







Java アプリケーションの開発

#### ご注意

本書および本書で紹介する製品をご使用になる前に、465 ページの『付録 B. 特記事項』に記載されている情報をお読みください。

#### 当版に関する特記事項

本書には、IBM の専有情報が含まれています。その情報は、使用許諾条件に基づき提供され、著作権により保護されています。本書に記載される情報には、いかなる製品の保証も含まれていません。また、本書で提供されるいかなる記述も、製品保証として解釈すべきではありません。

IBM 資料は、オンラインでご注文いただくことも、ご自分の国または地域の IBM 担当員を通してお求めいただくこともできます。

- オンラインで資料を注文するには、[www.ibm.com/shop/publications/order](http://www.ibm.com/shop/publications/order) にある IBM Publications Center をご利用ください。
- ご自分の国または地域の IBM 担当員を見つけるには、[www.ibm.com/planetwide](http://www.ibm.com/planetwide) にある IBM Directory of Worldwide Contacts をお調べください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

IBM 発行のマニュアルに関する情報のページ

<http://www.ibm.com/jp/manuals/>

こちらから、日本語版および英語版のオンライン・ライブラリーをご利用いただけます。また、マニュアルに関するご意見やご感想を、上記ページよりお送りください。今後の参考にさせていただきます。

(URL は、変更になる場合があります)

お客様の環境によっては、資料中の円記号がバックスラッシュと表示されたり、バックスラッシュが円記号と表示されたりする場合があります。

原 典： SC23-5853-00  
DB2 Version 9.5 for Linux, UNIX, and Windows  
Developing Java Applications

発 行： 日本アイ・ビー・エム株式会社

担 当： ナショナル・ランゲージ・サポート

# 目次

本書について . . . . .	vii	JDBC アプリケーションでのバッチ更新の作成 . . . . .	48
Who should use this book. . . . .	vii	ParameterMetaData メソッドを使用した、 PreparedStatement のパラメーターに関する情報の 取得 . . . . .	50
<b>第 1 章 DB2 用の Java アプリケーショ ン開発 . . . . .</b>	<b>1</b>	JDBC アプリケーションにおけるデータの取り出 し . . . . .	51
JDBC および SQLJ のサポートされるドライバー . . . . .	1	JDBC アプリケーションでのストアード・プロシ ージャの呼び出し . . . . .	64
JDBC ドライバーと DB2 データベースの互換性 . . . . .	3	IBM Data Server Driver for JDBC and SQLJ 使用 時の JDBC アプリケーションでの LOB . . . . .	68
<b>第 2 章 IBM Data Server Driver for JDBC and SQLJ のインストール . . . . .</b>	<b>5</b>	IBM Data Server Driver for JDBC and SQLJ 使用 時の JDBC での ROWID . . . . .	75
DB2Binder ユーティリティ . . . . .	9	JDBC アプリケーションでの特殊タイプ . . . . .	76
DB2LobTableCreator ユーティリティ . . . . .	17	JDBC アプリケーションでの ARRAY パラメータ ーを使用したストアード・プロシージャの呼び 出し . . . . .	77
IBM Data Server Driver for JDBC and SQLJ の構成 プロパティのカスタマイズ . . . . .	18	JDBC アプリケーションでのセーブポイント . . . . .	78
Java プログラムから DB2 for z/OS サーバーにアク セスするための特別なセットアップ . . . . .	19	JDBC アプリケーションでの自動生成キーの取得 . . . . .	79
DB2 UDB (OS/390 および z/OS 版) バージョン 7 サーバーを使用する分散トランザクション用の DB2T4XAIndoubtUtil . . . . .	21	IBM Data Server Driver for JDBC and SQLJ のみ のメソッドによって拡張クライアント情報をデー タ・ソースに提供する . . . . .	82
HP-UX 環境で Java ルーチンを実行するための特別 なセットアップ . . . . .	24	クライアント情報プロパティによって拡張クラ イアント情報をデータ・ソースに提供する . . . . .	83
<b>第 3 章 JDBC アプリケーション・プロ グラミング . . . . .</b>	<b>27</b>	JDBC アプリケーションでのオプティミスティッ ク・ロック . . . . .	85
簡単な JDBC アプリケーションの例 . . . . .	27	JDBC アプリケーションでの XML データ . . . . .	87
JDBC アプリケーションによるデータ・ソースへの 接続の方法 . . . . .	29	JDBC アプリケーションでの XML 列の更新 . . . . .	88
DB2 アプリケーションによる DriverManager イン ターフェースと DB2 JDBC Type 2 ドライバー を使用したデータ・ソースへの接続 . . . . .	31	JDBC アプリケーションにおける XML データの 取り出し . . . . .	90
DriverManager インターフェースと IBM Data Server Driver for JDBC and SQLJ を使用したデ ータ・ソースへの接続 . . . . .	32	Java アプリケーションでの XML パラメーターを 使用したルーチンの呼び出し . . . . .	94
DataSource インターフェースを使用したデータ ・ソースへの接続 . . . . .	37	XML スキーマの登録および除去の Java サポート . . . . .	95
使用する IBM Data Server Driver for JDBC and SQLJ 接続のタイプを判別する方法 . . . . .	39	JDBC アプリケーションにおけるトランザクシ ョン制御 . . . . .	97
JDBC 接続オブジェクト . . . . .	40	IBM Data Server Driver for JDBC and SQLJ の分 離レベル . . . . .	97
DataSource オブジェクトの作成およびデプロイ . . . . .	40	JDBC トランザクションのコミットまたはロール バック . . . . .	98
JDBC サポート用の Java パッケージ . . . . .	42	デフォルトの JDBC 自動コミット・モード . . . . .	98
DatabaseMetaData メソッドを使用した、データ・ソ ースに関する情報の取得 . . . . .	43	IBM Data Server Driver for JDBC and SQLJ での例 外と警告 . . . . .	99
JDBC アプリケーションでの変数 . . . . .	44	IBM Data Server Driver for JDBC and SQLJ 使 用時の SQLException の処理 . . . . .	102
SQL を実行するための JDBC インターフェース . . . . .	45	IBM Data Server Driver for JDBC and SQLJ 使 用時の SQLWarning の処理 . . . . .	105
Statement.executeUpdate メソッドによるデータバ ース・オブジェクトの作成と変更 . . . . .	45	BatchUpdateException からの情報の取り出し . . . . .	106
PreparedStatement.executeUpdate メソッドを使用し た表のデータの更新 . . . . .	46	DB2 JDBC Type 2 ドライバー使用時の SQLException の処理 . . . . .	107
DB2 for z/OS サーバーに対する JDBC executeUpdate メソッド . . . . .	48	DB2 JDBC Type 2 ドライバー使用時の SQLWarning の処理 . . . . .	108

IBM Data Server Driver for JDBC and SQLJ クライアント・リルート・サポート . . . . .	109
クライアント・リルート用の DB2 for z/OS サーバーの構成 . . . . .	110
クライアント上での IBM Data Server Driver for JDBC and SQLJ クライアント・リルート操作 . . . . .	111
JDBC アプリケーションでのデータ・ソースからの切断 . . . . .	114

## 第 4 章 SQLJ アプリケーション・プログラミング . . . . . 115

簡単な SQLJ アプリケーションの例 . . . . .	115
SQLJ を使用したデータ・ソースへの接続 . . . . .	117
SQLJ の接続技法 1: JDBC DriverManager インターフェース . . . . .	118
SQLJ の接続技法 2: JDBC DriverManager インターフェース . . . . .	119
SQLJ の接続技法 3: JDBC DataSource インターフェース . . . . .	121
SQLJ の接続技法 4: JDBC DataSource インターフェース . . . . .	123
SQLJ の接続技法 5: すでに作成されている接続の使用 . . . . .	124
SQLJ の接続技法 6: デフォルトの接続の使用 . . . . .	124
SQLJ サポート用の Java パッケージ . . . . .	125
SQLJ アプリケーションでの変数 . . . . .	125
SQLJ アプリケーションでのコメント . . . . .	127
SQLJ アプリケーションでの SQL ステートメントの実行 . . . . .	127
SQLJ アプリケーションでの DB2 オブジェクトの作成および変更 . . . . .	127
SQLJ アプリケーションでの位置指定 UPDATE および DELETE 操作の実行 . . . . .	128
SQLJ アプリケーションでのデータの検索 . . . . .	137
SQLJ アプリケーションでのストアード・プロシージャの呼び出し . . . . .	149
IBM Data Server Driver for JDBC and SQLJ 使用時の SQLJ アプリケーションでの LOB . . . . .	151
同じアプリケーションでの SQLJ および JDBC SQLJ での SQL ステートメントの実行の制御 . . . . .	157
IBM Data Server Driver for JDBC and SQLJ 使用時の SQLJ での ROWID . . . . .	157
SQLJ アプリケーションにおける特殊タイプ . . . . .	159
ARRAY パラメーターを持つストアード・プロシージャの、SQLJ アプリケーションでの呼び出し . . . . .	160
SQLJ アプリケーションにおけるセーブポイント . . . . .	161
SQLJ アプリケーションでの XML データ . . . . .	161
SQLJ アプリケーションでの XML 列の更新 . . . . .	162
SQLJ アプリケーションにおける XML データの取り出し . . . . .	164
SDK for Java バージョン 5 の機能の SQLJ での利用 . . . . .	166
SQLJ アプリケーションにおけるトランザクション制御 . . . . .	169
SQLJ トランザクションの分離レベルの設定 . . . . .	169

SQLJ トランザクションのコミットまたはロールバック . . . . .	169
SQLJ アプリケーションで SQL のエラーと警告を処理する . . . . .	170
SQLJ アプリケーションでの SQL エラーの処理 . . . . .	170
SQLJ アプリケーションでの SQL 警告の処理 . . . . .	171
SQLJ アプリケーションでのデータ・ソースへの接続のクローズ . . . . .	171

## 第 5 章 IBM Data Server Driver for JDBC and SQLJ 使用時のセキュリティ . . . . . 173

IBM Data Server Driver for JDBC and SQLJ 使用時のユーザー ID およびパスワード・セキュリティ . . . . .	175
IBM Data Server Driver for JDBC and SQLJ 使用時のユーザー ID のみのセキュリティ . . . . .	177
暗号化されたパスワード・セキュリティ、暗号化されたユーザー ID セキュリティ、または IBM Data Server Driver for JDBC and SQLJ 使用時の暗号化されたユーザー ID とパスワード・セキュリティ . . . . .	178
IBM Data Server Driver for JDBC and SQLJ 使用時の Kerberos セキュリティ . . . . .	179
IBM Data Server Driver for JDBC and SQLJ のセキュリティ・プラグインのサポート . . . . .	183
IBM Data Server Driver for JDBC and SQLJ のトラステッド・コンテキストのサポート . . . . .	185
SSL 用の IBM Data Server Driver for JDBC and SQLJ サポート . . . . .	187
IBM Data Server Driver for JDBC and SQLJ 使用時の SQLJ アプリケーション準備のセキュリティ . . . . .	188

## 第 6 章 DB2 JDBC Type 2 ドライバ使用時のセキュリティ . . . . . 191

## 第 7 章 Java データベース・アプリケーションの構築 . . . . . 193

JDBC アプレットの構築 . . . . .	193
JDBC アプリケーションの構築 . . . . .	193
JDBC ルーチンの構築 . . . . .	194
SQLJ アプレットの構築 . . . . .	195
SQLJ アプリケーションの構築 . . . . .	196
Java アプレットに関する考慮事項 . . . . .	197
UNIX の SQLJ アプリケーションおよびアプレット・オプション . . . . .	198
Windows の SQLJ アプリケーションおよびアプレット・オプション . . . . .	199
SQL ルーチンの構築 . . . . .	199
UNIX の SQLJ ルーチン・オプション . . . . .	200
Windows の SQLJ ルーチン・オプション . . . . .	201

## 第 8 章 IBM Data Server Driver for JDBC and SQLJ での問題診断 . . . . . 203

構成プロパティを使用して JDBC トレースを開始する例 . . . . .	205
--	-----

IBM Data Server Driver for JDBC and SQLJ 使用時の  
のトレース・プログラムの例 . . . . . 206

## 第 9 章 IBM Data Server Driver for JDBC and SQLJ のシステム・モニタ ー . . . . . 211

IBM Data Server Driver for JDBC and SQLJ のリモ  
ート・トレース・コントローラー . . . . . 214  
リモート・トレース・コントローラーの使用可能  
化 . . . . . 214  
リモート・トレース・コントローラーへのアクセ  
ス . . . . . 216

## 第 10 章 Java 2 Platform, Enterprise Edition . . . . . 219

Java 2 Platform, Enterprise Edition のアプリケーシ  
ョン・コンポーネントのサポート . . . . . 219  
Java 2 Platform, Enterprise Edition コンテナ . . . . . 220  
Java 2 Platform, Enterprise Edition Server . . . . . 221  
Java 2 Platform, Enterprise Edition のデータベース  
要件 . . . . . 221  
Java Naming and Directory Interface (JNDI) . . . . . 221  
Java トランザクション管理 . . . . . 221  
JTA メソッドを使用する分散トランザクシ  
ョンの例 . . . . . 223  
XAResource インスタンスのトランザクシ  
ョン・タイムアウト値の設定 . . . . . 227  
Enterprise Java Beans . . . . . 228

## 第 11 章 JDBC および SQLJ 接続プ ールのサポート . . . . . 231

## 第 12 章 JDBC 接続コンセントレータ ーおよび Sysplex ワークロード・バラ ンシング . . . . . 233

IBM Data Server Driver for JDBC and SQLJ 接続コ  
ンセントレーターを使用可能にする例 . . . . . 234  
IBM Data Server Driver for JDBC and SQLJ 接続コ  
ンセントレーターおよび Sysplex ワークロード・バラ  
ンシングのモニター技法 . . . . . 236

## 第 13 章 JDBC および SQLJ の参照情 報 . . . . . 239

Java アプリケーションでデータベース・データ・タ  
イプにマップするデータ・タイプ . . . . . 239  
IBM Data Server Driver for JDBC and SQLJ のプロ  
パティ . . . . . 245  
サポートされるすべてのデータベース製品に共通  
の IBM Data Server Driver for JDBC and SQLJ  
のプロパティ . . . . . 246  
DB2 for z/OS および DB2 Database for Linux,  
UNIX, and Windows に共通の IBM Data Server  
Driver for JDBC and SQLJ のプロパティ . . . . . 253

DB2 for z/OS および IBM Informix Dynamic  
Server に共通の IBM Data Server Driver for  
JDBC and SQLJ のプロパティ . . . . . 263  
IBM Informix Dynamic Server および DB2  
Database for Linux, UNIX, and Windows データ  
ベース・サーバーに共通の IBM Data Server  
Driver for JDBC and SQLJ のプロパティ . . . . . 264  
DB2 Database for Linux, UNIX, and Windows 用  
の IBM Data Server Driver for JDBC and SQLJ  
のプロパティ . . . . . 265  
DB2 for z/OS 用の IBM Data Server Driver for  
JDBC and SQLJ のプロパティ . . . . . 267  
IBM Informix Dynamic Server 用の IBM Data  
Server Driver for JDBC and SQLJ のプロパティ  
ー . . . . . 273

IBM Data Server Driver for JDBC and SQLJ 構成プ  
ロパティ . . . . . 277  
JDBC API でのドライバーのサポート . . . . . 281  
SQLJ ステートメント参照情報 . . . . . 311  
SQLJ 節 . . . . . 311  
SQLJ ホスト式 . . . . . 311  
SQLJ インプリメント節 . . . . . 312  
SQLJ with 節 . . . . . 312  
SQLJ 接続宣言節 . . . . . 314  
SQLJ イテレーター宣言節 . . . . . 315  
SQLJ 実行可能節 . . . . . 316  
SQLJ コンテキスト節 . . . . . 317  
SQLJ ステートメント節 . . . . . 317  
SQLJ SET TRANSACTION 節 . . . . . 319  
SQLJ 代入節 . . . . . 319  
SQLJ イテレーター変換節 . . . . . 320

sqlj.runtime パッケージ内のインターフェースおよび  
クラス . . . . . 321  
sqlj.runtime.ConnectionContext インターフェース 322  
sqlj.runtime.ForUpdate インターフェース . . . . . 327  
sqlj.runtime.NamedIterator インターフェース . . . . . 327  
sqlj.runtime.PositionedIterator インターフェース 328  
sqlj.runtime.ResultSetIterator インターフェース 328  
sqlj.runtime.Scrollable インターフェース . . . . . 331  
sqlj.runtime.ASCIIStream クラス . . . . . 334  
sqlj.runtime.BinaryStream クラス . . . . . 335  
sqlj.runtime.CharacterStream クラス . . . . . 335  
sqlj.runtime.ExecutionContext クラス . . . . . 337  
sqlj.runtime.SQLNullException クラス . . . . . 345  
sqlj.runtime.StreamWrapper クラス . . . . . 345  
sqlj.runtime.UnicodeStream クラス . . . . . 347

JDBC に対する IBM Data Server Driver for JDBC  
and SQLJ 拡張機能 . . . . . 347  
DB2Administrator クラス . . . . . 349  
DB2BaseDataSource クラス . . . . . 350  
DB2CataloguedDatabase クラス . . . . . 355  
DB2ClientRerouteServerList クラス . . . . . 356  
DB2Connection インターフェース . . . . . 357  
DB2ConnectionPoolDataSource クラス . . . . . 373  
DB2DatabaseMetaData インターフェース . . . . . 375  
DB2Diagnosable インターフェース . . . . . 376



DB2ExceptionFormatter クラス . . . . .	377
DB2JCCPlugin クラス . . . . .	377
DB2PooledConnection クラス . . . . .	378
DB2PoolMonitor クラス . . . . .	381
DB2PreparedStatement インターフェース . . . . .	383
DB2ResultSet インターフェース . . . . .	384
DB2ResultSetMetaData インターフェース . . . . .	384
DB2RowID インターフェース . . . . .	385
DB2SimpleDataSource クラス . . . . .	386
DB2Sqlca クラス . . . . .	386
DB2Statement インターフェース . . . . .	387
DB2SystemMonitor インターフェース . . . . .	390
DB2TraceManager クラス . . . . .	393
DB2TraceManagerMXBean インターフェース . . . . .	397
DB2XADataSource クラス . . . . .	401
DB2Xml インターフェース . . . . .	403
現在の IBM Data Server Driver for JDBC and SQLJ と以前の DB2 JDBC ドライバーとの JDBC の相違 点 . . . . .	405
ResultSetMetaData.getColumnNames および ResultSetMetaData.getColumnLabels の値の例 . . . . .	414
IBM Data Server Driver for JDBC and SQLJ と他の DB2 JDBC ドライバーとの SQLJ の相違点 . . . . .	416
IBM Data Server Driver for JDBC and SQLJ に影響 を与える SDK for Java の相違点 . . . . .	418
IBM Data Server Driver for JDBC and SQLJ から発 行されるエラー・コード . . . . .	419
IBM Data Server Driver for JDBC and SQLJ から発 行される SQLSTATE . . . . .	426

IBM Data Server Driver for JDBC and SQLJ のパー ジョンおよび環境の情報を確認する方法 . . . . .	428
SQLJ プログラムの準備のためのコマンド . . . . .	430
sqlj - SQLJ 変換プログラム . . . . .	430
db2sqljcustomize - SQLJ プロファイル・カスタ マイザー . . . . .	434
db2sqljbind - SQLJ プロファイル・バインダー . . . . .	447
db2sqljprint - SQLJ プロファイル・プリンター . . . . .	453

**付録 A. DB2 技術情報の概説 . . . . . 455**

DB2 テクニカル・ライブラリー (ハードコピーまた は PDF 形式) . . . . .	456
DB2 の印刷資料の注文方法 . . . . .	458
コマンド行プロセッサから SQL 状態ヘルプを表 示する . . . . .	459
異なるバージョンの DB2 インフォメーション・セ ンターへのアクセス . . . . .	459
DB2 インフォメーション・センターにおける特定 の言語でのトピックの表示 . . . . .	460
コンピューターまたはイントラネット・サーバーに インストールされた DB2 インフォメーション・セ ンターの更新 . . . . .	461
DB2 チュートリアル . . . . .	463
DB2 トラブルシューティング情報 . . . . .	463
ご利用条件 . . . . .	464

**付録 B. 特記事項 . . . . . 465**

**索引 . . . . . 469**



---

## 本書について

本書では、DB2<sup>®</sup> for Linux<sup>®</sup>, UNIX<sup>®</sup>, and Windows<sup>®</sup> の Java<sup>™</sup> サポートについて説明します。Java アプリケーション・プログラムからリレーショナル・データベースへのアクセスは、このサポートによって可能になります。

---

## Who should use this book

This book is for the following users:

- DB2 for Linux, UNIX, and Windows application developers who are familiar with Structured Query Language (SQL) and who know the Java programming language.
- DB2 for Linux, UNIX, and Windows system programmers who are installing JDBC and SQLJ support.



