変数 SQLJ_OPTIONS でのみ認識されます。プロパティ・ファイルでは認識されません。

javac のオプションのサポート

SQLJ は、Sun 社の JDK の Java コンパイラ javac のオプション設定を各種方法でサポートしています。

- 値を伴う一部の javac オプションは、SQLJ のオプションと一体化しています (-classpath、-d、-encoding)。
- 値を伴うその他の javac オプションについては、コンパイラに値を正確に渡すように特別な処理が実装されました (-bootclasspath、-extdirs、-target)。これらのオプションには、コンパイラの接頭辞が必要です。これらのオプションが SQLJ の操作に影響することはありません。
- javac のフラグは、コマンドラインでは compiler 接頭辞なしで認識されます (-depend、-deprecation、-g、-nowarn、-O、-verbose)。これらのフラグの一部は、SQLJ トランスレータ・フラグの設定にも影響を与えます。

表 8-2 に要約を示します。表に注記されているように、コンパイラ接頭辞が必須の場合もありますが、これらのオプションはすべて SQLJ コマンドラインまたはプロパティ・ファイルに設定できます。

注意：

- デフォルトの場合、javac は、送信されたプラットフォームのブートストラップ・クラスと拡張クラスに対して、クラスをコンパイルします。一方、javac は、異なる Java プラットフォームのブートストラップ・クラスと拡張クラスに対するクロスコンパイル・クラスもサポートします。javac の -bootclasspath オプションと -extdirs オプションは、クロスコンパイル専用です (JDK 1.2 以上)。
- デフォルトの場合、javac は、JDK 1.1 および JDK 1.2 以上の双方に互換性のある .class ファイルを生成します。-target オプションを使用して、この設定を変更します。

表 8-2 SQLJ による javac オプションのサポート

コマンドライン・ オプション (-C が 記されている場合 は -C 接頭辞付き)	説明	SQLJ との関連
-C-bootclasspath	指定されている一連のブートストラップ・クラスに対するクロスコンパイルを javac に指示します。	なし
-classpath	javac と JVM の CLASSPATH を設定します。	SQLJ のオプションも設定します。

表 8-2 SQLJ による `javac` オプションのサポート (続き)

コマンドライン・オプション (-C が記されている場合は -C 接頭辞付き)	説明	SQLJ との関連
-d	.class ファイルと SQLJ プロファイル・ファイルの出力ディレクトリを設定します。	SQLJ のオプションも設定します。
-depend	依存性のある古いファイルの再帰的コンパイルを <code>javac</code> に指示します。	SQLJ <code>-checksource</code> オプションを有効化します。
-deprecation	現在では使用されなくなった API が使用されている出力ソースの位置を <code>javac</code> に指示します。	なし
-encoding	SQLJ と <code>javac</code> の両方に対してエンコーディングを設定します。	SQLJ のオプションも設定します。
-C-extdirs	指定されている拡張ディレクトリに対するクロスコンパイルを <code>javac</code> に指示します。	なし
-g	<code>javac</code> デバッグ情報を生成します。	SQLJ <code>-linemap</code> オプションを有効化します。
-nowarn	警告を生成しないことを <code>javac</code> に指示します。	SQLJ オプション <code>-warn=none</code> を設定します。
-O	<code>javac</code> に最適化を指示します。	SQLJ <code>-linemap</code> オプションを無効化します。
-C-target	指定の JDK バージョン・レベル以上 (1.1、1.2 または 1.3) の JVM でのみ動作するように .class ファイルを生成することを <code>javac</code> に指示します。	なし
-verbose	<code>javac</code> にリアルタイム・ステータス・メッセージの出力を指示します。	SQLJ <code>-status</code> オプションを有効化します。

`javac` オプションの設定値と機能の詳細は、`javac` のマニュアルを参照してください。

`javac` オプションの構文に関する注意事項

- Java コンパイラの CLASSPATH と SQLJ を実行する JVM の CLASSPATH を異なる値に設定する場合は、前者に `-c` 接頭辞、後者に `-j` 接頭辞を使用して個別に設定する必要があります。それ以外の場合、接頭辞は不要です。
- `-d` または `-encoding` コンパイラ・オプションを指定するときに、`-c` 接頭辞を付けないでください。SQLJ とコンパイラは、`-d` と `-encoding` で同じ設定を使用します。

- 必要に応じて `-C` 接頭辞を `-depend`、`-deprecation`、`-g`、`-nowarn`、`-O` および `-verbose` に使用できます。
- SQLJ のオプションでもあるオプション (`-classpath`、`-d` および `-encoding`) を除くすべての `javac` オプションは、プロパティ・ファイルに設定する場合には「`compile.`」接頭辞が必要です。
- 一貫性を保持するために、値を伴うオプションには等号 (=) を使用することをお勧めします。ただし、コンパイラ接頭辞（コマンドラインでは `-C`、プロパティ・ファイルでは「`compile.`」）を使用する場合はスペースも使用できます。

例 次の構文は、`-C-bootclasspath`、`-C-extdirs` および `-C-target` オプションの使用例です（折り返されて表示されていますが、全体が 1 行で入力されています）。

```
% sqlj -vm=/usr/local/packages/jdk1.2.2/bin/java
        -compiler-executable=/usr/local/packages/jdk1.2.2/bin/javac
        -C-bootclasspath=/usr/local/packages/jdk1.3.1/jre/lib/rt.jar
        -C-extdirs="" -C-target=1.1.8 Demo.sqlj
```

プロファイル・カスタマイザのオプション

プロファイル・カスタマイザのオプションであるカスタマイザ・ハーネスのフロントエンド、Oracle デフォルトのカスタマイザおよびデバッグや配布時のセマンティクス・チェックに使用する特別なカスタマイザの詳細は、A-10 ページの「[カスタマイズ・オプションとカスタマイザの選択](#)」を参照してください。これは、ISO 標準コード生成の場合（`-codegen=iso`）にのみ使用します。

コマンドラインの構文と処理

スクリプト `sqlj` の実行によって発生するイベントの一般的なシーケンスは、1-9 ページの「[SQLJ 変換処理](#)」を参照してください。ここでは、コマンドラインの概要説明として、変換処理の詳細を説明します。

コマンドライン引数の使用方法

通常、コマンドラインでは次の構文を使用します。

```
sqlj <optionlist> filelist
```

JVM を起動するときに、`sqlj` スクリプトから JVM にすべてのコマンドライン引数が渡されます。JVM は渡された引数を該当機能（Java コンパイラやプロファイル・カスタマイザなど）に渡します。

等号 (=) を使用して、オプションとフラグの設定値を指定します。ただし、フラグをオンにするには、`=true` を指定する必要はありません。フラグ名の入力のみで十分です。ただし、フラグをオフにするには、`=false` を指定する必要があります。フラグが前の値から切り替わらないためです。次にその例を示します。

行マッピングを有効にするには、`-linemap=true` または単に `-linemap` を指定します。

行マッピングを無効化するには、`-linemap=false` を指定します。

注意： コマンドライン（またはプロパティ・ファイル）で同じオプションを 2 回以上指定すると、最後の値が使用されます。

オプション・リストの引数 オプション・リストの引数は、次のように使用されます。

- 接頭辞 `-J`、`-C` または `-P` が付いていないオプションは **SQLJ** のオプションです（直接サポートされるコンパイラ・オプションは除きます）。**SQLJ** トランスレータを起動するときに、**JVM** から **SQLJ** トランスレータに渡されます。
- `-J` 接頭辞が付いたオプションは **JVM** のオプションです。**JVM** でそのまま使用されます。これらのオプションは、コマンドラインまたは環境変数 `SQLJ_OPTIONS` で指定する必要があります。トランスレータのオプションと同様に、オプションの設定には等号（`=`）を使用します。

```
-J-Djavac.pipe.output=true
```

Java コンパイラの `CLASSPATH` と **SQLJ** を実行する **JVM** の `CLASSPATH` を異なる値に設定する場合は、前者に `-C` 接頭辞、後者に `-J` 接頭辞を使用して個別に設定する必要があります。

- `-C` 接頭辞が付いたオプションは **Java** コンパイラのオプションです。コンパイラを起動するときに、**JVM** からコンパイラに渡されます。値を伴うコンパイラ・オプションには特別なサポートが必要であり、`javac` のオプションとして実装されました。8-10 ページの「[javac のオプションのサポート](#)」を参照してください。これらについても、次のように等号を使用できます（スペースも使用できます）。

```
-C-bootclasspath=/usr/local/packages/jdk1.3.1/jre/lib/rt.jar
```

- オプションに `-P` 接頭辞を付けると、**SQLJ** プロファイル・カスタマイザのオプションになります。カスタマイザを起動するときに、**JVM** からカスタマイザに渡されます。これは、**ISO** 標準コード生成（`-codegen=iso`）の場合にのみ使用します。トランスレータのオプションと同様に、オプションの設定には等号（`=`）を使用します。

```
-P-user=scott/tiger
```

SQLJ での自動実行の対象でないプロファイルのカスタマイズは、拡張機能といえます。[付録 A「カスタマイズおよび専用カスタマイザ」](#) を参照してください。

ファイル・リストの引数 ファイル・リストの解析、ワイルドカード文字の処理およびファイル名の展開は、SQLJ フロントエンドによって行われます。デフォルトのファイル処理は、次のように展開されます。

- `.sqlj` ファイルが SQLJ トランスレータ、Java コンパイラおよび SQLJ プロファイル・カスタマイザによって処理されます（プロファイル・カスタマイザは `-codegen=iso` の場合のみ）。
- `.java` ファイルが Java コンパイラによって処理されます。SQLJ トランスレータは、このファイルを使用して型を解決します。
- `.ser` プロファイル・ファイルと `.jar` ファイルは、プロファイル・カスタマイザでのみ処理されます（`-codegen=iso` の場合のみ）。

コマンドラインで `.sqlj` ファイルと `.java` ファイルを一緒に指定することも、`.ser` ファイルと `.jar` ファイルを一緒に指定することも可能ですが、この 2 つのカテゴリを一緒に指定できません。（`.jar` ファイルの処理方法の詳細は、A-36 ページの「[プロファイルの JAR ファイル](#)」を参照してください。）

`.sqlj` ファイルと `.java` ファイル間に依存関係がある場合、つまり互いのコードにアクセスする必要がある場合は、SQLJ の各実行時に、依存し合うすべてのファイルをコマンドラインに入力する必要があります。依存関係のあるファイルをそれぞれ別の SQLJ 実行時に指定すると、SQLJ がすべての型を解決できなくなります。

注意： コマンドラインで `.java` ファイル名を入力する方法以外に、`-checksource` オプションを有効にし、その後で `.java` ファイルが CLASSPATH に含まれていることを確認する方法もあります。8-65 ページの「[型解決を目的としたソース・チェック \(-checksource\)](#)」を参照してください。

ソース間の競合防止 SQLJ トランスレータでは、同一ディレクトリの同一クラスを複数のソース・ファイルで定義できないようになっています。コマンドラインのファイル・リストに同じ `.sqlj` ファイルまたは `.java` ファイルへの参照が複数あると、2 番目以降の参照がすべてコマンドラインから破棄されます。また、リスト内の `.java` ファイルと `.sqlj` ファイルのベース名とディレクトリが同じときに、`-dir` オプションを指定しないと、`.sqlj` ファイルのみが処理されます。ファイル名にワイルドカード文字を使用した場合も、この処理が適用されます。

次のコマンドラインを指定するとします。カレント・ディレクトリ `/myhome/mypackage` に、ファイル `Foo.sqlj` と `Foo.java` があると想定します。

- `sqlj Foo.sqlj /myhome/mypackage/Foo.sqlj`
両方の参照先が同じファイルなので、トランスレータは `/myhome/mypackage/Foo.sqlj` をコマンドラインから破棄します。

- `sqlj Foo.sqlj Foo.java`

トランスレータは `Foo.java` をコマンドラインから破棄します。そうしないと、このコマンドラインは、トランスレータが `Foo.java` との読み書きを両方とも同時に実行することになります。

- `sqlj Foo.*`

この場合も、トランスレータは `Foo.java` をコマンドラインから破棄します。そうしないと、トランスレータは `Foo.sqlj` と `Foo.java` の両方を検索するので、書込み先と読込み元が同時に `Foo.java` になります。

- `sqlj -dir=outdir -d=outclasses Foo.sqlj Foo.java`

問題ありません。`Foo.java` が `outdir` サブディレクトリに生成され、`Foo.java` が `/myhome/mypackage` ディレクトリから読み込まれます。`Foo.java` と `Foo.sqlj` で別のパッケージのクラスを定義していると想定します。この場合、Java コンパイルで作成された `.class` ファイルは、`outclasses` ディレクトリ階層下の別のサブディレクトリに配置されます。

このようにコマンドラインが処理されるので、次のようなコマンドを入力しても、問題なく実行されます。(問題のあるファイル参照は自動的に破棄されます。)

```
sqlj *.sqlj *.java
```

この処理は様々な状況で利用できます。

コマンドラインの例と結果

次に、コマンドラインの例を示します。高度な概念が使用されていますが、コマンドラインの完全な構文例として示します。高度な概念については、この章で後述します。

```
sqlj -J-Duser.language=ja -warn=none -J-prof -encoding=SJIS *Bar.sqlj Foo*.java
```

`sqlj` スクリプトによって JVM が起動されると、この JVM に SQLJ トランスレータのクラス名が渡され、その後でコマンドラインの引数が渡されます。SQLJ のオプションは、JVM によってトランスレータとコンパイラに渡されます。JVM に対するオプション (`-J` が付いたオプション) があると、トランスレータのクラス・ファイル名の前に、このオプションが JVM に渡されます。(Java を手動で起動する場合は、クラス・ファイル名の前に、Java のオプションを入力します。) この例ではデフォルトの Oracle 固有コード生成を使用しているため、カスタマイズはありません。

前述の処理が完了すると、ユーザーが次のように入力したときと同じ結果になります (`SushiBar.sqlj`、`DiveBar.sqlj`、`FooBar.java` および `FooBaz.java` がすべてカレント・ディレクトリにある場合)。

```
java -Duser.language=ja -prof sqlj.tools.Sqlj -warn=none -encoding=SJIS
SushiBar.sqlj DiveBar.sqlj FooBar.java FooBaz.java
```

(このコマンドラインは折り返されて表示されていますが、全体が 1 行で入力されています。)

JVM オプションの処理方法の詳細は、8-58 ページの「[Java Virtual Machine \(JVM\) に渡すオプション \(-J\)](#)」を参照してください。

コマンドラインを実行せずにエコーする場合

コマンドラインをエコーするには、SQLJ の `-n` オプション (または `-vm=echo`) を使用します。sqlj スクリプトによって生成され、SQLJ トランスレータに渡されるコマンドラインが、実行されずに、エコーされます。エコー対象は、環境変数 `SQLJ_OPTIONS` およびコマンドラインの設定値です。プロパティ・ファイルの設定値はエコーされません。詳細は、8-26 ページの「[コマンドラインのエコー \(-n\)](#)」を参照してください。

プロパティ・ファイルによるオプション設定

コマンドラインのかわりに、プロパティ・ファイルでも、SQLJ トランスレータ、Java コンパイラおよび SQLJ プロファイル・カスタマイザに対してオプションを指定できます (プロファイル・カスタマイザは ISO コード生成の場合のみ)。

Java コンパイラを別の JVM で実行し、この JVM に対してコンパイル処理オプションを指定する場合も、プロパティ・ファイルで指定できます。SQLJ での変換後、コンパイラの実行時に、オプションが JVM に渡されます。(ただし、コマンドラインで `-c-J` 接頭辞を付けて、コンパイラの JVM に渡す方が一般的です。)

SQLJ の次のオプション、フラグおよび接頭辞は、プロパティ・ファイルで設定できません。

- `-classpath`
- `-help`、`-help-long`、`-help-alias`、`-C-help`、`-P-help`
- `-J`
- `-n`
- `-passes`
- `-props`
- `-version`、`-version-long`
- `-vm`

たとえば、JVM に対するオプションはプロパティ・ファイルで指定できません。プロパティ・ファイルの読み込みが、JVM の起動後に実行されるためです。

また、プロパティ・ファイルでは、`loadjava` との互換性のためにコマンドラインでは認められているオプションの略称は使用できません (`-e`、`-h`、`-p`、`-u`、`-v`)。

注意： SQLJ のプロパティ・ファイルの説明は、サーバー側 SQLJ ではなく、クライアント側 SQLJ を対象にしています。サーバー側の SQLJ オプションの指定はメカニズムが異なり、オプションの小規模なサブセットのみがサポートされます。詳細は、11-16 ページの「[サーバー側の埋込みトランスレータでサポートされるオプション](#)」を参照してください。

プロパティ・ファイルの構文

プロパティ・ファイルでは、1 行に 1 つのオプションを設定します。SQLJ のオプション行、コンパイラのオプション行およびカスタマイザのオプション行を混在できます。SQLJ フロントエンドによってオプション行が解析され、適切に処理されます。

次に、各種オプションの構文を示します。

- 各 SQLJ オプションの前に、ハイフンではなく、接頭辞「sqlj.」（ピリオドまで）を付けます。この接頭辞で始まるオプションのみが SQLJ トランスレータに渡されます。次にその例を示します。

```
sqlj.warn=none
sqlj.linemap=true
```

- 各 Java コンパイラ・オプションの前に、「-c-」ではなく、接頭辞「compile.」（ピリオドまで）を付けます。この接頭辞で始まるオプションが Java コンパイラに渡されます。次にその例を示します。

```
compile.verbose
compile.bootclasspath=/usr/local/packages/jdk1.3.1/jre/lib/rt.jar
```

- 汎用的なプロファイル・カスタマイズ・オプション（どのカスタマイザにも適用されるオプション）には、「-p-」ではなく、接頭辞「profile.」（ピリオドまで）を付けます。この接頭辞で始まるオプションのみがプロファイル・カスタマイザに渡されます。次にその例を示します。

```
profile.backup
profile.user=scott/tiger
```

特定のカスタマイザに対するオプションを指定するには、次のように「profile.c」を使用します。

```
profile.Csummary
profile.Coptparamdefaults=VAR%(50),LONG%(500),RAW_TYPE()
```

Oracle のデフォルト・カスタマイズの対象でないプロファイルのカスタマイズは、拡張機能といえます。付録 A「[カスタマイズおよび専用カスタマイザ](#)」を参照してください。

- コメント行はシャープ記号 (#) で始めます。次にその例を示します。

```
# Comment line.
```

- 空白行も使用できます。

コマンドラインと同じように、プロパティ・ファイルでもフラグの有効化 / 無効化を `=true/=false`、`=on/=off`、`=1/=0` または `=yes/=no` で指定できます。フラグを有効にするには、次のようにフラグ名の入力のみで十分です。設定値の入力は必要ありません。

```
sqlj.linemap
```

注意： 一貫性を保持するために、プロパティ・ファイル内の値を伴うオプションには（状況によってスペースが使用できる場合にも）常に等号 (=) を使用することが最善の方法です。

プロパティ・ファイル：簡単な例 プロパティ・ファイルの設定例を示します。

```
# Set user and JDBC driver
sqlj.user=scott
sqlj.driver=oracle.jdbc.OracleDriver

# Turn on the compiler verbose option
compile.verbose
```

この設定は、次の SQLJ コマンドラインに相当します。

```
sqlj -user=scott -driver=oracle.jdbc.OracleDriver -C-verbose
```

プロパティ・ファイル：デフォルト以外の接続コンテキスト・クラス 宣言した接続コンテキスト・クラス `SourceContext` の設定値をプロパティ・ファイルで指定する例を示します。

```
# JDBC driver
sqlj.driver=oracle.jdbc.OracleDriver

# Oracle 8.0.4 on spock.natdecsys.com
sqlj.user@SourceContext=sde
sqlj.password@SourceContext=fornow
sqlj.url@SourceContext=jdbc:oracle:thin:@207.67.155.3:1521:nds

# Warning settings
sqlj.warn=all

# Cache
sqlj.cache=on
```

デフォルトのプロパティ・ファイル

SQLJ コマンドラインでプロパティ・ファイルを指定してもしなくても、`sqlj.properties` というファイルが SQLJ フロントエンドで検索されます。このファイルの検索は、Java ホーム・ディレクトリ、ユーザー・ホーム・ディレクトリ、カレント・ディレクトリの順番で実行されます。見つかった `sqlj.properties` ファイルが順番に処理され、オプションの前の設定値が新しい設定値でオーバーライドされます。つまり、カレント・ディレクトリにある `sqlj.properties` ファイルのオプション値によって、ユーザーのホーム・ディレクトリまたは Java のホーム・ディレクトリにある `sqlj.properties` ファイルのオプション値がオーバーライドされます。

8-20 ページの「[オプション設定の優先順位](#)」も参照してください。

環境変数 SQLJ_OPTIONS によるオプションの設定

Oracle SQLJ では、コマンドラインのかわりに環境変数 `SQLJ_OPTIONS` を使用して、SQLJ のオプションを設定できます。「コマンドラインのみ」と示されているオプションは、プロパティ・ファイルでは設定できませんが、`SQLJ_OPTIONS` 変数では設定できます。

`SQLJ_OPTIONS` 変数では、あらゆる SQLJ オプションを設定できます。この変数で設定したオプション値は、JVM に渡されます。この変数は、同じ設定値を繰り返し使用するコマンドライン専用オプション（`-classpath` など）に特に便利です。

`SQLJ_OPTIONS` 変数は、次のように設定します。

```
-vm=jview -J-verbose
```

`SQLJ_OPTIONS` を使用すると、SQLJ によって `SQLJ_OPTIONS` の設定値が、SQLJ コマンドラインの先頭に、つまり他のコマンドライン・オプションの設定の前に、順番に挿入されます。

注意： 一般的に `SQLJ_OPTIONS` の構文はコマンドラインでの場合と同じですが、これはオペレーティング・システムによって異なります。オペレーティング・システム固有の制限が存在することもあります。たとえば Windows 95 では、「コントロール パネル」の「システム」の「ハードウェア環境」タブを使用します。また、Windows 95 では変数の設定に「=」文字を使用できないので、SQLJ では「=」のかわりに「#」を使用して `SQLJ_OPTIONS` を設定できます。詳細は、使用するオペレーティング・システムのドキュメントを参照してください。

オプション設定の優先順位

SQLJ では、オプション値が次の順に設定されます。同じオプションが複数回設定されていると、前の設定値が後の設定値によって上書きされます。

1. オプションをデフォルト値に設定します（デフォルト値がある場合）。
2. Java のホーム・ディレクトリで `sqlj.properties` ファイルを探します。見つかると、このファイルの指定に基づき、オプションを設定します。
3. ユーザーのホーム・ディレクトリで `sqlj.properties` ファイルを探します。見つかると、このファイルの指定に基づき、オプションを設定します。
4. カレント・ディレクトリで `sqlj.properties` ファイルを探します。見つかると、このファイルの指定に基づき、オプションを設定します。
5. 環境変数 `SQLJ_OPTIONS` でオプション設定を探します。見つかったオプション設定をコマンドラインの先頭に挿入します。`SQLJ_OPTIONS` の指定に基づき、オプション値を設定します。
6. コマンドラインでオプション設定を探し、この指定に基づき、オプション値を設定します。`SQLJ` はコマンドラインを処理しながら、`-props` オプションで指定されているファイルを調べ、その指定に基づき、オプション値を設定します。

注意：

- `sqlj.properties` ファイルのオプション設定は、上から下に読み込まれます。つまり、下方のエントリが上方のエントリより優先されます。
 - コマンドラインの `-props` オプションでプロパティ・ファイルを指定すると、このファイルのオプション設定がコマンドラインの `-props` オプションの位置に挿入されます。
 - `SQLJ` では、コマンドラインのオプションが、`-props` ファイルから挿入されたオプションも含めて、左から右に読み取られます。つまり、後（右方）の設定値が前（左方）の設定値より優先されます。
-
-

例 `SQLJ` を次のように実行するとします。

```
sqlj -user=scott -props=myprops.properties -dir=/home/java
```

ファイル `myprops.properties` はカレント・ディレクトリにあり、次の値が設定されているものとします。

```
sqlj.user=tony  
sqlj.dir=/home/myjava
```


これらの設定値は、コマンドラインの `-props` オプションの位置に挿入されているように処理されます。したがって、`user` オプションの設定値として、`scott` より `tony` が優先されます。一方、`dir` オプションの設定値として、`/home/myjava` より `/home/java` が優先されます。

基本的なトランスレータ・オプション

ここでは、SQLJ の実行時に指定できる基本的なフラグとオプションの構文と機能について説明します。ここで説明するオプションは、標準の操作モードで実行できます。プロパティ・ファイル (`sqlj.properties` など) でも指定できるオプションの構文も示します。(8-16 ページの「[プロパティ・ファイルによるオプション設定](#)」を参照してください。)

ここでは、次のカテゴリについて説明します。

- [基本的なコマンドライン専用オプション](#)
- [出力ファイルとディレクトリのオプション](#)
- [接続オプション](#)
- [レポートと行マッピングのオプション](#)
- [コード生成、最適化および CHAR 比較のオプション](#)

より高度なコマンドライン・フラグとオプションは、8-57 ページの「[拡張トランスレータ・オプション](#)」および 8-76 ページの「[トランスレータによる代替環境のサポートとオプション](#)」を参照してください。

基本的なコマンドライン専用オプション

次の基本的なオプションは SQLJ のコマンドラインまたは環境変数 `SQLJ_OPTIONS` でのみ指定できます。プロパティ・ファイルでは指定できません。

- `-props`
- `-classpath`
- `-help`、`-help-long`、`-help-alias`、`-P-help`、`-C-help`
- `-version`、`-version-long`
- `-n`

コマンドライン専用フラグ (`-help` フラグ群、`-version` フラグ群および `-n`) では、`=true` 構文を使用できません。これらのフラグを有効にするには、次のようにフラグ名のみに入力します。

使用可: `sqlj -version-long ...`

使用不可: `sqlj -version-long=true ...`

注意： コマンドラインまたは SQLJ_OPTIONS でのみ設定可能な拡張オプション、フラグおよび接頭辞として、`-J`、`-passes` および `-vm` があります。

入力プロパティ・ファイル (-props)

`-props` オプションでは、プロパティ・ファイルを指定します。SQLJ はこのファイルからオプション設定を読み取ります（コマンドラインでオプション設定を指定するかわりに、このオプションを指定できます）。

このファイル形式、コマンドライン・オプションとこのファイルの関係、および SQLJ がデフォルトのプロパティ・ファイルを検索するディレクトリは、8-16 ページの「[プロパティ・ファイルによるオプション設定](#)」を参照してください。

コマンドラインの構文 `-props=filename`

コマンドラインの例 `-props=myprops.properties`

プロパティ・ファイルの構文 該当なし

プロパティ・ファイルの例 該当なし

デフォルト値 なし

Java Virtual Machine (JVM) とコンパイラで使用する CLASSPATH (-classpath)

Java の大半の JVM およびコンパイラへの対応として、SQLJ はコマンドラインで指定された `-classpath` オプションを認識します。このオプションを設定するときは、大半の JVM またはコンパイラと同じようにスペースを使用することも、他の SQLJ オプションと同じように「`=`」を使用することも可能です。この例（両方とも UNIX 環境で実行）を次に示します。

```
-classpath=...$ORACLE_HOME/jdbc/lib/classes12.jar:$ORACLE_HOME/jdbc/lib/nls_charset12.jar
```

または

```
-classpath ...$ORACLE_HOME/jdbc/lib/classes12.jar:$ORACLE_HOME/jdbc/lib/nls_charset12.jar
```

`-classpath` オプションは、JVM と Java コンパイラの両方に対して、Java の CLASSPATH を設定します。それぞれ別の CLASSPATH を使用する場合は、SQLJ の `-J` および `-c` 接頭辞を使用します。8-57 ページの「[オプション設定を他の実行可能プログラムに渡す接頭辞](#)」を参照してください。

注意： この章で後述する他のオプションと同じように、`-classpath` オプションの設定で「=」を使用すると、JVM およびコンパイラにオプション文字列を渡すときに、「=」がコマンドラインから削除されます。JVM とコンパイラでは、オプション設定に「=」を使用できません。

コマンドラインの構文 `sqlj -classpath=<class_path>`

コマンドラインの例 `sqlj -classpath=/jdbc-1.2.zip:/classes/bin`

プロパティ・ファイルの構文 該当なし

プロパティ・ファイルの例 該当なし

デフォルト値 なし

SQLJ のオプション情報 (-help)

コマンドラインで `-help` フラグを次の 3 通りに設定して、SQLJ オプションに関する表示情報の詳細度を指定できます。

- `-help`
- `-help-long`
- `-help-alias`

このオプションを有効にするには、次のようにコマンドラインで設定します。

`sqlj -help`

または

`sqlj -help-long`

または

`sqlj -help-alias`

`-help` フラグをこのいずれかの形式で指定すると、コマンドラインでファイル名などのオプションを指定しても、入力ファイルが変換されません。SQLJ では、トランスレータの実行かヘルプ表示のどちらか一方を指定します。同時に両方は指定できません。

プロファイル・カスタマイズまたは Java コンパイラに関する説明を表示するには、次のように `-p` および `-c` 接頭辞を付けてヘルプを要求します。これらの接頭辞の詳細は、8-57 ページの「[オプション設定を他の実行可能プログラムに渡す接頭辞](#)」を参照してください。`-help` フラグと同じように、カスタマイズまたはコンパイラのヘルプを要求すると、変換されません。

```
sqlj -P-help
```

```
sqlj -C-help
```

他のコマンドライン専用フラグと同じように、`-help`（および `-P-help` と `-C-help`）では、`=true` 構文を使用できません。有効にするには、フラグ名のみを入力します。

注意：

- `loadjava` ユーティリティとの互換用に、`-help` のかわりに `-h` をコマンドラインで指定できます。8-9 ページの「[loadjava 互換オプション](#)」を参照してください。
 - 単一のコマンドラインで `-help` フラグを複数設定できます。
`-P-help` と `-C-help` も設定できます。
 - `-P` と `-C` は通常、プロパティ・ファイルで設定できますが、`-P-help` と `-C-help` はコマンドライン専用です。
 - 処理対象ファイルを指定しないで `SQLJ` を実行した場合も、ヘルプが表示されます。つまり、`-help` の設定時と同じ結果になります。
-

-help の設定 最も基本的なヘルプを表示するには、`-help` を指定します。次の情報が表示されます。

- 使用頻度が最も高い `SQLJ` オプションの概要
- 指定できるその他の `-help` フラグ値のリスト

-help-long の設定 この設定では、`SQLJ` オプションの詳細情報が一覧表示されます。具体的には、各オプションに関して次の情報が表示されます。

- オプション名
- オプションの型（`int` や `String` など、オプションの入力値としての `Java` 型）
- 説明
- 現行の値
- 現行の値の設定方法（コマンドライン、プロパティ・ファイルまたはデフォルト）

注意： オプション設定の結果を確認するには、そのオプションと `-help-long` オプションを同一のコマンドラインで指定します。この機能は、特に複雑なオプション（`-warn` など）を指定する場合やオプションを組み合わせる場合に便利です。（`-help-long` モードでは、すべてのオプションの現行の設定値が表示されます。）

-help-alias の設定 loadjava ユーティリティ対応として、コマンドラインでサポートされている略称が一覧表示されます。

コマンドラインの構文 `sqlj help_flag_settings`

コマンドラインの例

```
sqlj -help
sqlj -help -help-alias
sqlj -help-long
sqlj -warn=none,null -help-long
sqlj -help-alias
```

プロパティ・ファイルの構文 該当なし

プロパティ・ファイルの例 該当なし

デフォルト値 無効

SQLJ のバージョン番号 (-version)

コマンドラインで `-version` フラグを次の値に設定して、表示する SQLJ および JDBC ドライバのバージョンに関して様々なレベルの情報の表示を指定します。

- `-version`
- `-version-long`

このオプションを有効にするには、次のようにコマンドラインで設定します。

```
sqlj -version
```

または

```
sqlj -version-long
```

`-version` オプションを使用すると、コマンドラインでファイル名などのオプションを指定していても、入力ファイルが変換されません。SQLJ では、トランスレータの実行かバージョン情報の表示のどちらか一方を指定します。同時に両方は指定できません。コマンドラインに入力したプロパティ・ファイルなどはすべて無視されます。

コマンドライン専用フラグと同じように、`-version` には、`=true` 構文を使用できません。このフラグを有効にするには、フラグ名のみを入力します。

-version の設定 `-version` を設定すると、SQLJ のリリース番号 (Oracle SQLJ Release 9.2 など) が表示されます。

-version-long の設定 -version-long を設定すると、SQLJ についての情報と SQLJ ランタイム・ライブラリのリリースおよび作成のバージョン番号が表示されます。JDBC ドライバがある場合は、このリリース番号も表示され、Java 環境についても表示されます。たとえば、Oracle JDBC を使用している場合は、「Oracle JDBC driver version 9.2 (9.2.0.0)」のように表示されます。

このフラグによって、SQLJ のインストールと、現在使用している JDBC や JDK のバージョンを確認できます。

コマンドラインの構文 `sqlj version_flag_settings`

コマンドラインの例

```
sqlj -version
sqlj -version -version-long
sqlj -version-long
```

プロパティ・ファイルの構文 該当なし

プロパティ・ファイルの例 該当なし

デフォルト値 無効

コマンドラインのエコー (-n)

-n をコマンドラインで指定すると、SQLJ トランスレータに渡されるコマンドライン全体が `sqlj` スクリプトによって構築され、ユーザーにエコーされます。`SQLJ_OPTIONS` の設定値もエコーされます。SQLJ トランスレータはこのコマンドラインを実行しません。SQLJ トランスレータを起動する JVM の名前とトランスレータのクラス名全体がエコーされます。プロパティ・ファイルの設定値はエコーされません。

次の内容が表示されます。

- 略称で入力したオプション名を完全に展開した名前（`loadjava` 対応の `-u` などの略称が展開され、完全な名前が表示されます。）
- コマンドの文字列全体が構築されてトランスレータに渡されるときオプションの並び順
- `SQLJ_OPTIONS` の設定値とコマンドラインの設定値の競合の可能性

-n オプションは、コマンドラインまたは `SQLJ_OPTIONS` 変数のどこにでも指定できます。

コマンドライン専用フラグと同じように、-n には、`=true` 構文を使用できません。このフラグを有効にするには、フラグ名のみを入力します。

次の場合を想定してください。

- SQLJ_OPTIONS で次のオプションを指定するとします。

```
-user=scott/tiger@jdbc:oracle:thin:@ -classpath=/myclasses/bin
```

- 次のコマンドラインを入力します。

```
% sqlj -n -e SJIS myapp.sqlj
```

次のエコーが表示されます。

```
java -classpath /myclasses/bin sqlj.tools.Sqlj -user=scott/tiger@jdbc:oracle:thin:@  
-C-classpath=/myclasses/bin -encoding=SJIS myapp.sqlj
```

(このコマンドラインは折り返されて表示されていますが、全体が1行で入力されています。)

注意：

- -n のかわりに、-vm=echo と入力することも可能です。
 - オプション設定の確認方法として、-help-long フラグも使用できます。このフラグを設定すると、すべてのオプションの現行の設定値が表示されます。コマンドラインで設定したオプションの他、プロパティ・ファイルおよび SQLJ_OPTIONS で設定したオプションの値も表示されます。8-23 ページの「SQLJ のオプション情報 (-help)」を参照してください。
-
-

コマンドラインの構文 -n

コマンドラインの例 -n

プロパティ・ファイルの構文 該当なし

プロパティ・ファイルの例 該当なし

デフォルト値 無効

出力ファイルとディレクトリのオプション

次のオプションで、SQLJ の入力および出力ソース・ファイルのエンコーディングを指定します。

- `-encoding`

次のオプションで、SQLJ の出力ファイルを格納するディレクトリを指定します。

- `-d`
- `-dir`

入力および出力ソース・ファイルの文字エンコーディング（-encoding）

`-encoding` オプションで、文字エンコーディングを指定します。この文字エンコーディングは、グローバル化・サポートの目的で `.sqlj` および `.java` 入力ファイルと生成される `.java` ファイルに適用されます。javac 対応として、コマンドラインでこのオプションを指定するときに、次のようにスペースまたは「=」を使用できます。

```
-encoding=SJIS
```

```
-encoding SJIS
```

ただし、プロパティ・ファイルで `sqlj.encoding` を指定するときは、「=」を使用します。スペースは使用できません。

このオプションを指定すると、Java コンパイラにも渡されます（`-compiler-encoding-flag` がオンの場合）。Java コンパイラは、この値に基づき、処理する `.java` ファイルの文字エンコーディングを指定します。

留意点を次に挙げます。

- 後述の `-classpath` および `-d` オプションと同じように、「=」を使用して `-encoding` オプションを設定すると、JVM およびコンパイラにオプションが渡されるときに「=」が削除されます。JVM とコンパイラでは、オプション設定に「=」を使用できません。
- `loadjava` ユーティリティ対応として、コマンドラインで指定した `-e` は `-encoding` として認識されます。8-9 ページの「[loadjava 互換オプション](#)」を参照してください。
- Java プロパティ・ファイル（`sqlj.properties` や `connect.properties` など）には、`-encoding` オプションを使用できません。プロパティ・ファイルでは、常に `8859_1` エンコーディングを使用します。これは、特別な SQLJ の機能ということではなく、一般的な Java の機能です。ただし、Unicode のエスケープ・シーケンスなら、プロパティ・ファイルで使用できます。（ネイティブ・コードのファイル用のエスケープ・シーケンスを作成するには、`native2ascii` ユーティリティを使用します。9-29 ページの「[native2ascii によるソース・ファイルの文字エンコーディングの変換](#)」を参照してください。）

コマンドラインの構文 `-encoding=Java_character_encoding`

コマンドラインの例 `-encoding=SJIS`

プロパティ・ファイルの構文 `sqlj.encoding=Java_character_encoding`

プロパティ・ファイルの例 `sqlj.encoding=SJIS`

デフォルト値 JVM におけるシステム・プロパティ `file.encoding` の設定値

.ser および .class ファイルの出力ディレクトリ (-d)

`-d` オプションで、SQLJ トランスレータで生成されるプロファイルのルート出力ディレクトリを指定します。ISO 標準コード生成 (`-codegen=iso`) の場合にのみ使用します。この設定値は Java コンパイラにも渡され、コンパイラで生成される `.class` ファイルのルート出力ディレクトリを指定します。プロファイルを `.ser` ファイルとして生成する場合（デフォルト）でも、`.class` ファイルとして生成する場合（`-ser2class` オプションが有効なとき）でも、`-d` オプションを指定できます。

ディレクトリを指定すると、このディレクトリの下に該当パッケージに出力ファイルが生成されます。たとえば、ソース・ファイルがパッケージ `a.b.c` にあるときに、`/mydir` ディレクトリを指定すると、出力ファイルが `/mydir/a/b/c` ディレクトリに格納されます。

相対ディレクトリ・パスを指定すると、カレント・ディレクトリからの相対になります。

javac 互換として、コマンドラインでこのオプションを指定するときは、次のようにスペースまたは「=」を使用できます。（次の例では、どちらもルート・ディレクトリ `/root` の下に、プロファイル・ファイルが生成されます。）

```
-d=/root
-d /root
```

プロパティ・ファイルで `-d` を指定するときは、`sqlj.d=/root` のように「=」を使用します。スペースは使用できません。

カレント・ディレクトリが `/root/home/mydir` のときに、次のように `-d` オプションで相対ディレクトリ・パス `mysubdir/myothersubdir` を指定すると、`/root/home/mydir/mysubdir/myothersubdir` をルート・ディレクトリとして、プロファイル・ファイルが生成されます。

```
-d=mysubdir/myothersubdir
```

標準構文も使用できます。つまり、カレント・ディレクトリをピリオド1個で、1レベル上のディレクトリをピリオド2個で表現できます。(次の2番目の例では、1レベル上がってから、平行ディレクトリ `parallelidir` に下がります。)

`-d=.`

`-d=../parallelidir`

`-d` オプションに何も指定しないと、変換処理で生成された `.class` ファイルと `.ser` ファイルは、次の場所に格納されます。

- トランスレータで生成された `.java` ファイルに対応する `.class` ファイルは、生成された `.java` ファイルと同じディレクトリに格納されます。これは `-dir` オプションによる処理です。
- コマンドラインで指定した `.java` ファイルの格納ディレクトリに、その `.class` も一緒に格納されます。
- 元の `.sqlj` ソース・ファイルの格納ディレクトリに、`.ser` ファイルも一緒に格納されます。

注意：

- `-d` を次のように無指定にすることも可能です (プロパティ・ファイル中の設定値をオーバーライドする場合)。
`-d=`
 - この説明では、ファイル・セパレータとしてスラッシュ (/) を使用しています。ただし、このようなオプションを実際に指定するときは、JVM の `file.separator` システム・プロパティで指定したオペレーティング・システムのファイル・セパレータを使用する必要があります。
 - 前述の `-classpath` および `-encoding` オプションと同じように、「=」を使用して `-d` オプションを指定すると、オプション文字列が JVM およびコンパイラに渡されるときに「=」が削除されます。JVM とコンパイラでは、オプション設定に「=」を使用できません。
-

コマンドラインの構文 `-d=directory_path`

コマンドラインの例 `-d=/topleveldir/mydir`

プロパティ・ファイルの構文 `sqlj.d=directory_path`

プロパティ・ファイルの例 `sqlj.d=/topleveldir/mydir`

デフォルト値 なし（.class ファイルは .java ファイルの格納ディレクトリに、.ser ファイルは .sqlj の格納ディレクトリにそれぞれ出力されます。）

.java ファイルの出力ディレクトリ (-dir)

SQLJ トランスレータで生成する .java ファイルのルート・ディレクトリは、-dir オプションで指定します。

ディレクトリを指定すると、このディレクトリの下で該当パッケージに出力ファイルが生成されます。たとえば、ソース・ファイルがパッケージ a.b.c にあるときに、/mydir ディレクトリを指定すると、出力ファイルが /mydir/a/b/c ディレクトリに格納されます。

相対ディレクトリ・パスを指定すると、カレント・ディレクトリからの相対になります。

次の簡単な例では、/root をルート・ディレクトリとして .java ファイルが生成されます。

```
-dir=/root
```

カレント・ディレクトリが /root/home/mydir で、次のように -dir オプションに相対ディレクトリ・パス mysubdir/myothersubdir を設定している場合、

```
-dir=mysubdir/myothersubdir
```

/root/home/mydir/mysubdir/myothersubdir が生成された .java ファイルのルート・ディレクトリになります。

標準構文も使用できます。つまり、カレント・ディレクトリをピリオド1個で、1レベル上のディレクトリをピリオド2個で表現できます。（次の2番目の例では、1レベル上がってから、平行ディレクトリ paralleldir に下がります。）

```
-dir=.
```

```
-dir=../paralleldir
```

-dir オプションを指定しないと、元の .sqlj ソース・ファイルと同じディレクトリにファイルが生成されます。（カレント・ディレクトリには生成されません。）

.sqlj ソース・ディレクトリを出力ディレクトリとして指定する（プロパティ・ファイルなどで行った他の -dir 設定を無効にする）には、次のように -dir オプションを使用します。

```
-dir=
```

注意：

- `-d` オプションでなく、`-dir` オプションを指定した場合は、その際に `-dir` で指定したディレクトリに、生成済みの `.class` ファイルが一緒に格納されます。ただし、生成済みの `.ser` ファイルの格納先は、`.sqlj` ファイルと同じディレクトリとなります。
 - この説明では、ファイル・セパレータとしてスラッシュ (`/`) を使用しています。ただし、このようなオプションを指定するときは、JVM の `file.separator` システム・プロパティで指定した、オペレーティング・システム固有のファイル・セパレータを使用する必要があります。
-

コマンドラインの構文 `-dir=directory_path`

コマンドラインの例 `-dir=/topleveldir/mydir`

プロパティ・ファイルの構文 `sqlj.dir=directory_path`

プロパティ・ファイルの例 `sqlj.dir=/topleveldir/mydir`

デフォルト値 なし (`.sqlj` ソース・ファイルのディレクトリが使用されます。)

接続オプション

オンライン・セマンティクス・チェック用のデータベース接続の際は、次のオプションを使用します。

- `-user`
- `-password`
- `-url`
- `-default-url-prefix`
- `-driver`

SQLJ トランスレータは、アプリケーション実行時と同じデータベースまたはスキーマに接続する必要はありません。アプリケーションのソース・コード中の接続情報は、SQLJ オプションの接続情報とは別々に指定できます。実際に、配置環境が開発時とテスト時には使用できないことがあります。

オンライン・セマンティクス・チェックとユーザー名 (-user)

データベース接続を行わない簡単なセマンティクス・チェックをオフライン・チェックと呼びます。接続を使用する、より完璧なセマンティクス・チェックをオンライン・チェックと呼びます。オンライン・チェックには、SQLJ の厳密に分類するためのパラダイムが大いに活用されています。つまり、型の非互換によるランタイム SQL 例外が変換段階で捕捉できるので、ユーザーがアプリケーションを実行する前に修正できます。

-user オプションを使用すると、オンライン・セマンティクス・チェックを有効にし、基本スキーマのユーザー名（スキーマ名）を指定できます。基本スキーマとは、チェック実行時にトランスレータで指定するサンプル・データベースのスキーマです。-user オプションは、パスワードと URL も指定できます。-password と -url オプションを別々に指定する必要はありません。

-user フラグ以外では、オンライン・セマンティクス・チェックを有効化および無効化できません。SQLJ では、-user オプションの有無によって、オンライン・セマンティクス・チェックを有効化および無効化します。

-user オプションについては、次の 2 つのカテゴリに分けて説明します。1) デフォルトの接続コンテキスト・クラスのみを使用した場合の -user の結果。2) デフォルト以外または複数の接続コンテキスト・クラスを使用した場合の -user の結果。デフォルト以外の接続コンテキスト・クラスについては、7-2 ページの「[接続コンテキスト](#)」を参照してください。

接続に関する一般的な検討事項（DefaultContext クラスのインスタンスを複数使用する場合や接続コンテキスト・クラスをさらに宣言する場合など）は、4-6 ページの「[接続の際の考慮事項](#)」を参照してください。

注意：

- SQLJ の `-parse` オプションの設定によっては、`-user` オプションの効果が無視され、オンライン・セマンティクス・チェックも無効になります。8-75 ページの「[オフライン・パーサー \(-parse\)](#)」を参照してください。
- `loadjava` ユーティリティ対応として、`-user` のかわりに `-u` をコマンドラインで指定できます。8-9 ページの「[loadjava 互換オプション](#)」を参照してください。
- ユーザー名には文字「/」または「@」を使用できません。
- コマンドラインでユーザー名を指定するときは、次のように「=」のかわりにスペースを使用できます。

```
-user scott/tiger
-user@CtxClass scott/tiger
-u scott/tiger
-u@CtxClass scott/tiger
```

- 文字「@」を使用したパスワードは、`-user` オプションで設定できません。`-user` と `-password` をそれぞれ別に設定する必要があります。
 - ログイン名が DBA グループのメンバーである場合、`SYSDBA` として、`SYS` スキーマに接続する特別な権限があります。この場合、ユーザー名 `SYS` または `INTERNAL` を指定できます。
 - ISO コード生成では、トランスレータの `-user` 設定がプロファイル・カスタマイザに転送されますが、カスタマイザの `user` 設定によって無視されることもあります。
-

デフォルトの接続コンテキスト・クラスのみを使用した場合の `-user` の結果 次に、`-user` オプションの最も基本的な使用方法を示します。

```
-user=scott
```

`DefaultContext` クラスのデフォルト接続などのインスタンスのみを使用すると、設定値がすべての SQLJ 実行文に適用されます。この例では、`scott` スキーマに対してオンライン・チェックが行われます。

パスワードまたは URL、あるいはこの両方をユーザー名と一緒に次の構文で指定することも可能です。(パスワードの前に「/」を付け、URLの前に「@」を付けます。)

```
-user=scott/tiger
```

または

```
-user=scott@jdbc:oracle:oci:@
```

または

```
-user=scott/tiger@jdbc:oracle:oci:@
```

URL を `-url` オプションで指定し、パスワードを対話形式または `-password` オプションで指定する方法もあります。

オンライン・セマンティクス・チェックを無効化するには、`-user` オプションを空の文字列に設定します。

```
-user=
```

デフォルト接続や `DefaultContext` クラスの他のインスタンスのみを使用すると、この設定値がすべての `SQLJ` 実行文に適用されます。

オンライン・セマンティクス・チェックの無効化は、プロパティ・ファイルで有効にしたオンライン・チェックをコマンドラインでオーバーライドする場合や、デフォルトのプロパティ・ファイルで有効にしたオンライン・チェックをユーザー指定のプロパティ・ファイル (`-props` オプションで指定) でオーバーライドする場合などに便利です。

専用のユーザー名 `URL.CONNECT` もあります。このユーザー名を使用して、URL で接続のユーザーとパスワードなどの詳細を指定できます。このような場合の URL の詳細は、8-39 ページの「[オンライン・セマンティクス・チェックに使用する接続 URL \(-url\)](#)」を参照してください。

デフォルト以外または複数の接続コンテキスト・クラスを使用した場合の `-user` の結果 アプリケーションでさらに接続コンテキスト・クラスを宣言して使用する場合は、これらのクラスのインスタンスを使用する `SQLJ` 実行文をテストするときに `-user` を使用できます。次のように、特定の接続コンテキスト・クラス (`CtxClass` など) のオンライン・チェックを行うユーザー名を指定します。

```
-user@CtxClass=scott
```

この指定では、クラス `CtxClass` の接続コンテキストのインスタンスが指定されているすべての `SQLJ` 実行文に対して、`scott` スキーマのオンライン・チェックが行われます。

デフォルトの接続コンテキスト・クラスの場合と同様、次のように特定の接続コンテキスト・クラスに対するパスワードまたは URL を `-user` で指定できます。

```
-user@CtxClass=scott/tiger@jdbc:oracle:oci:@
```

接続コンテキスト・クラス `CtxClass` は、ソース・コード中に宣言するか、あらかじめ `.class` ファイルにコンパイルしておく必要があります。(詳細は、7-2 ページの「[接続コンテキスト](#)」を参照してください。)

各接続コンテキスト・クラスに対してオンライン・チェックの有効化やユーザー名の設定を行う場合、そのクラスごとに `-user` オプションを個別に指定します。次に示したように、この設定は他のユーザーには反映されません。

```
-user@CtxClass1=user1 -user@CtxClass2=user2 -user@CtxClass3=user3
```

アプリケーションで複数の接続コンテキスト・クラスを使用すると、クラスを指定していない `-user` 設定が `DefaultContext` クラスおよび `-user` 値を指定していないすべてのクラスに適用されます。ただし、`-user` 値は、接続コンテキスト・クラスごとに指定します。通常、接続コンテキスト・クラスは、SQL オブジェクトごとに指定されています。

接続コンテキスト・クラス `CtxClass1`、`CtxClass2` および `CtxClass3` を宣言し、`-user` を次のように設定した場合を想定します。

```
-user@CtxClass2=scott/tiger -user=bill/lion
```

`CtxClass2` のインスタンスを使用しているアプリケーション内の文が、`scott` スキーマと照合されます。`DefaultContext`、`CtxClass1` または `CtxClass3` のインスタンスを使用している文が、`bill` スキーマと照合されます。

また、`-user` オプションでオンライン・チェックを有効にした後で、特定の接続コンテキストのオンライン・チェックを無効化するには、その接続コンテキストに対して空のユーザー名で `-user` オプションを設定します。たとえば、次のような設定があるとします。

```
-user@CtxClass2=
```

`CtxClass2` のインスタンスである接続オプションを指定した SQLJ 実行文に対するオンライン・セマンティクス・チェックが無効になります。

デフォルトの接続コンテキスト・クラスおよびユーザー名を指定しなかった接続コンテキスト・クラスのオンライン・セマンティクス・チェックを無効化するには、次のように指定します。

```
-user=
```

コマンドラインの構文 `-user<@conn_context_class>=username</password><@url>`

コマンドラインの例

```
-user=scott
-user=scott/tiger
-user=scott@jdbc:oracle:oci:@
-user=scott/tiger@jdbc:oracle:oci:@
-user=
-user=URL.CONNECT
-user@CtxClass=scott/tiger
-user@CtxClass=
```


プロパティ・ファイルの構文 `sqlj.user<@conn _context_class>=username</password><@url>`

プロパティ・ファイルの例

```
sqlj.user=scott
sqlj.user=scott/tiger
sqlj.user=scott@jdbc:oracle:oci:@
sqlj.user=scott/tiger@jdbc:oracle:oci:@
sqlj.user=
sqlj.user=URL.CONNECT
sqlj.user@CtxClass=scott/tiger
sqlj.user@CtxClass=
```

デフォルト値 なし（オンライン・セマンティクス・チェックはありません。）

注意： `-user` オプションと `-url` オプションとは、ユーザー、パスワードおよび URL を指定する書式が異なることに注意してください。
`-url` オプションでは、ユーザーとパスワードを URL の一部として JDBC ドライバ・タイプのすぐ後ろに指定しますが、`-user` オプションでは、まずユーザーとパスワードを指定し、その後ろに URL を指定します。8-39 ページの「[オンライン・セマンティクス・チェックに使用する接続 URL \(-url\)](#)」も参照してください。

オンライン・セマンティクス・チェック時のユーザー・パスワード (-password)

`-password` オプションでは、オンライン・セマンティクス・チェック用のデータベース接続で使用するユーザー・パスワードを指定します。`-password` 値を有効にするには、`-user` オプションも設定する必要があります。

`-user` オプションでパスワードを指定することも可能です。8-33 ページの「[オンライン・セマンティクス・チェックとユーザー名 \(-user\)](#)」を参照してください。パスワードを `-user` オプションで指定した場合は、接続コンテキスト・クラスに対して `-password` を使用しないでください。`-user` オプションの値が優先されます。

`-password` オプションの大半の機能は、`-user` オプションの機能に相当します。つまり、アプリケーションで `DefaultContext` のデフォルト接続などのインスタンスのみを使用する場合は、すべての SQLJ 文のチェックに使用するスキーマのパスワードを次のように指定します。

```
-password=tiger
```

CtxClass1 などの接続コンテキスト・クラスをさらに宣言して使用する場合は、これらの接続コンテキスト・クラスを使用する文をテストするために追加するスキーマを `-user` オプションで指定できます。同様に、これらのスキーマのパスワードを次のように `-password` オプションで指定できます。

```
-password@CtxClass1=tiger
```

接続コンテキスト・クラスのパスワードを `-password` または `-user` で設定しないと、デフォルトの接続コンテキスト・クラスに対して設定されているパスワードが使用されます。デフォルトの接続コンテキスト・クラスに対してパスワードを指定しないと、パスワードを問い合わせる SQLJ の対話型プロンプトが表示されます。ユーザー定義の接続コンテキスト・クラスに対してパスワードを設定しないと、パスワードを問い合わせる SQLJ の対話型プロンプトが表示されます。ただし、ユーザー名に `URL.CONNECT` を使用する場合は例外です。

8-39 ページの「[オンライン・セマンティクス・チェックに使用する接続 URL \(-url\)](#)」を参照してください。この場合、ユーザー名とパスワードは `-url` で指定した文字列から特定されます。`-password` オプションの設定値は無視されます。

`-password` オプションの設定値を無効にし、対話型プロンプトで入力を要求するには、プロパティ・ファイルなどで空のパスワードを設定します。次のように、`DefaultContext` クラスまたは特定の接続コンテキスト・クラスに対して設定できます。

```
-password=
```

または

```
-password@CtxClass1=
```

実際に空のパスワードでログインするには、次のように `EMPTY.PASSWORD` を指定します。

```
-password=EMPTY.PASSWORD
```

または

```
-password@CtxClass2=EMPTY.PASSWORD
```

ただし、Oracle9i データベースでは空のパスワードを使用できません。

注意：

- コマンドラインで指定した `-p` は、`-password` と同じものとして認識されます。
- コマンドラインでパスワードを設定するときは、次のように「`=`」のかわりにスペースを使用できます。

```
-password tiger
-password@CtxClass tiger
-p tiger
-p@CtxClass tiger
```
- ISO コード生成では、トランスレータの `-password` 設定がプロファイル・カスタマイザに転送されますが、カスタマイザの `password` 設定によって無視されることもあります。

コマンドラインの構文 `-password<@conn_context_class>=user_password`

コマンドラインの例

```
-password=tiger
-password=
-password=EMPTY.PASSWORD
-password@CtxClass=tiger
```

プロパティ・ファイルの構文 `sqlj.password<@conn_context_class>=user_password`

プロパティ・ファイルの例

```
sqlj.password=tiger
sqlj.password=
sqlj.password=EMPTY.PASSWORD
sqlj.password@CtxClass=tiger
```

デフォルト値 なし（DefaultContext のパスワードが使用されるか、ユーザーの入力を要求するプロンプトが表示されます。）

オンライン・セマンティクス・チェックに使用する接続 URL（-url）

`-url` オプションでは、オンライン・セマンティクス・チェックのためのデータベース接続の確立に使用する URL を指定します。必要に応じて、URL にホスト名、ポート番号および Oracle SID を指定できます。

また、`-user` オプション設定の一部として URL を指定することも可能です。8-33 ページの「[オンライン・セマンティクス・チェックとユーザー名（-user）](#)」を参照してください。接続コンテキスト・クラスの URL を `-user` オプションで設定した場合は、その設定が優先され

ます。この場合は、接続コンテキスト・クラスに `-url` オプションを使用しないでください。

`-url` オプションの大半の機能は、`-user` オプションの機能に相当します。つまり、アプリケーションで `DefaultContext` のデフォルト接続などのインスタンスのみを使用する場合は、すべての SQLJ 文のチェックのための接続に使用する URL を、次のように指定します。

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

URL にホスト名、ポート番号および SID を含める場合は、次のように指定します。

```
-url=jdbc:oracle:thin:@hostname:1521:orcl
```

URL 設定の最初に `jdbc:` が付かない場合、設定形式は `host:port:sid` であると見なされます。この場合、デフォルトでは、設定の最初に次の接頭辞が自動的に付けられます。

```
jdbc:oracle:thin:@
```

`-url` 設定を `localhost:1521:orcl` にすると、URL が自動的に次のようになります。

```
jdbc:oracle:thin:@localhost:1521:orcl
```

このデフォルトの動作を削除または変更するには、`-default-url-prefix` オプションを使用します。8-42 ページの「[デフォルトの URL 接頭辞 \(-default-url-prefix\)](#)」を参照してください。

ユーザーおよびパスワードは、`-user` および `-password` 設定ではなく、`-url` 設定でも指定できます。この場合は、次に示すように、`-user` を `URL.CONNECT` に設定します。

```
-url=jdbc:oracle:oci:scott/tiger@ -user=URL.CONNECT
```

たとえば `CtxClass1` など、追加の接続コンテキスト・クラスを宣言して使用する場合は、その接続コンテキスト・クラスを使用する文のテストのための、基本スキーマを追加指定します。次に示す例のように、これらのスキーマの URL は `-url` オプションで指定できます。

```
-url@CtxClass1=jdbc:oracle:oci:@
```

`-url` 設定と `-user` 設定のいずれでも URL が設定されていない接続コンテキスト・クラスの場合は、デフォルトの接続コンテキスト・クラスに設定されている URL が使用されます。ただし、デフォルトのコンテキスト・クラスに URL が設定されていることを前提とします。

注意：

- URL が設定されている接続コンテキスト・クラスには、ユーザー名も設定されていることが必要です。設定されていないと、オンライン・チェックが行われません。
- 次に示す例のように、コマンドラインの URL 設定では、「=」のかわりにスペースを使用できます。

```
-url jdbc:oracle:oci:@
-url@CtxClass jdbc:oracle:oci:@
```

- ISO コード生成では、トランスレータの -url 設定がプロファイル・カスタマイザに転送されますが、カスタマイザの url 設定によって無視されることもあります。
-

コマンドラインの構文 -url<@conn_context_class>=URL

コマンドラインの例

```
-url=jdbc:oracle:oci:@
-url=jdbc:oracle:thin:@hostname:1521:orcl
-url=jdbc:oracle:oci:scott/tiger@
-url=hostname:1521:orcl
-url@CtxClass=jdbc:oracle:oci:@
```

プロパティ・ファイルの構文 sqlj.url<@conn_context_class>=URL

プロパティ・ファイルの例

```
sqlj.url=jdbc:oracle:oci:@
sqlj.url=jdbc:oracle:thin:@hostname:1521:orcl
sqlj.url=jdbc:oracle:oci:scott/tiger@
sqlj.url=hostname:1521:orcl
sqlj.url@CtxClass=jdbc:oracle:oci:@
```

デフォルト値 jdbc:oracle:oci:@

注意： -user オプションと -url オプションとは、ユーザー、パスワードおよび URL を指定する書式が異なることに注意してください。-url オプションでは、ユーザー名とパスワードを URL の一部として JDBC ドライバ・タイプの直後に指定します。-user オプションでは、ユーザー名とパスワードを指定し、その後ろに URL を指定します。8-33 ページの「[オンライン・セマンティクス・チェックとユーザー名 \(-user\)](#)」も参照してください。

デフォルトの URL 接頭辞 (-default-url-prefix)

デフォルトの接頭辞を変更または削除するには、`-default-url-prefix` オプションを使用します。

URL 設定の最初に `jdbc:` が付いていない場合、デフォルトでは、次の接頭辞が自動的に付けられます。

```
jdbc:oracle:thin:@
```

このため、`-user` と `-url` のいずれのオプションで URL を設定する場合でも、データベースのホスト、ポートおよび SID の指定のみです。次のように URL を指定するとします。

```
-url=myhost:1521:orcl
```

または

```
-user=scott/tiger@myhost:1521:orcl
```

デフォルトでは、URL が次のように解釈されます。

```
jdbc:oracle:thin:@myhost:1521:orcl
```

次の例のように、URL の指定が `jdbc:` で開始される場合、デフォルトの接頭辞は使用されません。

```
-url=jdbc:oracle:oci:@orcl
```

ただし、たとえば、デフォルトの URL 設定で Thin ドライバのかわりに OCI ドライバを使用するには、デフォルトの接頭辞を次のように設定します。

```
-default-url-prefix=jdbc:oracle:oci:@
```

このように接頭辞を設定した場合、`-url=orcl` と指定すると、`-url=jdbc:oracle:oci:@orcl` になります。

接頭辞を付けないようにするには、次に示すように、`-default-url-prefix` オプションを空の文字列に設定します。

```
-default-url-prefix=
```

コマンドラインの構文 `-default-url-prefix=url_prefix`

コマンドラインの例

```
-default-url-prefix=jdbc:oracle:oci:@  
-default-url-prefix=
```

プロパティ・ファイルの構文 `sqlj.default-url-prefix=url_prefix`

プロパティ・ファイルの例

```
sqlj.default-url-prefix=jdbc:oracle:oci:@  
sqlj.default-url-prefix=
```

デフォルト値 jdbc:oracle:thin:@

オンライン・セマンティクス・チェックのために登録する JDBC ドライバ (-driver)

-driver オプションでは、オンライン・セマンティクス・チェックに使用する JDBC 接続 URL の解釈のために登録する、JDBC ドライバ・クラスを指定します。ドライバ・クラスまたはカンマで区切られたクラスのリストを指定します。

デフォルトの OracleDriver では、Oracle9i での Oracle OCI、Thin およびサーバー側 JDBC ドライバの使用がサポートされています。

コマンドラインの構文 -driver=driver1<,driver2,driver3,...>

コマンドラインの例

```
-driver=oracle.jdbc.OracleDriver  
-driver=oracle.jdbc.OracleDriver,sun.jdbc.odbc.JdbcOdbcDriver
```

プロパティ・ファイルの構文 sqlj.driver=driver1<,driver2,driver3,...>

プロパティ・ファイルの例

```
sqlj.driver=oracle.jdbc.OracleDriver  
sqlj.driver=oracle.jdbc.OracleDriver,sun.jdbc.odbc.JdbcOdbcDriver
```

デフォルト値 oracle.jdbc.OracleDriver

レポートと行マッピングのオプション

次に示す各オプションでは、SQLJ がモニターする条件、リアルタイム・エラーとステータス・メッセージを生成するかどうか、およびトランスレータのエラー・メッセージに原因と処置の情報を含めるかどうかを指定します。

- -warn
- -status
- -explain

次のオプションを使用すると、生成済みの `Java.class` ファイルから元の `.sqlj` ソース・ファイルへの行マッピングができます。この行マッピングを行うと、ランタイム・エラーについて、元のソース・コードの該当箇所を確認できます。`-jdblinemap` を Sun 社の `jdb` デバッガで使用方法と、`-linemap` を使用方法があります。

- `-linemap`
- `-jdblinemap`

トランスレータからの警告 (-warn)

SQLJ トランスレータでは、変換時の状況に応じて、様々な警告や情報メッセージが表示されます。`-warn` オプションは、どの警告やメッセージを表示するかを指定する一連のフラグで構成されます（つまり、このオプションでは、モニターする条件と無視する条件を指定します）。

このオプションのすべてのフラグは、単一のカンマで区切られた文字列に結合されることが必要です。

表 8-3 に、テストする状況、各状況に対するフラグ値 `true` および `false` の意味、フラグ値 `true` の意味、およびデフォルトの値を示します。

表 8-3 SQLJ の警告のテストとフラグ

テストおよびフラグ機能	TRUE/FALSE 値
継承階層で宣言されたオブジェクト型のサブタイプの要件をテストする — <code>cast</code> を有効にすると、SQL 継承階層の SQL オブジェクト型を使用するには、宣言された型のサブタイプを実行時に渡す必要がある場合に、警告を受け取ります。	<code>cast</code> (デフォルト) <code>nocast</code>
データ精度のテスト — <code>-precision</code> を有効にした場合、データベースの列から Java ホスト変数に値を移動した際にデータの精度が失われる可能性があるとして、警告を受け取ります。	<code>precision</code> (デフォルト) <code>noprecision</code>
NULL 化可能なデータの変換ロスのテスト — <code>-nulls</code> を有効にした場合、データベースの列から Java ホスト変数に NULL 化可能な列または NULL 化可能な Java 型を移動した際に、変換でデータの損失が発生する可能性があるとして、警告を受け取ります。	<code>nulls</code> (デフォルト) <code>nonnulls</code>
移植可能性のテスト — <code>-portable</code> を有効にすると、SQLJ 句の移植可能性がチェックされ、移植できない句があると警告を受け取ります。(ベンダー固有の型や機能など、SQLJ の拡張機能如果使用されていると、移植できません。)	<code>portable</code> <code>noportable</code> (デフォルト)

表 8-3 SQLJ の警告のテストとフラグ (続き)

テストおよびフラグ機能	TRUE/FALSE 値
名前指定イテレータの厳密マッチングのテスト - <code>-strict</code> を有効にすると、データベースから選択した列の数とデータが取り込まれている名前指定イテレータ内の列の数が一致することが、SQLJ での必須条件になります。データベース・カーソル内の列に対応する列がイテレータにないと、警告が生成されます。 <code>nostrict</code> を設定した場合、データベース・カーソル内の列数を、イテレータ内の列数より多くすることが可能になります (ただし、イテレータ内の列数未満にはできません)。一致しない列は無視されます。	<code>strict</code> (デフォルト) <code>nostrict</code>
変換時の情報メッセージ - <code>-verbose</code> を有効にすると、オンライン・チェック用に確立されたデータベース接続など、変換処理についての追加の情報メッセージが生成されます。	<code>verbose</code> <code>noverbose</code> (デフォルト)
警告をグローバルに有効化または無効化する - <code>all</code> または <code>none</code> を指定して、すべての警告を有効または無効にします。	<code>all</code> <code>none</code>

`verbose/noverbose` フラグは、他のフラグとは異なった働きをします。このフラグによって、特定のテストが有効になることはありませんが、セマンティクス・チェックについての概要メッセージの出力が有効になります。

注意: `-warn=verbose` フラグと `-status` フラグを混同しないでください。`-status` フラグを有効にすると、変換、セマンティクス・チェック、コンパイルおよびプロファイルのカスタマイズ (使用している場合) など、SQLJ 変換にかかわるすべての処理についての情報メッセージが、リアルタイムで提供されます。これに対して `-warn=verbose` フラグでは、変換フェーズについてのみ、追加情報が提供されます。

グローバルな `all/none` フラグは、デフォルトの設定に優先します。このフラグを使用すると、すべてのフラグを有効化または無効化することが可能になります。または、選択されているフラグを有効にする前にすべてのフラグを無効化する、あるいはその反対の動作をするように、このフラグで初期化することも可能です。

`all` を設定すると、次のように指定した場合と同じ結果になります。

`cast,precision,nulls,portable,strict,verbose`

`none` を設定すると、次のように指定した場合と同じ結果になります。

`nocast,noprecision,nonnulls,noportable,nostrict,noverbose`

`all/none` のデフォルトはありません。個々のフラグのデフォルトのみがあります。

いくつかの例を示します。

- 次のシーケンスでは、`nulls` フラグのみが有効になります。
`-warn=none,nulls`
- 次のシーケンスは `verbose` 設定が無効にされるため、同じ結果になります。
`-warn=verbose,none,nulls`
- 次のシーケンスでは、移植可能性フラグ以外のすべてが有効になります。
`-warn=all,noportable`
- 次のシーケンスは `nonnulls` 設定が無効にされるため、同じ結果になります。
`-warn=nonnulls,all,noportable`

`all/none` フラグ以外の `-warn` 設定フラグの位置は重要ではありません。ただし、競合する設定の場合は別です。 `-warn=portable,noportable` のように競合が発生する場合は、最後に配布された（最右端の）設定が使用されます。

プロパティ・ファイルとコマンドラインで別々に `-warn` オプションを設定しても、両方の設定が受け付けられることはありません。最後の設定のみが処理されます。次の例では、`-warn=portable` 設定は無視されます。このフラグ、および `nulls/nonnulls` 以外のすべてのフラグは、それぞれのデフォルト設定に従って処理されます。

`-warn=portable -warn=nonnulls`

注意： キャスト、精度、NULL 化可能性および厳密性の各テストは、オンライン・セマンティクス・チェックの一部であり、データベース接続が必要です。

コマンドラインの構文 `-warn=comma-delimited_list_of_flags`

コマンドラインの例 `-warn=none,nulls,precision`

プロパティ・ファイルの構文 `sqlj.warn=comma-delimited_list_of_flags`

プロパティ・ファイルの例 `sqlj.warn=none,nulls,precision`

デフォルト値 `cast,precision,nulls,noportable,strict,noverbose`

リアルタイムのステータス・メッセージ (-status)

-status フラグを有効にすると、変換、セマンティクス・チェック、コンパイルおよびカスタマイズなど、すべての SQLJ 処理について追加のステータス・メッセージが出力されます。SQLJ 操作の各ステージで各ファイルが処理されると、メッセージが出力されます。

注意：

- -warn=verbose フラグと -status フラグを混同しないでください。
-status フラグでは、SQLJ 変換にかかわるすべての処理についての情報メッセージが、リアルタイムで提供されます。これに対して
-warn=verbose フラグでは、変換フェーズについてのみ、追加情報が提供されます。
 - loadjava ユーティリティとの互換性については、-v がコマンドラインで指定されると、-status に相当すると認識されます。8-9 ページの「[loadjava 互換オプション](#)」を参照してください。
-

コマンドラインの構文 -status<=true|false>

コマンドラインの例 -status

プロパティ・ファイルの構文 sqlj.status<=true|false>

プロパティ・ファイルの例 sqlj.status

デフォルト値 false

トランスレータ・エラーの原因と処置 (-explain)

-explain フラグを有効にすると、SQLJ トランスレータでエラーが最初に発生したときのみ、出力されるエラー・メッセージに原因および処置の情報が（ある場合は）出力されます。

この情報については、B-2 ページの「[SQLJ 変換時のメッセージ](#)」以降に解説します。

コマンドラインの構文 -explain<=true|false>

コマンドラインの例 -explain

プロパティ・ファイルの構文 sqlj.explain<=true|false>

プロパティ・ファイルの例 sqlj.explain

デフォルト値 false

SQLJ ソース・ファイルとの行マッピング (-linemap)

-linemap フラグを有効にすると、SQLJ ソース・コード・ファイルの行番号が、対応する .class ファイルの場所にマッピングされます。（このファイルは、SQLJ トランスレータで生成された .java ファイルのコンパイルで生成される .class ファイルになります。）このマッピングにより、Java ランタイム・エラーが発生した場合に、SQLJ ソース・コードの行番号と同じ行番号が JVM によって報告され、デバッグがはるかに容易になります。

通常、.class ファイル内の命令は、対応する .java ファイル内のソース・コードの行にマッピングされます。ただし、生成された .java ファイル内の行番号を元の .sqlj ファイル内の行番号にマッピングする必要があるため、SQLJ 開発者にはあまり役に立ちません。

SQLJ トランスレータは、-linemap オプションを実装するように、.class ファイルを修正します。その結果、生成された .java ファイルの行番号とファイル名が、元の .sqlj ファイルの対応する行番号とファイル名で置き換えられます。この処理を、クラス・ファイルのインストルメントと呼びます。

SQLJ でこの処理を行う際には、次の設定が考慮されます。

- .class ファイルのルート・ディレクトリを決定する -d オプション設定
- 生成された .java ファイルのルート・ディレクトリを決定する -dir オプション設定

注意：

- .sqlj ファイルの処理でエラーのためにコンパイルが行われなかった場合は、マッピング用の .class ファイルが生成されないため、行マッピングも行われません。
 - SQLJ から Java コンパイラが起動された場合、元の .sqlj ソース・ファイルの行番号を使用して、コンパイル・エラーが出力されます。生成された .java ファイルの行番号は使用されません。このマッピングには、オプションの設定は必要ありません。
 - .sqlj ファイル内の無名クラスは、インストルメントされません。
 - Sun 社の jdbc デバッガを使用する際は、-linemap オプションのかわりに、-jdbblinemap オプション（次に解説）を使用します。
-

コマンドラインの構文 -linemap<=true|false>

コマンドラインの例 -linemap

プロパティ・ファイルの構文 sqlj.linemap<=true|false>

プロパティ・ファイルの例 sqlj.linemap

デフォルト値 false

jdb デバッガでの SQLJ ソース・ファイルへの行マッピング (-jdblinemap)

このオプションと機能が同様のものとしては `-linemap` オプション（前述）がありますが、Sun 社の JDK で提供される jdb デバッガを使用する場合は、この `jdblinemap` の方が使用する必要があります。この理由は、jdb からアクセスできるソース・ファイルが `.java` ファイルという拡張子付きのものに限られているためです。

`-jdblinemap` を設定すると、SQLJ では次の処理が行われます。

- トランスレータで生成された `.java` ファイルの内容を、元の `.sqlj` ファイルの内容で上書きします。
- 生成された `.class` ファイルの中にあるファイル名のうち、`.sqlj` ファイル名でなく `.java` ファイル名を保持します。

このため SQLJ ソース・コードから jdb へのアクセスが可能になっています。

コマンドラインの構文 `-jdblinemap<=true|false>`

コマンドラインの例 `-jdblinemap`

プロパティ・ファイルの構文 `sqlj.jdblinemap<=true|false>`

プロパティ・ファイルの例 `sqlj.jdblinemap`

デフォルト値 `false`

コード生成、最適化および CHAR 比較のオプション

Oracle9i リリース 2 (9.2) のデフォルトでは、Oracle SQLJ は、ISO 標準 SQLJ コード生成のかわりに Oracle 固有のコード生成を使用します。これによって、Oracle JDBC コードが直接生成されます。Oracle 固有コード生成の場合、プロファイルは生成されず、コード実行時に SQLJ ランタイムはほとんどバイパスされます。

プロファイルのカスタマイズは Oracle 固有コード生成には利用できないので、以前に Oracle カスタマイザでのみ利用可能であった一般的に便利ないくつかの最適化オプションが、現在は SQLJ トランスレータで直接利用できます。

WHERE 句の CHAR 比較で、列の空白埋めを考慮するオプションもあります。このオプションは、トランスレータ・オプション（Oracle 固有コード生成の場合）としても、Oracle カスタマイザ・オプション（ISO 標準コード生成の場合）としても使用できます。

ここでは、コード生成、最適化および CHAR 比較のオプションについて説明します。

- `-codegen`
- `-optcols`

- `-optparams`
- `-optparamdefaults`
- `-fixedchar`

コードの生成 (-codegen)

Oracle SQLJ では、Oracle 固有の JDBC コードを直接生成できます。または、SQLJ ランタイムをコールし、次に JDBC をコールする ISO 標準コードを生成することもできます。Oracle 固有コード生成の場合、プロファイルは生成されず、プログラム実行時に SQLJ ランタイムはほとんどバイパスされます。

ISO 規格に従ってコードを生成する場合は（以前のリリースではこれがデフォルト設定でした）、次のように SQLJ トランスレータの `-codegen` オプションを使用します。

```
-codegen=iso
```

Oracle9i リリース 2 (9.2) の場合、デフォルトは Oracle 固有の SQLJ コード生成です。次のように Oracle 固有コード生成を明示的に指定することもできます。

```
-codegen=oracle
```

Oracle 固有コード生成に関するメリット、デメリット、制限事項および特別な考慮事項は、4-36 ページの「[Oracle 固有コード生成（プロファイルなし）](#)」を参照してください。

注意：

- `codegen=iso` の場合、`-user`、`-password`、`-url`、`-optparams`、`-optparamdefaults` および `-fixedchar` のトランスレータ設定は、プロファイル・カスタマイザにも転送されます。ただし、カスタマイザ・オプションを直接設定することによって、これらのカスタマイズ設定（特に `-user`、`-password` および `-url`）を無視できます。
 - JDK 1.1.x および JDK 1.2.x（以上）の両方がある環境でアプリケーションをコンパイルして、`-codegen=oracle` の設定で変換し、コンパイルに JDK 1.2.x（以上）を使用した場合、JDK 1.1.x 環境ではアプリケーションが実行できない可能性があります。JDK 1.1.x 環境で実行するアプリケーションは JDK 1.1.x を使用してコンパイルする必要があります。
-

コマンドラインの構文 `-codegen=iso|oracle`

コマンドラインの例 `-codegen=iso`

プロパティ・ファイルの構文 `sqlj.codegen=iso|oracle`

プロパティ・ファイルの例 `sqlj.codegen=iso`

デフォルト値 `oracle`

列の定義 (-optcols)

SQLJ トランスレータ `-optcols` フラグを使用して、イテレータまたは結果セット列の型およびサイズを定義するようにトランスレータに指定します。このようにすると、アプリケーション実行時に、列を Oracle JDBC ドライバに登録できます。その結果、ドライバの実装にもよりますが、Oracle9i へのラウンドトリップが減ります。この方法は、Thin ドライバと位置指定イテレータには特に効果的です。

列定義の概要は、10-22 ページの「[列の定義](#)」を参照してください。

注意： このトランスレータ・オプションは、Oracle カスタマイザの `optcols` オプションと同じで、プロファイルがないデフォルトの Oracle 固有コード生成 (`-codegen=oracle`) のために作成されたものです。しかし、ISO 標準コード生成 (`-codegen=iso`) にも適用されます。この場合、トランスレータ・オプションを設定すると、同様にカスタマイザ・オプションも自動的に設定されます。A-26 ページの「[Oracle カスタマイザの列定義オプション \(optcols\)](#)」を参照してください。

このフラグの有効化 / 無効化は、SQLJ コマンドラインまたはプロパティ・ファイルで設定します。

このフラグをコマンドラインで有効にするには、次のようにします。

`-optcols`

または

`-optcols=true`

このフラグは、デフォルトでは無効になっていますが、明示的に無効化することも可能です。このフラグをコマンドラインで無効化するには、次のようにします。

`-optcols=false`

列定義には、問合せが実行されている表の列を検証するためにデータベース接続が必要です。そのために、SQLJ トランスレータの `-user`、`-password` および `-url` オプションも設定する必要があります。次にその例を示します。

`sqlj -user=scott/tiger@jdbc:oracle:oci:@ -optcols MyApp.sqlj`

注意：

- 選択したすべての列に定義が作成されるため、選択したすべての列を実際に使用しない可能性がある場合は、SQL 操作で「SELECT *」構文を使用するのではなく、使用する列を明示的に選択することをお勧めします。選択した列の数が必要な数を上回ると、実行時エラーの可能性が高くなります。このことは、カスタマイズと実行時の間に表を変更した場合、特に列定義をカスタマイズした場合に当てはまります。変換時には **SQLJ -warn=strict** フラグを設定することをお勧めします。問合せの際にさらに余計な（不要な）列が選択された場合に、警告が出されるためです。
 - オブジェクトまたはコレクションが 1 つ以上含まれるイテレータや結果セットに対しては、列定義はできません。
 - データベース接続用ユーザー名、パスワードおよび URL を指定せずに、**-optcols** オプションを有効にすると、エラーが発生します。
 - 実行時にはデータベースの接続先と同じスキーマまたはデータベースに、トランスレータから接続する必要はありません。ただし、実行時のエラーを回避するために、関連のある列と、型およびサイズが同じ列は同じ順序で並べる必要があります。
-
-

コマンドラインの構文 `-optcols<=true|false>`

コマンドラインの例 `-optcols`

プロパティ・ファイルの構文 `sqlj.optcols<=true|false>`

プロパティ・ファイルの例 `sqlj.optcols`

デフォルト値 `false`

パラメータの定義 (-optparams)

パラメータ・サイズの定義を有効化するには、SQLJ トランスレータの **-optparams** フラグを使用します。SQLJ ではこのフラグを有効にすると、入出力パラメータ（ホスト変数）が登録され、JDBC リソース割当て量が指定したサイズに基づいて最適化されます。このサイズの優先順位は次のとおりです。

1. ソース・コードのヒント中に指定したサイズ（ある場合）
2. **-optparamdefaults** オプション設定のデータ型に対して指定したデフォルトのサイズ

指定されたホスト変数用のソース・コードのヒントまたはデフォルトのデータ型サイズが設定されていない場合、リソース割当て量は JDBC 次第になります。

パラメータ・サイズ定義の概要とソース・コード・ヒントの詳細は、10-23 ページの「[パラメータ・サイズの定義](#)」を参照してください。

注意： このトランスレータ・オプションは、Oracle カスタマイザの `optparams` オプションと同じです。このトランスレータ・オプションは、プロファイルがないデフォルトの Oracle 固有コード生成 (`-codegen=oracle`) のために作成されたものです。しかし、ISO 標準コード生成 (`-codegen=iso`) にも適用されます。この場合、トランスレータ・オプションを設定すると、同様にカスタマイザ・オプションも自動的に設定されます。A-28 ページの「[Oracle カスタマイザのパラメータ定義オプション \(optparams\)](#)」を参照してください。

`-optparams` フラグの有効化 / 無効化は、コマンドラインまたは SQLJ プロパティ・ファイルで設定します。

このフラグをコマンドラインで有効にするには、次のようにします。

```
-optparams
```

または

```
-optparams=true
```

このフラグは、デフォルトでは無効になっていますが、明示的に無効化することも可能です。このフラグをコマンドラインで無効化するには、次のようにします。

```
-optparams=false
```

注意： `-optcols` オプションとは異なり、`-optparams` オプションではサイズを自分で指定できるので、データベース接続は不要です。

コマンドライン (`-optparamdefaults` オプションの設定はここでは省略し、次の項で説明します) の例に次に示します。

```
sqlj -optparams -optparamdefaults=defaults_string MyApp.sqlj
```

コマンドラインの構文 `-optparams<=true|false>`

コマンドラインの例 `-optparams`

プロパティ・ファイルの構文 `sqlj.optparams<=true|false>`

プロパティ・ファイルの例 sqlj.optparams

デフォルト値 false

パラメータのデフォルト・サイズ (-optparamdefaults)

-optparams オプションを有効にしてパラメータ・サイズを設定した場合は、必要に応じて -optparamdefaults オプションを使用してデータ型のデフォルト・サイズを設定します。-optparams を有効にしないと、-optparamdefaults が設定されていても無視されるので注意してください。

ホスト変数のソース・コード・ヒントにパラメータ・サイズが指定してある場合、このオプションで設定されたデフォルトのサイズよりも優先されます。特定のホスト変数に対してソース・コード・ヒントも、対応するデータ型のデフォルト・サイズも指定されていない場合は、その変数のリソース割当て量は、JDBC ドライバによって判断されます。また、-optparams が有効でない場合も同様に JDBC ドライバによって判断されます。

-optparams が有効化してあるときは常に -optparamdefaults オプションを使用するのが一般的ですが、必須ではありません。-optparams が有効化されていても、デフォルト・サイズが設定されていない場合は、ソース・コード・ヒント（ある場合）または JDBC ドライバに基づいてリソースが割り当てられます。

パラメータ・サイズ定義の概要とソース・コード・ヒントの詳細は、10-23 ページの「[パラメータ・サイズの定義](#)」を参照してください。

注意： このトランスレータ・オプションは、Oracle カスタマイザの optparamdefaults オプションと同じです。このトランスレータ・オプションは、プロファイルがないデフォルトの Oracle 固有コード生成 (-codegen=oracle) のために作成されたものです。しかし、ISO 標準コード生成 (-codegen=iso) にも適用されます。この場合、トランスレータ・オプションを設定すると、同様にカスタマイザ・オプションも自動的に設定されます。A-29 ページの「[Oracle カスタマイザ用パラメータのデフォルト・サイズ・オプション \(optparamdefaults\)](#)」を参照してください。

-optparamdefaults フラグは、コマンドラインでも SQLJ プロパティ・ファイルでも設定できます。

このフラグをコマンドラインで設定するには、次のようにします。

```
-optparamdefaults=datatype1(size1),datatype2(size2),...
```

サイズはすべてバイト単位になっています。空白文字は、含まないようにしてください。NULL を設定するには、空のカッコを使用します。

たとえば、次のように指定すると、VARCHAR2 型で 30 バイト、RAW 型で 1000 バイトが設定され、また CHAR 型での NULL のサイズ設定が指定されます。CHAR データ型のどのホスト変数に対してもソース・コード・ヒントがない場合は、JDBC ドライバによってリソースが割り当てられます。

```
-optparamdefaults=VARCHAR2(30),RAW(1000),CHAR()
```

-optparamdefaults オプションでは、次のデータ型の名前が認識されます。

- CHAR
- VARCHAR、VARCHAR2 (同義)
- LONG、LONGVARCHAR (同義)
- BINARY、RAW (同義)
- VARBINARY
- LONGVARBINARY、LONGRAW (同義)

-optparamdefaults オプションでは、次のグループ名とワイルド・カードも認識されます。

- CHAR_TYPE には、CHAR、VARCHAR/VARCHAR2 および LONG/LONGVARCHAR があります。
- RAW_TYPE には、BINARY/RAW、VARBINARY および LONGVARBINARY/LONGRAW があります。
- パーセント記号 (%) 自体を指定した場合は、データ型として認識されるものがすべて表されます。一方、これをある部分的な名前の語尾に付加した場合は、データ型のサブセットが表されます。たとえば、VAR% を指定すると、「VAR」で始まるあらゆるデータ型が表されます。

-optparamdefaults の設定値は、左から右へと処理されます。グループ名またはワイルド・カードを使用すると、特定のデータ型に対するグループ設定を上書きできます。

次の例のようにすると、一般的なデフォルト・サイズとして 50 バイト分が設定され、その設定値が 500 バイト分の RAW 型の設定値に置き換わり、さらに、その RAW 型のグループ設定値が VARBINARY 型の NULL 設定に置き換わります (ソース・コード・ヒントのないホスト変数は、JDBC で処理されます)。

```
-optparamdefaults=%(50),RAW_TYPE(500),VARBINARY()
```

次にコマンドラインの例を示します。この例では、-optparams も設定します。

```
sqlj -optparams -optparamdefaults=CHAR_TYPE(50),RAW_TYPE(500),CHAR(10) MyApp.sqlj
```

注意： 実行時に、実際のサイズが登録されているパラメータのサイズを超えた場合は、実行時エラーが発生します。

コマンドラインの構文 `-optparamdefaults=defaults_string`

コマンドラインの例 `-optparamdefaults=VAR%(50),LONG%(500),RAW_TYPE()`

プロパティ・ファイルの構文 `sqlj.optparamdefaults=defaults_string`

プロパティ・ファイルの例 `sqlj.optparamdefaults=VAR%(50),LONG%(500),RAW_TYPE()`

デフォルト値 `null`

空白埋めを考慮した CHAR 比較 (-fixedchar)

WHERE 句の比較で文字列をバインドするときに CHAR 型のデータベース列の空白埋めを考慮するには、このフラグを true に設定します。この比較方法では、たとえば、"mystring" と "mystring " は同等であるとみなされます。

この機能は、Oracle JDBC の `setFixedCHAR()` メソッド（埋込みを考慮する Oracle の拡張機能）を使用します。標準 JDBC の `setString()` メソッドは、空白埋めを考慮しません。

-fixedchar の使用例を示します。

```
% sqlj -fixedchar MyProgram.sqlj AnotherProg.java ...
```

注意：

- このトランスレータ・オプションは、Oracle カスタマイザの `fixedchar` オプションと同じです。このトランスレータ・オプションは、プロファイルがないデフォルトの Oracle 固有コード生成（`-codegen=oracle`）のために作成されたものです。しかし、ISO 標準コード生成（`-codegen=iso`）にも適用されます。この場合、トランスレータ・オプションを設定すると、同様にカスタマイザ・オプションも自動的に設定されます。A-30 ページの「[Oracle カスタマイザによる空白埋めを考慮した CHAR 比較 \(fixedchar\)](#)」を参照してください。
 - CHAR 列または VARCHAR2 列の場合、Oracle SQL では NULL 値と "" 値（空の文字列）は同じ意味です。ただし、"" の文字列は挿入できませんが、IS NULL 構文を使用せずに "" の文字列を正しく比較することはできません。-fixedchar 機能を使用することは、この問題の解決にはなりません。
-

コマンドラインの構文 `-fixedchar<=true|false>`

コマンドラインの例 `-fixedchar`

プロパティ・ファイルの構文 `sqlj.fixedchar<=true|false>`

プロパティ・ファイルの例 `sqlj.fixedchar`

デフォルト値 `false`

拡張トランスレータ・オプション

ここでは、SQLJ を実行する際に指定できる、拡張オプションとフラグの構文および機能について説明します。また、JVM、Java コンパイラまたは SQLJ プロファイル・カスタマイザにオプションを渡す際に使用する接頭辞についても説明します。こうしたオプションを使用すると、Oracle SQLJ の拡張機能を実行できます。プロパティ・ファイル（`sqlj.properties` など）でも指定できるオプションの構文も示します。（詳細は、8-16 ページの「[プロパティ・ファイルによるオプション設定](#)」を参照してください。）

ここでは、次のカテゴリについて説明します。

- [オプション設定を他の実行可能プログラムに渡す接頭辞](#)
- [特殊処理のフラグ](#)
- [セマンティクス・チェックとオフライン解析のオプション](#)

Java 規格以外の環境を使用するときに設定する詳細オプションは、8-76 ページの「[トランスレータによる代替環境のサポートとオプション](#)」で説明します。基本的なコマンドライン・フラグとコマンドライン・オプションは、8-21 ページの「[基本的なトランスレータ・オプション](#)」で説明しています。

オプション設定を他の実行可能プログラムに渡す接頭辞

Java インタプリタ、Java コンパイラおよび SQLJ プロファイル・カスタマイザに渡すオプションには、それぞれ次のフラグが付けられます。

- `-J`（Java インタプリタに渡すオプションに付きます。）
- `-C`（Java コンパイラに渡すオプションに付きます。）
- `-P`（プロファイル・カスタマイザに渡すオプションに付きます。ISO コード生成の場合にのみ使用します。）

Java Virtual Machine (JVM) に渡すオプション (-J)

SQLJ を起動した JVM に渡すオプションをコマンドラインで指定するときは、`-J` 接頭辞を付けます。この接頭辞は、JVM オプションのすぐ前に入力します。スペースは挿入しません。sqlj スクリプトにより、Java オプションは `-J` 接頭辞が削除された後で JVM に渡されます。

次にその例を示します。

```
-J-Duser.language=ja
```

sqlj スクリプトにより、`-Duser.language=ja` 引数は `-J` 接頭辞が削除された後で JVM に渡されます。Sun 社の JDK では、フラグ `-Duser.language=ja` を指定すると、システム・プロパティ `user.language` が値 `ja` (日本語の意) に設定されます。ただし、使用する Java 実行可能プログラムに依存するフラグを使用すると、sqlj スクリプトで解釈や動作の決定が行われない場合もあります。

プロパティ・ファイルは JVM の起動後に読み取られるため、プロパティ・ファイルから JVM にオプションを渡すことは不可能です。

注意：

- プロパティ・ファイルを使用して、SQLJ トランスレータを実行する JVM にオプションを直接渡すことはできませんが、この目的で環境変数 `SQLJ_OPTIONS` を使用することは可能です。8-19 ページの「[環境変数 SQLJ_OPTIONS によるオプションの設定](#)」を参照してください。一方、Java コンパイラが実行されている JVM にオプションを渡す際は、プロパティ・ファイルがあればそれを使用します。詳細は、8-59 ページの「[Java コンパイラに渡すオプション \(-C\)](#)」を参照してください。
 - Java プロパティ・ファイル (`sqlj.properties` や `connect.properties` など) には、JVM の `file.encoding` 設定値を使用できません。プロパティ・ファイルでは、常に `8859_1` エンコーディングを使用します。これは、特別な SQLJ の機能ということではなく、一般的な Java の機能です。ただし、Unicode のエスケープ・シーケンスなら、プロパティ・ファイルで使用できます。(エスケープ・シーケンスを決める際は、`native2ascii` ユーティリティを使用することをお勧めします。9-29 ページの「[native2ascii によるソース・ファイルの文字エンコーディングの変換](#)」を参照してください。)
-

コマンドラインの構文 `-J-Java_option`

コマンドラインの例 `-J-Duser.language=ja`

プロパティ・ファイルの構文 該当なし

プロパティ・ファイルの例 該当なし

デフォルト値 該当なし

Java コンパイラに渡すオプション (-C)

sqlj スクリプトで起動された Java コンパイラに渡すオプションには、-C 接頭辞を付けます。この接頭辞は、Java コンパイラ・オプションのすぐ前に入力します。スペースは挿入しません。sqlj スクリプトにより、コンパイラ・オプションは -C 接頭辞が取り除かれた後で Java コンパイラに渡されます（通常は javac ですが、javac 以外を使用してもかまいません）。

次にその例を示します。

```
-C-nowarn
```

sqlj スクリプトにより、-nowarn 引数は -C 接頭辞が取り除かれた後でコンパイラに渡されます。（-nowarn フラグは、コンパイル中に警告メッセージが表示されないようにする javac オプションです。）

通常、コンパイラ・オプションは変更なしで渡されますが、値を伴うコンパイラ・オプションの設定に等号 (=) を使用している場合（-bootclasspath、-extdirs または -target など）は、オプションがコンパイラに渡されるときに等号が取り除かれます。次に示す例で確認してください。

```
sqlj -C-bootclasspath=/usr/local/packages/jdk1.3.1/jre/lib/rt.jar myfile.sqlj
```

Java コンパイラがそれ自体の JVM で実行されている場合は、コンパイラ経由でオプションを JVM に渡すことができます。この操作を行うには、JVM オプションに接頭辞の -C-J を付けます。接頭辞と JVM オプションの間には、スペースを挿入しません。

次にその例を示します。

```
-C-J-Duser.language=de
```

-C 接頭辞を使用するときは、次の制約を守ってください。

- Java コンパイラで処理された .java ファイルの文字エンコーディングの指定には、-C-encoding を使用しないでください。かわりに、SQLJ の -encoding オプションを使用します。このオプションでは、SQLJ で処理された .sqlj ファイルの文字エンコーディング、および SQLJ で生成された .java ファイルの文字エンコーディングを指定します。このオプションはコンパイラに渡されます。このオプションを使用すると、.sqlj ファイルと .java ファイルが同じ文字エンコーディングを受け取ります。-encoding オプションの詳細は、8-28 ページの「[入力および出力ソース・ファイルの文字エンコーディング \(-encoding\)](#)」を参照してください。

- `.class` ファイル用の出力ディレクトリの指定には、`-C-d` を使用しないでください。かわりに、SQLJ の `-d` オプションを使用します。このオプションでは、生成されたプロファイル・ファイル (`.ser`) 用の出力ディレクトリを指定します。このオプションは Java コンパイラに渡されます。このオプションを使用すると、`.class` ファイルと `.ser` ファイルが同じディレクトリに置かれます。(`-d` オプションの詳細は、8-29 ページの「[.ser および .class ファイルの出力ディレクトリ \(-d\)](#)」を参照してください。)

注意：

- コンパイラ・オプションを指定していても、コンパイルを無効化する (`-compile=false`) と、コンパイラ・オプションは無視されます。
 - コンパイラのヘルプ・オプション (`-help` がコンパイラでサポートされている場合は `-C-help`) は、コマンドラインまたは `SQLJ_OPTIONS` 変数でのみ指定できます。プロパティ・ファイルでは指定できません。SQLJ の `-help` オプションと同様に、変換は行われません。処理するファイルを指定しても実行されません。(SQLJ では、ヘルプと変換の両方でなく、どちらか一方を実行することが前提となっています。)
-

コマンドラインの構文 `-C-Java_compiler_option`

コマンドラインの例 `-C-nowarn`

プロパティ・ファイルの構文 `compile.Java_compiler_option`

プロパティ・ファイルの例 `compile.nowarn`

デフォルト値 該当なし

プロファイル・カスタマイザに渡すオプション (-P)

カスタマイズ・フェーズ (ISO 標準コード生成、つまり `-codegen=iso` の場合のみ) では、`sqlj` スクリプトにより、フロントエンドのカスタマイザ・ハーネスが起動されます。このカスタマイザ・ハーネスは、カスタマイズ処理を調整し、特定のカスタマイザを実行します。カスタマイズのオプションには、次のように `-P` 接頭辞を付けます。

- `-P` 接頭辞のみを使用すると、どのカスタマイザにも適用される汎用的なオプションが、カスタマイザ・ハーネスに渡されます。
- `-P-C` 接頭辞を使用すると、ベンダー固有のオプションが、使用するカスタマイザに渡されます。

`-P` 接頭辞と `-P-C` 接頭辞は、カスタマイザ・オプションのすぐ前に入力します。スペースは挿入しません。`sqlj` スクリプトにより、カスタマイザ・オプションは接頭辞が取り除かれた後で、プロファイル・カスタマイザに渡されます。

SQLJ の `-default-customizer` オプションで定義されたデフォルトのカスタマイザをオーバーライドする場合にも、`-P` 接頭辞を使用します。

`-P-customizer=your_customizer_class`

汎用的なカスタマイザ・オプションの例を示します。

`-P-backup`

`-backup` フラグは、新しいカスタマイズ結果を生成する前に既存のカスタマイズ結果をバックアップする、汎用的なカスタマイザ・オプションです。

次に示すのは、ベンダー固有のカスタマイザ・オプションの例です（ここでは、Oracle 固有のカスタマイザ・オプションを使用しています）。

`-P-Csummary`

(`summary` フラグは、実行されたカスタマイズの情報を出力する、Oracle カスタマイザ・オプションです。)

注意：

- 「`-P-C`」とベンダー固有のカスタマイザ・オプションの間には、ハイフンを挿入しません。他の接頭辞や他の接頭辞の組合せの場合は、接頭辞とオプションの間にハイフンを付けます。
 - カスタマイザのヘルプ・オプション (`-P-help`) は、コマンドラインまたは `SQLJ_OPTIONS` 変数でのみ指定できます。プロパティ・ファイルでは指定できません。SQLJ の `-help` オプションと同様に、変換は行われません。処理するファイルを指定しても実行されません。(SQLJ では、ヘルプと変換の両方でなく、どちらか一方を実行することが前提となっています。)
 - ISO コード生成では、カスタマイズ・オプションを指定していても、`.sqlj` ファイルのカスタマイズを無効にする（また、コマンドラインに `.ser` ファイルがない）と、カスタマイズ・オプションは無視されます。
 - デフォルトの Oracle 固有コード生成 (`-codegen=oracle`) では、プロファイルが生成されずカスタマイズが実行されないため、接頭辞 `-P` を使用できません。
-
-

汎用的なカスタマイザ・オプションと Oracle 固有のカスタマイザ・オプションの詳細は、A-10 ページの「[カスタマイズ・オプションとカスタマイザの選択](#)」を参照してください。

コマンドラインの構文 `-P-<C>profile_customizer_option`

コマンドラインの例

```
-P-driver=oracle.jdbc.OracleDriver
-P-Csummary
```

プロパティ・ファイルの構文 `profile.<C>profile_customizer_option`

プロパティ・ファイルの例

```
profile.driver=oracle.jdbc.OracleDriver
profile.Csummary
```

デフォルト値 該当なし

特殊処理のフラグ

前述のように、通常の場合、`.sqlj` ファイルは SQLJ トランスレータ、Java コンパイラおよび（ISO コード生成の場合は）SQLJ プロファイル・カスタマイザで処理されます。次のフラグは、この処理に制限を付けるものです。SQLJ の起動スクリプトで、指定された過程が省略されます。

- `-compile`
- `-profile`

ISO コード生成の場合、SQLJ では次のフラグを指定すると、シリアル化されたリソース（`.ser`）ファイルのプロファイルが、カスタマイズ後に `class` ファイルに変換されます。

- `-ser2class`

次のフラグを指定すると、特定の状況のもとで SQLJ の型解決が行われます。ソース・ファイルの他、`class` ファイルや SQLJ コマンドラインから指定したファイルが検査の対象となります。

- `-checksource`

次のフラグは、指定された SQLJ 文の中で複数回出現する同一のホスト変数を、単一のバインドとして処理することを指定します。

- `-bind-by-identifier`

コンパイル・フラグ（`-compile`）

`-compile` フラグでは、コンパイラによる `.java` ファイルの処理が、有効または無効にされます。この設定は、生成された `.java` ファイルとコマンドラインで指定された `.java` ファイルの両方に適用されます。たとえば、`javac` 以外のコンパイラを使用して、`.java` ファイルを後でコンパイルする場合、このフラグを使用すると便利です。このフラグのデフォルト値は `true` です。`false` に設定すると、コンパイルが無効になります。

-compile=false を指定して .sqlj ファイルを処理する場合は、必要に応じて、後でコンパイルとカスタマイズを行います。

-compile=false を設定すると、-profile=false も暗黙的に設定されます。つまり、-compile が false の場合は、コンパイルとカスタマイズの両方が省略されます。
-compile=false と -profile=true を設定した場合、-profile 設定は無視されます。

注意： 型の解決のために .java ファイルを参照する場合でも、-compile を false に設定する方が適切な場合があります。たとえば、.sqlj ファイルの変換中に行われる型解決に 1 つ以上の .java ファイルをコマンドラインで指定する場合、特定のコンパイラを使用してすべての .java ファイルを後でコンパイルするには、この設定を行います。

ただし、-checksource オプションは、SQLJ コマンドラインでの解決に .java ファイルを入力する必要性を排除することによって、型解決の処理を単純化できることに注意してください。8-65 ページの「[型解決を目的としたソース・チェック \(-checksource\)](#)」を参照してください。

コマンドラインの構文 -compile<=true|false>

コマンドラインの例 -compile=false

プロパティ・ファイルの構文 sqlj.compile<=true|false>

プロパティ・ファイルの例 sqlj.compile=false

デフォルト値 true (コンパイル)

プロファイル・カスタマイズ・フラグ (-profile)

ISO コード生成の場合、-profile フラグは、SQLJ プロファイル・カスタマイズによって生成されたプロファイル (.ser) ファイルの処理を有効または無効にします。この設定は、SQLJ トランスレータによって、現行のコマンドラインで指定した .sqlj ファイルから生成された .ser ファイルにのみ適用されます。コマンドラインで指定した、前に生成された .ser ファイル (または .jar ファイル) には適用されません。このフラグのデフォルト値は true です。false に設定すると、カスタマイズが無効になります。

このオプションは、コマンドラインで指定したファイルの -compile オプションとは異なった働きをします。コマンドラインで指定された .ser ファイルと .jar ファイルは、-profile=false を設定してもカスタマイズされます。これに対して、コマンドラインで指定された .java ファイルは、-compile=false を設定するとコンパイルされません。これは、.java ファイルに対しては、行マッピングなどの他の操作を実行できるためです。コマンドラインで指定された .ser ファイルや .jar ファイルに対しては、他の操作を実行できません。

-profile=false を指定して .sqlj ファイルを処理する場合は、必要に応じて、後でカスタマイズを行います。

注意：

- Oracle カスタマイズを変換処理で使用する場合は、アプリケーションの実行時に Oracle SQLJ ランタイムおよび Oracle JDBC ドライバを必要としない場合は、-profile=false を指定します。（または -default-customizer オプションを使用して、デフォルト以外のカスタマイズを指定する方法もあります。）カスタマイズを実行しないと、アプリケーション実行時に汎用 SQLJ ランタイムが使用されます。
 - -compile=false を設定すると、-profile=false も暗黙的に設定されます。つまり、-compile が false の場合は、コンパイルとカスタマイズの両方が省略されます。-compile=false と -profile=true を設定した場合、-profile 設定は無視されます。
 - デフォルトの Oracle 固有コード生成 (-codegen=oracle) では、プロファイルが生成されずカスタマイズが実行されないため、このオプションを使用できません。
-

コマンドラインの構文 -profile<=true|false>

コマンドラインの例 -profile=false

プロパティ・ファイルの構文 sqlj.profile<=true|false>

プロパティ・ファイルの例 sqlj.profile=false

デフォルト値 true（カスタマイズ）

.ser ファイルから .class ファイルへの変換（-ser2class）

ISO 標準 SQLJ コード生成 (-codegen=iso) の場合、-ser2class フラグでは、生成された .ser ファイルが .class ファイルに変換されます。末尾に .ser の付くリソース・ファイル名がサポートされないブラウザから実行されるアプレットを、SQLJ を使用して作成する場合、この処理が必要です。（たとえば、Netscape Navigator 4.x から実行するアプリケーションなどがこれに相当します。）

また、クライアント側で SQLJ プログラムを変換してから、クラス・ファイルとリソース・ファイルをサーバーにロードする場合、この処理を行うとプロファイル用のスキーマ・オブジェクトの名前付けが簡略化されます。ロード済みのクラス・スキーマ・オブジェクトのネーミング規則は、ロード済みのリソース・スキーマ・オブジェクトよりも簡略化されてい

ます。（詳細は、11-10 ページの「ロードされたクラスとリソース・スキーマ・オブジェクトの名前付け」を参照してください。）

プロファイルのカスタマイズ後に変換が行われ、独自のカスタマイズが取り込まれます。

変換後のファイルのベース名は、元のファイルのベース名と同じになります。ファイル名の `.ser` のみが `.class` に置き換えられます。次にその例を示します。

```
Foo_SJProfile0.ser
```

このファイル名は次のファイル名に変換されます。

```
Foo_SJProfile0.class
```

注意：

- 元の `.ser` ファイルは保存されません。
 - プロファイルを `.class` ファイルに変換すると、これ以上はカスタマイズできなくなります。この場合は、`.class` ファイルを削除し、SQLJ を再度実行してプロファイルを再作成する必要があります。
 - 文字エンコーディングが必要な場合、`-ser2class` オプションでは常に文字エンコーディング `8859_1` が使用され、SQLJ の `-encoding` 設定は無視されます。
 - デフォルトの Oracle 固有コード生成 (`-codegen=oracle`) を使用する場合は、プロファイルが生成されないため、`-ser2class` オプションは適用されません。
-
-

コマンドラインの構文 `-ser2class<=true|false>`

コマンドラインの例 `-ser2class`

プロパティ・ファイルの構文 `sqlj.ser2class<=true|false>`

プロパティ・ファイルの例 `sqlj.ser2class`

デフォルト値 `false`

型解決を目的としたソース・チェック (`-checksource`)

CLASSPATH 内のクラス・ファイルと SQLJ コマンドラインで指定されたクラス・ファイルまたはソース・ファイルのみを検証するのでは、SQLJ 型解決処理には十分ではありません。SQLJ では `-checksource` フラグを使用すると、次の条件に該当する場合、CLASSPATH 中にあるソース・ファイルも検査の対象になります。

- 要求されたクラスのクラス・ファイルは見つからないが、ソース・ファイルが見つかった場合
- ソース・ファイルの修正日付が、対応するクラス・ファイルよりも新しい場合

注意： この型解決が有効になるのは、Java 型の定義が #sql 文中にある場合にのみであって、Java コード中の他の位置では有効になりません。このため、必要な .sqlj ファイルの名前は、SQLJ コマンドラインで明示的に指定する必要があります。

コマンドラインの構文 -checksource<=true|false>

コマンドラインの例 -checksource=false

プロパティ・ファイルの構文 sqlj.checksource=<=true|false>

プロパティ・ファイルの例 sqlj.checksource=false

デフォルト値 true

識別子によるホスト式のバインド (-bind-by-identifier)

SQLJ 標準と同様に、Oracle SQLJ のデフォルトでは、同じホスト変数が複数回出現する場合でも、1 文中の各ホスト変数のバインド参照には一意の名前を作成します。SQLJ 標準は JDBC に基づいており、JDBC は同じ変数を異なる位置にバインドするように設計されていません。各バインド位置（「?」で識別）が個々の値にバインドされます。

このため、状況によっては、次の例のような場合にエラーが発生することがあります。

```
#sql emps = { SELECT substr(ename, 1, :bind_var), sum(sal) FROM emp
              GROUP BY substr(ename, 1, :bind_var) };
```

2 つの bind_var に対してバインド参照名が個別に作成されるため、実行時に結果が SQL 例外となります。異なるバインド名が検出されると、SQL エンジンでは GROUP BY 句が SELECT 構文のリストの一部ではないと判断します。

この問題を回避するために、Oracle SQLJ では、-bind-by-identifier フラグを使用して標準機能が拡張されています。true に設定されている場合は、指定された SQLJ 文または PL/SQL ブロック内に同じ識別子が複数回出現しても、単一のバインドとして処理されます。4 つのバインド操作 (:x、:x、:y、:x) が行われる SQLJ 文は、:1、:2、:3、:4 ではなく、:1、:1、:2、:1 としてバインドされます。

前述の例で、双方のバインド操作は、substr(ename, 1, :1) および substr(ename, 1, :2) ではなく、substr(ename, 1, :1) となります。

注意： `-bind-by-identifier` フラグは、ホスト変数が単純なホスト式にのみ適用します。

コマンドラインの構文 `-bind-by-identifier<=true|false>`

コマンドラインの例 `-bind-by-identifier`

プロパティ・ファイルの構文 `sqlj.bind-by-identifier=<=true|false>`

プロパティ・ファイルの例 `sqlj.bind-by-identifier`

デフォルト値 `false`

セマンティクス・チェックとオフライン解析のオプション

次の各オプションで、オンラインおよびオフラインのセマンティクス・チェックとオフライン解析を指定します。

- `-offline`
- `-online`
- `-cache`
- `-parse`

これらのオプションを説明する前に、次の2項目について説明します。

- セマンティクス・チェックのデフォルトのフロントエンド・クラス `OracleChecker` および `Oracle` のセマンティクス・チェッカ
- オンライン・セマンティクス・チェックとオフライン解析との比較

注意： オンライン・セマンティクス・チェックは、トランスレータの `-user` オプションを設定することによって有効になります。8-33 ページの「[オンライン・セマンティクス・チェックとユーザー名 \(-user\)](#)」を参照してください。ただし、オフライン解析を有効または無効にするための `-parse` オプションを設定すると、オンライン・セマンティクス・チェックの設定は無視されます。8-75 ページの「[オフライン・パーサー \(-parse\)](#)」を参照してください。

セマンティクス・チェッカおよび OracleChecker フロントエンド（デフォルトのチェッカ）

Oracle では、Oracle 専用のオフライン・チェッカ、汎用のオフライン・チェッカ、Oracle 専用のオンライン・チェッカ、汎用のオンライン・チェッカが提供されています。汎用チェッカは、SQL92 および JDBC の標準機能のみが使用されていることを前提に動作します。Oracle データベースを使用する場合は、Oracle 専用チェッカを使用することをお勧めします。

デフォルトのチェッカは、オンライン・チェックおよびオフライン・チェックとも `oracle.sqlj.checker.OracleChecker` で、ほとんどの環境で十分に機能します。このクラスはフロントエンドとして機能し、使用する環境やオフラインとオンラインの選択に応じて、適切なセマンティクス・チェッカを実行します。

Oracle の場合は、チェックがオンラインかオフラインかにかかわらず、次のチェッカがあります。

- Oracle9i および Oracle8i の型に対する Oracle8 チェッカ（Oracle9i および Oracle8i JDBC で使用）
- Oracle 8.0.x 型用の Oracle80 チェッカ（Oracle 8.0.x JDBC で使用）。
- Oracle 7.3.x 型用の Oracle7 チェッカ（Oracle 7.3.x または Oracle 8.0.x JDBC で使用）。
- Oracle8i または Oracle9i の JDBC ドライバを使用するための Oracle8To7 チェッカ。ただし、Oracle 7.3.x データベース対応の型のサブセットのみチェック可能です。

Oracle80 および Oracle7 チェッカは、Oracle8i および Oracle9i の JDBC ドライバに非互換で、また Oracle8 および Oracle8To7 チェッカは、Oracle 8.0.x および Oracle 7.3.x JDBC ドライバに非互換となっています。Oracle8i または Oracle9i の JDBC ドライバを使用して Oracle 7.3.x 対応の型のサブセットをチェックできるように、Oracle8To7 チェッカが作成されました。

Oracle データベースと JDBC ドライバによるオンライン・チェック オンライン・チェックに Oracle データベースと Oracle JDBC ドライバを使用する場合、データベースと JDBC ドライバの低い方のバージョンにあわせて、OracleChecker がチェッカを選択します。表 8-4 に、データベースのバージョンと JDBC ドライバのバージョンとの組合せと、他の有効な Oracle チェッカを一覧の形式で示します。

表 8-4 OracleChecker で選択される Oracle オンライン・セマンティクス・チェッカ

データベースのバージョン	JDBC のバージョン	オンライン・チェッカの選択	他の有効なオンライン・チェッカ
Oracle9i、Oracle8i または Oracle8 8.0.x	Oracle9i または Oracle8i	Oracle8JdbcChecker	Oracle8To7JdbcChecker
Oracle9i、Oracle8i または Oracle8 8.0.x	Oracle8 8.0.x	Oracle80JdbcChecker	Oracle7JdbcChecker
Oracle9i、Oracle8i または Oracle8 8.0.x	Oracle7 7.3.x	Oracle7JdbcChecker	なし
Oracle7 7.3.x	Oracle9i または Oracle8i	Oracle8To7JdbcChecker	なし
Oracle7 7.3.x	Oracle8 8.0.x	Oracle7JdbcChecker	なし
Oracle7 7.3.x	Oracle7 7.3.x	Oracle7JdbcChecker	なし

JDBC ドライバによるオフライン・チェック オフライン・チェックに Oracle JDBC ドライバを使用する場合、JDBC ドライバのバージョンにあわせて、OracleChecker がチェッカを選択します。チェッカの選択については、表 8-5 に要約します。（ここには Oracle8To7OfflineChecker も記載されていますが、これを使用するには手動で選択する必要があります。）

表 8-5 OracleChecker で選択される Oracle オフライン・セマンティクス・チェッカ

JDBC のバージョン	オフライン・チェッカの選択	他の有効なオフライン・チェッカ
Oracle9i または Oracle8i	Oracle8OfflineChecker	Oracle8To7OfflineChecker
Oracle8 8.0.x	Oracle80OfflineChecker	Oracle7OfflineChecker
Oracle7 7.3.x	Oracle7OfflineChecker	なし

Oracle データベースと JDBC ドライバを使用しない場合 Oracle JDBC ドライバが使用されないことを検出すると、OracleChecker は次のいずれかのチェッカを実行します。

- オンライン・チェックが無効になっている場合は、`sqlj.semantics.OfflineChecker` を実行します。
- オンライン・チェックが有効になっている場合は、`sqlj.semantics.JdbcChecker` を実行します。

オンライン・セマンティクス・チェックとオフライン解析との比較

SQLJ の初期リリースでは、変換時の SQL 検証にはオンライン・セマンティクス・チェックのためのデータベース接続が必要でした。Oracle9i リリース 2 (9.2) では、制限付きで代替チェックを提供するオフライン解析と呼ばれる機能があります。オフライン解析はデータベース接続を使用しません。したがって、データベース・スキーマに対する操作の検証は実行できませんが、SQL 文と PL/SQL 文すべての構文チェックを実行できます。以前のリリースでは、この機能はオフライン・モードでは使用できませんでした。

オフライン解析およびオンライン・セマンティクス・チェックで提供される機能の比較を表 8-6 にまとめます。

表 8-6 機能の比較：オフライン解析とオフライン・セマンティクス・チェック

機能	オフライン解析	オンライン・チェック
DML、SELECT および PL/SQL 構文の検証	あり	あり
DDL 構文の検証	あり	なし
DML、SELECT および PL/SQL セマンティクスの検証 (データベース・スキーマとの比較)	なし	あり
DDL セマンティクスの検証 (データベース・スキーマとの比較)	なし	なし

注意： 前述の表の「DDL」とは、CREATE、ALTER、DROP および SET などの SQL 文を意味します。

オンライン・チェックには、データベース・スキーマに対する SQL および PL/SQL の操作を検証できるという主なメリットがあります。オンライン・チェックでは、列の型が SQL 操作と一致することの検証、およびコールされるストアド・プロシージャの存在の確認も行われます。ただし、オンライン・チェックでは変換時にデータベース接続が必要です。状況によっては、これが問題の原因となる場合があります。オンライン・チェックでは DDL 操作の検証も実行されます。

オフライン解析には、変換時にデータベースに接続せずに SQL 構文をチェックできるメリットがあります。この構文チェックでは DDL 操作の検証も行われます。

いずれのモードでも、データベース・スキーマに対する DDL セマンティクス・チェックは行われません。

注意：

- オフライン解析とオンライン・チェックの両方が有効化されている場合、一部の種類のエラーは2回レポートされます。
- オフライン・パーサーまたはオンライン・チェッカによって検出される問題は、致命的なレベルではなく警告レベルまたは助言レベルでレポートされます。
- オフライン解析とオフライン・セマンティクス・チェックを混同しないでください。オフライン・チェックは、オンライン・チェックが有効かどうか、およびオフライン解析が有効かどうかに関係なく、常に発生する基本的なセマンティクス・チェック処理で構成されます。この処理には、SQLJの実行文で使用されているJava式の分析や、(SELECTなどの) キーワードに従って埋め込まれているSQL操作の分類などがあります。
- 弱い型指定のホスト式に対するデータの互換性はチェックされません。(弱い型指定のホスト式とは、`oracle.sql` パッケージの `STRUCT`、`REF`、`ARRAY` の各クラスが使用されたホスト式のことです。詳細は、6-74 ページの「[弱い型指定のオブジェクト](#)、[参照およびコレクション](#)」を参照してください。)
- PL/SQL の無名ブロック内で使用されている式のモード互換性 (IN、OUT または IN OUT) はチェックされません。

オフライン・セマンティクス・チェッカ (-offline)

-offline オプションには、オフライン・チェックのための、SQLJ のセマンティクス・チェック・コンポーネントを実装した Java クラスを指定します。オフライン・チェックでは、データベース接続がありません。SQL 構文および Java 型の使用方法のみがチェックされます。(オフライン・セマンティクス・チェッカとオンライン・セマンティクス・チェッカの詳細は、9-2 ページの「[SQL のセマンティクス・チェックとオフライン解析](#)」を参照してください。)

-offline オプションでオフライン・セマンティクスが有効または無効になることはありません。オンライン・チェックが有効になっていないか、データベース接続が確立できないと、オンライン・チェックが実行されません。このときに限って、オフライン・チェックが実行されます。

接続コンテキストごとに異なるオフライン・チェッカを指定できます。ただし、1つのコンテキストについて1つのチェッカのみを指定します。(1つの接続コンテキストに複数のオフライン・チェッカを指定しないでください。)

通常は、8-68 ページの「[セマンティクス・チェッカおよび OracleChecker フロントエンド \(デフォルトのチェッカ\)](#)」で説明したフロントエンド・クラスである、デフォルトの `OracleChecker` を使用できますが、`OracleChecker` で選択されない特定のチェッカを指

定する場合は別です。たとえば、オフライン・チェックの対象が Oracle8 リリース 8.0 JDBC ドライバがインストールされたマシンでも、アプリケーション（または特定の接続コンテキスト・クラスを使用する文）は Oracle7 リリース 7.3 データベースを実行対象としている場合があります。こうした文をチェックする場合には、Oracle7 のチェッカを使用することをお勧めします。

次の例は、特定の接続コンテキスト（CtxClass）をチェックするために Oracle7 のオフライン・チェッカを選択する方法を示しています。

```
-offline@CtxClass=oracle.sqlj.checker.Oracle7OfflineChecker
```

この例では、SQLJ で `oracle.sqlj.checker.Oracle7OfflineChecker` が使用され、CtxClass インスタンスである接続オブジェクトを指定する SQLJ 実行文について、オフライン・チェックが行われます。

接続コンテキスト・クラス CtxClass は、ソース・コード中に宣言するか、あらかじめ `.class` ファイルにコンパイルしておく必要があります。（詳細は、7-2 ページの「[接続コンテキスト](#)」を参照してください。）

-offline オプションは、各接続コンテキストのオフライン・チェッカごとに設定します。各接続コンテキストの設定は互いに影響しません。次にその例を示します。

```
-offline@CtxClass2=oracle.sqlj.checker.Oracle7OfflineChecker  
-offline@CtxClass3=sqlj.semantics.OfflineChecker
```

デフォルトの接続コンテキスト、およびオフライン・チェッカを指定しない他の接続コンテキストに対してオフライン・チェッカを指定するには、次のようにします。

```
-offline=oracle.sqlj.checker.Oracle7OfflineChecker
```

オフライン・チェッカが指定されていない接続コンテキストでは、デフォルトの接続コンテキストのオフライン・チェッカ設定が使用されます。ここでは、オフライン・チェッカがデフォルトのコンテキストに設定されていることを前提とします。

コマンドラインの構文 `-offline<@conn_context_class>=checker_class`

コマンドラインの例

```
-offline=oracle.sqlj.checker.Oracle80OfflineChecker  
-offline@CtxClass=oracle.sqlj.checker.Oracle80OfflineChecker
```

プロパティ・ファイルの構文 `sqlj.offline<@conn_context_class>=checker_class`

プロパティ・ファイルの例

```
sqlj.offline=oracle.sqlj.checker.Oracle80OfflineChecker  
sqlj.offline@CtxClass=oracle.sqlj.checker.Oracle80OfflineChecker
```

デフォルト値 `oracle.sqlj.checker.OracleChecker`

オンライン・セマンティクス・チェッカ (-online)

-online オプションには、SQLJ のオンライン・セマンティクス・チェック・コンポーネントを実装した Java クラスまたはクラスのリストを指定します。この処理では、データベースへの接続が行われます。

オンライン・チェックは、-online オプションでは有効になりません。-user オプションを使用して有効にする必要があります。-password、-url および -driver の各オプションも、適切に設定されていることが必要です。（オフライン・セマンティクス・チェッカとオンライン・セマンティクス・チェッカの詳細は、9-2 ページの「[SQL のセマンティクス・チェックとオフライン解析](#)」を参照してください。）

注意： SQLJ の -parse オプションの設定によっては、-user オプションの効果が無視され、オンライン・セマンティクス・チェックも無効になります。8-75 ページの「[オフライン・パーサー \(-parse\)](#)」を参照してください。

接続コンテキストごとに、別のオンライン・チェッカを指定できます。また、1つのコンテキストに対して複数のチェッカを指定することも可能です。ただし、各チェッカをカンマで区切る必要があります。1つのコンテキストに複数のチェッカが指定されている場合は、(リストの左から右に向かって) 最初に指定されているチェッカが使用されます。このチェッカは、オンライン・チェック用に確立されたデータベース接続を受け入れるチェッカになります。分析時には接続が各オンライン・チェッカに渡され、チェッカはデータベースが認識されたかどうかを確認します。

通常は、8-68 ページの「[セマンティクス・チェッカおよび OracleChecker フロントエンド \(デフォルトのチェッカ\)](#)」で説明したフロントエンド・クラスである、デフォルトの OracleChecker を使用できますが、OracleChecker で選択されない特定のチェッカを指定する場合は別です。たとえば、オンライン・チェックの対象が Oracle8 リリース 8.0 データベースと JDBC ドライバがインストールされたマシンでも、アプリケーション（または特定の接続コンテキスト・クラスを使用する文）は Oracle7 リリース 7.3 データベースを実行対象としている場合があります。こうした文をチェックする場合には、Oracle7 のチェッカを使用することをお勧めします。

次の例では、DefaultContext クラスおよび設定値を指定していない接続コンテキスト・クラスをチェックするために、Oracle7 のオンライン・チェッカを選択します。

```
-online=oracle.sqlj.checker.Oracle7JdbcChecker
```

ドライバのリストを指定し、アクセスしているデータベースの種類に応じて適切なクラスが選択されるようにするには、次のように指定します。

```
-online=oracle.sqlj.checker.Oracle7JdbcChecker,sqlj.semantics.JdbcChecker
```

このように指定した場合、Oracle データベースに接続した後で

`oracle.sqlj.checker.Oracle7JdbcChecker` セマンティクス・チェッカーが使用されます。他の種類のデータベースへの接続が行われた場合は、汎用的な `sqlj.semantics.JdbcChecker` セマンティクス・チェッカーが使用されます。これは、機能的には、デフォルトの `OracleChecker` の場合と同じです。ただし、Oracle データベースに接続した場合に、Oracle8 のチェッカーではなく、必ず Oracle7 のチェッカーが使用されます。

特定の接続コンテキスト (CtxClass) 用にオンライン・チェッカーを指定するには、次のようにします。

```
-online@CtxClass=oracle.sqlj.checker.Oracle7JdbcChecker
```

この例では、`oracle.sqlj.checker.Oracle7JdbcChecker` が使用され、CtxClass のインスタンスである接続オブジェクトを指定する SQLJ 実行文に対して、オンライン・チェックが行われます。ここでは、CtxClass に対するオンライン・チェックが有効になっていることを前提としています。

接続コンテキスト・クラス CtxClass は、ソース・コード中に宣言するか、あらかじめ .class ファイルにコンパイルしておく必要があります。(詳細は、7-2 ページの「[接続コンテキスト](#)」を参照してください。)

-online オプションは、各接続コンテキストのオンライン・チェッカーごとに設定します。これらの設定は互いに影響しません。

```
-online@CtxClass2=oracle.sqlj.checker.Oracle80JdbcChecker
-online@CtxClass3=sqlj.semantics.JdbcChecker
```

オンライン・チェッカーを設定していない接続コンテキストは、デフォルトの接続コンテキストのオンライン・チェッカー設定を使用します。

コマンドラインの構文 `-online<@conn_context_class>=checker_class(list)`

コマンドラインの例

```
-online=oracle.sqlj.checker.Oracle80JdbcChecker
-online=oracle.sqlj.checker.Oracle80JdbcChecker,sqlj.semantics.JdbcChecker
-online@CtxClass=oracle.sqlj.checker.Oracle80JdbcChecker
```

プロパティ・ファイルの構文 `sqlj.online<@conn_context_class>=checker_class(list)`

プロパティ・ファイルの例

```
sqlj.online=oracle.sqlj.checker.Oracle80JdbcChecker
sqlj.online=oracle.sqlj.checker.Oracle80JdbcChecker,sqlj.semantics.JdbcChecker
sqlj.online@CtxClass=oracle.sqlj.checker.Oracle80JdbcChecker
```

デフォルト値 `oracle.sqlj.checker.OracleChecker`

オンライン・セマンティクス・チェッカ実行結果のキャッシング (-cache)

-cache オプションを使用すると、オンライン・チェッカで生成された結果のキャッシングが有効になります。キャッシングを有効化すると、以後の SQLJ 変換実行時にさらにデータベース接続が行われることがなくなります。分析結果は、カレント・ディレクトリに格納してある SQLChecker.cache ファイル中にキャッシングされます。

キャッシュ中には、変換の正常終了（エラー・メッセージも警告メッセージも発生せずに完了）した SQL 文が、シリアル化されて格納されます。内容には、文のパラメータ、戻り型、トランスレータ設定およびモードが含まれます。

キャッシュは累積し、SQLJ トランスレータが起動されるたびに増大します。キャッシュを空にするには、SQLChecker.cache ファイルを削除します。

コマンドラインの構文 -cache<=true|false>

コマンドラインの例 -cache

プロパティ・ファイルの構文 sqlj.cache<=true|false>

プロパティ・ファイルの例 sqlj.cache

デフォルト値 false

オフライン・パーサー (-parse)

-parse オプションを使用してオフライン解析を有効化します。オフライン解析は、オンライン・セマンティクス・チェックを補完し、変換時にデータベースに接続せずに SQL および PL/SQL の構文チェックを実行します。オフライン解析では、オンライン・チェックでは実行されない DDL 文の構文もチェックされます。機能の比較は、8-70 ページの「[オンライン・セマンティクス・チェックとオフライン解析との比較](#)」を参照してください。

また、-parse オプションの設定は、-user オプションによって有効になるオンライン・チェックよりも優先されます。使用可能な -parse 設定は、次のとおりです。

- both (デフォルト) : オフライン・パーサーとオンライン・チェックを有効にします。この場合、オンライン・チェックは、-user オプションによって判断されます。
- online-only: オフライン・パーサーを無効にし、オンライン・チェックを有効にします。この場合もオンライン・チェックは、-user オプションによって判断されます。
- offline-only: オフライン・パーサーを有効にし、オンライン・チェックを無効にします。この設定によって、オンライン・チェックを有効にする -user オプションは無視されます。
- none: オフライン・パーサーとオンライン・チェックを無効にします。この設定によって、オンライン・チェックを有効にする -user オプションは無視されます。

- `parserclassname`: 代替 SQL パーサーを実装する Java クラス名を指定します。このクラスは `sqlj.framework.checker.SimpleChecker` インタフェースを実装する必要があります。この設定によって指定パーサーが有効になり、SQL チェックに使用されるパーサーは、このパーサーのみになります。標準のオフライン・パーサーとオンライン・チェックは、両方とも無効になります。

`offline-only` と `none` の設定は、通常の操作モードではなく、ユーザー独自のパーサーも一般的には指定しません。`-user` オプションでオンライン・チェックを判断します。

注意： オフライン解析とオンライン・チェックの両方を有効にするモードでは、一部の問題が重複してレポートされることがあります。

コマンドラインの構文 `-parse=both|online-only|offline-only|none|parserclassname`

コマンドラインの例 `-parse=online-only`

プロパティ・ファイルの構文

`sqlj.parse=both|online-only|offline-only|none|parserclassname`

プロパティ・ファイルの例 `sqlj.parse=online-only`

デフォルト値 `both`

トランスレータによる代替環境のサポートとオプション

デフォルトで、Oracle SQLJ は Sun 社の JDK 1.2.x 以上または 1.1.x 環境で実行するように構成され、Sun 社のコンパイラを使用するようになっています。ただし、これらは必須条件ではありません。SQLJ は、他の JVM やコンパイラで機能するように設定することも可能です。この設定を行うには、SQLJ で次の情報を指定する必要があります。

- 使用する JVM の名前 (`-vm` オプション)
- 使用する Java コンパイラの名前 (`-compiler-executable` オプション)
- コンパイラに必要な設定

この情報を指定する際には、一連の SQLJ オプションを使用できます。これらのオプションについては、「[Java およびコンパイラのオプション](#)」で説明します。

また、デフォルトの設定では、SQLJ で Oracle プロファイル・カスタマイザが使用されますが、他のカスタマイザを指定することも可能です。他のカスタマイザを指定する方法は、8-84 ページの「[カスタマイズのオプション](#)」を参照してください。

注意： どのようなオペレーティング・システムおよび環境を使用している場合も、制限事項に留意してください。特に、展開された SQLJ コマンドラインのサイズが、許容されるコマンドライン・サイズの最大値を超えないようにしてください（たとえば、Windows 95 の場合は 250 文字以内、Windows NT の場合は 4000 文字以内です）。詳細は、使用するオペレーティング・システムのドキュメントを参照してください。

Java およびコンパイラのオプション

次の各オプションは、JVM および Java コンパイラの操作を指定するオプションです。

- `-vm` (JVM を指定。コマンドラインのみ。)
- `-compiler-executable` (Java コンパイラを指定。)
- `-compiler-encoding-flag`
- `-compiler-output-file`
- `-compiler-pipe-output-flag`

標準 `javac` をはじめとする一部のコンパイラでは、定義済みの `Public` クラスがあれば、それと同じ名前を `Java` ソース・ファイル名に付ける必要があります。このため、このファイル名が `Public` クラス名と同じかどうかの検査が、SQLJ トランスレータではデフォルトで実行されます。SQLJ でこうした検証が実行されないように指定する場合は、次のオプションを使用します。

- `-checkfilename`

JVM やコンパイラの設定によっては、SQLJ でコンパイラを起動する通常の方法で、問題が発生する可能性があります。こうした問題を回避するには、SQLJ での処理を 2 段階で実行します。次のオプションを使用してください。

- `-passes`

`-J` および `-c` 接頭辞を指定すると、使用する JVM やコンパイラにオプションを直接渡すことが可能になります。これらの接頭辞については、8-57 ページの「[オプション設定を他の実行可能プログラムに渡す接頭辞](#)」で説明しています。

注意： `-vm` オプション、`-passes` オプションおよび接頭辞 `-J` は、プロパティ・ファイルでは使用できません。これらはコマンドラインで設定する方法も、より簡単に環境変数 `SQLJ_OPTIONS` で設定する方法もあります。8-19 ページの「[環境変数 SQLJ_OPTIONS によるオプションの設定](#)」を参照してください。

Java Virtual Machine (JVM) の名前 (-vm)

SQLJ で特定の JVM を使用する場合は、-vm オプションで指定します。それ以外の場合は、Sun 社の JDK 標準の java が使用されます。

このオプションはプロパティ・ファイルでは設定できません。プロパティ・ファイルは、JVM が起動された後に読み込まれるためです。

SQLJ では、JVM の実行可能ファイル名と一緒にディレクトリ・パスを指定しないと、オペレーティング・システムの PATH 変数に基づいて実行可能ファイルの検索が開始されます。

注意： このオプションの特殊機能、-vm=echo がサポートされています。この機能は、-n オプションに相当するものです。この機能を使用すると、sqlj スクリプトによって、SQLJ トランスレータに渡されるコマンドライン全体が構築され、トランスレータによる実行なしにユーザーにエコーされます。詳細は、8-26 ページの「[コマンドラインのエコー \(-n\)](#)」を参照してください。

コマンドラインの構文 -vm=JVM_path+name

コマンドラインの例 -vm=/myjavadir/myjavavm

プロパティ・ファイルの構文 該当なし

プロパティ・ファイルの例 該当なし

デフォルト値 java

Java コンパイラの名前 (-compiler-executable)

SQLJ で特定の Java コンパイラを使用する場合は、-compiler-executable オプションで指定します。それ以外の場合は、Sun 社の JDK の標準の javac が使用されます。

SQLJ では、コンパイラの実行可能ファイル名と一緒にディレクトリ・パスを指定しないと、オペレーティング・システムの PATH 変数に基づいて実行可能ファイルの検索が開始されません。

使用する Java コンパイラの要件は次のとおりです。

- 標準の出力デバイス (UNIX システムの STDOUT など)、または後述の -compiler-output-file オプションで指定するファイルに、エラー情報およびステータス情報を出力できること。
- クラス・ファイルのルート・ディレクトリを決定する SQLJ の -d オプションを認識できること。

- コンパイラ・エラーが発生するたびに、0（ゼロ）以外の終了コードを戻り値としてオペレーティング・システムに戻すこと。
- エラーやメッセージで示す行情報が、次のいずれかの形式であること。（<> 内の項目はオプションです。）

- Sun 社 javac 形式

```
filename.java:line<.column><-line<.column>>
```

```
例: myfile.java:15:Illegal character:'\u01234'
```

- Microsoft jvc 形式

```
filename.java(line,column)
```

```
例: myfile.java(15,7) Illegal character:'\u01234'
```

SQLJ では、コンパイラの行情報が、生成された .java ファイルではなくオリジナルの .sqlj ファイルの行番号を使用するように処理されます。

注意： -encoding オプションがサポートされていないコンパイラの場合は、8-79 ページの「[コンパイラが使用する文字エンコーディング \(-compiler-encoding-flag\)](#)」で説明する -compiler-encoding-flag を無効化してください。

コマンドラインの構文 -compiler-executable=Java_compiler_path+name

コマンドラインの例 -compiler-executable=/myjavadir/myjavac

プロパティ・ファイルの構文 sqlj.compiler-executable=Java_compiler_path+name

プロパティ・ファイルの例 sqlj.compiler-executable=myjavac

デフォルト値 javac

コンパイラが使用する文字エンコーディング (-compiler-encoding-flag)

8-28 ページの「[入力および出力ソース・ファイルの文字エンコーディング \(-encoding\)](#)」で説明したように、-encoding オプションを使用して、SQLJ で使用する文字エンコーディングを指定すると、通常、SQLJ はこの指定を Java コンパイラに渡します。SQLJ がコンパイラに文字エンコーディングの指定を渡さないようにするには、-compiler-encoding-flag を false に設定します（たとえば、javac 以外のコンパイラを使用し、そのコンパイラが -encoding オプションをその名前ではサポートしていないとき、この設定が必要です）。

コマンドラインの構文 `-compiler-encoding-flag<=true|false>`

コマンドラインの例 `-compiler-encoding-flag=false`

プロパティ・ファイルの構文 `sqlj.compiler-encoding-flag<=true|false>`

プロパティ・ファイルの例 `sqlj.compiler-encoding-flag=false`

デフォルト値 `true`

コンパイラの実出力ファイル (-compiler-output-file)

Java コンパイラが結果をファイルに出力するように指定する場合は、`-compiler-output-file` オプションを使用して、SQLJ にファイル名を通知します。この操作を行わない場合、SQLJ では、コンパイラが結果を標準の実出力デバイス（UNIX システムの `STDOUT` など）に出力するものと見なされます。絶対パスまたはカレント・ディレクトリからの相対パスは、必要に応じて指定してください。