---

## 第 1 章 DB2 用の Java アプリケーション開発

DB2 データベース・システムでは、JDBC を使用して Java で作成されたクライアント・アプリケーションおよびアプレット用のドライバー・サポートと、組み込み SQL for Java (SQLJ) 用のドライバー・サポートが提供されます。

JDBC とは、Java アプリケーションによるリレーショナル・データベースへのアクセスに使用されるアプリケーション・プログラミング・インターフェース (API) のことです。DB2 の JDBC のサポートを使用すると、ローカル DB2 データまたは DRDA<sup>®</sup> がサポートされたサーバー上のリモート・リレーショナル・データにアクセスする Java アプリケーションを作成することができます。

SQLJ では、Java アプリケーションでの組み込み静的 SQL のサポートが提供されます。SQLJ は、最初は、動的 SQL JDBC モデルを静的 SQL モデルで補完するために、IBM<sup>®</sup>、Oracle、および Tandem によって開発されました。

Java アプリケーションでは一般に、動的 SQL に JDBC を、静的 SQL に SQLJ を使用します。ただし、SQLJ は JDBC との相互運用が可能であるため、アプリケーション・プログラムでは同じ作業単位内で JDBC と SQLJ を使用できます。

---

### JDBC および SQLJ のサポートされるドライバー

DB2 製品には、2 つのタイプの JDBC ドライバー・アーキテクチャーのサポートが含まれています。

JDBC 仕様に従って、JDBC ドライバー・アーキテクチャーには以下の 4 つのタイプがあります。

#### タイプ 1

別のデータ・アクセス API (Open Database Connectivity (ODBC) など) へのマッピングとして JDBC API をインプリメントするドライバー。このタイプのドライバーは通常、ネイティブ・ライブラリーに依存しており、移植性が制限されています。DB2 データベース・システムはタイプ 1 ドライバーをサポートしていません。

#### タイプ 2

一部は Java プログラミング言語、一部はネイティブ・コードで作成されているドライバー。ドライバーは、接続先のデータ・ソースに固有のネイティブ・クライアント・ライブラリーを使用します。ネイティブ・コードのために、移植性は制限されています。

#### タイプ 3

pure Java クライアントを使用し、データベースに依存しないプロトコルを使用してデータベースと通信するドライバー。それから、データベースはクライアントの要求をデータ・ソースに送達します。DB2 データベース・システムはタイプ 3 ドライバーをサポートしていません。

## タイプ 4

pure Java であり、固有のデータ・ソース用のネットワーク・プロトコルをインプリメントするドライバー。クライアントはデータ・ソースに直接接続します。

DB2 Database for Linux, UNIX, and Windows は以下のドライバーをサポートしています。

ドライバー名	パッケージ	ドライバー・タイプ
Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバー	db2java.zip	タイプ 2
IBM Data Server Driver for JDBC and SQLJ	<ul style="list-style-type: none"><li>JDBC 3.0 サポート用の db2jcc.jar および sqlj.zip</li><li>JDBC 4.0 サポート用の db2jcc4.jar および sqlj.zip</li></ul>	タイプ 2 およびタイプ 4

## IBM Data Server Driver for JDBC and SQLJ (タイプ 2 およびタイプ 4)

IBM Data Server Driver for JDBC and SQLJ は、JDBC タイプ 2 および JDBC タイプ 4 の動作を組み込んだ単一ドライバーです。アプリケーションが IBM Data Server Driver for JDBC and SQLJ をロードすると、タイプ 2 およびタイプ 4 のインプリメンテーション用に単一のドライバー・インスタンスがロードされます。アプリケーションは、この単一のドライバー・インスタンスを使用して、タイプ 2 およびタイプ 4 接続を行うことができます。タイプ 2 およびタイプ 4 接続は同時に行うことができます。IBM Data Server Driver for JDBC and SQLJ タイプ 2 ドライバーの動作を *IBM Data Server Driver for JDBC and SQLJ Type 2 接続* と言います。IBM Data Server Driver for JDBC and SQLJ タイプ 4 ドライバーの動作を *IBM Data Server Driver for JDBC and SQLJ Type 4 接続* と言います。

2 つのバージョンの IBM Data Server Driver for JDBC and SQLJ が使用可能です。IBM Data Server Driver for JDBC and SQLJ バージョン 3.5 は JDBC 3.0 準拠です。IBM Data Server Driver for JDBC and SQLJ バージョン 4.0 は JDBC 3.0 準拠で、一部の JDBC 4.0 機能をサポートしています。

IBM Data Server Driver for JDBC and SQLJ は、以下の JDBC および SQLJ 機能をサポートしています。

- JDBC 3.0 仕様で記述されているすべてのメソッド。『JDBC API でのドライバーのサポート』を参照してください。
- IBM Data Server Driver for JDBC and SQLJ バージョン 4.0 をインストールしている場合、JDBC 4.0 仕様で記述されている一部のメソッド。
- Java アプリケーションからのデータ・アクセスを単純化する、SQLJ 標準によって定義された SQLJ アプリケーション・プログラミング・インターフェース。
- 接続プール用に使用可能にされている接続。WebSphere® Application Server または別のアプリケーション・サーバーによって接続プールが行われます。
- Java ユーザー定義関数およびストアド・プロシージャ (IBM Data Server Driver for JDBC and SQLJ Type 2 接続のみ)。

IBM Data Server Driver for JDBC and SQLJ は、Java ルーチンのデフォルト・ドライバです。

- 分散トランザクション管理のサポート。このサポートは、Java 2 Platform, Enterprise Edition (J2EE) Java Transaction Service (JTS)、および Java Transaction API (JTA) 仕様をインプリメントします。これらは分散トランザクションの X/Open 規格に準拠しています (*Distributed Transaction Processing: The XA Specification*。 <http://www.opengroup.org> で入手可能)。

## Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバ (DB2 JDBC Type 2 ドライバ) (非推奨)

DB2 JDBC Type 2 ドライバは、Java アプリケーションが JDBC を介して DB2 への呼び出しを行うようにします。DB2 JDBC Type 2 ドライバへの呼び出しは、Java ネイティブ・メソッドに変換されます。DB2 JDBC Type 2 ドライバは DB2 CLI インターフェースを使用して DB2 データベースと通信します。このドライバを使用する Java アプリケーションは DB2 クライアント上で実行する必要があります。JDBC 要求はこのクライアントを介して DB2 データベースに送信されます。DB2 JDBC アプリケーション・ドライバを使用して DB2 for i5/OS® データ・ソースまたは DB2 for z/OS 環境内のデータ・ソースにアクセスするには、その前に DB2 Connect™ をインストールする必要があります。

DB2 JDBC Type 2 ドライバは、以下の JDBC および SQLJ 機能をサポートしています。

- JDBC 1.2 仕様で記述されているほとんどのメソッド、および JDBC 2.0 仕様で記述されている一部のメソッド。
- すべての JDBC メソッドに相当する操作を実行する SQLJ ステートメント
- 接続プール
- 分散トランザクション
- Java ユーザー定義関数およびストアド・プロシージャ

Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバは、今後のリリースではサポートされません。そのため、IBM Data Server Driver for JDBC and SQLJ に移行することを考慮してください。

## JDBC ドライバと DB2 データベースの互換性

IBM Data Server Driver for JDBC and SQLJ Type 4 接続を使用して、異なるリリース・レベルで DB2 データベースに接続することができます。

IBM Data Server Driver for JDBC and SQLJ は、常に以前のリリース・レベルの DB2 データベースと下位互換性を持ちます。例えば、IBM Data Server Driver for JDBC and SQLJ Type 4 接続では、DB2 Database for Linux, UNIX, and Windows バージョン 9.5 に付属する IBM Data Server Driver for JDBC and SQLJ バージョン 3.50 から DB2 バージョン 8 データベースまでがサポートされます。

ドライバが稼働するアプリケーションが新規フィーチャーを全く使用しない場合、IBM Data Server Driver for JDBC and SQLJ は次のバージョンの DB2 データベースと上位互換性を持ちます。例えば、IBM Data Server Driver for JDBC and SQLJ Type 4 接続では、DB2 Database for Linux, UNIX, and Windows バージョン 8 フィックスパック 12 に付属する IBM Data Server Driver for JDBC and SQLJ

バージョン 2.9.31 から DB2 バージョン 9.5 データベース (ドライバーが稼働するアプリケーションに DB2 バージョン 9.5 のフィーチャーが含まれていない場合) までがサポートされます。

---

## 第 2 章 IBM Data Server Driver for JDBC and SQLJ のインストール

IBM Data Server Driver for JDBC and SQLJ のインストール後に、JDBC または SQLJ アプリケーションを準備して実行することができます。

IBM Data Server Driver for JDBC and SQLJ をインストールする前に、以下のソフトウェアが必要です。

- SDK for Java 1.4.2 以降。

IBM Data Server Runtime Client 以外のすべての DB2 製品。DB2 Database for Linux, UNIX, and Windows のインストール・プロセスにより、SDK for Java バージョン 5 は自動的にインストールされます。

JDBC 4.0 機能を使用する場合は、SDK for Java 6 以降をインストールする必要があります。

JDBC または SQLJ アプリケーションをシステム上で実行する予定があるが、準備する予定はない場合、必要なのは Java ランタイム環境だけです。

- JVM ネイティブ・スレッド・サポート

DB2 データベースにアクセスする Java アプリケーションを実行する JVM には、ネイティブ・スレッド・サポートが組み込まれていなければなりません。THREADS\_FLAG 環境変数を「native」に設定することで、ネイティブ・スレッドを一部の JVM に対するデフォルトのスレッド・サポートとして指定できます。ネイティブ・スレッドをシステム上のデフォルトにする手順については、Java 環境の資料を参照してください。

- System i サーバーの Unicode サポート

いずれかの SQLJ または JDBC プログラムが IBM Data Server Driver for JDBC and SQLJ Type 4 接続を使用して DB2 for i5/OS サーバーに接続する場合、OS/400® オペレーティング・システムは Unicode UTF-8 コード化スキームをサポートする必要があります。以下の表は、Unicode UTF-8 サポートに必要な OS/400 PTF をリストしています。

表 1. Unicode UTF-8 サポート用の OS/400 PTF

OS/400 のバージョン	PTF 番号
V5R3 以降	なし (サポートは組み込まれている)
V5R2	SI06541、SI06796、SI07557、SI07564、 SI07565、SI07566、および SI07567
V5R1	SI06308、SI06300、SI06301、SI06302、 SI06305、SI06307、および SI05872

- HP-UX クライアントおよびサーバーの Java サポート



*HP-UX* サーバー: IBM Data Server Driver for JDBC and SQLJ は、HP-UX のデフォルトの文字セットである Roman8 のデータベースはサポートしません。したがって、IBM Data Server Driver for JDBC and SQLJ を使用してアクセスする予定の HP-UX サーバー上にデータベースを作成するには、別の文字セットでデータベースを作成する必要があります。

*HP-UX* クライアントおよびサーバー: HP-UX システム上の Java 環境では、IBM Data Server Driver for JDBC and SQLJ でストアド・プロシージャを実行するために特別なセットアップが必要です。

詳しくは、24 ページの『HP-UX 環境で Java ルーチンを実行するための特別なセットアップ』を参照してください。

これらのステップに従って IBM Data Server Driver for JDBC and SQLJ をインストールします。

1. DB2 Database for Linux, UNIX, and Windows のインストール・プロセス中に、UNIX または Linux 上では Java support、Windows 上では JDBC support を選択します。これらの選択はデフォルトです。DB2 Database for Linux, UNIX, and Windows が JDBC サポートなしでインストール済みの場合、インストール・プロセスを Custom モードで実行して JDBC サポートを追加することができます。

Java support または JDBC support を選択すると、インストール・プロセスにより以下のアクションが実行されます。

- IBM Data Server Driver for JDBC and SQLJ のクラス・ファイルをインストールします。

これらのファイルは、Windows システムの場合は `sqllib\java` ディレクトリ、UNIX または Linux システムの場合は `sqllib/java` ディレクトリに置かれます。

これらのファイル名は以下のとおりです。

#### **db2jcc.jar または db2jcc4.jar**

**JDBC 3.0 以前の機能**しか組み込まれていないバージョンの IBM Data Server Driver for JDBC and SQLJ を使用する予定であれば、`db2jcc.jar` を CLASSPATH に組み込みます。

**JDBC 4.0 以降の機能と、JDBC 3.0 以前の機能と**を組み込んでいるバージョンの IBM Data Server Driver for JDBC and SQLJ を使用する予定であれば、`db2jcc4.jar` を CLASSPATH に組み込みます。

#### **sqlj.zip**

`sqlj.zip` は SQLJ プログラムの準備および実行用です。

- CLASSPATH を変更して、IBM Data Server Driver for JDBC and SQLJ のクラス・ファイルを組み込みます。

**重要:** このステップは `db2jcc.jar` ファイルに対してのみ自動的に実行されます。`db2jcc4.jar` ファイルを使用している場合は、CLASSPATH を手動で変更する必要があります。CLASSPATH で `db2jcc.jar` を `db2jcc4.jar` に変更します。

**重要:** CLASSPATH に db2jcc.jar または db2jcc4.jar を組み込みます。両方のファイルを組み込むことはしないでください。

- IBM Data Server Driver for JDBC and SQLJ のライセンス・ファイルをインストールし、CLASSPATH を変更してそれらを組み込むようにします。

これらのファイルは、Windows システムの場合は sqllib¥java ディレクトリー、UNIX または Linux システムの場合は sqllib/java ディレクトリーに置かれます。これらのファイル名は以下のとおりです。

表 2. IBM Data Server Driver for JDBC and SQLJ のライセンス・ファイル

ライセンス・ファイル	ライセンス・ファイルが接続先として許可するサーバー	ライセンス・ファイルを含む製品
db2jcc_license_cisuz.jar	DB2 for z/OS® DB2 for i5/OS	すべての DB2 Connect 製品

ライセンス・ファイルは、IBM Data Server Driver for JDBC and SQLJ バージョン 3.50 以降からの、DB2 Database for Linux, UNIX, and Windows、Cloudscape、または IBM Informix® Dynamic Server (IDS) データベースへの接続には不要です。

- IBM Data Server Driver for JDBC and SQLJ Type 2 接続のサポート用の IBM Data Server Driver for JDBC and SQLJ ネイティブ・ライブラリーをインストールします。

これらのファイルは、Windows システムの場合は sqllib¥bin ディレクトリー、UNIX または Linux システムの場合は sqllib/lib ディレクトリーに置かれます。

これらのファイル名は以下のとおりです。

**libdb2jct2.so**

AIX®, IPF 版 HP-UX、Linux、および Solaris の場合

**libdb2jct2.sl**

PA-RISC 版 HP-UX の場合

**db2jct2.dll**

Windows の場合

2. いずれかのデフォルトが不適切である場合は、ドライバー全体にわたる構成プロパティーをカスタマイズします。詳しくは、18 ページの『IBM Data Server Driver for JDBC and SQLJ の構成プロパティーのカスタマイズ』を参照してください。
3. TCP/IP を構成します。

サーバーは、以下の場合の TCP/IP 通信用に構成する必要があります。

- IBM Data Server Driver for JDBC and SQLJ Type 4 接続を使用する JDBC または SQLJ アプリケーション。
- IBM Data Server Driver for JDBC and SQLJ Type 2 接続を使用し、接続 URL に *server* および *port* を指定する JDBC または SQLJ アプリケーション。

TCP/IP リスナーが実行されていることを確認します。 TCP/IP リスナーを活動化するには、以下のようにします。

- a. 以下のようにして環境変数 DB2COMM を TCPIP に設定します。

```
db2set DB2COMM=TCPIP
```

- b. サービス・ファイルで指定されるように、データベース・マネージャーの構成ファイルを TCP/IP サービス名で更新します。

```
db2 update dbm cfg using SVCENAME TCP/IP-service-name
```

アプレットおよび SQLJ プログラムに使用するポート番号は、データベース・マネージャー構成ファイルで使用される TCP/IP SVCENAME 番号と同じものにする必要があります。

- c. サービス名の設定を有効にするには、db2stop および db2start コマンドを実行します。

4. Java ストアード・プロシージャまたはユーザー定義関数の実行を予定している DB2 Database for Linux, UNIX, and Windows サーバー上では、DB2\_USE\_DB2JCCT2\_JROUTINE 環境変数が設定されていないか、またはそのデータベース・サーバー上でデフォルト値の YES、yes、ON、on、TRUE、true、または 1 に設定されていることを確認してください。この設定は、Java ストアード・プロシージャが IBM Data Server Driver for JDBC and SQLJ の下で実行することを示しています。

ストアード・プロシージャを Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバーの下で実行する必要がある場合、DB2\_USE\_DB2JCCT2\_JROUTINE 環境変数を OFF に設定します。

5. Java ストアード・プロシージャまたはユーザー定義関数の実行を予定している DB2 Database for Linux, UNIX, and Windows サーバー上では、データベース・マネージャー構成を更新して、SDK for Java が置かれているパスを組み込みます。

これは、サーバーのコマンド行での場合と類似のコマンドを入力することで実行できます。

- UNIX または Linux 上のデータベース・システムの場合:

```
db2 update dbm cfg using JDK_PATH /home/db2inst/jdk15
```

/home/db2inst/jdk15 は、SDK for Java のインストール先のパスです。

- Windows 上のデータベース・システムの場合:

```
db2 update dbm cfg using JDK_PATH c:¥Program Files¥jdk15
```

c:¥Program Files¥jdk15 は、SDK for Java のインストール先のパスです。

DB2 データベース・マネージャー構成の JDK\_PATH フィールド用の正しい値を確認するには、以下のコマンドをデータベース・サーバー上に入力します。

```
db2 get dbm cfg
```

出力をファイルにリダイレクトすれば、一層容易に表示できます。JDK\_PATH フィールドは、出力の先頭近くに表示されます。

6. DB2 Database for Linux, UNIX, and Windows サーバー上にある SQL プロシージャを Java プログラムから呼び出す予定であり、データベース・サーバーのテリトリー・コードと関連付けられた日付と時刻の形式が USA 形式ではない場合、以下のアクションを実行します。

a. 以下のようにして DB2\_SQLROUTINE\_PREPOPTS レジストリー変数をデータベース・サーバー上で設定し、デフォルトの日時形式が ISO であることを示します。

```
db2set DB2_SQLROUTINE_PREPOPTS="DATETIME ISO"
```

b. Java プログラムから呼び出す予定の既存の SQL プロシージャを再定義します。

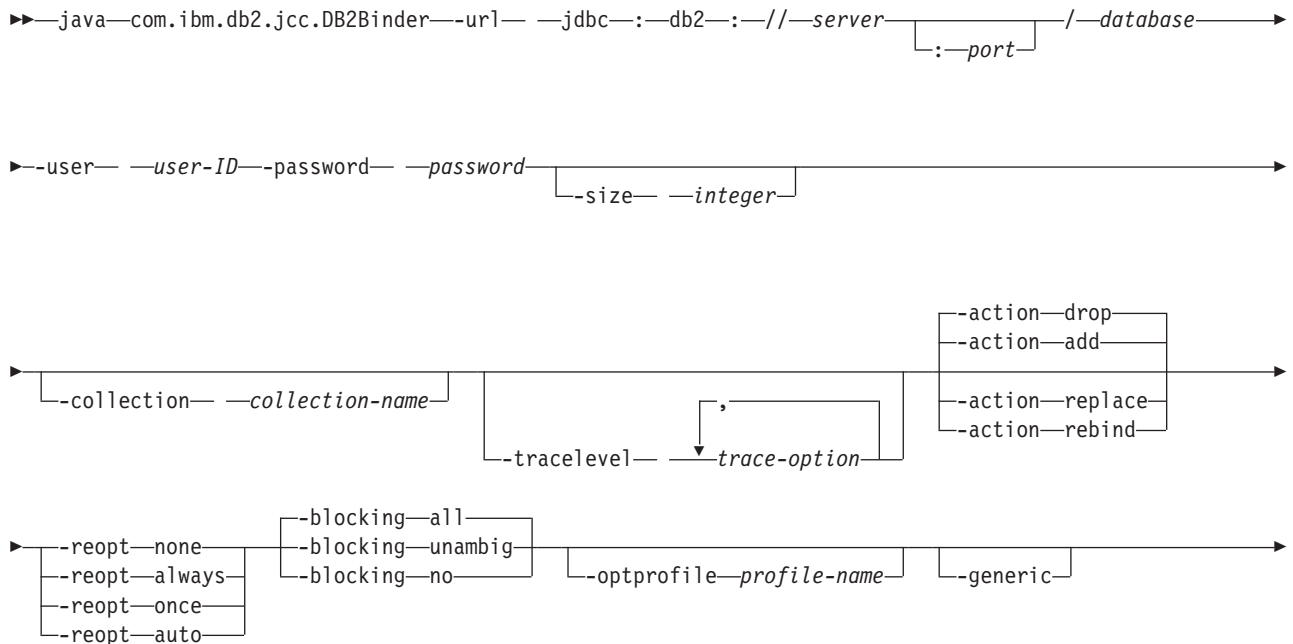
これらのステップは、呼び出し側のアプリケーションが日付と時刻の値を正しく受け取るために必要です。

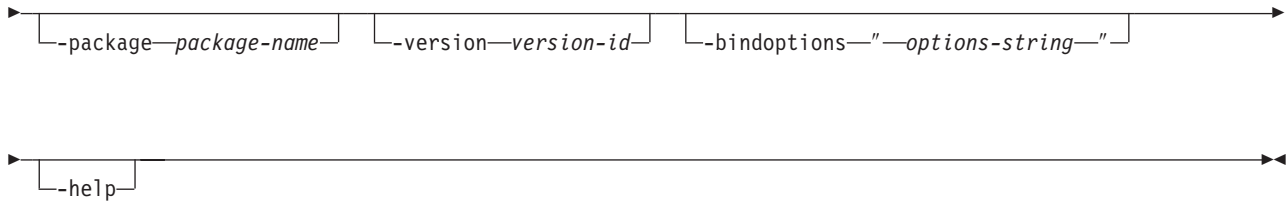
7. DB2 for z/OS データベース・サーバーに Java アプリケーションでアクセスする予定の場合は、19 ページの『Java プログラムから DB2 for z/OS サーバーにアクセスするための特別なセットアップ』にある手順に従ってください。

## DB2Binder ユーティリティー

DB2Binder ユーティリティーは、IBM Data Server Driver for JDBC and SQLJ によってデータベース・サーバーで使用される DB2 パッケージをバインドし、パッケージに対する EXECUTE 権限を PUBLIC に与えます。オプションで、DB2Binder ユーティリティーは IBM Data Server Driver for JDBC and SQLJ の一部ではない DB2 パッケージを再バインドすることができます。