注意： `-passes` を有効にする場合は、このオプションを使用できません。この場合は、`STDOUT` に出力されることが必要です。

コマンドラインの構文 `-compiler-output-file=output_file_path+name`

コマンドラインの例 `-compiler-output-file=/myjavadir/mycimploutput`

プロパティ・ファイルの構文 `sqlj.compiler-output-file=output_file_path+name`

プロパティ・ファイルの例 `sqlj.compiler-output-file=/myjavadir/mycimploutput`

デフォルト値 なし（標準の実出力）

コンパイラ・メッセージの実出力パイプ (-compiler-pipe-output-flag)

Sun 社の JDK の `javac` コンパイラでは、デフォルトでエラーとメッセージの実出力が `STDERR` に書き込まれます。SQLJ では、このようなコンパイラ実出力は、確実にとおかれるように `STDOUT` に書き込まれるものと仮定しています。

SQLJ が Java コンパイラを起動する際のデフォルトの動作では、システム・プロパティ `javac.pipe.output` が `true` に設定されます。この設定では、コンパイラの実エラーおよびメッセージの実出力が、`STDOUT` に送られます。SQLJ が Java コンパイラを起動する際に、このシステム・プロパティを設定しないようにするには、`-compiler-pipe-output-flag=false` を指定します。たとえば、使用する Java コンパイラでシステム・プロパティ `javac.pipe.output` がサポートされないとき、この操作を行う必要があります。

このフラグは、プロパティ・ファイル、コマンドライン、環境変数 `SQLJ_OPTIONS` のいずれでも設定できます。

注意：

- 出力を `STDERR` に書き込むようにデフォルト設定されている Sun 社の Java コンパイラでは、`STDOUT` への出力が必要となる `-passes` を有効にするときは、`-compiler-pipe-output-flag` を有効のままにしておく必要があります。
 - Sun 社の JDK 1.3.x バージョンでは `javac.pipe.output` 機能は現在サポートされていない可能性があります。
-

コマンドラインの構文 `-compiler-pipe-output-flag<=true|false>`

コマンドラインの例 `-compiler-pipe-output-flag=false`

プロパティ・ファイルの構文 `sqlj.compiler-pipe-output-flag<=true|false>`

プロパティ・ファイルの例 `sqlj.compiler-pipe-output-flag=false`

デフォルト値 `true`

ソース・ファイル名のチェック (-checkfilename)

ソース・ファイル名は、定義済みの `Public` クラス名と同一にするか、`Public` クラスが未定義の場合は、最初に定義されているクラス名と同一にすることをお勧めします。たとえば、`MyPublicClass.sqlj` ソース・ファイル中には、`Public` クラス `MyPublicClass` が定義してあることが望まれます。

`SQLJ` ソース・ファイル名が、ファイル中の定義済み `Public` クラス名（ある場合）と同じかどうかの検証は、`-checkfilename` フラグで要否を指定します。標準 `javac` のような一部のコンパイラでは、ソース・ファイル名を `Public` クラス名と同じ名前にする必要がありますが、そうでないコンパイラもあります。

コードの移植性を最大限に高めるには、このフラグを有効にしておく必要があります（デフォルトで有効になります）。

注意： サーバー側での変換にはネーミング要件が伴わないため、`-checkfilename` オプションは不要であり、トランスレータによる名前のチェックも実行されません。

コマンドラインの構文 `-checkfilename<=true|false>`

コマンドラインの例 `-checkfilename=false`

プロパティ・ファイルの構文 `sqlj.checkfilename<=true|false>`

プロパティ・ファイルの例 `sqlj.checkfilename=false`

デフォルト値 `true`

SQLJ での 2 段階による実行 (-passes)

デフォルトでは、sqlj スクリプトを実行すると、次の手順で処理されます。

1. sqlj スクリプトによって JVM が起動され、SQLJ トランスレータが実行されます。
2. トランスレータによって、セマンティクス・チェックおよび .sqlj ファイルの変換が行われ、変換された .java ファイルが生成されます。
3. トランスレータによって Java コンパイラが起動され、生成済みの .java ファイルがコンパイルされます。
4. トランスレータによって、コンパイラの出力が処理されます。
5. プロファイル・ファイルが生成されていなければ、トランスレータはプロファイル・カスタマイザを起動し、カスタマイズを実行します。

ただし、JVM とコンパイラの設定によっては、手順 3 でコンパイラが起動されないために変換が一時停止する場合があります。

この場合は、SQLJ が 2 段階で実行され、その間にコンパイル処理が行われるように指定します。これを行うには、2 段階による実行を指定するフラグを有効にする必要があります。

`-passes`

`-passes` オプションは、コマンドラインまたは環境変数 `SQLJ_OPTIONS` で指定します。プロパティ・ファイルでは指定できません。

注意：

- `-passes` を有効にする場合は、コンパイラの出力が `STDOUT` に書き込まれることが必要です。このため、`-compiler-pipe-output-flag` は有効のままにしておきます（デフォルトでは有効になります）。また、`-compiler-output-file` オプションは使用できません。使用すると、出力が `STDOUT` ではなくファイルに書き込まれます。
 - コマンドラインでのみ指定できる他のフラグ（`-help`、`-version`、`-n`）と同様に、`-passes` フラグでは `=true` 構文がサポートされません。
-

`-passes` を有効にした場合、`sqlj` スクリプトを実行すると、次の手順で処理されます。

1. `sqlj` スクリプトによって JVM が起動された後で、SQLJ トランスレータでの 1 段階目の処理が実行されます。
2. トランスレータによって、セマンティクス・チェックおよび `.sqlj` ファイルの変換が行われ、変換された `.java` ファイルが生成されます。
3. JVM を終了します。
4. `sqlj` スクリプトによって、Java コンパイラが起動され、生成済みの `.java` ファイルがこのコンパイラでコンパイルされます。
5. `sqlj` スクリプトによって JVM が再び起動された後で、SQLJ トランスレータでの 2 段階目の処理が実行されます。
6. トランスレータによって、コンパイラの出力が処理されます。
7. プロファイル・ファイルが生成されていなければ、JVM はプロファイル・カスタマイザを起動し、カスタマイズを実行します。

この手順では、JVM が Java コンパイラを起動する際に発生する問題が回避されます。

コマンドラインの構文 `-passes`

コマンドラインの例 `-passes`

プロパティ・ファイルの構文 該当なし

プロパティ・ファイルの例 該当なし

デフォルト値 `off`

カスタマイズのオプション

次の各オプションは、必要に応じて SQLJ プロファイルのカスタマイズを指定します。

- `-default-customizer`
- カスタマイザに直接渡すオプション

注意： デフォルトの Oracle 固有コード生成 (`-codegen=oracle`) を使用する場合は、SQLJ によってプロファイルが生成されないため、カスタマイズが実行されません。その場合、ここで説明したオプションは適用されません。

デフォルト・プロファイル・カスタマイザ (`-default-customizer`)

SQLJ で下記のデフォルト以外のプロファイル・カスタマイザを使用するには、`-default-customizer` を使用します。

`oracle.sqlj.runtime.util.OraCustomizer`

Oracle データベースを使用しない場合は、このオプションを設定します。

このオプションでは、Java クラスの完全修飾名を引数として使用します。

注意： このオプションをオーバーライドするには、SQLJ のコマンドライン（またはプロパティ・ファイル）で `-P-customizer` オプションを指定します。詳細は、8-60 ページの「[プロファイル・カスタマイザに渡すオプション \(-P\)](#)」を参照してください。

コマンドラインの構文 `-default-customizer=customizer_classname`

コマンドラインの例 `-default-customizer=sqlj.myutil.MyCustomizer`

プロパティ・ファイルの構文 `sqlj.default-customizer=customizer_classname`

プロパティ・ファイルの例 `sqlj.default-customizer=sqlj.myutil.MyCustomizer`

デフォルト値 `oracle.sqlj.runtime.util.OraCustomizer`

注意： Oracle データベースと ISO コード生成を使用する場合は、デフォルトの `OraCustomizer` でプロファイルのカスタマイズを行うことをお薦めします。

カスタマイザに直接渡すオプション

JVM やコンパイラと同様に、接頭辞（この場合は -P）を使用すると、オプションをプロファイル・カスタマイザ・ハーネスに直接渡すことが可能になります。詳細は、8-60 ページの「[プロファイル・カスタマイザに渡すオプション \(-P\)](#)」を参照してください。

一般的なカスタマイズ・オプションと Oracle 固有のカスタマイザ・オプションの詳細は、A-10 ページの「[カスタマイズ・オプションとカスタマイザの選択](#)」を参照してください。

トランスレータとランタイムの機能

この章では、Oracle SQLJ のトランスレータおよびランタイムの内部操作と機能について説明します。

この章では、次の項目について説明します。

- [トランスレータの内部操作](#)
- [トランスレータのエラー、メッセージおよび終了コード](#)
- [SQLJ ランタイム](#)
- [トランスレータおよびランタイムでのグローバリゼーション・サポート](#)

トランスレータの内部操作

ここでは、変換時に SQLJ トランスレータによって実行される主な操作について説明します。

- [Java と SQLJ のコード解析と構文チェック](#)
- [SQL のセマンティクス・チェックとオフライン解析](#)
- [コードの生成](#)
- [Java コンパイル](#)
- [プロファイルのカスタマイズ \(ISO コード生成の場合\)](#)

Java と SQLJ のコード解析と構文チェック

SQLJ 変換では、まず SQLJ と Java のパーサーによってすべてのソース・コードが処理され、構文チェックが行われます。

SQLJ トランスレータは、.sqlj ファイルの解析時に、Java パーサーと SQLJ パーサーを起動します。Java パーサーは、Java 文の構文をチェックします。SQLJ パーサーは、SQLJ の構文（先頭に #sql が付くもの）をチェックします。また、SQLJ パーサーは、Java パーサーを起動して、SQLJ 実行文に含まれる Java ホスト変数と式の構文をチェックします。

SQLJ パーサーは、SQLJ 言語の仕様に基づいて SQLJ 構文の文法をチェックします。ただし、埋込み SQL 操作の文法はチェックしません。SQL 構文は、セマンティクス・チェックまたはオフライン解析でチェックされます。

この構文チェックでは、セミコロンの抜け、中カッコの対の抜け、明らかな型の不一致（乗算する値の一方が文字列になっている）などのエラーが検出されます。

このチェックで構文エラーまたは型の不一致が検出されると、変換が中止されてエラーが出力されます。

SQL のセマンティクス・チェックとオフライン解析

SQLJ および Java アプリケーションのソース・コードに構文の誤りがないことが確認されると、次のセマンティクス・チェックに進みます。このチェックでは、SQLJ のオプション設定に基づいて、SQL セマンティクス・チェッカまたは SQL オフライン・パーサー（あるいはその両方）が起動します。

-user オプションの設定によって、オンライン・チェックが有効化されます。-user オプションでパスワードと URL が指定されていない場合は、-password オプションと -url オプションによって、データベース接続が指定されます。-offline オプションまたは -online オプションで、使用するチェッカが指定されます。デフォルト（通常はこの設定で十分です）では、OracleChecker と呼ばれるチェッカ・フロントエンドが指定され、このチェッカによってオンライン・チェックが設定されているかどうか、どの JDBC を使用しているかに応じて適切なチェッカが選択されます。

-parse はオフライン・パーサーを有効にするオプションで、デフォルトは true です。このオフライン・パーサーは、変換時のデータベース接続を必要とせずに、SQL と PL/SQL の構文を検証する手段を提供します（ただし、データベース列に対するデータ型の検証は行いません）。-parse オプションの設定によっては、-user オプションは無視され、オンライン・チェックが無効になります。

オンライン・チェックとオフライン解析の詳細および機能の比較は、8-32 ページの「[接続オプション](#)」および 8-67 ページの「[セマンティクス・チェックとオフライン解析のオプション](#)」を参照してください。

注意： ISO コード生成の場合は、SQLJ トランスレータを前回実行した際に生成されたプロファイルに対しても、セマンティクス・チェックを実行できます。A-38 ページの「[プロファイルのセマンティクス・チェック用の SQLCheckerCustomizer](#)」を参照してください。

セマンティクス・チェックでは、オンライン・チェックまたはオフライン解析のステータスに関係なく、次の 2 つのタスクが常に実行されます。

1. SQLJ によって、SQLJ 実行文で使用されている Java 式の型が解析されます。

変換される SQLJ ソース・ファイルの他に、コマンドラインで指定した .java ファイルと、インポート済みの Java クラスの .class ファイルまたは .java ファイルのうち CLASSPATH で指定されたファイルも解析の対象になります。解析では、SELECT 文または CAST 文でのストリーム型が使用されているかどうか、使用されている場合はその使用方法がチェックされます。また、イテレータ列または INTO リストで使用されている Java 型、入力ホスト変数として使用されている Java 型、出力ホスト変数として使用されている Java 型もチェックされます。

また、SQLJ では、FETCH、CAST、CALL、SET TRANSACTION、VALUES および SET の各文が構文に基づいて処理されます。

SQLJ 実行文内の Java 式には、それぞれの状況と用途に適した Java 型を指定する必要があります。たとえば、次のような文があるとします。

```
#sql [myCtx] { UPDATE ... };
```

myCtx は、この文に対する接続コンテキスト・インスタンスまたは実行コンテキスト・インスタンスを指定する変数です。この変数は、実際に SQLJ の接続コンテキスト型または実行コンテキスト型に解決できることが必要です。

ここでは、次の例について考えてみます。

```
#sql { UPDATE emp SET sal = :newSal };
```

`newSal` がフィールドではなく変数の場合、`newSal` があらかじめ宣言されていないときは、エラーが生成されます。また、この変数を有効な Java 型に代入できない場合や、この変数の Java 型を SQL 文 (`java.util.Vector` など) で使用できない場合も、エラーが生成されます。

注意： Java 型のセマンティクス・チェックは、SQLJ 実行文内の Java 式に対してのみ行われます。標準の Java 文に含まれるエラーは、Java コンパイラでのコンパイル時に検出されます。

2. SQLJ で、埋込み SQL 操作の分類が試みられます。このため、操作の種類を識別するために、操作ごとに `SELECT` や `INSERT` などの認識可能なキーワードが必要です。たとえば、次の文ではエラーが生成されます。

```
#sql { foo };
```

オンライン・チェックまたはオフライン解析（あるいはその両方）が有効な場合は、次のタスクが実行されます。

3. SQLJ によって、SQL と PL/SQL の埋込み操作の構文が分析および検証されます。

オンライン・チェックのみが有効な場合は、次のタスクが実行されます。

4. SQLJ によって、SQLJ 実行文で使用されている Java 式の型が、1) データベースの列の SQL 型 2) ストアド・プロシージャおよびストアド・ファンクションの引数と戻り値の SQL 型に対してチェックされます。

この時点で、SQLJ によって SQLJ 実行文で使用されている SQL エンティティ（表、ビュー、ストアド・プロシージャなど）が実際にデータベースに存在するかどうかを検証されます。また、データベース列からデータを選択して、Java 基本型のイテレータ列に取り込む場合に、そのデータベース列で `NULL` 値が許容されているかどうかもチェックされます。基本型のイテレータ列では、`NULL` データを処理できないためです。ただし、ストアド・プロシージャやストアド・ファンクションの出力パラメータと戻り値に対しては、`NULL` 値が許容されているかどうかはチェックされません。

コードの生成

ISO 標準 SQLJ コード生成の場合、SQLJ トランスレータは、1 つの .sqlj アプリケーション・ソース・ファイルに対して、1 つの .java ファイルと、1 つ以上のプロファイル (.ser ファイルまたは .class ファイル) を生成します。.java ファイルには、変換済みのアプリケーション・ソース・コード、宣言した **private** イテレータと接続コンテキスト用のクラス定義、および、ISO コードの場合は、SQLJ によって生成されて内部的に使用されるプロファイルキーのクラス定義が含まれています。

注意： デフォルトの Oracle 固有コード生成モード (-codegen=oracle) を使用している場合、プロファイルやプロファイルキー・クラスは生成されません。4-36 ページの「[Oracle 固有コード生成 \(プロファイルなし\)](#)」を参照してください。

ISO コード生成では、SQLJ 実行文をコードで使用しない場合、プロファイルやプロファイルキー・クラスはありません。

.java ファイル内に生成されたアプリケーション・コード

デフォルトの Oracle 固有コード生成の場合、アプリケーション用に生成された .java ファイルには、元の SQLJ 実行文のかわりに Oracle JDBC への直接コールが含まれています。Oracle 固有の SQLJ ランタイムへのコールもあります。

ISO 標準 SQLJ コード生成の場合、SQLJ 実行文は、SQLJ ランタイムへのコールに置き換えられます。この SQLJ ランタイムには、JDBC ドライバへのコールが含まれています。

また、生成された .java ファイルには、記述した #sql 文全体が参照用に含まれており、それぞれコメントが付けられています。

生成された .java ファイルには、入力した .sqlj ファイルと同じベース名が使用されます。通常、これは、.sqlj ファイルに定義されている Public クラス名と同じになります。Public クラスが存在しない場合は、最初に定義されているクラス名になります。たとえば、クラス Foo が定義されている場合、その定義ファイル名は Foo.sqlj となり、トランスレータで生成されるソース・ファイル名は Foo.java となります。

生成された .java ファイルの格納場所は、SQLJ の -dir オプション設定の有無および設定方法によって決まります。デフォルトでは、生成された .java ファイルは、.sqlj 入力ファイルと同じディレクトリに格納されます。詳細は、8-31 ページの「[.java ファイルの出力ディレクトリ \(-dir\)](#)」を参照してください。

.java ファイル内に生成されるプロファイルキー・クラス (ISO コード生成の場合)

ISO 標準 SQLJ コード生成を使用している場合、SQLJ ではプロファイルキー・クラスが生成されます。このクラスは実行時に内部的に使用され、シリアル化されたプロファイルのロードやアクセスが実行されます。このクラスには、変換済みアプリケーション内の SQLJ ランタイム・コールと、シリアル化プロファイル内に挿入される SQL 操作間のマッピング情報が含まれています。また、シリアル化プロファイルにアクセスするためのメソッドも含まれています。

注意： デフォルトの Oracle 固有コード生成を使用している場合、プロファイルやプロファイルキー・クラスは生成されません。

このプロファイルキー・クラスは、変換されたアプリケーション・ソース・コードが出力された .java ファイル内に定義され、次のように、.sqlj ソース・ファイルのベース名を使用した名前が付けられます。

Baseline_SJProfileKeys

たとえば、Foo.sqlj を変換すると、生成された .java ファイルには、次のプロファイルキー・クラスが定義されます。

Foo_SJProfileKeys

これは、パッケージに含まれるアプリケーションの場合も同じです。たとえば、a.b というパッケージに含まれる Foo.sqlj を変換すると、次のクラスが定義されます。

a.b.Foo_SJProfileKeys

.ser ファイルまたは .class ファイル内に生成されるプロファイル (ISO コード生成の場合)

ISO 標準 SQLJ コード生成を使用する場合、SQLJ では、入力ファイル内の SQL 操作に関する情報を格納するプロファイルが生成されます。プロファイルは、アプリケーションで使用する接続コンテキスト・クラスごとに 1 つ生成されます。各プロファイルには、関連付けする接続コンテキスト・クラスのインスタンスを使用して実行される操作が記述されています。つまり、実行する SQL 操作、アクセスする表、コールするストアド・プロシージャとストアド・ファンクションなどが記述されています。

注意： デフォルトの Oracle 固有コード生成を使用する場合、SQL 操作に関する情報は Oracle JDBC を直接コールする生成コードに埋め込まれます。この場合、SQLJ ではプロファイルが生成されません。

プロファイルは、`.ser` シリアル化リソース・ファイル内に生成されます。ただし、SQLJ の `-ser2class` オプションが有効になっている場合は、変換時に `.class` ファイルに自動的に変換されます。この場合、これ以降のプロファイルのカスタマイズはできなくなります。(`.class` ファイルを削除し、SQLJ トランスレータを再度実行してプロファイルを再作成する必要があります。)

プロファイルのベース名は、プロファイルキー・クラス名の場合と同じように生成されます。このベース名には、パッケージ名の後に `.sqlj` ファイルのベース名が続き、最後に次の文字列が付きます。

```
_SJProfilen
```

`n` は 0 (ゼロ) から始まる一意の番号で、特定の `.sqlj` 入力ファイルに対して生成された各プロファイルに付けられます。

再び、`Foo.sqlj` という入力ファイルを例として使用します。この入力ファイルがパッケージに含まれていない場合、生成される 2 つのプロファイルのベース名は、次のようになります。

```
Foo_SJProfile0  
Foo_SJProfile1
```

`Foo.sqlj` が `a.b` というパッケージに含まれる場合、プロファイルのベース名は、次のようになります。

```
a.b.Foo_SJProfile0  
a.b.Foo_SJProfile1
```

実際には、プロファイルは、リソース・ファイル内のシリアル化された Java オブジェクトとして存在します。プロファイルを含むリソース・ファイルには、プロファイルのベース名 (パッケージ名は除く) に従って名前が付けられ、拡張子として `.ser` が付きます。前述の例の 2 つのプロファイルを含むリソース・ファイルには、それぞれ次のように名前が付けられます。

```
Foo_SJProfile0.ser  
Foo_SJProfile1.ser
```

ただし、`-ser2class` オプションが有効になっている場合は、`Foo_SJProfile0.class` および `Foo_SJProfile1.class` となります。このオプションが選択されていると、後述のカスタマイズの後に、`.class` への変換が行われます。詳細は、8-64 ページの「[.ser ファイルから .class ファイルへの変換 \(-ser2class\)](#)」を参照してください。

これらのファイルは、SQLJ の `-d` オプションで設定されているディレクトリに置かれます。生成されたすべての `.ser` ファイルおよび `.class` ファイルの格納先は、このオプションで指定されます。詳細は、8-29 ページの「[.ser および .class ファイルの出力ディレクトリ \(-d\)](#)」を参照してください。

プロファイルは、SQLJ プロセスの次段階で、使用するデータベースにあわせてカスタマイズされます。9-10 ページの「[プロファイルのカスタマイズ \(ISO コード生成の場合\)](#)」を参照してください。

SQLJ ランタイムに対して生成されたコールについて

#sql 文が Oracle JDBC (Oracle 固有コード生成の場合) または SQLJ ランタイム (ISO 標準 SQLJ コード生成の場合) へのコールに置き換えられている場合、これらのコールは次の動作を行います。

表 9-1 生成されたコールの動作、ISO 標準と Oracle 固有の比較

ISO 標準コード生成の場合	Oracle 固有コード生成の場合
関連付けされているプロファイル・エントリ中に格納されている情報を使用して、SQLJ 文オブジェクトを取得します。	Oracle JDBC 文オブジェクトを取得します。
SQLJ 文オブジェクトの <code>setXXX()</code> メソッドを使用して、入力パラメータを文にバインドします。	Oracle JDBC 文メソッドを使用して入力をバインドして、必要な場合は、出力パラメータを登録します。
文オブジェクトの <code>executeUpdate()</code> メソッドまたは <code>executeQuery()</code> メソッドを使用して、文を実行します。	Oracle 文を実行します。
可能な場合は、イテレータ・インスタンスを生成します。	可能な場合は、イテレータ・インスタンスを生成します。
SQLJ 文オブジェクトの <code>getXXX()</code> メソッドを使用して、文から出力パラメータを取り出します。	Oracle JDBC 取得メソッドを使用して、文から出力を取り出します。
SQLJ 文オブジェクトを終了します (デフォルトで、SQLJ 文キャッシュによって再使用されます)。	Oracle JDBC 文オブジェクトを終了します (デフォルトで、JDBC 文キャッシュによって再使用されます)。

SQLJ ランタイムでは、JDBC 文オブジェクトに似た SQLJ 文オブジェクトが使用されます。ただし、SQLJ の実装状態によっては、JDBC 文クラスを直接使用できる場合とできない場合があります。SQLJ 文クラスを使用すると、SQLJ 固有の機能を利用できます。次にその例を示します。

- 標準 SQLJ 文オブジェクトは、データベースから取り出した NULL 値が、NULL 値を使用できない `int` や `float` などの Java 基本型に出力されると、SQL 例外を生成します。
- Oracle SQLJ 文オブジェクトでは、ユーザー定義のオブジェクト型やコレクション型を Oracle データベースに渡したり、Oracle データベースから取り出したりできます。

Java コンパイル

コードが生成されると、SQLJ では Java コンパイラが起動され、生成済 .java ファイルがコンパイルされます。これにより、イテレータ宣言や接続コンテキスト宣言を含み、アプリケーション内で定義されているクラスごとに、.class ファイルが生成されます。また、ISO コード生成を使用し、アプリケーションでは SQLJ 実行文を使用している場合は、生成されたプロファイルキー・クラスに対しても、.class ファイルが生成されます。SQLJ コマンドラインで型解決などのために直接指定した .java ファイルも、この時点でコンパイルされます。

9-5 ページの「[コードの生成](#)」で示した例では、ソース・コードにパッケージ情報が指定されている場合、次の .class ファイルが適切なディレクトリに生成されます。

- Foo.class
- Foo_SJProfileKeys.class (ISO コード生成のみ)
- Foo.sqlj に定義した追加クラスに対する .class ファイル
- Foo.sqlj で宣言した各イテレータ・クラスおよび接続コンテキスト・クラス (public および private) に対する .class ファイル

.class ファイルとプロファイル（ある場合は、.ser または .class のいずれか）が、同じディレクトリに格納されるようにするために、SQLJ で設定された -d オプションが Java コンパイラに渡されます。-d オプションが設定されていない場合、.class ファイルとプロファイルは、生成された .java ファイルと同じディレクトリ（-dir オプションで指定されたディレクトリ）に格納されます。

また、SQLJ と Java コンパイラでは、同じ文字エンコーディングが使用されます。そのため、SQLJ で設定された -encoding オプションが Java コンパイラに渡されます。ただし、SQLJ の -compiler-encoding-flag がオフになっている場合を除きます。-encoding オプションが設定されていない場合、SQLJ とコンパイラでは、JVM の file.encoding プロパティの設定が使用されます。

デフォルトでは、標準の Sun 社の JDK の javac コンパイラが使用されますが、他のコンパイラを使用することも可能です。他の Java コンパイラを使用するには、SQLJ の -compiler-executable オプションを設定します。

注意： SQLJ の -encoding オプションを使用しているが、コンパイラに -encoding オプションがない場合は、SQLJ の -compiler-encoding-flag を無効にします。無効にしないと、SQLJ はコンパイラに -encoding オプションを渡そうとします。

コンパイラのオプションとコンパイラに関連する SQLJ オプションに対する、SQLJ のサポートに関する情報は、次の各項を参照してください。

- 8-10 ページ「[javac のオプションのサポート](#)」
- 8-29 ページ「[.ser および .class ファイルの出力ディレクトリ \(-d\)](#)」
- 8-28 ページ「[入力および出力ソース・ファイルの文字エンコーディング \(-encoding\)](#)」
- 8-59 ページ「[Java コンパイラに渡すオプション \(-C\)](#)」
- 8-62 ページ「[コンパイル・フラグ \(-compile\)](#)」
- 8-79 ページ「[コンパイラが使用する文字エンコーディング \(-compiler-encoding-flag\)](#)」
- 8-78 ページ「[Java コンパイラの名前 \(-compiler-executable\)](#)」
- 8-80 ページ「[コンパイラの出力ファイル \(-compiler-output-file\)](#)」
- 8-80 ページ「[コンパイラ・メッセージの出力パイプ \(-compiler-pipe-output-flag\)](#)」

プロファイルのカスタマイズ (ISO コード生成の場合)

ISO 標準コード生成 (-codegen=iso) を使用している場合は、Java コンパイルが終了すると、生成されたプロファイルのカスタマイズが行われます。プロファイルには、埋込みの SQL 命令に関する情報が含まれています。カスタマイズを行うと、使用しているデータベースでのアプリケーションの処理効率が向上し、ベンダー固有の拡張機能を利用できるようになります。

注意： デフォルトの Oracle 固有コード生成 (-codegen=oracle) を使用する場合、SQLJ ではプロファイルが生成されず、カスタマイズも行われません。コードでは、Oracle JDBC API を直接コールすることによって、Oracle 固有の機能をサポートします。詳細は、4-36 ページの「[Oracle 固有コード生成 \(プロファイルなし\)](#)」を参照してください。

カスタマイズ時に、SQLJ によってカスタマイザ・ハーネスと呼ばれるフロントエンドが起動されます。これは、コマンドライン・ユーティリティとして機能する Java クラスの 1 つです。このカスタマイザ・ハーネスは、デフォルトの Oracle カスタマイザ、または SQLJ オプション設定を介して指定したカスタマイザを起動します。

プロファイルのカスタマイズには、次の 2 つの目的があります。

- 可能な場合は、ベンダー固有のデータベース型や機能をアプリケーションで使用できるようにします。
- データベース環境にあわせて、アプリケーションが最も効率的に機能するプロファイルを生成します。

カスタマイズを行わなかった場合、アクセスして使用できるのは標準の JDBC 型のみです。

たとえば、Oracle カスタマイザを使用すると、Oracle9i で独自に定義した PERSON 型をサポートするように、プロファイルを更新できます。これにより、サポートされている他のデータ型と同じように PERSON 型も使用できるようになります。

oracle.sql の拡張型を使用する場合も、Oracle カスタマイザを使用してカスタマイズする必要があります。

注意： プロファイルのカスタマイズについて、次の点に注意してください。

- Oracle SQLJ ランタイムと Oracle JDBC ドライバは、変換時に Oracle カスタマイザを使用するたびにアプリケーションで必要となるものであり、Oracle 拡張型をコード中に使用しない場合にもこのことが当てはまります。
 - アプリケーションでカスタマイズ（接続のための処理）を行わない場合には、汎用 SQLJ ランタイムが使用されます。
 - .ser ファイル、または .ser ファイルを含む .jar ファイルをコマンドラインで指定すると、以前に生成されたプロファイルのカスタマイズもできます。ただし、これは、変換用の SQLJ 実行とは別の実行で行う必要があります。指定できるのは、.ser ファイルや .jar ファイルのカスタマイズ、または .sqlj ファイルや .java ファイルの変換、コンパイルおよびカスタマイズのいずれかです。両方は指定できません。.jar ファイルの使用の詳細は、A-36 ページの「[プロファイルの JAR ファイル](#)」を参照してください。
-
-

プロファイルのカスタマイズの詳細は、[付録 A「カスタマイズおよび専用カスタマイザ」](#)を参照してください。

プロファイルのカスタマイズに関連する SQLJ オプションの詳細は、次の各項を参照してください。

- 8-84 ページ「[デフォルト・プロファイル・カスタマイザ \(-default-customizer\)](#)」
- 8-60 ページ「[プロファイル・カスタマイザに渡すオプション \(-P\)](#)」
- 8-63 ページ「[プロファイル・カスタマイズ・フラグ \(-profile\)](#)」
- A-10 ページ「[カスタマイズ・オプションとカスタマイザの選択](#)」

トランスレータのエラー、メッセージおよび終了コード

ここでは、SQLJ トランスレータのメッセージと終了コードの概要を示します。

トランスレータのエラー、警告および情報メッセージ

変換時に出力される SQLJ メッセージは、エラー、警告および情報の 3 レベルに区分されています。警告メッセージは、さらに非表示にできない警告と非表示にできる警告の 2 つのレベルに区分されています。次は、メッセージの 4 つのレベル（重要度の順）です。

1. エラー
2. 非表示にできない警告
3. 非表示にできる警告
4. 情報

非表示にできる警告と情報は、SQLJ の `-warn` オプションを使用して制御します。後述を参照してください。

エラー・メッセージは「Error:」で開始され、次のいずれかの状態が発生した場合に表示されます。

- コンパイルを実行できない（たとえば、ファイルのベース名とは別の名前の `Public` クラスがソース・ファイルに含まれている）場合。
- コードを実行すると、実行時エラーが発生する（たとえば、Oracle JDBC ドライバを使用して、`VARCHAR` を `java.util.Vector` にフェッチしようとするコードが含まれている）場合。

SQLJ 変換時にエラーが発生すると、出力（.java ファイルやプロファイル）は生成されず、コンパイルとカスタマイズは実行されません。

非表示にできない警告メッセージは「Warning:」で開始され、次のいずれかの状態が発生した場合に表示されます。

- コードを実行したときに、実行時エラーが発生する可能性のある（たとえば、`SELECT` 文の出力先が指定されていない）場合。
- SQLJ の機能で、ソース・コードの実行時の状況が検証できない（たとえば、オンライン・チェック用に指定したデータベースに接続できない）場合。
- コーディングの誤りや見落としが原因でエラーが発生した場合。

非表示にできない警告が発生した場合でも、SQLJ 変換は最後まで実行されます。ただし、アプリケーションを実行する前に問題を分析して、修復の必要があるかどうかを確認してください。オンライン・チェックが指定されている場合に、オンライン・チェックが最後まで実行されないと、オフライン・チェックが実行されます。

注意： 処理上の理由で、SQLJ トランスレータによる SQL 操作の解析には、実行時に使用される SQL パーサーよりも精度の低い解析プログラムが使用されます。このため、変換時にエラーが検出されても、実際にはアプリケーションの実行時には問題にならない場合があります。そのようなエラーは、致命的なエラーと区別して、非表示にできない警告として出力されます。

非表示にできる警告も「Warning:」で開始され、移植性などアプリケーション固有の問題に関する情報が表示されます。たとえば、`oracle.sql.NUMBER` などの Oracle 固有の型を使用して Oracle9i に対して読み込みと書き込みを行った場合に、警告が出力されます。

情報メッセージや状態メッセージは Info: で始まりますが、エラーの状態に関する情報は出力されません。これらのエラーには、変換時に発生した事柄に関する追加情報が表示されます。

非表示にできる警告メッセージやステータス・メッセージを非表示にするには、`-warn` オプションの次のフラグを使用します。

- `cast/nocast` — `nocast` を設定すると、オブジェクト型インスタンスをサブタイプのインスタンスにキャストしようとするときに発生する実行時エラーに関する警告が非表示になります。
- `precision/noprecision` — `noprecision` に設定すると、変換時にデータの精度が低下することを示す警告が非表示になります。
- `nulls/nonulls` — `nonulls` に設定すると、NULL 値が許されている列や型が原因で発生する実行時エラーに関する警告が非表示になります。
- `portable/noportable` — `noportable` に設定すると、Oracle 固有または非標準の機能を使用しているために、他の環境に移植できない SQLJ コードに関する警告が非表示になります。
- `strict/nostRICT` — `nostRICT` に設定すると、名前指定イテレータの列数が、そこに取り込むために選択したデータの列数よりも少ない場合に出力される警告が非表示になります。
- `verbose/noverbose` — `noverbose` に設定すると、エラーや警告の状態ではなく情報のみを表示するステータス・メッセージが非表示になります。

`-warn` オプションとそのフラグの設定方法の詳細は、8-44 ページの「[トランスレータからの警告 \(-warn\)](#)」を参照してください。

SQLJ 変換時に警告が表示された場合は、`-warn=none` オプションを設定して、トランスレータを再実行し、その警告が重要度の高いもの（非表示にできない警告）であるかどうかを確認します。

注意： 各エラー・メッセージ、警告メッセージおよび情報メッセージの詳細は、B-2 ページの「SQLJ 変換時のメッセージ」および B-44 ページの「SQLJ ランタイム・メッセージ」を参照してください。

表 9-2 は、SQLJ トランスレータで生成されるエラー・メッセージおよびステータス・メッセージの分類です。

表 9-2 トランスレータのエラー・メッセージの分類

メッセージの分類	接頭辞	内容	非表示に設定する方法
エラー	Error:	コンパイル時または実行時に発生する障害の原因となる致命的なエラー。(変換は異常終了します)	該当なし
非表示にできない警告	Warning:	変換が正常に実行されない原因、または実行時に発生する障害の原因となる状態(変換は最後まで実行されます。)	該当なし
非表示にできる警告	Warning:	アプリケーション固有の問題(変換は最後まで実行されます。)	-warn オプション・フラグ: nocast noprecision nonnulls noportable nostrict
情報 / ステータス・メッセージ	Info:	変換プロセスに関する情報	-warn オプション・フラグ: noverbose

トランスレータのステータス・メッセージ

SQLJ では、エラー、警告、情報の各メッセージ以外にも、ステータス・メッセージが表示されます。これらのメッセージは、変換、コンパイル、カスタマイズの全フェーズで生成されます。ステータス・メッセージは、SQLJ 操作の各段階で、各ファイルの処理時に出力されます。

SQLJ の -status オプションを使用して、ステータス・メッセージを制御できます。このオプションは、8-47 ページの「リアルタイムのステータス・メッセージ (-status)」を参照してください。

トランスレータの終了コード

SQLJ トランスレータでは、変換終了時に次の終了コードがオペレーティング・システムに戻されます。

- 0 = 実行時にエラーなし
- 1 = SQLJ 実行時にエラー
- 2 = Java コンパイル時にエラー
- 3 = プロファイルのカスタマイズ時にエラー
- 4 = クラスのインストールメントのエラー（.sqlj ソース・ファイルから、生成される .class ファイルへの行番号のマッピング（任意）時のエラー）
- 5 = ser2class 変換時のエラー（.ser ファイルから .class ファイルへのプロファイル・ファイルの変換（任意）時のエラー）

注意：

- -help オプションまたは -version オプションが指定されている場合は、SQLJ 終了コードは 0（ゼロ）です。
 - 処理対象のファイルを指定せずに SQLJ を実行すると、SQLJ によってヘルプ情報が出力され、終了コード 1 が戻されます。
-
-

SQLJ ランタイム

ここでは、Oracle SQLJ ランタイムについて説明します。SQLJ ランタイムは、Pure Java コードのシン・レイヤーで、JDBC ドライバよりも上のレイヤーで実行されます。

デフォルトの Oracle 固有コード生成（-codegen=oracle）を使用する場合は、Oracle JDBC ドライバとともに使用されるランタイム・サブセットによって、SQLJ ランタイム・レイヤーがより軽量になります。大部分のランタイム機能は、Oracle JDBC コールに直接コンパイルされます。Oracle 以外の JDBC ドライバは使用できません。4-36 ページの「[Oracle 固有コード生成（プロファイルなし）](#)」を参照してください。

ISO 標準コード生成（-codegen=iso）を使用して、Oracle SQLJ トランスレータで SQLJ ソース・コードを変換すると、Java アプリケーション内の埋込み SQL コマンドは SQLJ ランタイムへのコールに置き換えられます。ランタイム・クラスは、それに相当する JDBC クラスのラッパーとして機能し、SQLJ の特別な機能を提供します。エンド・ユーザーがアプリケーションを実行すると、プロファイルから SQL 操作に関する情報を読み込んで JDBC ドライバに命令を渡します。

ただし、一般的に SQLJ ランタイムは、あらゆる JDBC ドライバおよびベンダー固有のデータベース・アクセス手段に対応しています。Oracle SQLJ ランタイムには JDBC ドライバが必要ですが、どの標準 JDBC ドライバでも使用できます。ただし、Oracle 固有のデータ型や機能を使用する場合は、必ず Oracle JDBC ドライバを使用してください。このマニュアルで

は、Oracle データベースと Oracle JDBC ドライバの 1 つを使用していることを前提としています。

注意： ISO 標準 SQLJ コード生成の場合、Oracle SQLJ ランタイムと Oracle JDBC ドライバは、変換時に Oracle カスタマイザを使用するたびにアプリケーションで必要となるものであり、Oracle 拡張型をコード中に使用しない場合にもこのことが当てはまります。アプリケーションでカスタマイズ（接続のための処理）を行わない場合には、汎用 SQLJ ランタイムが使用されます。

SQLJ ランタイム・パッケージ

Oracle SQLJ ランタイムのパッケージは、インポートして直接使用できるものもあれば、間接的にしか使用できないものもあります。

注意： これらのパッケージはランタイム・ライブラリである `runtime12`、`runtime12ee`、`runtime11`、`runtime` および `runtime-nonoracle` に含まれています。

直接的に使用可能なパッケージ

ここでは、アプリケーションに直接インポートして使用できるパッケージを取り上げます。名前が `oracle` で始まるパッケージには、Oracle 固有の SQLJ 機能が実装されています。

- `sqlj.runtime`

このパッケージには、`ExecutionContext` クラス、`ConnectionContext` インタフェース、`ConnectionContextFactory` インタフェース、`ResultSetIterator` インタフェース、`ScrollableResultSetIterator` インタフェースが含まれ、さらに、ストリーム（`BinaryStream` や `CharacterStream`、および現在は使用されなくなった `AsciiStream` や `UnicodeStream`）に対するラッパー・クラスも含まれています。

このパッケージに含まれるインタフェースおよび抽象クラスは、`sqlj.runtime.ref` パッケージや `oracle.sqlj.runtime` パッケージ内のクラス、または SQLJ トランスレータによって生成されるクラスによって実装されます。

- `sqlj.runtime.ref`

このパッケージのクラスには、`sqlj.runtime` パッケージのインタフェースと抽象クラスが実装されます。`sqlj.runtime.ref.DefaultContext` クラスを使用して、デフォルト接続の指定し、デフォルト接続コンテキスト・インスタンスを作成できます。このパッケージに含まれるその他のクラスは、コード生成時のクラス定義の際に SQLJ 内部で使用されます。たとえば、SQLJ コード中でイテレータ・クラスや接続コンテキスト・クラスを宣言するときに使用されます。

- `oracle.sqlj.runtime`

このパッケージには、`DefaultContext` クラスのインスタンス化やデフォルト接続の確立に使用する `Oracle` クラスが含まれています。また、**Oracle SQLJ** で使用される **Oracle** 固有のランタイム・クラスもこのパッケージに含まれ、**Oracle** の拡張型と他の型との相互変換などが可能となっています。

間接的に使用可能なパッケージ

ここでは、SQLJ で内部的に使用されるクラスを含んだパッケージを取り上げます。

- `sqlj.runtime.profile`

このパッケージには、SQLJ プロファイルを定義するインタフェースと抽象クラスが含まれています（適用の対象は ISO 標準コード生成のみ）。たとえば、`EntryInfo` クラスや `TypeInfo` クラスなどが含まれています。プロファイル内の各エントリには、`EntryInfo` オブジェクトを使用して説明が付加されます。（このオブジェクトでは、プロファイルのエントリと、アプリケーション内の SQL 操作が対応付けられています。）プロファイルのエントリ内の各パラメータには、`TypeInfo` オブジェクトによって説明が付加されます。

このパッケージのインタフェースおよびクラスは、`sqlj.runtime.profile.ref` パッケージ内のクラスによって実装されます。

- `sqlj.runtime.profile.ref`

このパッケージには、`sqlj.runtime.profile` パッケージのインタフェースと抽象クラスを実装するクラスが含まれています。これらのクラスは、プロファイルの定義時に SQLJ 内部で使用されます（ISO 標準コード生成の場合のみ）。また、デフォルトの JDBC ベースのランタイムの実現にも使用されます。

- `sqlj.runtime.error`

このパッケージは、SQLJ 内部で使用されます。このパッケージには、SQLJ トランスレータによって生成される一般的な（Oracle 固有ではない）エラー・メッセージのリソース・ファイルが含まれています。

- `oracle.sqlj.runtime.error`

このパッケージは、SQLJ 内部で使用されます。このパッケージには、SQLJ トランスレータによって生成される Oracle 固有のエラー・メッセージのリソース・ファイルが含まれています。

ランタイム・エラーの分類

ランタイム・エラーは、次のいずれかによって生成されます。

- SQLJ ランタイム
- JDBC ドライバ
- RDBMS

いずれの場合も、SQL 例外が、`java.sql.SQLException` クラスまたは `sqlj.runtime.SQLNullException` などのサブクラスのインスタンスとして生成されます。

エラーの発生場所によっては、有用な情報を取得できることがあります。その場合、`getSQLState()`、`getErrorCode()` および `getMessage()` の各メソッドを使用します。たとえば、SQLJ エラーに関しては、SQL の状態とメッセージが提供されます。詳細は、4-22 ページの「[SQL の状態およびエラー・コードの取出し](#)」を参照してください。

実行時に Oracle JDBC ドライバまたは RDBMS でエラーが発生した場合は、メッセージの接頭辞を確認して、次の各マニュアルを参照してください。

- 『Oracle9i JDBC 開発者ガイドおよびリファレンス』（JDBC エラーの場合）
- 『Oracle9i データベース・エラー・メッセージ』（RDBMS エラーの場合）

SQLJ ランタイム・エラーのリストは、B-44 ページの「[SQLJ ランタイム・メッセージ](#)」を参照してください。

トランスレータおよびランタイムでのグローバリゼーション・サポート

Oracle SQLJ では、グローバリゼーション・サポート（各国語サポートまたは NLS と呼ばれる）のために Java の組込み機能を使用します。ここでは、次の内容について説明します。

- SQLJ におけるグローバリゼーション・サポートとネイティブ文字エンコーディングの基本事項。文字エンコーディングおよび言語サポートが、Oracle SQLJ でどのように実装されているかを説明します。
- Oracle SQLJ コマンドラインで Oracle のグローバリゼーション・サポート構成の調整に使用できるオプション。
- 拡張 Oracle SQLJ グローバリゼーション・サポート。
- グローバリゼーション・サポートのための Oracle SQLJ 外部での関連操作。

ここでは、Oracle のグローバリゼーション・サポートについての知識があることを前提としています。特に、文字エンコーディングおよびロケールについての知識が必要です。詳細は、『Oracle9i 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。

文字エンコーディングと言語サポート

SQLJ のグローバリゼーション・サポートは、次の 2 つに大別できます。

- 文字エンコーディング

文字エンコーディングは、3 つに分類できます。

- SQLJ 変換時のソース・ファイルの読取りおよび生成のための文字エンコーディング。
- SQLJ 変換時のエラーおよびステータス・メッセージ生成のための文字エンコーディング。
- アプリケーション実行時のエラーおよびステータス・メッセージ生成のための文字エンコーディング。

- 言語サポート

SQLJ 変換時または SQLJ 実行時に、SQLJ からエラー・メッセージおよびステータス・メッセージが出力される場合に、どの言語のメッセージ・リストを使用するのかを指定します。

実行時のグローバリゼーション・サポートの機能は、データベース・キャラクタ・セットにある文字のみを SQLJ ソース・コードおよび SQL 文字データに使用していることを想定しており、ユーザーは特に意識する必要がありません。SQL 文字データと Unicode とのマッピングは、透過的に行われます。

多言語対応のアプリケーションでは、次のいずれかを使用することをお勧めします。

- キャラクタ・セットで Unicode をサポートしているデータベースの使用。

または

- Unicode をサポートしていないデータベース・キャラクタ・セットを使用している場合でも、各国語キャラクタ・セットで Unicode を指定できます。（詳細は、『Oracle9i 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。）この場合は通常、9-25 ページの「Oracle SQLJ 拡張グローバリゼーション・サポート」で説明されているように SQLJ Unicode 文字型を使用できます。

注意：

- SQLJ トランスレータでは、Unicode 2.0 および Java Unicode のエスケープ・シーケンスが完全にサポートされています。これに対し SQLJ コマンドライン・ユーティリティでは、Unicode エスケープ・シーケンスがサポートされていないため、オペレーティング・システムでサポートされているネイティブの文字しか使用できません。コマンドライン・オプションに Unicode エスケープ・シーケンスの入力が必要な場合は、かわりに SQLJ プロパティ・ファイルに入力してください。SQLJ プロパティ・ファイルでは Unicode エスケープ・シーケンスがサポートされているためです。
 - 埋込み SQL 操作で使用する文字と、データベースから読み書きする文字のエンコーディングおよび変換は、JDBC で直接処理されます。SQLJ は、これらの処理に介入しません。ただし、変換時にオンラインのセマンティクス・チェックが有効になっていると、データベース・キャラクタ・セットと互換性のない文字が SQL DML 操作のテキストに含まれている場合は、警告メッセージが表示されます。
 - JDBC のグローバリゼーション・サポート機能の詳細は、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。
-
-

文字エンコーディングの概要

Oracle SQLJ では、ソース・ファイルの文字エンコーディング設定に従って、次の 2 つが決定されます。

- .sqlj および .java の入力ファイル内のソース・コードの記述方法。これらのファイルは、SQLJ トランスレータが解読する必要があります。
- SQLJ によって生成された .java 出力ファイル内のソース・コードの表記方法。

SQLJ では、デフォルトで JVM `file.encoding` プロパティで指定されたエンコーディングを使用します。ソース・ファイルに他のエンコーディングを使用している場合は、適切に変換を行うために SQLJ にこのエンコーディングを指定する必要があります。

その場合、SQLJ の `-encoding` オプションを使用します。この `-encoding` オプションの設定は、SQLJ からコンパイラに渡され、.java ファイルの読み込み時に使用されます。ただし、SQLJ の `-compiler-encoding-flag` がオフになっている場合を除きます。

重要： ソース・ファイルの文字エンコーディングを指定するときに、`file.encoding` システム・プロパティを変更しないでください。プラットフォームやオペレーティング・システム的环境によっては、Java 操作の他の機能に影響し、一部の文字エンコーディングしか利用できなくなる場合があります。

変換時やエンド・ユーザーによるアプリケーション実行時の SQLJ メッセージ出力にも、このシステム文字エンコーディング設定が使用されます。この文字エンコーディングは、SQLJ の `-encoding` オプションとは関係なく、`file.encoding` プロパティに従って設定されます。

ソース・ファイルの文字エンコーディングを指定する場合は、`-encoding` オプションを使用します。このオプションには、Java 環境でサポートされているどの文字エンコーディングも指定できます。Sun 社の JDK を使用している場合、これらの文字エンコーディングは、`native2ascii` のドキュメントにリストされています。次の Web サイトにアクセスして参照してください（または、必要に応じて URL の「1.2」を「1.1」または「1.3」などの適切な JDK のレベルに置換してください）。

<http://www.javasoft.com/products/jdk/1.2/docs/tooldocs/solaris/native2ascii.html>

Sun 社の JDK では、数多くの文字エンコーディングがサポートされています。たとえば、`8859_1` ～ `8859_9` (ISO Latin-1 ～ ISO Latin-9)、JIS (日本語)、SJIS (Shift-JIS、日本語)、UTF8 などが含まれています。

文字エンコーディングでの注意事項 次の点に注意してください。

- メッセージとソース・ファイルのいずれの場合も、使用している文字エンコーディングで表現できない文字は、Java Unicode エスケープ・シーケンスで表現されます。その形式は、`\uHHHH` (H は 16 進数) です。
- 変換時の `.sqlj` ソース・ファイルの読み込みと処理で表示されるエラー・メッセージには、入力文字エンコーディング中の (バイト位置ではなく) 文字位置により、ソースの位置が示されます。
- 文字エンコーディング設定は、SQLJ `-encoding` オプションまたは Java `file.encoding` のどちらで設定したかにはかかわらず、Java プロパティ・ファイル (`sqlj.properties` や `connect.properties` など) には適用されません。プロパティ・ファイルでは、常に `8859_1` エンコーディングを使用します。これは、特別な SQLJ の機能ということではなく、一般的な Java の機能です。ただし、Unicode のエスケープ・シーケンスなら、プロパティ・ファイルで使用できます。エスケープ・シーケンスを決める際は、`native2ascii` ユーティリティを使用することをお勧めします。9-29 ページの「[native2ascii によるソース・ファイルの文字エンコーディングの変換](#)」を参照してください。

言語サポートの概要

SQLJ のエラーおよびステータス・レポートでは、変換時または実行時に、JVM の `user.language` プロパティの Java ロケール設定を使用します。通常、ユーザーはこの設定を変更する必要はありません。

言語サポートは、キーと値が対になったメッセージ・リソースを使用して実装されます。たとえば、英語の「OkKey」というキーと「Okay」という値は、ドイツ語の「OkKey」と「Gut」に相当します。使用するメッセージ・リソースは、ロケール設定の値で決定されます。

SQLJ でサポートされているロケール設定は、en（英語）、de（ドイツ語）、fr（フランス語）および ja（日本語）です。

注意： Java のロケール設定では、言語拡張の他にも、国拡張およびバリエーション拡張がサポートされています。たとえば、`ErrorMessage de_CH_var1` では、CH はドイツ語の国拡張であるスイスを示します。var1 は追加バリエーションです。ただし、現行の SQLJ でサポートされているのは言語拡張（ここでの例では de）のみで、国拡張およびバリエーション拡張は無視されます。

SQLJ および Java の文字エンコーディングと言語サポートの設定

Oracle SQLJ では、次の設定を行う構文が提供されています。

- SQLJ トランスレータおよび Java コンパイラで、ソース・コードの表示に使用される文字エンコーディング

SQLJ の `-encoding` オプションを使用します。

- SQLJ のトランスレータおよびランタイムで、エラー・メッセージやステータス・メッセージの表示に使用される文字エンコーディング

SQLJ の接頭辞 `-J` を使用して、Java の `file.encoding` プロパティを設定します。

- SQLJ のトランスレータおよびランタイムで、エラー・メッセージやステータス・メッセージの表示に使用されるロケール

SQLJ の接頭辞 `-J` を使用して、Java の `user.language` プロパティを設定します。

ソース・コードの文字エンコーディングの設定

SQLJ の `-encoding` オプションを使用して、トランスレータで読み込まれる `.sqlj` ファイル、トランスレータで生成される `.java` ファイル、コンパイラで読み込まれる `.java` ファイルの表現に使用されている文字エンコーディングを指定します。オプション設定は、SQLJ の `-compiler-encoding-flag` がオフになっていなければ、SQLJ によってコンパイラに渡されます。

このオプションは、コマンドラインで設定するか、または次の例のように環境変数 `SQLJ_OPTIONS` を使用して設定します。

```
-encoding=SJIS
```

SQLJ プロパティ・ファイルでは、次の構文を使用します。

```
sqlj.encoding=SJIS
```

エンコーディングのオプションが設定されていない場合、トランスレータとコンパイラでは、JVM の `file.encoding` プロパティで指定されたエンコーディングが使用されます。こ

これは、SQLJ コマンドラインからも設定できます。9-23 ページの「[SQLJ メッセージの文字エンコーディングとロケールの設定](#)」を参照してください。

詳細は、8-28 ページの「[入力および出力ソース・ファイルの文字エンコーディング \(-encoding\)](#)」および 8-79 ページの「[コンパイラが使用する文字エンコーディング \(-compiler-encoding-flag\)](#)」を参照してください。

注意： `-encoding` オプションを常に同じ値に設定する場合は、前述の 2 番目の例のようにプロパティ・ファイルで設定します。詳細は、8-16 ページの「[プロパティ・ファイルによるオプション設定](#)」を参照してください。

SQLJ メッセージの文字エンコーディングとロケールの設定

変換時と実行時に SQLJ のエラー・メッセージおよびステータス・メッセージをユーザーに出力するための文字エンコーディングおよびロケールは、Java の `user.language` プロパティで指定します。通常は必要ありませんが、SQLJ の `-J` 接頭辞を使用して、これらの JVM プロパティや他の JVM プロパティを SQLJ コマンドラインで設定できます。この接頭辞の付いたオプションは、JVM に渡されます。

文字エンコーディングは、次のように設定します。この例では、日本語の文字エンコーディングである Shift-JIS が設定されています。

```
-J-Dfile.encoding=SJIS
```

注意： プラットフォームやオペレーティング・システム的环境によっては、一部の文字エンコーディングしか利用できない場合があります。

ロケールは、次のように設定します。（この例では、日本語が設定されています。）

```
-J-Duser.language=ja
```

`-J` 接頭辞は、コマンドラインまたは環境変数 `SQLJ_OPTIONS` で設定する場合にのみ使用できます。プロパティ・ファイルでは使用できません。プロパティ・ファイルは、JVM が起動された後に読み込まれるためです。

注意：

- `file.encoding`、`user.language` などの Java プロパティを常に同じ値に設定する場合は、環境変数 `SQLJ_OPTIONS` に `-J` を設定しておきます。このように設定すると、コマンドラインで何度も同じ値を指定する必要がありません。この構文は、基本的にコマンドラインの構文と同じです。詳細は、8-19 ページの「[環境変数 SQLJ_OPTIONS によるオプションの設定](#)」を参照してください。
 - `SQLJ` の `-encoding` オプションが設定されていない場合は、`file.encoding` の設定がソース・ファイル、エラー・メッセージおよびステータス・メッセージに適用されます。
 - `file.encoding` プロパティを変更すると、予測できないところで Java 操作が影響される場合があります。新規に設定する場合は、オペレーティング・システムとの互換性に注意する必要があります。
-
-

`SQLJ` の接頭辞 `-J` の詳細は、8-12 ページの「[コマンドラインの構文と処理](#)」および 8-58 ページの「[Java Virtual Machine \(JVM\) に渡すオプション \(-J\)](#)」を参照してください。

SQLJ コマンドラインの例：文字エンコーディング化およびロケールの設定

次は、JVM の `file.encoding` および `user.language` が設定された完全な `SQLJ` コマンドラインの例です。

```
sqlj -encoding=8859_1 -J-Dfile.encoding=SJIS -J-Duser.language=ja Foo.sqlj
```

この例では、`SQLJ` の `-encoding` オプションを使用して、`SQLJ` 変換時のソース・コード表示に `8859_1` (`Latin-1`) を指定しています。`.sqlj` 入力ファイルの読み込み時と `.java` 出力ファイルの生成時には、この文字エンコーディングがトランスレータに使用されます。その後で、この文字エンコーディング設定が Java コンパイラに渡されて、生成された `.java` ファイルの読み込み時に使用されます。`-encoding` オプションが指定されていると、その設定は Java コンパイラに渡されます。ただし、`SQLJ` の `-compiler-encoding-flag` が無効に設定されている場合を除きます。

`Foo.sqlj` の変換時に `SQLJ` トランスレータによって出力されるエラー・メッセージおよびステータス・メッセージには、文字エンコーディングとして `SJIS`、ロケールとして `ja` が使用されます。

Oracle SQLJ 拡張グローバリゼーション・サポート

Oracle SQLJ では、既存の文字とストリーム型から導出された Java 型（Unicode 文字型）のサポートを追加しています。この型では、サーバーでのグローバリゼーション・サポートの使用方法が伝達されます。（SQLJ では、グローバリゼーション・サポートの目的を満たす JDBC 文または結果セットのメソッドを直接使用できません。それらのメソッドの詳細は、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。）

データベースがネイティブに Unicode をサポートする場合は、後述の「[グローバリゼーション・サポートのための Java 型](#)」に説明されている型は不要で、グローバリゼーション・サポートが透過的に処理されます。ネイティブに Unicode をサポートしていないが、データベースに Unicode をサポートする各国語キャラクタ・セットがある場合は、通常これらの型（各国語キャラクタ・セットを使用する列の型）を使用します。

グローバリゼーション・サポートのための Java 型

Oracle SQLJ では、グローバリゼーション・サポート用の Java 型がいくつか用意されています。表 9-3 には、これらのグローバリゼーション・サポート型と、汎用の JDBC と SQLJ の文字およびストリーム型の関係を示しています。NString 以外の各グローバリゼーション・サポート型は、JDBC または SQLJ 型のサブクラスです。

表 9-3 JDBC 型および SQLJ 型とグローバリゼーション型

JDBC 型および SQLJ 型	グローバリゼーション・サポート型
JDBC 型:	
oracle.sql.CHAR	oracle.sql.NCHAR
java.lang.String	oracle.sql.NString
oracle.sql.CLOB	oracle.sql.NCLOB
SQLJ 型:	
sqlj.runtime.CharacterStream	oracle.sqlj.runtime.NcharCharacterStream
sqlj.runtime.AsciiStream (現在は使用されていない。CharacterStream で使用)	oracle.sqlj.runtime.NcharAsciiStream (現在は使用されていない。 NcharCharacterStream で使用)
sqlj.runtime.UnicodeStream (現在は使用されていない。CharacterStream で使用)	oracle.sqlj.runtime.NcharUnicodeStream (現在は使用されていない。 NcharCharacterStream で使用)

アプリケーションで各国語文字列を処理する必要がある場合、各国語キャラクタ・セットの列との間の各国語文字列の受渡しには、対応する汎用型ではなく、グローバリゼーション・サポート型を使用します。

注意：

- すべてのグローバリゼーション・サポート型では、IN および OUT パラメータの使用を自動的に登録する機能が追加されますが、JDBC または SQLJ 型（コンストラクタなど）の使用方法は同じです。
 - グローバリゼーション・サポート型の使用は、イテレータ列では不要です。基になるネットワーク・プロトコルで、基になる結果セット用に各国語文字が暗黙的にサポートされているためです。
 - Oracle9i では、Oracle JDBC でのこれらのグローバリゼーション・サポート型が直接的にはサポートされていません。
-
-

NString クラスの使用方法和注意事項

oracle.sql.CHAR クラスおよびその NCHAR サブクラスは、データベース・キャラクタ・セットに関する明示的な知識が必要なコンストラクタのみを提供します。したがって、たいていの環境では java.lang.String のラッパーである oracle.sql.NString クラスの方が好ましい選択です。

NString クラスにはより簡単なコンストラクタがあり、各国語文字の使用形式が確実に JDBC ドライバに登録されます。

次に主要な NString メソッドを示します。

- NString(String) —このコンストラクタでは、既存の String インスタンスから NString インスタンスが作成されます。
- String toString() —このメソッドでは、基になる String インスタンスが戻されます。
- String getString() —このメソッドでも、基になる String インスタンスが戻されます。

toString() メソッドによって、文字列連結式に NString インスタンスを使用できます（たとえば、"a"+b で、b が文字列です）。CHAR スーパークラスにある getString() メソッドは、一貫性のためにもサポートされています。さらに、String クラスのメンバー・メソッドは、NString ラッパー・クラスに引き継がれます。これによって、より簡潔なコードを記述できます。

グローバリゼーション・サポートの例

次の例では、NSString クラスの使用を説明しています。

入力引数としての NSString この例では、NSString インスタンスをデータベースへの入力パラメータとして使用します。

```
import oracle.sql.NString;
...
NSString nc_name = new NSString("Name with strange characters");
#sql { update PEOPLE
        set city = :(new NSString("\uffff2")), name = :nc_name
        where num= :n };
...
```

出力引数としての NSString この例では、NSString インスタンスをデータベースからの出力パラメータとして使用します。

```
import oracle.sql.NString;
...
NSString nstr;
#sql { call foo(:out nstr) };
System.out.println("Result is: "+nstr);
// or, explicitly: System.out.println("Result is: "+nstr.toString());
...
```

結果セット列としての NSString この例では、イテレータ列に NSString 型を使用します。このような使用法は不要ですが、基になるネットワーク・プロトコルで各国語文字を暗黙的にサポートしているため特に問題とはなりません。またこの例では、NSString に引き継がれる String メソッドの 1 つである substring() の使用方法を示しています。

```
import oracle.sql.NString;
import oracle.sql.NCLOB;
...
#sql iterator NIter(NSString title, NCLOB article);

NIter nit;
#sql nit = { SELECT article, title FROM page_table };
while (nit.next())
{
    System.out.println("<TITLE>"+nit.title()+"</TITLE>");
    ...
    nit.article().substring(0, 1000); ...
}
```

注意： 前述の例で `NString` 型ではなくて `NCHAR` 型を使用するには、次のように変更する必要があります。

- 適切な `NCHAR` コンストラクタを使用します。次のように、`NCHAR` コンストラクタは `CHAR` コンストラクタを反映しています。

```
NCHAR(String str, oracle.sql.CharacterSet charset)
```

- `NString` インスタンスからその基になる `String` を取得するには、`toString()` または `getString()` のいずれかを使用できますが、`NCHAR` インスタンスの場合、`getString()` メソッドを使用する必要があります。`NString` 型を使用する場合、文字列連結のために `toString()` メソッドが自動的に使用されます。前述の「[出力引数としての NString](#)」を参照してください。
-
-

SQLJ 外部でのグローバリゼーション・サポート用の操作

ここでは、SQLJ 外での Oracle グローバリゼーション・サポートの設定の操作方法について説明します。

アプリケーションの実行時に行う文字エンコーディングとロケールの設定

通常の Java アプリケーションと同じように、JVM を起動して SQLJ アプリケーションを実行する際に、`file.encoding` や `user.language` などの JVM のプロパティを直接指定できます。これにより、アプリケーション実行時に出力されるメッセージに使用される文字エンコーディングおよびロケールが決定します。

たとえば、次のように指定します。

```
java -Dfile.encoding=SJIS -Duser.language=ja Foo
```

この場合、文字エンコーディングには `SJIS` が使用され、ロケールには日本語が使用されます。

API による Java プロパティの設定

Java コードで Java プロパティの値を確認するには、`java.lang.System.getProperty()` メソッドを使用し、該当のプロパティを指定します。次にその例を示します。

```
public class Settings
{
    public static void main (String[] args)
    {
        System.out.println("Encoding: " + System.getProperty("file.encoding")
            + ", Language: " + System.getProperty("user.language"));
    }
}
```

このコードをコンパイルすると、スタンドアロン・ユーティリティとして実行できます。

すべてのプロパティの値を戻り値とする `getProperties()` メソッドもあります。ただし、サーバー側で実行されるコード中にこのメソッドを使用すると、セキュリティの例外が生成されます。

`java.lang.System` クラスの詳細は、次の Web サイトを参照してください（または、必要に応じて URL の「1.2」を「1.1」または「1.3」などの適切な JDK のレベルに置換してください）。

<http://www.javasoft.com/products/jdk/1.2/docs/api/java.lang.System.html>

native2ascii によるソース・ファイルの文字エンコーディングの変換

Sun 社の JDK を使用している場合は、SQLJ を使用せずにソース・コードの文字エンコーディングを変換できます。`native2ascii` というユーティリティを使用すると、ネイティブな文字エンコーディングのソースが、Unicode エスケープ・シーケンスを使用した 7 ビット ASCII に変換されます。

注意： `native2ascii` で生成されたソース・コードを SQLJ で変換する場合は、SQLJ を起動する側の JVM の `file.encoding` に、7 ビット ASCII のスーパーセットをサポートする文字エンコーディングを設定するようにしてください。EBCDIC または Unicode の場合は、この設定は必要ありません。

`native2ascii` を実行するには、次のように指定します。

```
% native2ascii <options> <inputfile> <outputfile>
```

入力ファイルまたは出力ファイルが指定されていない場合は、標準入力または標準出力が使用されます。このコマンドでは、次の 2 つのオプションを使用できます。

- `-reverse` (逆変換を行います。つまり、Latin-1 または Unicode からネイティブ・コードに変換します。)
- `-encoding <encoding>`

次にその例を示します。

```
% native2ascii -encoding SJIS Foo.sqlj Temp.sqlj
```

詳細は、次の Web サイトを参照してください（または、必要に応じて URL の「1.2」を「1.1」または「1.3」などの適切な JDK のレベルに置換してください）。

<http://www.javasoft.com/products/jdk/1.2/docs/tooldocs/solaris/native2ascii.html>

パフォーマンスとデバッグ

この章では、SQLJ アプリケーションのパフォーマンスを向上させ、SQLJ のソース・コードを実行時にデバッグするための機能、ユーティリティおよびヒントについて説明します。次の項目について説明します。

- [パフォーマンス強化の機能](#)
- [SQLJ のデバッグ機能](#)

パフォーマンス強化の機能

Oracle SQLJ には、データへのアクセスを効率化することによって、パフォーマンスを向上させるための機能が提供されています。具体的には、次の機能があります。

- 行プリフェッチ: 問合せ結果を一度に 1 行ではなく、複数行単位でアプリケーションへ送る機能です。
- バッチ更新: データベースでの更新、挿入、削除内容を一度に 1 つではなく、バッチ単位で転送する機能です。
- 文のキャッシング: プリコンパイルされた SQL 文をメモリーに格納し再利用することにより、サーバーで処理が繰り返されるのを防ぐ機能です。
- 列定義: 列型とサイズを事前定義することにより、データベースへのラウンドトリップを減らす効果をもたらす機能です。
- パラメータ・サイズ定義: ホスト変数のサイズを事前定義することにより、メモリーの使用効率を改善する機能です。

上に示した機能に対する Oracle JDBC サポートの詳細は、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。

デフォルトの Oracle 固有コード生成 (-codegen=oracle) の使用は、アプリケーションにとってメリットがあります。生成されたコードは Oracle JDBC への直接コールによって最適化されるため、SQLJ ランタイムへの中間コールのオーバーヘッドが削減できます。詳細は、4-36 ページの「[Oracle 固有コード生成 \(プロファイルなし\)](#)」を参照してください。

注意: 行のセットを値の配列にフェッチするバッチ・フェッチは、Oracle SQLJ でも Oracle JDBC でもサポートされていません。ただし、Oracle 行プリフェッチを使用すると、バッチ・フェッチのメリットをいくつか得られます。

前述した Oracle SQLJ (および JDBC) パフォーマンス強化に加えて、SQLJ プログラムでは SQL 操作に関するオプティマイザ・ヒントを使用できます。同様に、他の Oracle SQL 操作に関するヒントも使用できます。

Oracle SQL では、SQL 文をチューニングできます。その場合、「/*+」や「--+」で開始されるコメントを使用して、Oracle オプティマイザにヒントを渡します。SQLJ トランスレータは、これらのオプティマイザ・ヒントを認識してそれらに従い、実行時に SQL 文の一部としてヒントを渡します。

Oracle9i の SQL の最適化のための拡張機能を使用すると、SQLJ ストアド・ファンクションやその他のストアド・ファンクションのコストおよび選択性の情報を定義できます。オプティマイザを使用すると、SQL 実行時に、ストアド・ファンクションのコスト関数および選択関数の起動、他の実行方法の評価、ならびに最も効率的な方法の選択ができます。

Oracle オプティマイザの詳細は、『Oracle9i SQL リファレンス』を参照してください。

コード中に Oracle パフォーマンス拡張機能を使用する場合は、次の要件が伴うので注意してください。

- Oracle JDBC ドライバを使用する必要があります。
- デフォルトの Oracle 固有コード生成を使用するか、プロファイルを適切にカスタマイズします。
ISO 標準コード生成の場合は、デフォルトのカスタマイザ `oracle.sqlj.runtime.util.OraCustomizer` をお勧めします。
- アプリケーション実行時に Oracle SQLJ ランタイムを使用する必要があります。

Oracle SQLJ ランタイムと Oracle JDBC ドライバは、Oracle カスタマイザを使用してプロファイルをカスタマイズするときは常にアプリケーションで必要となるもので、Oracle 拡張機能をコード中に使用しない場合にもこのことが当てはまります。

行プリフェッチ

標準の JDBC は、問合せ結果を 1 行ずつ受信します。この場合、行ごとにデータベースへのラウンドトリップが必要です。行プリフェッチを使用すると、一度に複数行の結果を効率的に取り出すことが可能になります。

`ExecutionContext` インスタンスの `setFetchSize()` メソッドを使用して、`SELECT` 文 (特定の `ExecutionContext` インスタンスを使用する SQLJ 文用) を実行するたびにプリフェッチされる行数を設定します。

`ExecutionContext` インスタンスの `getFetchSize()` メソッドは、現行のプリフェッチ・サイズを `int` 値として戻します。

次の例では、デフォルトの接続コンテキスト・インスタンスのデフォルトの実行コンテキスト・インスタンスを取得し、メソッドをコールして、プリフェッチ・サイズを 20 に設定します。

```
DefaultContext.getDefaultContext().getExecutionContext().setFetchSize(20);
```

(JDBC API を使用して、基になる `OracleConnection` オブジェクトに直接プリフェッチ・サイズを設定することも可能ですが、SQLJ 内ではこれは設定できません。)

特定の接続コンテキストのインスタンスを使用した問合せに対して、プリフェッチする行数を指定するには、基になる JDBC 接続を使用して `Connection` インスタンスにキャストします。次の例は、デフォルト接続で、プリフェッチする行数を 20 に設定しています。

```
((Connection)DefaultContext.getDefaultContext().getConnection()).setDefaultRowPrefetch(20);
```

この値は、使用する接続コンテキストのインスタンスごとに個別に設定する必要があります。たとえば、宣言された接続コンテキスト・クラスの `ctx` というインスタンスに対しては、プリフェッチする行数を次のように設定します。

```
((Connection)ctx.getConnection()).setDefaultRowPrefetch(20);
```

プリフェッチできる行数に制限はありません。JDBC でのデフォルト値は 10 で、この値は SQLJ に継承されます。通常環境では、この値で十分に対応できます。受け取る行数が多い場合でも、この値を大きくする必要はありません。

文のキャッシング

SQLJ には文のキャッシング機能が用意されています。この機能は、ループや繰り返して呼ばれるメソッドなどで使用される実行文を減少させることによりパフォーマンスを改善する機能です。文のキャッシュはクライアントに配置されます。再実行の前に文がキャッシュされている場合、コードの再解析（クライアントまたはサーバーで）、**Statement** オブジェクトの再作成、およびパラメータ・サイズの定義の再計算は不要です。この機能がない場合、繰り返し使用される文は、クライアント上とサーバー内で再解析する必要があります（この再解析の必要性は、文の再検出時に、その文がサーバー側の SQL キャッシュで使用可能かどうかによって異なります）。

Oracle 固有コード生成（デフォルトの `-codegen=oracle` 設定）の場合、Oracle SQLJ の文キャッシングは Oracle JDBC ドライバに依存し、このドライバは、Oracle JDBC の明示的なキャッシング・メカニズムを使用します。後述のような相互依存性がありますが、この明示的なキャッシング・メカニズムは、Oracle JDBC の暗黙的なキャッシング・メカニズムとは異なります。これらのメカニズムの詳細は、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。Oracle 固有のコードでは、文のキャッシングは接続メソッドを介して制御されます。

ISO コード生成（`-codegen=iso`）の場合、SQLJ には、SQLJ ランタイムの機能を介した SQLJ 自体の文キャッシング・メカニズムがあります。ISO コードでは、文のキャッシングは Oracle カスタマイザの `stmtcache` オプションを介して制御されます。

ここでは、次の項目について説明します。

- 文キャッシングのための接続コンテキスト・メソッド（Oracle 固有コードの場合）
- 文キャッシングの有効化と無効化（Oracle 固有コードの場合）
- SQLJ/ 明示的なキャッシングと JDBC の暗黙的なキャッシング間の主な相互作用
- 文キャッシングに対する JDBC サポート（ISO コードの場合）
- 文のキャッシュ・サイズに対応した Oracle カスタマイザ・オプション（ISO コードの場合）
- 文のキャッシングに関するその他の動作
- 文キャッシングの制限事項と考慮事項

注意：

- Oracle 固有コード生成の場合は、明示的なキャッシングが SQLJ の API を介して操作できる唯一の文キャッシング・メカニズムです。このマニュアルでは、このメカニズムを「SQLJ/ 明示的な」文キャッシングと呼びます。
 - 過去に文のキャッシングを使用した経験のあるユーザーは、Oracle9i リリース 2 (9.2) で、SQLJ と JDBC の文キャッシングの多くの機能が大幅に変更されていることに注意してください。ただし、下位互換性は保証されています。
-

文キャッシングのための接続コンテキスト・メソッド (Oracle 固有コードの場合)

Oracle 固有コード生成を使用する場合 (SQLJ トランスレータのデフォルトの `-codegen=oracle` 設定を使用する場合) は、文のキャッシング機能に接続コンテキスト・メソッドを使用します。

文のキャッシュ・サイズに 0 (ゼロ) よりも大きい値を指定することによって、SQLJ/ 明示的な文キャッシングが有効になります。デフォルトの場合、キャッシュ・サイズが 5 (5 文) で有効となります。

次の Oracle 固有 (非標準) static メソッドが `sqlj.runtime.ref.DefaultContext` クラスに追加されました。また、宣言した接続コンテキスト・クラスには、これらのメソッドが含まれています。

- `public static void setDefaultStmtCacheSize(int)`

デフォルトの文キャッシュ・サイズをすべての接続コンテキストに設定します。これは、メソッドをコールするクラスのみでなく、後で作成される接続コンテキスト・クラスのインスタンスのための初期の文キャッシュ・サイズになります。メソッド・コールは、既存の接続コンテキスト・インスタンスには影響しません。

- `public static int getDefaultStmtCacheSize()`

接続コンテキスト対する現行のデフォルトの文キャッシュ・サイズを取得します。

また、次の Oracle 固有インスタンス・メソッドが `DefaultContext` クラスに追加されました。これらのメソッドは、他のどの接続コンテキスト・クラスにも含まれています。

- `public void setStmtCacheSize(int)`
`throws java.sql.SQLException`

特定の接続コンテキスト・インスタンスの基になる接続に文キャッシュ・サイズを設定します (デフォルトを上書きします)。

注意： SQLJ/ 明示的なキャッシングがすでに無効な場合は、サイズを 0（ゼロ）に設定すると無効のままになります。すでに有効な場合にサイズを 0（ゼロ）に設定すると、有効のままですが機能は停止します。

■ `public int getStmtCacheSize()`

SQLJ/ 明示的な文キャッシングが、接続コンテキストの基になる接続に対して有効かどうかを検証します。有効な場合は、現行の文キャッシュ・サイズを戻します。また、次の整数のいずれかを戻すこともできます。

```
public static int STMT_CACHE_NOT_ENABLED
public static int STMT_CACHE_EXCEPTION
```

`getStmtCacheSize()` コールで SQL 例外が発生する可能性があります。ただし、下位互換性を維持するために、このメソッドでは例外を直接発生させることはありません。例外が発生した場合、このメソッドは定数 `STMT_CACHE_EXCEPTION` を戻します。この場合は、`getStmtCacheException()` メソッド（次に説明します）をコールして、例外の内容を確認できます。

SQLJ/ 明示的なキャッシングが無効に設定されている場合に `getStmtCacheSize()` をコールすると、定数 `STMT_CACHE_NOT_ENABLED` が戻ります。Oracle9i リリース 2 (9.2) の場合、これはキャッシュ・サイズが 0（ゼロ）であることとは区別されます。0（ゼロ）のキャッシュ・サイズで SQLJ/ 明示的なキャッシングを有効にする（役には立ちませんが）ことは、技術的には可能です。

■ `public java.sql.Exception getStmtCacheException()`

文キャッシングの例外があるかどうかを確認します。このメソッドの使用方法は 2 種類あります。

- `getStmtCacheSize()` コールで `STMT_CACHE_EXCEPTION` が戻る場合にコールします。
- 文のキャッシングを使用するための接続コンテキスト・インスタンスを作成する場合は、必ずこのメソッドをコールします。これは、接続コンテキスト・インスタンスの作成時に、常に文キャッシュ・サイズに関して発生する自動操作のためのコールです。（詳細は、10-9 ページの「[文のキャッシングに関するその他の動作](#)」を参照してください。）接続コンテキスト・インスタンスに対して文のキャッシングを行う場合は、インスタンスの作成後に `getStmtCacheException()` をコールして、問題がなかったことを確認します。

文キャッシングの有効化と無効化（Oracle 固有コードの場合）

Oracle 固有コードでは、前に実行された文を繰り返すと、0（ゼロ）でない文キャッシュ・サイズによって SQLJ の明示的なキャッシングが有効になります。デフォルト・サイズが5のため、文のキャッシングがデフォルトで有効になっています。

SQLJ の明示的な文キャッシングは、SQLJ の API を介して明示的に無効にすることはできませんが、文のキャッシュ・サイズを 0（ゼロ）に設定することによって、事実上無効（機能停止）にできます。この場合、接続コンテキストの `getStmtCacheSize()` メソッドは、`STMT_CACHE_NOT_ENABLED` ではなく 0（ゼロ）を返します。

SQLJ の明示的な文キャッシングまたは JDBC の暗黙的なキャッシングは、JDBC 接続 API を介して明示的に無効にできます。SQLJ/ 明示的なキャッシングおよび JDBC の暗黙的なキャッシングでは、10-7 ページの「[SQLJ/ 明示的なキャッシングと JDBC の暗黙的なキャッシング間の主な相互作用](#)」に説明されているように、同じキャッシュ・サイズを使用していることが、明示的に無効にする理由の一部である場合があります。次のメソッドは、`OracleConnection` クラスを介して使用できます。

- `public void setExplicitCachingEnabled(boolean)`
- `public boolean getExplicitCachingEnabled()`
- `public void setImplicitCachingEnabled(boolean)`
- `public boolean getImplicitCachingEnabled()`

SQLJ 接続コンテキスト・インスタンス内から `OracleConnection` インスタンスを取得すると、これらのメソッドにアクセスできます。（7-48 ページの「[SQLJ 接続コンテキストと JDBC 接続の連係動作](#)」を参照してください。）

注意： SQLJ では、JDBC の暗黙的なキャッシングはデフォルトで無効です。これは `setImplicitCachingEnabled()` メソッドを介して明示的に有効にしないかぎり無効のままです。

SQLJ/ 明示的なキャッシングと JDBC の暗黙的なキャッシング間の主な相互作用

Oracle 固有コードでの文のキャッシングに関して、このマニュアルでは、JDBC の暗黙的なキャッシングではなく、SQLJ の明示的なキャッシングに説明の重点をおいています。アプリケーションで JDBC コードを使用しない場合は、SQLJ の明示的なキャッシングが唯一の適切な文のキャッシングです。ただし、SQLJ と JDBC コードの両方をアプリケーションで使用する環境では、暗黙的なキャッシングも使用する可能性があります。

前述のように、SQLJ の明示的なキャッシングと JDBC の暗黙的なキャッシングは、相互に関係なく有効です。さらに、ユーザーは SQLJ を介して暗黙的なキャッシュにアクセスすることはできません。

ただし、この2種類のキャッシング間には、同じキャッシュ・サイズを共有するという主な相互作用があります。たとえば、文のキャッシュ・サイズが5の場合は、SQLJの明示的なキャッシングと暗黙的なキャッシングの組合せで最大5個の文をキャッシュできます。

ここで重要な点は、キャッシュ・サイズを0（ゼロ）に設定して、SQLJの明示的な文キャッシングを無効に設定すると、暗黙的なキャッシングも無効になることです。

また、SQLJの明示的なキャッシングが無効な場合に、キャッシュ・サイズを0（ゼロ）より大きい値に変更するとそのキャッシングが有効になりますが、これは暗黙的なキャッシングの有効化には影響を与えません。

文キャッシングに対する JDBC サポート（ISO コードの場合）

SQLJ トランスレータの `-codegen=iso` 設定を介して指定された ISO 標準コード生成の場合、文のキャッシングは標準 SQLJ 機能であり、特定の JDBC ドライバは不要です。ただし、インタフェース `sqlj.runtime.profile.ref.ClientDataSupport` を実装したドライバを使用すると、キャッシングの堅牢性が向上します。Oracle9i JDBC ドライバには、このインタフェースが実装され、次の機能が提供されています。

- アプリケーション全体で静的キャッシュを1つのみ使用するのではなく、データベース接続のたびに別個のキャッシュを使用できます。
- 同じ接続からの複数の接続コンテキスト・クラスのインスタンスの間でキャッシュされた文を共有できます。

キャッシュを1つ使用した場合、`ClientDataSupport` が実装されていない汎用 JDBC ドライバの場合と同様、一方の接続で文を実行したために、もう一方の接続で実行した文がフラッシュされてしまうことがあります（文のキャッシュ・サイズ、つまりキャッシングの許容される文の上限の数が超過した場合）。

文のキャッシュ・サイズに対応した Oracle カスタマイザ・オプション（ISO コードの場合）

ISO 標準コード生成の場合、文のキャッシングは Oracle カスタマイザの使用時にアプリケーション内で有効です。デフォルトのキャッシュ・サイズは5（Oracle 固有コードと同じデフォルトのサイズ）です。文のキャッシングは通常、Oracle SQLJ 変換の一部として実行されます。

必要に応じて文のキャッシュ・サイズを変更したり、キャッシュ・サイズを0（ゼロ）に設定して文のキャッシングを無効にするには、Oracle カスタマイザの `stmtcache` オプションを使用できます。`-P-Cstmtcache=n` のように設定します（`n` は整数）。詳細は、A-32 ページの「[Oracle カスタマイザ用の文のキャッシュ・サイズ・オプション（stmtcache）](#)」を参照してください。

接続コンテキスト・クラスをいくつか使用するに伴ってプロファイルもいくつか使用する場合は、プロファイルごとに個別に SQLJ（事実上はカスタマイザ）を実行すると、それぞれの文のキャッシュ・サイズを個別に設定できます。

実行時、文のキャッシュ・サイズは、適切な SQLJ プロファイルによって接続に対して決定されます。この SQLJ プロファイルとは、この接続でインスタンス化された最初の接続コンテキスト・クラスに対応するプロファイルのことです。文のキャッシュ・サイズの設定値は、プロファイルのカスタマイズ時に設定した Oracle カスタマイザの `stmtcache` オプションの値に応じて決まります。接続時に最初の文が実行されたとき、その接続に使用する実行時の文のキャッシュ・サイズが設定されます。

文のキャッシングに関するその他の動作

ここでは、Oracle 固有または ISO のコード生成に適用可能な、文のキャッシングに関するその他の動作と相互作用について説明します。

JDBC 文キャッシュ・サイズの自動操作 SQLJ 接続コンテキスト・オブジェクトがインスタンス化されるときに、基になる JDBC 接続の文キャッシュ・サイズが接続コンテキスト・クラスのデフォルトのサイズよりも小さい場合、SQLJ ランタイムは、JDBC の文キャッシュ・サイズを接続コンテキストのデフォルト値まで増分しようとします。この操作は、ISO コード生成でも発生し、処理中の Oracle JDBC の明示的な文キャッシングを有効にします。ただし、この操作は ISO コードの場合実際には関連がありません。

反対に、実際の JDBC 文キャッシュ・サイズが大きい場合は、SQLJ ランタイムはキャッシュ・サイズを変更しません。SQLJ ランタイムでは、SQLJ 接続コンテキスト・インスタンスを作成すると、デフォルトのサイズ・セットに対する実際の JDBC キャッシュ・サイズを必ず確認します。

これらのメソッドは、同一の基になる静的フィールドを変更または報告するので、メソッドが発行されたコンテキスト・クラスにかかわらず、同じ効果があることに注意します。

次の接続コンテキスト・クラス宣言を例として挙げます。

```
#sql context CtxtA;  
#sql context CtxtB;
```

この場合、次の 3 つのコード命令には、新しい SQLJ 接続コンテキスト・インスタンスが続いて作成されるときに SQLJ/ 明示的な文キャッシングを有効にしないという同じ効果があります。

```
sqlj.runtime.ref.DefaultContext.setDefaultStmtCacheSize(0);
```

または

```
CtxtA.setDefaultStmtCacheSize(0);
```

または

```
CtxtB.setDefaultStmtCacheSize(0);
```

重要： SQLJ 接続コンテキスト・インスタンスが基になる JDBC のプールされた接続上に作成される場合は、SQLJ は JDBC 文キャッシュ・サイズを変更できません。（Oracle 固有コードの場合、発生した例外は接続コンテキストの `getStmtCacheException()` メソッドを介して取得できます。）この場合、必要な JDBC 文キャッシュ・サイズを基になる物理接続に明示的に設定する必要があります。データ・ソースの場合、キャッシュ・サイズは、ベンダー固有のデータ・ソース属性を介して設定されます。

文キャッシングのセマンティクス SQLJ/ 明示的なキャッシングと JDBC の暗黙的なキャッシングの機能には、異なるセマンティクスと動作があります。前述のように、SQLJ 文キャッシングは、ループや同じメソッドへ繰り返して実行されるコールなど、繰り返し使用される単一の文にのみ適用されます。次に示す例で確認してください。

```
#sql { same SQL operaton }; // occurrence #1
...
Java code
...
#sql { same SQL operaton }; // occurrence #2
...
Java code
...
#sql { same SQL operaton }; // occurrence #3
...
```

3 つの SQL 操作は、空白も含めてすべて同じであると想定します。

SQLJ キャッシングでは、これら 3 つの同じ SQL 操作は、それぞれ異なる文であると考えられています。それらは、キャッシュ内で 3 つの別々のスロットを占有します。ただし、JDBC の暗黙的なキャッシングでは、3 つすべてに 1 つのキャッシュ・スロットを使用するので、これらを同じ文として認識します。文は、2 番目と 3 番目の文で再利用されます。

文キャッシングの制限事項と考慮事項

文のキャッシュを使用すると、キャッシュ・サイズが 1 の場合でも、たいていの SQLJ アプリケーションではパフォーマンスが向上します。ただしこの場合は、次の留意点があります。

- 各文が 1 回しか実行されない場合は、メリットがあるわけではありません。
- 複数回実行した文を使用して 1 回実行した文をインターリーブするのは避けてください。実行回数が 1 回のみである文を使用すると、文のキャッシュ中に不必要な領域がとられてしまうため、キャッシュ・サイズが上限を超える原因となります。ISO コード生成を使用している場合は、前述の方法のかわりに、実行回数が 1 回のみである文に対して個別の接続コンテキスト・クラスを使用して、この接続コンテキスト・クラスの文キャッシングを無効にできます。

- 文が別々の場合は各文中の SQL 操作が同じでも、別個の文として認識され、処理もキャッシングも別々に実行されます。代替方法として、SQL 操作をメソッド中に記述して、このメソッドを繰り返しコールする方法もあります。
- 文のキャッシュ・サイズは、慎重に選択してください。キャッシュ・サイズが小さすぎるとキャッシュが満杯になり、文が再実行されないうちにフラッシュしてしまいます。大きすぎる場合は、データベースやプログラムのリソースが使い果たされてしまう可能性があります。

文キャッシングに関する次の一般的な注意に従ってください。

- Oracle 固有コード生成の場合、別々の SQLJ 接続コンテキスト・インスタンスを使用して、別の文キャッシングを動作させることは、接続コンテキストで同一の基になる JDBC 接続インスタンスを共有していると、うまくいきません。これは、Oracle 固有コード生成の場合、SQLJ は JDBC 文キャッシュを使用するためです。
- Oracle アプリケーションでは、文のキャッシュ・サイズとアプリケーション（直接または SQLJ を使用）でオープンしている JDBC 文の最大個数とを合計した数を、各セッションで利用できるカーソルの最大個数よりも少ない数とする必要があります。同時にオープンすることのできる文の最大個数は、カーソルの最大個数に基づいて決まるからです。
- 文のキャッシングを使用したからといって、操作自体の実行セマンティクスは通常は変わりませんが、場合によっては変わることもあります。たとえば、リソース解放時に例外を送出するための文を作成した場合がこれに該当します。キャッシングを使用すると、接続のクローズまたはキャッシュから文がフラッシュ（キャッシュ・サイズが上限を超えたときにも、文がフラッシュされます）が行われなにかぎり、例外が送出されなくなります。

バッチ更新機能

バッチ更新機能（Sun 社 JDBC 2.0 の仕様ではバッチ更新と呼ばれる）では、UPDATE、DELETE および INSERT 文が処理可能です。これらの文は、バッチ可能でしかも互換性があります（後述）。そのため、一括してデータベースに転送し実行するまでの処理は一度で済むので、データベースへの無駄なラウンドトリップが省けます。この機能は、JDBC 2.0 および SQLJ 仕様に含まれているので、Oracle9i JDBC および Oracle SQLJ でサポートされています。バッチ更新機能は、一般に、ループ内で繰り返し実行する操作に有効です。

SQLJ のバッチ更新機能は、実行コンテキストの使用とつながりがあります。このバッチ更新機能は、実行コンテキストごとに有効化 / 無効化の個別指定が可能で、各実行コンテキストのインスタンスにはそのインスタンス自体のバッチが保持されます。

ここでは、次の項目について説明します。

- [バッチ可能かつ互換性のある文](#)
- [バッチ更新機能の有効化 / 無効化](#)
- [明示的および暗黙的なバッチ実行](#)

- バッチの取消し
- 実行コンテキストの更新カウント
- バッチ制限の設定
- 非互換の文に対するバッチ処理
- 暗黙的な実行コンテキストを使用したバッチ更新機能
- バッチ更新機能に関する一般的な注意
- バッチ実行中のエラー状態
- 再帰的コールインとバッチ更新機能

重要： バッチ更新について、次の点に注意してください。

- デフォルトの Oracle 固有コード生成を使用するか、ISO コード生成の場合はアプリケーションを Oracle カスタマイザでカスタマイズする必要があります。
 - 自動コミット・モードは使用禁止にすることをお勧めします。これによって、バッチ実行時にエラーが発生した場合に、コミットする対象とロールバックする対象を制御できるようになります。
-
-

バッチ可能かつ互換性のある文

ある文を既存のバッチの文に追加できるかどうかは、次のような 2 つの基準によって決まります。

- バッチ可能かどうか。どんな場合にも、1 つに一括できない文もあります。
- 既存のバッチの文との互換性がある文か。

バッチ可能 Oracle SQLJ の場合、次の文の種類をバッチで実行できます。

- UPDATE
- INSERT
- DELETE

ただし、次のような制約があるので注意してください。

- UPDATE 文や INSERT 文にストリーム・ホスト式を 1 つ以上記述した場合、バッチ可能ではなくなります。

将来のリリースや他の SQLJ 実装では、バッチ可能な文がさらに追加される可能性があります（具体的には、ストアド・プロシージャ・コールや DDL 文など）。

互換性 Oracle SQLJ では、同じ文の複数のインスタンス間でのみ互換性が確保されます。このことは、次のどちらかの場合に起こります。

- ループ内で文が繰り返し実行される場合
- メソッドで文が実行され、しかもそのメソッドが繰り返しコールされる場合

将来のリリースや他の SQLJ 実装では、互換性の確保された文がさらに追加される可能性があります（具体的には、ホスト式を含まない文のインスタンスなど）。

バッチ更新機能の有効化 / 無効化

SQLJ のバッチ更新機能は、実行コンテキストごとに個別に実行されます。各実行コンテキスト・インスタンスは、他の実行コンテキスト・インスタンスとは独立してバッチ更新機能の有効化することが可能で、しかも各インスタンスは、そのインスタンス自体のバッチが保持されます。

特定の実行コンテキスト・インスタンスに対してバッチ更新機能の有効化 / 無効化を指定するには、その実行コンテキスト・インスタンスの `setBatching()` メソッドを使用します。このメソッドは、入力としてブール値をとります。

```
...
ExecutionContext ec = new ExecutionContext();
ec.setBatching(true);
...
```

または

```
...
ExecutionContext ec = new ExecutionContext();
ec.setBatching(false);
...
```

バッチ更新機能は、デフォルトでは無効になります。

注意： 既存の文のバッチは、`setBatching()` メソッドの処理対象にはなりません。バッチ更新の有効化 / 無効化には関係なく、既存のバッチについては実行や取消しが行われないようになっています。

ある実行コンテキストに対してバッチ更新機能が有効になっているかどうかを調べるには、その実行コンテキスト・インスタンスの `isBatching()` メソッドを使用します。

```
ExecutionContext ec = new ExecutionContext();
...
boolean batchingOn = ec.isBatching();
```

ただし、上に示した指定では、現在のバッチが未完了のままになっているかどうかは、提示されません。

明示的および暗黙的なバッチ実行

未完了のバッチ更新は、必要であれば明示的に実行できますが、状況によっては暗黙的に実行されます。

注意： バッチ実行中に例外が発生する場合もあるので、そうした例外による影響を考慮することは重要です。10-21 ページの「[バッチ実行中のエラー状態](#)」を参照してください。

明示的な実行 バッチ更新機能を明示的に実行するには、実行コンテキスト・インスタンスの `executeBatch()` メソッドを使用します。このメソッドは、更新カウン트의 `int` 型配列を戻り値としています。この詳細は、10-17 ページの「[実行コンテキストの更新カウント](#)」を参照してください。

バッチを明示的に実行する例を次に示します。

```
...
ExecutionContext ec = new ExecutionContext();
ec.setBatching(true);
...
double[] sals = ...;
String[] empnos = ...;
for (int i = 0; i < empnos.length; i++)
{
    #sql [ec] { UPDATE emp SET sal = :(sals[i]) WHERE empno = :(empnos[i]) };
}
int[] updateCounts = ec.executeBatch();
...
```

注意： `executeBatch()` の起動時に、実行コンテキスト・インスタンスに未完了のバッチがなければ、このメソッドの戻り値は `null` になります。

暗黙的な実行 未完了のバッチが存在するときは、次のような場合に暗黙的に実行されます。

- バッチ処理を行うことができない実行文が出現した場合。この場合最初に既存のバッチが実行の対象となり、次にバッチ処理の行うことができない文が実行されます。
- ある UPDATE 文がバッチ可能であっても、既存のバッチとは非互換（つまり、その文のインスタンスではない）の場合は、バッチが実行された後で、非互換の文から新規のバッチが生成されます。
- 事前定義されたバッチ数の上限（指定された文の数）に達した場合。詳細は、10-18 ページの「[バッチ制限の設定](#)」を参照してください。

次に、その例を示します。最初にあるバッチが生成され、バッチ処理の行われない文が出現した時に暗黙的に実行されます。その後で、新しいバッチが生成され、バッチ可能であるが非互換の文が出現した時に、暗黙的に実行されます。

```
ExecutionContext ec = new ExecutionContext();
ec.setBatching(true);
...
/* Statements in the following loop will be placed in a batch */
double[] sals = ...;
String[] empnos = ...;
for (int i = 0; i < empnos.length; i++)
{
    #sql [ec] { UPDATE emp SET sal = :(sals[i]) WHERE empno = :(empnos[i]) };
}

/* a SELECT is unbatchable so causes the batch to be executed */
double avg;
#sql [ec] { SELECT avg(sal) INTO :avg FROM emp };

/* Statements in the following loop will be placed in a new batch */
double[] comms = ...;
for (int i = 0; i < empnos.length; i++)
{
    #sql [ec] { UPDATE emp SET comm = :(comms[i]) WHERE empno = :(empnos[i]) };
}

/* the following update is incompatible with the second batch, so causes it to be
executed */
int smithdeptno = ...;
#sql [ec] { UPDATE emp SET deptno = :smithdeptno WHERE ename = 'Smith' };
```

暗黙的に実行されたバッチの更新カウントを取得するには、その実行コンテキスト・インスタンスの `getBatchUpdateCounts()` メソッドを起動します。このメソッドの起動後、この実行コンテキスト・インスタンスで正常に実行されたバッチのうち、最後のバッチの実行更新カウントが戻り値として戻されます。SELECT 文および最後の UPDATE 文の後に、次のコード文が挿入される場合があります。

```
int[] updateCounts = ec.getBatchUpdateCounts();
```

これらの更新カウントの意味は、10-17 ページの「[実行コンテキストの更新カウント](#)」を参照してください。

注意： 実行コンテキスト・インスタンスに対して正常に実行されたバッチ更新がなければ、`getBatchUpdateCounts()` の戻り値は `null` になります。

バッチの取消し

実行コンテキストの未完了のバッチを取り消すには、その実行コンテキスト・インスタンスの `cancel()` メソッドを使用します。たとえば、バッチ実行時に発生した例外のイベント内で、実行済みであるが、コミットされていないバッチを取り消すことができます。次に例を示します。

```
...
ExecutionContext ec = new ExecutionContext();
ec.setBatching(true);
...
double[] sals = ...;
String[] empnos = ...;
for (int i = 0; i < empnos.length; i++)
{
    #sql [ec] { UPDATE emp SET sal = :(sals[i]) WHERE empno = :(empnos[i]) };
    if (!check(sals[i], empnos[i])) //assume "check" is a user-supplied function
    {
        ec.cancel();
        throw new SQLException("Process canceled.");
    }
}

try
{
    int[] updateCounts = ec.executeBatch();
} catch ( SQLException exception) { ec.cancel(); }
...
```

あるバッチを取り消した場合、次のバッチ可能な文から新規のバッチが開始されます。

注意：

- `cancel()` をコールすると、現時点で実行中の文もすべて取り消されます。
 - バッチを取り消しても、バッチ更新は無効になりません。
-

実行コンテキストの更新カウント

Oracle SQLJ では、更新カウントの配列（ある実行コンテキスト・インスタンスの `executeBatch()` メソッドまたは `getBatchUpdateCounts()` メソッドの戻り値）からは、バッチの文で更新された行数カウントはわかりません。この戻り値からわかるのは、各文の更新の成功または不成功を示す値のみです。このため、バッチ処理が無効である場合、実行コンテキスト・インスタンスの `getUpdateCount()` メソッドの戻り値である単一の更新カウントと前に示したカウントは機能が異なるものです。この詳細は、7-25 ページの「[状態メソッド](#)」で解説します。

いくつかの文を 1 つに一括してバッチ実行すると、`getUpdateCount()` からの戻り値である単一の更新カウントの内容も変更されます。

`getUpdateCount()` の戻り値 バッチ可能な環境では、実行コンテキスト・インスタンスの `getUpdateCount()` メソッドの戻り値は、各文が出現するたびに修正されます。この戻り値は、`ExecutionContext` クラスの `static int` 定数の値（次のうちの 1 つ）で更新されます。

- `NEW_BATCH_COUNT` —最後に出現した文に対して新規のバッチが生成されたことを示す定数
- `ADD_BATCH_COUNT` —最後に出現した文が、既存のバッチに追加されたことを示す定数
- `EXEC_BATCH_COUNT` —最後の文が出現した後で、未完了のバッチが明示的または暗黙的に実行されたことを示す定数

上に示した定数を参照するには、次の修飾名を使用します。

```
ExecutionContext.NEW_BATCH_COUNT  
ExecutionContext.ADD_BATCH_COUNT  
ExecutionContext.EXEC_BATCH_COUNT
```

`executeBatch()` または `getBatchUpdateCounts()` の戻り値 バッチが明示的または暗黙的に実行された後に `executeBatch()` や `getBatchUpdateCounts()` から戻された配列には、文の実行の正常・異常終了のみが示されます。各バッチの文には、それぞれ配列要素があります。JDBC 2.0 仕様に従って、配列要素の値が -2 であれば、対応する文の実行は正常終了したのが、更新行数は不明であるという意味になります。

バッチの実行後に配列の値すべてに目を通して、大して意味がありません。この戻り値としての配列に対するチェックとしては、実行前にバッチ形態にまとめられた文の数を確認す

ることのみが挙げられ、具体的には実行の正常終了後（基本的には、例外を発生しないバッチ実行後）に配列の要素数をチェックします。

複数の文が一括されてそのバッチが実行される段階では、更新カウン트의配列は未更新になっています。

バッチ制限の設定

事前定義された数の文がバッチされてから次の文が追加されるまでの間に、各更新バッチが実行されるように指定できます。次のように、実行コンテキスト・インスタンスの `setBatchLimit()` メソッドを使用し、0（ゼロ）でない正の整数を入力します。

```
...
ExecutionContext ec = new ExecutionContext();
ec.setBatching(true);
ec.setBatchLimit(10);
...
double[] sals = ...;
String[] empnos = ...;
for (int i = 0; i < 20; i++)
{
    #sql [ec] { UPDATE emp1 SET sal = :(sals[i]) WHERE empno = :(empnos[i]) };
}
```

このループの実行回数は 20 回ですが、文のバッチとバッチの実行は 11 回目のループ中、つまり 11 個目の文がバッチに追加される前に完了します。ループの中では、バッチはループの 2 回目には処理されないのに注意してください。アプリケーションがループを抜けた後は、別の文を実行するか、バッチを明示的に実行するまでは最後の 10 個の文はこの未実行のバッチの中に残ります。

`ExecutionContext` クラスの `static int` 定数（次の 2 つのどちらか）は、`setBatchLimit()` への入力として使用できます。

- `AUTO_BATCH` — SQLJ ランタイムにバッチ制限を判断させるための定数
- `UNLIMITED_BATCH`（デフォルト） — バッチ制限のないことを明示する定数

次にその例を示します。

```
...
ExecutionContext ec = new ExecutionContext();
ec.setBatching(true);
ec.setBatchLimit(ExecutionContext.AUTO_BATCH);
...
```

または

```
ec.setBatchLimit(ExecutionContext.UNLIMITED_BATCH);
...
```

現在のバッチ制限をチェックするには、実行コンテキスト・インスタンスの `getBatchLimit()` メソッドを使用します。

非互換の文に対するバッチ処理

既存のバッチ形態の文とは互換性がない文をバッチ処理する際に、このバッチを暗黙的に実行しない場合は、別個の実行コンテキスト・インスタンスを使用してください。次に例を示します。

```
...
ExecutionContext ec1 = new ExecutionContext();
ec1.setBatching(true);
ExecutionContext ec2 = new ExecutionContext();
ec2.setBatching(true);
...
double[] sals = ...;
String[] empnos = ...;
for (int i = 0; i < empnos.length; i++)
{
    #sql [ec1] { UPDATE emp1 SET sal = :(sals[i]) WHERE empno = :(empnos[i]) };
    #sql [ec2] { UPDATE emp2 SET sal = :(sals[i]) WHERE empno = :(empnos[i]) };
}
int[] updateCounts1 = ec1.executeBatch();
int[] updateCounts2 = ec2.executeBatch();
...
```

注意： この例では、2つの UPDATE 文が互いに依存しないことを前提としています。相互に依存しあう文を別々の実行コンテキストでバッチ処理するのは避けてください。必ずしも各文を実行した順序で実行できるとは限らないからです。

かわりの方法としては、単一の実行コンテキストと別個のループを使用し、EMP1 の更新をすべて一括して実行した後で、EMP2 の更新を実行する方法があります。具体的には、次のようにします。

```
...
ExecutionContext ec = new ExecutionContext();
ec.setBatching(true);
...
double[] sals = ...;
String[] empnos = ...;
for (int i = 0; i < empnos.length; i++)
{
    #sql [ec] { UPDATE emp1 SET sal = :(sals[i]) WHERE empno = :(empnos[i]) };
}
for (int i = 0; i < empnos.length; i++)
```

```
{
    #sql [ec] { UPDATE emp2 SET sal = :(sals[i]) WHERE empno = :(empnos[i]) };
}
ec.executeBatch();
...
```

この例の最初のバッチは暗黙的に実行されているのに対し、2 番目のバッチは明示的に実行されます。

暗黙的な実行コンテキストを使用したバッチ更新機能

ここまでのすべてのバッチ更新の例では、明示的な実行コンテキスト・インスタンスを指定してきました。あらゆる実行コンテキスト・インスタンスが暗黙的な実行コンテキスト・インスタントを持てば必要ありません。たとえば、デフォルト接続の暗黙的な実行コンテキスト・インスタンスを参照するには、次のようにします。

```
DefaultContext.getDefaultContext().getExecutionContext().setBatching(true);
...
double[] sals = ...;
String[] empnos = ...;
for (int i = 0; i < empnos.length; i++)
{
    #sql { UPDATE emp SET sal = :(sals[i]) WHERE empno = :(empnos[i]) };
}
// implicitly execute the batch and commit
#sql { COMMIT };
```

上の方法のかわりに、このバッチを明示的に実行することも可能です。

```
DefaultContext.getDefaultContext().getExecutionContext().executeBatch();
```

バッチ更新機能に関する一般的な注意

バッチ更新機能処理を使用する場合、特に、実行コンテキスト・インスタンスがバッチの文とバッチ形態でない文とを混合する場合は、次の点に注意してください。

- バッチ形態でない文はバッチの文に依存するので、まずバッチの文を実行し、その後でバッチ形態でない文を実行する必要があります。
- JDBC COMMIT 操作や ROLLBACK 操作、つまり、JDBC Connection インスタンスの自動コミットまたは `commit()` メソッドまたは `rollback()` メソッドを使用する場合は、バッチ中の未完了の文は実行されません。

ただし、SQLJ COMMIT 文または ROLLBACK 文を次のように使用すると、バッチ中の完了していない文が実行されます。

```
#sql { COMMIT };
```

または

```
#sql { ROLLBACK };
```

この理由からも、変更内容のコミットやロールバックには、常に `#sql` 構文を使用することをお勧めします。この構文を使用すると、SQLJ リソースと JDBC リソースの両方をクリーンアップできます。

- バッチ処理ができない文または非互換の文が現れてバッチが暗黙的に実行された場合、そのバッチは、バッチ処理が実行されない文または非互換の文が実行される前に、文の入力パラメータが評価されて目的の文に渡された後で実行されます。
- バッチ可能な実行コンテキスト・インスタンスのうち特定のインスタンスのみの使用を中止する際は、リソースを解放するために、未完了のバッチを暗黙的に実行するか、取り消すことをお勧めします。

バッチ実行中のエラー状態

ある文が原因でバッチ実行中に例外が発生した場合、次の点に注意してください。

- 例外発生の原因となった文よりも後で処理されるバッチの文は、実行の対象とはなりません。
- 例外が発生する前にすでに実行されたバッチの文は、ロールバックの対象とはなりません。
- 別の（バッチ処理が行えないか非互換の）文が現れたために例外が発生したバッチは、暗黙的な実行を試みても実行されません。

注意： バッチ更新の使用時には自動コミット・モードを無効にしてください。これによって、バッチ実行時にエラーが発生した場合、コミットおよびロールバックを制御できます。

JDBC 2.0 以上の環境でバッチ実行時に例外が発生した場合、通常は標準 `java.sql.BatchUpdateException` クラスのインスタンスで、`java.sql.SQLException` クラスのサブクラスです。（JDK 1.1.x 環境下で、Oracle JDBC ドライバでは、`oracle.jdbc2.BatchUpdateException` クラスによって、バッチ更新例外をサポートしています。）

`BatchUpdateException` クラスの `getUpdateCounts()` メソッドは、例外が発生する前に正常に実行した文に対して、戻り値として更新カウントの配列を返します。この配列は、`ExecutionContext class executeBatch()` メソッドや `getBatchUpdateCounts()` メソッドの戻り値と同じ配列です。

再帰的コールインとバッチ更新機能

11-23 ページの「[サーバーでの再帰的 SQLJ コール](#)」でも説明したとおり、SQLJ ストアド・プロシージャでは、あるプロシージャから別のプロシージャをコールすると、2つのプロシージャが同時に1つの実行コンテキスト・インスタンスを使用することになります。バッチ更新フラグ（実行コンテキスト・インスタンスの `setBatching()` メソッドを使用して設定）は、フラグを設定したストアド・プロシージャには関係なく、その実行コンテキストの属性と同じ動作をします。このフラグ設定値は、どちらかのストアド・プロシージャ内にある次の実行文に反映されます。

このため、再帰的コールインのたびにサーバーではバッチ更新機能が自動的に無効化されます。再帰的に起動されるプロシージャでは、未完了のバッチは実行されますが、バッチ処理は行われなくなります。

こうした動作を回避するには、バッチ可能なストアド・プロシージャで明示的な実行コンテキスト・インスタンスを使用します。

列の定義

Oracle SQLJ では、列の型およびサイズの定義に Oracle JDBC のサポートが反映されています。ドライバの実装は、Oracle JDBC ドライバごとに若干異なりますが、列の型およびサイズを登録すると、問合せごとのデータベースとのやり取りが軽減されます。このことは特に、Oracle JDBC Thin ドライバと位置指定イテレータの使用に当てはまります。

列の定義に関する Oracle SQLJ 実装

Oracle SQLJ では列の定義を有効にすると、次に示したような、列型とサイズを登録するための手順が自動的に開始されます。

- デフォルトの Oracle 固有コード生成を使用した（`-codegen=oracle`）カスタマイズ時または変換時に、Oracle SQLJ は、特定のデータベース・スキーマに接続して、取得する列の型およびサイズを確認します。ISO 標準 SQLJ コード生成の場合、列のデフォルトは SQLJ プロファイルの一部になります。この処理が行われるのは、ソース・コードの変換をカスタマイズするための手順の際または既存プロファイルを個別にカスタマイズする際です。
- アプリケーション稼働時、SQLJ ランタイムでは列情報を利用し、JDBC ドライバで列型とサイズを登録します。この際に Oracle JDBC 文のクラスの `defineColumnType()` メソッドへのコールが使用されます。

列の定義に使用するカスタマイザおよびトランスレータのオプション

列の定義を有効化するには、SQLJ のオプションを次のように設定します。

- `optcols` フラグを有効にします。Oracle 固有コード生成の場合は、SQLJ トランスレータの `-optcols` オプションを使用します。ISO 標準コード生成の場合は、トランスレータのオプションまたは Oracle カスタマイザのオプションを使用します（SQLJ コマンドラインで `-P-Coptcols` と入力します）。

- データベース接続用のユーザー、パスワードおよび URL を設定します。Oracle 固有コード生成の場合、この設定は SQLJ トランスレータの `-user`、`-password` および `-url` の各オプションを介して行います。ISO 標準コード生成の場合、この設定は、トランスレータのオプションを介して、またはカスタマイザのオプションを個別に使用して行われます (SQLJ コマンドラインで `-P-user`、`-P-password` および `-P-url` を入力します)。デフォルトの `OracleDriver` クラスを使用する場合は、さらに、JDBC ドライバ・クラスも設定します (SQLJ コマンドラインで `-P-driver` と入力します)。

トランスレータのオプションの詳細は、8-51 ページの「[列の定義 \(-optcols\)](#)」および 8-32 ページの「[接続オプション](#)」を参照してください。

カスタマイザのオプションの詳細は、A-23 ページの「[カスタマイザ固有のオプションの概要](#)」の `optcols` の項および A-11 ページの「[カスタマイザ・ハーネスのオプションの概要](#)」の `user`、`password`、`url` および `driver` の項を参照してください。

パラメータ・サイズの定義

Oracle JDBC および Oracle SQLJ では、JDBC リソース割当て量が最適化できます。このためには、パラメータ・サイズ (次のいずれかに使用する Java ホスト変数のサイズ) を定義します。

- ストアド・プロシージャまたはファンクション・コールの入出力パラメータ
- ストアド・ファンクション・コールからの戻り値
- SET 文の入出力パラメータ
- PL/SQL ブロックの入出力パラメータ

パラメータ・サイズの定義に関する Oracle SQLJ 実装

Oracle SQLJ では、オプション設定値とソース・コードのコメントに埋込みの「ヒント」との組合せによって、パラメータ・サイズが実装されます。ISO 標準 SQLJ コード生成の場合、Oracle カスタマイザのオプションが使用できます。デフォルトの Oracle 固有コード生成の場合、同等の SQLJ トランスレータのオプションが使用できます。

次のように、オプションおよびヒントを使用します。

- SQLJ トランスレータまたは Oracle カスタマイザのパラメータ定義フラグを使用して、パラメータ・サイズの定義を有効にします。
- SQLJ トランスレータまたは Oracle カスタマイザのパラメータ・デフォルト・サイズ・オプションを使用して、特定のデータ型のデフォルト・サイズを指定します。
- ソース・コードのコメントにヒントを次の書式で埋め込んで、特定のデータ型のデフォルト・サイズを指定変更します。

指定したホスト変数に対するパラメータ・サイズの定義を使用可能にすると、ソース・コード・ヒントがあればそれに従ってリソースが割り当てられます。ソース・コード・ヒントがない場合、対応するデータ型のデフォルト・サイズが指定されていれば、そのデフォルト・

サイズが使用されます。ソース・コード・ヒントも該当のデフォルト・サイズも指定されていない場合、JDBC 実装に基づいて最大のリソース量が割り当てられます。

アプリケーション実行時に、Oracle JDBC 文のクラスで使用できる `defineParameterType()` メソッドと `registerOutParameter()` メソッドをコールすることによって、パラメータ・サイズが登録されます。

注意： パラメータ定義フラグを有効にしなかった場合は、パラメータ・サイズのデフォルトとソース・コード・ヒントが無視され、JDBC 実装に基づいて最大またはデフォルトのリソースが割り当てられます。

パラメータ・サイズの定義に使用するカスタマイザとトランスレータのオプション

パラメータ・サイズの定義には、次の SQLJ のオプションを使用します。

- パラメータ・サイズの定義を有効化するには、`optparams` フラグを使用します。Oracle 固有コード生成の場合は、SQLJ トランスレータの `-optparams` オプションを使用します。ISO 標準コード生成の場合は、トランスレータのオプションまたは Oracle カスタマイザのオプションを使用します (SQLJ コマンドラインで `-P-Coptparams` と入力します)。
- 特定のデータ型にデフォルトのサイズを設定するには、`optparamdefaults` を使用します。Oracle 固有コード生成の場合は、SQLJ トランスレータの `-optparamdefaults=xxxx` オプションを使用します。ISO 標準コード生成の場合は、トランスレータのオプションまたは Oracle カスタマイザのオプションを使用します。SQLJ コマンドラインで `-P-Coptparamdefaults=xxxx` と入力します。

トランスレータのオプションの詳細は、8-52 ページの「[パラメータの定義 \(-optparams\)](#)」および 8-54 ページの「[パラメータのデフォルト・サイズ \(-optparamdefaults\)](#)」を参照してください。

カスタマイザのオプションの詳細は、A-23 ページの「[カスタマイザ固有のオプションの概要](#)」の該当する項を参照してください。

パラメータ・サイズの定義に使用するソース・コード・ヒント

パラメータ・サイズ定義のためのヒントは、次の形式で SQLJ 文のソース・コードに埋め込みます (必要であれば、コメント内に空白文字を追加してもかまいません)。

```
/*(size)*/
```

サイズはバイト単位になっています。`optparams` オプションを無効にすると、ヒントが無視されます。

デフォルトのパラメータ・サイズは、新規のサイズを指定せずに（JDBC 実装でのサイズの自動割当てにより）指定変更するには、次のようにします。

```
/*()*/
```

次にその例を示します。

```
byte[] hash;  
String name=Tyrone;  
String street=2020 Meryl Street;  
String city=Wichita;  
String state=Kansas;  
String zipcode=77777;  
#sql hash = { /* (5) */ VALUES (ADDR_HASH(:name /* (20) */ , :street /* () */ ,  
                                :city, :state, :INOUT zipcode /* (10) */ )) };
```

結果式のヒント（前述の例では結果式 hash）は、SQLJ 文の大カッコの内側に記述します。入出力ホスト変数のヒントは、前述の例に示したように、このホスト変数のすぐ後ろに記述する必要があります。

この例では、パラメータ・サイズが次のように設定されます。

- hash — 5 バイト
- name — 20 バイト
- street — デフォルトを設定なしで指定変更します（JDBC による自動割当て）
- city — なし（該当のデータ型のデフォルトが指定されていればそれが使用されます）
- state — なし（該当のデータ型のデフォルトが指定されていればそれが使用されます）
- zipcode — 10 バイト

注意： いずれかのパラメータ・サイズが変更された場合（実行時に実際のサイズが登録済みのサイズを超えた場合など）、SQL の例外が送出されます。

SQLJ のデバッグ機能

ここでは、Oracle SQLJ のデバッグ機能をまとめます。また、このマニュアルでこれら機能の詳細を説明している部分も示します。

- SQLJ をコマンドラインから実行する場合は、トランスレータのオプション `-linemap` (`jdb` デバッガを使用している場合は `-jdblinemap` オプション) を使用できます。このオプションは、SQLJ コードのデバッグに有用です。
- サーバー側の埋込みトランスレータには、Java コードのデバッグに有用なオプションが提供されています。ただし、このオプションは SQLJ コードのデバッグには使用できません。
- ISO 標準コード生成を使用している場合は、専用カスタマイザ `AuditorInstaller` を使用して、デバッグに使用する「オーディタ」をインストールできます。
- SQLJ は、Oracle9i JDeveloper 統合開発環境に統合されています。そのため、JDeveloper のデバッグ機能を利用できます。

デバッグ用の SQLJ `-linemap` フラグ

`-linemap` フラグを有効にすると、SQLJ ソース・コード・ファイルの行番号が、対応する `.class` ファイルの場所にマッピングされます。このファイルは、SQLJ トランスレータで生成された `.java` ファイルのコンパイルで生成される `.class` ファイルになります。このマッピングにより、Java の実行時エラーが発生した場合に Java Virtual Machine (JVM) から出力される行番号が、SQLJ ソース・コードの行番号と同じになります。このため、デバッグがはるかに容易になります。

Sun 社 `jdb` デバッガを使用する際は、`-linemap` オプションのかわりに、`-jdblinemap` オプションを使用します。この 2 つのオプションは機能がほとんど同じですが、`-jdblinemap` ではさらに特別な処理が実行できます。この特別な処理とは、`jdb` デバッガでサポートされる Java ソース・ファイルが `.java` 拡張子付きファイル名のみに限られているために必要となる処理のことです。

詳細は、8-48 ページの「SQLJ ソース・ファイルとの行マッピング (`-linemap`)」および 8-49 ページの「`jdb` デバッガでの SQLJ ソース・ファイルへの行マッピング (`-jdblinemap`)」を参照してください。

注意： サーバー側で変換すると、その変換で生成されたクラス・スキーマ・オブジェクトが、SQLJ ソース・コードにマッピングしてある行番号を自動的に参照します。クライアント側での変換時に `-linemap` オプションを有効化すると、サーバー側と同様の自動参照が行われます。

サーバー側の debug オプション

SQLJ ソースをサーバーにロードし、サーバー側の埋込みトランスレータを使用して変換する場合、サーバー側デバッグ・オプションを指定してサーバー側コンパイラからデバッグ情報を出力できます。この情報は、.sqlj または .java ソース・ファイルをサーバー側でコンパイルしたときに出力されます。これは、クライアント上で標準の javac コンパイラを実行するときに、-g オプションを使用するのに相当します。このオプションは、SQLJ コードのデバッグには有用ではありませんが、Java コードのデバッグには有効です。

このオプションと、サーバー側オプションの設定方法の詳細は、11-16 ページの「[サーバー側の埋込みトランスレータでサポートされるオプション](#)」を参照してください。

Oracle JVM でのデバッグ方法の概要は、『Oracle9i Java Developer's Guide』を参照してください。

AuditorInstaller 専用カスタマイズの概要

ISO コード生成用に、SQLJ には専用カスタマイザ AuditorInstaller が用意されています。このカスタマイザは、SQLJ コマンドラインで指定されたプロファイルに、オーディタと呼ばれる一連のデバッグ文を挿入します。これらのプロファイルは、前回のカスタマイズによって生成済みであることが必要です。

アプリケーションを実行すると、SQLJ ランタイムでデバッグ文が実行され、メソッド・コールと戻り値のトレース情報が表示されます。

デバッグ文を挿入するには、カスタマイズの汎用オプションの場合と同じように、先頭に -P- を付けて、カスタマイザ・ハーネスの debug オプションを使用します。

AuditorInstaller の詳細は、A-42 ページの「[デバッグ用の AuditorInstaller カスタマイザ](#)」を参照してください。

Oracle9i JDeveloper での開発とデバッグの概要

Oracle SQLJ は、Oracle9i JDeveloper ビジュアル・プログラミング・ツールに完全に統合されています。

JDeveloper には、SQLJ をサポートする統合デバッガも用意されています。アプリケーションの実行時に、標準の Java 文と同じように SQLJ 文を 1 行ずつデバッグできます。報告される行番号は、生成された Java コードの行番号ではなく、SQLJ ソース・コードの行番号に対応しています。

JDeveloper の概要は、1-29 ページの「[Oracle9i JDeveloper などの IDE での SQLJ の使用](#)」を参照してください。

サーバー側 SQLJ

SQLJ アプリケーションは、Oracle9i サーバー側に格納後、直接実行できます。SQLJ アプリケーションを実行する場合、クライアント側で変換とコンパイルを行った後で、生成されたクラスとリソースをサーバー側にロードする方法と、SQLJ のソース・コードをサーバー側にロードした後で、サーバーの埋込みトランスレータで変換とコンパイルを行う方法があります。

この章では、マルチスレッドや再帰的な SQLJ コールなどの注意事項を含め、サーバー側での SQLJ の機能と使用方法について説明します。

次の項目について説明します。

- [サーバー側 SQLJ の概要](#)
- [サーバーで使用する SQLJ コードの作成](#)
- [クライアント側での SQLJ ソースの変換とコンポーネントのロード](#)
- [SQLJ ソースのロードとサーバーでの変換](#)
- [Java スキーマ・オブジェクトの削除](#)
- [その他の考慮事項](#)

サーバー側 SQLJ の概要

SQLJ コードは、Java コードと同様に、Oracle9i のストアド・プロシージャ、ストアド・ファンクションまたはトリガーでも実行できます。データ・アクセスは、SQLJ ランタイムのサーバー側実装（SQLJ ランタイム・パッケージがすべて自動的に利用できる状態）とサーバー側の Oracle JDBC 内部ドライバを介して行います。（この JDBC 内部ドライバは、KPRB ドライバと呼ばれています。）

また、Oracle9i には埋込みの SQLJ トランスレータがあるため、サーバー側で使用する SQLJ ソース・ファイルをサーバーで直接変換することも可能です。

SQLJ をサーバーで実行するときは、サーバー側コーディングの問題、コードの変換先およびサーバーへのロード方法について考慮する必要があります。生成された出力の名前が、サーバーで決定される方法についても注意する必要があります。クライアント側でコードを変換およびコンパイルして、クラス・ファイルとリソース・ファイルをサーバーにロードすることも、.sqlj ソース・ファイルをサーバーにロードして、埋込み SQLJ トランスレータで自動的に変換することも可能です。

サーバー側の埋込みトランスレータのユーザー・インタフェースと、クライアント側のトランスレータのユーザー・インタフェースは異なります。サポートされているオプションは、データベース表を使用して指定します。エラーはデータベース表に出力されます。トランスレータによる出力ファイルを開発者が意識する必要はありません。

注意：

- Oracle9i リリース 2 (9.2) では、サーバーは JDK 1.3 J2SE 環境を使用します。サーバー側の SQLJ 環境は、SQLJ 固有の接続 Bean のサポートと 11-3 ページの「サーバーで使用する SQLJ コードの作成」に記述されている関連する例外を除いて、runtime12ee ライブラリを使用しているクライアント側の環境とほぼ等しいといえます。
 - このマニュアルでは、システム設定にかかわる作業を、SQLJ 開発者の仕事としては位置付けていません。そのため、Oracle9i Java Virtual Machine (JVM) の設定はここでは説明していません。Java 関連の設定パラメータ (JAVA_POOL_SIZE など) の詳細は、『Oracle9i Java Developer's Guide』を参照してください。マルチスレッド・サーバー、ディスパッチャまたはリスナーの構成に関する情報が必要な場合は、『Oracle9i Net Services 管理者ガイド』を参照してください。
-
-

Oracle9i データベースの J2EE の非サポートに関する注意事項:

Oracle9i Application Server Containers for J2EE (OC4J) (新しく軽量で、使いやすく高速な、認定済みの J2EE コンテナ) の導入に従って、Oracle9i リリース 2 (9.2) からは、Java 2 Enterprise Edition (J2EE) と CORBA のスタックがデータベースでサポートされなくなります。ただし、データベース埋込みの Java VM (Oracle JVM) はまだ存在し、データベースでの Java 2 Standard Edition (J2SE) の機能、Java ストアド・プロシージャ、JDBC および SQLJ を提供するために機能拡張が継続して行われます。Oracle9i リリース 2 (9.2) では、次のテクノロジーがデータベースでサポートされなくなります。

- 次の要素で構成される J2EE スタック
 - Enterprise Beans (EJB) コンテナ
 - JavaServer Pages (JSP) コンテナ
 - Oracle9i Servlet Engine (OSE)
- Visibroker for Java をベースにした、埋込み Common Object Request Broker Architecture (CORBA) フレームワーク

サブレット、JSP ページ、EJB および CORBA のオブジェクトを Oracle データベースに配置することはできません。Oracle9i リリース 1 (9.0.1) は、J2EE と CORBA スタックをサポートする最後のデータベース・リリースです。データベースで実行している既存の J2EE アプリケーションは、OC4J に移行することをお勧めします。

サーバーで使用する SQLJ コードの作成

ほとんどの場合、ターゲットの Oracle9i データベースで使用する SQLJ コードの記述方法は、クライアント側で使用する SQLJ コードの記述方法と同じです。記述方法に若干の相違はありますが、これは Oracle JDBC の特性や一般的な Java 特性によるもので、SQLJ に固有のものではありません。ただし、次のような考慮事項があります。

- サーバー自体への暗黙的な接続が行われます。
- 自動コミット機能など、コーディング上の注意点があります。
- サーバー側のデフォルトの出力デバイスは、現行のトレース・ファイルになります。
- サーバー側とクライアント側とでは、名前解決の機能が異なります。
- SQL と Java とでは、名前の解釈および処理の方法が異なります。
- サーバーには、JSP、EJB または CORBA の機能はありません。JSP コンテナがないため、サーバー側のコードでは SQLJ JSP 接続 Bean を使用できません。

注意： あるサーバーからサーバー側 **Thin** ドライバを介して別のサーバーに接続するための SQLJ コードを記述する場合は、クライアント側 **Thin** ドライバを使用したアプリケーション用のコードを記述する場合と同じようにします。この項で取り上げた留意点が当てはまる場面は、ほとんどありません。

サーバー側でのデータベース接続

サーバー自体で SQLJ コードを実行した場合には、データベース接続が明示的に行われなくなる点で、サーバー接続の概念が若干異なってきます。デフォルトでは、データベースへの暗黙的な接続が使用されます。これはサーバー側で実行されるあらゆる Java プログラムで使用可能な接続です。SQLJ プログラムのための接続は、自動的に初期化されます。手動で初期化する必要はありません。ドライバの登録や指定、接続インスタンスの作成、デフォルトの接続コンテキストの指定、`#sql` 文の接続オブジェクトの指定、接続の終了といったことは、いずれも必要ありません。

注意： サーバー側で、次に示したようにデフォルトの接続コンテキストを `NULL` に設定すると、デフォルトの接続コンテキスト（サーバーへの暗黙的な接続）が再設定されます。

```
DefaultContext.setDefaultContext (null);
```

サーバー側でのコーディングの注意事項

ターゲットのサーバーでサーバー側内部ドライバを使用してコードを実行する際には、コーディング上考慮する必要がある問題が多少伴います。留意点を次に挙げます。

- 内部ドライバで発行された結果セットは、複数のコールにわたって使用され続けます。また、ファイナライザを実行しても、カーソルは解放されません。このため、すべてのイテレータを閉じて、使用可能なカーソルを確保することが重要です。ただし、複数のコールで実際に使用されている場合など、イテレータを開いたままにしておく特別の理由がある場合は別です。
- 内部ドライバでは自動コミット機能がサポートされないため、サーバー側では自動コミット設定が無視されます。データ更新を実行または取り消すには、明示的な `COMMIT` 文または `ROLLBACK` 文を使用します。

```
#sql { COMMIT };  
...  
#sql { ROLLBACK };
```

注意： Java Transaction Service (JTS) のトランザクションなどの、XA トランザクションを使用する場合は、SQLJ や JDBC の COMMIT/ROLLBACK 文またはメソッドを使用できません。

- ISO 標準コード生成の場合、JDBC コードと連係する SQLJ コードを使用して、デフォルトでない接続コンテキスト・インスタンスを使用する場合は、そこにキャッシュされている文をクリーンアップするために最後に接続コンテキスト・インスタンスを終了する必要があります（同じ接続コンテキスト・インスタンスをセッション継続中に使用する場合は除く）。次に例を示します。

```
DefaultContext ctx = new DefaultContext(conn); // conn is JDBC connection
#sql [ctx] { SQL operation };
...
ctx.close(sqlj.runtime.ConnectionContext.KEEP_CONNECTION);
...
```

接続コンテキスト・インスタンスを終了しない場合、文ハンドルがセッション中に不足する可能性があります。また、基になる JDBC 接続オブジェクトの終了のみで、文ハンドルは再生されません。これは、アプリケーションをクライアントで実行するときの動作とは異なります。

(デフォルトの Oracle 固有コード生成の場合、文は基になる JDBC 文キャッシュにキャッシュされ、自動的に再生できます。)

- Oracle 固有コード生成をサーバーで実行するコードに使用する場合は、明示的な `ExecutionContext` インスタンスを使用します。これによって、アプリケーションは、ISO 標準 SQLJ コード生成で変換されたアプリケーションと完全に連係できるようになります。

1 つの接続に 1 つのスレッドを使用する（つまり、1 つの Oracle セッションに 1 つのスレッドを使用する）場合、次の例のように、静的インスタンスを 1 つ使用すれば十分です。

```
public static ExecutionContext ec = new ExecutionContext();
...
#sql [ec] { SQL operation };    // use ec for all operations
```

1 つの接続に複数のスレッドを使用する場合、実行メソッドを起動するたびに別のコンテキスト・インスタンスを使用する必要があります。

サーバー側 JDBC、サーバー側内部および Thin ドライバの詳細は、『Oracle9i JDBC 開発者ガイドおよびリファレンス』を参照してください。

サーバーのデフォルトの出力デバイス

Oracle Java Virtual Machine (JVM) では、現行のトレース・ファイルがデフォルトの出力デバイスになります。

サーバーで実行中のプログラムの標準出力 (`System.out.println()` コールからの出力など) のすべてをユーザー画面に表示する場合は、次に示すように、DBMS_JAVA パッケージの `SET_OUTPUT()` プロシージャを実行します。バッファ・サイズはバイト単位 (この場合は 10,000 バイト) で入力します。

```
sqlplus> execute dbms_java.set_output(10000);
```

バッファ・サイズを超過した出力は、失われます。

サーバー側で実行するコードをユーザー画面に明示的に出力されるようにする場合は、Java `System.out.println()` メソッドを使用するか、PL/SQL DBMS_OUTPUT.PUT_LINE() プロシージャを使用します。

PUT_LINE() プロシージャをオーバーロードする際には、VARCHAR2、NUMBER または DATE のいずれかを入力パラメータとして設定し、出力対象を指定できます。

DBMS_OUTPUT パッケージの詳細は、『Oracle9i 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』を参照してください。

サーバー側での名前解決

サーバー側でのクラスのロードと名前変換は、クライアント側での場合とは異なるパラダイムに従って行われます。これは、サーバー側とクライアント側の環境自体が異なるためです。ここでは概要のみに絞って説明します。この項目の詳細は、『Oracle9i Java Developer's Guide』を参照してください。

Oracle JVM での Java 名の解決には次が必要です。

- クラス・リゾルバ仕様。クラス・スキーマ・オブジェクト (機能的には、クライアント側の CLASSPATH に相当) の解決の際に検索を行うスキーマのリストです。
- リゾルバ。サーバー内で相互参照するクラス・スキーマ・オブジェクト間のマッピングを維持します。

Java 名へのクラス・スキーマ・オブジェクトの外部参照のすべてがバインドされると、そのクラス・スキーマ・オブジェクトの解決が可能になります。通常、Java プログラムのすべてのクラスは、そのコンパイルまたはロードが終わるまで解決されません。これは、通常、再帰的に互いを参照できる複数のソース・ファイルで Java プログラムが記述されているためです。

サーバーの Java プログラムのすべてのクラス・スキーマ・オブジェクトが解決され、解決後にいずれも変更されていない場合、プログラムは事前リンクされて実行可能な状態になります。

クラス・スキーマ・オブジェクトが解決されるまで、クラスの Java オブジェクトのインスタンス化やクラスのメソッドの実行はできません。

注意： loadjava ユーティリティによってクラスへの参照は解決されますが、リソースへの参照は解決されません。（クライアント側での変換が必要な）ISO 標準コードの場合は、リソースをサーバーのリソース・スキーマ・オブジェクトにロードする方法に注意してください。この問題については、11-10 ページの「[ロードされたクラスとリソース・スキーマ・オブジェクトの名前付け](#)」で説明しています。クライアント側変換用の SQLJ -ser2class フラグを有効化すると、SQLJ プロファイルはクラス・ファイル中にロードされますが、リソース・ファイルは生成されません。この逆に -ser2class を有効化しなかった場合は、プロファイルが .ser リソース・ファイル中にロードされます。

SQL 名と Java 名

ソース名、クラス名、リソースのスキーマ・オブジェクト名などの SQL 名は、Java 名の場合とは異なり、グローバルではありません。Java 言語仕様では、パッケージ名にインターネットのネーミング規則を適用して、グローバルに一意な Java プログラム名を作成するように規定されています。対照的に、SQL の完全修飾名は、現行のスキーマおよびデータベースに関連してのみ解釈されます。たとえば、1 つのデータベースに SCOTT.FIZZ という名前のプログラムがあるとき、同じプログラムが他のデータベースでも SCOTT.FIZZ と示されるとは限りません。実際、あるデータベースの SCOTT.FIZZ から、他のデータベースの SCOTT.FIZZ をコールすることも可能です。

このような理由から、SQL 名は Java 名とは異なる方法で解釈および処理する必要があります。SQL 名は相対名であり、プログラムが実行されるスキーマに従って解釈されます。これは、プログラムがそのスキーマに格納されたローカル・データをバインドする際に重要です。Java 名はグローバル名であり、その名前が指定するクラスは、任意の実行サイトにロードできます。ただし、これらのクラスはプログラムのコンパイルに使用されたクラスであることを前提とします。

クライアント側での SQLJ ソースの変換とコンポーネントのロード

Oracle9i サーバーで使用する SQLJ コードを配置する方法の1つは、クライアント・マシン上で SQLJ トランスレータを実行し、変換、コンパイルおよびプロファイルのカスタマイズ(必要な場合)を行います。次に、通常はJava アーカイブ (.jar) ファイルを使用して、生成されたクラス・ファイルとリソース・ファイル(ある場合)をサーバーにロードします。サーバー側のトランスレータがサポートするのは Oracle 固有コード生成のみであるため、実際には、これがサーバー側で ISO 標準コードを使用する唯一の方法です。

クライアント・マシン上でソースを開発する場合、SQLJ トランスレータを使用できるときは、この方法をお勧めします。サーバー側でオプション設定やエラー処理を行うと手間がかかるため、クライアント・マシン上でトランスレータを実行した方が効果的です。

ISO 標準コード (-codegen=iso) の場合、アプリケーションをサーバーにロードするときは、変換時に SQLJ -ser2class オプションを使用することをお勧めします。このオプションを使用すると、SQLJ プロファイルが .ser シリアル化リソース・ファイルから .class ファイルに変換され、名前付けが簡略化されます。ただし、.class ファイルに変換されたプロファイルはそれ以上カスタマイズできないため、注意が必要です。さらにカスタマイズするには、トランスレータを再実行して、プロファイルを再生成する必要があります。-ser2class オプションの詳細は、8-64 ページの「[.ser ファイルから .class ファイルへの変換 \(-ser2class\)](#)」を参照してください。