### DB2Binder の構文





## DB2Binder オプションの説明

### **-url**

IBM Data Server Driver for JDBC and SQLJ パッケージをバインドするデータ・ソースを指定します。 `-url` 値の可変部は、以下のとおりです。

#### **server**

データベース・サーバーが存在するオペレーティング・システムのドメイン名または IP アドレス。

#### **port**

データベース・サーバーに割り当てられている TCP/IP サーバーのポート番号。デフォルトは 446 です。

#### **database**

プロファイルがカスタマイズされるデータベース・サーバーの名前。

接続を DB2 for z/OS サーバーに対して行う場合、*database* は、インストール中に定義される DB2 ロケーション名です。この値の中の文字はすべて大文字でなければなりません。サーバーで以下の SQL ステートメントを実行することにより、ロケーション名を判別できます。

```
SELECT CURRENT SERVER FROM SYSIBM.SYSDUMMY1;
```

接続を DB2 Database for Linux, UNIX, and Windows サーバーに対して行う場合、*database* は、インストール中に定義されるデータベース名です。

接続を IBM Cloudscape サーバーに対して行う場合、*database* は、データベースを含むファイルの完全修飾名です。この名前は、二重引用符 (") で囲まなければなりません。以下に例を示します。

```
"c:/databases/testdb"
```

### **-url**

IBM Data Server Driver for JDBC and SQLJ パッケージをバインドするデータ・ソースを指定します。 `-url` 値の可変部は、以下のとおりです。

#### **server**

データベース・サーバーが存在するオペレーティング・システムのドメイン名または IP アドレス。

#### **port**

データベース・サーバーに割り当てられている TCP/IP サーバーのポート番号。デフォルトは 446 です。

### database

SYSIBM.LOCATIONS カタログ表で定義されている、データベース・サーバーのロケーション名。

### -user

パッケージをバインドするユーザー ID を指定します。このユーザーは、パッケージに対する BIND 権限を持っている必要があります。

### -action

IBM Data Server Driver for JDBC and SQLJ パッケージを置き換えられるかどうかを指定します。

**add** パッケージを作成できるのは、そのパッケージがまだ存在しない場合のみであることを指示します。add はデフォルトです。

### replace

同名のパッケージがすでに存在する場合でも、パッケージを作成できることを指示します。新規のパッケージが、古いパッケージに取って代わります。

**rebind** 既存のパッケージを再バインドすることを示します。このオプションは IBM Data Server Driver for JDBC and SQLJ パッケージには適用されません。-action rebind を指定する場合、-generic も指定する必要があります。

**drop** 一部またはすべての IBM Data Server Driver for JDBC and SQLJ パッケージをドロップすることを示します。パッケージの数は、-size パラメーターに応じて異なります。

-action drop は、ターゲット・データベース・サーバーが DB2 for z/OS の場合にのみ適用されます。

### -size

同時にオープンできる Statement、PreparedStatement、または CallableStatement オブジェクトの数、あるいはドロップされる IBM Data Server Driver for JDBC and SQLJ パッケージの数を制御します。-size パラメーターの意味は、-action パラメーターに以下のように依存します。

- -action の値が add または replace であれば、-size の値は IBM Data Server Driver for JDBC and SQLJ がバインドする DB2 パッケージの数の計算に使用される整数です。-size の値が *integer* であれば、パッケージの合計数は次のようになります。

```
number-of-isolation-levels*  
number-of-holdability-values*  
integer+  
number-of-packages-for-static-SQL  
= 4*2*integer+1
```

-action add または -action replace のデフォルトの -size 値は 3 です。

たいていの場合、デフォルトの 3 が適切です。アプリケーションが SQLException を -805 SQLCODE でスローする場合、アプリケーションが未使用のすべてのリソースをクローズしていることを確認してください。クローズしている場合は、-size 値を増やします。

-action の値が replace で、-size の値が既存のものより少ない数のパッケージという結果になる場合は、ドロップされるパッケージはありません。

- -action の値が drop の場合、-size の値はドロップされるパッケージの数です。-size が指定されない場合、すべての IBM Data Server Driver for JDBC and SQLJ パッケージがドロップされます。
- -action の値が rebind の場合、-size は無視されます。

#### **-collection**

IBM Data Server Driver for JDBC and SQLJ またはユーザー・パッケージのコレクション ID を指定します。デフォルトは NULLID です。DB2Binder はこの値を大文字に変換します。

com.ibm.db2.jcc.DB2Binder を複数回実行して、-collection に毎回異なる値を指定すれば、単一のデータベース・サーバー上で IBM Data Server Driver for JDBC and SQLJ パッケージのインスタンスを複数作成することができます。実行時に、currentPackageSet プロパティを -collection 値に一致する値に設定することによって、IBM Data Server Driver for JDBC and SQLJ のコピーを選択します。

#### **-tracelevel**

DB2Binder の実行時に何をトレースするかを指定します。

#### **-reopt**

DB2 for z/OS データベース・サーバーが実行時にアクセス・パスを判別するかどうかを指定します。このオプションは、DB2 for z/OS データベース・サーバーへの接続の場合にのみ有効です。指定しない場合、このオプションはデータベース・サーバーには送られません。その場合、データベース・サーバーは再最適化の動作を決定します。

**none** 実行時にアクセス・パスを判別しないよう指定します。

**always** ステートメントを実行するたびにアクセス・パスを判別することを指定します。

**once** 実行時に DB2 が一度だけ動的ステートメントのアクセス・パスを判別してキャッシュに入れることを指定します。DB2 は、準備されたステートメントが無効にされるか、動的ステートメントのキャッシュから除去されて準備が再度必要になるまで、このアクセス・パスを使用します。

**auto** データベース・サーバーによってアクセス・パスが自動的に決定されることを指定します。

#### **-blocking**

カーソルの行ブロッキングのタイプを指定します。

**ALL** FOR READ ONLY 節で指定したカーソル、または FOR UPDATE として指定していないカーソルに対して、ブロッキングが行われます。

#### **UNAMBIG**

FOR READ ONLY 節で指定したカーソルに対して、ブロッキングが行われます。



FOR READ ONLY または FOR UPDATE 節で宣言されておらず、未確定ではなく読み取り専用 のカーソルはブロックされます。未確定カーソルはブロックされません。

**NO** どのカーソルにもブロッキングは行われません。

読み取り専用カーソルおよび未確定カーソルの定義については、「DECLARE CURSOR」を参照してください。

#### **-optprofile**

パッケージのデータ変更ステートメントの最適化に使用される最適化プロファイルを指定します。このプロファイルは、ターゲット・サーバー上に存在しなければならない XML ファイルです。-optprofile が指定されておらず、CURRENT OPTIMIZATION PROFILE 特殊レジスターが設定されている場合、CURRENT OPTIMIZATION PROFILE の値が使用されます。-optprofile が指定されておらず、CURRENT OPTIMIZATION PROFILE が設定されていない場合、最適化プロファイルは使用されません。

-optprofile は、DB2 Database for Linux, UNIX, and Windows データベース・サーバーへの接続の場合にのみ有効です。

#### **-generic**

DB2Binder が IBM Data Server Driver for JDBC and SQLJ パッケージの代わりにユーザー・パッケージを再バインドすることを指定します。-generic を指定する場合、-action rebind および -package も指定しなければなりません。

#### **-package**

再バインドされるパッケージの名前を指定します。このオプションはユーザー・パッケージだけに適用されます。-package を指定する場合、-action rebind および -generic も指定しなければなりません。

#### **-version**

再バインドされるパッケージのバージョン ID を指定します。-version を指定する場合、-action rebind、-package、および -generic も指定しなければなりません。

#### **-bindoptions**

引用符で区切られるストリングを指定します。このストリングの内容は、ユーザー・パッケージの再バインド用のオプションを表す、1 つ以上のパラメーターと値の組です。

指定可能なパラメーターと値は以下のとおりです。

##### **bindObjectExistenceRequired**

すべてのオブジェクトまたは必要とされる特権が再バインド時に存在しない場合、データベース・サーバーはエラーを発行して、パッケージを再バインドしないかどうかを指定します。可能な値は以下のとおりです。

**true** このオプションは SQLERROR(NOPACKAGE) バインド・オプションに対応します。

**false** このオプションは SQLERROR(CONTINUE) バインド・オプションに対応します。

### degreeIOParallelism

パフォーマンスを最大化するために、並列処理を使用して静的照会の実行を試行するかどうかを指定します。可能な値は以下のとおりです。

- 1 並列処理はありません。  
このオプションは DEGREE(1) バインド・オプションに対応します。
- 1 並列処理を許可します。  
このオプションは DEGREE(ANY) バインド・オプションに対応します。

### packageAuthorizationRules

以下の動的 SQL 属性に実行時に適用される値を決定します。

- 許可の検査に使用される許可 ID
- 非修飾オブジェクトに使用される修飾子
- データベース・サーバーが動的 SQL ステートメントを構文解析して意味的に検証するために使用する、アプリケーション・プログラミング・オプションのソース
- 動的 SQL ステートメントが GRANT、REVOKE、ALTER、CREATE、DROP、および RENAME ステートメントを組み込めるかどうか

可能な値は以下のとおりです。

- 0 実行の動作を使用します。これはデフォルトです。  
このオプションは DYNAMICRULES(RUN) バインド・オプションに対応します。
- 1 バインドの動作を使用します。  
このオプションは DYNAMICRULES(BIND) バインド・オプションに対応します。
- 2 パッケージがストアド・プロシージャまたはユーザー定義関数パッケージとして、またはその下で実行する場合、データベース・サーバーは呼び出しの動作を使用して動的 SQL ステートメントを処理します。そうでない場合、データベース・サーバーは実行の動作を使用して動的 SQL ステートメントを処理します。  
このオプションは DYNAMICRULES(INVOKERUN) バインド・オプションに対応します。
- 3 パッケージがストアド・プロシージャまたはユーザー定義関数パッケージとして、またはその下で実行する場合、データベース・サーバーは呼び出しの動作を使用して動的 SQL ステートメントを処理します。そうでない場合、データベース・サーバーはバインドの動作を使用して動的 SQL ステートメントを処理します。  
このオプションは DYNAMICRULES(INVOKEBIND) バインド・オプションに対応します。

- 4 パッケージがストアード・プロシージャまたはユーザー定義関数パッケージとして、またはその下で実行する場合、データベース・サーバーは定義の動作を使用して動的 SQL ステートメントを処理します。そうでない場合、データベース・サーバーは実行の動作を使用して動的 SQL ステートメントを処理します。

このオプションは DYNAMICRULES(DEFINERUN) バインド・オプションに対応します。

- 5 パッケージがストアード・プロシージャまたはユーザー定義関数パッケージとして、またはその下で実行する場合、データベース・サーバーは定義の動作を使用して動的 SQL ステートメントを処理します。そうでない場合、データベース・サーバーはバインドの動作を使用して動的 SQL ステートメントを処理します。

このオプションは DYNAMICRULES(DEFINEBIND) バインド・オプションに対応します。

#### **packageOwnerIdentifier**

パッケージの所有者の許可 ID を指定します。

このオプションは OWNER バインド・オプションに対応します。

#### **isolationLevel**

他の実行中のアプリケーションの影響からアプリケーションをどの程度隔離するかを指定します。可能な値は以下のとおりです。

- 1 非コミット読み取り

このオプションは ISOLATION(UR) バインド・オプションに対応します。

- 2 カーソル固定

このオプションは ISOLATION(CS) バインド・オプションに対応します。

- 3 読み取り固定

このオプションは ISOLATION(RS) バインド・オプションに対応します。

- 4 反復可能読み取り

このオプションは ISOLATION(RR) バインド・オプションに対応します。

#### **releasePackageResourcesAtCommit**

各コミット・ポイントでプログラムが使用するリソースをいつ解放するかを指定します。可能な値は以下のとおりです。

**true** このオプションは RELEASE(COMMIT) バインド・オプションに対応します。

**false** このオプションは RELEASE(DEALLOCATE) バインド・オプションに対応します。

-bindoptions を指定する場合、-generic も指定する必要があります。

## DB2Binder 戻りコード

DB2Binder は以下の戻りコードの 1 つを戻します。

表 3. DB2Binder 戻りコード

戻りコード	意味
0	成功した実行。
-100	バインド・オプションが指定されていません。
-101	-url 値が指定されていません。
-102	-user 値が指定されていません。
-103	-password 値が指定されていません。
-104	-action 値が指定されていません。
-105	-blocking 値が指定されていません。
-106	-collection 値が指定されていません。
-107	-dbprotocol 値が指定されていません。
-108	-keepdynamic 値が指定されていません。
-109	-owner 値が指定されていません。
-110	-reopt 値が指定されていません。
-111	-size 値が指定されていません。
-112	-tracelevel 値が指定されていません。
-113	-optprofile 値が指定されていません。
-200	有効なバインド・オプションが指定されていません。
-201	-url 値が無効です。
-204	-action 値が無効です。
-205	-blocking 値が無効です。
-206	-collection 値が無効です。
-207	-dbprotocol 値が無効です。
-208	-keepdynamic 値が無効です。
-210	-reopt 値が無効です。
-211	-size 値が無効です。
-212	-tracelevel 値が無効です。
-307	-dbprotocol 値はターゲット・データベース・サーバーによりサポートされていません。
-308	-keepdynamic 値はターゲット・データベース・サーバーによりサポートされていません。
-310	-reopt 値はターゲット・データベース・サーバーによりサポートされていません。
-313	-optprofile 値はターゲット・データベース・サーバーによりサポートされていません。
-401	Binder クラスが検出されませんでした。
-402	データベース・サーバーへの接続が失敗しました。

表 3. DB2Binder 戻りコード (続き)

戻りコード	意味
-403	データベース・サーバーの DatabaseMetaData 検索が失敗しました。
-501	クラスターで使用可能なパッケージはこれ以上ありません。
-502	既存のパッケージは無効です。
-503	バインド処理がエラーを戻しました。
-999	文書化されていないバインド・オプションの処理中にエラーが発生しました。

## DB2LobTableCreator ユーティリティ

DB2LobTableCreator ユーティリティは、DB2 for z/OS データベース・サーバー上に表を作成します。それらの表は、DBCLOB または CLOB 列のデータにアクセスするために LOB ロケーターを使用する JDBC または SQLJ アプリケーションにより必要とされます。

### DB2LobTableCreator 構文

```

▶▶—java—com.ibm.db2.jcc.DB2LobTableCreator—-url—jdbc:db2:—//server—/—database—
                                     [—:port—]
▶—-user—user-ID—-password—password—
                                     [—-help—]

```

### DB2LobTableCreator オプションの説明

#### -url

DB2LobTableCreator が実行するデータ・ソースを指定します。-url 値の可変部は、以下のとおりです。

#### jdbc:db2:

接続先が DB2 ファミリーのサーバーであることを示します。

#### server

データベース・サーバーのドメイン名または IP アドレス。

#### port

データベース・サーバーに割り当てられている TCP/IP サーバーのポート番号。これは 0 から 65535 の間の整数です。デフォルトは 446 です。

#### database

データベース・サーバーの名前。

database は、インストール中に定義される DB2 ロケーション名です。この値の中の文字はすべて大文字でなければなりません。サーバーで以下の SQL ステートメントを実行することにより、ロケーション名を判別できます。

```
SELECT CURRENT SERVER FROM SYSIBM.SYSDUMMY1;
```

#### -user

DB2LobTableCreator を実行するユーザー ID を指定します。このユーザーは DSNATPDB データベース内に表を作成する権限を持っている必要があります。

### **-password**

ユーザー ID のパスワードを指定します。

### **-help**

DB2LobTableCreator ユーティリティーがサポートする各オプションを記述することを指定します。 **-help** と共に他のオプションが指定されても、それらは無視されます。

---

## **IBM Data Server Driver for JDBC and SQLJ の構成プロパティースタマイズ**

IBM Data Server Driver for JDBC and SQLJ 構成プロパティースタマイズを使用して、ドライバー全体にわたる有効範囲を持つプロパティースタマイズを設定することができます。これらの設定は、アプリケーションおよび DataSource インスタンス全体に適用されます。アプリケーションのソース・コードまたは DataSource 特性を変更せずに、この設定を変更することができます。

それぞれの IBM Data Server Driver for JDBC and SQLJ 構成プロパティースタマイズの設定は以下の形式になります。

*property=value*

*property* は以下に示す 1 つ以上の形式になります。

- *db2.jcc.override.property-name*
- *db2.jcc.property-name*
- *db2.jcc.default.property-name*

構成プロパティースタマイズが *db2.jcc.override* で始まる場合、それはすべての接続に適用可能であり、同じ *property-name* を持つすべての Connection または DataSource プロパティースタマイズをオーバーライドします。構成プロパティースタマイズが *db2.jcc* または *db2.jcc.default* で始まる場合、構成プロパティースタマイズの値はデフォルトになります。Connection または DataSource プロパティースタマイズの設定値はその値をオーバーライドします。

構成プロパティースタマイズは以下の方法で設定できます。

- 構成プロパティースタマイズを Java システム・プロパティースタマイズとして設定します。これらの設定はその他の設定をオーバーライドします。

スタンドアロン Java アプリケーションの場合、java コマンドを実行する際に構成プロパティースタマイズごとに **-Dproperty=value** を指定することによって、構成プロパティースタマイズを Java システム・プロパティースタマイズとして設定できます。

- *db2.jcc.propertiesFile* Java システム・プロパティースタマイズで指定した名前を持つリソースに構成プロパティースタマイズを設定します。例えば、*db2.jcc.propertiesFile* 値に絶対パス名を指定することができます。

スタンドアロン Java アプリケーションの場合、java コマンドを実行する際に **-Ddb2.jcc.propertiesFile=path** オプションを指定することによって、構成プロパティースタマイズを設定できます。

- *DB2JccConfiguration.properties* という名前のリソースに構成プロパティースタマイズを設定します。標準 Java リソース検索は、*DB2JccConfiguration.properties* を検索するのに

使用されます。IBM Data Server Driver for JDBC and SQLJ がこのリソースを検索するのは、db2.jcc.propertiesFile Java システム・プロパティを設定していない場合のみです。

DB2JccConfiguration.properties はスタンドアロン・ファイルにするか、あるいは JAR ファイルに組み込むことができます。

DB2JccConfiguration.properties ファイルが ISO 8859-1 (Latin-1) コード化スキームであるか、あるいは一部の Unicode エンコード (¥udddd) 文字を含む Latin-1 コード化スキームである場合、IBM Data Server Driver for JDBC and SQLJ がそのファイルを使用する前に文字変換を行う必要はありません。

DB2JccConfiguration.properties ファイルがその他のコード化スキームである場合、Java native2ascii コンバーターを使用して、ファイルの内容を Latin-1 または Unicode エンコード文字に変換する必要があります。

DB2JccConfiguration.properties がスタンドアロン・ファイルである場合、DB2JccConfiguration.properties のパスは CLASSPATH 連結内に存在する必要があります。

DB2JccConfiguration.properties が JAR ファイルにある場合、その JAR ファイルは CLASSPATH 連結内に存在する必要があります。

---

## Java プログラムから DB2 for z/OS サーバーにアクセスするための特別なセットアップ

DB2 for z/OS データベース・サーバーにアクセスする JDBC アプリケーションまたは SQLJ アプリケーションを作成する計画がある場合は、IBM Data Server Driver for JDBC and SQLJ のインストール・プロセスで追加の手順が必要になります。

DB2 for z/OS サーバーへの接続を可能にするには、以下の手順を実行します。

1. DB2 for z/OS バージョン 7 またはバージョン 8 のデータベース・サーバーに接続する計画がある場合は、それらのデータベース・サーバーに以下の PTF をインストールします。

表 4. DB2 for z/OS ストアード・プロシージャのための PTF

DB2 for z/OS	PTF 番号または APAR 番号
バージョン 7	UQ72083、UQ93889、UK21848
バージョン 8	UQ93890、UK21849
バージョン 9	PK44166

2. com.ibm.db2.jcc.DB2Binder ユーティリティを実行して、サーバーで IBM Data Server Driver for JDBC and SQLJ が使用する DB2 パッケージをバインドします。詳しくは、9 ページの『DB2Binder ユーティリティ』を参照してください。
3. DB2 for z/OS データベース・サーバーでジョブ DSNTIJMS をカスタマイズして実行します。

DSNTIJMS は、データ・セット *prefix.SDSNSAMP* にあります。このジョブは、以下の機能を実行します。



- DatabaseMetaData のメソッド、トレース、エラー・メッセージ・フォーマットをサポートするために、以下のストアード・プロシージャを作成します。
  - SQLCOLPRIVILEGES
  - SQLCOLUMNS
  - SQLFOREIGNKEYS
  - SQLFUNCTIONS
  - SQLFUNCTIONCOLUMNS
  - SQLGETTYPEINFO
  - SQLPRIMARYKEYS
  - SQLPROCEDURECOLS
  - SQLPROCEDURES
  - SQLSPECIALCOLUMNS
  - SQLSTATISTICS
  - SQLTABLEPRIVILEGES
  - SQLTABLES
  - SQLUDTS
  - SQLCAMESSAGE
- CLOB 列または DBCLOB 列にデータを効率的に格納し、CLOB または DBCLOB の検索で LOB ロケータを使用するために、以下の表を作成します。
  - SYSIBM.SYSDUMMYU
  - SYSIBM.SYSDUMMYA
  - SYSIBM.SYSDUMMYE

これらの表を作成する別の方法として、それぞれの DB2 for z/OS サーバーに対してクライアントで com.ibm.db2.jcc.DB2LobTableCreator ユーティリティを実行することも可能です。詳しくは、17 ページの『DB2LobTableCreator ユーティリティ』を参照してください。