.class ファイルや .ser リソース・ファイルを直接または .jar ファイルを使用して Oracle9i にロードすると、作成されるライブラリ・ユニットは、Java クラス・スキーマ・オブジェクト (Java クラス) および Java リソース・スキーマ・オブジェクト (Java リソース) とみなされます。SQLJ プロファイル(ある場合)は、リソース・スキーマ・オブジェクト (.ser ファイルをロードしたときと同様にロードした場合) またはクラス・スキーマ・オブジェクト (変換時に -ser2class を有効化した場合や、.class ファイルとしてロードした場合) の中にロードされます。

クラスおよびリソースのサーバーへのロード

クライアント上でトランスレータを実行した後、Oracle loadjava クライアント側ユーティリティを使用して、クラス・ファイルとリソース・ファイルをサーバーのスキーマ・オブジェクトにロードします。このユーティリティの詳細は、『Oracle9i Java Developer's Guide』を参照してください。

loadjava コマンドラインでクラス・ファイルとリソース・ファイル(ある場合)を個別に指定するか、両ファイルを .jar ファイルにまとめてからコマンドラインで .jar ファイルを指定します。.jar ファイルまたはコマンドラインで指定された .class または .ser ファイル用に、個別のスキーマ・オブジェクトが作成されます。

次のように実行する例について考えてみます。

1. ISO 標準コード生成を使用して、MyIter へのイテレータ宣言がある Foo.sqlj を変換しコンパイルします。
2. Foo.sqlj を変換するときに、-ser2class オプションを有効化します。
3. 結果ファイル (Foo.class、MyIter.class、Foo_SJProfileKeys.class および Foo_SJProfile0.class) を Foo.jar にアーカイブします。

コマンドラインで loadjava を実行します (必要なオプションも指定します)。次にデフォルトの OCI ドライバを使用した例を示します。

```
loadjava -user scott/tiger Foo.jar
```

または、次のように元のファイルを使用することも可能です。

```
loadjava -user scott/tiger Foo.class MyIter.class Foo_SJProfileKeys.class Foo_SJProfile0.class
```

または

```
loadjava -user scott/tiger Foo*.class MyIter.class
```

ロードに Thin ドライバを使用する場合は、次の例のように指定して、-thin オプションと適切な URL を指定します。

```
loadjava -thin -user scott/tiger@localhost:1521:ORCL Foo.jar
```

SQLJ トランスレータで生成されるファイルの詳細は、9-5 ページの「[コードの生成](#)」および 9-9 ページの「[Java コンパイル](#)」を参照してください。

注意：

- プロファイルの変換および生成時に -codegen=iso 設定を使用し、そのプロファイルを .ser ファイルとしてサーバーにロードすると、クライアント側でカスタマイズされていなかったプロファイルが最初に変換されます。プロファイルがすでにカスタマイズされている場合は、そのままロードされます。
 - スキーマの USER_OBJECTS ビューにアクセスすると、クラスとリソースが適切にロードされたかどうかを確認できます。この詳細は、『Oracle9i Java Developer's Guide』を参照してください。
-

SQLJ および Java アプリケーションのサーバーへのロードには、loadjava ユーティリティの使用をお勧めします。ただし、次のように Oracle SQL CREATE JAVA コマンドを使用する方法もあります。

```
CREATE OR REPLACE <AND RESOLVE> JAVA CLASS <NAMED name>;
```

```
CREATE OR REPLACE JAVA RESOURCE <NAMED name>;
```

CREATE JAVA コマンドの詳細は、『Oracle9i SQL リファレンス』を参照してください。

ロードされたクラスとリソース・スキーマ・オブジェクトの名前付け

ここでは、クラスやプロファイルをサーバー側にロードしたときの、この両方のスキーマ・オブジェクトの名前付けについて説明します。ただし、プロファイルが作成されるのは、ISO 標準コード生成（-codegen=iso）の場合のみです。

ISO 標準コード生成では、アプリケーションをクライアント側で変換したときに SQLJ -ser2class オプションが有効であった場合、プロファイルは .class ファイルに変換されているため、サーバーのクラス・スキーマ・オブジェクトにロードされます。-ser2class が有効になっていなかった場合、プロファイルは .ser シリアル化リソースとして生成されるため、サーバーのリソース・スキーマ・オブジェクトにロードされます。

この後の説明では、サーバーで実行されるすべてのアプリケーションについて、デフォルトの接続コンテキスト・クラスを使用することを前提とします。このため、プロファイルは1つのみになります。

フル・ネームとショート・ネーム

サーバー側のスキーマ・オブジェクト名には、フル・ネームとショート・ネームという2つの形式があります。

フル・ネームは完全修飾名であり、可能であるかぎりには常にスキーマ・オブジェクト名として使用されます。フル・ネームが 32 文字以上の場合、またはフル・ネームに不正な文字や変換できない文字が含まれている場合は、Oracle9i でフル・ネームがショート・ネームに変換され、スキーマ・オブジェクト名として使用されます。フル・ネームとショート・ネームおよびその変換方法は記録されます。フル・ネームが 31 文字以内で、不正な文字や変換できない文字が含まれていない場合は、スキーマ・オブジェクト名として使用されます。

DBMS_JAVA プロシージャで、ショート・ネームからフル・ネームを取得するときやその逆を実行するときの考慮事項を含む、スキーマ・オブジェクト名および他のファイルのネーミング規則に関する考慮事項の詳細は、『Oracle9i Java Developer's Guide』を参照してください。

ロードされたクラスのフル・ネーム

ISO 標準コード生成（-codegen=iso）を使用し、-ser2class フラグが有効な場合、ロードされるクラスにはプロファイル・ファイルが含まれます。

.class ファイルをサーバーにロードすると作成されるクラス・スキーマ・オブジェクトのフル・ネームは、元のソース・コード中のパッケージおよびクラス名によって決定されます。コマンドラインで指定する PATH（loadjava が認識できるように指定する場合など）や .jar ファイルで指定する PATH は、スキーマ・オブジェクトの名前付けには影響しません。たとえば、Foo.class が、ソース・コードでパッケージ x.y にあると指定された Foo

クラスで構成される場合、作成されるクラス・スキーマ・オブジェクトのフル・ネームは次のようになります。

```
x/y/Foo
```

「.class」が削除されていることに注意してください。

Foo.sqlj でイテレータ `MyIter` が宣言されている場合、そのクラス・スキーマ・オブジェクトのフル・ネームは、次のようになります（ネストされたクラスでない場合は、それ自体のスキーマ・オブジェクトはありません）。

```
x/y/MyIter
```

さらに、ISO 標準コード生成を使用している場合のフル・ネームは、次のようになります。

- Foo.sqlj の変換時に SQLJ で生成される関連プロファイルキーのクラス・ファイルは、`Foo_SJProfileKeys.class` です。したがって、そのクラス・スキーマ・オブジェクトのフル・ネームは、次のようになります。

```
x/y/Foo_SJProfileKeys
```

- アプリケーションへの変換時に `-ser2class` オプションが有効であった場合、プロファイルは `Foo_SJProfile0.class` ファイルに生成されています。したがって、クラス・スキーマ・オブジェクトのフル・ネームは、次のようになります。

```
x/y/Foo_SJProfile0
```

ロードされたリソース（`-ser2class` を有効にしなかった場合はプロファイルも含む）のフル・ネーム

ここで説明する内容が適用されるのは、ISO 標準コード生成（`-codegen=iso`）を使用していて、アプリケーションの変換時に `-ser2class` オプションが無効であった場合、またはアプリケーションで他の Java シリアル化リソース（`.ser`）ファイルを使用する場合のみです。

リソース・スキーマ・オブジェクトとクラス・スキーマ・オブジェクトでは、名前付けの方法が異なります。リソース・スキーマ・オブジェクトの名前は、リソースの内容には依存しません。そのフル・ネームには、`.jar` ファイルや `loadjava` コマンドラインに示される名前と同様に、`PATH` も含まれます。また、拡張子 `.ser` は削除されません。

リソース名は、実行時にリソースの場所を示すために使用されます。このため、リソース名には必ず正しい `PATH` を指定することが重要です。サーバーでは、Java がクライアント上でリソースを検索するときに使用する相対パスおよびファイル名が、リソースの正しいフル・ネームになります。

SQLJ プロファイルの場合、トランスレータの `-d` オプションで指定されたディレクトリ（パッケージ名をベースとした名前）の配下にあるサブディレクトリがフル・ネームになります。生成された `.class` ファイルや `.ser` ファイルの最上位の出力ディレクトリを指定するオプションです。`-d` オプションが `/mydir` に設定され、アプリケーションがパッケージ

abc.def にある場合は、変換で生成される .class ファイルおよび .ser ファイルが /mydir/abc/def ディレクトリに置かれます。デフォルト値を含む SQLJ の -d オプションの詳細は、8-29 ページの「[.ser および .class ファイルの出力ディレクトリ \(-d\)](#)」を参照してください。

実行時には、/mydir を CLASSPATH で指定することになります。Java ではこのディレクトリの下にある abc/def ディレクトリ中のアプリケーション・コンポーネントが検索されます。

このため、このアプリケーションをサーバーにロードするときは、loadjava または jar を -d ディレクトリから実行する必要があります。この場合、次の例のように、コマンドラインでファイルを検索するために指定する PATH にはパッケージ名を入力します。

```
cd /mydir
loadjava <...options...> abc/def/*.class abc/def/*.ser
```

.jar ファイルを使用する場合は、次のように指定します。

```
cd /mydir
jar -cvf myjar.jar abc/def/*.class abc/def/*.ser
loadjava <...options...> myjar.jar
```

アプリケーションが App で、プロファイルが App_SJProfile0.ser の場合、前述の例のいずれでも、作成されるリソース・スキーマ・オブジェクトのフル・ネームが次のように正しく設定されます。

```
abc/def/App_SJProfile0.ser
```

「.ser」が保持されていることに注意してください。

-d には、そのディレクトリ階層に他の内容が含まれないディレクトリを設定することをお勧めします。このように設定すると、jar を次のように実行して、アプリケーション・コンポーネントを再帰的に取得できます。

```
cd /mydir
jar -cvf myjar.jar *
loadjava <...options...> myjar.jar
```

クラス・ファイルとリソース・ファイルをロードした後のアプリケーションの公開

サーバー側で Java コードを使用する場合と同様に、サーバー側で SQLJ コードを使用する場合にも、事前に最上位メソッドを公開する必要があります。公開する場合は、コール記述子の記述、データ型のマッピング、パラメータ・モードの設定を行います。詳細は、『Oracle9i Java Stored Procedures Developer's Guide』を参照してください。

要約：サーバー側でのクライアント・アプリケーションの実行

ここでは、サーバーでクライアント・アプリケーションを実行する手順について要約します。例では、NamedIterDemo と呼ばれるデモ・アプリケーションを使用します。

1. アプリケーション・コンポーネントとして .jar ファイルを作成します。
NamedIterDemo のコンポーネントとしては、SalesRec.class の他、アプリケーション・クラスやプロファイル（ある場合）などが挙げられます。

.jar ファイル niter-server.jar を作成するには、次のようにします。

```
jar cvf niter-server.jar Named*.class Named*.ser SalesRec.class connect.properties
```

.ser ファイルは、ISO 標準コード生成の場合にのみ使用します。

2. .jar ファイルをサーバーにロードします。

loadjava は次のように指定します。この例に示したように指定した場合、loadjava でのファイルのロードには OCI ドライバが使用されます。-resolve オプションにより、クラス・ファイルが解決されます。

```
loadjava -oci -resolve -force -user scott/tiger niter-server.jar
```

3. アプリケーション用に、SQL ラッパーをサーバーに作成します。

たとえば、次の SQL*Plus スクリプトを実行します。

```
set echo on
set serveroutput on
set termout on
set flush on
```

```
execute dbms_java.set_output(10000);
```

```
create or replace procedure SQLJ_NAMED_ITER_DEMO as language java
name 'NamedIterDemo.main (java.lang.String[])';
/
```

DBMS_JAVA.SET_OUTPUT() ルーチンでは、デフォルトの出力が、トレース・ファイルではなくユーザー画面へ戻されます。入力パラメータには、バイト単位でバッファ・サイズを指定します。

4. ラッパーを実行します。

次にその例を示します。

```
sqlplus> call SQLJ_NAMED_ITER_DEMO();
```

SQLJ ソースのロードとサーバーでの変換

サーバーで使用する SQLJ コードを作成する場合には、ソース・コードをサーバーにロードして、サーバーで直接変換する方法もあります。この方法では、Oracle JVM の埋込み SQLJ トランスレータを使用します。ここでも、クライアント・マシン上でソースが作成されたことを前提とします。

注意： Oracle9i リリース 2 (9.2) では、サーバー側 SQLJ トランスレータによる ISO 標準コード生成 (-codegen=iso 設定) のサポートはありません。そのようなコードをサーバーで使用する場合、クライアントで変換して、個々のクラス・ファイルおよびリソースをサーバーにロードする必要があります。11-8 ページの「[クライアント側での SQLJ ソースの変換とコンポーネントのロード](#)」を参照してください。

サーバーへの SQLJ ソースのロードは、原則的にはサーバーへの Java ソースのロードと同じです。コンパイル・オプションが設定されている場合は、変換が暗黙的に実行されます（後述する `loadjava -resolve` オプションなど）。

.sqlj ソース・ファイルを .jar ファイルを使用するかまたは Oracle9i に直接ロードすると、その結果として生成されたライブラリ・ユニットにソース・ファイルが格納されます。これが、Java ソース・スキーマ・オブジェクトと呼ばれるライブラリ・ユニットです。ソース・ファイルごとに個別のスキーマ・オブジェクトが作成されます。

変換とコンパイルが行われると、生成されたクラスに対するライブラリ・ユニットは、Java クラス・スキーマ・オブジェクトと呼ばれます。これは、ライブラリ・ユニットがクライアント側で作成された .class ファイルからサーバーに直接にロードした場合と同じ名前です。各クラスに個別のスキーマ・オブジェクトが作成されます。

リソース・スキーマ・オブジェクトは、サーバーにロードしてあるプロパティ・ファイルに使用されます。

SQLJ ソース・コードのサーバーへのロード

ソースをサーバーにロードするには、(.class ファイルではなく) .sqlj ファイルに対して、Oracle `loadjava` クライアント側ユーティリティを実行します。このユーティリティの詳細は、『Oracle9i Java Developer's Guide』を参照してください。

`loadjava -resolve` オプションを有効にして .sqlj ファイルをロードすると、ロード時にサーバー側の埋込みトランスレータが実行され、アプリケーションの変換およびコンパイルが行われます。それ以外の場合は、変換が行われずにソースがソース・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、ソース中に定義されているクラスが最初に変更されたときに、ソースに対して暗黙的に変換およびコンパイルが行われます。クライアント側 SQLJ にはこの暗黙的変換に相当するものはありません。

たとえば、loadjava は、次のように実行します。

```
loadjava -user scott/tiger -resolve Foo.sqlj
```

ロードに Thin ドライバを使用する場合は、次のように -thin オプションと該当の URL を指定します。

```
loadjava -thin -user scott/tiger@localhost:1521:ORCL -resolve Foo.sqlj
```

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。詳細は、11-20 ページの「[ロードされたソース、生成されたクラスおよびリソース・スキーマ・オブジェクトの名前付け](#)」を参照してください。

ただし、loadjava を実行する際は、事前に SQLJ オプションを正しく設定する必要があります。詳細は、11-16 ページの「[サーバー側の埋込みトランスレータでサポートされるオプション](#)」を参照してください。encoding の設定は、次のように loadjava のコマンドラインで指定します。サーバー側 SQLJ encoding オプションは、このかわりとしては使用できないので注意してください。

```
loadjava -user scott/tiger -resolve -encoding SJIS Foo.sqlj
```

実際のユーティリティを実行する loadjava スクリプトの格納場所は、\$Oracle_Home ディレクトリの bin サブディレクトリです。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。

注意：

- .jar ファイルの処理では、loadjava によって最初に .sqlj ファイル、.java ファイルおよび .class ファイルが処理されます。次に、セカンド・パスが作成され、他のすべてが Java リソース・ファイルとして処理されます。
 - .sqlj ファイルは、同じ .sqlj ファイルの処理で生成された .class ファイルとともにロードすることはできません。このようにロードすると、サーバーで生成中のクラスやプロファイルと同じクラスに対してサーバーがロードを試みることになるため、競合が発生します。
 - 複数の .sqlj ファイルを .jar ファイルに入れて、.jar ファイルを loadjava に指定できます。
 - スキーマの USER_OBJECTS ビューにアクセスすると、クラスが適切にロードされていることを確認できます。この詳細は、『Oracle9i Java Developer's Guide』を参照してください。
-
-

SQLJ および Java アプリケーションのサーバーへのロードには、loadjava ユーティリティの使用をお勧めします。ただし、次のように Oracle SQL CREATE JAVA コマンドを使用する方法もあります。

```
CREATE OR REPLACE <AND COMPILE> JAVA SOURCE <NAMED srcname> <AS loadname>;
```

.sqlj ファイルに AND COMPILE を指定すると、その時点でのソースが変換およびコンパイルされ、ソース・スキーマ・オブジェクトの他に、必要に応じてクラス・スキーマ・オブジェクトが作成されます。それ以外の場合は、変換やコンパイルが行われず、ソース・スキーマ・オブジェクトのみが作成されます。この場合は、ソース内のクラスを最初に使用するとき、そのソースが暗黙的に変換およびコンパイルされます。

CREATE JAVA コマンドの詳細は、『Oracle9i SQL リファレンス』を参照してください。

注意： 初めてソース・ファイルをロードすると、定義済みクラスの確認など、ソース・コードのチェックが行われます。このときにエラーが検出されると、ロードが失敗します。

サーバー側の埋込みトランスレータでサポートされるオプション

サーバー側 SQLJ のトランスレータでは、次のオプションを使用できます。

- encoding
- online
- debug

ここでは、サーバー側 SQLJ での固定した設定を説明してから、これらのオプションについて説明します。

loadjava ユーティリティとその -resolve オプションについても説明します。詳細は、『Oracle9i Java Developer's Guide』を参照してください。

サーバー側 SQLJ トランスレータでの固定の設定

クライアント側の SQLJ トランスレータのオプションによってサポートされる次の設定は、サーバー側トランスレータでは固定です。

- サーバーのオンライン・セマンティクス・チェックとオフライン解析はともにデフォルトで有効です。これは、クライアントのデフォルトである -parse=both 設定に相当します。この設定を上書きし、オンライン・セマンティクス・チェックを online オプションを介して無効にできますが、オフライン解析を無効にすることはできません。これらの機能の詳細は、8-70 ページの「[オンライン・セマンティクス・チェックとオフライン解析との比較](#)」を参照してください。

- サーバー側では、Oracle 固有コード生成が使用されます。これは、クライアントのデフォルトである `-codegen=oracle` 設定に相当します。この設定は固定です。この機能の詳細は、4-36 ページの「[Oracle 固有コード生成（プロファイルなし）](#)」を参照してください。
- サーバー側での変換時に生成されたクラス・スキーマ・オブジェクトは、SQLJ ソース・コードをマッピング先とする行番号を自動的に参照します。クライアント側での変換時に `-linemap` オプションを有効化すると、サーバー側と同様の自動参照が行われます。このオプションの詳細は、8-48 ページの「[SQLJ ソース・ファイルとの行マッピング（`-linemap`）](#)」を参照してください。

encoding オプション

このオプションには、サーバー側にロードするときのソース・コードの解釈に使用するエンコーディング（SJIS など）を指定します。encoding オプションは、ソースのロード時にコンパイルの有無にかかわらず、使用されます。

また、loadjava を使用して SQLJ アプリケーションをサーバー側にロードする別の方法としては、loadjava コマンドラインでエンコーディングを指定する方法もあります。これについては、11-14 ページの「[SQLJ ソース・コードのサーバーへのロード](#)」で説明します。loadjava コマンドラインで設定された文字エンコーディングは、この encoding オプションに優先します。

このオプションの概要は、8-28 ページの「[入力および出力ソース・ファイルの文字エンコーディング（`-encoding`）](#)」を参照してください。

注意： このオプションまたは loadjava で文字エンコーディングが指定されていない場合は、loadjava を実行したクライアントの `file.encoding` で指定された文字エンコーディングが使用されます。

online オプション

online オプションに `true`（デフォルト値）を設定すると、オンライン・セマンティクス・チェックが有効になります。セマンティクス・チェックは、ソースがロードされたスキーマに対して行われます。クライアント側でのオンライン・チェックでは、基本スキーマの指定は不要です。

online オプションを `false` に設定すると、オフラインでチェックが行われます。

いずれの場合も、デフォルトのチェッカは `oracle.sqlj.checker.OracleChecker` です。このチェッカにより、使用する JDBC ドライバのバージョンと Oracle のバージョンに応じてチェッカが選択されます。OracleChecker の詳細は、8-68 ページの「[セマンティクス・チェックおよび OracleChecker フロントエンド（デフォルトのチェッカ）](#)」を参照してください。

online オプションは、ソースの変換およびコンパイル時に使用されます。loadjava `-resolve` オプションを有効にしてこれをロードすると、チェックが即座に実行されます。

それ以外の場合は、ソースで定義されたクラスが初めて使用されるときに実行されます。(変換およびコンパイルは暗黙的に行われます。)

注意： サーバー側とクライアント側とでは、`online` オプションの使用方法が異なります。サーバー側で `online` オプション使用した場合、デフォルトのチェッカでのオンライン・チェックを有効にするフラグとしてのみ機能します。一方、クライアント側では、使用するチェッカを `-online` オプションで指定できますが、別途 `-user` オプションでオンライン・チェックを有効にする必要があります。

debug オプション

このオプションに `true` を設定すると、`.sqlj` または `.java` ソース・ファイルがサーバーでコンパイルされるときに、サーバー側の **Java** コンパイラがデバッグ情報を出力します。これは、クライアント上で標準の `javac` コンパイラを実行するときに、`-g` オプションを使用するのに相当します。

`debug` オプションはソースのコンパイル時に使用されます。`loadjava -resolve` オプションを有効にしてこれをロードすると、デバッグが即座に実行されます。`.sqlj` ファイルの場合は、SQLJ 変換の直後に実行されます。それ以外の場合は、ソースで定義されたクラスが初めて使用されるときに実行されます。変換およびコンパイルは暗黙的に行われます。

サーバーでの SQLJ オプションの設定

サーバーで SQLJ トランスレータを実行するときは、コマンドラインもプロパティ・ファイルも存在しません。トランスレータやコンパイラのオプションについての情報は、`JAVA$OPTIONS` という名前の表の各スキーマに保持されます。この表のオプションを操作するには、`DBMS_JAVA` パッケージの次のファンクションおよびプロシージャを使用します。

- `dbms_java.get_compiler_option()`
- `dbms_java.set_compiler_option()`
- `dbms_java.reset_compiler_option()`

`dbms_java` パッケージおよびこれらの機能の詳細は、『Oracle9i Java パッケージ・プロシージャ・リファレンス』を参照してください。

個々のパッケージやリソースに対して個別のオプションを指定するには、`set_compiler_option()` を使用します。このファンクションには、次を入力パラメータとして指定します。その場合、各パラメータを引用符で囲みます。

- パッケージ名（ドット区切りの名前）またはソース名

ショート・ネームではなくフル・ネームで指定します。

パッケージ名を指定すると、そのパッケージのすべてのソースとサブパッケージに、そのオプションが適用されます。ただし、特定のサブパッケージやソースについて設定を上書きした場合は除きます。

- オプション名
- オプション設定

SQL*Plus を使用して、DBMS_JAVA ルーチンを実行します。たとえば、次のように指定します。

```
sqlplus> execute dbms_java.set_compiler_option('x.y', 'online', 'true');
sqlplus> execute dbms_java.set_compiler_option('x.y.Create', 'online', 'false');
```

これら 2 つのコマンドで、**x.y** パッケージ内のすべてのソースに対するオンライン・チェックが有効になります。次に、**Create** ソースのオンライン・チェックを無効し、**Create** ソースに対してチェックが行われないようにします。

同様に、次のように指定して、パッケージ **x.y** のエンコーディングを **SJIS** に設定します。

```
sqlplus> execute dbms_java.set_compiler_option('x.y', 'encoding', 'SJIS');
```

サーバー側オプションについての注意事項 次の点に注意してください。

- スキーマ・オブジェクト名ではスラッシュ構文（パッケージ名としての **abc/def** など）を使用しますが、パッケージ名やソース名の **set_compiler_option()** パラメータはドット区切りの名前（パッケージ名としての **abc.def** など）を使用します。
- パッケージ名を指定するときは、サブパッケージにもオプションが適用されることに注意してください。**a.b.MyPackage** の設定では、次の形式の名前の付いたソース・スキーマ・オブジェクトすべてに対して、オプションが設定されます。

```
a/b/MyPackage/subpackage/...
```

- パッケージ名として **''**（スペースを挿入せずに一重引用符 2 つ）を指定した場合、ルートおよびすべてのサブパッケージに対して、オプションが適用されます。結果的には、スキーマ中のすべてのパッケージにオプションが適用されることになります。

ロードされたソース、生成されたクラスおよびリソース・スキーマ・オブジェクトの名前付け

-resolve オプションを有効にして loadjava を .sqlj ファイルに使用するときなど、サーバー側の SQLJ トランスレータを使用する場合、基本的には、クライアント側で生成される内容と同じものが生成されます。つまり、ソース中に定義した各クラスのコンパイル済みクラスおよび各イテレータや接続コンテキスト・クラスのコンパイル済みクラスが生成されます。

このため、loadjava で .sqlj ファイルをサーバーにロードして、変換およびコンパイルを行うと、次に示した各スキーマ・オブジェクトが作成されます

- 元のソース・コード用のソース・スキーマ・オブジェクト
- ソースで定義したすべてのクラス用のクラス・スキーマ・オブジェクト
- ソースで定義した各イテレータまたは接続コンテキスト・クラス用のクラス・スキーマ・オブジェクト

ただし、ユーザー定義型に型マップを指定するためでない場合は、サーバーで実行するコードに接続コンテキスト・クラスを宣言する必要はありません。(6-12 ページの「[SQLData を実装したクラスに対する要件](#)」を参照してください。)

これらのスキーマ・オブジェクトのフル・ネームは、後述の方法で決定されます。作成されるスキーマ・オブジェクトとその名前を出力するには、loadjava -verbose オプションを指定します。

注意： サーバー側のスキーマ・オブジェクト名には、フル・ネームとショート・ネームという 2 つの形式があります。11-10 ページの「[フル・ネームとショート・ネーム](#)」を参照してください。

ソースのフル・ネーム

ソース・ファイルをサーバーにロードすると、変換やコンパイルが行われるかどうかにかかわらず、ソース・スキーマ・オブジェクトが作成されます。このスキーマ・オブジェクトのフル・ネームは、ソース・コード中のパッケージ名とクラス名によって決定されます。コマンドラインで loadjava に対して指定する PATH は、スキーマ・オブジェクトの名前付けには影響しません。

たとえば、Foo.sqlj が x.y パッケージ内の Foo クラスを定義し、他のクラスを定義または宣言しない場合、作成されるソース・スキーマ・オブジェクトのフル・ネームは次のようになります。

x/y/Foo

「.sqlj」が削除されていることに注意してください。

クラスをさらに定義したり、イテレータ・クラスや接続コンテキスト・クラスを宣言すると、ソース・スキーマ・オブジェクトの名前が付けられます。この名前には、通常は最初に出現した **Public** クラス定義またはクラス宣言が使用されますが、**Public** クラスの定義がない場合は、最初のクラス定義が使用されます。サーバー側では、1 つのソースの中に複数のクラスを定義することも可能です。

前の例と同じパッケージ `x.y` 中の `Foo.sqlj` に、まず **Public** クラス `Bar` を、次にクラス `Foo` を定義したとします。この `Bar` の定義の前に **public** イテレータ・クラスまたは接続コンテキスト・クラスを宣言しなかった場合には、作成されるソース・スキーマ・オブジェクトのフル・ネームは次のようになります。

```
x/y/Bar
```

ただし、`Bar` クラスと `Foo` クラスの定義の前に、**public** イテレータ・クラス `MyIter` が宣言されている場合、作成されるソース・スキーマ・オブジェクトのフル・ネームは次のようになります。

```
x/y/MyIter
```

生成されたクラスのフル・ネーム

クラス・スキーマ・オブジェクトは、ソース中に定義されているクラス、宣言済みイテレータおよびプロファイルキー・クラスのそれぞれに対して生成されます。クラス・スキーマ・オブジェクトは、ソース・コードのクラス名およびパッケージ名に基づいて名前付けされます。

ここでも、前述の「ソースのフル・ネーム」の例を使用します。ソース・コード中にパッケージ `x.y` を指定し、最初に **Public** クラス `Bar`、次にクラス `Foo` を定義した後で、**public** イテレータ・クラス `MyIter` を宣言したとします。定義および宣言されたクラスのクラス・スキーマ・オブジェクトには、次のようにフル・ネームが付けられます。

```
x/y/Bar
```

```
x/y/Foo
```

```
x/y/MyIter
```

「`.class`」が付けられていないことに注意してください。

注意： ソース名は、最初に定義されている **Public** クラスの名前と同じにするか、**Public** クラスが未定義の場合は、最初に定義されているクラスの名前と同じにすることをお勧めします。このようにすると、クライアント側とサーバー側で違う動作をすることを回避できます。

元のソース・ファイルの名前や、ソースのサーバーへのロード時に指定する `PATH` は、生成されるクラスの名前付けには影響しません。

コードで内部クラスや無名クラスを定義する場合、これらのクラスは標準の `javac` コンパイラのネーミング規則に従って名前付けされます。

サーバー側埋込みトランスレータからのエラー出力

サーバー側の SQLJ エラー処理は、サーバー側での一般的な Java エラー処理と同様に行われます。SQLJ エラーは、ユーザー・スキーマの `USER_ERRORS` 表に出力されます。この表の `TEXT` 列から `SELECT` を行うと、特定のエラー・メッセージを取得できます。

ただし、SQLJ ソースのロードに `loadjava` を使用する場合は、`loadjava` によってサーバー側トランスレータからのエラー・メッセージのキャプチャおよび出力が行われます。

情報メッセージや非表示にできる警告は、サーバー側トランスレータで出力しないようにできます。これは、クライアント側トランスレータで `-warn=noportable,noverbose` (デフォルト) が設定された場合と同じです。これらの設定の意味は、8-44 ページの「[トランスレータからの警告 \(-warn\)](#)」を参照してください。

ソース・ファイルをロードした後の、アプリケーションの公開

サーバー側で Java コードを使用する場合と同様に、サーバー側で SQLJ コードを使用する場合にも、事前に最上位メソッドを公開する必要があります。公開する場合は、コール記述子の記述、データ型のマッピング、パラメータ・モードの設定を行います。詳細は、『[Oracle9i Java Stored Procedures Developer's Guide](#)』を参照してください。

Java スキーマ・オブジェクトの削除

Oracle には、`loadjava` ユーティリティを補足する `dropjava` ユーティリティがあります。このユーティリティは、Java ソース、クラスおよびリソース・スキーマ・オブジェクトを削除 (ドロップ) します。`loadjava` を使用してサーバーにロードしたスキーマ・オブジェクトを削除する場合は、必ず `dropjava` を使用してください。ここでは `dropjava` の概要のみに絞って説明します。この詳細は、『[Oracle9i Java Developer's Guide](#)』を参照してください。

`dropjava` ユーティリティでは、コマンドラインのファイル名と `.jar` ファイルの内容がスキーマ・オブジェクト名に変換され、スキーマ・オブジェクトが削除されます。`.sqlj`、`.java`、`.class`、`.ser` および `.jar` の各ファイルは、任意の順序でコマンドラインに指定できます。

Java スキーマ・オブジェクトは、必ず最初にロードしたときと同じ方法で削除することをお勧めします。`.sqlj` ソース・ファイルをロードしてからサーバー側で変換した場合は、同じソース・ファイルに対して `dropjava` を実行します。クライアント側で変換を行ってからクラスとリソースを直接ロードした場合は、同じクラスとリソースに対して `dropjava` を実行します。

たとえば、`Foo.sqlj` に対して `loadjava` を実行した場合は、次のように同じファイル名を指定して `dropjava` を実行します。

```
dropjava -user scott/tiger Foo.sqlj
```

プログラムをクライアント側で変換してから、生成されたコンポーネントを含む .jar ファイルでロードした場合は、同じ .jar ファイル名を使用してプログラムを削除します。

```
dropjava -user scott/tiger Foo.jar
```

プログラムをクライアント側で変換してから、生成されたコンポーネントを loadjava コマンドラインでロードした場合は、次のように dropjava コマンドラインで削除します。この例では、-codegen=oracle で、イテレータ・クラスがないことを想定しています。

```
dropjava -user scott/tiger Foo*.class
```

その他の考慮事項

ここでは、サーバーでの Java マルチスレッドと再帰的 SQLJ コールについて説明します。

サーバーでの Java マルチスレッド

Java マルチスレッドを使用するプログラムは、変更を加えなくても Oracle9i で実行できます。ただし、クライアント側プログラムがマルチスレッドの使用によりスループットが向上するのに対して、サーバー側で Java マルチスレッド・コードが実行される場合はそのようなメリットがありません。マルチスレッド・アプリケーションをサーバーに移植する場合は、次に示す Oracle JVM とクライアント側 JVM とのマルチスレッド機能の相違を考慮してください。

- サーバー内のスレッドは、同時処理ではなく順次処理されます。
- サーバーでは、1つのコール内のスレッドは、そのコールの終わりに消滅します。
- サーバー内のスレッドは、プリエンプティブにスケジュールされません。1つのスレッドが無限ループに入った場合、他のスレッドは実行できません。

Oracle9i サーバーの Java マルチスレッドを、通常の Oracle サーバーのマルチスレッドと混同しないでください。後者の場合は Oracle の同時セッションであり、Java マルチスレッドではありません。サーバーでは、多くのユーザーが自身のセッションで同時に実行することにより、拡張性とスループットが得られます。スループットを最大にするための Java の実行のスケジューリング（1セッション内でのコールのスループットなど）は、Java ではなく Oracle サーバーで行われます。

SQLJ での Java マルチスレッドの概要は、7-31 ページの「[SQLJ でのマルチスレッド](#)」を参照してください。

サーバーでの再帰的 SQLJ コール

7-25 ページの「[実行コンテキストの同期](#)」で説明したように、SQLJ では通常、複数の SQLJ 文で同時には同じ実行コンテキスト・インスタンスを使用できません。つまり、すでに使用されている実行コンテキスト・インスタンスの使用を試みた文は、前の文が完了するまでブロックされます。

ただし、クライアントに比べて、Oracle サーバーではこの機能はあまり望ましくありません。すべてのストアド・プロシージャおよびストアド・ファンクションは通常デフォルトの実行コンテキスト・インスタンスを使用します。ただし、異なるストアド・プロシージャやストアド・ファンクションが、再帰的な状況で同じ実行コンテキスト・インスタンスを同時に使用することがあります。たとえば、あるストアド・プロシージャが SQLJ 文を使用して、SQLJ 文を使用する他のストアド・プロシージャをコールする可能性があります。これらのストアド・プロシージャを最初に作成するときには、いつこうした状況になるかはわからないため、SQLJ 文に対して特定の実行コンテキスト・インスタンスが指定されるとは断定できません。

この問題に対処するため、再帰的コールによってこのような状況が発生した場合、SQLJ では複数の SQLJ 文で同時に同じ実行コンテキスト・インスタンスを使用できます。

ここでは、再帰的な状況の例を考えて、実行コンテキスト・インスタンスのステータス情報がどのようになるかを説明します。すべての文で、デフォルトの接続コンテキスト・インスタンスとそのデフォルトの実行コンテキスト・インスタンスが使用されると想定します。ストアド・プロシージャ `proc1` が持つ SQLJ 文が、同じく SQLJ 文を持つストアド・プロシージャ `proc2` をコールする場合、`proc2` 内の各文は、`proc1` のプロシージャ・コールが使用しているときに同じ実行コンテキスト・インスタンスを使用します。

`proc2` の各 SQLJ 文によって、その文のステータス情報が実行コンテキスト・インスタンスに書き込まれます。必要な場合は、文のコンパイル後に、その情報を取得できます。`proc2` をコールする `proc1` の文からのステータス情報は、`proc2` が実行を終え、プログラム・フローが `proc1` に戻り、`proc2` をコールした `proc1` 内の操作が完了した後で、実行コンテキスト・インスタンスに書き込まれます。

再帰的な状況でも実行コンテキストのステータス情報が正しく処理されるように、実行コンテキスト・メソッドは、SQL 操作が完了した後にステータス情報を更新するように定義されています。

注意：

- サーバー側で実行されるコードの中に、実行コンテキスト・ステータスや制御メソッドを使用する計画を立てるたびに、必要に応じて別個の実行コンテキスト・インスタンスを使用することをお勧めします。
 - 上の例で別個の実行コンテキスト・インスタンスを使用せず、しかも制御パラメータを変更するために `proc2` から実行コンテキスト・インスタンスへのメソッドをコールした場合、それ以降に `proc1` で実行される操作に影響します。
 - バッチ更新は、再帰的コールでは使用できません。デフォルトでは、バッチ処理（有効化してある場合）を実行できるのは、最上位プロシージャのみに限定されています。こうした制限を免れるには、実行コンテキスト・インスタンスを明示的に使用する必要があります。
-
-

ExecutionContext メソッドの詳細は、7-25 ページの「[実行コンテキストのメソッド](#)」を参照してください。

サーバーでのコードの実行状況の確認

作成したコードが実際にサーバー側で実行可能かどうかを確認する場合、`java.lang.System` クラスの `static getProperty()` メソッドを使用して `oracle.server.version` の Java プロパティを取得する方法が便利です。このプロパティにバージョン・ナンバーが指定されている場合、コードは Oracle サーバーで実行されます。このバージョン番号が `NULL` の場合、Oracle サーバーではコードが実行されません。次にその例を示します。

```
...
if (System.getProperty("oracle.server.version") != null)
{
    // (running in server)
}
...
```

注意： サーバー側で `getProperties()` メソッドは使用しないでください。セキュリティ例外の原因となります。

カスタマイズおよび専用カスタマイザ

プロファイルとプロファイルのカスタマイズについては、1-5 ページの「[SQL プロファイル \(ISO 標準コードの場合\)](#)」で簡単に説明しました。この付録では、技術的に詳しく説明します。また、カスタマイザ・オプションと、デフォルトの Oracle カスタマイザ以外のカスタマイザの使用方法を示します。

また、Oracle の専用カスタマイザについても、特にプロファイルのセマンティクス・チェック用の SQLCheckerCustomizer、およびデバッグに使用する「オーディタ」のインストール用の AuditorInstaller を中心に説明します。

この章では、次の項目について説明します。

- [プロファイルについて](#)
- [プロファイルのカスタマイズについて](#)
- [カスタマイズ・オプションとカスタマイザの選択](#)
- [プロファイルの JAR ファイル](#)
- [プロファイルのセマンティクス・チェック用の SQLCheckerCustomizer](#)
- [デバッグ用の AuditorInstaller カスタマイザ](#)

重要： この付録の説明は、デフォルトの Oracle 固有コード生成 (-codegen=oracle) を利用するアプリケーションには関係がありません。

プロファイルについて

SQLJ プロファイルには、埋込み SQL 操作に関する情報が含まれています。プロファイルは、アプリケーションで使用される接続コンテキスト・クラスごとに、個別に生成されます。プロファイルは、SQLJ トランスレータのコード生成フェーズで生成され、カスタマイズ・フェーズでカスタマイズされます。カスタマイズが行われると、アプリケーションでベンダー固有のデータベース機能を使用できるようになります。これらのベンダー固有の操作を別々にプロファイルにまとめると、それ以外の部分を汎用コードとして使用できます。

各プロファイルには、関連する接続コンテキスト・クラスを使用する SQLJ 文に対する一連のエントリが含まれています。各エントリは、アプリケーション内の 1 つの SQL 操作に対応しています。

プロファイルは、アプリケーションとパッケージ化されるリソース・ファイルに、シリアル化されたオブジェクトとして保存されます。そのため、プロファイルは、随時にロード、読み込みおよび変更（追加または再カスタマイズ）ができます。プロファイルのカスタマイズ時に、情報は追加されますが、削除されることはありません。プロファイルは、複数回カスタマイズできます。その場合、以前にカスタマイズされたプロファイルは変更されません。そのため、アプリケーションを各種の環境で実行できます。これをバイナリ移植性と呼びます。

プロファイルにバイナリ移植性を持たせるために、Oracle SQLJ は SQLJ の業界標準に従って実装されています。

コード生成時のプロファイルの生成

トランスレータは、コード生成時に次のようにプロファイルを生成します。

1. `sqlj.runtime.profile.Profile` クラスのインスタンスとして、プロファイル・オブジェクトを生成します。
2. 生成されたプロファイル・オブジェクトに、埋込み SQL 操作（関連する接続コンテキスト・クラスを使用する SQLJ 文）に関する情報を挿入します。
3. プロファイル・オブジェクトをシリアル化して、Java リソース・ファイルに保存します。このファイルはプロファイル・ファイルと呼ばれ、拡張子として `.ser` が付加されます。

注意： Oracle SQLJ には、トランスレータに自動的にこれらの `.ser` ファイルを `.class` ファイルに変換させるオプションがあります。`.ser` ファイルをサポートしていないブラウザがあるため、変換されたアプリケーションをサーバーにロードするときに、`.ser` ファイルのロードに手間が必要なためです。ただし、いったん自動変換したプロファイルは、カスタマイズの対象にならなくなります。詳細は、8-64 ページの「[.ser ファイルから .class ファイルへの変換 \(-ser2class\)](#)」を参照してください。

9-5 ページの「[コードの生成](#)」で説明したように、アプリケーション Foo のプロファイル・ファイル名は、次のようになります。

```
Foo_SJProfilen.ser
```

SQLJ では、Foo_SJProfile0.ser、Foo_SJProfile1.ser などのファイルが、コード中に使用した接続コンテキスト・クラスの数に応じて生成されます。-ser2class オプションが設定されている場合は、Foo_SJProfile0.class、Foo_SJProfile1.class などのファイルが生成されます。

各プロファイルには、SQLJ の実行時にコールされる `getConnectedProfile()` メソッドが含まれています。このメソッドは、JDBC の `Connection` オブジェクトに相当するものを戻します。ただし、そのオブジェクトには機能が追加されています。詳細は、A-9 ページの「[カスタマイズされたプロファイルの実行時の機能](#)」を参照してください。

注意： プロファイル・オブジェクトとは、シリアル化される前の元の状態のプロファイルを示します。プロファイル・ファイルとは、シリアル化された状態（.ser ファイル）を示します。

プロファイル・エントリの例

ここでは、SQLJ 実行文と、生成されるプロファイル・エントリの例を示します。ここに示すプロファイル・エントリは、わかりやすいように関係のない部分は省略して、テキストで表記してあります。

プロファイル・エントリでは、ホスト変数が JDBC 構文（疑問符）に置き換えられることに注意してください。

SQLJ 実行文

次のような宣言があるとします。

```
#sql iterator Iter (double sal, String ename);
```

また、次のような実行文があるとします。

```
String empname = 'Smith';
Iter it;
...
#sql it = { SELECT ename, sal FROM emp WHERE ename = :empname };
```

対応する SQLJ プロファイル・エントリ

```
=====
...
#sql { SELECT ename, sal FROM emp WHERE ename = ? };
...
PREPARED_STATEMENT executed via EXECUTE_QUERY
role is QUERY
descriptor is null
contains one parameter
1. mode: IN, java type: java.lang.String (java.lang.String),
   sql type: VARCHAR, name: ename, ...
result set type is NAMED_RESULT
result set name is Iter
contains 2 result columns
1. mode: OUT, java type: double (double),
   sql type: DOUBLE, name: sal, ...
2. mode: OUT, java type: java.lang.String (java.lang.String),
   sql type: VARCHAR, name: ename, ...
=====
```

注意： このプロファイル・エントリは、わかりやすいようにテキストで表記してありますが、実際のエントリはテキスト・フォーマットではありません。ただし、SQLJ の `-p-print` オプションを使用すると、テキストとして出力できます。詳細は、A-11 ページの「[カスタマイザ・ハーネスのオプションの概要](#)」を参照してください。

プロファイルのカスタマイズについて

デフォルトでは、SQLJ ソース・ファイルに対して `sqlj` スクリプトを実行すると、自動的にカスタマイズが行われます。その場合、コード生成時に生成された各プロファイルが、使用するデータベースにあわせてカスタマイズされます。デフォルトの `oracle.sqlj.runtime.OraCustomizer` という Oracle カスタマイザを使用すると、Oracle9i 固有の型拡張およびパフォーマンス強化機能を利用できるように、プロファイルが最適化されます。

`sqlj` スクリプトを実行して、既存のプロファイルのカスタマイズすることも可能です。SQLJ コマンドラインで、個々の `.ser` ファイル、`.ser` ファイルを含む `.jar` ファイル、またはその両方を指定できます。

注意：

- デフォルトの Oracle カスタマイザをトランスレータ実行時に使用する場合は、その実行時に Oracle SQLJ ランタイムおよび Oracle JDBC ドライバがアプリケーションで必要になります。これらは、Oracle の拡張型をコード中に使用しない場合にも必要です。
 - アプリケーションでカスタマイズ（接続のための処理）を行わない場合には、汎用 SQLJ ランタイムが使用されます。
 - SQLJ で一度に実行できるのは、.sqlj ファイルや .java ファイルに対する処理（変換、コンパイルおよびカスタマイズの場合）、または .ser ファイルや .jar ファイルに対する処理（カスタマイズのみの場合）のいずれか一方です。
-

カスタマイザ・ハーネスおよびカスタマイズの概要

Oracle カスタマイザを使用する場合も、その他のカスタマイザを使用する場合も、SQLJ のカスタマイズには、カスタマイザ・ハーネスと呼ばれるフロントエンドのカスタマイズ・ユーティリティが使用されます。

SQLJ の実行時に指定できるカスタマイズ・オプションとしては、カスタマイザ・ハーネス用（あらゆるカスタマイザに適用できる一般的なカスタマイザ設定）およびカスタマイザ用（カスタマイザ固有の設定）があります。これらのオプションは、コマンドラインまたはプロパティ・ファイルで指定します。詳細は、A-10 ページの「[カスタマイズ・オプションとカスタマイザの選択](#)」を参照してください。

カスタマイザは、プロパティを公開するために、標準の JavaBeans API に適合する JavaBeans コンポーネントである必要があります。また、customize() メソッドを指定している sqlj.runtime.profile.util.ProfileCustomizer インタフェースを実装する必要があります。カスタマイザ・ハーネスは、カスタマイズするプロファイルごとに、カスタマイザ・オブジェクトの customize() メソッドをコールします。

Oracle カスタマイザは前述の要件を満たしています。このカスタマイザは oracle.sqlj.runtime.OraCustomizer クラスに定義されています。

カスタマイズ処理の手順

変換時のカスタマイズ処理は、次のように行われます。これは、SQLJ の全体的な実行におけるカスタマイズ・フェーズの場合も、既存のプロファイルのみをカスタマイズする場合も同じです。

1. SQLJ は、カスタマイザ・ハーネスを起動して、指定されているカスタマイズの汎用オプションをそのカスタマイザ・ハーネスに渡します。
2. カスタマイザ・ハーネスは、使用するカスタマイザを起動して、指定されているカスタマイザ固有のオプションをそのカスタマイザに渡します。

3. カスタマイズのみを行うために、コマンドラインで `.jar` ファイルを指定して `SQLJ` を実行した場合、カスタマイザ・ハーネスは、これらの `.jar` ファイル内のプロファイル・ファイルを検出して抽出します。
4. カスタマイザ・ハーネスは、各プロファイル・ファイルをプロファイル・オブジェクトにシリアル化解除します（対象は、`SQLJ` の全体的な実行で自動的に生成される `.ser` ファイル、カスタマイズのみを行うためにコマンドラインで指定した `.ser` ファイル、またはカスタマイズのみを行うために、コマンドラインで指定した `.jar` から抽出された `.ser` ファイルです）。
5. データベース接続が必要なカスタマイザを使用している場合、カスタマイザ・ハーネスが接続を確立します。
6. カスタマイザ・ハーネスは、プロファイルごとに手順 2 でインスタンス化されたカスタマイザ・オブジェクトの `customize()` メソッドをコールします（カスタマイザを `OracleSQLJ` で使用するには、`customize()` メソッドが必要です）。
7. 通常、`customize()` メソッドは、各プロファイルの内容をカスタマイズして、そのカスタマイズ内容を同じプロファイルに登録します。ただし、これはカスタマイザで処理する内容に依存します。カスタマイザによっては、特別な処理を行うために、このようなカスタマイズと登録を必要としない場合があります。
8. カスタマイザ・ハーネスは、各プロファイルを再度シリアル化して `.ser` ファイルを生成します。
9. カスタマイズのみを行うために、コマンドラインで `.jar` ファイルを指定して `SQLJ` を実行した場合、カスタマイザ・ハーネスは、カスタマイズされた各 `.ser` ファイルで、カスタマイズ前の元の `.ser` ファイルを上書きし、`.jar` の内容を再度生成します。

注意：

- プロファイルのカスタマイズ時にエラーが発生した場合、元の `.ser` ファイルは上書きされません。
 - `.jar` ファイル内のプロファイルのカスタマイズ時にエラーが発生した場合、元の `.jar` ファイルは上書きされません。
 - `SQLJ` では、一度に複数のカスタマイザを実行することは不可能です。1 つのプロファイルに対して複数回カスタマイズを行うには、`SQLJ` を複数回実行する必要があります。追加でカスタマイズを行う場合は、`SQLJ` のコマンドラインでプロファイル名を直接入力します。
-

プロファイルのカスタマイズと登録

カスタマイザ・ハーネスは、`customize()` メソッドをコールして、プロファイルのカスタマイズします。その場合、このメソッドにプロファイル・オブジェクト、SQLJ 接続コンテキスト・オブジェクト（接続が必要なカスタマイザを使用している場合）およびエラー・ログ・オブジェクト（カスタマイズ時に出力されたエラー・メッセージのログに使用）を渡します。

1 回の SQLJ の実行では、すべてのカスタマイズに同じエラー・ログ・オブジェクトが透過的に使用されます。カスタマイザ・ハーネスは、エラー・ログ・オブジェクトに書き込まれたメッセージを読み込み、標準出力（通常はディスプレイ）にリアルタイムで出力します。

各プロファイルには複数のエントリが含まれており、それぞれ SQL 操作に対応しています。（これらはアプリケーション内の SQL 操作であり、そのプロファイルに対応付けられている接続コンテキスト・クラスのインスタンスが使用されます。）

`customize()` は、これらのエントリに対して特別な処理を行います。各エントリの構文の妥当性チェックのような簡単な処理の場合もあれば、元のエントリに特定のデータベースの機能を使用するための変更を加えて、新しいエントリを生成するような複雑な処理の場合もあります。

注意：

- `customize()` でプロファイル・エントリを処理しても、元のエントリが変更されることはありません。
 - 特定の環境で使用するためにプロファイルのカスタマイズしても、他の環境でアプリケーションを実行できなくなることはありません。各種の環境で使用できるようにするには、プロファイルを複数回カスタマイズします。これらのカスタマイズが他のカスタマイズに影響することはありません。
-
-

実装の詳細 ここでは、Oracle でのカスタマイズ処理について説明します。これらの情報はほとんどの SQLJ 開発者には必要ありません。

Oracle カスタマイザの場合、`customize()` メソッドは、元のプロファイルの各エントリと対応するエントリで構成されるデータ構造を生成します。元のエントリが変更されるのではなく、新しく生成されたエントリが Oracle9i の機能を利用できるようにカスタマイズされます。たとえば、BLOB が使用されている場合は、元のエントリで BLOB を取り出すために使用されている汎用の `getObject()` コールが、`getBLOB()` コールに置き換えられます。

これらの新しいエントリは、`sqlj.runtime.profile.Customization` インタフェースを実装したカスタマイズ・クラスのオブジェクトにカプセル化されます。そのカスタマイズ・オブジェクトが、プロファイル・オブジェクトに挿入されます。プロファイル・オブジェクトと同様に、カスタマイズ・オブジェクトもシリアル化可能です。

次に、カスタマイザ・ハーネスは、プロファイル・オブジェクトの機能を使用して、カスタマイズ内容を登録します。この処理により、プロファイルに対して行われたカスタマイズ内容の履歴が記録されます。

カスタマイズ時に発生したエラーはエラー・ログに書き込まれ、カスタマイザ・ハーネスによって適宜通知されます。

Customization オブジェクトには、`acceptsConnection()` メソッドが定義されています。特定の SQLJ 接続コンテキスト・オブジェクトの接続プロファイル・オブジェクトがカスタマイズで生成されるかどうかを判断する場合、このメソッドを実行時にコールします。接続プロファイル・オブジェクトは、`sqlj.runtime.profile.ConnectedProfile` インタフェースを実装したクラスのインスタンスです。このオブジェクトは、プロファイル・オブジェクトと JDBC 接続間のマッピングを示します。これは JDBC の `Connection` オブジェクト（接続コンテキスト・オブジェクトの基になる）に相当し、文を生成します。その他に、ベンダー固有の機能を実装します。

カスタマイズ時のエラー・メッセージとステータス・メッセージ

カスタマイザ・ハーネスでのエラー・メッセージとステータス・メッセージは、SQLJ トランスレータと同じ出力デバイスに出力されます。ただし、カスタマイズに関する警告メッセージは、非表示にはできません。

カスタマイザ・ハーネスによって生成されるエラー・メッセージは、次の 4 つのカテゴリに分類できます。

- 認識できないオプションまたは不正なオプション
- 接続のインスタンス化エラー
- プロファイルのインスタンス化エラー
- カスタマイザのインスタンス化エラー

カスタマイズ時に、カスタマイザ・ハーネスによって、ステータス・メッセージが生成されます。これらのメッセージを参照して、プロファイルが正しくカスタマイズされたかどうかを確認できます。ステータス・メッセージは、次の 3 つのカテゴリに分類できます。

- プロファイルの変更状態
- .jar ファイルの変更状態
- 作成されたバックアップ・ファイル名（カスタマイザ・ハーネスの `-backup` オプションが設定されている場合）

この他にも、特定のカスタマイザの `customize()` のメソッドによって、カスタマイザ固有のエラーや警告が出力される場合があります。

カスタマイズ時に、プロファイル・カスタマイザによって、メッセージがエラー・ログに書き込まれます。その後、カスタマイザ・ハーネスによって、そのログ内容がリアルタイムで読み込まれます。最後に、ハーネスからの他の出力とあわせて、そのメッセージが SQLJ の出力デバイスに出力されます。ユーザーがエラー・ログの内容に直接アクセスする必要はありません。

カスタマイズされたプロファイルの実行時の機能

カスタマイズされたプロファイルは、それが対応付けられている接続コンテキスト・クラスの静的なメンバーです。SQLJ ランタイムでは、アプリケーション内の各 SQLJ 文に対して、その文が対応付けられている接続コンテキスト・クラスとインスタンスが確認されます。次に、その接続コンテキスト・クラスのカスタマイズされたプロファイルと、特定の接続コンテキスト・インスタンスの基になる JDBC 接続を使用して、接続プロファイルが生成されます。この接続プロファイルは SQLJ ランタイムで使用すると、SQLJ アプリケーションの実行時にベンダー固有の機能を利用できるようになります。

実装の詳細 ここでは、Oracle SQLJ ランタイムでのカスタマイズ済みプロファイルの利用について説明します。これらの情報はほとんどの SQLJ 開発者には必要ありません。

SQLJ ランタイムでは、SQLJ 文の実行時に、その文に対応付けられた接続コンテキスト・クラスのメソッドと、その接続コンテキスト・クラスに対応付けられたプロファイル・オブジェクトが使用されます。その場合、次のように処理が行われます。

1. エンド・ユーザーが実行しているアプリケーション内に実行に必要な SQL 操作がある
と、SQLJ ランタイムで接続コンテキストの `getConnectedProfile()` メソッドが
コールされます。
2. 接続コンテキストの `getConnectedProfile()` メソッドは、その接続コンテキスト・
クラスに対応付けられているプロファイル・オブジェクトの
`getConnectedProfile()` メソッドをコールし、接続を渡します。(この接続インタ
ンスが、SQL 操作に使用する接続コンテキスト・クラスの基礎となります。)
3. プロファイル・オブジェクトの `getConnectedProfile()` メソッドは、そのプロフ
ァイルに登録されている各 `Customization` オブジェクトの `acceptsConnection()` メ
ソッドをコールします。接続を最初に受け取った `Customization` オブジェクトが接続
プロファイルを生成し、それをランタイムに渡します。
4. 接続プロファイルは、SQL 操作の実行時に JDBC 接続と同じように使用され、実行する
文を生成します。その他に、カスタマイズされた特別な機能を実装します。

カスタマイズ・オプションとカスタマイザの選択

ここでは、プロファイルのカスタマイズ時のオプションについて説明します。これらのオプションは、次の3つのカテゴリに分類できます。

- カスタマイザ・ハーネスに対して指定するオプション。使用するカスタマイザに関係なく適用されます。

このオプションの例としては、汎用オプション、接続オプション、専用カスタマイザを起動するオプションなどがあります。

- カスタマイザ固有のオプション。これらのオプションはカスタマイザ・ハーネスを介して、使用するカスタマイザに対して指定します。
- SQLJ オプション。これらのオプションは、カスタマイズを行うかどうか、どのカスタマイザを使用するかなど、カスタマイズの基本事項を指定します。

どのカテゴリのオプションも、すべて SQLJ コマンドラインまたはプロパティ・ファイルを使用して指定します。

ここでは、次の内容を取り上げます。

- [カスタマイザ・ハーネスのオプションの概要](#)
- [カスタマイザ・ハーネスの汎用オプション](#)
- [カスタマイザ・ハーネスの接続用オプション](#)
- [専用のカスタマイザの起動に使用するカスタマイザ・ハーネスのオプション](#)
- [カスタマイザ固有のオプションの概要](#)
- [Oracle カスタマイザのオプション](#)
- [プロファイルのカスタマイズ用の SQLJ トランスレータのオプション](#)

デフォルトの Oracle カスタマイザ以外のカスタマイザを選択するには、カスタマイザ・ハーネスの `-customizer` オプション (A-11 ページの「[カスタマイザ・ハーネスのオプションの概要](#)」を参照) または SQLJ の `-default-customizer` オプション (A-36 ページの「[プロファイルのカスタマイズ用の SQLJ トランスレータのオプション](#)」を参照) を使用します。

カスタマイザ・ハーネスのオプションの概要

Oracle SQLJ のカスタマイザ・ハーネスには、特定のカスタマイザ以外にも使用できるオプションがいくつか用意されています。このハーネスの各オプションは、カスタマイズ処理をフロントエンドで調整する際に使用されます。

カスタマイザ・ハーネスのオプションの構文

SQLJ コマンドラインでカスタマイザ・ハーネス・オプションを設定する場合は、次の構文を使用します。

```
-P-option=value
```

SQLJ プロパティ・ファイルでは、次の構文を使用します。

```
profile.option=value
```

ブール値のオプション（フラグ）を有効にするには、次のように指定します。

```
-P-option
```

または

```
-P-option=true
```

ブール値のオプションは、デフォルトでは無効になっています。明示的に無効にするには、次のように指定します。

```
-P-option=false
```

このオプションの構文の詳細は、8-60 ページの「[プロファイル・カスタマイザに渡すオプション \(-P\)](#)」および 8-17 ページの「[プロパティ・ファイルの構文](#)」を参照してください。

カスタマイザ・ハーネスでサポートされているオプション

カスタマイザ・ハーネスには、次の汎用オプションを使用できます。

- **backup**: カスタマイズ前にプロファイルのバックアップ・コピーを作成するためのオプション
- **context**: リストされた接続コンテキスト・クラスに対応付けられているプロファイルのみをカスタマイズするためのオプション
- **customizer**: 使用するカスタマイザを指定するためのオプション
- **digests**: .jar マニフェスト・ファイルのダイジェスト値を指定するためのオプション (.jar ファイルのカスタマイズを指定した場合のみ)

- **help:** カスタマイザ・オプションを表示するためのオプション (SQLJ コマンドラインを使用した場合のみ)

- **verbose:** カスタマイズ時にステータス・メッセージを表示するためのオプション

カスタマイザ・ハーネスでは、次のカスタマイザ・データベース接続用オプションを使用できます。現在、Oracle カスタマイザでは、列定義 (パフォーマンスの最適化) 用の **optcols** オプションを有効化した場合にこれらのオプションが使用されます。また、**SQLCheckerCustomizer** と呼ばれる、プロファイルに対してセマンティクス・チェックを実行する特別なカスタマイザを使用した場合にも、これらのオプションが使用されます。

- **user:** このカスタマイズで使用する接続用ユーザー名を指定するためのオプション
- **password:** このカスタマイズで使用する接続用パスワードを指定するためのオプション
- **url:** このカスタマイズで使用する接続用 URL を指定するためのオプション
- **driver:** このカスタマイズで使用する接続用 JDBC ドライバを指定するためのオプション

Oracle カスタマイザ **optcols** フラグの詳細は、A-26 ページの「[Oracle カスタマイザの列定義オプション \(optcols\)](#)」を参照してください。**SQLCheckerCustomizer** の詳細は、A-38 ページの「[プロファイルのセマンティクス・チェック用の SQLCheckerCustomizer](#)」を参照してください。

次のコマンドは、カスタマイザ・ハーネス・オプションとして機能しますが、Oracle SQLJ で提供される特別なカスタマイザを介して実装されます。

- **debug:** デバッグ情報を実行時に出力される特定のプロファイルに挿入するためのオプション。これは Oracle SQLJ の **AuditorInstaller** を起動する簡単な方法です。A-42 ページの「[デバッグ用の AuditorInstaller カスタマイザ](#)」を参照してください。
- **print:** プロファイルの内容をテキスト・フォーマットで出力するためのオプション
- **verify:** SQLJ トランスレータ実行時にすでに生成されたプロファイルに対してセマンティクス・チェック (変換時に行われる、ソース・コードへのセマンティクス・チェックに相当) を行うためのオプション。これは Oracle SQLJ の **SQLCheckerCustomizer** を起動する簡単な方法です。A-38 ページの「[プロファイルのセマンティクス・チェック用の SQLCheckerCustomizer](#)」を参照してください。

カスタマイザ・ハーネスの汎用オプション

この項では、カスタマイザ・ハーネスでサポートされている汎用オプションについて説明します。

プロファイル・バックアップ・オプション (backup)

backup フラグを使用すると、元のファイルに上書きする前に、各 .jar ファイルとスタンダアロンの .ser ファイルのバックアップ・コピーをハーネスで保存できます。(.jar 内の .ser ファイル用のバックアップを別に作成する必要はありません。)

バックアップ・ファイル名には、拡張子として .bakn が付加されます。n は、似た名前のファイルが存在する場合に、必要に応じて使用される数値です。バックアップ・ファイルが作成されるたびに、情報メッセージが出力されます。

スタンダアロンの .ser ファイルのカスタマイズ時にエラーが発生した場合は、元の .ser ファイルは上書きされず、バックアップも作成されません。同様に、.jar ファイル内の .ser ファイルのカスタマイズ時にエラーが発生した場合も、元の .jar ファイルは上書きされず、バックアップも作成されません。

コマンドラインの構文 -P-backup<=true|false>

コマンドラインの例 -P-backup

プロパティ・ファイルの構文 profile.backup<=true|false>

プロパティ・ファイルの例 profile.backup

デフォルト値 false

カスタマイズ接続コンテキスト・オプション (context)

context オプションを使用すると、指定された接続コンテキスト・クラスに対応するプロファイルのみに、カスタマイズの対象を限定できます。クラスの完全修飾名を指定します。複数のクラスを指定する場合は、各クラスをカンマで区切ります。次にその例を示します。

-P-context=sqlj.runtime.ref.DefaultContext,foo.bar.MyCtxtClass

カンマの前後には、スペースは不要です。

このオプションが指定されていない場合は、対応付けられている接続コンテキスト・クラスに関係なく、すべてのプロファイルがカスタマイズされます。

コマンドラインの構文 -P-context=ctx_class1<,ctx_class2,...>

コマンドラインの例 -P-context=foo.bar.MyCtxtClass

プロパティ・ファイルの構文 `profile.context=ctx_class1<,ctx_class2,...>`

プロパティ・ファイルの例 `profile.context=foo.bar.MyCtxtClass`

デフォルト値 なし（すべてのプロファイルがカスタマイズされます）

カスタマイザ・オプション（customizer）

`customizer` オプションを使用すると、使用するカスタマイザを指定できます。次に示すように、クラスの完全修飾名を指定します。

`-P-customizer=oracle.sqlj.runtime.util.OraCustomizer`

このオプションが設定されていない場合は、SQLJ の `-default-customizer` オプションで指定されているカスタマイザが使用されます。このオプションでの設定を行わない場合、デフォルトの設定は次のようになります。

`oracle.sqlj.runtime.util.OraCustomizer`

コマンドラインの構文 `-P-customizer=customizer_class`

コマンドラインの例 `-P-customizer=a.b.c.MyCustomizer`

プロパティ・ファイルの構文 `profile.customizer=customizer_class`

プロパティ・ファイルの例 `profile.customizer=a.b.c.MyCustomizer`

デフォルト値 なし（SQLJ の `-default-customizer` オプションで設定されたデフォルトが使用されます）

カスタマイズ用 JAR ファイルのダイジェスト・オプション（digests）

`.jar` ファイルの生成時に、`jar` ユーティリティを使用すると、エントリごとに 1 つ以上のダイジェストを設定できます。これにより、指定された 1 つ以上のアルゴリズムに基づいて、後で `.jar` ファイルの整合性チェックを実行できます。ダイジェストは、概念的にはチェックサムと似ています。

`.jar` ファイルの更新時に、`.jar` ファイルに含まれているプロファイルをカスタマイズし、`jar` ユーティリティを使用して新しいダイジェストを追加（または、既存のダイジェストを更新）する場合は、`digests` オプションを使用して、1 つ以上のアルゴリズムをカンマで区切って指定します。`jar` ユーティリティは、これらのアルゴリズムを使用して、各エントリのダイジェストを生成します。`jar` ユーティリティは、`jar` マニフェスト・ファイル内の各 `.jar` ファイル・エントリに対して、アルゴリズムごとに 1 つのダイジェストを生成します。次のように、アルゴリズムを指定します。

`-P-digests=SHA,MD5`

カンマの前後には、スペースは不要です。

この例では、.jar マニフェスト・ファイル内の各エントリについて、SHA および MD5 の 2 つのダイジェストが生成されます。

.jar ファイルおよび jar ユーティリティの詳細は、次のいずれかの Web サイトを参照してください。

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

または

<http://java.sun.com/j2se/1.3/docs/guide/jar/index.html>

コマンドラインの構文 `-P-digests=algo1<, algo2, ...>`

コマンドラインの例 `-P-digests=SHA,MD5`

プロパティ・ファイルの構文 `profile.digests=algo1<, algo2, ...>`

プロパティ・ファイルの例 `profile.digests=SHA,MD5`

デフォルト値 `SHA,MD5`

カスタマイズ用ヘルプ・オプション (help)

help オプションを使用すると、カスタマイザ・ハーネスと、デフォルトのカスタマイザまたは指定されているカスタマイザのオプションを表示できます。ハーネスおよび Oracle カスタマイザに関しては、各オプションの簡単な説明と現行の設定も表示されます。

ハーネスとデフォルトのカスタマイザ (Oracle カスタマイザ、または SQLJ の `-default-customizer` オプションで指定されているカスタマイザ) のオプションを表示するには、次のように指定します。

`-P-help`

次のように、help オプションに customizer オプションを指定すると、特定のカスタマイザのオプションを表示できます。

`-P-help -P-customizer=sqlj.runtime.profile.util.AuditorInstaller`

注意：

- `-P-help` オプションは、SQLJ コマンドラインでのみ指定できます。SQLJ プロパティ・ファイルでは使用できません。
 - `-P-help` フラグが有効になっている場合は、カスタマイズするプロファイルをコマンドラインで指定しても、カスタマイズは行われません。
-
-

コマンドラインの構文 `-P-help <-P-customizer=customizer_class>`

コマンドラインの例 `-P-help`

プロパティ・ファイルの構文 該当なし

プロパティ・ファイルの例 該当なし

デフォルト値 なし

カスタマイズ用の詳細オプション (verbose)

`verbose` フラグを使用すると、カスタマイズ時にハーネスからステータス・メッセージを表示できます。これらのメッセージは、SQLJ の他のメッセージと同じように、標準出力に書き込まれます。

コマンドラインの構文 `-P-verbose<=true|false>`

コマンドラインの例 `-P-verbose`

プロパティ・ファイルの構文 `profile.verbose<=true|false>`

プロパティ・ファイルの例 `profile.verbose`

デフォルト値 `false`

カスタマイザ・ハーネスの接続用オプション

ここでは、カスタマイザ・ハーネスでサポートされている接続オプションについて説明します。次のように使用されます。

- Oracle カスタマイザでは、列定義にのみデータベース接続を使用します。Oracle カスタマイザの `optcols` オプションを有効にしない場合は、カスタマイザ・ハーネスの `user`、`password`、`url` および `driver` オプションを設定する必要はありません。
- `SQLCheckerCustomizer` と呼ばれる、プロファイルに対してセマンティクス・チェックを実行する特別なカスタマイザでは、オンライン・チェックの際にカスタマイザ・ハーネスの `user`、`password`、`url` および `driver` 設定が使用されます。