4. OS/390<sup>®</sup> サーバーと z/OS サーバーで Unicode サポートを使用可能にします。

SQLJ プログラムまたは JDBC プログラムで IBM Data Server Driver for JDBC and SQLJ Type 4 接続を使用して DB2 for z/OS バージョン 7 サーバーに接続する場合は、OS/390 または z/OS のオペレーティング・システムで Unicode UTF-8 コード化スキームをサポートする必要があります。そのサポートのためには、OS/390 バージョン 2 リリース 9 APAR OW44581、あるいは OS/390 または z/OS のそれ以降のリリースと、OS/390 R8/R9/R10 Unicode サポートが必要です。追加情報については、情報 APAR II13048 と II13049 を参照してください。

5. IBM Data Server Driver for JDBC and SQLJ Type 4 接続を使用して DB2 for z/OS バージョン 7 サーバーに分散トランザクションをインプリメントする計画がある場合は、それぞれの DB2 for z/OS バージョン 7 サーバーに対して DB2T4XAIndoubtUtil ユーティリティを 1 回ずつ実行します。詳しくは、21 ページの『DB2 UDB (OS/390 および z/OS 版) バージョン 7 サーバーを使用する分散トランザクション用の DB2T4XAIndoubtUtil』を参照してください。

## DB2 UDB (OS/390 および z/OS 版) バージョン 7 サーバーを使用する分散トランザクション用の DB2T4XAIndoubtUtil

IBM Data Server Driver for JDBC and SQLJ Type 4 接続を使用して、DB2 UDB (OS/390 および z/OS 版) バージョン 7 サーバーを組み込んだ分散トランザクションをインプリメントすることを計画している場合、それらのサーバーに対して DB2T4XAIndoubtUtil ユーティリティを実行する必要があります。

DB2T4XAIndoubtUtil により、XA 仕様をインプリメントする分散トランザクションの組み込みサポートを持たないバージョン 7 サーバーは、サポートをエミュレートすることができます。

DB2T4XAIndoubtUtil は、次のタスクの一方または両方を実行します。

- SYSIBM.INDOUBT という名前の表および関連する索引を作成します。
- T4XAIN01、T4XAIN02、T4XAIN03、および T4XAIN04 という名前の DB2 パッケージをバインドします。

パッケージ T4XAIN01、T4XAIN02、T4XAIN03、および T4XAIN04 の作成およびドロップを、DB2T4XAIndoubtUtil を実行するだけで行います。

SYSTEM.INDOUBT とその索引は手動で作成およびドロップできますが、このユーティリティを使用することをお勧めします。これらのオブジェクトを手動で作成する方法についての説明は、『DB2T4XAIndoubtUtil の使用上の注意』を参照してください。

### DB2T4XAIndoubtUtil の許可

DB2T4XAIndoubtUtil ユーティリティを実行して、SYSTEM.INDOUBT の作成とパッケージ T4XAIN01、T4XAIN02、T4XAIN03、および T4XAIN04 のバインドを行うには、SYSADM 権限が必要です。

DB2T4XAIndoubtUtil を、パッケージ T4XAIN01、T4XAIN02、T4XAIN03、および T4XAIN04 をバインドするためだけに実行するには、パッケージに対する BIND 権限が必要です。

### DB2T4XAIndoubtUtil の構文

```
▶▶—java—com.ibm.db2.jcc.DB2T4XAIndoubtUtil—url—jdbc:db2:—//server—[:port]—/—database—▶▶
▶—user—user-ID—password—password—[-owner—owner-ID]—[-help]—[-delete]—▶▶
▶[-priqty—integer]—[-secqty—integer]—[-bindonly]—[-showSQL]—▶▶
▶[-jdbcCollection—NULLID]—▶▶
▶[-jdbcCollection—collection-ID]—▶▶
```

## DB2T4XAIndoubtUtil のパラメーターの説明

### -url

DB2T4XAIndoubtUtil が実行するデータ・ソースを指定します。-url 値の可変部は、以下のとおりです。

### **jdbc:db2:**

接続先が DB2 ファミリーのサーバーであることを示します。

### **server**

データベース・サーバーのドメイン名または IP アドレス。

### **port**

データベース・サーバーに割り当てられている TCP/IP サーバーのポート番号。これは 0 から 65535 の間の整数です。デフォルトは 446 です。

### **database**

データベース・サーバーの名前。

*database* は、インストール中に定義される DB2 ロケーション名です。この値の中の文字はすべて大文字でなければなりません。サーバーで以下の SQL ステートメントを実行することにより、ロケーション名を判別できます。

```
SELECT CURRENT SERVER FROM SYSIBM.SYSDUMMY1;
```

### -user

DB2T4XAIndoubtUtil を実行するユーザー ID を指定します。このユーザーは、SYSADM 権限を持っているか、または SYSADM 権限がある 2 次許可 ID に対応する RACF<sup>®</sup> グループのメンバーでなければなりません。

### -password

ユーザー ID のパスワードを指定します。

### -owner

SYSADM 権限を持つ 2 次許可 ID を指定します。-user パラメーター値が SYSADM 権限を持たない場合は、-owner パラメーターを使用します。-user パラメーター値は、名前が *owner-ID* の RACF グループのメンバーでなければなりません。

-owner パラメーターの指定時に、DB2T4XAIndoubtUtil は *owner-ID* を以下のものとして使用します。

- SYSIBM.INDOUBT 表を作成するための許可 ID。
- T4XAIN01、T4XAIN02、T4XAIN03、および T4XAIN04 パッケージの所有者の許可 ID。それらのパッケージ内の SQL ステートメントは、*owner-ID* の権限を使用して実行されます。

### -help

DB2T4XAIndoubtUtil ユーティリティーがサポートする各オプションを記述することを指定します。-help と共に他のオプションが指定されても、それらは無視されます。

### -delete

DB2T4XAIndoubtUtil の以前の実行時に作成されたオブジェクトを、DB2T4XAIndoubtUtil ユーティリティーが削除することを指定します。

### **-priqty**

SYSIBM.INDOUBT 表を含む表スペースの 1 次スペース割り振りを、K バイト (KB) 単位で指定します。-priqty のデフォルト値は 1000 です。

**重要:** ページ・サイズで除算した、SYSIBM.INDOUBT が置かれる表スペースの -priqty 値は、一定の時間に許可される未確定トランザクションの最大数よりも大きくなければなりません。例えば、4 KB のページ・サイズの場合、デフォルトの -priqty 値である 1000 では、およそ 250 の並行未確定トランザクションが可能です。

### **-secqty**

SYSIBM.INDOUBT 表を含む表スペースの 2 次スペース割り振りを、K バイト (KB) 単位で指定します。-secqty のデフォルト値は 0 です。

**推奨事項:** -secqty 値には必ずデフォルト値 0 を使用し、-priqty 値には並行未確定トランザクションの最大数に対応する十分な大きさの値を指定してください。

### **-bindonly**

DB2T4XAIndoubtUtil ユーティリティーで T4XAIN01、T4XAIN02、T4XAIN03、および T4XAIN04 パッケージをバインドし、そのパッケージを実行する許可を PUBLIC に与えるが、SYSIBM.INDOUBT 表は作成しないように指定します。

### **-showSQL**

DB2T4XAIndoubtUtil ユーティリティーが、実行する SQL ステートメントを表示することを指定します。

### **-jdbcCollection *collection-name* NULLID**

IBM Data Server Driver for JDBC and SQLJ パッケージが DB2Binder ユーティリティーでバインドされるときに使用された -collection パラメーターの値を指定します。明示的または暗黙に指定された -collection パラメーターの値が NULLID でなかった場合は、-jdbcCollection パラメーターを指定しなければなりません。

デフォルトは -jdbcCollection NULLID です。

## **DB2T4XAIndoubtUtil の使用上の注意**

SYSTEM.INDOUBT 表およびその索引を手動で作成するには、以下の SQL ステートメントを使用します。

```
CREATE TABLESPACE INDBTTS
  USING STOGROUP
  LOCKSIZE ROW
  BUFFERPOOL BP0
  SEGSIZE 32
  CCSID EBCDIC;

CREATE TABLE SYSIBM.INDOUBT(indbtXid VARCHAR(140) FOR BIT DATA NOT NULL,
                             uowId VARCHAR(25) FOR BIT DATA NOT NULL,
                             pSyncLog VARCHAR(150) FOR BIT DATA,
                             cSyncLog VARCHAR(150) FOR BIT DATA)
  IN INDBTTS;

CREATE UNIQUE INDEX INDBTIDX ON SYSIBM.INDOUBT(indbtXid, uowId);
```

## DB2T4XAIndoubtUtil の例

DB2T4XAIndoubtUtil を実行して、IP アドレス mvs1、ポート番号 446、および DB2 ロケーション名 SJCEC1 を持つ DB2 for OS/390 and z/OS バージョン 7 サブシステムが、XA 分散トランザクションに参加できるようにします。

```
java com.ibm.db2.jcc.DB2T4XAIndoubtUtil -url jdbc:db2://mvs1:446/SJCEC1 ¥  
-user SYSADM -password mypass
```

---

## HP-UX 環境で Java ルーチンを実行するための特別なセットアップ

PA-RISC プロセッサ上の HP-UX オペレーティング・システムの場合、Java ストアード・プロシージャおよびユーザー定義関数を実行するための特別な前提条件があります。

『IBM Data Server Driver for JDBC and SQLJ のインストール』にある前提条件に加えて、以下の前提条件のステップを実行する必要があります。

1. コマンド行で以下のコマンドを実行して、db2hpjv ツールを使用可能にします。

```
db2hpjv -e  
db2stop  
db2start
```

db2hpjv を使用不可にする必要がある場合は、以下のコマンドを実行します。

```
db2hpjv -d  
db2stop  
db2start
```

db2hpjv を使用不可にする必要がある場合は、以下のコマンドを実行します。db2hpjv -e を実行する前に、オペレーティング・システム上に Java をインストールしておかなければなりません。DB2 Database for Linux, UNIX, and Windows は、Java ルーチン・サポートが使用可能であるが、Java がオペレーティング・システム上にない場合には、HP-UX 上で実行できません。

2. HP-UX ランタイム・リンカーが Java 共用ライブラリーにアクセスできるようにします。

Java ストアード・プロシージャまたはユーザー定義関数を実行するには、HP-UX ランタイム・リンカーが特定の Java 共用ライブラリーにアクセスできなければならず、DB2 システムはこれらのライブラリーおよび JVM をロードできなければなりません。そのロードを行うプログラムは、setuid 特権のもとで実行されるので、/usr/lib/pa20\_64 内の従属ライブラリーだけが探索されることとなります。Java 共用ライブラリーへのアクセスを作成するには、以下のメソッドの 1 つを選択します。

- Java 共用ライブラリーへのシンボリック・リンクを作成します。これを実行するには、ルートとしてログインし、以下のコマンドを実行して Java 共用ライブラリーへのシンボリック・リンクを作成します。

```
ln -s /opt/java1.4/jre/lib/PA_RISC2.0W/*.sl /usr/lib/pa20_64  
ln -s /opt/java1.4/jre/lib/PA_RISC2.0W/hotspot/*.sl /usr/lib/pa20_64
```

これらのコマンドにより以下のライブラリーへのシンボリック・リンクが作成されます。

```
/opt/java1.4/jre/lib/PA_RISC2.0W/libnet.sl  
/opt/java1.4/jre/lib/PA_RISC2.0W/libzip.sl  
/opt/java1.4/jre/lib/PA_RISC2.0W/librmi.sl  
/opt/java1.4/jre/lib/PA_RISC2.0W/libnio.sl  
/opt/java1.4/jre/lib/PA_RISC2.0W/libverify.sl  
/opt/java1.4/jre/lib/PA_RISC2.0W/libmlib_image.sl  
/opt/java1.4/jre/lib/PA_RISC2.0W/libhprof.sl  
/opt/java1.4/jre/lib/PA_RISC2.0W/libjaas_unix.sl  
/opt/java1.4/jre/lib/PA_RISC2.0W/libawt.sl  
/opt/java1.4/jre/lib/PA_RISC2.0W/libcmm.sl  
/opt/java1.4/jre/lib/PA_RISC2.0W/libdcpr.sl  
/opt/java1.4/jre/lib/PA_RISC2.0W/libdt_socket.sl  
/opt/java1.4/jre/lib/PA_RISC2.0W/libfontmanager.sl  
/opt/java1.4/jre/lib/PA_RISC2.0W/libioser12.sl  
/opt/java1.4/jre/lib/PA_RISC2.0W/libmawt.sl  
/opt/java1.4/jre/lib/PA_RISC2.0W/libjsound.sl  
/opt/java1.4/jre/lib/PA_RISC2.0W/libjava.sl  
/opt/java1.4/jre/lib/PA_RISC2.0W/libjawt.sl  
/opt/java1.4/jre/lib/PA_RISC2.0W/libjcov.sl  
/opt/java1.4/jre/lib/PA_RISC2.0W/libjcpm.sl  
/opt/java1.4/jre/lib/PA_RISC2.0W/libjdpw.sl  
/opt/java1.4/jre/lib/PA_RISC2.0W/libjpeg.sl  
/opt/java1.4/jre/lib/PA_RISC2.0W/hotspot/libjsig.sl  
/opt/java1.4/jre/lib/PA_RISC2.0W/hotspot/libjvm.sl
```

- /opt/java1.4/jre/lib/PA\_RISC2.0W ディレクトリーと opt/java1.4/jre/lib/PA\_RISC2.0W/hotspot ディレクトリーを、/etc/dld.sl.conf ファイルおよび SHLIB\_PATH 環境に追加します。

3. HP-UX ランタイム・リンカーが Java 共用ライブラリーにアクセスできるようにします。

DB2 サーバーが、Java ルーチンの実行時に共用 Java ライブラリーを検出できない場合、-4300 エラーを生成します。





---

## 第 3 章 JDBC アプリケーション・プログラミング

JDBC アプリケーションの作成は、他の言語での SQL アプリケーションの作成と多くの点で共通しています。

一般に、以下のことを行う必要があります。

- JDBC メソッドが含まれている Java パッケージにアクセスする。
- DB2 表へのデータの送信用またはこの表からのデータの取得用の変数を宣言する。
- データ・ソースに接続する。
- SQL ステートメントを実行する。
- SQL エラーおよび警告を処理する。
- データ・ソースへの接続を切断する。

実行する必要があるタスクは他の言語のものと同様ですが、これらのタスクを実行する方法は多少異なります。

---

### 簡単な JDBC アプリケーションの例

この簡単な JDBC アプリケーションでは、JDBC アプリケーションに含める必要がある基本要素を示しています。

図 1. 簡単な JDBC アプリケーション

```
import java.sql.*; 1

public class EzJava
{
    public static void main(String[] args)
    {
        String urlPrefix = "jdbc:db2:";
        String url;
        String empNo; 2
        Connection con;
        Statement stmt;
        ResultSet rs;

        System.out.println ("**** Enter class EzJava");

        // Check the that first argument has the correct form for the portion
        // of the URL that follows jdbc:db2:,
        // as described
        // in the Connecting to a data source using the DriverManager
        // interface with the IBM Data Server Driver for JDBC and SQLJ topic.
        // For example, for IBM Data Server Driver for
        // JDBC and SQLJ type 2 connectivity,
        // args[0] might be MVS1DB2M. For
        // type 4 connectivity, args[0] might
        // be //stlmvs1:10110/MVS1DB2M.

        if (args.length==0)
        {
            System.err.println ("Invalid value. First argument appended to "+
                "jdbc:db2: must specify a valid URL.");
        }
    }
}
```

```

        System.exit(1);
    }
    url = urlPrefix + args[0];

    try
    {
        // Load the driver
        Class.forName("com.ibm.db2.jcc.DB2Driver");
        System.out.println("**** Loaded the JDBC driver");
        3a

        // Create the connection using the IBM Data Server Driver for JDBC and SQLJ
        con = DriverManager.getConnection(url);
        3b
        // Commit changes manually
        con.setAutoCommit(false);
        System.out.println("**** Created a JDBC connection to the data source");

        // Create the Statement
        stmt = con.createStatement();
        System.out.println("**** Created JDBC Statement object");
        4a

        // Execute a query and generate a ResultSet instance
        rs = stmt.executeQuery("SELECT EMPNO FROM EMPLOYEE");
        System.out.println("**** Created JDBC ResultSet object");
        4b

        // Print all of the employee numbers to standard output device
        while (rs.next()) {
            empNo = rs.getString(1);
            System.out.println("Employee number = " + empNo);
        }
        System.out.println("**** Fetched all rows from JDBC ResultSet");
        // Close the ResultSet
        rs.close();
        System.out.println("**** Closed JDBC ResultSet");

        // Close the Statement
        stmt.close();
        System.out.println("**** Closed JDBC Statement");

        // Connection must be on a unit-of-work boundary to allow close
        con.commit();
        System.out.println ( "**** Transaction committed" );

        // Close the connection
        con.close();
        System.out.println("**** Disconnected from data source");
        6

        System.out.println("**** JDBC Exit from class EzJava - no errors");
    }

    catch (ClassNotFoundException e)
    {
        System.err.println("Could not load JDBC driver");
        System.out.println("Exception: " + e);
        e.printStackTrace();
    }

    catch(SQLException ex)
    {
        System.err.println("SQLException information");
        while(ex!=null) {
            System.err.println ("Error msg: " + ex.getMessage());
            System.err.println ("SQLSTATE: " + ex.getSQLState());
            System.err.println ("Error code: " + ex.getErrorCode());
            ex.printStackTrace();
            ex = ex.getNextException(); // For drivers that support chained exceptions
        }
    }
    5

```

```

    }
  }
} // End main
} // End EzJava

```

27 ページの図 1 の注:

注	説明
1	このステートメントにより、JDBC コア API が含まれている <code>java.sql</code> パッケージがインポートされます。アクセスが必要になる可能性のある他の Java パッケージについては、『JDBC サポート用の Java パッケージ』を参照してください。
2	String 変数 <code>empNo</code> により、ホスト変数の機能が実行されます。つまり、この変数は SQL 照会から検索されたデータを保持するために使用されます。詳細については、『JDBC アプリケーションでの変数』を参照してください。
3a および 3b	これら 2 つのステートメントのセットでは、使用可能な 2 つのインターフェースのいずれかを使用して、データ・ソースに接続する方法が示されています。詳細については、『JDBC アプリケーションによるデータ・ソースへの接続の方法』を参照してください。
	手順 3a (JDBC ドライバーのロード) は、JDBC 4.0 を使用する場合は不要です。
4a および 4b	これら 2 つのステートメントのセットでは、JDBC で <code>SELECT</code> を実行する方法が示されています。他の SQL 操作を実行する方法については、『SQL を実行するための JDBC インターフェース』を参照してください。
5	この <code>try/catch</code> ブロックでは、SQL エラー処理用の <code>SQLException</code> クラスの使用が示されています。SQL エラーの処理の詳細については、『IBM Data Server Driver for JDBC and SQLJ 使用時の <code>SQLException</code> の処理』を参照してください。SQL 警告の処理については、『IBM Data Server Driver for JDBC and SQLJ 使用時の <code>SQLWarning</code> の処理』を参照してください。
6	このステートメントにより、アプリケーションがデータ・ソースから切断されます。『JDBC アプリケーションでのデータ・ソースからの切断』を参照してください。

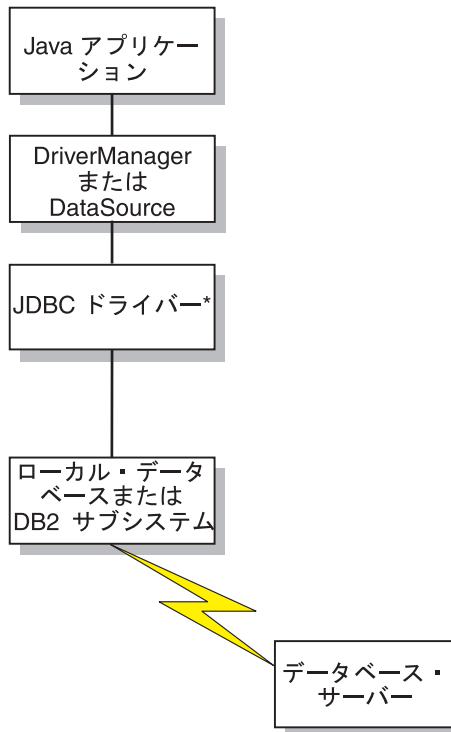
---

## JDBC アプリケーションによるデータ・ソースへの接続の方法

SQL プログラムで SQL ステートメントを実行する前に、データ・ソースに接続する必要があります。

IBM Data Server Driver for JDBC and SQLJ では、タイプ 2 接続およびタイプ 4 接続がサポートされます。DB2 データベースへの接続では、タイプ 2 またはタイプ 4 の接続を使用することができます。IBM Informix Dynamic Server (IDS) データベースへの接続では、タイプ 4 の接続を使用することができます。

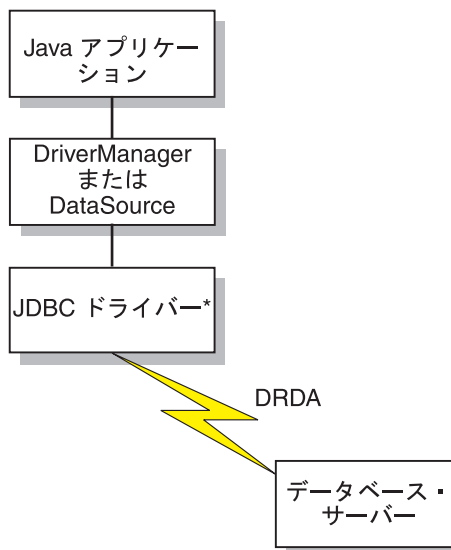
次の図では、Java アプリケーションによる IBM Data Server Driver for JDBC and SQLJ Type 2 接続を使用したデータ・ソースへの接続方法が示されています。



\*JVM で実行される Java バイトコードとネイティブ・コード

図2. IBM Data Server Driver for JDBC and SQLJ Type 2 接続での Java アプリケーションの流れ

次の図では、Java アプリケーションによる IBM Data Server Driver for JDBC and SQLJ Type 4 接続を使用したデータ・ソースへの接続方法が示されています。