SQLJ コマンドラインで `-P-verify` を使用して、カスタマイザを起動します。

注意： カスタマイザ・ハーネスの `user`、`password`、`url` および `driver` オプションと同じ名前のオプションがトランスレータにもありますが、混同しないようにしてください。トランスレータのオプションは、変換処理でのセマンティクス・チェックに使用されるオプションです。ただし、変換と同じ接続を使用するようにカスタマイズした場合は、トランスレータの設定がカスタマイザに渡され、利用されます。必要に応じて、カスタマイザ・ハーネス・オプションを使用してこれらの初期設定を上書きしてください。

カスタマイズ用ユーザー・オプション (user)

`user` オプションを設定すると、カスタマイザでデータベース接続を使用する場合のデータベース・スキーマを指定できます。

`user` オプション設定には、ユーザー・スキーマの他にも、必要に応じてパスワードまたは URL、あるいはその両方を指定できます。次のように、パスワードの先頭にはスラッシュ (/) を付け、URL の先頭にはアットマーク (@) を付けます。

```
-P-user=scott/tiger
-P-user=scott@jdbc:oracle:oci:@
-P-user=scott/tiger@jdbc:oracle:oci:@
```

注意： 列定義 (`optcols` オプション) を使用するとき、SQLJ トランスレータのユーザー設定もプロファイル・カスタマイザに転送されます。カスタマイザの `user` オプションを使用すると、トランスレータの設定を上書きできます。

コマンドラインの構文 `-P-user=username</password><@url>`

コマンドラインの例

```
-P-user=scott
-P-user=scott/tiger
-P-user=scott/tiger@jdbc:oracle:oci:@
```

プロパティ・ファイルの構文 `profile.user=username</password><@url>`

プロパティ・ファイルの例

```
profile.user=scott
profile.user=scott/tiger
profile.user=scott/tiger@jdbc:oracle:oci:@
```

デフォルト値 `null`

カスタマイズ用パスワード・オプション (password)

password オプションは、カスタマイザでデータベース接続を使用する場合に使用します。

パスワードは、user オプションでも設定できます。A-17 ページの「[カスタマイズ用ユーザー・オプション \(user\)](#)」を参照してください。

注意： 列定義 (optcols オプション) を使用するとき、SQLJ トランスレータのパスワード設定もプロファイル・カスタマイザに転送されます。カスタマイザの password オプションを使用すると、トランスレータの設定を上書きできます。

コマンドラインの構文 `-P-password=password`

コマンドラインの例 `-P-password=tiger`

プロパティ・ファイルの構文 `profile.password=password`

プロパティ・ファイルの例 `profile.password=tiger`

デフォルト値 `null`

カスタマイズ用 URL オプション (url)

url オプションは、カスタマイザでデータベース接続を使用する場合に使用します。

URL は、user オプションでも設定できます。A-17 ページの「[カスタマイズ用ユーザー・オプション \(user\)](#)」を参照してください。

注意： 列定義 (optcols オプション) を使用するとき、SQLJ トランスレータの URL 設定もプロファイル・カスタマイザに転送されます。カスタマイザの url オプションを使用すると、トランスレータの設定を上書きできます。

コマンドラインの構文 -P-url=url

コマンドラインの例 -P-url=jdbc:oracle:oci:@

プロパティ・ファイルの構文 profile.url=url

プロパティ・ファイルの例 profile.url=jdbc:oracle:oci:@

デフォルト値 jdbc:oracle:oci:@

カスタマイズ用 JDBC ドライバ・オプション (driver)

カスタマイザでデータベース接続を使用する場合は、driver オプションを使用し、複数の JDBC ドライバ・クラスをカンマで区切って登録します。次にその例を示します。

-P-driver=sun.jdbc.odbc.JdbcOdbcDriver,oracle.jdbc.OracleDriver

カンマの前後には、スペースは不要です。

コマンドラインの構文 -P-driver=dvr_class1<,dvr_class2,...>

コマンドラインの例 -P-driver=sun.jdbc.odbc.JdbcOdbcDriver

プロパティ・ファイルの構文 profile.driver=dvr_class1<,dvr_class2,...>

プロパティ・ファイルの例 profile.driver=sun.jdbc.odbc.JdbcOdbcDriver

デフォルト値 oracle.jdbc.OracleDriver

専用のカスタマイザの起動に使用するカスタマイザ・ハーネスのオプション

特別なカスタマイザの起動には、次に示したカスタマイザ・ハーネスのオプションを使用できます。

- **debug:** デバッグに使用する AuditorInstaller カスタマイザを起動するためのオプション
- **print:** プロファイルをテキスト形式で出力するカスタマイザを起動するためのオプション
- **verify:** プロファイルに対してセマンティクス・チェックを実行する SQLCheckerCustomizer カスタマイザを起動するためのオプション

重要： これらの各オプションを使用すると特定のカスタマイザを起動できますが、SQLJ の 1 回の実行で実行できるカスタマイザは 1 つのみに限られています。そのため、このオプションを使用した場合は、他のカスタマイズを実行できなくなります。

print、debug および verify のうち、同時に使用できるのは 1 つのみです。

専用カスタマイザ：プロファイル・デバッグ・オプション (debug)

debug オプションを使用すると、AuditorInstaller と呼ばれる、デバッグ文をプロファイルに挿入する専用のカスタマイザを起動できます。このオプションに、SQLJ コマンドラインでファイルを指定すると、指定されたプロファイルにデバッグ文が出力されます。これらのプロファイルは、あらかじめ SQLJ を実行してカスタマイズしておく必要があります。

このカスタマイザでサポートされている他のオプションなどの詳細は、A-42 ページの「[デバッグ用の AuditorInstaller カスタマイザ](#)」を参照してください。

アプリケーションを実行すると、SQLJ ランタイムでデバッグ文が実行され、メソッド・コールと戻り値のトレース情報が表示されます。

たとえば、次のように debug オプションを指定します。

```
sqlj -P-debug Foo_SJProfile0.ser Bar_SJProfile0.ser
```

```
sqlj -P-debug *.ser
```

コマンドラインの構文 `sqlj -P-debug profile_list`

コマンドラインの例 `sqlj -P-debug Foo_SJProfile*.ser`

プロパティ・ファイルの構文 `profile.debug`

(SQLJ ファイル・リストのプロファイルにも指定します。)

プロパティ・ファイルの例 `profile.debug`

デフォルト値 該当なし

専用カスタマイザ: プロファイル出力オプション (print)

print オプションを使用すると、プロファイルをテキスト・フォーマットで表示するためのカスタマイザを実行できます。このオプションに、SQLJ コマンドラインでファイルを指定すると、指定された 1 つ以上のプロファイルの内容が出力されます。これは、SQLJ の標準出力（通常はディスプレイ）に出力されます。

たとえば、次のように print オプションを指定します。

```
sqlj -P-print Foo_SJProfile0.ser Bar_SJProfile0.ser
```

```
sqlj -P-print *.ser
```

サンプル出力については、A-3 ページの「[プロファイル・エントリの例](#)」を参照してください。

コマンドラインの構文 `sqlj -P-print profile_list`

コマンドラインの例 `sqlj -P-print Foo_SJProfile*.ser`

プロパティ・ファイルの構文 `profile.print`

(SQLJ ファイル・リストのプロファイルにも指定します。)

プロパティ・ファイルの例 `profile.print`

デフォルト値 該当なし

専用カスタマイザ：プロファイルのセマンティクス・チェック用オプション (verify)

verify オプションを指定すると、プロファイルに対してセマンティクス・チェックを実行する SQLCheckerCustomizer カスタマイザが起動されます。これと同様のセマンティクス・チェックは、変換時にソース・コードに対しても実行されます。このプロファイルは、SQLJ トランスレータ実行時にすでに作成されているためです。

このオプションは、配布後やソース・コードが使用されなくなった後の、ランタイム・データベースに対するセマンティクス・チェックに役立ちます。

このカスタマイザでサポートされている他のオプションなどの詳細は、A-38 ページの「[プロファイルのセマンティクス・チェック用の SQLCheckerCustomizer](#)」を参照してください。

注意： プロファイルのオンライン・セマンティクス・チェックの際にも、カスタマイザ・ハーネスの user、password および url オプションを使用する必要があります。

たとえば、次のように verify オプションを指定します。次の 2 つの例では、どちらも SQLCheckerCustomizer のデフォルトのセマンティクス・チェッカを使用し、特定のデータベース接続によるオンライン・チェックを行います。（最初の例ではコマンドラインが折り返されて表示されていますが、全体が 1 行で入力されています。）

```
sqlj -P-verify -P-user=scott -P-password=tiger -P-url=jdbc:oracle:oci:@
Foo_SJProfile0.ser Bar_SJProfile0.ser
```

```
sqlj -P-verify -P-user=scott -P-password=tiger -P-url=jdbc:oracle:oci:@ *.ser
```

コマンドラインの構文 `sqlj -P-verify <conn params> profile_list`

コマンドラインの例 `sqlj -P-verify <conn params> Foo_SJProfile*.ser`

プロパティ・ファイルの構文 `profile.verify`

(SQLJ コマンドラインにはプロファイルを必ず指定してください。通常はカスタマイザ・ハーネスの接続オプションを指定します。)

プロパティ・ファイルの例 `profile.verify`

デフォルト値 該当なし

カスタマイザ固有のオプションの概要

カスタマイザ固有のオプション（Oracle カスタマイザ用のオプションなど）は、SQLJ コマンドラインまたは SQLJ プロパティ・ファイルに設定できます。この構文は、カスタマイザ・ハーネスのオプション設定に使用する構文とほとんど同じです。

SQLJ コマンドラインでカスタマイザ・オプションを設定する場合は、先頭に次の文字列を付けます。

```
-P-C
```

SQLJ プロパティ・ファイルで設定するときは、先頭に次の文字列を付けます。

```
profile.C
```

このオプションの構文の詳細は、8-60 ページの「[プロファイル・カスタマイザに渡すオプション \(-P\)](#)」および 8-17 ページの「[プロパティ・ファイルの構文](#)」を参照してください。

ここでは、Oracle カスタマイザについて説明します。Oracle カスタマイザでは、いくつかのオプションがサポートされています。これらのオプションの大半はブール値であり、有効にするには次のようにします。

```
-P-Option
```

または

```
-P-Option=true
```

ブール値のオプションは、デフォルトでは無効になっています。明示的に無効にするには、次のように指定します。

```
-P-Option=false
```

数値オプションや文字列オプションの設定は、ほとんど同じです。

```
-P-Option=value
```

Oracle カスタマイザのオプション

ここでは、Oracle カスタマイザ固有のオプションについて説明します。まず、サポートされているオプションの概要を示します。

Oracle カスタマイザでサポートされているオプション

Oracle カスタマイザでは、次のオプションを使用できます。

- **compat:** バージョンの互換性に関する情報を表示するためのオプション
- **force:** すでに有効なカスタマイズが行われている場合でも、カスタマイズを実行するためのオプション

- **optcols:** イテレータ列型およびサイズを、パフォーマンスが最適化されるように定義するためのオプション。
- **optparams:** パラメータ・サイズを、JDBC リソース割当て量が最適化されるように定義するためのオプション (**optparamdefaults** と併用します)。
- **optparamdefaults:** 特定のデータ型に対してパラメータ・サイズのデフォルトを設定するためのオプション (**optparams** と併用します)。
- **fixedchar:** WHERE 句に関して、空白埋めを考慮した CHAR 比較を有効にするためのオプション。
- **showSQL:** SQL 文の変換情報を表示するためのオプション。
- **stmtcache:** 文のキャッシュ・サイズ (実行時に各接続でキャッシングできる文の数) を設定するためのオプション。パフォーマンスの最適化に必要なサイズを設定する場合と、文のキャッシングを無効化するために 0 (ゼロ) が設定される場合があります。
- **summary:** アプリケーションで使用されている Oracle の機能の概要を表示するためのオプション。

これらのオプションによって出力された内容は、SQLJ の他のメッセージと同じように、標準出力に書き込まれます。

Oracle カスタマイザのバージョン互換性オプション (compat)

compat フラグを有効にすると、アプリケーションと、Oracle データベースおよび Oracle JDBC ドライバの各バージョンとの互換性に関する情報が、Oracle カスタマイザによって表示されます。これは、SQLJ 変換を全体的に実行した場合でも、以前に生成されたプロファイルに対して実行した場合でも、実行できます。

たとえば、次のように指定すると、MyApp というアプリケーションを変換してカスタマイズする際に、互換性の情報が出力されます。

```
sqlj <...SQLJ options...> -P-Ccompat MyApp.sqlj
```

この例では、MyApp のプロファイルの生成、カスタマイズおよび互換性チェックが、SQLJ の 1 回の実行で行われます。

次のように指定すると、以前に生成された MyApp のプロファイルに関する互換性情報が出力されます。

```
sqlj <...SQLJ options...> -P-Ccompat MyApp_SJProfile*.ser
```

この例のように指定して SQLJ を実行すると、以前に SQLJ を実行したときに生成された (場合によってはカスタマイズもされた) MyApp のプロファイルに対して、カスタマイズ (必要な場合) と互換性チェックが行われます。

次の 2 つの例は、デフォルトの Oracle カスタマイザを使用している場合に、**-P-Ccompat** オプションを指定して SQLJ を実行したときに出力される内容です。最初に、アプリケーションが、すべてのバージョンの Oracle JDBC ドライバで使用できる例を示します。

```
MyApp_SJProfile0.ser: Info: compatible with all Oracle JDBC drivers
```

2 番目の例では、アプリケーションが、Oracle JDBC の 8.1 以上のリリースでのみ使用できる場合を示します。

```
MyApp_SJProfile0.ser: Info: compatible with Oracle 8.1 or later JDBC driver
```

注意： 以前の有効なカスタマイズが検出されてカスタマイズが実行されなかった場合でも、`compat` オプションが指定されていると、互換性の情報が出力されます。

コマンドラインの構文 `-P-Ccompat<=true|false>`

コマンドラインの例 `-P-Ccompat`

プロパティ・ファイルの構文 `profile.Ccompat<=true|false>`

プロパティ・ファイルの例 `profile.Ccompat`

デフォルト値 `false`

Oracle カスタマイザの強制オプション (force)

`force` フラグを有効にすると、Oracle カスタマイザによって特定の（コマンドラインで指定された）プロファイルのカスタマイズが強制的に実行されます。そのプロファイルに、すでに有効なカスタマイズが認識される場合でも、実行されます。次にその例を示します。

```
sqlj -P-Cforce MyApp_SJProfile*.ser
```

これを実行すると、以前にカスタマイズされているかどうかにかかわらず、MyApp のプロファイルがすべてカスタマイズされます。デフォルトでは、以前のカスタマイズが認識された場合は、Oracle カスタマイザはそれを上書きしません。ただし、以前のカスタマイズで使われたカスタマイザのバージョンが、現行のバージョンよりも古い場合は除きます。

コマンドラインの構文 `-P-Cforce<=true|false>`

コマンドラインの例 `-P-Cforce`

プロパティ・ファイルの構文 `profile.Cforce<=true|false>`

プロパティ・ファイルの例 `profile.Cforce`

デフォルト値 `false`

Oracle カスタマイザの列定義オプション (optcols)

optcols フラグを使用すると、Oracle カスタマイザでイテレータや結果セットの列型およびサイズを判断し、この情報をプロファイルに追加できます。このようにすると、アプリケーション実行時に、SQLJ ランタイムではこの列を Oracle JDBC ドライバに自動的に登録できます。その結果、ドライバの実装にもよりますが、Oracle へのラウンドトリップが減ります。この方法は、Thin ドライバと位置指定イテレータには特に効果的です。

列定義の概要は、10-22 ページの「[列の定義](#)」を参照してください。

データベース接続用のユーザー名、パスワードおよび URL を指定せずに、Oracle カスタマイザの -optcols オプションを有効にすると、エラーが発生します。この指定を行うには、ISO 標準のコード生成時にカスタマイザに転送される、トランスレータのオプション -user、-password および -url を使用するか、またはカスタマイザのオプション user、password および url を直接使用します。

実行時にはデータベースの接続先と同じスキーマまたはデータベースに、カスタマイザから接続する必要はありません。ただし、実行時のエラーを回避するために、関連のある列は、同じ順序で、型およびサイズが同じであることが必要です。

カスタマイザ・ハーネスの接続オプションについては、A-11 ページの「[カスタマイザ・ハーネスのオプションの概要](#)」の user、password、url および driver の項を参照してください。

注意： このオプションのかわりに、SQLJ トランスレータの -optcols オプション（カスタマイザのオプションを自動的に設定するオプション）を使用できます。（また、プロファイルを使用しない Oracle 固有コード生成の場合は、トランスレータのオプションを使用する必要があります。）
8-51 ページの「[列の定義 \(-optcols\)](#)」を参照してください。

参照先の項では、さらに概念についても説明されています。

カスタマイザの optcols フラグの有効化 / 無効化は、SQLJ のコマンドラインまたはプロパティ・ファイルで設定します。

このフラグをコマンドラインで有効にするには、次のようにします。

-P-Coptcols

または

-P-Coptcols=true

このフラグは、デフォルトでは無効になっていますが、明示的に無効化することも可能です。このフラグをコマンドラインで無効化するには、次のようにします。

-P-Coptcols=false

列を定義するには、カスタマイザでデータベース接続を行い、問合せの対象とする表の列を調べる必要があります。そのために、カスタマイザ・ハーネスの `user`、`password` および `url` オプションを（デフォルトの `OracleDriver` クラスを使用しない場合は、カスタマイザ・ハーネスの `driver` オプションも）適切な設定にしてください。次にその例を示します。

```
sqlj <...SQLJ options...> -P-user=scott/tiger@jdbc:oracle:oci:@ -P-Coptcols MyApp.sqlj
```

SQLJ トランスレータの場合と同様に、必要に応じて、パスワードと URL を `password` オプションと `url` オプションでそれぞれ設定するかわりに、`user` オプションで設定できます。

または、次のようにして、列定義を既存のプロファイルに挿入することも可能です。この場合は、Oracle カスタマイザの `force` オプションを使用して、カスタマイズを再実行する必要があります。

```
sqlj -P-user=scott/tiger@jdbc:oracle:oci:@ -P-Cforce -P-Coptcols MyApp_SJProfile*.ser
```

また、次のようにして、列定義を `.jar` ファイル中の既存のプロファイルに挿入することも可能です。

```
sqlj -P-user=scott/tiger@jdbc:oracle:oci:@ -P-Cforce -P-Coptcols MyAppProfiles.jar
```

Oracle カスタマイザ実行時（新規のプロファイルの変換時や生成時、または既存プロファイルのカスタマイズ時）に `optcols` フラグを有効化すると、カスタマイザ・ハーネスの `verbose` フラグも有効化できます。このフラグを有効化すると、処理対象のイテレータおよび結果セット、列型およびサイズ定義の情報が Oracle カスタマイザに表示されます。次にその例を示します。

```
sqlj -P-user=scott/tiger@jdbc:oracle:oci:@ -P-verbose -P-Cforce -P-Coptcols MyApp_SJProfile*.ser
```

`verbose` のフラグの概要は、A-11 ページの「[カスタマイザ・ハーネスのオプションの概要](#)」の該当の項を参照してください。

Oracle カスタマイザの `summary` フラグを有効にして、既存のプロファイルの処理を実行すると、列定義がそのプロファイルに追加されたかどうかを確認できます。次のようにします。

```
sqlj -P-Csummary MyApp_SJProfile*.ser
```

`summary` フラグの概要は、A-23 ページの「[カスタマイザ固有のオプションの概要](#)」を参照してください。

コマンドラインの構文 `-P-Coptcols<=true|false>`

コマンドラインの例 `-P-Coptcols`

プロパティ・ファイルの構文 `profile.Coptcols<=true|false>`

プロパティ・ファイルの例 `profile.Coptcols`

デフォルト値 `false`

Oracle カスタマイザのパラメータ定義オプション (optparams)

パラメータ・サイズの定義を有効化するには、`optparams` フラグを使用します。SQLJ ではこのフラグを有効にすると、入出力パラメータ（ホスト変数）が登録され、JDBC リソース割当て量が指定したサイズに基づいて最適化されます。

パラメータ・サイズ定義の概要とソース・コード・ヒントの詳細は、10-23 ページの「[パラメータ・サイズの定義](#)」を参照してください。

注意： このオプションのかわりに、SQLJ トランスレータの `-optparams` オプション（カスタマイザのオプションを自動的に設定するオプション）を使用できます。（また、プロファイルを使用しない Oracle 固有コード生成の場合は、トランスレータのオプションを使用する必要があります。）8-52 ページの「[パラメータの定義 \(-optparams\)](#)」を参照してください。

参照先の項では、さらに概念についても説明されています。

`optparams` フラグの有効化 / 無効化は、コマンドラインまたは SQLJ プロパティ・ファイルで設定します。

このフラグをコマンドラインで有効にするには、次のようにします。

`-P-Coptparams`

または

`-P-Coptparams=true`

このフラグは、デフォルトでは無効になっていますが、明示的に無効化することも可能です。このフラグをコマンドラインで無効化するには、次のようにします。

`-P-Coptparams=false`

注意： `optcols` オプションとは異なり、`optparams` オプションではサイズを自分で指定できるので、カスタマイザでのデータベース接続は不要です。

コマンドライン（`optparamdefaults` オプションの設定はここでは省略し、次の項で説明します）の例を次に示します。

```
sqlj <...SQLJ options...> -P-Coptparams -P-Coptparamdefaults=defaults_string MyApp.sqlj
```

次の例のようにすると、既存プロファイルのパラメータ・サイズ定義が有効になります。

```
sqlj -P-Coptparams -P-Coptparamdefaults=defaults_string MyApp_SJProfile*.ser
```

.jar ファイル中の既存プロパティに対しては、次のようにしてパラメータ・サイズ定義を有効化します。

```
sqlj -P-Coptparams -P-Coptparamdefaults=defaults_string MyAppProfiles.jar
```

コマンドラインの構文 -P-Coptparams<=true|false>

コマンドラインの例 -P-Coptparams

プロパティ・ファイルの構文 profile.Coptparams<=true|false>

プロパティ・ファイルの例 profile.Coptparams

デフォルト値 false

Oracle カスタマイザ用パラメータのデフォルト・サイズ・オプション (optparamdefaults)

optparams オプションを有効にしてパラメータ・サイズを設定した場合は、必要に応じて optparamdefaults オプションを使用してデータ型のデフォルト・サイズを設定します。optparams を有効にしないと、optparamdefaults が設定されていても無視されるので注意してください。

パラメータ・サイズ定義の概要とソース・コード・ヒントの詳細は、10-23 ページの「[パラメータ・サイズの定義](#)」を参照してください。

注意： このオプションのかわりに、SQLJ トランスレータの -optparamdefaults オプション（カスタマイザのオプションを自動的に設定するオプション）を使用できます。（また、プロファイルを使用しない Oracle 固有コード生成の場合は、トランスレータのオプションを使用する必要があります。）8-54 ページの「[パラメータのデフォルト・サイズ \(-optparamdefaults\)](#)」を参照してください。

参照先の項では、さらに概念と構文についても説明されています。2つのオプションの機能は同じです。

optparamdefaults フラグは、コマンドラインでも SQLJ プロパティ・ファイルでも設定できます。

このフラグをコマンドラインで設定するには、次のようにします。

```
-P-Coptparamdefaults=datatype1(size1),datatype2(size2),...
```

次にコマンドラインの例を示します。この例では、optparams も設定します。

```
sqlj <...SQLJ options...> -P-Coptparams -P-Coptparamdefaults=CHAR_TYPE(50),RAW_TYPE(500),CHAR(10) MyApp.sqlj
```

構文は、8-54 ページの「[パラメータのデフォルト・サイズ \(-optparamdefaults\)](#)」を参照してください。

既存のプロファイルに対してパラメータ・サイズのデフォルトを指定することも可能です。ただし、この場合は Oracle カスタマイザの force オプションを使用して、カスタマイズを再実行する必要があります。

```
sqlj -P-Cforce -P-Coptparams -P-Coptparamdefaults=CHAR_TYPE(50),RAW_TYPE(500),CHAR(10) MyApp_SJProfile*.ser
```

また、次のようにして、.jar ファイル中の既存のプロファイルに対してパラメータ・サイズのデフォルトを指定することも可能です。

```
sqlj -P-Cforce -P-Coptparams -P-Coptparamdefaults=CHAR_TYPE(50),RAW_TYPE(500),CHAR(10) MyAppProfiles.jar
```

注意： 実行時に、実際のサイズが登録されているパラメータのサイズを超えた場合は、実行時エラーが発生します。

コマンドラインの構文 -P-Coptparamdefaults=defaults_string

コマンドラインの例 -P-Coptparamdefaults=VAR%(50),LONG%(500),RAW_TYPE()

プロパティ・ファイルの構文 profile.Coptparamdefaults=defaults_string

プロパティ・ファイルの例 profile.Coptparamdefaults=VAR%(50),LONG%(500),RAW_TYPE()

デフォルト値 null

Oracle カスタマイザによる空白埋めを考慮した CHAR 比較 (fixedchar)

このフラグを true に設定すると、WHERE 句比較に関する文字列のバインド時に、CHAR データベース列の空白埋めが考慮されます。この比較方法では、たとえば、"mystring" と "mystring " は同等であるとみなされます。

次に、Oracle カスタマイザの fixedchar の使用例を示します。

```
% sqlj -P-Cfixedchar MyProgram.sqlj AnotherProg.java ...
```

注意：

- このオプションのかわりに、SQLJ トランスレータの `-fixedchar` オプション（カスタマイザのオプションを自動的に設定するオプション）を使用できます。（また、プロファイルを使用しない Oracle 固有コード生成の場合は、トランスレータのオプションを使用する必要があります。）8-56 ページの「空白埋めを考慮した CHAR 比較 (`-fixedchar`)」を参照してください。参照先の項では、さらに概念についても説明されています。
- Oracle カスタマイザの `summary` フラグも有効にすると、(`fixedchar` 機能の背後で使用される) Oracle の `setFixedCHAR()` API の使用数が表示されます。例については、A-34 ページの「Oracle カスタマイザのサマリー・オプション (`summary`)」を参照してください。

コマンドラインの構文 `-P-Cfixedchar<=true|false>`

コマンドラインの例 `-P-Cfixedchar`

プロパティ・ファイルの構文 `profile.Cfixedchar<=true|false>`

プロパティ・ファイルの例 `profile.Cfixedchar`

デフォルト値 `false`

Oracle カスタマイザの SQL 表示オプション（`showSQL`）

`showSQL` フラグを有効にすると、Oracle カスタマイザによって SQL 文に実行されたすべての変換が表示されます。このような変換は、Oracle9i がサポートしていない構文を SQLJ がサポートしている場合に必要です。たとえば、次のように指定すると、MyApp というアプリケーションを変換してカスタマイズする際に、SQL 変換情報が表示されます。

```
sqlj <...SQLJ options...> -P-CshowSQL MyApp.sqlj
```

この例では、MyApp のプロファイルの生成、カスタマイズおよび SQL 変換情報の表示が、SQLJ の 1 回の実行で行われます。

次のように指定すると、以前に生成された MyApp のプロファイルのカスタマイズ時に、SQL 変換情報が表示されます。

```
sqlj <...SQLJ options...> -P-CshowSQL MyApp_SJProfile*.ser
```

この例のように指定して SQLJ を実行すると、以前に SQLJ を実行したときに生成された（場合によってはカスタマイズもされた）MyApp のプロファイルに対して、カスタマイズ（必要な場合）が行われ、SQL 変換情報が表示されます。

次の例は、showSQL によって出力される内容です。

```
MyApp.sqlj:14: Info: <<<NEW SQL>>> #sql {BEGIN ? := VALUES(tkjsSET_f1); END};
```

```
in file MyApp, line 14, we had:
```

```
#sql {set :v1= VALUES(tkjsSET_f1) };
```

SQLJ では SET 文がサポートされていますが、Oracle9i ではサポートされていません。SET 文は、カスタマイズ時に Oracle カスタマイザによって、それに相当する PL/SQL ブロックに置き換えられます。

注意： 以前の有効なカスタマイズが検出されてカスタマイズが実行されなかった場合でも、showSQL オプションが指定されていると SQL 変換情報が出力されます。

コマンドラインの構文 -P-CshowSQL<=true|false>

コマンドラインの例 -P-CshowSQL

プロパティ・ファイルの構文 profile.CshowSQL<=true|false>

プロパティ・ファイルの例 profile.CshowSQL

デフォルト値 false

Oracle カスタマイザ用の文のキャッシュ・サイズ・オプション (stmtcache)

Oracle カスタマイザの stmtcache オプションを使用すると、文のキャッシュ・サイズ（アプリケーション実行時にデータベース接続ごとにキャッシングできる文の数）を設定したり、文のキャッシングを無効化することが可能です。

文のキャッシュ・サイズのデフォルトは 5 です。文のキャッシングの概要は、10-4 ページの「[文のキャッシング](#)」を参照してください。

重要： デフォルトの Oracle 固有コード生成 (-codegen=oracle) を使用する場合、SQLJ ではプロファイルが生成されず、カスタマイズが実行されません。この場合、接続コンテキスト・メソッドを使用して SQLJ 文キャッシングを制御します。10-5 ページの「[文キャッシングのための接続コンテキスト・メソッド \(Oracle 固有コードの場合\)](#)」を参照してください。

文のキャッシュ・サイズは、コマンドラインでもプロパティ・ファイルでも設定できます。

コマンドラインを使用して、アプリケーション MyApp での文のキャッシュ・サイズを 15 に設定するには、次のようにします。

```
sqlj <...SQLJ options...> -P-Cstmtcache=15 MyApp.sqlj
```

文のキャッシングを無効化するには、キャッシュ・サイズを 0 (ゼロ) に設定します。

```
sqlj <...SQLJ options...> -P-Cstmtcache=0 MyApp.sqlj
```

アプリケーションを再変換せずに、既存プロファイル中の文のキャッシュ・サイズを変更することも可能です。ただし、この場合は、Oracle カスタマイザの force オプションを使用して、カスタマイズを再実行する必要があります。

```
sqlj -P-Cforce -P-Cstmtcache=15 MyApp_SJProfile0.ser
```

プロファイルが複数ある場合、各プロファイルに対して文のキャッシュ・サイズを個別に設定できます。この個別設定を行うには、アプリケーションの変換が完了した後で、各プロファイルに対して SQLJ を個別に実行します。

```
sqlj -P-Cforce -P-Cstmtcache=10 MyApp_SJProfile0.ser
sqlj -P-Cforce -P-Cstmtcache=15 MyApp_SJProfile1.ser
sqlj -P-Cforce -P-Cstmtcache=0 MyApp_SJProfile2.ser
```

各接続コンテキスト・クラスに対応するプロファイルを決めることは、必須です。このプロファイルは、次のようにして決めます。プロファイル 0 (ゼロ) は、アプリケーションにおける最初の実行文で使用する接続コンテキスト・クラスに対応し、プロファイル 1 は、最初の実行文 (ただし、最初の接続コンテキスト・クラスを使用しない) で使用する接続コンテキスト・クラスに対応します。プロファイル 2 以降についても同様の対応になります。各プロファイルを調べ、相関関係を検証するには、カスタマイザ・ハーネスの print オプションを使用します。

コマンドラインの構文 -P-Cstmtcache=value

コマンドラインの例 -P-Cstmtcache=10

プロパティ・ファイルの構文 profile.Cstmtcache=value

プロパティ・ファイルの例 `profile.Cstmtcache=10`

デフォルト値 5

Oracle カスタマイザのサマリー・オプション (summary)

`summary` フラグを有効にすると、Oracle カスタマイザによって変換するアプリケーション、または指定されたプロファイル・ファイルで使用されている Oracle の機能のサマリーが表示されます。このオプションは、別のプラットフォームへの移植を妨げる機能を確認する場合に使用します。これは、SQLJ 変換を全体的に実行した場合でも、以前に生成されたプロファイルに対して実行した場合でも、実行できます。

次のように指定すると、MyApp というアプリケーションを変換してカスタマイズする際に、サマリーが表示されます。

```
sqlj <...SQLJ options...> -P-Csummary MyApp.sqlj
```

この例では、MyApp のプロファイルの生成、カスタマイズおよびサマリーの表示が、SQLJ の 1 回の実行で行われます。

次のように指定すると、以前に生成された MyApp のプロファイルに関するサマリーが表示されます。

```
sqlj <...SQLJ options...> -P-Csummary MyApp_SJProfile*.ser
```

この例のように指定して SQLJ を実行すると、以前に SQLJ を実行したときに生成された (場合によってはカスタマイズもされた) MyApp のプロファイルに対して、カスタマイズ (必要な場合) が行われ、サマリーが表示されます。

次の 2 つの例は、デフォルトの Oracle カスタマイザを使用している場合に、`-P-Csummary` オプションを指定して SQLJ を実行したときに出力される内容です。最初に、Oracle の機能を使用しない例を示します。

```
MyApp_SJProfile0.ser: Info: Oracle features used:  
MyApp_SJProfile0.ser: Info: * none
```

2 番目の例は、Oracle の機能が複数使用されていることを示しています。具体的には、`oracle.sql` パッケージの Oracle 拡張データ型がいくつか使用されており、それらの一覧が出力されます。

```
MyApp_SJProfile0.ser: Info: Oracle features used:  
MyApp_SJProfile0.ser: Info: * oracle.sql.NUMBER: 2  
MyApp_SJProfile0.ser: Info: * oracle.sql.DATE: 2  
MyApp_SJProfile0.ser: Info: * oracle.sql.CHAR: 2  
MyApp_SJProfile0.ser: Info: * oracle.sql.RAW: 2
```


次の例では、Oracle の `setFixedCHAR()` API の使用数が出力されます（この API は、Oracle カスタマイザの `fixedchar` オプションによって有効化され、CHAR データと比較するために、WHERE 句に文字列をバインドするときに空白埋めが考慮されます）。

```
% sqlj -P-Cfixedchar -P-Csummary -P-Cforce *.ser
FC_SJProfile0.ser: Info: re-installing Oracle customization
FC_SJProfile0.ser: Info: Oracle features used:
FC_SJProfile0.ser: Info: * setFixedCHAR(): 4
```

注意： 以前の有効なカスタマイズが検出されてカスタマイズが実行されなかった場合でも、`summary` オプションが指定されていると、サマリーが出力されます。

コマンドラインの構文 `-P-Csummary<=true|false>`

コマンドラインの例 `-P-Csummary`

プロパティ・ファイルの構文 `profile.Csummary<=true|false>`

プロパティ・ファイルの例 `profile.Csummary`

デフォルト値 `false`

他のカスタマイザ用のオプション

Oracle SQLJ では、この章で後述する特別なカスタマイザも提供されています。これらのカスタマイザにも、コマンドラインのオプションがあります。

- `SQLCheckerCustomizer`（プロファイルのセマンティクス・チェック用）－概要は、A-38 ページの「[プロファイルのセマンティクス・チェック用の SQLCheckerCustomizer](#)」を参照してください。また、オプションについては、A-40 ページの「[SQLCheckerCustomizer オプション](#)」を参照してください。
- `AuditorInstaller`（デバッグ用）－概要は、A-42 ページの「[デバッグ用の AuditorInstaller カスタマイザ](#)」を参照してください。また、オプションについては、A-45 ページの「[AuditorInstaller オプション](#)」を参照してください。

プロファイルのカスタマイズ用の SQLJ トランスレータのオプション

次のオプションは、プロファイルのカスタマイズに関連する SQLJ トランスレータのオプションです。これらのオプションの詳細は、このマニュアルの各項を参照してください。

- `-default-customizer` —使用するデフォルトのカスタマイザを指定します。このオプションは、カスタマイザ・ハーネスの `-customizer` に、プロファイル・カスタマイザが指定されていない場合に使用します。
8-84 ページの「[デフォルト・プロファイル・カスタマイザ \(-default-customizer\)](#)」を参照してください。
- `-profile` —今回の SQLJ の実行でカスタマイズを行うかどうかを指定します。
8-63 ページの「[プロファイル・カスタマイズ・フラグ \(-profile\)](#)」を参照してください。

プロファイルの JAR ファイル

前述のように、SQLJ コマンドラインで `.jar` ファイルを指定して、その `.jar` ファイルに含まれるすべてのプロファイルをカスタマイズします。

注意：

- SQLJ コマンドラインでは、`.sqlj` ファイルまたは `.java` ファイル、あるいはその両方を指定して通常の SQLJ 処理を実行するか、`.ser` ファイルまたは `.jar` ファイル、あるいはその両方を指定してカスタマイズのみを実行できます。ただし、その両方の処理を同時には実行できません。
 - `.jar` ファイルには、プロファイル以外のファイルが含まれている場合があります。マニフェスト・エントリでそのファイルがプロファイルではないことが示されている場合、そのファイルはカスタマイズ時に無視されます。
 - `.jar` ファイルは、それに含まれる各プロファイルに対して、クラスをロードするコンテキストとして使用されます。プロファイルで `.jar` ファイル内のクラスが参照される場合、そのクラスは `.jar` ファイルからロードされます。プロファイルで参照されるクラスが `.jar` ファイル内にない場合は、通常の処理と同じように、システム・クラス・ローダーによって、`CLASSPATH` の指定に従ってクラスが検出されてロードされます。
-

JAR ファイルの要件

プロファイルに `.jar` ファイルを使用する場合、各プロファイルのマニフェスト・エントリには、次の文字列が含まれている必要があります。

```
SQLJProfile: TRUE
```

その場合、次の作業を行います。1) `.jar` ファイルに含めるプロファイルごとに 2 行 (PATH またはパッケージと名前に 1 行、前述の 1 行) でテキスト・ファイルを作成します。2) `jar` ユーティリティの `-m` オプションを使用して、このファイルを入力します。

この 2 行は連続していることが必要です (間に空白行を挿入しないでください)。ただし、プロファイルを追加する場合は、追加する 2 行の前に空白行を挿入する必要があります。

たとえば、`foo/bar` ディレクトリに、3 つのプロファイルを持つ `MyApp` というアプリケーションがあるとします。そして、それらのプロファイルを含む `.jar` ファイルを作成するとします。次の手順で実施してください。

1. 次のように、セパレータとして使用する空白行を含めて 8 行で構成されたテキスト・ファイルを作成します。ここでは、ファイルの名前を `MyAppJarEntries.txt` とします。

```
Name: foo/bar/MyApp_SJProfile0.ser
SQLJProfile: TRUE
```

```
Name: foo/bar/MyApp_SJProfile1.ser
SQLJProfile: TRUE
```

```
Name: foo/bar/MyApp_SJProfile2.ser
SQLJProfile: TRUE
```

2. `jar` を実行して `.jar` ファイルを作成する際に、次のように `-m` オプションを指定して、作成したテキストを入力します (ここでは `.jar` ファイルの名前を `myjarfile.jar` とします)。

```
jar -cvfm myjarfile.jar MyAppJarEntries.txt foo/bar/MyApp_SJProfile*.ser foo/bar/*.class
```

`.jar` ファイルの作成時に、`jar` ユーティリティがマニフェストを生成する場合、このユーティリティは指定されたテキスト・ファイルを読み込み、`SQLJProfile:TRUE` の行を各プロファイルのマニフェスト・エントリに挿入します。この処理は、マニフェスト内のプロファイル名と、テキスト・ファイルに指定されているプロファイル名が合致すると行われます。

JAR ファイルのロジスティクス

SQLJ コマンドラインで .jar ファイルを指定すると、その .jar 内の各プロファイルはシリアル化解除され、カスタマイズされます。

.jar ファイルのカスタマイズは、それに含まれるすべてのプロファイルが正しくカスタマイズされたときに、正常に実行されます。カスタマイズが正常に終了すると、各プロファイルは .ser ファイルに再びシリアル化されます。そして、.jar が変更されて、元の .ser ファイルが、カスタマイズ後の .ser で上書きされます。最後に、.jar マニフェスト・ファイルが更新され、新しいプロファイル・エントリが反映されます。

.jar ファイル内のプロファイルのカスタマイズ時に 1 度でもエラーが発生すると、.jar ファイルのカスタマイズは失敗し、元の .jar ファイルは完全には変更されません。

注意： 認証用にシグネチャ・ファイルを使用している場合、元の .jar ファイルに含まれているシグネチャ・ファイルは、変更されずに更新後の .jar ファイルに残ります。プロファイルへのアクセスに署名が必要な場合は、新しい .jar ファイルの署名を再登録する必要があります。

プロファイルのセマンティクス・チェック用の SQLCheckerCustomizer

Oracle には SQLCheckerCustomizer と呼ばれる特殊なカスタマイザが用意されており、トランスレータを以前実行した際に生成されたプロファイルに対してセマンティクス・チェックを実行できます。このセマンティクス・チェックは、ソース・コードの変換時に通常実行されるものとほとんど同じです。

このセマンティクス・チェックは、実行時に使用するデータベースが変換時のセマンティクス・チェックで使用できない場合に特に有効です。このような場合は、配布後にランタイム・データベースに対して SQLCheckerCustomizer を使用できます。このカスタマイザは、通常はソース・コードを使用しなくなった場合に使用します。

使用するチェッカは、指定可能です。SQLCheckerCustomizer では、デフォルトの OracleChecker フロントエンドを受け入れると、適切なオンライン・チェッカを使用したオンライン・セマンティクス・チェックが実行されます。

注意： プロファイルのオンライン・セマンティクス・チェックでは、カスタマイザ・ハーネスの接続オプションを使用して、接続パラメータを指定することも可能です。

カスタマイザ・ハーネスの verify オプションによる SQLCheckerCustomizer の起動

次に、Oracle カスタマイザ・ハーネスの verify オプションを指定して、SQLCheckerCustomizer をデフォルト・モードで実行する例を示します。デフォルトではオンライン・チェックが使用されるので、カスタマイザ・ハーネスの user、password および url オプションを使用して、接続パラメータを指定する必要があります。（最初の例ではコマンドラインが折り返されて表示されていますが、全体が 1 行で入力されています。）

```
sqlj -P-verify -P-user=scott -P-password=tiger -P-url=jdbc:oracle:oci:@
Foo_SJProfile0.ser Bar_SJProfile0.ser
```

```
sqlj -P-verify -P-user=scott -P-password=tiger -P-url=jdbc:oracle:oci:@ *.ser
```

verify オプションを指定すると、カスタマイザ・ハーネスのインスタンスが生成され、次のクラスが起動されます。

```
sqlj.runtime.profile.util.SQLCheckerCustomizer
```

プロファイル中の SQL 操作に対するセマンティクス・チェックは、このクラスによって調整されます。セマンティクス・チェックを指定することも、デフォルトの OracleChecker セマンティクス・チェックのフロントエンドを受け入れることも可能です。

次のオプションを使用すると、-P-verify オプションを使用した場合と同じ結果になります。

```
-P-customizer=sqlj.runtime.profile.util.SQLCheckerCustomizer
```

このオプションで指定したカスタマイザは、SQLJ の -default-customizer オプションで指定されたカスタマイザよりも優先されます。

注意：

- どの Oracle カスタマイザでも同様ですが、SQLJ コマンドラインで -P-verify と -P-help とを指定すると、ヘルプ出力とオプション・リストを見ることが可能です。
 - verify オプションを使用するとカスタマイザが起動しますが、SQLJ の 1 回の実行で実行できるカスタマイザは 1 つのみです。このため、このオプションを指定した場合は、他のカスタマイザを起動できません。
 - -P-print、-P-debug および -P-verify のうち、同時に使用できるのは 1 つのみです。このうちのどれを使用した場合にも、専用のカスタマイザが起動されるためです。
-
-

コマンドラインの構文 `sqlj -P-verify <conn params> profile_list`

コマンドラインの例 `sqlj -P-verify <conn params> Foo_SJProfile*.ser`

プロパティ・ファイルの構文 `profile.verify`

(SQLJ コマンドラインにはプロファイルを必ず指定してください。通常はカスタマイザ・ハーネスの接続オプションを指定します。)

プロパティ・ファイルの例 `profile.verify`

デフォルト値 該当なし

SQLCheckerCustomizer オプション

他のカスタマイザと同様に、SQLCheckerCustomizer のオプションも、SQLJ コマンドラインで接頭辞 `-P-C` を付けて設定できます。SQLJ プロパティ・ファイルでは、接頭辞 `profile.C` を使用します。

SQLCheckerCustomizer では、次のオプションがサポートされています。

- **checker:** 使用するセマンティクス・チェッカを指定するためのオプション。このオプションのデフォルトでは、OracleChecker フロントエンドが使用されます。これは、SQLJ 変換時のチェックの際も同様です。
- **warn:** プロファイルのセマンティクス・チェック時に出力される警告およびメッセージの分類を指定するためのオプション。このオプションは、SQLJ の `-warn` フラグと同じ機能があります。このフラグは、変換時のセマンティクス・チェックにおける警告の種類を出力するためのフラグですが、サポートされている設定値も使用されるデフォルトもこのオプションと同様です。8-44 ページの「[トランスレータからの警告 \(-warn\)](#)」を参照してください。

SQLCheckerCustomizer セマンティクス・チェッカのオプション (checker)

`checker` オプションを使用すると、プロファイル中の SQL 操作のチェックに使用するセマンティクス・チェッカを指定できます。

このオプションのデフォルトでは、Oracle セマンティクス・チェッカのフロントエンド `oracle.sqlj.checker.OracleChecker` が使用されます。このチェッカは、ユーザーの環境に適切なオンライン・チェッカとして SQLCheckerCustomizer で選択されます。OracleChecker の詳細は、8-68 ページの「[セマンティクス・チェッカおよび OracleChecker フロントエンド \(デフォルトのチェッカ\)](#)」を参照してください。

次に完全なコマンドラインの例を挙げて、SQLCheckerCustomizer の `checker` オプションと、カスタマイザ・ハーネスの `verify` オプションおよび接続オプションとを併用する方法を示します。

```
sqlj -P-verify -P-user=scott -P-password=tiger -P-url=jdbc:oracle:oci:@
-P-Cchecker=abc.def.MyChecker *.ser
```

(このコマンドラインは折り返されて表示されていますが、全体が 1 行で入力されています。)

コマンドラインの構文 -P-Cchecker=checker_class

コマンドラインの例 -P-Cchecker=a.b.c.MyChecker

プロパティ・ファイルの構文 profile.Cchecker=checker_class

プロパティ・ファイルの例 profile.Cchecker=a.b.c.MyChecker

デフォルト値 oracle.sqlj.checker.OracleChecker

SQLCheckerCustomizer の警告オプション (warn)

warn オプションでは、SQLJ トランスレータの -warn オプションと同様に、プロファイルのセマンティクス・チェック時に出力される警告やメッセージのカテゴリを選択できます。

これらのオプションの機能と使用可能な設定の詳細は、8-44 ページの「[トランスレータからの警告 \(-warn\)](#)」を参照してください。

このオプションのデフォルトでは、verbose および portable が有効化されていなければ、all, noverbose, noportable の設定値が使用され、すべての警告カテゴリが出力されます。このときに出力されるのは、継承階層の要件、データの精度、NULL 化可能なデータの変換による損失、および名前指定イテレータの厳密な合致に関する警告です。これらのデフォルトは、SQLJ 変換時の警告のデフォルトと同じです。

次に完全なコマンドラインの例を挙げて、SQLCheckerCustomizer の warn オプションと、カスタマイザ・ハーネスの verify オプションおよび接続オプションとを併用する方法を示します。このように指定すると、移植性に関する警告のみが出力されます。

```
sqlj -P-verify -P-user=scott -P-password=tiger -P-url=jdbc:oracle:oci:@
-P-Cwarn=none,portable *.ser
```

(折り返されて表示されていますが、全体が 1 行で入力されています。)

コマンドラインの構文 -P-Cwarn=comma-delimited_list_of_flags

コマンドラインの例 -P-Cwarn=none,verbose

プロパティ・ファイルの構文 profile.Cwarn=comma-delimited_list_of_flags

プロパティ・ファイルの例 `profile.Cwarn=none,verbose`

デフォルト値 `all,noverbose,noportable`

デバッグ用の AuditorInstaller カスタマイザ

ISO コード生成の場合、SQLJ では、AuditorInstaller という特別なカスタマイザが提供されます。このカスタマイザは、SQLJ コマンドラインで指定したプロファイルに、オーディタと呼ばれるデバッグ文のセットを挿入します。これらのプロファイルは、前回のカスタマイズによって生成済みであることが必要です。

アプリケーションを実行すると、SQLJ ランタイムでデバッグ文が実行され、メソッド・コールと戻り値のトレース情報が表示されます。

デバッグ文を挿入するには、カスタマイズの汎用オプションの場合と同じように、先頭に `-P-` を付けて、カスタマイザ・ハーネスの `debug` オプションを使用します。（このオプションの構文の詳細は、A-43 ページの「[カスタマイザ・ハーネスの -debug オプションによる AuditorInstaller の起動](#)」を参照してください。）

オーディタとコード・レイヤーの概要

Oracle カスタマイザでは、アプリケーションのカスタマイズ時に、ランタイム機能のレベルに応じたコードのレイヤー（通常のレイヤー数は 4 以下）に、プロファイルが実装されます。一番下のレイヤーは純粋な Oracle JDBC コールを使用し、JDBC の機能を通じて実行できる SQLJ 文を実装します。上位の各レイヤーは、JDBC ではサポートされていない SQLJ 機能用のレイヤーで、SQLJ ランタイムによってのみ処理されます。たとえば、JDBC 結果セットを SQLJ イテレータに変換するイテレータ変換文（CAST）用のレイヤーがあります。他には、代入文（SET）用のレイヤーがあります。

コード・レイヤーでは、実行時に各 SQLJ 実行文が、まず一番上のレイヤーに渡されます。次に、その文を処理できるレイヤー（通常は、すべての JDBC コールを実行できるレイヤー）まで、1 レベルずつレイヤーを経過します。

AuditorInstaller の 1 回の実行でデバッグ文を挿入できるレイヤーは 1 つです。特定のコード・レイヤーに挿入されたデバッグ文のセットを総称して、オーディタと呼びます。実行時には、レイヤーにコールが渡されるたびに、そのレイヤーに挿入されているオーディタが起動されます。

通常、JDBC レイヤーより上にある特別なコード・レイヤーは、本来のデバッグ対象ではありません。このため、オーディタは一番下か一番上のレイヤーに挿入します。一番上のレイヤーにオーディタを挿入してランタイム・デバッグを行うと、すべての SQLJ 実行文からのメソッド・コールのトレースが出力されます。一番下のレイヤーにオーディタを挿入してランタイム・デバッグを行うと、JDBC コールを戻すすべての SQLJ 実行文からのメソッド・コールのトレースが出力されます。

複数のレベルにオーディタを挿入するには、AuditorInstaller を複数回実行します。これは、一番下のレイヤーと一番上のレイヤーにオーディタを挿入する場合などに行います。

オーディタを挿入するレイヤーの指定方法の詳細は、A-46 ページの「[AuditorInstaller の depth オプション \(depth\)](#)」を参照してください。

カスタマイザ・ハーネスの -debug オプションによる AuditorInstaller の起動

次に、Oracle カスタマイザ・ハーネスの -debug オプションを指定して、AuditorInstaller をデフォルト・モードで実行する例を示します。

```
sqlj -P-debug Foo_SJProfile0.ser Bar_SJProfile0.ser
```

```
sqlj -P-debug *.ser
```

```
sqlj -P-debug myappjar.jar
```

debug オプションを指定すると、カスタマイザ・ハーネスのインスタンスが生成され、次のクラスが起動されます。

```
sqlj.runtime.profile.util.AuditorInstaller
```

上に示したクラスは、デバッグ文の挿入作業を実行するクラスです。

次のオプションを使用すると、-P-debug オプションを使用した場合と同じ結果になります。

```
-P-customizer=sqlj.runtime.profile.util.AuditorInstaller
```

このオプションで指定したカスタマイザは、SQLJ の -default-customizer オプションで指定されたカスタマイザよりも優先されます。

次の点に留意してください。

- オーディタを挿入してアプリケーションを実行するには、CLASSPATH に Oracle SQLJ ファイル translator.jar (または .zip) があることが必要です。(通常、変換前の SQLJ アプリケーションを実行するには、ランタイム・ライブラリのみが必要です。)
- どの Oracle カスタマイザの場合も、SQLJ コマンドラインで -P-debug と -P-help とを指定すると、ヘルプ出力とオプション・リストを見ることが可能です。
- -debug オプションを使用するとカスタマイザが起動しますが、SQLJ の 1 回の実行で実行できるカスタマイザは 1 つのみです。このため、このオプションを指定した場合は、他のカスタマイザを起動できません。
- -P-print、-P-debug および -P-verify のうち、同時に使用できるのは 1 つのみです。このうちのどれを使用した場合にも、専用のカスタマイザが起動されるためです。

コマンドラインの構文 sqlj -P-debug profile_list

コマンドラインの例 sqlj -P-debug Foo_SJProfile*.ser

プロパティ・ファイルの構文 profile.debug

(ファイル・リストのプロファイルにも指定します。)

プロパティ・ファイルの例 profile.debug

デフォルト値 該当なし

AuditorInstaller のランタイム出力

AuditorInstaller でデバッグ文を挿入すると、実行時にコールされたメソッドと戻り値がトレースされます。この処理は、デバッグ文が挿入されたすべてのプロファイル・レイヤーに対して行われます。実行時にはデバッグ出力を限定できません。

AuditorInstaller の出力は、プロファイルのみに対応付けられています。元の .sqlj ソース・ファイル内の行とのマッピングは、現行の時点ではサポートされていません。

次は、AuditorInstaller のランタイム出力の例の抜粋です。この出力は、SQLJ の SELECT INTO 文の出力と似ています。

```
oracle.sqlj.runtime.OraProfile@1 . getProfileData ( )
oracle.sqlj.runtime.OraProfile@1 . getProfileData returned
sqlj.runtime.profile.ref.ProfileDataImpl@2
oracle.sqlj.runtime.OraProfile@1 . getStatement ( 0 )
oracle.sqlj.runtime.OraProfile@1 . getStatement returned
oracle.sqlj.runtime.OraRTStatement@3
oracle.sqlj.runtime.OraRTStatement@3 . setMaxRows ( 1000 )
oracle.sqlj.runtime.OraRTStatement@3 . setMaxRows returned
oracle.sqlj.runtime.OraRTStatement@3 . setMaxFieldSize ( 3000 )
oracle.sqlj.runtime.OraRTStatement@3 . setMaxFieldSize returned
oracle.sqlj.runtime.OraRTStatement@3 . setQueryTimeout ( 1000 )
oracle.sqlj.runtime.OraRTStatement@3 . setQueryTimeout returned
oracle.sqlj.runtime.OraRTStatement@3 . setBigDecimal ( 1 , 5 )
oracle.sqlj.runtime.OraRTStatement@3 . setBigDecimal returned
oracle.sqlj.runtime.OraRTStatement@3 . setBoolean ( 2 , false )
oracle.sqlj.runtime.OraRTStatement@3 . setBoolean returned
oracle.sqlj.runtime.OraRTStatement@3 . executeRTQuery ( )
oracle.sqlj.runtime.OraRTStatement@3 . executeRTQuery returned
oracle.sqlj.runtime.OraRTResultSet@6
oracle.sqlj.runtime.OraRTStatement@3 . getWarnings ( )
oracle.sqlj.runtime.OraRTStatement@3 . getWarnings returned null
oracle.sqlj.runtime.OraRTStatement@3 . executeComplete ( )
oracle.sqlj.runtime.OraRTStatement@3 . executeComplete returned
```

```

oracle.sqlj.runtime.OraRTResultSet@6 . next ( )
oracle.sqlj.runtime.OraRTResultSet@6 . next returned true
oracle.sqlj.runtime.OraRTResultSet@6 . getBigDecimal ( 1 )
oracle.sqlj.runtime.OraRTResultSet@6 . getBigDecimal returned 5
oracle.sqlj.runtime.OraRTResultSet@6 . getDate ( 7 )
oracle.sqlj.runtime.OraRTResultSet@6 . getDate returned 1998-03-28

```

メソッド・コールごとに、2行出力されます。1行目はそのコールと入力パラメータ、2行目は戻り値を示します。

注意： oracle.sqlj.runtime パッケージ内に含まれるクラスは、SQLJ ランタイム・クラスです。これらのクラスは、同じような名前の付いた JDBC のクラスと同じ機能を持ちます。たとえば、OraRTResultSet は、JDBC ResultSet インタフェースによる SQLJ ランタイム実装で、JDBC のクラスと同じ属性およびメソッドが含まれています。

AuditorInstaller オプション

他のカスタマイザと同様に、AuditorInstaller のオプションも、SQLJ コマンドラインで接頭辞 -P-C を付けて設定できます (SQLJ プロパティ・ファイルでは、profile.C を使用します)。

AuditorInstaller では、次のオプションがサポートされています。

- **depth** — プロファイル内のランタイム機能のレイヤーをどこまで下がるのかを指定するためのオプション。
- **log** — 挿入されたオーディタのデバッグ文からのランタイム出力を書き込むファイルを指定するためのオプション。
- **prefix** — 挿入されたデバッグ文からのランタイム出力の各行に付ける接頭辞を指定するためのオプション。
- **showReturns** — ランタイム・コールをトレースした際の戻りの引数を、挿入済みのオーディタに書き込むためのオプション。
- **showThreads** — ランタイム・コールをトレースした際のスレッド名を、挿入済みのオーディタに書き込むためのオプション (マルチスレッド・アプリケーション専用)。
- **uninstall** — プロファイルに対して、前回 AuditorInstaller が起動したときにプロファイルに挿入されたデバッグ文を削除するためのオプション。

AuditorInstaller の depth オプション (depth)

A-42 ページの「[オーディタとコード・レイヤーの概要](#)」で説明したように、AuditorInstaller では、オーディタと呼ばれるデバッグ文のセットを 1 回の実行で 1 つのコード・レイヤーにのみ挿入します。AuditorInstaller の depth オプションを使用すると、挿入先のレイヤーを指定できます。複数のレベルにオーディタを挿入するには、AuditorInstaller を複数回実行します。

レイヤーには、整数で番号が付けられます。一番上がレイヤー 0 (ゼロ) になり、レイヤーの階層は通常最大で 2 つまたは 3 つになります。通常、一番上のレイヤーには 0 (ゼロ)、一番下のレイヤーには -1 を設定します。実際には、これ以外の特定のレイヤーにオーディタを挿入するのは困難です。各種の SQLJ 実行文に使用されるレイヤー番号が、公開されていないためです。

depth オプションは、prefix オプションとともに設定できます。AuditorInstaller を複数回実行して、異なるレイヤーに異なる接頭辞を指定すると、実行時に出力される情報がどのレイヤーのものか確認できます。

depth オプションが設定されていない場合や、プロファイルに指定されているレイヤー数を超えた値が指定された場合、オーディタは一番下のレイヤーに挿入されます。

コマンドラインの構文 -P-Cdepth=n

コマンドラインの例 -P-Cdepth=0

プロパティ・ファイルの構文 profile.Cdepth=n

プロパティ・ファイルの例 profile.Cdepth=0

デフォルト値 -1 (一番下のレイヤー)

AuditorInstaller の log_file オプション (log)

log オプションを使用すると、現行の実行で挿入しているオーディタからのランタイム出力を書き込むファイルを指定できます。このオプションが設定されていない場合は、標準出力が使用されます。つまり、デバッグ結果は SQLJ メッセージと同じ出力先に出力されます。

オーディタが出力ファイルに書き込むメッセージは、以前の内容を上書きするのではなく、追加されます。そのため、複数のオーディタに対して同じログ・ファイルを指定しても、競合は起こりません。通常、アプリケーション内のすべてのレイヤーからのデバッグ情報は、同じログ・ファイルに書き込まれます。

コマンドラインの構文 -P-Clog=log_file

コマンドラインの例 -P-Clog=foo/bar/mylog.txt

プロパティ・ファイルの構文 profile.Clog=log_file

プロパティ・ファイルの例 `profile.Clog=foo/bar/mylog.txt`

デフォルト値 指定なし（標準出力を使用）

AuditorInstaller の prefix オプション (prefix)

prefix オプションを使用して、ランタイム出力の各行に付ける接頭辞を指定します。この出力は、今回の実行で起動する AuditorInstaller によって挿入されるデバッグ文からの結果です。

通常、このオプションは、depth オプションとともに使用されます。AuditorInstaller を複数回実行して、各レイヤーに異なる接頭辞を指定すると、実行時に出力される情報がどのレイヤーのものか確認できます。

コマンドラインの構文 `-P-Cprefix="string"`

コマンドラインの例 `-P-Cprefix="layer 2: "`

プロパティ・ファイルの構文 `profile.Cprefix="string"`

プロパティ・ファイルの例 `profile.Cprefix="layer 2: "`

デフォルト値 指定なし

AuditorInstaller の戻り引数オプション (showReturns)

showReturns オプションは、ランタイム・コールのトレース時に戻り引数の出力を有効化 / 無効化するためのオプションです。デフォルトでは、出力が有効化されます。

showReturns を有効にした場合（デフォルト）のサンプル出力を数行のみ次に示します。

```
oracle.sqlj.runtime.OraRTStatement@3 . executeComplete ( )
oracle.sqlj.runtime.OraRTStatement@3 . executeComplete returned
oracle.sqlj.runtime.OraRTResultSet@6 . next ( )
oracle.sqlj.runtime.OraRTResultSet@6 . next returned true
oracle.sqlj.runtime.OraRTResultSet@6 . getBigDecimal ( 1 )
oracle.sqlj.runtime.OraRTResultSet@6 . getBigDecimal returned 5
oracle.sqlj.runtime.OraRTResultSet@6 . getDate ( 7 )
oracle.sqlj.runtime.OraRTResultSet@6 . getDate returned 1998-03-28
```

showReturns を無効にした場合の出力は、次のようになります。

```
oracle.sqlj.runtime.OraRTStatement@3 . executeComplete ( )
oracle.sqlj.runtime.OraRTResultSet@6 . next ( )
oracle.sqlj.runtime.OraRTResultSet@6 . getBigDecimal ( 1 )
oracle.sqlj.runtime.OraRTResultSet@6 . getDate ( 7 )
```

各メソッドのコールに必要なのはコール行のみであって、コール行と戻り行の両方を使用する必要はありません。

コマンドラインの構文 -P-CshowReturns<=true|false>

コマンドラインの例 -P-CshowReturns=false

プロパティ・ファイルの構文 profile.CshowReturns<=true|false>

プロパティ・ファイルの例 profile.CshowReturns=false

デフォルト値 true

AuditorInstaller のスレッド名オプション (showThreads)

showThreads オプションは、ランタイム・コールのトレース時にスレッド名の出力を有効化 / 無効化するためのオプションです (マルチスレッド・アプリケーション専用)。デフォルトでは、出力が無効化されます。

このオプションを有効化すると、トレース出力中のメソッド名の先頭にスレッド名が付加されます。

コマンドラインの構文 -P-CshowThreads<=true|false>

コマンドラインの例 -P-CshowThreads

プロパティ・ファイルの構文 profile.CshowThreads<=true|false>

プロパティ・ファイルの例 profile.CshowThreads

デフォルト値 false

AuditorInstaller の uninstall オプション (uninstall)

uninstall オプションを使用すると、前回 AuditorInstaller を起動したときに挿入されたデバッグ文が削除されます。uninstall オプションを使用するたびに、最後に挿入されたオーディタが削除されます。

プロファイル内のすべてのオーディタを削除するには、プロファイルが変更されなかったことを示すメッセージが表示されるまで、AuditorInstaller を繰り返し実行します。

コマンドラインの構文 -P-Cuninstall

コマンドラインの例 -P-Cuninstall

プロパティ・ファイルの構文 `profile.Cuninstall`

プロパティ・ファイルの例 `profile.Cuninstall`

デフォルト値 無効

完全なコマンドラインの例

ここでは、AuditorInstaller の各種オプションを指定した完全な SQLJ コマンドラインの例を示します。

次のコマンドラインはデバッグ文のセット、つまりオーディタを一番下のレイヤー（デフォルトのレイヤー）に挿入し、ランタイム出力を標準出力に出力します。

```
sqlj -P-debug MyApp_SJProfile*.ser
```

次のコマンドラインは、オーディタを一番下のレイヤーに挿入し、ランタイム出力を `log.txt` に出力します。

```
sqlj -P-debug -P-Clog=foo/bar/log.txt MyApp_SJProfile*.ser
```

次のコマンドラインは、オーディタを一番下のレイヤーに挿入し、ランタイム出力を標準出力に出力します。この指定では、スレッド名は表示されますが、戻り引数は表示されません。

```
sqlj -P-debug -P-CshowThreads=true -P-CshowReturns=false MyApp_SJProfile*.ser
```

オーディタをレイヤー 0（一番上のレイヤー）に挿入します。ランタイム出力を `log.txt` に送信し、ランタイム出力の各行に「Layer 0: 」という接頭辞を付けます（次のコマンドラインは折り返されて表示されていますが、全体が 1 行で入力されています）。

```
sqlj -P-debug -P-Clog=foo/bar/log.txt -P-Cdepth=0 -P-Cprefix="Layer 0: "  
MyApp_SJProfile*.ser
```

次のコマンドラインは、オーディタを削除します。（このコマンドを実行すると、最後に挿入されたオーディタが削除されます。すべてのオーディタを削除するには、このコマンドを繰り返し実行します。）

```
sqlj -P-debug -P-Cuninstall MyApp_SJProfile*.ser
```

SQLJ エラー・メッセージ

この付録では、SQLJ トランスレータと SQLJ ランタイムによって出力されるエラー・メッセージを示します。原因と処置の他に、ランタイム・エラーの場合は SQL の状態も示します。

- [SQLJ 変換時のメッセージ](#)
- [SQLJ ランタイム・メッセージ](#)

注意： この付録には、英語のエラー・メッセージが掲載されていますが、製品では実行時の環境により、英語か日本語のどちらかの出力となります。また、原因および処置は日本語で掲載されていますが、製品では英語で出力されます。

SQLJ 変換時のメッセージ

ここでは、SQLJ トランスレータから出力されるエラー・メッセージおよびその原因と処置の一覧を示します。

注意： SQLJ トランスレータの `-explain` フラグを有効にすると、エラー・メッセージの出力時に、原因と処置についての情報もあわせて出力できます。その内容は、次のエラー・リストに示す内容と同じです。8-47 ページの「[トランスレータ・エラーの原因と処置 \(-explain\)](#)」を参照してください。

<<<NEW SQL>>>

原因： 次のメッセージに示すように、Oracle カスタマイザによって、SQL 操作が Oracle 固有の言語に変換されています。これらのメッセージは、Oracle カスタマイザの「showSQL」オプションを有効にすると、出力されます。

処置： これは単なる情報メッセージです。特に必要な処置はありません。

[Connecting to user *user* at *connection*]

原因： SQLJ がユーザー *user* として URL *connection* のデータベースに接続することをユーザーに通知します。

[Preserving SQL checking info]

原因： 今回のオンライン・チェックで取得される分析情報が、SQLJ に保存されます。

[Querying database with "*sqlquery*"]

原因： データベース問合せが発行されたことをユーザーに通知します。

[Re-using cached SQL checking information]

原因： 前回のオンライン・チェックでキャッシュされた分析結果が再利用されていることをユーザーに通知します。

[Registered JDBC drivers: *class*]

原因： 登録されている JDBC ドライバをリストします。

[SQL checking: read *m* of *n* cached objects.]

原因： オンライン・チェックでキャッシングされた分析情報が取り出されました。

[SQL function call "*sqlj call*" transformed into ODBC syntax "*jdbc call*"]

原因： SQLJ ファンクション・コール構文が、JDBC ファンクション・コール構文に変換されたことをユーザーに通知します。

A call to a stored function must return a value.

原因： ストアド・ファンクション・コールからの戻り値の処理が定義されていません。

A call to a stored procedure cannot return a value.

原因： ユーザーが戻り値を取得しようとして、ストアド・プロシージャを起動しました。

A non-array type cannot be indexed.

原因: 配列アクセス演算子 ([]) の基底オペランドとして使用できるのは、配列型のみです。

処置: 基底オペランドの型を確認します。

A SQL quote was not terminated.

処置: 引用符を閉じる " または ' を挿入します。

Access modifiers *modifier1* and *modifier2* are not compatible.

原因: 名前付きアクセス修飾子は同じクラス、メソッドまたはメンバーに適用できません。たとえば、`private` と `public` はアクセス修飾子として両立しません。

処置: 競合するアクセス修飾子のいずれかを変更または削除します。

Ambiguous column names *columns* in SELECT list.

原因: 大文字と小文字の違いのみで区別される列名は使用できません。

処置: 列名を区別するために、列の別名を使用します。

Ambiguous constructor invocation.

原因: 標準変換後、複数のコンストラクタ宣言が引数に一致します。

処置: 使用するコンストラクタの引数型を明示的に指定します。

Ambiguous method invocation.

原因: 標準変換後、オーバーロードされた複数のメソッド宣言が引数に合致します。

処置: 使用するメソッドの引数型を明示的に指定します。

An error occurred when determining result set column sizes: *message*

原因: `-P-Coptcols` オプションが指定されました。プロファイル・カスタマイザが結果セットの列の型とサイズを決定しようとしたときに、エラーが発生しました。

処置: SQL 文を確認します。接続して変換処理を行うことで、エラーの原因を詳しく突きとめることもできます。

an io error occurred while generating output: *message*

処置: SQLJ 出力のための適切なアクセス権と十分な領域があることを確認します。

Anonymous classes are not allowed in bind expressions.

原因: ホスト式には無名クラスを指定できません。

処置: 無名クラスが指定されている式を `#sql` 文の外側に移動し、その値を有効な型の一時変数に保存します。そして、その一時変数をホスト式で使います。

Argument #*n* of *name* must be a host variable, since this argument has mode OUT or INOUT.

原因: モードが OUT および INOUT の場合、引数位置に変数または代入可能な式（配列変数など）が指定されていることが必要です。

Argument #*n* of *name* requires mode IN.

原因: ストアド・プロシージャまたはストアド・ファンクション *name* では、ホスト式 #*n* のモードが IN であることが必要です。

処置: SQLJ 文内のホスト式を IN として宣言します。

Argument #*n* of *name* requires mode INOUT.

原因: ストアド・プロシージャまたはストアド・ファンクション *name* では、ホスト式 #*n* のモードが INOUT であることが必要です。

処置: SQLJ 文内のホスト式を INOUT として宣言します。

Argument #*n* of *name* requires mode OUT.

原因: ストアド・プロシージャまたはストアド・ファンクション *name* では、ホスト式 #*n* のモードが OUT であることが必要です。

処置: SQLJ 文内のホスト式を OUT として宣言します。

Argument #*pos* is empty.

原因: ストアド・ファンクションまたはストアド・プロシージャの引数リストで、*pos* 番目にある引数に何も指定されていません。次にその例を示します。proc(1, , :x)

処置: 空の引数をホスト式または SQL 式に置き換えます。

Arithmetic expression requires numeric operands.

原因: 算術演算の左右両側とも数値型であることが必要です。

処置: オペランドの型を修正します。

Array index must be a numeric type.

原因: 数値索引を使用した場合のみ配列オブジェクトの索引を作成できます。

処置: 索引オペランドの型を修正します。

Attributes *attribute1* and *attribute2* are not compatible.

原因: 指定された属性は同じクラスまたはメソッドに適用できません。たとえば、abstract と final は属性として両立しません。

処置: 競合する属性のいずれかを変更または削除します。

auditing layer added

原因: 監査カスタマイザを使用して、プロファイルがカスタマイズされました。

処置: プロファイルの使用時に、監査コールを含みます。特に必要な処置はありません。オーディタを削除するには、uninstall オプションを使用します。

auditing layer removed

原因: プロファイルに組み込まれた最新の監査カスタマイズの内容が削除されました。複数のオーディタが組み込まれている場合は、直前に組み込まれたもののみが削除されます。

処置: さらに他のオーディタを削除する場合は、`uninstall` コールを指定する必要があります。

backup created as filename

原因: プロファイルのバックアップ・ファイルが *filename* という名前で作成されました。バックアップ・ファイルには、カスタマイズ前の元のプロファイルが保存されています。

処置: 特に必要な処置はありません。バックアップ・ファイルを新しいプロファイルにコピーすると、元のプロファイルをリストアできます。

bad filename: filename

原因: カスタマイザ・ハーネス・ユーティリティへの入力として、ファイル *filename* は使用できません。「.ser」または「.jar」という拡張子のファイル名のみサポートされます。

処置: ファイル名を変更して、有効な拡張子を指定します。

Bad octal literal 'token'.

原因: 数字 0 で始まる数値リテラルは 8 進数として解析されます。このため、リテラル内に 8 や 9 の数字を含んではいけません。

処置: 不正なリテラルを変更します。8 進数を指定する場合、8 をベースとしてその値を再計算します。10 進数を指定する場合は、先行するすべての 0 (ゼロ) を削除します。

Badly placed #sql construct -- not a class declaration.

原因: 宣言を指定する場所に、実行可能な SQLJ 文が指定されています。

処置: `#sql` コンストラクタを正しい場所に移動します。

Bitwise operator requires boolean or numeric operands.

原因: ビット単位演算子は、両方のオブジェクトがブール型または数値型の場合のみ処理を行います。2 つのオブジェクトのカテゴリが異なる場合は、ビット単位演算は失敗します。

処置: オペランドの型を確認します。

Boolean operator requires boolean operands.

原因: ブール演算子は、ブール型の引数に対してのみ処理を行います。

処置: オペランドの型を確認します。

cannot access option *option name*

原因: *option name* という名前のオプションで、カスタマイザ・ハーネスにアクセスできません。標準ではないカスタマイザ固有のオプションである可能性があります。

処置: オプションの用途を確認します。エラーを回避するには、オプションの使用を中止するか、別のカスタマイザを使用します。

Cannot analyze SQL statement online: unable to determine SQL types for *count host items*.

原因: SQLJ では、各 Java ホスト式に対する SQL 型が決められています。これらの SQL 型は、文のオンライン・チェックに必要です。

処置: Oracle SQLJ でサポートされている Java 型を使用します。

Cannot determine default arguments for stored procedures and functions. May need to install SYS.SQLJUTL.

原因: パッケージ SYS.SQLJUTL で宣言されているファンクションが見つかりません。

処置: SQL ファイル \$ORACLE_HOME/sqlj/lib/sqljutl.sql を見つけ、それを実行します。なお、ストアド・ファンクションまたはストアド・プロシージャがデフォルトの引数を使用していない場合は、このメッセージを無視します。

Cannot load JDBC driver class *class*.

処置: JDBC ドライバの名前 *class* を確認します。

Cannot load type map from resource *map name*.

処置: 型マップ・リソースのマップ名が CLASSPATH にあることを確認します。

cannot remove java file without first compiling it

原因: プロファイル変換ユーティリティに対して nc オプションと rj オプションが同時に指定されています。クラス・ファイルにコンパイルされていないかぎり、Java ファイルを削除できません。

処置: nc オプションと rj オプションのいずれか一方を使用します。

Cannot resolve identifier because the enclosing class has errors.

原因: エラーを含むクラスは、名前解決で使用できません。アクセス権は、完全なクラスのものに割り当てられるためです。

処置: ベース型、フィールド型、メソッドの引数型およびメソッドの戻り型のスペルに注意して、クラスを修正します。また、ベース名でのみ参照される外部クラスがインポートされていることも確認してください。

Cannot retrieve type map for context class *context class*: error message

原因: 接続コンテキスト・クラス *context class* の型マップを取得しようとするときに、エラーが発生しました。

cannot specify both *option name* and *option name options*

原因: プロファイル変換ユーティリティに対して、両立しない2つのオプションが同時に指定されています。

処置: 指定されたオプションのいずれか一方を使用します。

Class *class* does not implement the checker interface.

原因: チェッカは、`sqlj.framework.checker.SQLChecker` を実装している必要があります。

Class *classname* not found.

原因: *classname* というクラスへの参照がプログラムに記述されています。このクラスの定義は、現在変換中のソース・ファイル中にも CLASSPATH 中にも見つかりませんでした。

処置: クラス名を確認します。CLASSPATH 中のクラス形式またはトランスレータに渡されるソース・ファイルにおいてクラス名が定義されることを確認します。

class cannot be constructed as an iterator: *class name*

原因: この SQL 操作で使用されているイテレータ・クラス *class name* のコンストラクタが適切ではありません。標準ではないトランスレータによって生成されたイテレータの可能性がありえます。

処置: 標準トランスレータを使用して、イテレータの宣言を再変換します。

class has already been defined: *classname*

原因: クラス *classname* が、SQLJ に渡すソース・ファイルのうち、1つのファイル内でのみ定義されていることを確認します。

class has errors - unable to resolve method *method()*.

原因: SQLJ 変換では、Java クラスに `oracle.sql.CustomDatum` または `java.sql.SQLData` インタフェースが正しく実装されているかどうかを確認できません。これは Java クラス内のエラーが原因です。

処置: Java クラスの問題を修正します。個別にコンパイルして、クラス上のエラーを確認することもできます。

class implements both `sqlj.runtime.NamedIterator` and `sqlj.runtime.PositionedIterator`: *class name*

原因: この SQL 操作で使用されているイテレータ・クラス *class name* が、名前指定イテレータであるか、位置指定イテレータであるかを特定できません。イテレータが標準ではないトランスレータによって生成されたか、またはその `implements` 句のインタフェースにエラーが含まれている可能性があります。

処置: イテレータ宣言の `implements` 句に、問題の原因となるインタフェースが含まれていないことを確認します。標準トランスレータを使用して、イテレータの宣言を再変換します。

Class not found: *mesg*. The problem is likely due to the fact that either your program or the SQLJ runtime references *javax.sql.DataSource*.

原因: *javax.sql.DataSource* と静的にリンクしている、接続コンテキストや *runtime12ee.zip* などの SQLJ ランタイム・バージョンに、WITH 属性「*dataSource*」を使用しているようです。

処置: CLASSPATH に *javax.sql.** および *javax.naming.** パッケージが指定されていることを確認します。または、接続コンテキスト宣言から「*dataSource*」属性を削除して、*runtime12ee.zip* を使用しないようにします。

Column *javatype* column not found in SELECT list.

処置: 問合せによって戻された結果セットの中に、列 *column* が見つかりません。イテレータ宣言を修正するか、別名を使用して、SELECT 文を修正します。

Column *name1* #*pos1* will cause column *name2* #*pos2* to be lost. Use a single stream column at the end of the select list.

原因: 位置指定イテレータに指定できるストリーム列は 1 つ以下で、この列はイテレータの最終列である必要があります。

処置: ストリーム列をイテレータの最終位置に移動します。ストリーム列が複数ある場合は、名前指定イテレータを使用し、ストリーム列（およびその他の列）が順にアクセスされるようにします。

Column *type* column is not compatible with database type *sqltype*

原因: Java 型と SQL 型の互換性がありません。

Comparison operator requires numeric operands.

原因: 大きさを比較する操作では、数値のみが有効です。

処置: オペランドの型を確認します。

compatible with the following drivers:

原因: Oracle カスタマイズの「*compat*」オプションが有効になっています。このメッセージの後に、現行のプロファイルで使用できる Oracle JDBC ドライバのバージョンがリストされます。

処置: リストされた JDBC ドライバのいずれかのバージョンを使用して、プログラムを実行します。

compiling *filename*

原因: ファイル *filename* 内のプロファイルが、プロファイル変換ユーティリティによってクラス・ファイル形式にコンパイルされました。

処置: 特に必要な処置はありません。

Complement operator requires integral operand.

原因: ビット単位で補数演算できるのは、整数のみです。

処置: オペランドの型を確認します。

Conditional expression requires boolean for its first operand.

原因: 条件式は、先頭オペランドを使用して、他の 2 つのオペランドのどちらを実行するかを選択します。このため、先頭オペランドは、ブール型であることが必要です。

処置: 先頭オペランドの型を確認します。

Conditional expression result types must match.

原因: 条件式の値は、2 番目のオペランドと 3 番目のオペランドのいずれかになります。両方ともブール型または数値型またはオブジェクト型で、型が一致している必要があります。

処置: オペランドの型を確認します。

Connection context expression does not have a Java type.

原因: 接続コンテキスト式に、有効な Java 型が指定されていません。

Connection context must have been declared with #sql context ... It can not be declared as a ConnectionContext.

処置: 接続コンテキスト型を `#sql context ConnectionContext;` で宣言します。

ConnectionContext attribute *attribute* is not defined in the SQLJ specification.

処置: `with` 句の属性 *attribute* は、SQLJ 仕様にありません。指定した属性名のスペルを確認します。

ConnectionContext cannot implement the *interface* interface.

原因: SQLJ コンテキスト宣言で、`implements` 句にインタフェース *interface* が指定されています。ただし、接続コンテキストでは、このインタフェースは実装されません。

Constructor not found.

原因: コンストラクタの起動が試みられましたが、見つかりません。

処置: コンストラクタの引数を確認します。または、コンストラクタに必要な引数を追加します。

Context *context* ignored in FETCH statement.

原因: 問合せによるカーソルの初期化時に、コンテキストがカーソル・オブジェクトに対応付けられます。このため、`FETCH` 文のコンテキスト情報が余分となり、SQLJ では無視されます。

converting profile *filename*

原因: ファイル *filename* 内のプロファイルが、プロファイル変換ユーティリティによって、シリアル化ファイルから Java ソース・ファイル形式に変換されました。

処置: 特に必要な処置はありません。

Cursor has *item count* items. Argument *#pos* of INTO-list is invalid.

原因: INTO リストには、フェッチ元の位置指定イテレータよりも多くの要素があります。

処置: 余分な INTO リスト項目を削除します。

Cursor type in FETCH statement does not have a Java type.

原因: FETCH 文のイテレータ式に、有効な Java 型が指定されていません。

customized

原因: プロファイルが正しくカスタマイズされました。

処置: 特に必要な処置はありません。

customizer does not accept connection: *connection url*

原因: *connection url* で指定された接続が確立されました。ただし、現行のカスタマイザにとって不要であるか、またはカスタマイザで認識できません。

処置: 現行のカスタマイザが接続を要求していることを確認します。接続が不要な場合は、カスタマイザ・ハーネスから「user」オプションを削除します。接続が必要な場合は、接続先のデータベースとスキーマが、カスタマイザと互換性があることを確認します。

Database error during signature lookup for stored procedure or function name: *message*

原因: SQLJ トランスレータがファンクションまたはプロシージャ *name* の存在とシグネチャを特定する際に、エラーが発生しました。

処置: エラーを回避するには、SQLJ プログラムをオフラインで変換します。

Database issued an error: *error*.

原因: 基本スキーマに対して SQL 文を解析中に、データベースでエラーが発生しました。

処置: SQL 文の妥当性を確認します。

Database issued an error: *error sqltext*

原因: 基本スキーマに対して SQL 文を解析中に、データベースでエラーが発生しました。

処置: SQL 文の妥当性を確認します。

deleting *filename*

原因: 中間ファイル *filename* が、プロファイル変換ユーティリティによって削除されました。

処置: 特に必要な処置はありません。

Did not find a stored procedure or function *name* with *n* arguments.

原因: データベースに、*name* という名前で *n* 個の引数を持つプロシージャまたはファンクションは存在しません。

処置: ストアド・プロシージャまたはストアド・ファンクションの名前を確認します。

Did not find a stored procedure or function *name* with *n* arguments. *found functions/procedures with different numbers of arguments*

原因: データベースに、*name* という名前で *n* 個の引数を持つプロシージャまたはファンクションは存在しません。ただし、同じ名前で引数の個数が異なるプロシージャまたはファンクションは存在します。

処置: ストアド・プロシージャまたはストアド・ファンクションの名前と、引数の指定の過不足を確認します。

Did not find stored function *name* with *n* arguments.

原因: 指定された *name* という名前のストアド・ファンクションは見つかりません。

処置: ストアド・ファンクションの名前を確認します。

Did not find stored function *proc* with *n* arguments. *found functions/procedures with different numbers of arguments*

原因: データベースに、*proc* という名前で *n* 個の引数を持つストアド・ファンクションは存在しません。ただし、同じ名前で引数の個数が異なるプロシージャまたはファンクションは存在します。

処置: ストアド・ファンクションの名前と、引数の指定の過不足を確認します。

Did not find stored procedure *name* with *n* arguments.

原因: 指定された *name* という名前のストアド・プロシージャは見つかりません。

処置: ストアド・プロシージャの名前を確認します。

Did not find stored procedure *proc* with *n* arguments. *found functions/procedures with different numbers of arguments*

原因: データベースに、*proc* という名前で *n* 個の引数を持つストアド・プロシージャは存在しません。ただし、同じ名前で引数の個数が異なるプロシージャまたはファンクションは存在します。

処置: ストアド・プロシージャの名前と、引数の指定の過不足について確認します。

Do not know how to analyze this SQL statement.

原因: この文を分析するには、オンライン接続を行うことが必要です。

Do not understand this statement.

原因: 先頭に SQL キーワード (SELECT、UPDATE、DELETE、BEGIN、...) または SQLJ キーワード (CALL、VALUES、FETCH、CAST、...) がないため、この文を識別できません。

Duplicate access modifier.

原因: 同じクラス、メソッドまたはメンバーに対して、同じアクセス修飾子が 2 回以上指定されています。

処置: 余分なアクセス修飾子を削除します。

Duplicate method *method*.

原因: メソッド *method* が 2 回以上宣言されています。

Duplicate methods *method1* and *method2*.

原因: メソッド *method1* および *method2* が同じ SQL 名にマッピングされています。名前指定イテレータの宣言では、同じ SQL 名にマッピングされているメソッドを 2 つ指定することは不可能です。

Equality operator operand types must match.

原因: 等価演算子でオブジェクトを比較するには、オブジェクトが両方ともブール型または数値型またはオブジェクト型で、型が一致している必要があります。

処置: 等価演算子のオペランドの型を確認します。

error converting profile: *filename*

原因: ファイル *filename* 内のプロファイルをシリアル化ファイルからクラス・ファイル形式に変換する際に、エラーが発生しました。このメッセージの後に、エラーの詳細が表示されます。

処置: エラーの詳細を参照し、適宜修正します。

Error in Java compilation: *message*

原因: Java コンパイラを起動して、.java ソース・ファイルをコンパイルする際にエラーが発生しました。

処置: -compiler-executable フラグに正しい Java コンパイラが指定され、そのコンパイラが PATH に指定されていることを確認します。または、-passes オプションを使用して、Java コンパイラを SQLJ からではなくコマンドラインから起動します。

error loading customizer harness

原因: カスタマイザ・ハーネス・ユーティリティを正しく初期化できません。Java Runtime Environment に互換性がない可能性があります。

処置: Java Runtime Environment が JRE 1.1 以上と互換性があることを確認します。

Expected "*token1*" and found "*token2*" instead.

原因: この文の構文では終了のトークン *token1* が必要ですが、それが指定されていません。

Expected 'FROM' to follow 'SELECT ...INTO ...'

原因: SELECT 文の構文が正しくありません。

処置: INTO 句の後に FROM 句を追加します。

Expected cast to be assigned to an iterator, found that cast was assigned to *type*.

原因：キャスト代入の左辺には、*type* 型の式ではなく、SQLJ イテレータのインスタンスを指定する必要があります。

Expected cast to be assigned to an iterator.

原因：SQLJ キャスト文では、代入文の左辺が SQLJ イテレータのインスタンスである必要があります。

Expected cursor host variable or NEXT, or PRIOR, or FIRST, or LAST, or ABSOLUTE, or RELATIVE.

原因：イテレータ型のホスト変数またはキーワードを指定する必要があります。

Expected cursor host variable. Encountered: "*token*"

原因：イテレータ型のホスト変数を指定する必要があります。

Expected end of cast statement. Found "*token*" ...

原因：キャスト文の後に予期しないトークン *token* があります。

Expected end of FETCH statement. Encountered: "*token*"

原因：この FETCH 文で、トークンの指定はこれ以上必要ありません。

Expected host variable of type *java.sql.ResultSet*, found "*token*" ...

原因：CAST キーワードの後にホスト変数が指定されていません。

Expected host variable of type *java.sql.ResultSet*, found host variable of invalid Java type.

原因：ホスト式に有効な Java 型が指定されていません。

Expected host variable of type *java.sql.ResultSet*, found host variable of type *type*.

原因：ホスト式の型は Java 型の *type* であり、要求された *java.sql.ResultSet* ではありません。

処置：*java.sql.ResultSet* 型のホスト式を使用します。必要な場合は、Java キャストを使用して、式をこの型にキャストできます。

Expected host variable of type *java.sql.ResultSet*.

原因：SQLJ キャスト文では *java.sql.ResultSet* がイテレータ型に代入されます。変換を試みた型は、*java.sql.ResultSet* ではありません。

処置：*java.sql.ResultSet* 型のホスト式を使用する必要があります。必要な場合は、Java キャストを使用して、式をこの型にキャストできます。

Expected INTO bind expression.

原因：この文には、1 つ以上の INTO ホスト式を指定する必要があります。

expected ODBC function call syntax "{ call func(...) }".

原因：ストアド・プロシージャをコールする際に使用する JDBC エスケープ構文が無効です。

Expected stored function name. Found: *token*

原因: トークン *token* ではなく、ストアド・ファンクションの名前を指定する必要があります。

Expected stored function or procedure name. Found: *token*

原因: トークン *token* ではなく、ストアド・ファンクションまたはストアド・プロシージャの名前を指定する必要があります。

Expected stored procedure name. Found: *token*

原因: トークン *token* ではなく、ストアド・プロシージャの名前を指定する必要があります。

Expected: FETCH : cursor INTO ...

原因: FETCH 文には、値のフェッチ元であるカーソル・ホスト変数を指定する必要があります。

Expected: WHERE CURRENT OF :hostvar. Found: WHERE CURRENT *token* ...

処置: 正しい構文で WHERE CURRENT OF 句を指定します。

Expected: WHERE CURRENT OF :hostvar. Found: WHERE CURRENT OF *token* ...

処置: 正しい構文で WHERE CURRENT OF 句を指定します。

field "*field name*" in class name is not a class name type

原因: カスタム Java クラス *class name* 内のフィールド *field name* が、必要な型である *class name* ではありません。クラスと Oracle データベースの型間で正しい変換を行うには、この型のフィールドが必要です。

処置: カスタム Java クラスに、必要な型でフィールド *field name* を宣言します。

field "*field name*" in class name is not accessible

原因: フィールド *field name* は、カスタム Java クラス *class name* 内で **public** と指定されていません。クラスと Oracle データベースの型間で正しい変換を行うには、このフィールドが必要です。

処置: フィールド *field name* をカスタム Java クラスで **public** として宣言します。

field "*field name*" in class name is not uniquely defined

原因: カスタム Java クラス *class name* で、*field name* という名前のフィールドが 2 つ以上見つかりました。このエラーは、*class name* で 2 つの異なるインタフェースが実装され、この 2 つのインタフェースで *field name* が定義されている場合に起こります。クラスと Oracle データベースの型間で正しい変換を行うには、一意に定義されたフィールドが必要です。

処置: *field name* が 1 度のみ定義されるように、カスタム Java クラスを更新します。

field "*field name*" not found in class name

原因: カスタム Java クラス *class name* に、*field name* という名前のフィールドはありません。クラスと Oracle データベースの型間で正しい変換を行うには、このフィールドが必要です。

処置: カスタム Java クラスに、必要なフィールドを宣言します。

Field not accessible.

原因: このクラスは、フィールドにアクセスできません。

処置: フィールドのアクセス権が正しく設定されていることを確認します。

File *fileName* does not contain type *className* as expected. Please adjust the class path so that the file does not appear in the unnamed package.

原因: クラス *className* が、SQLJ トランスレータに渡すファイル *fileName* で定義されていることを確認します。

file too large

原因: JAR ファイルに含まれているプロファイル・ファイルが大きすぎて、カスタマイズできません。

処置: JAR ファイルの一部としてではなく、1つのファイルとしてプロファイル・ファイルを解凍して、カスタマイズします。

filename must be a valid java identifier: *filename*

原因: ファイル名が Java 識別子として無効です。SQLJ では、入力ファイルの名前に基づいて、クラスとリソースの定義が追加されます。このため、名前を Java 識別子として使用できることが必要です。

処置: ファイル名を変更し、Java 識別子として使用できるようにします。

found incompatible types

原因: プロファイルに含まれる型に、どの Oracle JDBC ドライバにもサポートされない型があります。

処置: プログラムから互換性のない型を削除します。互換性のない型は、「summary」オプションで出力される型のリストに含まれています。

Host item #*n* cannot be OUT or INOUT.

原因: 位置 #*n* にあるホスト項目は SQL 式に埋め込まれ、ストアド・プロシージャまたはストアド・ファンクションの引数になります。このため、この引数位置はモード IN であることが必要です。なお、このメッセージは、引数を名前でバインドするときにも表示されます。

処置: 引数のモードを IN に変更します。OUT または INOUT の引数を名前でバインドしている場合は、このメッセージを無視します。

Host item #pos must be an lvalue.

原因: 位置 *pos* の OUT または INOUT ホスト式は、代入可能な式であることが必要です。Java 変数、フィールドおよび配列要素は、代入可能な式です。

Host item name (at position #n) cannot be OUT or INOUT.

原因: 位置 *#n* にあるホスト項目 *name* は SQL 式に埋め込まれ、ストアド・プロシージャまたはストアド・ファンクションの引数になります。このため、この引数位置はモード IN であることが必要です。なお、このメッセージは、引数を名前でバインドするときにも表示されます。

処置: 引数のモードを IN に変更します。OUT または INOUT の引数を名前でバインドしている場合は、このメッセージを無視します。

Identifier identifier may not begin with __sJT_.

処置: `__sJT_` で始まる識別子を使用していないことを確認します。

Ignored type map entries: entry list.

原因: 接続コンテキスト型マップ上に 1 つ以上の標準でない、移植不可能なエントリが見つかり、無視されました。

ignoring context name context name

原因: 接続コンテキスト *context name* に対応付けられているプロファイルが見つかりました。このコンテキストはカスタマイザ・ハーネス「context」オプション・リストに含まれていないので、このプロファイルはカスタマイズされていません。

処置: 必要に応じて、必要なコンテキストを含む context を設定して、カスタマイザを再実行します。

Illegal entry for option option. Expected a boolean value, received: "value"

処置: *option* にはブール値 (true、false、yes、no、0、1 など) を使用します。

Illegal INTO ... bind variable list: error.

原因: INTO リストを構成する要素の中に、有効な Java 型ではないものが 1 つ以上あります。

Illegal Java type in cursor for WHERE CURRENT OF

原因: WHERE CURRENT OF 句のイテレータに、有効な Java 型が指定されていません。

Illegal token 'token' will be ignored.

原因: どの Java トークンにも一致しない文字列が、ソース・ファイルに含まれています。

処置: ソース・ファイルを変更してエラーを修正し、ソース・ファイルが有効な Java ソース・コードで構成されていることを確認します。

illegal value: オプション設定

原因：オプションに設定されている値が範囲外か、または無効です。

処置：メッセージの詳細を参照し、その内容に従ってオプション値を修正します。

In FETCH clause: expected *expected token or expression*.

原因：FETCH 句に特定の構文キーワードまたは式が指定されていません。

IN mode is not allowed for INTO-variables.

原因：INTO 変数は Java で値を戻します。

処置：かわりに OUT を使用します（これはデフォルトなので、指定子も省略できます）。

Inaccessible Java type for host item #*n*: *type*.

原因：Java クラス *type* は、public 属性を持つクラスではありません。このため、ドライバによってインスタンス化できません。

処置：ホスト式では、public な Java 型を使用します。

Inaccessible Java type for host item *name* (at position #*n*): *type*.

原因：ホスト式 *name* は Java 型 *type* ですが、public 属性を持っていません。このため、ドライバによってインスタンス化できません。

処置：ホスト式では、public な Java 型を使用します。

Inaccessible Java type for item #*pos* of INTO-list: *type*.

原因：INTO リスト項目 *pos* の Java クラス *type* は、public 属性を持つクラスではありません。このため、ドライバによってインスタンス化できません。

処置：INTO リスト内では、public な Java 型を使用します。

Increment/decrement operator requires numeric operand.

原因：インクリメント演算子とデクリメント演算子は、整数値に対してのみ処理を行います。

処置：オペランドの型を確認します。

Initialization lists are not allowed in bind expressions.

原因：ホスト式には、初期化リストを指定できません。

処置：初期化リストを使用している式を #sql 文の外側に移動し、その値を有効な型の一時変数に保存します。そして、その一時変数をホスト式で使用します。

INOUT mode is not allowed for INTO-variables.

原因：INTO 変数は Java で値を戻します。

処置：かわりに OUT を使用します（これはデフォルトなので、指定子も省略できます）。

Instance of operator requires an object reference operand.

原因: 演算子のインスタンスは、オブジェクトに対してのみ処理を行います。

処置: オペランドの型を確認します。

INTERNAL ERROR SEM-label. Should not occur - please notify.

処置: エラー・メッセージの内容をオラクル社カスタマ・サポート・センターに連絡してください。

INTO-list item #*position* must be an lvalue.

原因: INTO リスト項目は、代入可能な式であることが必要です。Java 変数、フィールドおよび配列要素は、代入可能な式です。

INTO-lists may only occur in SELECT and FETCH statements.

原因: 現行の SQL 文には、INTO... バインド・リストを指定できません。

Invalid CustomDatum or SQLData implementation in type: *mesg*

原因: oracle.sql.CustomDatum または java.sql.SQLData インタフェースを実装するユーザー定義 Java 型 *type* を使用しています。しかし、メッセージの詳細にあるように、ユーザー定義型に対するすべての要件に型が適合していません。

処置: ユーザー定義の型での問題を修正します。または、JPublisher ユーティリティを使用して、ユーザー定義の型を生成します。

Invalid bind variable or expression.

原因: バインド変数（問合せの戻り値を保存するときに使用するホスト変数、コンテキスト式またはイテレータ式など）が Java 構文として無効です。

処置: ホスト変数または式を修正します。

Invalid cursor type in FETCH statement: *type*.

処置: FETCH 文のイテレータは、sqlj.runtime.FetchableIterator を実装する必要があります。

Invalid iterator declaration.

原因: SQL 宣言に構文エラーがあります。

処置: SQL 宣言の構文を確認します。

Invalid Java type *classname* in entry "*typemap entry*".

原因: Java クラス名は、必ず Java 環境に存在する有効な Java クラスの名前にします。

Invalid Java type *java type* in map at entry "*entry*"

原因: *java type* 型は、有効な Java クラスの名前ではありません。

Invalid Java type for host item #*n*.

原因: ホスト式 #*n* に、有効な Java 型が指定されていません。

Invalid Java type for host item #n: error.

原因: ホスト式 #n に、有効な Java 型が指定されていません。

Invalid Java type for host item #position: classname. Oracle SQL では、この型はサポートされていません。

原因: Oracle SQLJ ランタイムでは、この型のインスタンスをデータベースに書き込むことはサポートされていません。

処置: 問題の型が oracle.sql.STRUCT、oracle.sql.REF または oracle.sql.ARRAY である場合、oracle.XXX 型のかわりに JPublisher で生成されたラッパー・クラスを使用できます。

Invalid Java type for host item name (at position #n).

原因: ホスト式 name (位置 #n) に、有効な Java 型が指定されていません。

Invalid Java type for host item name (at position #n): error.

原因: ホスト式 name (位置 #n) に、有効な Java 型が指定されていません。

Invalid Java type for item #pos of INTO-list: type.

原因: INTO リスト項目 #pos に、有効な Java 型が指定されていません。

invalid option "option name" set from option origin: problem description

原因: オプション option name の値が無効です。

処置: problem description にあわせてオプション値を修正します。

invalid option: オプション設定

原因: option setting で指定されたオプションが、カスタマイザ・ハーネスで認識されません。

処置: 不明なオプションを修正または削除します。

invalid profile name: profile name

原因: JAR ファイルの MANIFEST ファイルに指定されている SQLJ プロファイル・エントリが、JAR ファイルに含まれていません。

処置: 指定のプロファイルを JAR ファイルに追加するか、またはそのエントリを MANIFEST ファイルから削除します。

Invalid SQL iterator declaration.

原因: 宣言された SQLJ 型のインスタンスは、その宣言にエラーがあるか不明な点があるために、完全には処理できません。

処置: イテレータ列型リストに示される型に注意して、SQL イテレータの宣言を確認します。また、ベース名のみで参照されている場合は、型がインポートされていることを確認します。

Invalid SQL string.

原因: SQL 文に構文エラーがあります。

処置: デリミタ（終了のカッコ、中カッコまたは大カッコ、引用符、コメント・デリミタなど）の不足に特に注意して、SQL 文の構文を確認します。

Invalid SQL type in entry "entry" of type map type map message.

原因: エントリ *entry* 内の SQL 型が正しく指定されていないか、エントリが重複しています。

Invalid type cast

原因: オブジェクトを指定された型にキャストできません。

処置: オペランドの型を確認します。

Item #pos of INTO-list does not have a Java type.

原因: INTO リスト項目 #pos に、有効な Java 型が指定されていません。

Iterator iterator must implement the interface interface.

原因: このイテレータで使用される動作コマンドのために、*interface* インタフェースを実装する必要があります。

処置: 次のようにイテレータ型を宣言します。

`#sql iterator iterator implements interface (...);`

iterator class name must implement either sqlj.runtime.NamedIterator or sqlj.runtime.PositionedIterator

原因: この SQL 操作で使用されているイテレータ・クラス *class name* は、名前指定イテレータと位置指定イテレータのいずれでもありません。標準ではないトランスレータによって生成されたイテレータの可能性があります。

処置: 標準トランスレータを使用して、イテレータの宣言を再変換します。

Iterator attribute attribute is not defined in the SQLJ specification.

処置: *with* 句の属性 *attribute* は、SQLJ 仕様がありません。指定した属性名のスペルを確認します。

Iterator with attribute updateColumns must implement sqlj.runtime.ForUpdate

処置: イテレータ宣言で、`implements` 句を `implements sqlj.runtime.ForUpdate` と指定します。

JAR does not contain MANIFEST file

原因: JAR ファイルに MANIFEST ファイルが含まれていません。MANIFEST ファイルは、JAR ファイルに含まれているプロファイルを示すために必要です。

処置: MANIFEST を JAR ファイルに追加します。JAR ファイルに含まれる各プロファイルに対して、「SQLJProfile=TRUE」という行を MANIFEST に指定する必要があります。

JAR MANIFEST file format unknown

原因: JAR MANIFEST ファイルが不明な形式で作成されているので、JAR ファイルをカスタマイズできません。

処置: JDK MANIFEST ファイル形式指定に従って、JAR ファイルを再度作成します。
jar ユーティリティで生成される MANIFEST ファイルは、この形式に従います。

Java class *class* specified in *type map* does not implement *interface*.

原因: コンテキスト型マップ *type map* に従って、クラス *class* にインタフェース *type map* を実装する必要があります。この場合はそのようになっていません。

Java class *class* specified in *type map* implements neither *interface1* nor *interface2*.

原因: コンテキスト型マップ *type map* に従って、クラス *class* にインタフェース *interface1* または *interface2* インタフェースを実装する必要があります。この場合はそのようになっていません。

Java class *classname* in entry "*entry*" must implement *interface*

原因: データベースから読み書きされた Java オブジェクトのインスタンスに、特定の Java インタフェースを実装する必要があります。

Java type *javatype* for column *column* is illegal.

原因: *javatype* の Java クラス宣言が無効です。

Java type *type* of iterator for WHERE CURRENT OF is not supported. It must implement *sqlj.runtime.ForUpdate*.

原因: WHERE CURRENT OF 句のイテレータは、インタフェース *sqlj.runtime.ForUpdate* を実装するものとして宣言する必要があります。

JDBC does not specify that column *column type* is compatible with database type *sqltype*. Conversion is non-portable and may result in a runtime error.

処置: 別の JDBC ドライバへの移植性を最大限にするには、この変換を避けます。

JDBC reports a mode other than IN/OUT/INOUT/RETURN for *name* in position *n*.

原因: ストアド・プロシージャまたはストアド・ファンクションの引数に対して、不明なモードが報告されました。

処置: ストアド・ファンクションまたはストアド・プロシージャが正しく定義されていることを確認します。場合によっては、JDBC ドライバを更新します。

JDBC reports an error during the retrieval of argument information for the stored procedure/function *name*: *error*.

処置: エラーが原因で、このファンクションまたはプロシージャのモードを特定できません。エラーを回避できない場合は、変換を繰り返すか、オフラインで変換します。

JDBC reports more than one return value for *name*.

原因: JDBC ドライバがストアド・プロシージャまたはストアド・ファンクションに対して、間違って複数の戻り引数を報告しました。

処置: JDBC ドライバを更新します。

JDBC reports the return value for *function* in position *pos* instead of position 1.

原因: JDBC ドライバが、最初のストアド・ファンクションの戻り引数を正しく報告していません。

処置: JDBC ドライバを更新します。

Left hand side of assignment does not have a Java type.

原因: 代入文の左辺の式に、有効な Java 型が指定されていません。

list item value may not be empty

原因: 「driver」や「context」など、値をリスト形式で出力するオプションに、空のリスト項目が指定されています。

処置: リストから空の項目を削除します。

Loss of precision possible in conversion from *sqltype* to column *column type*.

原因: SQL の数値を Java に変換すると、精度が低下する場合があります。

Method name *method* is reserved by SQLJ.

原因: SQLJ では、イテレータに対して各種のメソッドが事前に定義されています。これらのメソッド名は、使用できません。

Method not accessible.

原因: このクラスは、メソッドにアクセスできません。

処置: メソッドのアクセス権が正しく設定されていることを確認します。

Method not found.

原因: メソッドが見つかりません。

処置: メソッドの引数を確認します。または、オーバーロードしたメソッドに必要な引数を追加します。

Missing *count* elements in INTO list: 型

原因: FETCH 文のフェッチ・カーソル上の列数が、INTO バインド変数リストで要求された列数よりも少なくなっています

Missing closing ")" on argument list of stored procedure/function call.

処置: 引数リストの最後には、「)」を指定します。

Missing colon.

原因: 必要な箇所にコロンが指定されていません。

処置: 不足するコロンを追加します。

Missing comma.

原因：必要な箇所にカンマが指定されていません。

処置：不足するカンマを追加します。

Missing curly brace.

原因：必要な箇所に開始中カッコが指定されていません。

処置：不足する開始中カッコを追加します。

Missing dot operator.

原因：必要な箇所にドット演算子が指定されていません。

処置：不足するドット演算子を追加します。

Missing element in INTO list: *element*

処置：*element* を INTO リストに追加する必要があります。

Missing equal sign in assignment.

原因：戻り変数の場所に Java 式がありますが、式の後に等号がありません。代入構文では、式の後に等号が必要です。

処置：不足する代入演算子を追加します。

Missing parenthesis.

原因：必要な箇所に開始カッコが指定されていません。

処置：不足する開始カッコを追加します。

Missing semicolon.

原因：必要な箇所にセミコロンが指定されていません。

処置：不足するセミコロンを追加します。

Missing square bracket.

原因：必要な箇所に開始大カッコが指定されていません。

処置：不足する開始大カッコを追加します。

Missing terminating "*token*".

原因：SQL 文に対応しないトークン *token* があります。

Mode of left-hand-side expression in SET statement was changed to OUT.

原因：SET :*x* = ... 文で、ホスト式 *x* のモードが IN または INOUT として指定されています。これは正しくありません。

処置：モードを省略するか、またはモードを OUT として指定します。

Modifier *modifier* not allowed in declaration.

原因: SQLJ クラス宣言には、どの修飾子でも指定できるわけではありません。

Modifier *modifier* not allowed in top-level declarations.

原因: SQLJ クラス宣言には、どの修飾子でも指定できるわけではありません。

More than one INTO ... bind list in SQL statement.

処置: 余分な INTO ... バインド・リストを削除します。

moving *original filename* to *new filename*

原因: プロファイル変換ユーティリティによって、プロファイルのバックアップが生成されました。バックアップ・ファイルの名前は、*new filename* です。

処置: 特に必要な処置はありません。

Must be connected online to perform optimization for result set columns.

原因: -P-Coptcols オプションが指定されました。プロファイル・カスタマイザからデータベースへのログインが不可能であるため、結果セットの列型と列サイズを決定できません。

処置: -P-user、-P-password および -P-url オプションを使用し、接続情報を指定します。

Must specify STRUCT or JAVA_OBJECT in entry "*entry*"

原因: Java クラスに対応する SQL 型には、構造化された型 (STRUCT xxx) または Java オブジェクト・インスタンスを保持できる SQL 型 (JAVA_OBJECT xxx) のいずれかを指定する必要があります。

Name '*illegal identifier*' cannot be used as an identifier.

原因: 文字列 *illegal identifier* は、他の言語の要素 (演算子、区切り記号、制御構造など) に相当するために、識別子として使用できません。

処置: 識別子に別の名前を使用します。

Negation operator requires boolean operand.

原因: 否定演算子は、ブール・オペランドに対してのみ処理を行います。

処置: オペランドの型を確認します。

No ";" permitted after stored procedure/function call.

原因: SQLJ では、ストアド・プロシージャまたはストアド・ファンクションの呼出しコードの後に、終了のセミコロンを指定できません。

No connect string specified for context *context*.

原因: *context* に対して JDBC 接続 URL が指定されていません。

処置: -url@*context* オプションまたは -user@*context* オプションに、JDBC URL を指定します。

No connect string specified.

原因: JDBC 接続 URL が指定されていません。

処置: -url オプションまたは -user オプションに、JDBC URL を指定します。

No connection specified for context *context*. Will attempt to use connection *defaultconnection* instead.

原因: *context* のオンライン・チェックに、接続情報が明示的に指定されていません。その場合、デフォルトのオンライン基本スキーマの値が使用されます。

no customizer specified

原因: プロファイルのカスタマイズが要求されましたが、カスタマイザが指定されていません。

処置: 「customizer」または「default-customizer」オプションを使用して、プロファイル・カスタマイザを設定します。

No instrumentation: class already instrumented.

原因: このクラス・ファイルは、元の .sqlj ファイルにあわせて、すでにインストールメントされています。

No instrumentation: no line info in class.

原因: このクラス・ファイルには行情報がないので、インストールメントできません。Java コンパイラの -O (optimize) フラグが使用された可能性があります。このオプションが有効になっていると、クラス・ファイルから行情報が削除されます。

No INTO variable for column #pos: "name" type

原因: SELECT-INTO 文で、位置 #pos にある型 *type* の列 *name* には、対応する Java ホスト式がありません。

処置: INTO リストを拡張するか、または SELECT 文を変更します。

No offline checker specified for context *context*.

原因: *context* に対して、オフライン分析を実行できません。

No offline checker specified.

原因: オフライン分析を実行できません。

No online checker specified for context *context*. Attempting to use offline checker instead.

原因: オンライン・チェックが要求された場合でも、*context* にはオフライン・チェックが実行されます。

No online checker specified. Attempting to use offline checker instead.

原因: オンライン・チェックが要求された場合でも、オフライン・チェックが実行されます。

No SQL code permitted after stored procedure/function call. Found: "token" ...

原因: SQLJ では、ストアド・プロシージャまたはストアド・ファンクションの呼出しコードの後に、別の文を指定できません。

No suitable online checker found for context *context*. Attempting to use offline checker instead.

原因: *context* をチェックできるオンライン・チェッカがありません。

No suitable online checker found. Attempting to use offline checker instead.

原因: デフォルト・コンテキストをチェックできるオンライン・チェッカがありません。

No user specified for context *context*. Will attempt to connect as user *user*.

原因: デフォルト・コンテキストにユーザーが指定されている場合は、すべてのコンテキストに対してオンライン・チェックが試みられます。

No variable *name* defined in class *classname*

原因: *name* という変数は、クラス *classname* の中には見つかりませんでした。

処置: この変数の有無と、名前付きクラスでこの変数を参照可能かどうかを確認します。

not a directory: *name*

原因: `-d` オプションまたは `-dir` オプションによって、ルート・ディレクトリ *name* を最上位とするディレクトリ階層内に、出力ファイルを作成することが指定されています。ルート・ディレクトリが存在し、書込み可能であることを確認します。

not a valid input filename: *filename*

原因: SQLJ トランスレータへの入力ファイルの拡張子は、「.sqlj」、「.java」、「.ser」または「.jar」である必要があります。

Not an interface: *name*

原因: 名前 *name* が `implements` 句で使用されています。しかし、これは Java インタフェースを表すものではありません。

Not an original sqlj file - no instrumentation.

原因: クラス・ファイルのコンパイル元である Java ファイルが、SQLJ トランスレータによって生成されていません。

Not found: *name*. There is no stored procedure or function of this name.

原因: ストアド・ファンクションまたはストアド・プロシージャが見つかりません。

option is read only: *option name*

原因: *option name* という名前の読取り専用オプションに対して、オプション値が指定されました。

処置: オプションの用途を確認します。

Option optparamdefaults: Invalid JDBC type in *size hint*

原因: ユーザーが指定した -P-Coptparamdefaults オプションには、<JDBC-type>(<number>) 形式または <JDBC-type>() 形式のサイズ・ヒントのカンマ区切りリストが含まれています。<JDBC-type> に属さない型としては、CHAR、VARCHAR、VARCHAR2、LONG、LONGVARCHAR、BINARY、RAW、VARBINARY、LONGVARBINARY、LONGRAW、wildcard XXX% (このワイルドカードで示される型は、前述の型や CHAR_TYPE、RAW_TYPE など) が挙げられます。

Option optparamdefaults: Invalid or missing size indicator in *size hint*

原因: ユーザーが指定した -P-Coptparamdefaults オプションには、サイズ・ヒントのカンマ区切りリストが含まれており、そのうちの 1 つ以上は <JDBC-type>(<number>) 形式または <JDBC-type>() の形式になっていません。

Oracle features used:

原因: Oracle カスタマイザの「summary」オプションが有効になっています。このメッセージの後に、現行のプロファイルで使用されている Oracle 固有の型と機能がリストされます。

処置: 移植性を高くするには、リストされた型と機能をプログラムから削除します。

PLEASE ENTER PASSWORD FOR *user* AT *connection* >

処置: ユーザー・パスワードを入力し、<Enter> キーを押します。

positioned update/delete not supported

原因: 特定の行を参照するには、ROWID を選択して使用します。

処置: プロファイルに、SQL の位置指定更新または位置指定削除が指定されています。Oracle では、この操作は実行時に実行できません。

Premature end-of-file.

原因: クラス宣言が完了する前に、ソース・ファイルが終了しました。

処置: ソース・ファイルで、引用符の不足、カッコ、中カッコまたは大カッコの配置と不足、コメントの区切り記号の不足を調べます。また、有効な Java クラスが 1 つ以上指定されていることも確認します。

Public class *class name* must be defined in a file called *filename.sqlj* or *filename.java*

原因: Java では、クラス名と、そのクラスを定義しているソース・ファイルのベース名とを同じ名前にする必要があります。

処置: クラスまたはファイルの名前を変更します。

Public declaration must reside in file with base name *name*, not in the file *file*.

処置: SQLJ ファイル名と public なクラス名が一致することを確認します。

re-installing Oracle customization

原因: 旧バージョンの Oracle カスタマイザを使用して、プロファイルがすでにカスタマイズされています。古いカスタマイズ内容は、新しいカスタマイズ内容に上書きされました。

処置: プロファイルは、Oracle 用にカスタマイズされています。特に必要な処置はありません。

recursive iterators not supported: iterator name

原因: SQL 操作で、再帰的に定義されているイテレータ型が使用されています。再帰定義のイテレータ型「A」とは、そのいずれかの列型に最終的に「A」を含むイテレータのことです。イテレータが最終的に「A」を含むと見なされるのは、「A」の列型を持つか、またはその列型に最終的に「A」を含む場合です。

処置: 再帰的ではないイテレータを使用します。

registering Oracle customization

原因: Oracle カスタマイザを使用して、プロファイルがカスタマイズされました。

処置: プロファイルは、Oracle 用にカスタマイズされています。特に必要な処置はありません。

Repeated host item name in positions pos1 and pos2 in SQL block. Behavior is vendor-defined and non portable.

原因: ホスト変数 *name* が、モード OUT または INOUT で複数の位置に指定されています。または、OUT や INOUT に加えて、モード IN で指定されています。

処置: ホスト変数は参照で渡されるのではなく、それぞれ値で個別に渡されます。このエラーを防ぐには、各 OUT または INOUT 位置に個別のホスト変数を使用します。

Result expression must be an lvalue.

原因: SQLJ 代入文の左側は、代入可能な式であることが必要です。Java 変数、フィールドおよび配列要素は、代入可能な式です。

Return type javatype of stored function is not legal.

原因: ストアド・ファンクションの戻り値の Java 型 *javatype* が、有効な Java クラスを参照していません。

Return type type is not a visible Java type.

原因: 型 *type* は、public 属性を持つ Java 型ではありません。このため、この型のインスタンスを生成したり、データベース・ドライバから戻したりできません。

処置: 型 *type* を public として宣言します。

Return type type is not supported in Oracle SQL.

原因: Java 型 *type* は、SQL 文の戻り値とはなりません。

Return type type of stored function is not a JDBC output type. This will not be portable.

原因: 移植性を最大限にするには、JDBC 仕様に従った型を使用します。

Return type *type* of stored function is not a visible Java type.

原因: 型 *type* は、public 属性を持つ Java 型ではありません。このため、この型のインスタンスを生成したり、データベース・ドライバから戻したりできません。

処置: 型 *type* を public として宣言します。

Return type incompatible with SELECT statement: *type* is not an iterator type.

処置: 値を戻す SQL 問合せは、java.sql.ResultSet、位置指定イテレータ・オブジェクトまたは名前指定イテレータ・オブジェクトに代入する必要があります。

Select list has only *n* elements. Column type #*pos* is not available.

原因: データベース問合せから戻された列数が、イテレータまたは INTO ホスト変数リストで要求される列数よりも少なくなっています。

処置: 問合せを変更するか、または INTO リストから要素を削除します。

Select list has only one element. Column type #*pos* is not available.

原因: データベース問合せから戻された列数が、イテレータまたは INTO ホスト変数リストで要求される列数よりも少なくなっています。

処置: 問合せを変更するか、または INTO リストから要素を削除します。

Shift operator requires integral operands.

原因: シフト演算子は、数値オペランドに対してのみ処理を行います。

処置: オペランドの型を確認します。

Sign operator requires numeric operand.

原因: 符号演算子は、数値オペランドに対してのみ処理を行います。

処置: オペランドの型を確認します。

Size designation *size hint* for parameter *param* ignored.

原因: パラメータ *param* にサイズ・ヒントが指定されましたが、本来このパラメータには可変サイズ型は指定できません。ただし、このパラメータは可変サイズ型ではありません。そのため、このサイズ・ヒントは無視されます。

SQL checker did not categorize this statement.

原因: 指定された SQL チェッカが、この SQL 文を分類できません。

処置: SQL チェッカは、すべての SQL 文を分類する必要があります。使用している SQL チェッカ (-online および -offline オプション) を確認します。

SQL checking did not assign mode for host variable #*n* - assuming IN.

原因: 指定された SQL チェッカが、このホスト変数にモード情報を代入していません。モードは IN と見なされます。

処置: SQL チェッカが、すべてのホスト式にモードを代入する必要があります。使用している SQL チェッカ (-online および -offline オプション) を確認します。

SQL checking did not assign mode for host variable #n.

原因: 指定された SQL チェッカが、このホスト変数にモード情報を代入していません。モードは IN と見なされます。

処置: SQL チェッカが、すべてのホスト式にモードを代入する必要があります。使用している SQL チェッカ (-online および -offline オプション) を確認します。

SQL checking did not assign mode for host variable *name* (at position #n) - assuming IN.

原因: 指定された SQL チェッカが、このホスト変数にモード情報を代入していません。モードは IN と見なされます。

処置: SQL チェッカが、すべてのホスト式にモードを代入する必要があります。使用している SQL チェッカ (-online および -offline オプション) を確認します。

SQL checking did not assign mode for host variable *name* (at position #n).

原因: 指定された SQL チェッカが、このホスト変数にモード情報を代入していません。モードは IN と見なされます。

処置: SQL チェッカが、すべてのホスト式にモードを代入する必要があります。使用している SQL チェッカ (-online および -offline オプション) を確認します。

SQL statement could not be categorized.

原因: この SQL 文は、SELECT、UPDATE、DELETE、...、CALL、VALUES、FETCH、CAST などの認識可能な SQL キーワードまたは SQLJ キーワードで開始されていません。

処置: SQL 文の構文を確認します。

SQL statement does not return a value.

原因: プログラムに指定されている代入文が、問合せでもストアド・ファンクションの呼出しでもありません。問合せとファンクションのみが、即時に結果を戻すことが可能です。

SQL statement with INTO ... bind variables can not additionally return a value.

処置: INTO ... バインド・リストを削除するか、またはイテレータへの代入を削除します。

SQLJ declarations cannot be inside method blocks.

原因: メソッド・ブロックに SQLJ 宣言を指定できません。

処置: SQLJ 宣言をメソッド・ブロック・スコープからクラス・スコープまたはファイル・スコープに移動します (必要に応じて、あいまいさを防ぐために、宣言した型とその型へのすべての参照の名前を変更します)。

SQLJ runtime library is missing. You need to provide *sqlj runtime library* on the CLASSPATH.

原因: リリース 8.1.7 より、`translator.zip` ライブラリに SQLJ ランタイム・クラスはありません。

処置: `runtime.zip`、`runtime11.zip` または `runtime12.zip` のいずれか 1 つが CLASSPATH または `-classpath` オプションに指定してあることを確認します。JDBC および Java 環境によっては、エラー・メッセージに特定のランタイム・バージョンが表示されます。

Statement execution expression does not have a Java type.

原因: 実行コンテキスト式に有効な Java 型が指定されていません。

Stored function or procedure syntax does not follow SQLJ specification.

原因: ストアド・ファンクションでは `VALUES(...)` 構文を使用し、ストアド・プロシージャでは `CALL ...` 構文を使用します。

処置: 指定されたファンクションおよびプロシージャ構文は、SQLJ で解釈できます。ただし、SQLJ プログラムの移植性を最大限にするには、指示されている構文を使用します。

Stored function syntax does not follow SQLJ specification.

原因: ストアド・ファンクションは、`VALUES(...)` 構文を使用します。

処置: SQLJ では、指定されたファンクション構文を解釈できます。ただし、SQLJ プログラムの移植性を最大限にするには、指示されている構文を使用します。

Stream column name #pos not permitted in SELECT INTO statement.

原因: `SELECT INTO` 文で、`sqlj.runtime.AsciiStream` などのストリーム型は使用できません。

処置: 1 つのストリーム列に対して、位置指定イテレータを使用し、ストリーム列を最後に配置します。または、名前指定イテレータを使用する場合は、ストリーム列（およびその他の列）が順にアクセスされるようにします。

Syntax [`<connection context>`, `<execution context>`, ...] is illegal. Only two context descriptors are permitted.

処置: 接続コンテキストと実行コンテキストの両方を指定する場合は、`#sql [<connection context>, <execution context>] { ... };` を使用します。

The class prefix is *prefix*, which has the SQLJ reserved shape `<file>_SJ`.

原因: `<file>_SJ<suffix>` 形式のクラス名は使用しないでください。これは、SQLJ 内部で使用するために予約されています。

The column *column type* is not nullable, even though it may be NULL in the select list. This may result in a runtime error.

原因: Java 内の NULL は、データベース内の NULL とは異なります。

The keyword *non-portable* keyword for iterator movement is not portable - use *portable* expression instead.

原因: ここで使用されている構文は、ISO SQLJ 規格の一部ではありません。

The option value *-warn=value* is invalid. Permitted values are: *all, none, nulls, nonulls, precision, noprecision, strict, nostrict, verbose, noverbose*.

処置: *-warn* オプションに許可されている値のみを指定します。

The result set column "*name*" type was not used by the named cursor.

原因: 問合せによって、*type* 型の列 *name* が選択されました。しかし、この列は名前指定イテレータが要求するものではありません。

処置: 問合せを変更するか、またはこのメッセージを無視します（このメッセージは、*-warn=nostrict* オプションで非表示にできます）。

The tag *tag* in option *option* is invalid. This option does not permit tags.

処置: タグを使用できるのは、*-user*、*-url*、*-password*、*-offline* および *-online* オプションのみです。オプションは、*-option@tag* ではなく *-option* として指定します。

The type of the context expression is *type*. It does not implement a connection context.

原因: 接続コンテキストは、`sqlj.runtime.ConnectionContext` を実装している必要があります。

The type of the statement execution context is *type*. It does not implement an ExecutionContext.

原因: 実行コンテキストは、クラス `sqlj.runtime.ExecutionContext` のインスタンスであることが必要です。

This SQLJ runtime version requires JDK version 1.2 or later.

原因: JDK 1.1.x 環境で `runtime12.zip` を使用しています。

処置: JDK 1.2 環境で実行するか、`runtime.zip` または `runtime11.zip` などの JDK 1.1.x 互換のランタイムを使用します。

This type is not legal as an IN argument.

原因: Java 型は、JDBC ドライバで IN 引数ではなく、OUT 引数としてサポートされています。

This type is not legal as an OUT argument.

原因: Java 型は、JDBC ドライバで OUT 引数ではなく、IN 引数としてサポートされています。

Type *type* for column *column* is not a JDBC type. Column declaration is not portable.

処置: 移植性を最大限にするには、JDBC 仕様に従った型を使用します。

Type *type* for column *column* is not a valid Java type.

原因: *type* の Java クラス宣言が有効ではありません。

Type *type* of column *column* is not publicly accessible.

原因: SELECT リスト列 *column* の Java クラス *type* は、`public` 属性を持つクラスではありません。このため、ドライバによってインスタンス化できません。

処置: SELECT リストでは、`public` Java 型を使用します。

Type *type* of host item #*n* is not permitted in JDBC. This will not be portable.

処置: 移植性を最大限にするには、JDBC 仕様に従った型を使用します。

Type *type* of host item *item* (at position #*n*) is not permitted in JDBC. This will not be portable.

処置: 移植性を最大限にするには、JDBC 仕様に従った型を使用します。

Type *type* of INTO-list item *n* is not publicly accessible.

原因: INTO リスト項目 *n* の Java クラス *type* は、`public` 属性を持つクラスではありません。このため、ドライバによってインスタンス化できません。

処置: INTO リスト内では、`public` な Java 型を使用します。

Type cast operator requires non-void operand.

原因: `void` 型は、実際の型にはキャストできません。

処置: オペランドの型を修正するか、またはキャスト操作を削除します。

Type map *map* value at key is not a String.

原因: 接続コンテキストに型マップ・リソース *map* を指定しています。キー *key* のエントリが `java.lang.String` のインスタンスではありません。

処置: すべてのキーが非 NULL 文字列の値に対応していることを確認します。

Type map *map* value at key is null.

原因: 接続コンテキストに型マップ・リソース *map* を指定しています。キー *key* のエントリが NULL です。

処置: すべてのキーが非 NULL 文字列の値に対応していることを確認します。

Type map *map*: inner Java class *java type* must be specified as *required type* at entry "*entry*"

原因: 型マップ内のインナー・クラスの参照時に、Java ソースに記述する方法 (`<package name>.<outer class>.<inner class>`) と同じ方法でクラス名を記述しています。しかし、実行時に Java VM ではこのクラスを `Class.forName` でロードできません。

処置: 型マップでは、`<package name>.<outer class>$<inner class>` のようにインナー・クラスを参照します。

Type map resource *resource* appears to have the same name as a class. You should rename the resource.

原因: リソース名が既存のクラス名と重複しています。これによって、プログラムの実行時に問題が生じます。

Type mismatch in argument #*n* of INTO-list. Expected: *type1* Found: *type2*

原因: INTO リスト内のホスト式 #*n* の Java 型 *type2* が、位置指定イテレータで指示される Java 型 *type1* と一致しません。

Unable to check SQL query. Error returned by database is: error

原因: 基本スキーマに対して SQL 問合せをチェック中に、データベースからエラー・メッセージが出力されました。

処置: SQL 問合せが正しいことを確認します。

Unable to check SQL statement. Could not parse the SQL statement.

原因: SQL 文を解析中にエラーが発生し、選択リストの内容を認識できません。

処置: SQL 問合せの構文を確認します。

Unable to check SQL statement. Error returned by database is: error

原因: 基本スキーマに対して SQL 文をチェック中に、データベースでエラー・メッセージが発生しました。

処置: SQL 文が正しいことを確認します。

Unable to check WHERE clause. Error returned by database is: error

原因: 基本スキーマからの問合せの形式を判定中に、データベースからエラー・メッセージが出力されました。

処置: SQL 問合せの構文を確認します。

Unable to connect to data source "*data source*". Will attempt to use JDBC connection instead.

原因: 接続コンテキストには `dataSource` 属性値 *data source* があります。トランスレータでこのデータ・ソースに接続できなかったため、かわりに JDBC 接続を使用しようとしています。

Unable to convert profile to a class file.

原因: SQLJ でプロファイル・ファイル *profile* をクラス・ファイルに変換できません。

処置: プロファイル・ファイルが存在していること、`-d` オプションで指定されたディレクトリが書き込み可能であること、および Java コンパイラがアクセス可能であることを確認してください。

Unable to create a connection context instance of *context name*: message.

原因: SQLJ カスタマイザでは、接続コンテキスト型 *context name* をインスタンス化できません。

処置: コンテキスト・クラス *context name* が **public** として宣言され、CLASSPATH に指定されていることを確認します。これは、このコンテキストが型マップで宣言されている場合は特に重要です。

unable to create backup file

原因: 現行のプロファイルのバックアップ・ファイルを作成できません。プロファイルが置かれているディレクトリに、新しいファイルを作成できません。元のプロファイルは、変更されないままの状態になります。

処置: プロファイルが置かれたディレクトリのアクセス権を確認し、カスタマイザ・ハーネスを再実行します。「backup」オプションを省略し、バックアップ・ファイルを作成せずに、プロファイルのカスタマイズします。

unable to create output file *file*

処置: ファイル *file* を作成するためのアクセス権が設定されていることを確認します。

unable to create package directory *directory*

原因: -d オプションまたは -dir オプションによって、ディレクトリ階層内に出力ファイルを作成することが指定されています。サブディレクトリを作成できることを確認します。

unable to delete *filename*

原因: プロファイル変換ユーティリティで、プロファイル・ファイル *filename* を削除できません。

処置: ファイル *filename* のアクセス権を確認します。

Unable to determine type of WITH-clause attribute *name*: circular reference.

原因: WITH 句属性 *name* の値が、直接または間接的に自己参照されています。こうした場合は、属性のタイプを判定できません。

処置: WITH 句の値を自己参照しないように変更します。

unable to find input file *filename*

処置: ファイル *filename* が存在することを確認します。

Unable to initialize system classes: error. This can be caused by a version mismatch between the SQLJ runtime and the Java environment.

原因: SQLJ ランタイムには Java 環境と互換性がないように思われます。

処置: JDK 1.1.x 環境では runtime.zip または runtime11.zip を使用し、JDK 1.2 以上（望ましいバージョン）では runtime12.zip または runtime.zip を使用します。

Unable to instantiate the offline checker class.

原因: クラス *class* には、`public` なデフォルト・コンストラクタがありません。

Unable to instantiate the online checker class.

原因: クラス *class* には、`public` なデフォルト・コンストラクタがありません。

Unable to instrument args: message

原因: インストルメント中に発生したエラーが原因で、クラス・ファイル *args* をインストルメントできません。

処置: クラス・ファイルが存在し、破損がなく、書込み可能であることを確認します。

unable to load class class name: error description

原因: この SQL 文で使用されている型 *class name* のパラメータまたはイテレータ列を、カスタマイザがロードできません。カスタマイズするには、カスタマイザが SQL 操作で使用されているすべてのクラスをロードできる必要があります。

処置: *class name* 型が「.class」形式で存在し、CLASSPATH で指定されていることを確認します。問題の詳細は、*error description* を参照してください。

Unable to load the offline checker class.

原因: Java クラス *class* が見つかりません。

Unable to load the online checker class.

原因: Java クラス *class* が見つかりません。

unable to move original filename to new filename

原因: プロファイル変換ユーティリティで、プロファイル・ファイル *original filename* を *new filename* という名前に変更できません。

処置: ファイルと出力ディレクトリのアクセス権を確認します。

Unable to obtain DatabaseMetaData to determine the online checker to use for context context. Attempting to use offline checker instead.

原因: JDBC データベースのメタ・データを使用できないか、またはデータベース名とバージョン番号が見つかりませんでした。

処置: 有効な JDBC ドライバを使用できることを確認します。

Unable to obtain description of stored function or procedure: error.

原因: ストアド・ファンクションまたはストアド・プロシージャの起動を認識できないために、エラーが発生しました。

処置: 有効なストアド・プロシージャまたはストアド・ファンクションを呼び出していることを確認します。SQLJ プログラムのチェックに、有効な JDBC ドライバを使用していることを確認します。

Unable to obtain line mapping information from Java file *args: message*

原因: SQLJ では、エラーの発生が原因で、Java ファイル *args* から行マッピング情報を取得できませんでした。

処置: Java クラス・ファイルが存在し、破損がなく、読取り可能であることを確認します。

unable to open temporary output file *filename*

処置: 一時ファイル *filename* を作成できること、およびディレクトリが書き込み可能であることを確認します。

Unable to perform online type checking on weakly typed host item *untypables*

原因: SQLJ では、各 Java ホスト式に対応する SQL 型が決められています。これらの SQL 型は、文のオンライン・チェックに必要です。「弱い型指定」を使用している場合は、SQL 文をオンライン・チェックできない場合があります。

処置: 弱い型指定をユーザー定義の型指定に置き換えます。

Unable to perform semantic analysis on connection *connectionUrl* by user *user*. Error returned by database is: *error*

原因: オンライン・チェック用の接続を確立できませんでした。

unable to read input file *filename*

処置: ファイル *filename* が存在し、その読取りアクセス権を持っていることを確認します。

Unable to read password from user: *error*.

原因: ユーザー・パスワードの読取り中に、エラーが発生しました。

unable to read property file *property file*

処置: `-props=property file` オプションに、プロパティ・ファイルが指定されています。このファイルが存在し、読取り可能であることを確認します。

Unable to read translation state from *file: message*

処置: SQLJ で一時ファイル *file* の作成と、その読取りが可能であることを確認します。

Unable to remove file *file1* or *file2*

原因: 変換時に生成された一時ファイルを削除できませんでした。

処置: 新しく作成されたファイルのデフォルトのアクセス権を確認します。

unable to remove file *filename*

原因: プロファイルのカスタマイズ時に、削除できない一時ファイル *filename* が生成されました。

処置: 新しく作成されたファイルのデフォルトのアクセス権を確認します。一時ファイルを手動で削除します。

unable to rename file *original filename* to *new filename*

原因: プロファイルのカスタマイズ時に、一時ファイル *original filename* を *new filename* という名前に変更できませんでした。カスタマイザ・ハーネスが、元のプロファイルまたは .jar ファイルをカスタマイズされたバージョンに置き換えられませんでした。

処置: 元のプロファイルまたは jar ファイルが書き込み可能であることを確認します。

unable to rename output file from *original filename* to *new filename*

処置: *new filename* が書き込み可能であることを確認します。

Unable to resolve stored function *function* - *n* declarations match this call.

原因: ストアド・ファンクションの起動が、データベース内の複数のストアド・ファンクションのシグネチャと一致します。

処置: ストアド・ファンクションの引数に SQL 式ではなく Java ホスト式を使用して、シグネチャを解決します。

Unable to resolve stored procedure *procedure* - *n* declarations match this call.

原因: ストアド・プロシージャの起動が、データベース内の複数のストアド・プロシージャのシグネチャと一致します。

処置: ストアド・プロシージャの引数に SQL 式ではなく Java ホスト式を使用して、シグネチャを解決します。

Unable to resolve type or value of WITH attribute *attribute*.

原因: イテレータまたはコンテキストの宣言で、WITH 属性が使用されています。WITH 属性の値がリテラル定数またはシンボリック定数ではないので、属性の Java 型と値を特定できません。

処置: リテラル定数またはシンボリック定数を使用して、WITH 属性の値を指定します。

Unable to write Java compiler command line to file: *message*

処置: SQLJ で一時ファイル *file* の作成と、その読取りが可能であることを確認します。

Unable to write translation state to file: *message*

処置: SQLJ で一時ファイル *file* への書き込みが可能であることを確認します。

Unbalanced curly braces.

原因: 必要な箇所に閉じ中カッコが指定されていません。

処置: 不足する閉じ中カッコを追加します。

Unbalanced parenthesis.

原因: 必要な箇所に閉じカッコが指定されていません。

処置: 不足する閉じカッコを追加します。

Unbalanced square brackets.

原因: 必要な箇所に閉じ大カッコが指定されていません。

処置: 不足する閉じ大カッコを追加します。

unchanged

原因: カスタマイズで、プロファイルが変更されませんでした。

処置: エラーが原因でカスタマイズが正常に実行されなかった場合は、それを修正します。プロファイル・プリンタなど、プロファイルを変更しないカスタマイズもあります。そのような場合は、このメッセージを無視してください。

Undefined variable or class name: *name*

原因: 名前 *name* は、式の中に使用されていますが、参照可能な変数またはクラス名に対応していません。

処置: この変数の有無と、名前付きクラスでこの変数を参照可能かどうかを確認します。

Undefined variable, class, or package name: *name*

原因: 名前 *name* は、式の中に使用されていますが、参照可能な変数またはクラス名に対応していません。

処置: この変数の有無と、名前付きクラスでこの変数を参照可能かどうかを確認します。

Undefined variable: *name*

原因: 名前 *name* は、式の中に使用されていますが、参照可能な変数には対応していません。

処置: この名前で参照変数を参照可能かどうかを確認します。

unexpected error occurred...

処置: SQLJ 変換時に予期しないエラーが発生しました。このエラーが発生した場合は、オラクル社カスタマ・サポート・センターに連絡してください。

Unexpected token '*unexpected token*' in Java statement.

原因: Java 文では、ソース・コードで出現する箇所において、トークン *unexpected token* を指定できません。

処置: 文の構文を確認します。

unknown digest algorithm: *algorithm name*

原因: カスタマイザ・ハーネスの「digests」オプションに、不明な jar メッセージ・ダイジェスト・アルゴリズムが指定されています。

処置: *algorithm name* が有効なメッセージ・ダイジェスト・アルゴリズムであることと、対応する MessageDigest 実装クラスが CLASSPATH に指定されていることを確認します。

Unknown identifier '*unknown identifier*'.