\*JVM で実行される Java バイトコード

図3. IBM Data Server Driver for JDBC and SQLJ Type 4 接続での Java アプリケーションの流れ

## DB2 アプリケーションによる DriverManager インターフェースと DB2 JDBC Type 2 ドライバーを使用したデータ・ソースへの接続

JDBC アプリケーションでは、`java.sql` パッケージの一部である JDBC DriverManager インターフェースを使用して、データ・ソースへの接続を確立することができます。

Java アプリケーションでは、最初に `Class.forName` メソッドが呼び出されて、JDBC ドライバーがロードされます。アプリケーションによりドライバーがロードされた後、`DriverManager.getConnection` メソッドが呼び出されて、データ・ソースに接続されます。

Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバー (DB2 JDBC Type 2 ドライバー) の場合、次の引数で `Class.forName` メソッドを呼び出して、このドライバーをロードすることができます。

```
COM.ibm.db2.jdbc.app.DB2Driver
```

以下のコードでは、DB2 JDBC Type 2 ドライバーのロードが示されています。

```
try {
    // Load the DB2 JDBC Type 2 Driver with DriverManager
    Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
```

`catch` ブロックは、ドライバーが見つからなかった場合のエラーの出力に使用されます。

ドライバーのロード後、`DriverManager.getConnection` メソッドを呼び出して、データ・ソースに接続します。以下のいずれかの形式の `getConnection` を使用できます。

```
getConnection(String url);
getConnection(String url, user, password);
getConnection(String url, java.util.Properties info);
```

`url` 引数により、データ・ソースが表されます。

DB2 JDBC Type 2 ドライバーの場合、次の形式の URL を指定します。

*DB2 JDBC Type 2 ドライバーの URL の構文*

▶—jdbc—:—db2—:—database—▶

URL の各部の意味は、以下のとおりです。

**jdbc:db2:**

`jdbc:db2:` は、接続が DB2 データ・ソースに対するものであることを示します。

## database

データベース別名。この別名は、DB2 クライアント上の DB2 のデータベース・カタログ項目を指します。

*info* 引数は、接続用の一連のドライバー・プロパティーを含んだタイプ `java.util.Properties` のオブジェクトです。*info* 引数の指定は、URL での `property=value` ストリングの指定の代わりとなります。

接続用のユーザー ID およびパスワードの指定: 接続用のユーザー ID およびパスワードを指定するには、次の方法があります。

- *user* および *password* を指定する `getConnection` メソッドの形式を使用します。
- `java.util.Properties` オブジェクトで *user* および *password* プロパティーを設定してから、*info* を指定する `getConnection` メソッドの形式を使用します。

例: *user* および *password* パラメーターへのユーザー ID およびパスワードの設定

```
String url = "jdbc:db2:toronto";  
  
// Set URL for data source  
  
String user = "db2adm";  
String password = "db2adm";  
Connection con = DriverManager.getConnection(url, user, password);  
// Create connection
```

例: `java.util.Properties` オブジェクトへのユーザー ID およびパスワードの設定

```
Properties properties = new Properties(); // Create Properties object  
properties.put("user", "db2adm"); // Set user ID for connection  
properties.put("password", "db2adm"); // Set password for connection  
String url = "jdbc:db2:toronto";  
  
// Set URL for data source  
  
Connection con = DriverManager.getConnection(url, properties);  
// Create connection
```

## DriverManager インターフェースと IBM Data Server Driver for JDBC and SQLJ を使用したデータ・ソースへの接続

JDBC アプリケーションでは、`java.sql` パッケージの一部である JDBC `DriverManager` インターフェースを使用して、データ・ソースへの接続を確立することができます。

接続を確立するための手順は、以下のとおりです。

1. `Class.forName` メソッドを呼び出して、JDBC ドライバーをロードします。

JDBC 4.0 を使用している場合は、JDBC ドライバーを明示的にロードする必要はありません。

IBM Data Server Driver for JDBC and SQLJ では、`Class.forName` メソッドを呼び出すときに以下の引数を指定してドライバーをロードします。

```
com.ibm.db2.jcc.DB2Driver
```

以前の JDBC ドライバーとの互換性を維持するために、代わりに以下の引数を使用することもできます。

```
COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver
```

IBM Data Server Driver for JDBC and SQLJ をロードするコードを以下に示します。

```
try {
    // Load the IBM Data Server Driver for JDBC and SQLJ with DriverManager
    Class.forName("com.ibm.db2.jcc.DB2Driver");
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
```

catch ブロックは、ドライバーが見つからなかった場合のエラーの出力に使用されます。

2. DriverManager.getConnection メソッドを呼び出して、データ・ソースに接続します。

以下のいずれかの形式の getConnection を使用できます。

```
getConnection(String url);
getConnection(String url, user, password);
getConnection(String url, java.util.Properties info);
```

IBM Data Server Driver for JDBC and SQLJ Type 4 接続では、パラメーターまたはプロパティ値によって getConnection メソッドにユーザー ID とパスワードを指定する必要があります。

url 引数はデータ・ソースに対応しており、どのタイプの JDBC 接続を使用するのかをこの引数で示します。

info 引数は、接続用の一連のドライバー・プロパティを含んだタイプ java.util.Properties のオブジェクトです。info 引数を指定すれば、URL で *property=value*; スtring を指定する代わりになります。指定できるプロパティについては、『IBM Data Server Driver for JDBC and SQLJ のプロパティ』を参照してください。

接続用のユーザー ID とパスワードを指定するには、いくつかの方法があります。

- url に *property=value*; 節を指定した形式の getConnection メソッドを使用して、URL に user と password のプロパティを組み込みます。
- user および password を指定する getConnection メソッドの形式を使用します。
- java.util.Properties オブジェクトで user および password プロパティを設定してから、info を指定する getConnection メソッドの形式を使用します。

例: 接続を確立し、URL でユーザー ID とパスワードを設定する:

```
String url = "jdbc:db2://myhost:5021/mydb:" +
    "user=dbadm;password=dbadm;";

// Set URL for data source
Connection con = DriverManager.getConnection(url);
// Create connection
```

例: 接続を確立し、user と password のパラメーターでユーザー ID とパスワードを設定する:



```
String url = "jdbc:db2://myhost:5021/mydb";
// Set URL for data source
String user = "dbadm";
String password = "dbadm";
Connection con = DriverManager.getConnection(url, user, password);
// Create connection
```

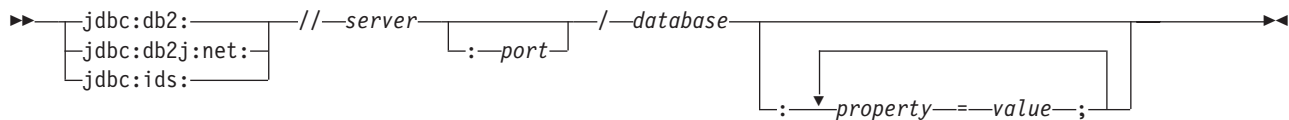
例: 接続を確立し、`java.util.Properties` オブジェクトでユーザー ID とパスワードを設定する:

```
Properties properties = new Properties(); // Create Properties object
properties.put("user", "dbadm"); // Set user ID for connection
properties.put("password", "dbadm"); // Set password for connection
String url = "jdbc:db2://myhost:5021/mydb";
// Set URL for data source
Connection con = DriverManager.getConnection(url, properties);
// Create connection
```

## IBM Data Server Driver for JDBC and SQLJ Type 4 接続の URL 形式

JDBC アプリケーションでタイプ 4 接続を使用しており、`DriverManager` インターフェースを使用して接続を行っている場合、`DriverManager.getConnection` 呼び出しでタイプ 4 接続を示す URL を指定する必要があります。

### IBM Data Server Driver for JDBC and SQLJ Type 4 接続 URL 構文



### IBM Data Server Driver for JDBC and SQLJ Type 4 接続 URL オプションの説明

URL の各部の意味は、以下のとおりです。

#### jdbc:db2: または jdbc:db2j:net:

URL の最初の部分の意味は、以下のとおりです。

##### jdbc:db2:

接続先が DB2 for z/OS、DB2 Database for Linux, UNIX, and Windows であることを示します。

また、アプリケーションの移植性のために、`jdbc:db2:` も IBM Informix Dynamic Server (IDS) データベースへの接続に使用できます。

##### jdbc:db2j:net:

接続先がリモートの IBM Cloudscape™ サーバーであることを示します。

##### jdbc:ids:

接続先が IDS データ・ソースであることを示します。`jdbc:informix-sqli:` も接続先が IDS データ・ソースであることを示しますが、`jdbc:ids:` を使用してください。

##### server

データ・ソースのドメイン名または IP アドレス。

## port

データ・ソースに割り当てられている TCP/IP サーバーのポート番号。これは 0 から 65535 の間の整数です。デフォルトは 446 です。

## database

データ・ソースの名前。

- 接続を DB2 for z/OS サーバーに対して行う場合、*database* は、インストール中に定義される DB2 ロケーション名です。DB2 ロケーション名の中の文字はすべて大文字でなければなりません。IBM Data Server Driver for JDBC and SQLJ は、データベース値の小文字を IBM Data Server Driver for JDBC and SQLJ Type 4 接続用の大文字に変換しません。

サーバーで以下の SQL ステートメントを実行することにより、ロケーション名を判別できます。

```
SELECT CURRENT SERVER FROM SYSIBM.SYSDUMMY1;
```

- 接続を DB2 for z/OS サーバーに対して行う場合、*database* 内のすべての文字を大文字にしなければなりません。
- 接続を DB2 Database for Linux, UNIX, and Windows サーバーに対して行う場合、*database* は、インストール中に定義されるデータベース名です。
- 接続を IDS サーバーに対して行う場合、*database* は、データベース名です。この名前は大/小文字を区別しません。サーバーにより、名前は小文字に変換されます。
- 接続を IBM Cloudscape サーバーに対して行う場合、*database* は、データベースを含むファイルの完全修飾名です。この名前は、二重引用符 (") で囲まなければなりません。以下に例を示します。

```
"c:/databases/testdb"
```

## *property=value;*

JDBC 接続のプロパティとその値。1 つ以上のプロパティと値の対を指定できます。プロパティと値の各対 (最後のものを含む) はセミコロン (;) で終わらなければなりません。プロパティと値のストリングのリスト内のどこにも、スペースまたはその他の空白文字を組み込まないでください。

**int** データ・タイプのプロパティには、事前定義定数フィールド値を持つものがあります。定数フィールド値は、*url* パラメーターで使用する前に、整数値に解決する必要があります。例えば、

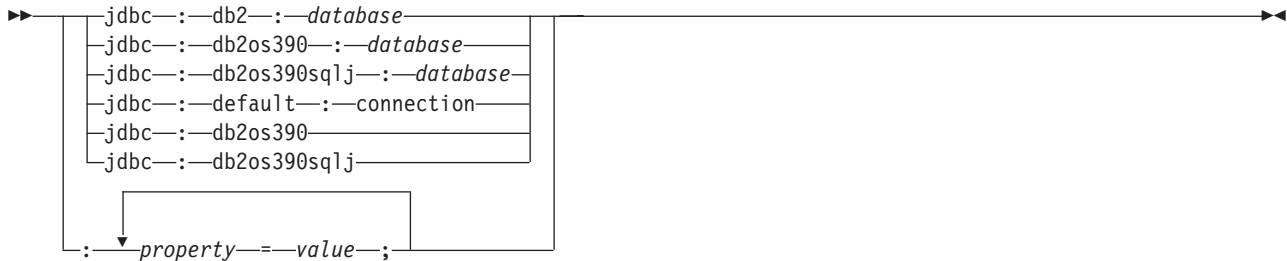
`com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL` を *url* パラメーターに使用することはできません。しかし、`com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL` を含む URL ストリングを作成して、その URL ストリングをストリング変数に割り当てることができます。その後、以下のようにして、そのストリング変数を *url* パラメーターで使用できます。

```
String url =
    "jdbc:db2://sysmvs1.st1.ibm.com:5021" +
    "user=dbadm;password=dbadm;" +
    "traceLevel=" +
    (com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL) + ";";
Connection con =
    java.sql.DriverManager.getConnection(url);
```

## IBM Data Server Driver for JDBC and SQLJ Type 2 接続の URL 形式

JDBC アプリケーションでタイプ 2 接続を使用しており、DriverManager インターフェースによって接続を行っている場合、DriverManager.getConnection 呼び出し内の URL がタイプ 2 接続を示すことを指定する必要があります。

### IBM Data Server Driver for JDBC and SQLJ Type 2 接続 URL 構文



### IBM Data Server Driver for JDBC and SQLJ Type 2 接続 URL オプションの説明

URL の各部の意味は、以下のとおりです。

#### jdbc:db2: または jdbc:db2j:net:

URL の最初の部分の意味は、以下のとおりです。

##### jdbc:db2:

DB2 for z/OS または DB2 Database for Linux, UNIX, and Windows サーバーへの接続を示します。

##### jdbc:default:connection

URL が、CICS<sup>®</sup>、IMS<sup>™</sup>、または Java ストアード・プロシージャ環境によって制御される DB2 スレッドを介したローカル・サブシステムへの接続のためのものであることを示します。

##### jdbc:db2j:net:

接続先がリモートの IBM Cloudscape サーバーであることを示します。

#### database

データベース・サーバーの名前。

- `serverName` 接続プロパティの値がヌルの場合、`database` は、インストール中に定義されるデータベース名になります。 `serverName` プロパティの値がヌルではない場合、`database` はデータベース別名になります。
- 接続を DB2 for z/OS サーバーまたは DB2 for i5/OS サーバーに対して行う場合、`database` 内のすべての文字を大文字にしなければなりません。

#### property=value;

JDBC 接続のプロパティとその値。1 つ以上のプロパティと値の対を指定できます。プロパティと値の各対 (最後のものを含む) はセミコロン (;) で終わらなければなりません。プロパティと値のストリングのリスト内のどこにも、スペースまたはその他の空白文字を組み込まないでください。

`int` データ・タイプのプロパティには、事前定義定数フィールド値を持つものがあります。定数フィールド値は、`url` パラメーターで使用する前に、整数値に

解決する必要があります。例えば、`com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL` を `url` パラメーターに使用することはできません。しかし、`com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL` を含む URL スtringを作成して、その URL スtringをString変数に割り当てることができます。その後、以下のようにして、そのString変数を `url` パラメーターで使用できます。

```
String url =
    "jdbc:db2://sysmvs1.st1.ibm.com:5021" +
    "user=dbadm;password=dbadm;" +
    "traceLevel=" +
    (com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL) + ";";
Connection con =
    java.sql.DriverManager.getConnection(url);
```

## DataSource インターフェースを使用したデータ・ソースへの接続

さまざまなデータ・ソースで移植可能なアプリケーションを作成する必要がある場合は、DataSource インターフェースを使用します。

DriverManager を使用してデータ・ソースに接続すると、アプリケーションは、特定の JDBC ドライバー・クラス名とドライバー URL を識別しなければならなくなるので、移植性が低下します。ドライバー・クラス名とドライバー URL は、JDBC ベンダー、ドライバーの実装、データ・ソースによって異なります。

DataSource インターフェースによってデータ・ソースに接続する場合は、DataSource オブジェクトを使用します。

DataSource オブジェクトを使用するための最も簡単な方法は、DriverManager インターフェースの場合のように、オブジェクトの作成と使用を同じアプリケーションで実行することです。ただし、この方法では移植性が得られません。

DataSource オブジェクトを使用する最善の方法は、システム管理者が WebSphere Application Server などのツールを使用して、オブジェクトの作成と管理を別途行うようにすることです。DataSource オブジェクトを作成して管理するプログラムでは、Java Naming and Directory Interface (JNDI) も使用して、論理名を DataSource オブジェクトに割り当てます。DataSource オブジェクトを使用する JDBC アプリケーションでは、その論理名でオブジェクトを参照できるので、基礎になっているデータ・ソースの情報が不要になります。さらに、システム管理者がデータ・ソースの属性を変更する場合でも、アプリケーション・プログラムを使用する側でアプリケーション・プログラムを変更する必要はありません。

WebSphere を使用して DataSource オブジェクトをデプロイする方法の詳細については、Web で以下の URL を参照してください。

<http://www.ibm.com/software/webservers/appserv/>

DataSource オブジェクトを自分でデプロイする方法については、『DataSource オブジェクトの作成およびデプロイ』を参照してください。

DataSource インターフェースと DriverManager インターフェースを同じアプリケーションで使用することは可能ですが、移植性を最大限に高めるために、DataSource インターフェースだけを使用して接続を取得することをお勧めします。

システム管理者が既に作成して論理名を割り当てた `DataSource` オブジェクトを使用して接続を取得するには、以下の手順を実行します。

1. システム管理者から、接続する必要があるデータ・ソースの論理名を取得します。
2. 次の手順で使用する `Context` オブジェクトを作成します。 `Context` インターフェースは、`JDBC` ではなく `Java Naming and Directory Interface (JNDI)` に組み込まれています。
3. アプリケーション・プログラムで `JNDI` を使用して、論理データ・ソース名に関連した `DataSource` オブジェクトを取得します。
4. `DataSource.getConnection` メソッドを使用して接続を取得します。

以下のいずれかの形式の `getConnection` メソッドを使用できます。

```
getConnection();  
getConnection(String user, String password);
```

接続のユーザー ID とパスワードとして、`DataSource` のデプロイ時に指定されているユーザー ID とパスワード以外の値を指定する必要がある場合は、2 番目の形式を使用します。

システム管理者が作成した `DataSource` オブジェクトを使用して接続を取得する例:  
この例で接続する必要があるデータ・ソースの論理名は、`jdbc/sampledb` です。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```
import java.sql.*;  
import javax.naming.*;  
import javax.sql.*;  
...  
Context ctx=new InitialContext();  
DataSource ds=(DataSource)ctx.lookup("jdbc/sampledb");  
Connection con=ds.getConnection();
```

2  
3  
4

図4. `DataSource` オブジェクトによる接続の取得

`DataSource` オブジェクトの作成と使用を同じアプリケーションで実行する例:

図5. `DataSource` オブジェクトの作成と使用を同じアプリケーションで実行する

```
import java.sql.*;          // JDBC base  
import javax.sql.*;        // Additional methods for JDBC  
import com.ibm.db2.jcc.*;   // IBM Data Server Driver for JDBC and SQLJ  
                            // interfaces  
DB2SimpleDataSource dbds=new DB2SimpleDataSource();  
dbds.setDatabaseName("dbloc1");  
                            // Assign the location name  
dbds.setDescription("Our Sample Database");  
                            // Description for documentation  
dbds.setUser("john");  
                            // Assign the user ID  
dbds.setPassword("dbadm");  
                            // Assign the password  
Connection con=dbds.getConnection();  
                            // Create a Connection object
```

1

2  
3

4

注	説明
1	DataSource インターフェースの実装が含まれているパッケージをインポートしません。
2	DB2SimpleDataSource オブジェクトを作成します。DB2SimpleDataSource は、DataSource インターフェースの IBM Data Server Driver for JDBC and SQLJ 実装の 1 つです。DB2 の DataSource 実装については、『DataSource オブジェクトの作成およびデプロイ』を参照してください。
3	setDatabaseName メソッド、setDescription メソッド、setUser メソッド、setPassword メソッドによって、DB2SimpleDataSource オブジェクトに属性を割り当てます。IBM Data Server Driver for JDBC and SQLJ 使用時に DB2SimpleDataSource オブジェクトに設定できる属性については、『IBM Data Server Driver for JDBC and SQLJ のプロパティ』を参照してください。
4	DB2SimpleDataSource オブジェクト dbds に対応するデータ・ソースへの接続を確立します。

## 使用する IBM Data Server Driver for JDBC and SQLJ 接続のタイプを判別する方法

IBM Data Server Driver for JDBC and SQLJ は、タイプ 2 接続とタイプ 4 接続という 2 種類のタイプの接続をサポートしています。

DriverManager インターフェースの場合、DriverManager.getConnection メソッドの URL を介して接続のタイプを指定します。DataSource インターフェースの場合、driverType プロパティを介して接続のタイプを指定します。

次の表は、タイプ 2 接続とタイプ 4 接続の違いを要約しています。

表 5. IBM Data Server Driver for JDBC and SQLJ Type 2 接続と IBM Data Server Driver for JDBC and SQLJ Type 4 接続の比較

関数	IBM Data Server Driver for JDBC and SQLJ Type 2 接続のサポート	IBM Data Server Driver for JDBC and SQLJ Type 4 接続のサポート
SYSPLEX ワークロード・バランシングおよび接続コンセントレーター	DB2 Connect によるサポート	単一の JVM 内の接続用ドライバーによる直接サポート  複数の JVM にわたる DB2 Connect を介したサポート
通信プロトコル	TCP/IP	TCP/IP
パフォーマンス	ローカル DB2 サーバーへのアクセスに適している	リモート DB2 サーバーへのアクセスに適している
インストール	Java クラスに加えてネイティブ・ライブラリーのインストールが必要	Java クラスのみのインストールが必要
ストアド・プロシージャ	ストアド・プロシージャの呼び出しまたは実行に使用できる	ストアド・プロシージャの呼び出しにのみ使用できる
分散トランザクション処理 (XA)	サポートされる	サポートされる
J2EE 1.4 準拠	準拠	準拠

以下の点は、使用する接続のタイプを判別するのに役立ちます。



以下の環境では、IBM Data Server Driver for JDBC and SQLJ Type 2 接続を使用します。

- JDBC または SQLJ アプリケーションは大抵の場合ローカルで実行される。

ローカル・アプリケーションでは、タイプ 2 接続を使用するとパフォーマンスが向上します。

- Java ストアード・プロシージャを実行している。

ストアード・プロシージャ環境は 2 つの部分から成ります。1 つはクライアント・プログラムで、そこからストアード・プロシージャを呼び出します。もう 1 つはサーバー・プログラムで、これがストアード・プロシージャです。ストアード・プロシージャは、タイプ 2 またはタイプ 4 接続を使用する JDBC または SQLJ プログラムで呼び出すことができますが、Java ストアード・プロシージャはタイプ 2 接続を使用して実行する必要があります。

以下の環境では、IBM Data Server Driver for JDBC and SQLJ Type 4 接続を使用します。

- JDBC または SQLJ アプリケーションは大抵の場合リモートで実行される。

リモート・アプリケーションでは、タイプ 4 接続を使用するとパフォーマンスが向上します。

- IBM Data Server Driver for JDBC and SQLJ 接続コンセントレータおよび Sysplex ワークロード・บาลランシング・サポートを使用している。

## JDBC 接続オブジェクト

いずれかの接続メソッドによってデータ・ソースに接続する場合、データ・ソースへの接続を表す Connection オブジェクトを作成します。

この Connection オブジェクトを使用して、以下の事柄を行います。

- SQL ステートメントを実行するための Statement、PreparedStatement、および CallableStatement オブジェクトを作成します。このことについては、『JDBC アプリケーションでの SQL ステートメントの実行』で解説されています。
- 接続先のデータ・ソースに関する情報を収集します。このプロセスについては、『DatabaseMetaData メソッドを使用した、データ・ソースに関する情報の取得』で解説されています。
- トランザクションをコミットまたはロールバックします。トランザクションのコミットは手動でも自動でも行えます。これらの操作については、『JDBC トランザクションのコミットまたはロールバック』で解説されています。
- データ・ソースへの接続をクローズします。この操作については、『JDBC アプリケーションでのデータ・ソースからの切断』で解説されています。

## DataSource オブジェクトの作成およびデプロイ

JDBC バージョン 2.0 以降には、データ・ソースに接続するための DataSource インターフェースが用意されています。この DataSource インターフェースを使用することが、データ・ソースに接続するための好ましい方法です。

DataSource インターフェースを使用するには、2 部からなる手順が必要です。



- DataSource オブジェクトの作成およびデプロイ。この手順は通常、システム管理者が WebSphere Application Server などのツールを使用して実行します。
- DataSource オブジェクトによる接続の作成。この手順は、アプリケーション・プログラムで実行します。

このトピックでは、DataSource オブジェクトを自分で作成してデプロイするときに必要な情報を取り上げます。

IBM Data Server Driver for JDBC and SQLJ には、以下の DataSource 実装が用意されています。

- com.ibm.db2.jcc.DB2SimpleDataSource。接続プールはサポートしていません。この実装は、IBM Data Server Driver for JDBC and SQLJ Type 2 接続または IBM Data Server Driver for JDBC and SQLJ Type 4 接続で使用できます。
- com.ibm.db2.jcc.DB2ConnectionPoolDataSource。接続プールをサポートしています。この実装は、IBM Data Server Driver for JDBC and SQLJ Type 2 接続または IBM Data Server Driver for JDBC and SQLJ Type 4 接続で使用できます。
- com.ibm.db2.jcc.DB2XADataSource。接続プールと分散トランザクションをサポートしています。接続プールは、WebSphere Application Server や他のアプリケーション・サーバーによって用意します。この実装は、IBM Data Server Driver for JDBC and SQLJ Type 4 接続でのみ使用できます。

DB2 JDBC Type 2 ドライバーには、以下の DataSource 実装が用意されています。

- COM.ibm.db2.jdbc.DB2DataSource。接続プールに対応しています。この実装の場合、接続プールは内部で処理され、アプリケーションからは認識されません。
- COM.ibm.db2.jdbc.DB2XADataSource。分散トランザクションと接続プールの組み込みサポートはありません。この実装の場合は、分散トランザクションと接続プールを自分で管理する必要があります。そのためには、独自のコードを記述するか、WebSphere Application Server などのツールを使用します。

DataSource オブジェクトを作成してデプロイするときには、以下のタスクを実行する必要があります。

1. 適切な DataSource 実装のインスタンスを作成します。
2. DataSource オブジェクトのプロパティを設定します。
3. そのオブジェクトを Java Naming and Directory Interface (JNDI) ネーム・サービスに登録します。

これらのタスクを実行する例を以下に示します。

```

import java.sql.*;          // JDBC base
import javax.naming.*;     // JNDI Naming Services
import javax.sql.*;        // Additional methods for JDBC
import com.ibm.db2.jcc.*;  // IBM Data Server Driver for
                          // JDBC and SQLJ
                          // implementation of JDBC
                          // standard extension APIs

DB2SimpleDataSource dbds = new com.ibm.db2.jcc.DB2SimpleDataSource(); 1

dbds.setDatabaseName("db2loc1"); 2
dbds.setDescription("Our Sample Database");
dbds.setUser("john");
dbds.setPassword("mypw");
...
Context ctx=new InitialContext(); 3
Ctx.bind("jdbc/sampledb",dbds); 4

```

図 6. *DataSource* オブジェクトを作成してデプロイする例

注	説明
1	DB2SimpleDataSource クラスのインスタンスを作成します。
2	このステートメントとその後の 3 つのステートメントによって、この DB2SimpleDataSource オブジェクトのプロパティの値を設定します。
3	JNDI で使用するコンテキストを作成します。
4	DBSimple2DataSource オブジェクト dbds を論理名 jdbc/sampledb に関連付けます。このオブジェクトを使用するアプリケーションは、その名前 jdbc/sampledb でオブジェクトを参照できます。

---

## JDBC サポート用の Java パッケージ

JDBC メソッドを呼び出すには、それらのメソッドを含む様々な Java パッケージのすべてまたは一部にあらかじめアクセスできるようにしておく必要があります。

パッケージまたは特定のクラスをインポートするか、または完全修飾クラス名を使用することにより、これを行えます。JDBC プログラムについて以下のパッケージまたはクラスが必要になることがあります。

### **java.sql**

コア JDBC API が含まれます。

### **javax.naming**

*DataSource* をインプリメントするためにしばしば使用される Java Naming and Directory Interface (JNDI) のクラスおよびインターフェースが含まれています。

### **javax.sql**

Java を使用してサーバー・サイドのアプリケーションを作成するためのメソッドが含まれています。

### **javax.transaction**

Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバー (DB2 JDBC Type 2 ドライバー) 用の分散トランザクションの JDBC サポートが含まれています。

### com.ibm.db2.jcc

IBM Data Server Driver for JDBC and SQLJ 用の JDBC のインプリメンテーションが含まれています。

### COM.ibm.db2.jdbc

DB2 JDBC Type 2 ドライバー用の JDBC のインプリメンテーションが含まれています。

---

## DatabaseMetaData メソッドを使用した、データ・ソースに関する情報の取得

DatabaseMetaData インターフェースには、データ・ソースの情報を取得するためのメソッドが含まれています。それらのメソッドは、さまざまなデータ・ソースにアクセスできる汎用アプリケーションを作成するときに便利です。

さまざまなデータ・ソースにアクセスできる汎用アプリケーションを実行する場合は、まずデータ・ソースがさまざまなデータベース操作を処理できるかどうかをテストする必要があります。例えば、JDBC 3.0 のメソッドをデータ・ソースのドライバーに対して呼び出すのであれば、まずそのドライバーが JDBC 3.0 のレベルかどうかを確認する必要があります。

DatabaseMetaData メソッドでは、以下のタイプの情報を確認できます。

- データ・ソースがサポートしているフィーチャー (ANSI SQL レベルなど)
- JDBC ドライバーに関する具体的な情報 (ドライバーのレベルなど)
- 制限事項 (索引に組み込める列の最大数など)
- データ・ソースがデータ定義ステートメント (CREATE、ALTER、DROP、GRANT、REVOKE) をサポートしているかどうか
- データ・ソースのオブジェクト・リスト (表、索引、プロシージャなど)
- データ・ソースがさまざまな JDBC 機能 (バッチ更新、スクロール可能 ResultSet など) をサポートしているかどうか
- ドライバーがサポートしているスカラー関数のリスト

DatabaseMetaData メソッドを呼び出すには、以下の基本手順を実行する必要があります。

1. 接続に対して getMetaData メソッドを呼び出して、DatabaseMetaData オブジェクトを作成します。
2. DatabaseMetaData メソッドを呼び出して、データ・ソースに関する情報を取得します。
3. メソッドから ResultSet が返された場合は、以下のようになります。
  - a. ループの中で、next メソッドを使用してカーソルを配置し、getXXX メソッドを使用して、ResultSet オブジェクトの現在行の各列からデータを取得します。
  - b. close メソッドを呼び出して ResultSet オブジェクトをクローズします。

**例:** DatabaseMetaData メソッドを使用して、ドライバーのバージョンを確認し、データ・ソースで使用できるストアード・プロシージャのリストを取得し、ドライバーがサポートしている日時関数のリストを取得するためのコードを以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対

応しています。

図7. *DatabaseMetaData* メソッドによるデータ・ソース情報の取得

```
Connection con;
DatabaseMetaData dbmtadta;
ResultSet rs;
int mtadtaint;
String procSchema;
String procName;
String dtfnList;
...
dbmtadta = con.getMetaData(); // Create the DatabaseMetaData object 1
mtadtaint = dbmtadta.getDriverVersion(); // Check the driver version 2
System.out.println("Driver version: " + mtadtaint);
rs = dbmtadta.getProcedures(null, null, "%"); // Get information for all procedures
while (rs.next()) { // Position the cursor 3a
    procSchema = rs.getString("PROCEDURE_SCHEM"); // Get procedure schema
    procName = rs.getString("PROCEDURE_NAME"); // Get procedure name
    System.out.println(procSchema + "." + procName); // Print the qualified procedure name
}
dtfnList = dbmtadta.getTimeDateFunctions(); // Get list of supported datetime functions
System.out.println("Supported datetime functions:");
System.out.println(dtfnList); // Print the list of datetime functions
rs.close(); // Close the ResultSet 3b
```

---

## JDBC アプリケーションでの変数

他の Java アプリケーションと同様に、JDBC アプリケーションを作成するときには変数を宣言します。Java アプリケーションでは、そのような変数は Java ID として知られます。

これらの ID のいくつかは他の言語のホスト変数と同じ関数を持っています。これはデータベース表に受け渡すデータや、データベース表から取り出すデータを保持します。以下のコードの ID `empNo` は `EMPNO` 表列から取り出すデータを保持します。このデータのデータ・タイプは `CHAR` です。

```
String empNo;
// Execute a query and generate a ResultSet instance
rs = stmt.executeQuery("SELECT EMPNO FROM EMPLOYEE");
while (rs.next()) {
    String empNo = rs.getString(1);
    System.out.println("Employee number = " + empNo);
}
```

Java データ・タイプを選択すると、パフォーマンスに影響を与えることがあります。なぜなら、Java 変数のデータ・タイプが DB2 データ・タイプに緊密にマップされている場合、DB2 は最適なアクセス・パスを選択するからです。

## SQL を実行するための JDBC インターフェース

従来の SQL プログラムでは、SQL ステートメントを実行して、表へのデータの挿入、更新、削除、またはマージ、表からのデータの取り出し、あるいはストアード・プロシージャの呼び出しを行います。JDBC プログラムで同じ機能を実行するには、メソッドを呼び出します。

これらのメソッドは以下のインターフェースで定義されます。

- **Statement** インターフェースはすべての SQL ステートメントの実行をサポートします。以下のインターフェースは **Statement** インターフェースからメソッドを継承します。
  - **PreparedStatement** インターフェースは、入力パラメーター・マーカーを含む SQL ステートメントをサポートします。パラメーター・マーカーは入力変数を表します。また、**PreparedStatement** インターフェースはパラメーター・マーカーがない SQL ステートメントにも使用できます。

IBM Data Server Driver for JDBC and SQLJ では、**PreparedStatement** インターフェースは、入力パラメーターを持っていても出力パラメーターを持たないために結果セットを戻さないストアード・プロシージャを呼び出すために使用できます。ただし、推奨されるインターフェースは **CallableStatement** です。

- **CallableStatement** インターフェースは、ストアード・プロシージャの呼び出しをサポートします。

**CallableStatement** インターフェースは、入力パラメーター、出力パラメーター、入出力パラメーターのいずれかを持つか、またはパラメーターを持たないストアード・プロシージャを呼び出すために使用できます。IBM Data Server Driver for JDBC and SQLJ では、**Statement** インターフェースを使用してストアード・プロシージャを呼び出すこともできますが、そのストアード・プロシージャにはパラメーターは指定できません。

- **ResultSet** インターフェースは、照会が生成する結果にアクセスできるようにします。**ResultSet** インターフェースには、他の言語の SQL アプリケーションで使われるカーソルと同じ目的があります。

### Statement.executeUpdate メソッドによるデータベース・オブジェクトの作成と変更

JDBC メソッドの 1 つである **Statement.executeUpdate** を使用して、表を更新したり、ストアード・プロシージャを呼び出したりできます。

**Statement.executeUpdate** メソッドを使用して、以下の操作を実行できます。

- データ定義ステートメント (CREATE、ALTER、DROP、GRANT、REVOKE など) の実行
- パラメーター・マーカーのない INSERT、UPDATE、DELETE、MERGE の各ステートメントの実行
- IBM Data Server Driver for JDBC and SQLJ では、CALL ステートメントを実行して、パラメーターがなく結果セットを戻さないストアード・プロシージャを呼び出せます。

これらの SQL ステートメントを実行するには、以下の手順を実行する必要があります。

1. `Connection.createStatement` メソッドを呼び出して、`Statement` オブジェクトを作成します。
2. `Statement.executeUpdate` メソッドを呼び出して、SQL 操作を実行します。
3. `Statement.close` メソッドを呼び出して、`Statement` オブジェクトを閉じます。

以下の SQL ステートメントを実行するとしましょう。

```
UPDATE EMPLOYEE SET PHONENO='4657' WHERE EMPNO='000010'
```

`Statement` オブジェクト `stmt` を作成し、その `UPDATE` ステートメントを実行し、更新された行の数を `numUpd` で返すコードを以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```
Connection con;
Statement stmt;
int numUpd;
...
stmt = con.createStatement();           // Create a Statement object 1
numUpd = stmt.executeUpdate(          // Perform the update 2
    "UPDATE EMPLOYEE SET PHONENO='4657' WHERE EMPNO='000010'");
stmt.close();                          // Close Statement object 3
```

図8. `Statement.executeUpdate` の使用

## PreparedStatement.executeUpdate メソッドを使用した表のデータの更新

DB2 表を定数値で更新する場合は、`Statement.executeUpdate` メソッドを使用できません。ところが、更新時には、変数の値を DB2 表に渡さなければならない場合が多くあります。その場合に使用できるのが、`PreparedStatement.executeUpdate` メソッドです。

入力パラメーターがあり、出力パラメーターがなく、結果セットを返さないストアード・プロシージャであれば、IBM Data Server Driver for JDBC and SQLJ で `PreparedStatement.executeUpdate` を使用して、そのようなストアード・プロシージャを呼び出すこともできます。

DB2 for z/OS は、CALL ステートメントの動的実行をサポートしていません。DB2 for z/OS データ・ソースに格納されているストアード・プロシージャの呼び出しでは、パラメーターとしてパラメーター・マーカーやリテラルを使用できますが、式は使用できません。以下のタイプのリテラルがサポートされています。

- Integer
- Double
- Decimal
- Character
- Hexadecimal
- Graphic



IBM Informix Dynamic Server データ・ソースに格納されているストアード・プロシージャの呼び出しでは、PreparedStatement オブジェクトとして CALL ステートメントまたは EXECUTE PROCEDURE ステートメントを使用できます。

1 つの SQL ステートメントを何度も実行する場合は、SQL ステートメントを PreparedStatement として作成する方がパフォーマンスが良くなります。

例えば、以下の UPDATE ステートメントでは、従業員表の 1 つの電話番号と 1 つの従業員番号だけを更新します。

```
UPDATE EMPLOYEE SET PHONENO='4657' WHERE EMPNO='000010'
```

この操作を一般化して、従業員表の電話番号と従業員番号のあらゆる組み合わせを更新するとしましょう。その場合は、定数の電話番号と従業員番号を変数に置き換える必要があります。

```
UPDATE EMPLOYEE SET PHONENO=? WHERE EMPNO=?
```

この形式の変数のことをパラメーター・マーカーストatementとといいます。パラメーター・マーカーストatementを指定した SQL ステートメントを実行するには、以下の手順を実行する必要があります。

1. Connection.prepareStatement メソッドを呼び出して、PreparedStatement オブジェクトを作成します。
2. PreparedStatement.setXXX メソッドを呼び出して、変数に値を渡します。
3. PreparedStatement.executeUpdate メソッドを呼び出して、変数値で表を更新します。
4. PreparedStatement オブジェクトの作業が完了した時点で、PreparedStatement.close メソッドを呼び出してそのオブジェクトを閉じます。

これまでに取り上げた手順を実行して、従業員番号「000010」の従業員の電話番号を「4657」に変更するコードを以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```
Connection con;
PreparedStatement pstmt;
int numUpd;
...
pstmt = con.prepareStatement(
    "UPDATE EMPLOYEE SET PHONENO=? WHERE EMPNO=?");
// Create a PreparedStatement object 1
pstmt.setString(1,"4657"); // Assign first value to first parameter 2
pstmt.setString(2,"000010"); // Assign first value to second parameter
numUpd = pstmt.executeUpdate(); // Perform first update 3
pstmt.setString(1,"4658"); // Assign second value to first parameter
pstmt.setString(2,"000020"); // Assign second value to second parameter
numUpd = pstmt.executeUpdate(); // Perform second update
pstmt.close(); // Close the PreparedStatement object 4
```

図9. パラメーター・マーカーストatementを指定した SQL ステートメントのために PreparedStatement.executeUpdate を使用する

パラメーター・マーカーストatementのないステートメントの場合も、PreparedStatement.executeUpdate メソッドを使用できます。パラメーター・マーカーストatementのない PreparedStatement オブジェクトを実行する手順は、パラメーター・マーカーストatementを指定した PreparedStatement オブジェクトを実行する手順とよく似ていますが、こ



の場合は、手順 2 (47 ページ) をスキップします。そのための手順を以下の例で示します。

```
Connection con;
PreparedStatement pstmt;
int numUpd;
...
pstmt = con.prepareStatement(
    "UPDATE EMPLOYEE SET PHONENO='4657' WHERE EMPNO='000010'");
// Create a PreparedStatement object
numUpd = pstmt.executeUpdate(); // Perform the update
pstmt.close(); // Close the PreparedStatement object
```

図 10. パラメーター・マーカのない SQL ステートメントのために `PreparedStatement.executeUpdate` を使用する

## DB2 for z/OS サーバーに対する JDBC `executeUpdate` メソッド

JDBC 標準は、`executeUpdate` メソッドが行カウントまたは 0 を戻すことを示していますが、`executeUpdate` メソッドが DB2 for z/OS サーバーに対して実行されると、値 -1 が戻される可能性があります。

DB2 for z/OS サーバーに対する `executeUpdate` ステートメントの場合、戻される値は、実行中の SQL ステートメントのタイプによって異なります。

- INSERT、UPDATE、または DELETE ステートメントなどの更新カウントを持つことができる SQL ステートメントの場合、戻される値は影響を受ける行の数になります。以下ようになります。
  - 正数 (操作によって影響を受ける行の数が正であり、その操作がセグメント表スペースでの一括削除ではない場合)。
  - 0 (操作によって影響を受ける行がない場合)。
  - -1 (操作がセグメント表スペースでの一括削除である場合)。
- SQL CALL ステートメントの場合、データ・ソースは影響を受ける行の数を判別できないため、値 -1 が戻されます。CALL ステートメントに対して `getUpdateCount` または `getMoreResults` を呼び出した場合にも -1 が戻されます。
- その他の SQL ステートメントの場合、値 -1 が戻されます。

## JDBC アプリケーションでのバッチ更新の作成

表の各行を 1 行ずつ更新する代わりにバッチ更新を使用すれば、JDBC によって一群の更新を同時に実行できます。同じバッチ更新に組み込めるステートメントのことをバッチ可能 ステートメントといいます。

入力パラメーターまたはホスト式が含まれているステートメントをバッチに組み込めるのは、そのバッチが同じステートメントの他のインスタンスだけで構成されている場合に限られます。このタイプのバッチのことを同種バッチ といいます。入力パラメーターのないステートメントをバッチに組み込めるのは、そのバッチに含まれている他のステートメントにも入力パラメーターやホスト式がない場合に限られ

ます。このタイプのバッチのことを異種バッチ といいます。2 つのステートメントを同じバッチに組み込める場合、それらのステートメントはバッチ互換 であるといえます。

SQL バッチ更新の作成、実行、除去には、以下の `Statement` メソッドを使用します。

- `addBatch`
- `executeBatch`
- `clearBatch`

以下の `PreparedStatement` メソッドと `CallableStatement` メソッドを使用してパラメーターのバッチを作成すれば、1 つのステートメントをバッチの中で複数回実行し、各実行時に別々のパラメーター・セットを適用する、という操作を実行できます。

- `addBatch`

#### バッチの中でステートメントを実行するときの制限:

- バッチの中で `SELECT` ステートメントを実行しようとする、`BatchUpdateException` がスローされます。
- バッチの中で実行する `CallableStatement` オブジェクトには、出力パラメーターを組み込めます。ただし、出力パラメーターの値を取得することはできません。そうしようすると、`BatchUpdateException` がスローされます。
- バッチの中で実行する `CallableStatement` オブジェクトから `ResultSet` オブジェクトを取得することはできません。`BatchUpdateException` はスローされませんが、`getResultSet` メソッド呼び出しでは `NULL` 値が返されます。

入力パラメーターのない複数のステートメントを使用するバッチ更新を作成するには、以下の基本手順を実行します。

1. バッチの中で実行する各 `SQL` ステートメントごとに、`addBatch` メソッドを呼び出します。
2. `executeBatch` メソッドを呼び出して、ステートメントのバッチを実行します。
3. エラーを確認します。エラーがなければ、以下のようになります。
  - a. `executeBatch` の呼び出しで返された配列から、各 `SQL` ステートメントで更新された行の数を取得します。この数には、トリガーで更新された行や参照整合性の適用によって更新された行は含めません。
  - b. `Connection` オブジェクトの `AutoCommit` が無効になっている場合は、`commit` メソッドを呼び出して変更内容をコミットします。

`Connection` オブジェクトの `AutoCommit` が有効になっている場合は、バッチの最後に `IBM Data Server Driver for JDBC and SQLJ` が `commit` メソッドを追加します。

1 つのステートメントで複数の入力パラメーター・セットを使用するバッチ更新を作成するには、以下の基本手順を実行します。

1. `createStatement` メソッドを呼び出して、`Statement` オブジェクトを作成します。
2. 入力パラメーター値の各セットごとに、以下のようになります。
  - a. `setXXX` メソッドを実行して、入力パラメーターに値を割り当てます。

- b. `addBatch` メソッドを呼び出して、入力パラメーターのセットをバッチに追加します。
3. `executeBatch` メソッドを呼び出して、すべてのパラメーター・セットでステートメントを実行します。
4. エラーを確認します。エラーがなければ、以下のようになります。
  - a. `executeBatch` の呼び出しで返された配列から、SQL ステートメントのそれぞれの実行で更新された行の数を取得します。
  - b. `Connection` オブジェクトの `AutoCommit` が無効になっている場合は、`commit` メソッドを呼び出して変更内容をコミットします。

`Connection` オブジェクトの `AutoCommit` が有効になっている場合は、バッチの最後に IBM Data Server Driver for JDBC and SQLJ が `commit` メソッドを追加します。

以下のコード断片では、2 つのパラメーター・セットをバッチの中に組み込みます。1 つの `UPDATE` ステートメントを 2 つの入力パラメーター・セットのそれぞれに基づいて 2 回実行します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```
try {
...
    PreparedStatement prepStmt = con.prepareStatement(
        "UPDATE DEPT SET MGRNO=? WHERE DEPTNO=?");
    prepStmt.setString(1,mgrnum1);
    prepStmt.setString(2,deptnum1);
    prepStmt.addBatch();

    prepStmt.setString(1,mgrnum2);
    prepStmt.setString(2,deptnum2);
    prepStmt.addBatch();
    int [] numUpdates=prepStmt.executeBatch();
    for (int i=0; i < numUpdates.length; i++) {
        if (numUpdates[i] == SUCCESS_NO_INFO)
            System.out.println("Execution " + i +
                ": unknown number of rows updated");
        else
            System.out.println("Execution " + i +
                "successful: " + numUpdates[i] + " rows updated");
    }
    con.commit();
} catch (BatchUpdateException b) {
    // process BatchUpdateException
}
```