原因: 識別子 *unknown identifier* が定義されていません。

処置: 識別子の入力にタイプ・ミスがないこと、または識別子が定義されていることを確認します。

Unknown identifier.

原因: 識別子が定義されていません。

処置: 識別子の入力にタイプ・ミスがないこと、または識別子が定義されていることを確認します。

unknown option found in location: *name*

処置: 有効な SQLJ オプションを使用していることを確認します。sqlj -help-long を実行すると、サポートされているオプションがリストされます。

unknown option type: *option name*

原因: カスタマイザ・ハーネスで、オプション *option name* を処理できません。このオプションに適切な JavaBeans プロパティ・エディタが見つかりませんでした。このオプションは、カスタマイザ固有の非標準オプションの可能性がありま

処置: 現行のカスタマイザに対応付けられているプロパティ・エディタに、CLASSPATH でアクセスできることを確認します。エラーを回避するには、オプションの使用を中止するか、別のカスタマイザを使用します。

Unknown target type in cast expression.

原因: キャスト操作のターゲット型が定義されていません。

処置: 型の名前を確認します。また、型の名前が定義されていることを確認します。

unrecognized option: *option*

原因: プロファイル変換ユーティリティに、不明なオプションが指定されています。

処置: オプションのスペルが正しいことを確認します。

Unrecognized SET TRANSACTION syntax at "*token*" ...

原因: この SET TRANSACTION 文は、SQLJ で認識できません。

処置: この SET TRANSACTION 句を SQLJ で実行する場合は、指示された構文を使用します。

Unrecognized SET TRANSACTION syntax.

原因: この SET TRANSACTION 文は、SQLJ で認識できません。

処置: この SET TRANSACTION 句を SQLJ で実行する場合は、指示された構文を使用します。

Unrecognized SQL statement: *keyword*

原因: SQL 文にキーワード *keyword* が指定されています。SQLJ も JDBC ドライバも、これを SQL キーワードとして認識できません。

処置: SQL 文を確認します。このキーワードが JDBC ドライバも SQL チェッカも認識しないベンダー固有のキーワードの場合は、このメッセージを無視してください。

Unsupported file encoding

処置: -encoding オプションで指定されている文字エンコーディングが、使用している JVM でサポートされていることを確認します。

Unsupported Java type for host item #*n*: *type*.

原因: 使用している JDBC ドライバでは、Java 型 *type* がホストの項目としてサポートされていません。

処置: ホスト式で別の Java 型を使用します。場合によっては、JDBC ドライバを更新します。

Unsupported Java type for host item *name* (at position #*n*): *type*.

原因: 使用している JDBC ドライバでは、Java 型 *type* がホストの項目としてサポートされていません。

処置: ホスト式で別の Java 型を使用します。場合によっては、JDBC ドライバを更新します。

Unsupported Java type for item #*pos* of INTO-list: *type*.

原因: 使用している JDBC ドライバでは、INTO リスト項目 *pos* の Java クラス *type* はサポートされていません。

処置: INTO リスト内では、サポートされている Java 型を使用します。場合によっては、JDBC ドライバを更新します。

Unterminated comment.

原因: クラス宣言が完了する前に、ソース・ファイルはコメントで終了しています。

処置: ソース・ファイル中にコメント・デリミタが不足していないことを確認します。

valid Oracle customization exists

原因: 有効な Oracle カスタマイザを使用して、プロファイルがすでにカスタマイズされています。プロファイルは、変更されませんでした。

処置: プロファイルは、Oracle 用にカスタマイズされています。特に必要な処置はありません。

Value of iterator attribute *attribute* must be a boolean.

処置: このイテレータ with 句属性には、ブール値を指定する必要があります。
attribute=true または *attribute*=false を指定します。

Value of iterator attribute *updateColumns* must be a String containing a list of column names.

処置: イテレータ with 句に、*updateColumns* 属性を指定します。次のように指定します。*updateColumns*="col1,col2,col3"。この列名は更新可能な列を表します。

Value of the iterator with-clause attribute *sensitivity* must be one of SENSITIVE, ASENSITIVE, or INSENSITIVE.

処置: *sensitivity* を設定するには、イテレータ宣言の with 句に、次のいずれかを指定します。*sensitivity*=SENSITIVE、*sensitivity*=ASENSITIVE または *sensitivity*=INSENSITIVE。

Value returned by SQL query is not assigned to a variable.

原因: 問合せの戻り値の処理が定義されていません。

処置: SQL 文を調べ、SELECT の結果を破棄してもよいかどうかを確認します。

Value returned by SQL stored function is not assigned to a variable.

原因: ストアド・ファンクションの戻り値の処理が定義されていません。

処置: SQL 文を調べ、ストアド・ファンクションの戻り値を破棄してもよいかどうかを確認します。

WITH attribute *attribute* must be of type *Java type expected*, not *Java type seen*.

原因: イテレータまたはコンテキストの宣言で、WITH 属性が使用されています。この属性の Java 型には、*Java type expected* を指定する必要があります。しかし属性の実際の型は、*Java type seen* です。

処置: この属性には *Java type expected* を使用します。

You are using a non-Oracle JDBC driver to connect to an Oracle database. Only JDBC-generic checking will be performed.

原因: Oracle 固有のチェックを行うには、Oracle JDBC ドライバが必要です。

You are using an Oracle 8.0 JDBC driver, but connecting to an Oracle7 database. SQLJ will use Oracle7 specific SQL checking.

原因: オンライン接続による変換は、接続先のデータベースの機能に自動的に限定されます。

処置: Oracle 8.0 JDBC ドライバを使用して、Oracle7 データベースに接続する場合は、オフライン・チェックの場合は `oracle.sqlj.checker.Oracle7OfflineChecker`、オンライン・チェックの場合は `oracle.sqlj.checker.Oracle7JdbcChecker` と、それぞれ明示的に指定します。

You are using an Oracle 8.1 JDBC driver, but are not connecting to an Oracle8 or Oracle7 database. SQLJ will perform JDBC-generic SQL checking.

原因: この SQLJ のバージョンでは、接続先のデータベースが認識されません。

処置: Oracle7 データベースまたは Oracle8 データベースに接続します。

You are using an Oracle 8.1 JDBC driver, but connecting to an Oracle7 database. SQLJ will use Oracle7 specific SQL checking.

原因: オンライン接続による変換は、接続先のデータベースの機能に自動的に限定されます。

処置: Oracle 8.1 JDBC ドライバを使用して、Oracle7 データベースに接続する場合は、オフライン・チェックの場合は `oracle.sqlj.checker.Oracle8To7OfflineChecker`、オンライン・チェックの場合は `oracle.sqlj.checker.Oracle8To7JdbcChecker` と、それぞれ明示的に指定します。

You are using an Oracle JDBC driver, but connecting to a non-Oracle database. SQLJ will perform JDBC-generic SQL checking.

原因: この SQLJ のバージョンでは、接続先のデータベースが認識されません。

処置: Oracle7 データベースまたは Oracle8 データベースに接続します。

You cannot specify both, source files (.sqlj,.java) and profile files (.ser,.jar)

原因: SQLJ では、`.sqlj` ソース・ファイルと `.java` ソース・ファイルを変換、コンパイルおよびカスタマイズできます。また、`.ser` ファイルと、`.ser` ファイルを含む `.jar` アーカイブを指定してプロファイル・ファイルをカスタマイズできます。ただし、この両方は行えません。

SQLJ ランタイム・メッセージ

ここでは、SQLJ ランタイムで出力されるエラー・メッセージ、SQL の状態、原因および処置の一覧を示します。

SQL の状態の詳細は、4-22 ページの「[SQL の状態およびエラー・コードの取出し](#)」を参照してください。

java.io.InvalidObjectException: invalid descriptor: *descriptor value*

原因：プロファイル・オブジェクトのロード中に、ある SQL 操作の記述子オブジェクトが無効だと判定されました。プロファイルが標準仕様ではないか、または破損ファイルから読み込まれた可能性があります。

処置：元のソース・ファイルを再変換して、プロファイルを再度作成します。

java.io.InvalidObjectException: invalid execute type: *type value*

原因：プロファイル・オブジェクトのロード中に、いずれかの SQL 操作を実行するためのメソッドが無効だと判定されました。プロファイルが標準仕様ではないか、または破損ファイルから読み込まれた可能性があります。

処置：元のソース・ファイルを再変換して、プロファイルを再度作成します。

java.io.InvalidObjectException: invalid modality: *mode value*

原因：プロファイル・オブジェクトのロード中に、いずれかの SQL 操作パラメータのモダリティが無効だと判定されました。プロファイルが標準仕様ではないか、または破損ファイルから読み込まれた可能性があります。

処置：元のソース・ファイルを再変換して、プロファイルを再度作成します。

java.io.InvalidObjectException: invalid result set type: *type value*

原因：プロファイル・オブジェクトのロード中に、いずれかの SQL 操作で生成された結果の型が無効だと判定されました。プロファイルが標準仕様ではないか、または破損ファイルから読み込まれた可能性があります。

処置：元のソース・ファイルを再変換して、プロファイルを再度作成します。

java.io.InvalidObjectException: invalid role: *role value*

原因：プロファイル・オブジェクトのロード中に、いずれかの SQL 操作の内容が無効だと判定されました。プロファイルが標準仕様ではないか、または破損ファイルから読み込まれた可能性があります。

処置：元のソース・ファイルを再変換して、プロファイルを再度作成します。

java.io.InvalidObjectException: invalid statement type: *type value*

原因：プロファイル・オブジェクトのロード中に、いずれかの SQL 操作の文の型が無効だと判定されました。プロファイルが標準仕様ではないか、または破損ファイルから読み込まれた可能性があります。

処置：元のソース・ファイルを再変換して、プロファイルを再度作成します。

java.lang.ClassNotFoundException: not a profile: *profile name*

原因: *profile name* という名前でプロファイルとして作成されたオブジェクトが、プロファイルとして使用できません。プロファイルが記述されたファイルに不明なデータが存在するか、またはファイルが破損している可能性があります。

処置: 元のソース・ファイルを再変換して、プロファイルを再度作成します。

java.lang.ClassNotFoundException: unable to instantiate profile *profile name*

原因: *profile name* という名前のプロファイルは存在しますが、インスタンス化できません。プロファイルに無効なデータが含まれているか、または破損ファイルから読み込まれた可能性があります。

処置: 元のソース・ファイルを再変換して、プロファイルを再度作成します。

java.lang.ClassNotFoundException: unable to instantiate serialized profile *profile name*

原因: *profile name* という名前で型 `sqlj.runtime.SerializedProfile` のプロファイルは存在しますが、インスタンス化できません。通常、この型のプロファイルは、プロファイルが `.class` 形式に変換されていることを表します。プロファイルに無効なデータが含まれているか、または破損ファイルから読み込まれた可能性があります。

処置: 元のソース・ファイルを再変換して、プロファイルを再度作成します。プロファイルを `.class` 形式で作成する場合は、`ser2class` オプションを使用します。

java.sql.SQLException: closed connection

SQL の状態: 08000

原因: 基になる JDBC 接続オブジェクトがすでに終了している接続コンテキスト・オブジェクトを終了しようとしています。

処置: 基になる JDBC 接続が不注意で終了されていないことを確認します。また、いくつかの SQLJ 接続コンテキストで基になる同じ JDBC 接続を共有する場合、接続コンテキスト・オブジェクトを終了するときに、最後の `close()` 以外のすべてがメソッド `close(ConnectionContext.KEEP_CONNECTION)` を発行していることを確認する必要があります。これによって、基になる JDBC 接続が継続中にオープンになっていることと、JDBC 文オブジェクトなどの関連する JDBC リソースを解放できることが確認されます。

java.sql.SQLException: could not establish connection to DataSource *data source name*: *message*

SQL の状態: 08003

原因: データ・ソース *data source name* に接続を確立できません。

処置: メッセージ・テキストを確認して、データ・ソース *data source name* を接続するのに必要な処理を決定します。

java.sql.SQLException: could not load context type map *type map resource: message*

SQL の状態: 08000

原因: 型マップが見つからないか、リソース *type map resource* からロードされていないか、型マップに無効なエントリが含まれているか、型マップで参照される Java クラスが見つかりません。

処置: メッセージを確認して、失敗の原因と修復する方法を特定します。

java.sql.SQLException: expected *x* columns in select list but found *y*

SQL の状態: 42122

原因: 問合せの結果、*x* 個の項目が選択されましたが、INTO リスト項目数は *y* 個です。または、*y* 個の列で構成されるイテレータに代入されています。

処置: INTO リスト項目またはイテレータ列の個数と、選択された項目の個数が一致するようにプログラムを修正します。

java.sql.SQLException: expected instance of ForUpdate iterator at parameter *x*, found class *class name*

SQL の状態: 46130

原因: 位置指定の SQL 操作に、CURRENT OF 句のターゲットとしてランタイム型 *class name* のホスト式が指定されています。*class name* は、`sqlj.runtime.ForUpdate` インタフェースのインスタンスであることが必要です。

処置: CURRENT OF 句のターゲットとして渡されるイテレータ型の宣言を更新します。`implements` 句に `ForUpdate` インタフェースを指定します。

java.sql.SQLException: expected statement with no OUT parameters: {*statement*}

SQL の状態: 46130

原因: SQL 操作に予期しない 1 つ以上の OUT または INOUT パラメータが指定されています。操作が SQLJ ランタイム標準に従っていないために、特別なカスタマイズを行う必要があります。または、プロファイルが破損ファイルから読み込まれた可能性があります。

処置: 元の SQL 操作が有効であることを確認します。ソース・ファイルを再変換するか、または拡張機能をサポートするカスタマイザを使用します。

java.sql.SQLException: expected statement with OUT parameters: {*statement*}

SQL の状態: 46130

原因: SQL 操作に 1 つ以上指定する必要のある OUT または INOUT パラメータが指定されていません。操作が SQLJ ランタイム標準に従っていないために、特別なカスタマイズを行う必要があります。または、プロファイルが破損ファイルから読み込まれた可能性があります。

処置: 元の SQL 操作が有効であることを確認します。ソース・ファイルを再変換するか、または拡張機能をサポートするカスタマイザを使用します。

java.sql.SQLException: expected statement {statement} to be executed via executeQuery**SQL の状態:** 46130

原因: SQL 操作に対して、結果セットではなく更新カウントを生成するように要求されました。操作が SQLJ ランタイム標準に従っていないために、特別なカスタマイズを行う必要があります。または、プロファイルが破損ファイルから読み込まれた可能性があります。

処置: 元の SQL 操作が有効であることを確認します。ソース・ファイルを再変換するか、または拡張機能をサポートするカスタマイザを使用します。

java.sql.SQLException: expected statement {statement} to be executed via executeUpdate**SQL の状態:** 46130

原因: SQL 操作に対して、更新カウントではなく結果セットを生成するように要求されました。操作が SQLJ ランタイム標準に従っていないために、特別なカスタマイズを行う必要があります。または、プロファイルが破損ファイルから読み込まれた可能性があります。

処置: 元の SQL 操作が有効であることを確認します。ソース・ファイルを再変換するか、または拡張機能をサポートするカスタマイザを使用します。

java.sql.SQLException: expected statement {statement} to use x parameters, found y**SQL の状態:** 46130

原因: y 個のホスト式が必要な SQL 操作で、 x 個のホスト式が指定されています。操作が SQLJ ランタイム標準に従っていないために、特別なカスタマイズを行う必要があります。または、プロファイルが破損ファイルから読み込まれた可能性があります。

処置: 元の SQL 操作が有効であることを確認します。ソース・ファイルを再変換するか、または拡張機能をサポートするカスタマイザを使用します。

java.sql.SQLException: found null connection context**SQL の状態:** 08003

原因: 実行可能な SQL 文で使用されている接続コンテキストのインスタンスが NULL です。

処置: 接続コンテキストのインスタンスを NULL 以外の値に初期化します。SQL 文が暗黙的接続コンテキストを使用している場合は、`sqlj.runtime.ref.DefaultContext` クラスの `static setDefaultContext` メソッドを使用して初期化します。

java.sql.SQLException: found null execution context**SQL の状態:** 08000

原因: 実行可能な SQL 文で使用されている実行コンテキストのインスタンスが NULL です。

処置: 実行コンテキストのインスタンスを NULL 以外の値に初期化します。

java.sql.SQLException: invalid batch limit: バッチ制限**SQL の状態:** 08000**原因:** *batch limit* の値が有効なバッチ制限ではありません。**処置:** バッチ制限が、負数でないかまたは定数 `sqlj.runtime.ExecutionContext.AUTO_BATCH` であることを確認します。**java.sql.SQLException: Invalid column name****SQL の状態:** 46121**原因:** この SQL 操作で使用されている名前指定イテレータで宣言されている列名と、基になる結果セットに含まれている列名が一致しません。名前指定イテレータの各列は、大文字 / 小文字の区別なく、基になる結果セット内の列名と一意に一致する必要があります。**処置:** 名前指定イテレータ内の列名と、対応する問合せ内の列名のいずれかを変更して、この 2 つの名前が一致するようにします。**java.sql.SQLException: invalid iterator type: *type name*****SQL の状態:** 46120**原因:** この SQL 操作の戻り値 (処理対象) は *type name* 型のオブジェクトですが、これは有効なイテレータ型ではありません。標準ではないトランスレータによって生成されたイテレータの可能性があります。**処置:** 元の SQL 操作を確認し、使用しているイテレータ型が有効であることを確認します。必要に応じて、ソース・ファイルを再変換します。**java.sql.SQLException: Java serialization not possible into SQL type: *code*****SQL の状態:** 08000**原因:** SQL 型の問題。OracleTypes コードに反映されているように、*code* が Java オブジェクトのシリアル化またはシリアル化解除に適していません。**処置:** これをサポートしている RAW や BLOB などの SQL 型に、Java オブジェクトをシリアル化またはシリアル化解除します。**java.sql.SQLException: key is not defined in connect properties: *key name*****SQL の状態:** 08000**原因:** キー *key name* は、接続プロパティ・リソース・ファイルで定義されていません。接続プロパティ・リソース・ファイル内の情報を使用して、データベース接続が確立されます。このファイルには、*key name* という名前のキーが指定されている必要があります。**処置:** 必要な接続に対する値と、キー *key name* を接続プロパティ・ファイルに追加します。

java.sql.SQLException: multiple rows found for select into statement**SQL の状態:** 21000**原因:** SELECT INTO 文を実行した結果、複数の行が出力されました。**処置:** 1 行のみ選択されるように、SELECT INTO 問合せまたは問合せ結果を修正します。**java.sql.SQLException: no rows found for select into statement****SQL の状態:** 02000**原因:** SELECT INTO 文を実行した結果、行が何も出力されませんでした。**処置:** 1 行のみ選択されるように、SELECT INTO 問合せまたは問合せ結果を修正します。**java.sql.SQLException: null connection****SQL の状態:** 08000**原因:** 接続コンテキスト・クラスに、NULL の SQLJ 接続コンテキストまたは JDBC 接続オブジェクトが渡されました。**処置:** JDBC 接続を使用している場合は、JDBC 接続オブジェクトでデータベース接続を確立してから、それを接続コンテキストのコンストラクタに渡します。Oracle JDBC ドライバの場合、この処理には、`java.sql.DriverManager` クラスの static な `getConnection` メソッドの 1 つを使用します。接続コンテキスト・オブジェクトを使用している場合は、そのオブジェクトを正しく初期化してからコンストラクタに渡します。デフォルトの接続コンテキストを使用している場合は、デフォルト・コンテキストを使用する前に `setDefaultContext` を呼び出します。**java.sql.SQLException: only fetch forward direction permitted****SQL の状態:** 46110**原因:** この Oracle JDBC ドライバのみで、FETCH FORWARD 結果セットがサポートされています。**処置:** スクロール可能な結果セットを完全にサポートしている 8.1.6 以上の JDBC ドライバにアップデートします。**java.sql.SQLException: profile *profile name* not found: *error description*****SQL の状態:** 46130**原因:** プロファイル *profile name* が見つからないか、インスタンス化できません。*error description* に詳細が示されています。**処置:** *error description* を参照して対処します。**java.sql.SQLException: SQL operation currently in use****SQL の状態:** 46000

java.sql.SQLException: streaming of BLOB data not supported in this driver**SQL の状態:** 46110

原因: JDBC ドライバでは、BLOBS へのロー・データのストリーミングをサポートしていません。

処置: 新しい Oracle JDBC ドライバ・バージョンにアップデートするか、RAW 列を使用してシリアル化された Java オブジェクトを保持します。

java.sql.SQLException: unable to convert database class *found type* to client class *expected type***SQL の状態:** 22005

原因: ホスト式でクラス *expected type* が必要とされる場合に、データベース型から Java オブジェクトへのデフォルト・マッピングによって、クラス *found type* が生成されました。クライアント側クラス `java.math.BigDecimal` への変換に失敗した可能性があります。または、特別なカスタマイザを使用している場合のみサポートされる標準ではないクラスへの変換に失敗した可能性があります。

処置: 選択したデータベース型が、ホスト変数またはフェッチ先のイテレータ列の型に代入できるデフォルト・マッピングを持っていることを確認します。別のクライアント側型を使用することが必要な場合もあります。クライアント側型をサポートするために必要なカスタマイザが使用されていることを確認します。

java.sql.SQLException: Unable to create CallableStatement for RTStatement**SQL の状態:** 46110

原因: この SQL 操作を実行するには、実行時に JDBC CallableStatement オブジェクトを使用する必要があります。ただし、このオブジェクトは、操作の実行に使用されるカスタマイザからは使用できません。互換性のないカスタマイザがアプリケーションで使用されている可能性があります。または、操作の実行に、特別なカスタマイザが必要な場合もあります。

処置: ソース・ファイルを再変換するか、または拡張機能をサポートするカスタマイザを使用します。

java.sql.SQLException: Unable to create PreparedStatement for RTStatement**SQL の状態:** 46110

原因: この SQL 操作を実行するには、実行時に JDBC PreparedStatement オブジェクトを使用する必要があります。ただし、このオブジェクトは、操作の実行に使用されるカスタマイザからは使用できません。互換性のないカスタマイザがアプリケーションで使用されている可能性があります。または、操作の実行に、特別なカスタマイザが必要な場合もあります。

処置: ソース・ファイルを再変換するか、または拡張機能をサポートするカスタマイザを使用します。

java.sql.SQLException: unable to load connect properties file: *filename***SQL の状態:** 08000

原因: 接続プロパティ・ファイル *filename* をリソース・ファイルとしてロードできません。このファイルは、データベース接続の確立に使用されます。このファイルはアプリケーション・リソース・ファイルとしてロードされるので、アプリケーション・クラスでパッケージ化する必要があります。ファイルが期待された場所にないか、または読み取り不可能の可能性があります。

処置: 接続プロパティ・ファイルが読み取り可能で、アプリケーション・クラスでパッケージ化されていることを確認します。

java.sql.SQLException: unexpected call to method *method name***SQL の状態:** 46130

原因: SQL 操作の実行に、メソッド *method name* への予期しないコールが含まれています。操作が SQLJ ランタイム標準に従っていないために、特別なカスタマイズを行う必要があります。標準ではない SQLJ トランスレータが使用されている可能性もあります。

処置: 元の SQL 操作が有効であることを確認します。ソース・ファイルを再変換するか、または拡張機能をサポートするカスタマイザを使用します。

java.sql.SQLException: unexpected exception raised by constructor *constructor name: exception description***SQL の状態:** 46120

原因: ランタイム結果または出力パラメータの生成時に、コンストラクタがランタイム例外を発生させました。

処置: *exception description* の内容を確認し、例外の原因を特定します。

java.sql.SQLException: unexpected exception raised by method *method name: exception description***SQL の状態:** 46120

原因: ホスト式とデータベース型間の変換がメソッド *method name* の呼出しと関係しており、このメソッドが `SQLException` 以外の例外を発生させました。

処置: *exception description* の内容を確認し、例外の原因を特定します。

sqlj.runtime.SQLNullException: cannot fetch null into primitive data type**SQL の状態:** 22002

原因: 基本型の Java イテレータ列型、結果、OUT パラメータまたは INOUT パラメータに、SQL NULL の格納を試みました。

処置: 基本型ではなく、NULL 値を許容する Java ラッパー型を使用します。

A

ASENSITIVE (カーソル状態), 3-7
AuditorInstaller
 オプション, A-45
 起動, A-43
 コマンドラインの例, A-49
 デバッグ用のカスタマイザ, A-42
 ランタイム出力, A-44

B

backup オプション (カスタマイザ・ハーネス), A-13
BetterDate (カスタム Java クラス), 6-17
BFILE
 BFILE のサポート, 5-29
 ストアド・ファンクションの結果, 5-33
BigDecimal
 サポート, 5-41
 マッピング (属性が対象), 6-35
bind-by-identifier オプション
 (sqlj-bind-by-identifier), 8-66
BLOB のサポート, 5-29
BOOLEAN 型 (PL/SQL), 5-9
builtintypes オプション (JPublisher -builtintypes),
 6-34

C

cache オプション (sqlj-cache), 8-75
CALL 構文、ストアド・プロシージャ, 3-54
case オプション (JPublisher -case), 6-33
CHAR 比較、空白埋め, 8-56、A-30
checker オプション (SQLCheckerCustomizer), A-40
checkfilename オプション (sqlj-checkfilename), 8-81

checksource オプション (sqlj-checksource), 8-65
CLASSPATH オプション (sqlj-classpath), 8-22
CLASSPATH と PATH, 2-11
CLOB のサポート, 5-29
close() メソッド (DefaultContext), 4-16
close() メソッド (ExecutionContext), 7-29
close() メソッド (Oracle クラス), 4-13、4-16
CLOSE_CONNECTION, 7-52
codegen オプション (SQLJ-codegen), 8-50
compatible オプション (JPublisher -compatible), 6-31
compat (互換性) オプション (Oracle カスタマイザ),
 A-24
compiler encoding サポート・オプション (sqlj), 8-79
compiler executable オプション (sqlj), 8-78
compiler output file オプション (sqlj-compiler...),
 8-80
compile オプション (sqlj-compile), 8-62
connect() メソッド (Oracle クラス), 4-12
context オプション (カスタマイザ・ハーネス), A-13
CURSOR 構文 (ネストした表), 6-57
CustomDatum (現在使用されていない), 6-8
customizer オプション (カスタマイザ・ハーネス),
 A-14
C 接頭辞 (sqlj-C-x), 8-59

D

DBMS_JAVA パッケージ
 サーバー出力デバイスの設定, 11-6
 サーバーでの SQLJ オプションの設定, 11-18
DBMS_LOB パッケージ, 5-29
debug オプション (カスタマイザ・ハーネス), A-20
debug オプション、コンパイル (サーバー側), 11-18
default customizer オプション (sqlj), 8-84

DefaultContext クラス

- close() メソッドのパラメータ, 4-16
- コンストラクタ, 4-15
- 主要メソッド, 4-14
- 単一または複数の接続での使用, 4-6

depth オプション (AuditorInstaller), A-46

dirty read, 7-46

dir オプション (sqlj-dir), 8-31

driver オプション (カスタマイザ・ハーネス), A-19

dropjava, 11-22

d オプション (sqlj-d), 8-29

E

echo オプション、実行しない, 8-26

encoding オプション (サーバー側), 11-17

encoding オプション、ソース・ファイル
(sqlj-encoding), 8-28

explain オプション (sqlj-explain), 8-47

F

FETCH CURRENT 構文 (イテレータ), 7-42

FETCH 構文 (スクロール可能な位置指定イテレータ),
7-41

fixedchar オプション (Oracle カスタマイザ), A-30

fixedchar オプション (SQLJ-fixedchar), 8-56

force オプション (Oracle カスタマイザ), A-25

ForUpdate/updateColumns (WITH 句), 3-7

G

getConnection() メソッド (Oracle クラス), 4-12

getProperty() を使用した Java プロパティの設定, 9-28

H

help オプション (sqlj-help-xxxx), 8-23

help オプション (カスタマイザ・ハーネス), A-15

holdability (カーソル状態、WITH 句), 3-7

I

IDE への SQLJ の統合, 1-29

IMPLEMENTS 句

イテレータ宣言, 7-36

構文, 3-5

接続コンテキスト宣言, 7-10

INSENSITIVE (カーソル状態), 3-7

J

JAR ファイルのダイジェスト・オプション (カスタマイザ・ハーネス), A-14

javac 互換, 8-10

Java スキーマ・オブジェクトの削除, 11-22

Java バインド式 (動的 SQL), 7-58

Java プロパティ、getProperty(), 9-28

Java 名と SQL 名 (サーバー側), 11-7

JDBC 2.0

LOB 型のサポート, 5-27

サポートされている型, 5-8

弱い型指定の構造体、参照、配列のサポート, 6-74

JDBC 接続メソッド (トランザクション), 7-47

JDBC と SQLJ、サンプル・アプリケーション, 1-16

JDBC との連係動作

イテレータと結果セット, 7-53

接続コンテキストと接続, 7-48

JDBC ドライバ

Oracle ドライバ, 4-2

カスタマイズの選択および登録, A-19

検証, 2-15

変換の用途に選択, 4-4

ランタイム用の選択 / 登録, 4-5

JDBC ドライバ登録オプション (sqlj-driver), 8-43

JDBC の注意事項 (サーバー側), 11-4

JDBC マッピング (属性が対象), 6-35

jdbcblinemap オプション (sqlj-jdblinemap), 8-49

JDeveloper

SQLJ の統合, 1-29

デバッグ, 10-27

JDK

サポートされているバージョン, 2-4

対応する JDBC クラス・ファイル, 2-11

JNDI

データ・ソースの使用、接続, 7-12

デフォルト・データ・ソースの名前, 7-14

JPublisher

- builtintypes オプション, 6-34
- case オプション, 6-33
- compatible オプション, 6-31
- JPublisher の出力内容, 6-26
- lobtypes オプション, 6-34
- numbertypes オプション, 6-34
- sql オプション, 6-31
- types オプション, 6-32
- user オプション, 6-32
- カスタム Java クラスの作成, 6-26
- カスタム Java クラスの生成, 6-30
- カスタム Java クラスの例, 6-44
- 型の分類とマッピング・オプション, 6-34
- 型のマッピング, 6-34
- 型のマッピング指定, 6-31
- 型のマッピング・モードとオプション設定, 6-35
- 生成クラスの詳細, 6-48
- 代替クラスのマッピング, 6-36、6-38
- 入力ファイル, 6-39
- プロパティ・ファイル, 6-41
- メンバー名の指定, 6-42
- ラッパー・メソッドの実行, 6-43

JVM

- CLASSPATH オプション, 8-22
- SQLJ オプション, 8-58
- 名前の指定, 8-78

J 接頭辞 (sqlj -j-x), 8-58

K

KEEP_CONNECTION, 7-52

L

linemap オプション (sqlj -linemap), 8-48

loadjava

- SQLJ ソースのロードとサーバーでの変換, 11-14
- SQLJ ソースをロードした時の出力, 11-20
- クラス / リソースのロード, 11-8
- 互換オプション、SQLJ, 8-9

LOB

- FETCH INTO LOB ホスト変数, 5-35
- SELECT INTO LOB ホスト変数, 5-34
- イテレータ列として, 5-34
- サポート (oracle.sql と DBMS_LOB), 5-29
- ストアド・ファンクションの結果, 5-33

lobtypes オプション (JPublisher -lobtypes), 6-34
log オプション (AuditorInstaller), A-46

N

native2ascii、文字エンコーディング, 9-29

NcharAsciiStream クラス (グローバリゼーション・サポート), 9-25

NcharUnicodeStream クラス (グローバリゼーション・サポート), 9-25

NCHAR クラス (グローバリゼーション・サポート), 9-25

NCLOB クラス (グローバリゼーション・サポート), 9-25

NLS、「グローバリゼーション・サポート」を参照
non-repeatable read, 7-46

NString クラス (グローバリゼーション・サポート), 9-25

NULL の処理

ラッパー・クラス、NULL の処理用, 4-18
例, 4-19

numbertypes オプション (JPublisher -numbertypes), 6-34

n オプション (sqlj -n) (実行せずにエコー), 8-26

O

Object JDBC マッピング (属性が対象), 6-35

OCI ドライバ (JDBC), 4-3

offline オプション (sqlj -offline), 8-71

online オプション (sqlj -online), 8-73

online オプション (サーバー側), 11-17

opaque 型, 6-75

optcols オプション (Oracle カスタマイザ), A-26

optcols オプション (SQLJ -optcols), 8-51

optparamdefaults オプション

(SQLJ -optparamdefaults), 8-54

optparamdefaults オプション (Oracle カスタマイザ), A-29

optparams オプション (Oracle カスタマイザ), A-28

optparams オプション (SQLJ -optparams), 8-52

Oracle9i Lite での SQLJ, 1-27

OracleChecker デフォルトのチェック, 8-68

oracle.sql パッケージ, 5-28

Oracle オプティマイザ, 10-2

Oracle カスタマイザ

- CHAR 比較での空白埋め, A-30
- SQL 表示オプション, A-31
- オプション, A-23
- 強制カスタマイズ, A-25
- 使用している Oracle 拡張機能の一覧, A-34
- デフォルトのパラメータ・サイズの設定, A-29
- バージョン互換性, A-24
- パラメータ・サイズの定義, A-28
- 文のキャッシュ・サイズ, A-32
- 列型 / サイズの定義, A-26

Oracle カスタマイザのオプション, A-23

Oracle クラス

- close() メソッドのパラメータ, 4-13
- connect() メソッド, 4-12
- DefaultContext のインスタンス, 4-12
- getConnection() メソッド, 4-12

Oracle 固有コード生成

- 概要, 1-3、4-36
- 環境の要件, 4-37
- コーディング上の考慮事項、制限事項, 4-38
- サーバー側の考慮事項, 4-41
- トランスレータおよびカスタマイザ使用方法の変更, 4-39
- メリットとデメリット, 4-36

Oracle の拡張

- 概要, 1-7
- 拡張型, 5-27
- サマリー機能使用, A-34
- パフォーマンスの拡張, 10-2

Oracle マッピング (属性が対象), 6-35

ORADData

- CustomDatum との比較, 6-8
- カスタム Java クラスでの使用, 6-5
- 指定, 6-6
- その他の使用例, 6-16

P

- passes オプション (sqlj -passes), 8-82
- passes、2 段階で行うコンパイル, 8-82
- password オプション (カスタマイザ・ハーネス), A-18
- password オプション、チェック (sqlj), 8-37
- path (WITH 句接続コンテキスト), 3-7
- PATH と CLASSPATH, 2-11
- phantom read, 7-46

PL/SQL

- BOOLEAN 型, 5-9
- RECORD 型, 5-9
- TABLE 型, 5-9
- 実行文のブロック, 3-13
- prefix オプション (AuditorInstaller), A-47
- print オプション (カスタマイザ・ハーネス), A-21
- profile オプション (sqlj -profile), 8-63
- props オプション (sqlj -props), 8-22
- Public クラス名 / ソース名のチェック, 8-81
- P 接頭辞 (sqlj -P-x), 8-60

R

- READ COMMITTED トランザクション, 7-46
- READ ONLY トランザクション, 7-46
- READ UNCOMMITTED トランザクション, 7-46
- READ WRITE トランザクション, 7-46
- RECORD 型 (PL/SQL), 5-9
- REF CURSOR
 - REF CURSOR 型について, 5-39
 - SQLJ のサポート, 5-40
 - 例, 5-40
- REPEATABLE READ トランザクション, 7-46
- ResultSetIterator 型, 7-37
- returnability (カーソル状態、WITH 句), 3-7
- ROWID
 - FETCH INTO ROWID ホスト変数, 5-38
 - SELECT INTO ROWID ホスト変数, 5-38
 - サポート, 5-36
 - ストアド・ファンクションの結果, 5-38

S

- ScrollableResultSetIterator 型, 7-43
- SELECT INTO 文
 - エラー状態, 3-32
 - 構文, 3-30
 - 例, 3-31
- SENSITIVE (カーソル状態), 3-7
- sensitivity (カーソル状態、WITH 句), 3-7
- ser2class オプション (sqlj -ser2class), 8-64
- SERIALIZABLE トランザクション, 7-46
- .ser プロファイルから .class への変換, 8-64
- SET TRANSACTION 構文, 7-45
- SET (代入) 文, 3-53
- showReturns オプション (AuditorInstaller), A-47

showSQL オプション (Oracle カスタマイザ), A-31
showThreads オプション (AuditorInstaller), A-48
SQLCheckerCustomizer
 オプション, A-40
 起動, A-39
 プロファイルのセマンティクス・チェック, A-38
SQLData
 カスタム Java クラスでの使用, 6-5
 指定, 6-8
SQLException サブクラスの使用方法, 4-23
SQL_OPTIONS 環境変数, 8-19
sqljutil パッケージ, 2-12
SQLJ 句, 3-9
SQLJ 接続のための JavaBeans, 7-18
SQLJ での動的 SQL のサポート
 概要, 7-57
 実行時の動作, 7-59
 変換時の動作, 7-59
 メタ・バインド式, 7-57
 例, 7-59
SQLJ と JDBC のサンプル・アプリケーション, 1-16
SQLJ について, 1-2
SQLJ の設定、テスト, 2-13
sql オプション (JPublisher -sql), 6-31
SQL オプティマイザ, 10-2
SQL 置換コード (動的 SQL), 7-58
SQL の状態 (エラー時), 4-22
SQL 名と Java 名 (サーバー側), 11-7
status オプション (sqlj -status), 8-47
stmtcache オプション (Oracle カスタマイザ), A-32
summary オプション (Oracle カスタマイザ), A-34
Sun JDK
 サポートされているバージョン, 2-4
 対応する JDBC クラス・ファイル, 2-11

T

TABLE 型 (PL/SQL), 5-9
TABLE 構文 (ネストした表), 6-57、6-60
Thin ドライバ (JDBC), 4-3
transformGroup (WITH 句接続コンテキスト), 3-7
TRANSLATE (オブジェクト・メンバー名), 6-42
typeMap (WITH 句接続コンテキスト), 3-7
types オプション (JPublisher -types), 6-32

U

uninstall オプション (AuditorInstaller), A-48
updateColumns/ForUpdate (WITH 句), 3-7
url オプション (sqlj -url), 8-39
url オプション (カスタマイザ・ハーネス), A-19
user オプション (JPublisher -user), 6-32
user オプション (sqlj -user), 8-33
user オプション (カスタマイザ・ハーネス), A-17

V

VALUES トークンでのストアド・ファンクションの呼出し, 3-55
VARRAY
 VARRAY 型, 6-4
 行の挿入, 6-65
 ホスト式への取出し, 6-63
verbose オプション (カスタマイザ・ハーネス), A-16
verify オプション (カスタマイザ・ハーネス), A-22
VM
 CLASSPATH オプション, 8-22
 SQLJ オプション, 8-58
 名前の指定, 8-78
vm オプション (sqlj -vm), 8-78

W

warn オプション (SQLCheckerCustomizer), A-41
warn オプション (sqlj -warn), 8-44
WHERE CURRENT OF, 5-36
Windows での SQLJ の開発, 1-30
WITH 句の構文, 3-6

あ

アクセス・モードの設定 (トランザクション), 7-46
アプレット
 SQLJ の使用, 1-22

い

位置指定イテレータ

- next() によるナビゲーション, 3-46
- アクセス, 3-44
- インスタンス化と移入, 3-44
- 使用, 3-43
- スクロール可能, 7-41
- 宣言, 3-43

位置指定更新, 5-36

位置指定削除, 5-36

イテレータ

- 位置指定イテレータの使用, 3-43
- 位置指定イテレータの宣言, 3-43
- 位置指定イテレータへのアクセス, 3-44
- 位置指定イテレータ、next() の使用, 3-46
- イテレータ・クラスの機能, 7-35
- イテレータ列として (ネスト), 3-50
- オブジェクト参照の取出し, 6-52
- 概要, 3-33
- 結果セット・イテレータ (弱い型指定), 3-36、7-37
- 結果セットへの変換, 7-53、7-55
- コミット / ロールバックの影響, 4-27
- サブクラス化, 7-37
- シリアル化されたオブジェクト, 6-71
- スクロール可能, 7-38
- スクロール可能な結果セット・イテレータ, 7-43
- ストアド・ファンクションの戻り値, 3-57
- 宣言, 3-3
- 宣言での IMPLEMENTS 句, 7-36
- 通常の使用手順, 3-37
- 名前指定イテレータのインスタンス化と移入, 3-40、3-44
- 名前指定イテレータの使用, 3-38
- 名前指定イテレータの設定 (例), 4-33
- 名前指定イテレータの宣言, 3-39
- 名前指定イテレータへのアクセス, 3-41
- 名前指定と位置指定, 3-37
- ネスト・イテレータとして表されたネストした表, 6-61
- ホスト変数として, 3-47
- 弱い型指定のイテレータの使用, 7-56
- イテレータ・クラスのサブクラス化, 7-37
- インストールおよび設定の確認, 2-10
- インストールメント、クラスファイル (linemap), 8-48

え

エラー

- カスタマイズ時のメッセージ, A-8
 - 原因と処置の出力, 8-47
 - サーバー側のエラー出力, 11-22
 - トランスレータ・エラー・リスト, B-2
 - トランスレータのエラー、警告および情報メッセージ, 9-12
 - メッセージとコードおよび SQL の状態, 4-22
 - メッセージの文字エンコーディング, 9-23
 - ランタイム・エラーの分類, 9-18
 - ランタイム・エラー・リスト, B-44
- ### エラーの原因 / 処置の出力, 8-47
- ### エンコーディング
- native2ascii の使用, 9-29
 - コマンドラインの例, 9-24
 - コンパイラへのオプションの無効化, 8-79
 - 実行時の設定, 9-28
 - 指定、サーバー, 11-17
 - ソース・コードの文字エンコーディングの設定, 9-22
 - メッセージの文字エンコーディング, 9-23
 - 文字エンコーディング概要, 9-20

お

- オーディタを持つプロファイルでのデバッグ, A-42
 - 大 / 小文字を区別する SQL の UDT 名, 6-12、6-13、6-32、6-40
- ### オブジェクト

- ORADData の指定, 6-6
- RAW および BLOB 列のシリアル化, 6-66
- SQLData の指定, 6-8
- SQLJ 上での厳密な分類, 6-51
- イテレータへの取出し, 6-52
- オブジェクト型の作成, 6-21
- 概要, 6-2
- カスタム Java クラスについて, 6-5
- カスタム Java クラスによるシリアル化, 6-68
- 型のマッピング指定, 6-31、6-34
- 基本概念, 6-4
- 更新、SQLJ, 6-53
- 参照の更新, 6-56
- シリアル化 (概要), 6-65
- 挿入、SQLJ, 6-55
- 代替クラスのマッピング, 6-38

- データ型, 6-5
- メソッドのサポート, 6-9
- 弱い型指定のサポート, 6-74
- 弱い型指定の制限, 6-75
- ラッパー・メソッド, 6-36
- オブジェクト参照
 - SQLJ 上での厳密な分類, 6-51
 - イテレータへの取出し, 6-52
 - 更新、SQLJ, 6-56
 - 弱い型指定のサポート, 6-74
 - 弱い型指定の制限, 6-75
- オブジェクトのメソッドのラッパー (JPub), 6-43
- オプション (トランスレータ)
 - help, 8-23
 - javac 互換, 8-10
 - loadjava 互換, 8-9
 - VM およびコンパイラ, 8-77
 - オプションを渡す接頭辞, 8-57
 - 概要, 8-3
 - カスタマイズ, 8-84
 - コマンドライン専用, 8-21
 - サマリー・リスト, 8-3
 - 出力ファイルとディレクトリ, 8-28
 - 接続, 8-32
 - セマンティクス・チェック、オフライン解析, 8-67
 - 代替環境のサポート, 8-76
 - 特殊処理のフラグ, 8-62
 - 優先順位, 8-20
 - レポートと行マッピング, 8-43
- オペティマイザ、SQL, 10-2
- オフライン解析
 - sqlj-parse オプション, 8-75
 - オンライン・チェックとの比較, 8-70
 - 手順, 9-2
- オフライン・チェック
 - チェックの指定, 8-71
 - デフォルトのチェック、Oracle チェック, 8-68
- オンライン・チェック
 - URL の設定, 8-39
 - オフライン解析との比較, 8-70
 - 結果のキャッシング, 8-75
 - チェックの指定, 8-73
 - デフォルトの URL 接頭辞の設定, 8-42
 - デフォルトのチェック、Oracle チェック, 8-68
 - ドライバの登録, 8-43
 - パスワードの設定, 8-37

- 有効化 (サーバー側), 11-17
- ユーザー・スキーマの設定の有効化, 8-33

か

- 下位互換性
 - Oracle SQLJ、概略, 2-9
 - Oracle8i, 5-11
 - Oracle8/Oracle7, 5-11
- 解析オプション (sqlj-parse), 8-75
- 拡張
 - 概要, 1-7
 - 拡張型, 5-27
 - サマリー機能使用, A-34
 - パフォーマンスの拡張, 10-2
- 拡張、JPub 生成クラス, 6-48
- 拡張型, 5-27
- カスタマイザ
 - 概要, A-5
 - カスタマイザの選択, A-14
 - 選択, A-10
 - デフォルトの指定, 8-84
 - 渡されるオプション, 8-60
- カスタマイザ・ハーネス
 - オプションの概要, A-11
 - 概要, A-5
 - 接続オプション, A-17
 - 特別なカスタマイザの起動, A-20
 - 汎用オプション, A-13
- カスタマイザ・ハーネスのオプション
 - 概要, A-11
 - 接続オプション, A-17
 - 特別なカスタマイザの起動, A-20
 - 汎用オプション, A-13
- カスタマイズ
 - jar ファイルの用途, A-36
 - Oracle カスタマイザのオプション, A-23
 - .ser プロファイルから .class への変換, 8-64
 - SQL 変換の表示, A-31
 - エラー・メッセージとステータス・メッセージ,
A-8
 - オプション, A-10
 - カスタマイザ固有のオプションの概要 / 構文, A-23
 - カスタマイザ・ハーネスのオプションの概要, A-11
 - カスタマイザ・ハーネスの接続オプション, A-17
 - カスタマイザ・ハーネスの汎用オプション, A-13
 - カスタマイズと登録, A-7

カスタマイズについて, A-4
関連する SQLJ オプション, A-36
強制カスタマイズ, A-25
使用している Oracle 拡張機能の一覧, A-34
処理の手順, A-5
特別なカスタマイズの起動のためのオプション,
A-20
バージョン互換性, A-24
パラメータ・サイズの定義, A-28
パラメータのデフォルト・サイズ, A-29
文のキャッシュ・サイズ, A-32
変換時, 9-10
有効化 / 無効化, 8-63
列型 / サイズの定義, A-26
カスタマイズ用 jar ファイルのダイジェスト・オプ
ション, A-14
カスタム Java クラス
JPublisher での作成, 6-26
JPublisher による生成, 6-30
オブジェクト・メソッドでのサポート, 6-9
拡張, 6-48
カスタム Java クラスについて, 6-5
コンパイル, 6-15
サンプル・クラス, 6-17
シリアル化オブジェクトの使用方法, 6-68
代替クラスのマッピング, 6-38
強い型指定、定義, 6-2
データの読み込みと書き込み, 6-16
メンバー名の指定, 6-42
要件, 6-10
弱い型指定、定義, 6-2
例, 6-44
型解決, 8-65
型の解決を目的としたソース・チェック, 8-65
型のサポート, 5-1
型のマッピング
BigDecimal マッピング, 6-35
JDBC マッピング, 6-35
JPublisher マッピング・オプション, 6-31
Object JDBC マッピング, 6-35
Oracle マッピング, 6-35
型の分類とマッピング・モード, 6-34
各国語サポート, 「グローバル化・サポート」
を参照
環境による使用例とガイドライン, 2-4
環境変数によるオプションの指定, 8-19

き

基本スキーマ, 4-17
キャッシング、オンライン・チェック結果, 8-75
キャッシング、文, 10-4
行プリフェッチ, 10-3
行マッピング
SQLJ ソースから class (jdb), 8-49
SQLJ ソースと class ファイル, 8-48

く

クライアント側変換とサーバーでの実行, 11-8
クラス・スキーマ・オブジェクトのネーミング
生成済み, 11-21
ロード済み, 11-10
クラスのロード (サーバー側), 11-6
グローバル化・サポート
SQLJ 外部, 9-28
Unicode 文字のサポート, 9-25
概要, 1-29
関連 Java 型, 9-25
関連 SQLJ および Java 設定, 9-22
関連するデータ型, 5-5
文字エンコーディング、言語サポート, 9-19

け

結果式
概要, 3-19
実行時評価, 3-20
結果セット
イテレータへの変換, 7-53、7-55
イテレータ列として, 3-50
コミット / ロールバックの影響, 4-27
ストアド・ファンクションの戻り値, 3-57
複数のコールで使用 (サーバー側), 11-4
ホスト変数として, 3-47
結果セット・イテレータ (弱い型指定)
概要, 3-36、7-37
スクロール可能, 7-43
言語サポート (グローバル化・サポート),
9-21

コ

構文

- トランスレータのコマンドライン, 8-12
- トランスレータのプロパティ・ファイル, 8-17

コードの生成

- Oracle 固有と ISO 規格の比較, 4-36
- translator -codegen オプション, 8-50
- 概要, 9-5

コード・レイヤー、プロファイル, A-42

コール、ストアド・ファンクション, 3-55

コール、ストアド・プロシージャ, 3-54

コール、ランタイムへの生成, 9-8

コマンドライン (トランスレータ)

- 概要, 8-2
- 構文と引数, 8-12
- 非実行時のエコー, 8-16
- 例, 8-15

コマンドラインを実行せずにエコー, 8-16

コミット

- 新しい接続での自動コミットの指定, 4-25
- イテレータおよび結果セットに対する影響, 4-27
- 既存の接続の自動コミットに対する変更, 4-26
- 自動と手動, 4-24
- 手動, 4-26

コレクション

- ORADATA の指定, 6-6
- カスタム Java クラスについて, 6-5
- 型のマッピング指定, 6-31、6-34
- 基本概念, 6-4
- コレクション型の作成, 6-23
- コレクションのサポートの概要, 6-2
- 代替クラスのマッピング, 6-38
- 強い型指定, 6-57
- データ型, 6-5
- 弱い型指定のサポート, 6-74
- 弱い型指定の制限, 6-75

コンテキスト式

- 概要, 3-19
- 実行時評価, 3-20

コンパイラ

- CLASSPATH オプション, 8-22
- javac のオプションのサポート, 8-10
- SQLJ オプション, 8-59
- 関連オプション, 8-77
- 名前の指定, 8-78
- 必要な動作, 8-78

コンパイラの出力ファイル, 8-80

コンパイラ・メッセージの出力パイプ・オプション (sqlj), 8-80

コンパイル

- debug オプション (サーバー側), 11-18
- コンパイル、2 段階, 8-82
- サーバー, 11-6
- 変換時, 9-9
- 有効化 / 無効化, 8-62

さ

サーバー側 SQLJ

- dropjava, 11-22
- Java スキーマ・オブジェクトの削除, 11-22
- Java マルチスレッド, 11-23
- JDBC における違い, 11-4
- SQLJ ソースのロードとサーバーでの変換, 11-14
- SQL 名と Java 名, 11-7
- エラー出力, 11-22
- オプション, 11-16
- オプションの設定, 11-18
- 概要, 1-26、11-2
- クラスのロード, 11-6
- コーディングでの注意事項, 11-3
- 固定の設定, 11-16
- コンパイル, 11-6
- サーバー側でのクライアント・プログラムの実行, 11-13
- サーバーでのコード実行状況の確認, 11-25
- サーバーへのクラス / リソースのロード, 11-8
- 再帰的コール, 11-23
- 生成されたクラス・スキーマ・オブジェクトの名前付け, 11-21
- ソース・スキーマ・オブジェクトの名前付け, 11-20
- データベース接続, 11-4
- デフォルトの出力デバイス, 11-6
- 変換 (クライアント側), 11-8
- 変換 (サーバー側), 11-14
- 変換で生成される出力, 11-20
- ロードされたクラス・スキーマ・オブジェクトの名前付け, 11-10
- ロードされたリソース・スキーマ・オブジェクトの名前付け, 11-11
- サーバー側 Thin ドライバ (JDBC), 4-3
- サーバー側トランスレータからの出力, 11-20

- サーバー側トランスレータのエラー出力, 11-22
- サーバー側内部ドライバ (JDBC), 4-4
- サーバー側変換のオプション
 - オプションの設定, 11-18
 - 固定の設定, 11-16
 - サポートされているオプション, 11-16
- サーバーのデフォルトの出力デバイス, 11-6
- サーバーへのクラス / リソースのロード, 11-8
- 再帰的 SQLJ コール、サーバー, 11-23
- サポートされている型
 - JDBC 2.0, 5-8
 - Oracle8i, 5-11
 - Oracle8/Oracle7, 5-11
 - 型の概要, 5-2
- サンプル・アプリケーション
 - JDBC と SQLJ, 1-16
 - 単一行問合せ (SELECT INTO), 4-32
 - 複数行問合せ (名前指定イテレータ), 4-34
 - 複数の接続コンテキスト, 7-7
 - マルチスレッド, 7-33
- サンプル・クラス
 - SerializableDatum クラス, 6-72
 - カスタム Java クラス (BetterDate), 6-17

- 接続 / 実行コンテキストの指定, 3-11
- 例, 3-12
- 自動コミット
 - 新しい接続での指定, 4-25
 - 既存の接続の変更, 4-26
 - 未サポート (サーバー側), 11-4
- 出力ディレクトリ
 - 生成される .class および .ser, 8-29
 - 生成される java, 8-31
- 出力パイプ、コンパイラのメッセージ, 8-80
- 出力ファイルとディレクトリのオプション (トランスレータ), 8-28
- ショート・ネーム (スキーマ名), 11-10
- シリアル化されたオブジェクト
 - RAW および BLOB 列に対する, 6-66
 - SerializableDatum クラス (サンプル), 6-72
 - イテレータ列, 6-71
 - 概要, 6-65
 - カスタム Java クラス, 6-68
 - ホスト変数として, 6-71
- シリアル化プロファイル (.ser)
 - .class への変換, 8-64
 - 生成されるプロファイル, 9-6

し

- 実行、コンテキスト
 - close() メソッド, 7-29
 - 概要, 7-22
 - 作成と指定, 7-24
 - 実行文の指定, 3-11
 - 状態メソッド, 7-25
 - 制御メソッド, 7-26
 - セーブポイント・メソッド, 7-29
 - 接続コンテキストとの関係, 7-23
 - 同期, 7-25
 - 取消しメソッド, 7-27
 - バッチ更新メソッド, 7-28
 - マルチスレッドとの関係, 7-31
 - メソッドの使用例, 7-30
- 実行コンテキストの同期, 7-25
- 実行文
 - PL/SQL ブロックの使用, 3-13
 - SQLJ 句, 3-9
 - 概要, 3-8
 - 規則, 3-8

す

- スキーマ・オブジェクト
 - 生成されたクラスの名前付け, 11-21
 - ソースの名前付け, 11-20
 - ロードされたクラスの名前付け, 11-10
 - ロードされたリソースの名前付け, 11-11
- スキーマ・オブジェクトの名前付け
 - 生成されたクラス, 11-21
 - ソース, 11-20
 - ロードされたクラス, 11-10
 - ロードされたリソース, 11-11
- スクロール可能なイテレータ
 - スクロール可能な位置指定イテレータ, 7-41
 - スクロール可能なインタフェース, 7-39
 - スクロール可能な名前指定イテレータ, 7-40
 - 宣言, 7-38
 - 機密, 7-38
- ステータス・メッセージ
 - カスタマイズ, A-8
 - トランスレータの有効化 / 無効化, 8-47
 - 変換, 9-14
- ストアド・ファンクション・コール, 3-55

ストアド・プロシージャ・コール, 3-54

ストリーム

一般的な使用方法、SQLJ, 5-13

クラスとメソッド, 5-21

サポート用クラス, 5-13

出力パラメータとして, 5-25

注意, 5-18

データの取だし, 5-19

データベースへのデータの送信, 5-15

ファンクションの戻り値, 5-26

例, 5-23

せ

静的 SQL、定義, 1-2

セーブポイント

ExecutionContext セーブポイント・メソッド, 7-29

セーブポイント文, 4-28

接続

JDBC との共有接続の終了, 7-52

JDBC トランザクション・メソッド, 7-47

JDBC の共有接続, 7-51

Oracle クラスによる接続先指定, 4-12

SQLJ データ・ソースからの, 7-14、7-18

検証, 2-14

自動コミットの指定, 4-25

自動コミットの変更, 4-26

終了, 4-10

設定, 2-13

単一または複数のデフォルト・コンテキストの
使用, 4-6

データベース接続（サーバー側）, 11-4

トランスレータ・オプション, 8-32

複数の宣言済みの接続コンテキストの使用, 4-12

接続キャッシングのための SqljConnCacheBean, 7-20

接続、コンテキスト

IMPLEMENTS 句の宣言, 7-10

JDBC 接続の取得, 7-9

JDBC 接続への変換, 7-48、7-50

SQLJ データ・ソースからの, 7-14、7-18

概要, 7-2

実行コンテキストとの関係, 7-23

実行コンテキストの取得, 7-9

実行文の指定, 3-11

実装と機能, 7-8

接続オブジェクトのインスタンス化, 7-6

接続コンテキスト・クラスの宣言, 7-5

接続の終了, 7-9

セマンティクス・チェック, 7-11

宣言, 3-4

デフォルトの接続の取得, 7-9

デフォルトの接続の設定, 7-9

複数の接続例, 7-7

文の接続の指定, 7-6

メソッド, 7-9

設定およびインストールの確認, 2-10

接頭辞

Java コンパイラに対するオプション設定, 8-59

JVM に対するオプション設定, 8-58

カスタマイザに対するオプション設定, 8-60

セマンティクス・チェック

SQLCheckerCustomizer オプション, A-40

SQLCheckerCustomizer の起動, A-39

URL の設定, 8-39

オプション, 8-67

オフライン・チェッカの指定, 8-71

オンライン・セマンティクス・チェッカ実行結果の
キャッシング, 8-75

オンライン・チェッカの指定, 8-73

オンライン・チェックの有効化（サーバー側）,
11-17

カスタマイザ・ハーネスとプロファイル, A-22
手順, 9-2

デフォルトの URL 接頭辞の設定, 8-42

デフォルトのチェッカ、Oracle チェッカ, 8-68

ドライバの登録, 8-43

パスワードの設定, 8-37

ユーザー・スキーマの設定によるオンライン・
チェックの有効化, 8-33

宣言

IMPLEMENTS 句, 3-5

WITH 句, 3-6

イテレータ宣言, 3-3

概要, 3-2

接続コンテキスト宣言, 3-4

前提、環境の, 2-2

そ

ソース・スキーマ・オブジェクトのネーミング, 11-20

ソースのチェック、拡張解決検索, 8-65

ソースのロードとサーバーでの変換, 11-14

ソース・ファイルとの行マッピング
jdb, 8-49
概要, 8-48
ソース・ファイルのエンコーディングのオプション,
8-28
ソース名 /Public クラス名のチェック, 8-81

た

代替環境のサポート, 8-76
代入文 (SET), 3-53
他の実行可能プログラムに対するオプション設定,
8-57
単純な接続での SqljConnBean, 7-19

ち

チェック、Public クラス名とソース名, 8-81
中間層での考慮事項, 4-45

つ

強い型指定のオブジェクトおよび参照, 6-51
強い型指定のカスタム Java クラス, 6-2
強い型指定のコレクション, 6-57

て

ディレクトリ
生成される .class および .ser, 8-29
生成される .java, 8-31
データ・ソースのサポート
dataSource (WITH 句接続コンテキスト), 3-7
SQLJ 固有のデータ・ソース, 7-14
SQLJ データ・ソースのインタフェース, 7-15
SQLJ データ・ソースのクラス, 7-16
概要, 7-12
自動コミット・モード, 7-13
接続の関連付け, 7-12
デフォルト・コンテキストの関連付け, 7-14
要件, 7-14
データベース接続の検証, 2-14
デバッグ
AuditorInstaller オプション, A-45
AuditorInstaller カスタマイザ, A-42
AuditorInstaller の起動, A-43
AuditorInstaller のコマンドラインの例, A-49

AuditorInstaller のランタイム出力, A-44
debug オプション、コンパイル (サーバー側),
11-18
JDeveloper, 10-27
SQLJ ソースから class への行マッピング (jdb),
8-49
SQLJ ソースとの行マッピング、class, 8-48
デバッグ・オプション (カスタマイザ・ハーネス),
A-20

デフォルト接続

Oracle の connect() による設定, 4-6
setDefaultContext() による設定, 4-10

デフォルトの URL 接頭辞, 8-42

デフォルトの URL 接頭辞のオプション (sqlj), 8-42

デフォルトの出力デバイス (サーバー側), 11-6

デフォルトのセマンティクス・チェッカ, 8-68

デフォルトのプロパティ・ファイル (トランスレータ), 8-19

と

動的 SQL

JDBC コード内, 7-48

PL/SQL コード, 3-13

定義, 1-2

登録 JDBC ドライバ

変換, 8-43

ランタイム用, 4-5

特殊処理のフラグ, 8-62

ドライバ登録オプション (sqlj-driver), 8-43

トランザクション

JDBC 接続メソッド, 7-47

アクセス・モードの設定, 7-46

概要, 4-24

基本的なトランザクション制御, 4-24

自動コミットと手動コミットとの違い, 4-24

自動コミットの指定, 4-25

自動コミットの変更, 4-26

手動コミットおよびロールバック, 4-26

詳細なトランザクション制御, 7-44

分離レベルの設定, 7-46

ロールバックのためのセーブポイント, 4-28

トランスレータ

Java と SQLJ のコード解析と構文チェック, 9-2

SQL のセマンティクス・チェックとオフライン

解析, 9-2

エラー、警告、情報メッセージ, 9-12

エラー・リスト, B-2
概要, 1-4
カスタマイズ, 9-10
グローバルゼーション・サポート, 9-18
コードの生成, 9-5
コンパイル, 9-9
終了コード, 9-15
出力 (サーバー側), 11-20
ステータス・メッセージ, 9-14
代替環境のサポート, 8-76
テスト, 2-15
内部操作, 9-2
入出力, 1-12
変換処理, 1-9
トランスレータからの警告, 8-44
トランスレータからの出力, 1-12
トランスレータの警告メッセージ, 9-12
トランスレータの終了コード, 9-15
トランスレータの情報メッセージ, 9-12

な

名前、JVM, 8-78
名前、コンパイラ, 8-78
名前指定イテレータ
 アクセス, 3-41
 インスタンス化と移入, 3-40
 使用, 3-38
 スクロール可能, 7-40
 宣言, 3-39

に

入力トランスレータへの, 1-12

ね

ネーミング要件および制限
 SQLJ のネームスペース, 4-43
 SQL のネームスペース, 4-44
 ファイル名, 4-44
 ローカル変数およびクラス (Java のネームスペース), 4-42
ネスト・イテレータ, 6-61
ネストした表
 アクセス, 6-57
 型, 6-4

操作, 6-60
挿入、SQLJ, 6-58
ネスト・イテレータの使用法, 6-61
ホスト式への取出し, 6-59

は

バージョン互換性 (Oracle カスタマイザ), A-24
バージョン・ナンバー・オプション
 (sqlj-version-xxxx), 8-25
パイプ、コンパイラの出力メッセージ, 8-80
バッチ更新
 暗黙的な実行コンテキストの使用, 10-20
 概要, 10-11
 更新カウント, 10-17
 再帰的コールイン, 10-22
 実行中のエラー状態, 10-21
 注意, 10-20
 バッチ可能かつ互換性のある文, 10-12
 バッチ制限, 10-18
 バッチの取消し, 10-16
 非互換の文に対するバッチ処理, 10-19
 明示的および暗黙的なバッチ実行, 10-14
 有効化 / 無効化, 10-13
バッチ更新機能
 暗黙的な実行コンテキストの使用, 10-20
 概要, 10-11
 更新カウント, 10-17
 再帰的コールイン, 10-22
 実行中のエラー状態, 10-21
 注意, 10-20
 バッチ可能かつ互換性のある文, 10-12
 バッチ制限, 10-18
 バッチの取消し, 10-16
 非互換の文に対するバッチ処理, 10-19
 明示的および暗黙的なバッチ実行, 10-14
 有効化 / 無効化, 10-13
パフォーマンスの強化, 10-2
パラメータ・サイズの定義, 10-23
パラメータ・サイズの定義に使用するヒント, 10-24
パラメータ・サイズの登録, 10-23
パラメータの定義 (サイズ)
 Oracle カスタマイザ optparamdefaults オプション, A-29
 Oracle カスタマイザ optparams オプション, A-28
SQLJ-optparamdefaults オプション, 8-54

SQLJ -optparams オプション, 8-52
概要, 10-23

ひ

必要なクラスのインポート, 4-29

ふ

ファイル名の要件と制限, 4-44
ファンクション・コール、ストアド, 3-55
プリフェッチ、行の, 10-3
フル・ネーム (スキーマ名), 11-10
プロパティ・ファイル (トランスレータ)
 概要, 8-16
 構文, 8-17
 デフォルトのプロパティ・ファイル, 8-19
 入力ファイルの設定, 8-22
プロファイル
 debug オプション, A-20
 jar ファイルの使用, A-36
 print オプション, A-21
 verify オプション, A-22
 オーディタによるデバッグ, A-42
 概要, 1-5
 コード生成時, A-2
 コード・レイヤー, A-42
 実行時の機能, A-9
 生成されるプロファイル, 9-6
 バイナリ移植性, 1-6
 プロファイル・エントリの例, A-3
 プロファイルについて, A-2
プロファイルキー・クラス, 9-6
プロファイルの jar ファイルの使用, A-36
プロファイルのカスタマイズ, 「カスタマイズ」を
 参照, 9-10
プロファイルのバイナリ移植性, 1-6
文のキャッシング, 10-4
分離レベルの設定 (トランザクション), 7-46

へ

変換、クライアント側、サーバーで実行, 11-8
変換、サーバー側、サーバーで実行, 11-14

ほ

ホスト式
 JDBC 2.0 でサポートされている型, 5-8
 Oracle8i での型サポート, 5-11
 イテレータおよび結果セット、ホスト変数, 3-47
 概要, 3-15
 型サポートの概要, 5-2
 型のサポート、Oracle8/Oracle7, 5-11
 基本的な構文, 3-15
 識別子によるバインド, 8-66
 実行時評価, 3-20
 実行時評価の例, 3-21
 制限, 3-29
 ネストした表の取出し, 6-59
 例, 3-18

ま

マッピング、代替クラス (UDT), 6-38
マルチスレッド
 SQLJ 概要, 7-31
 サーバー, 11-23
 サンプル・アプリケーション, 7-33
 実行コンテキストとの関係, 7-31

め

メソッドのサポート、オブジェクト, 6-9
メソッド・ラッパー (JPub)、実装, 6-43
メタ・バインド式 (動的 SQL), 7-57
メッセージ・パイプ (コンパイラ), 8-80
メンバー名 (オブジェクト), 6-42

も

文字エンコーディング
 native2ascii の使用, 9-29
 概要, 9-20
 コマンドラインの例, 9-24
 実行時の設定, 9-28
 ソース, 9-22
 メッセージ, 9-23

ゆ

ユーザー定義型, 6-21

よ

- 要件、環境の, 2-2、2-3
- 弱い型指定のイテレータ, 7-37
- 弱い型指定のオブジェクト / コレクション型
 - サポート, 6-74
 - 制限, 6-75
- 弱い型指定のカスタム Java クラス, 6-2

ら

- ラッパー・クラス、NULL の処理用, 4-18
- ラッパー・メソッドの生成 (JPub), 6-36
- ランタイム
 - JDBC ドライバの選択と登録, 4-5
 - エラーの分類, 9-18
 - エラー・リスト, B-44
 - 概要, 1-4
 - 機能, 9-15
 - グローバル化・サポート, 9-18
 - 接続の設定, 2-13
 - テスト, 2-15
 - デバッグ出力 (AuditorInstaller), A-44
 - パッケージ, 9-16
 - プロファイルの機能, A-9
 - ランタイム処理の手順, 1-15
 - ランタイムに対し生成されたコール, 9-8

り

- リソース・スキーマ・オブジェクトのネーミング
 - ロード済み, 11-11

れ

- 例外
 - SQLException サブクラスの使用方法, 4-23
 - 処理, 4-21
 - 例外処理の設定, 4-30
 - 例外処理の要件, 4-20
- 列型 / サイズの定義, 10-22
- 列の定義、型 / サイズ
 - Oracle カスタマイザ optcols オプション, A-26
 - SQLJ-optcols オプション, 8-51
 - 概要, 10-22
- 列の登録、型 / サイズ, 10-22
- レポート・オプション (トランスレータ), 8-43

関係動作、JDBC

- イテレータと結果セット, 7-53
- 接続コンテキストと接続, 7-48

ろ

- ロールバック
 - イテレータおよび結果セットに対する影響, 4-27
 - 手動, 4-26
 - セーブポイント, 4-28
- ロケール
 - コマンドラインの例, 9-24
 - 実行時の設定, 9-28
 - メッセージ, 9-23