図 11. バッチ更新の実行

## ParameterMetaData メソッドを使用した、PreparedStatement のパラメーターに関する情報の取得

IBM Data Server Driver for JDBC and SQLJ には、`ParameterMetaData` インターフェースのサポートが組み込まれています。`ParameterMetaData` インターフェースには、`PreparedStatement` オブジェクトのパラメーター・マーカーに関する情報を取得するためのメソッドが含まれています。

ParameterMetaData メソッドでは、以下のタイプの情報を確認できます。

- パラメーターのデータ・タイプ (10 進パラメーターの精度やスケールなど)。
- パラメーターのデータベース固有のタイプ名。特殊タイプで定義されている表列に対応するパラメーターの場合は、特殊タイプ名になります。
- パラメーターが NULL 可能かどうか。
- 入力パラメーターか出力パラメーターか。
- 数値パラメーターの値に符号を付けられるかどうか。
- PreparedStatement.setObject がパラメーター値を設定するときに使用する完全修飾 Java クラス名。

ParameterMetaData メソッドを呼び出すには、以下の基本手順を実行する必要があります。

1. Connection.prepareStatement メソッドを呼び出して、PreparedStatement オブジェクトを作成します。
2. PreparedStatement.getParameterMetaData メソッドを呼び出して、ParameterMetaData オブジェクトを取得します。
3. ParameterMetaData.getParameterCount を呼び出して、PreparedStatement に含まれているパラメーターの数を確認します。
4. 個々のパラメーターに対して ParameterMetaData メソッドを呼び出します。

ParameterMetaData メソッドを使用して、SQL の UPDATE ステートメントに含まれているパラメーターの数とデータ・タイプを確認するコードを以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```
Connection con;
ParameterMetaData pmtadta;
int mtadtacnt;
String sqlType;
...
pstmt = con.prepareStatement(
    "UPDATE EMPLOYEE SET PHONENO=? WHERE EMPNO=?");
pmtadta = pstmt.getParameterMetaData(); // Create a PreparedStatement object 1
mtadtacnt = pmtadta.getParameterCount(); // Create a ParameterMetaData object 2
// Determine the number of parameters 3
System.out.println("Number of statement parameters: " + mtadtacnt);
for (int i = 1; i <= mtadtacnt; i++) {
    sqlType = pmtadta.getParameterTypeName(i); // Get SQL type for each parameter 4
    System.out.println("SQL type of parameter " + i + " is " + sqlType);
}
...
pstmt.close(); // Close the PreparedStatement
```

図 12. ParameterMetaData メソッドを使用して PreparedStatement に関する情報を取得する

## JDBC アプリケーションにおけるデータの取り出し

JDBC アプリケーションでは、ResultSet オブジェクトを使用してデータを取り出します。ResultSet は照会の結果セットを表します。

## Statement.executeQuery メソッドを使用して表からデータを取得する

パラメーター・マーカのない SELECT ステートメントを使用して表からデータを取得する場合は、Statement.executeQuery メソッドを使用できます。

このメソッドは、結果表を ResultSet オブジェクトとして返します。結果表を取得したら、ResultSet メソッドを使用して結果表の中を移動し、各行の個々の列値を取得する必要があります。

IBM Data Server Driver for JDBC and SQLJ では、Statement.executeQuery メソッドを使用して、ストアード・プロシージャ呼び出しの結果セットを取得することもできます。ただし、その操作が可能なのは、そのストアード・プロシージャが 1 つの結果セットだけを返す場合に限られます。ストアード・プロシージャが複数の結果セットを返す場合は、Statement.execute メソッドを使用する必要があります。

このトピックでは、最もシンプルな ResultSet、つまり読み取り専用の ResultSet を取り上げます。この結果セットでは、1 行ずつ前方に移動することしかできません。IBM Data Server Driver for JDBC and SQLJ では、更新可能の ResultSet とスクロール可能の ResultSet もサポートされています。

パラメーター・マーカのない SELECT ステートメントを使用して表から行を取得するには、以下の手順を実行する必要があります。

1. Connection.createStatement メソッドを呼び出して、Statement オブジェクトを作成します。
2. Statement.executeQuery メソッドを呼び出して、SELECT ステートメントの結果表を ResultSet オブジェクトとして取得します。
3. ループの中で、next メソッドを使用してカーソルを配置し、getXXX メソッドを使用して、ResultSet オブジェクトの現在行の各列からデータを取得します。XXX は、データ・タイプに相当します。
4. ResultSet.close メソッドを呼び出して、ResultSet オブジェクトを閉じます。
5. Statement オブジェクトの作業が完了した時点で、Statement.close メソッドを呼び出してそのオブジェクトを閉じます。

従業員表からすべての行を取得するコードを以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```

String empNo;
Connection con;
Statement stmt;
ResultSet rs;
...
stmt = con.createStatement(); // Create a Statement object      1
rs = stmt.executeQuery("SELECT EMPNO FROM EMPLOYEE");           2
// Get the result table from the query
while (rs.next()) { // Position the cursor                        3
    empNo = rs.getString(1); // Retrieve only the first column value
    System.out.println("Employee number = " + empNo);
    // Print the column value
}
rs.close(); // Close the ResultSet                               4
stmt.close(); // Close the Statement                             5

```

図 13. `Statement.executeQuery` の使用

## PreparedStatement.executeQuery メソッドを使用した表からのデータの取得

パラメーター・マーカーを指定した SELECT ステートメントを使用して表からデータを取得するには、`PreparedStatement.executeQuery` メソッドを使用します。

このメソッドは、結果表を `ResultSet` オブジェクトとして返します。結果表を取得したら、`ResultSet` メソッドを使用して結果表の中を移動し、各行の個々の列値を取得する必要があります。

IBM Data Server Driver for JDBC and SQLJ では、`PreparedStatement.executeQuery` メソッドを使用して、ストアード・プロシージャ呼び出しの結果セットを取得することもできます。ただし、その操作が可能なのは、そのストアード・プロシージャが 1 つの結果セットだけを返し、入力パラメーターだけを使用する場合に限られます。ストアード・プロシージャが複数の結果セットを返す場合は、`Statement.execute` メソッドを使用する必要があります。詳細については、『JDBC アプリケーションでのストアード・プロシージャからの複数の結果セットの検索』を参照してください。

パラメーター・マーカーのないステートメントの場合も、`PreparedStatement.executeQuery` メソッドを使用できます。1 つの照会を何度も実行する場合は、SQL ステートメントを `PreparedStatement` として作成する方がパフォーマンスが良くなります。

パラメーター・マーカーを指定した SELECT ステートメントを使用して表から行を取得するには、以下の手順を実行する必要があります。

1. `Connection.prepareStatement` メソッドを呼び出して、`PreparedStatement` オブジェクトを作成します。
2. `PreparedStatement.setXXX` メソッドを呼び出して、入力パラメーターに値を渡します。
3. `PreparedStatement.executeQuery` メソッドを呼び出して、SELECT ステートメントの結果表を `ResultSet` オブジェクトとして取得します。



4. ループの中で、`ResultSet.next` メソッドを使用してカーソルを配置し、`getXXX` メソッドを使用して、`ResultSet` オブジェクトの現在行の各列からデータを取得します。
5. `ResultSet.close` メソッドを呼び出して、`ResultSet` オブジェクトを閉じます。
6. `PreparedStatement` オブジェクトの作業が完了した時点で、`PreparedStatement.close` メソッドを呼び出してそのオブジェクトを閉じます。

従業員表から特定の従業員の行を取得するコードを以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```
String empnum, phonenum;
Connection con;
PreparedStatement pstmt;
ResultSet rs;
...
pstmt = con.prepareStatement(
    "SELECT EMPNO, PHONENO FROM EMPLOYEE WHERE EMPNO=?");
pstmt.setString(1,"000010");

rs = pstmt.executeQuery();
while (rs.next()) {
    empnum = rs.getString(1);
    phonenum = rs.getString(2);
    System.out.println("Employee number = " + empnum +
        "Phone number = " + phonenum);
}
rs.close();
pstmt.close();
```

1  
2  
3  
4  
5  
6

図 14. `PreparedStatement.executeQuery` の使用例

## JDBC アプリケーションでのバッチ照会の作成

IBM Data Server Driver for JDBC and SQLJ には、同種バッチでバッチ照会を実行するために、IBM Data Server Driver for JDBC and SQLJ のみの `DB2PreparedStatement` インターフェースが用意されています。

1 つのステートメントで複数の入力パラメーター・セットを使用するバッチ照会を作成するには、以下の基本手順を実行します。

1. `prepareStatement` メソッドを呼び出して、入力パラメーターのある SQL ステートメントの `PreparedStatement` オブジェクトを作成します。
2. 入力パラメーター値の各セットごとに、以下のようになります。
  - a. `PreparedStatement.setXXX` メソッドを実行して、入力パラメーターに値を割り当てます。
  - b. `PreparedStatement.addBatch` メソッドを呼び出して、入力パラメーターのセットをバッチに追加します。
3. `PreparedStatement` オブジェクトを `DB2PreparedStatement` オブジェクトにキャストします。
4. `DB2PreparedStatement.executeBatch` メソッドを呼び出して、すべてのパラメーター・セットでステートメントを実行します。



5. エラーを確認します。

**例:** 以下のコード断片では、2つのパラメーター・セットをバッチの中に組み込みます。1つの SELECT ステートメントを1つの入力パラメーターのそれぞれのパラメーター値に基づいて2回実行します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```
try {
...
    PreparedStatement prepStmt = con.prepareStatement(
        "SELECT EMPNO FROM EMPLOYEE WHERE EMPNO=?");
    prepStmt.setString(1,empnum1);
    prepStmt.addBatch();
    prepStmt.setString(1,empnum2);
    prepStmt.addBatch();
    ((com.ibm.db2.jcc.DB2PreparedStatement)prepStmt).executeDB2QueryBatch();
} catch (BatchUpdateException b) {
    // process BatchUpdateException
}
```

1  
2a  
2b

3,4  
5

### ResultSetMetaData のメソッドを使用した ResultSet の情報入手

表または結果セットの列の数やデータ・タイプがどんな場合にもわかるとは限りません。リモート・データ・ソースからデータを取得する場合は、特にそういえません。

不明な ResultSet を取得するプログラムを作成するときには、ResultSet からデータを取得する前に、ResultSetMetaData メソッドを使用して結果セットの特性を確認する必要があります。

ResultSetMetaData メソッドでは、以下のタイプの情報を確認できます。

- ResultSet の列の数
- ResultSet の基礎表の修飾子
- 列に関する情報 (データ・タイプ、長さ、精度、スケール、NULL 可能など)
- 列が読み取り専用かどうか

executeQuery メソッドを呼び出して、表の照会の ResultSet を生成したら、以下の基本手順を実行して、ResultSet の内容を確認します。

1. ResultSet オブジェクトに対して getMetaData メソッドを呼び出し、ResultSetMetaData オブジェクトを作成します。
2. getColumnCount メソッドを呼び出して、ResultSet の列の数を確認します。
3. ResultSet の各列ごとに ResultSetMetaData メソッドを呼び出して、列の特性を確認します。

ResultSetMetaData.getColumnNames() 呼び出しの結果は、データ・ソースに関して DB2 のカタログに格納されている列名情報を反映しています。

従業員表のすべての列のデータ・タイプを確認するコードを以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```

String s;
Connection con;
Statement stmt;
ResultSet rs;
ResultSetMetaData rsmdta;
int colCount;
int mtadtaint;
int i;
String colName;
String colType;
...
stmt = con.createStatement(); // Create a Statement object
rs = stmt.executeQuery("SELECT * FROM EMPLOYEE");
// Get the ResultSet from the query
rsmdta = rs.getMetaData(); // Create a ResultSetMetaData object 1
colCount = rsmdta.getColumnCount(); // Find number of columns in EMP 2
for (i=1; i<= colCount; i++) {
    colName = rsmdta.getColumnName(); // Get column name
    colType = rsmdta.getColumnTypeName(); // Get column data type
    System.out.println("Column = " + colName +
        " is data type " + colType);
    // Print the column value
}

```

図 15. *ResultSetMetaData* メソッドを使用して *ResultSet* に関する情報を取得する

## IBM Data Server Driver for JDBC and SQLJ 使用時の JDBC *ResultSet* の特性

IBM Data Server Driver for JDBC and SQLJ では、両方向スクロール可能、更新可能、および保持可能なカーソルのサポートが提供されます。

*ResultSet* を順方向に 1 行ずつ移動することに加えて、以下のことも実行することができます。

- 逆方向に移動する、または特定の行へ直接移動する
- *ResultSet* に対して行を更新、削除、または挿入する
- COMMIT 後に *ResultSet* をオープンしたままにしておく

以下の各用語により、*ResultSet* の特性が説明できます。

### スクロール可能性

*ResultSet* のカーソルを順方向のみ、1 つ以上の行を順方向、1 つ以上の行を逆方向、または特定の行に移動できるかどうか。

*ResultSet* のカーソルが両方向にスクロール可能な場合、このカーソルには感度属性もあり、カーソルが基礎表に対する変更にセンシティブであるかどうかが表示されます。

### 更新可能性

カーソルが行の更新または削除に使用できるかどうか。この特性は、ストアード・プロシージャの *ResultSet* が更新できないため、ストアード・プロシージャから返された *ResultSet* には当てはまりません。

IBM Informix Dynamic Server データ・ソースの場合、カーソルは更新可能ではありません。

### 保持可能性

COMMIT 後にカーソルがオープンしたままかどうか。

Connection.prepareStatement または Connection.createStatement メソッド内のパラメーターによって、ResultSet の更新可能性、スクロール可能性、および保持可能性の特性を設定します。ResultSet 設定は、データベース内のカーソルの属性にマップされます。以下の表に、JDBC のスクロール可能性、更新可能性、保持可能性の設定、および対応するカーソルの属性をリストしています。

表 6. JDBC ResultSet の特性および SQL カーソルの属性

JDBC 設定	DB2 カーソル設定	IBM Informix Dynamic Server カーソル設定
CONCUR_READ_ONLY	FOR READ ONLY	FOR READ ONLY
CONCUR_UPDATABLE	FOR UPDATE	FOR UPDATE
HOLD_CURSORS_OVER_COMMIT	WITH HOLD	WITH HOLD
TYPE_FORWARD_ONLY	SCROLL 指定なし	SCROLL 指定なし
TYPE_SCROLL_INSENSITIVE	INSENSITIVE SCROLL	SCROLL
TYPE_SCROLL_SENSITIVE	cursorSensitivity Connection および DataSource プロパティに基づく、SENSITIVE STATIC、SENSITIVE DYNAMIC、または ASENSITIVE	サポートされていない

JDBC の ResultSet が静的である場合は、カーソルがオープンされた後も、結果表のサイズや結果表の行の順序は変わりません。つまり、行を基礎表に挿入した場合も、静的 ResultSet の結果表は変化しません。結果表の行を削除すると削除ホールが生じます。削除ホールを更新または削除することはできません。

### JDBC アプリケーションにおける ResultSets の更新可能性、スクロール可能性、および保持可能性の指定:

ResultSet の更新可能性、スクロール可能性、および保持可能性を指定するには、Connection.prepareStatement または Connection.createStatement メソッド内の特殊なパラメーターを使用します。

デフォルトでは、ResultSet オブジェクトはスクロール可能ではなく、更新可能でもありません。デフォルトの保持可能性は、データ・ソースによって異なります。保持可能性を確認するには、DatabaseMetaData.getResultSetHoldability メソッドを使用します。ResultSet のスクロール可能性、更新可能性、保持可能性の属性を変更するには、以下の手順を実行します。

1. ResultSet を定義する SELECT ステートメントに入力パラメーターが含まれていない場合、createStatement メソッドを呼び出して Statement オブジェクトを作成します。それ以外の場合は、prepareStatement メソッドを呼び出して PreparedStatement オブジェクトを作成します。resultSetType、resultSetConcurrency、または resultSetHoldability パラメーターが含まれる createStatement または prepareStatement メソッドの形式を指定する必要があります。

スクロール可能性および更新可能性をサポートする `createStatement` メソッドの形式は次のようになります。

```
createStatement(int resultSetType, int resultSetConcurrency);
```

スクロール可能性、更新可能性、および保持可能性をサポートする `createStatement` メソッドの形式は次のようになります。

```
createStatement(int resultSetType, int resultSetConcurrency,  
int resultSetHoldability);
```

スクロール可能性および更新可能性をサポートする `prepareStatement` メソッドの形式は次のようになります。

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

スクロール可能性、更新可能性、および保持可能性をサポートする `prepareStatement` メソッドの形式は次のようになります。

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

以下の表には、`resultSetType` および `resultSetConcurrency` の有効値のリストが含まれています。

表7. スクロール可能な `ResultSets` の `resultSetType` および `resultSetConcurrency` の有効な組み合わせ

<code>resultSetType</code> 値	<code>resultSetConcurrency</code> 値
<code>TYPE_FORWARD_ONLY</code>	<code>CONCUR_READ_ONLY</code>
<code>TYPE_FORWARD_ONLY</code>	<code>CONCUR_UPDATABLE</code>
<code>TYPE_SCROLL_INSENSITIVE</code>	<code>CONCUR_READ_ONLY</code>
<code>TYPE_SCROLL_SENSITIVE</code>	<code>CONCUR_READ_ONLY</code>
<code>TYPE_SCROLL_SENSITIVE</code>	<code>CONCUR_UPDATABLE</code>

`resultSetHoldability` には、`HOLD_CURSORS_OVER_COMMIT` および `CLOSE_CURSORS_AT_COMMIT` の 2 つの可能な値が含まれます。これらの値はいずれも、`resultSetConcurrency` および `resultSetHoldability` の有効な組み合わせを使用して指定できます。設定した値は、接続のデフォルトの保持可能性をオーバーライドします。

**制約事項:** `ResultSet` がスクロール可能であり、DB2 Database for Linux, UNIX, and Windows サーバーで `ResultSet` を使用して表から列を選択する場合、`ResultSet` を定義する `SELECT` ステートメントの `SELECT` リストには、以下のデータ・タイプの列を含めることはできません。

- `BLOB`
  - `CLOB`
  - `XML`
  - このリスト内の前述のデータ・タイプのいずれかに基づいた特殊タイプ
2. `SELECT` ステートメントに入力パラメーターが含まれる場合、`setXXX` メソッドを呼び出して値を入力パラメーターに渡します。
  3. `executeQuery` メソッドを呼び出して、`ResultSet` オブジェクトの `SELECT` ステートメントから結果表を取得します。

4. アクセスする各行ごとに以下のようにします。
  - a. 以下の表にリストされているメソッドのいずれかを使用してカーソルを置きます。

表 8. 両方向スクロール・カーソルを置くための *ResultSet* メソッド

メソッド	カーソルの位置
first	<i>ResultSet</i> の最初の行
last	<i>ResultSet</i> の最後の行
next <sup>1</sup>	<i>ResultSet</i> の次の行
previous <sup>2</sup>	<i>ResultSet</i> の前の行
absolute(int <i>n</i> ) <sup>3</sup>	<i>n</i> >0 の場合は <i>ResultSet</i> の <i>n</i> 行。 <i>n</i> <0 であり、 <i>m</i> が <i>ResultSet</i> の行数である場合は、 <i>ResultSet</i> の <i>m+n+1</i> 行。
relative(int <i>n</i> ) <sup>4,5</sup>	<i>n</i> >0 の場合、現在行の <i>n</i> 行後ろにある行。 <i>n</i> <0 の場合、現在行の <i>n</i> 行前にある行。 <i>n</i> =0 の場合は現在行。
afterLast	<i>ResultSet</i> の最後の行の後ろ
beforeFirst	<i>ResultSet</i> の最初の行の前

注:

1. カーソルが *ResultSet* の最初の行の前にある場合、このメソッドではカーソルは最初の行に置かれます。
2. カーソルが *ResultSet* の最後の行の後ろにある場合、このメソッドではカーソルは最後の行に置かれます。
3. このメソッドでは、*n* の絶対値が結果セットの行数よりも大きい場合、*n* が正であれば最後の行の後ろ、*n* が負であれば最初の行の前にカーソルが置かれます。
4. このメソッドを使用する前に、カーソルは *ResultSet* の有効な行を示している必要があります。カーソルが最初の行の前あるいは最後の行の後ろにある場合、メソッドは *SQLException* をスローします。
5. *m* が *ResultSet* の行数、そして *x* が *ResultSet* の現在行の番号だとします。 *n*>0 であり、*x+n>m* である場合、ドライバーは最後の行の後ろにカーソルを置きます。 *n*<0 であり、*x+n<1* である場合、ドライバーは最初の行の前にカーソルを置きます。

- b. 現行カーソル位置を知る必要がある場合には、`getRow`、`isFirst`、`isLast`、`isBeforeFirst`、または `isAfterLast` メソッドを使用してこの情報を取得します。
- c. ステップ 1 (57 ページ) で、`TYPE_SCROLL_SENSITIVE` の `resultSetType` 値を指定しており、現在行の最新の値を参照する必要がある場合には、`refreshRow` メソッドを呼び出します。

**推奨:** *ResultSet* の行をリフレッシュするとご使用のアプリケーションのパフォーマンスに悪影響を及ぼす可能性があるため、最新のデータを参照する必要がある場合にのみ、`refreshRow` を呼び出してください。

- d. 以下の操作の 1 つ以上を実行します。
  - *ResultSet* オブジェクトの現在行の各列からデータを取り出すには、`getXXX` メソッドを使用します。

- 基礎表から現在行を更新するには、updateXXX メソッドを使用して列値を ResultSet の現在行に割り当てます。次に、updateRow を使用して基礎表の対応する行を更新します。基礎表を更新しない場合は、updateRow メソッドの代わりに cancelRowUpdates メソッドを呼び出します。

これらのメソッドを使用するには、ResultSet の *resultSetConcurrency* 値は CONCUR\_UPDATABLE でなければなりません。

- 基礎表から現在行を削除するには、deleteRow メソッドを使用します。deleteRow を呼び出すと、ドライバーは ResultSet の現在行をホールに置き換えます。

このメソッドを使用するには、ResultSet の *resultSetConcurrency* 値は CONCUR\_UPDATABLE でなければなりません。

5. close メソッドを呼び出して ResultSet オブジェクトをクローズします。
6. close メソッドを呼び出して Statement または PreparedStatement オブジェクトをクローズします。

以下のコードは、従業員表からすべての行を逆順で取り出し、従業員番号 "000010" の電話番号を更新する方法について説明しています。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```
String s;
Connection con;
Statement stmt;
ResultSet rs;
...
stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                            ResultSet.CONCUR_UPDATABLE);           1
                            // Create a Statement object
                            // for a scrollable, updatable
                            // ResultSet
rs = stmt.executeQuery("SELECT EMPNO FROM EMPLOYEE FOR UPDATE OF PHONENO");
rs.afterLast();           3
                            // Create the ResultSet
                            // Position the cursor at the end of
                            // the ResultSet           4a
while (rs.previous()) {   // Position the cursor backward
    s = rs.getString("EMPNO"); // Retrieve the employee number 4d
                                // (column 1 in the result
                                // table)
    System.out.println("Employee number = " + s);
                                // Print the column value
    if (s.compareTo("000010") == 0) { // Look for employee 000010
        rs.updateString("PHONENO","4657"); // Update their phone number
        rs.updateRow(); // Update the row
    }
}
rs.close(); // Close the ResultSet           5
stmt.close(); // Close the Statement           6
```

図 16. 両方向スクロール・カーソルの使用

JDBC アプリケーションで ResultSet の現在行が削除ホールまたは更新ホールになっているかどうかをテストする:



ResultSet に TYPE\_SCROLL\_SENSITIVE 属性があり、基礎になっているカーソルが SENSITIVE STATIC の場合は、ResultSet の行を取得する前に、削除ホールまたは更新ホールがあるかどうかをテストする必要があります。

SENSITIVE STATIC の ResultSet を開いた後も、サイズ変更はありません。削除された行は、プレースホルダーに置き換えられているからです。そのような行のことをホールといいます。更新された行が ResultSet の基準に適合しなくなった場合も、ホールになります。ホールになった行を取得することはできません。

ResultSet の現在行が削除ホールまたは更新ホールになっているかどうかをテストするには、以下の手順を実行します。

1. DatabaseMetaData.deletesAreDetected メソッドまたは DatabaseMetaData.updatesAreDetected メソッドに TYPE\_SCROLL\_SENSITIVE 引数を指定して呼び出し、データ・ソースが TYPE\_SCROLL\_SENSITIVE の ResultSet でホールを作成するかどうかを確認します。
2. DatabaseMetaData.deletesAreDetected または DatabaseMetaData.updatesAreDetected から true が返された場合は、データ・ソースがホールを作成できるという意味なので、ResultSet.rowDeleted メソッドまたは ResultSet.rowUpdated メソッドを呼び出して、現在行が削除ホールまたは更新ホールになっているかどうかを確認します。メソッドから true が返された場合は、現在行がホールになっています。

現在行が削除ホールになっているかどうかをテストするコードを以下に示します。

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
// Create a Statement object
// for a scrollable, updatable
// ResultSet

ResultSet rs =
    stmt.executeQuery("SELECT EMPNO FROM EMPLOYEE FOR UPDATE OF PHONENO");
// Create the ResultSet

DatabaseMetaData dbmd = con.getMetaData();
// Create the DatabaseMetaData object

boolean dbSeesDeletes =
    dbmd.deletesAreDetected(ResultSet.TYPE_SCROLL_SENSITIVE);
// Can the database see delete holes?

rs.afterLast();
// Position the cursor at the end of
// the ResultSet

while (rs.previous()) {
    // Position the cursor backward
    if (dbSeesDeletes) {
        // If delete holes can be detected
        if (!(rs.rowDeleted()))
            // If this row is not a delete hole
            {
                s = rs.getString("EMPNO");
                // Retrieve the employee number
                System.out.println("Employee number = " + s);
                // Print the column value
            }
    }
}

rs.close();
// Close the ResultSet
stmt.close();
// Close the Statement
```

### JDBC アプリケーションで ResultSet に行を挿入する:

ResultSet の *resultSetConcurrency* 属性が CONCUR\_UPDATABLE の場合は、ResultSet に行を挿入できます。

ResultSet に行を挿入するには、以下の手順を実行します。



1. 挿入する行それぞれに対して、以下のステップを実行します。
  - a. `ResultSet.moveToInsertRow` メソッドを呼び出して、挿入する行を作成します。行は、`ResultSet` の外にあるバッファに作成されます。

既存の挿入バッファがある場合は、バッファから以前の値すべてがクリアされます。

- b. `ResultSet.updateXXX` メソッドを呼び出して、挿入する行に値を割り当てます。

`ResultSet` 内の少なくとも 1 つの列に値を割り当てる必要があります。そうしない場合、その行が `ResultSet` に挿入されるときに、`SQLException` がスローされます。

`ResultSet` の列に値を割り当てない場合、基礎表の更新時にデータ・ソースはその関連表の列にデフォルト値を挿入します。

`NOT NULL` として定義されている列に `NULL` 値を割り当てた場合、`JDBC` ドライバーは `SQLException` をスローします。

- c. `ResultSet.insertRow` を呼び出して、その行を `ResultSet` に挿入します。

`ResultSet.insertRow` を呼び出すと、`ResultSet.insertRow` が失敗したとしても、挿入バッファから常にすべての値がクリアされます。

2. `ResultSet` 内にカーソルを位置変更します。

カーソルを挿入行から `ResultSet` に移動するには、

`ResultSet.first`、`ResultSet.absolute`、または `ResultSet.relative` といった、カーソルを特定の行に配置する任意のメソッドを呼び出します。あるいは、

`ResultSet.moveToCurrentRow` を呼び出して、`ResultSet` 内の、挿入操作が実行される前に現在行であった行にカーソルを移動できます。

`ResultSet.moveToCurrentRow` の呼び出し後には、すべての値が挿入バッファからクリアされます。

**例:** 以下のコードは、サンプル `DEPARTMENT` 表にあるすべての行から成る `ResultSet` に 1 つの行を挿入する例を示します。行を挿入した後、このコードは、挿入操作前にカーソルが置かれていた `ResultSet` 内の位置にカーソルを配置します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```
stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,  
                            ResultSet.CONCUR_UPDATABLE);  
ResultSet rs = stmt.executeQuery("SELECT * FROM DEPARTMENT");  
rs.moveToInsertRow();  
rs.updateString("DEPT_NO", "M13");  
rs.updateString("DEPTNAME", "TECHNICAL SUPPORT");  
rs.updateString("MGRNO", "000010");  
rs.updateString("ADMRDEPT", "A00");  
rs.insertRow();  
rs.moveToCurrentRow();
```

1a  
1b

1c  
2

**JDBC アプリケーションで現在行が `ResultSet` に挿入されたかどうかをテストする**  
:

ResultSet が動的であれば、行を挿入できます。行を ResultSet に挿入した後に、どの行が挿入されたかを確認しなければならない場合もあります。

現在行が ResultSet に挿入されたかどうかをテストするには、以下の手順を実行します。

1. DatabaseMetaData.ownInsertsAreVisible メソッドと DatabaseMetaData.othersInsertsAreVisible メソッドを呼び出して、このタイプの ResultSet で挿入内容を参照できるかどうかを確認します。
2. この ResultSet で挿入内容を参照できるようであれば、 DatabaseMetaData.insertsAreDetected メソッドを呼び出して、このタイプの ResultSet で挿入内容を検出できるかどうかを確認します。
3. この ResultSet で挿入内容を検出できるようであれば、ResultSet.rowInserted メソッドを呼び出して、現在行が挿入されたかどうかを確認します。

## IBM Data Server Driver for JDBC and SQLJ を使用した複数行の SQL 操作

IBM Data Server Driver for JDBC and SQLJ では、複数行の INSERT、UPDATE、およびこれらの操作をサポートするデータ・ソースへの接続用の FETCH がサポートされます。

### 複数行 INSERT

複数の INSERT ステートメントをバッチで実行する場合、データ・ソースで複数行 INSERT がサポートされる場合は、IBM Data Server Driver for JDBC and SQLJ では、複数行 INSERT が使用されて行が挿入されます。複数行 INSERT では、個別の INSERT ステートメントの場合よりも良好なパフォーマンスを提供できます。

複数行 INSERT ステートメントを JDBC アプリケーション内に組み込んで、複数行の挿入操作を実行することはできません。

### 複数行 FETCH

両方向スクロール・カーソルを使用して FETCH を JDBC アプリケーション内で実行する場合、useRowsetCursor プロパティ値を true に設定したか、または設定していない場合は、IBM Data Server Driver for JDBC and SQLJ では、データ・ソースで複数行 FETCH がサポートされていれば、複数行 FETCH が使用されて行が取得されます。複数行 FETCH では、個別の FETCH ステートメントの場合よりも良好なパフォーマンスを提供できます。

### 複数行の位置指定 UPDATE または位置指定 DELETE

IBM Data Server Driver for JDBC and SQLJ では、JDBC 1 標準に従った位置指定の更新または削除操作を実行するための手法がサポートされます。この手法には、ResultSet.getCursorName メソッドを使用して、ResultSet のカーソルの名前を取得して、以下の形式の位置指定 UPDATE ステートメントまたは位置指定 DELETE ステートメントを定義することが含まれます。

```
UPDATE table SET col1=value1,...coln=valueN WHERE CURRENT OF cursorname  
DELETE FROM table WHERE CURRENT OF cursorname
```

JDBC 1 手法を使用して、複数行 FETCH をサポートするデータ・ソースでデータを更新または削除すると、単一行を更新または削除しようとしても、位置指定 UPDATE または DELETE ステートメントが複数の行を更新または削除してしまう場合があります。予期しない更新または削除が実行されないようにするために、下記の処置のいずれかを実行することができます。

- 更新可能な ResultSet を使用して、一度に 1 行を取得して更新する。
- UPDATE または DELETE ステートメントで FOR ROW *n* OF ROWSET 節を使用して、変更または削除する特定の行を指定する。

## JDBC アプリケーションでのストアード・プロシージャの呼び出し

ストアード・プロシージャを呼び出すには、CallableStatement クラスのメソッドを呼び出します。

ストアード・プロシージャを呼び出すための基本手順は、以下のとおりです。

1. CALL ステートメントを引数として Connection.prepareCall メソッドを呼び出し、CallableStatement オブジェクトを作成します。

**制約事項:** 使用できるパラメーター・タイプは、データ・ソースが CALL ステートメントの動的実行をサポートしているかどうかによって異なります。DB2 for z/OS は、CALL ステートメントの動的実行をサポートしていません。DB2 for z/OS データベース・サーバーに格納されているストアード・プロシージャの呼び出しでは、パラメーターとしてパラメーター・マーカーやリテラルを使用できますが、式は使用できません。サポートされているリテラルのタイプとそれに対応する JDBC タイプを以下の表にまとめます。

表 9. DB2 for z/OS のストアード・プロシージャ呼び出しのパラメーターでサポートされているリテラル・タイプ

リテラル・パラメーター・タイプ	JDBC タイプ	例
整数	java.sql.Types.INTEGER	-122, 40022, +27
浮動小数点 10 進数	java.sql.Types.DOUBLE	23E12, 40022E-4, +2723E+15, 1E+23, 0E0
固定小数点 10 進数	java.sql.Types.DECIMAL	-23.12, 40022.4295, 0.0, +2723.23, 10000000000
文字	java.sql.Types.VARCHAR	'Grantham Lutz', 'O', 'Conner', 'ABcde?z?'
16 進数	java.sql.Types.VARBINARY	X'C1C30427', X'00CF18E0'
Unicode ストリング	java.sql.Types.VARCHAR	UX'0041', UX'0054006500730074'

2. CallableStatement.setXXX メソッドを呼び出して、入力パラメーター (CREATE PROCEDURE ステートメント内で IN として定義されているパラメーター) に値を渡します。

**制約事項:** データ・ソースが CALL ステートメントの動的実行をサポートしていない場合は、CALL ステートメントの入力パラメーターのデータ・タイプをストアード・プロシージャ定義で指定されているとおりに指定する必要があります。

3. CallableStatement.registerOutParameter メソッドを呼び出して、各パラメーターが出力専用パラメーター (CREATE PROCEDURE ステートメント内で OUT とし

て定義されているパラメーター) か、入出力パラメーター (CREATE PROCEDURE ステートメント内で INOUT として定義されているパラメーター) かを指定します。

**制約事項:** データ・ソースが CALL ステートメントの動的実行をサポートしていない場合は、CALL ステートメントの出力専用パラメーターまたは入出力パラメーターのデータ・タイプをストアード・プロシージャ定義で指定されているとおりに指定する必要があります。

- 以下のいずれかのメソッドを呼び出して、ストアード・プロシージャを呼び出します。

#### **CallableStatement.executeUpdate**

ストアード・プロシージャが結果セットを返さない場合は、このメソッドを呼び出します。

#### **CallableStatement.executeQuery**

ストアード・プロシージャが 1 つの結果セットを返す場合は、このメソッドを呼び出します。

#### **CallableStatement.execute**

ストアード・プロシージャが複数の結果セットを返す場合や、結果セットの数が不明な場合は、このメソッドを呼び出します。

**制約事項:** IBM Informix Dynamic Server (IDS) データ・ソースは、複数の結果セットをサポートしていません。

- ストアード・プロシージャが複数の結果セットを返す場合は、それらの結果セットを取得します。

**制約事項:** IDS データ・ソースは、複数の結果セットをサポートしていません。

- CallableStatement.getXXX メソッドを呼び出して、OUT パラメーターまたは INOUT パラメーターから値を取得します。
- CallableStatement オブジェクトの作業が完了した時点で、CallableStatement.close メソッドを呼び出してそのオブジェクトを閉じます。

**例:** 1 つの入力パラメーター、4 つの出力パラメーターがあり、ResultSet を返さないストアード・プロシージャを呼び出すコード例を以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```
int ifcaret;  
int ifcareas;  
int xsbytes;  
String errbuff;  
Connection con;  
CallableStatement cstmt;  
ResultSet rs;  
...  
cstmt = con.prepareCall("CALL DSN8.DSN8ED2(?,?,?,?)");  
// Create a CallableStatement object 1  
cstmt.setString (1, "DISPLAY THREAD(*)");  
// Set input parameter (DB2 command) 2  
cstmt.registerOutParameter (2, Types.INTEGER);  
// Register output parameters 3  
cstmt.registerOutParameter (3, Types.INTEGER);  
cstmt.registerOutParameter (4, Types.INTEGER);  
cstmt.registerOutParameter (5, Types.VARCHAR);
```

```

cstmt.executeUpdate();           // Call the stored procedure
ifcaret = cstmt.getInt(2);       // Get the output parameter values
ifcareas = cstmt.getInt(3);
xsbytes = cstmt.getInt(4);
errbuff = cstmt.getString(5);
cstmt.close();

```

4  
6

7

## JDBC アプリケーションでのストアード・プロシージャからの複数の結果セットの検索

結果セットを返すストアード・プロシージャを呼び出す場合は、結果セットを取得するためのコードを組み込む必要があります。

実行する手順は、返される結果セットの数や内容がわかっているかどうかによって異なります。

### JDBC アプリケーションでストアード・プロシージャから既知の数の結果セットを取得する:

ストアード・プロシージャから既知の数の結果セットを取得する手順は、結果セットの数が不明な場合の手順よりもシンプルになります。

結果セットの数と内容がわかっている場合に結果セットを取得するには、以下の手順を実行します。

1. `Statement.execute` メソッド、`PreparedStatement.execute` メソッド、`CallableStatement.execute` メソッドのいずれかを呼び出して、ストアード・プロシージャを呼び出します。

ストアード・プロシージャに入力パラメーターがある場合は、`PreparedStatement.execute` を使用します。

2. `getResultSet` メソッドを呼び出して、`ResultSet` オブジェクトに含まれている最初の結果セットを取得します。
3. ループの中で、`next` メソッドを使用してカーソルを配置し、`getXXX` メソッドを使用して、`ResultSet` オブジェクトの現在行の各列からデータを取得します。
4. 結果セットが  $n$  個ある場合は、以下の手順を  $n-1$  回繰り返します。
  - a. `getMoreResults` メソッドを呼び出して、現在の結果セットを閉じ、次の結果セットを参照します。
  - b. `getResultSet` メソッドを呼び出して、`ResultSet` オブジェクトに含まれている次の結果セットを取得します。
  - c. ループの中で、`next` メソッドを使用してカーソルを配置し、`getXXX` メソッドを使用して、`ResultSet` オブジェクトの現在行の各列からデータを取得します。

**例:** 以下のコード例では、2 つの結果セットを取得します。最初の結果セットには `INTEGER` 列が含まれていて、2 番目の結果セットには `CHAR` 列が含まれています。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```

CallableStatement cstmt;
ResultSet rs;
int i;
String s;
...

```

```

cstmt.execute(); // Call the stored procedure 1
rs = cstmt.getResultSet(); // Get the first result set 2
while (rs.next()) { // Position the cursor 3
    i = rs.getInt(1); // Retrieve current result set value
    System.out.println("Value from first result set = " + i);
    // Print the value
}
cstmt.getMoreResults(); // Point to the second result set 4a
// and close the first result set
rs = cstmt.getResultSet(); // Get the second result set 4b
while (rs.next()) { // Position the cursor 4c
    s = rs.getString(1); // Retrieve current result set value
    System.out.println("Value from second result set = " + s);
    // Print the value
}
rs.close(); // Close the result set
cstmt.close(); // Close the statement

```

### JDBC アプリケーションでストアード・プロシージャから不明な数の結果セットを取得する:

ストアード・プロシージャから不明な数の結果セットを取得する手順は、結果セットの数がわかっている場合の手順よりも複雑になります。

結果セットの数と内容がわからない場合に結果セットを取得するには、`ResultSet` が返されなくなるまで `ResultSet` を取得する必要があります。それぞれの `ResultSet` ごとに `ResultSetMetaData` メソッドを使用すれば、結果セットの内容を確認できます。

ストアード・プロシージャを呼び出した後に、以下の基本手順を実行して、不明な数の結果セットの内容を取得します。

1. ストアード・プロシージャを呼び出した `execute` ステートメントから返された値をチェックします。

戻り値が `true` であれば、少なくとも 1 つの結果セットが存在しているので、次の手順に進む必要があります。

2. ループの中で以下の手順を繰り返します。
  - a. `getResultSet` メソッドを呼び出して、`ResultSet` オブジェクトに含まれている結果セットを取得します。このメソッドを呼び出すと、前の結果セットが閉じます。
  - b. `ResultSetMetaData` メソッドを使用して、`ResultSet` の内容を確認し、`ResultSet` からデータを取得します。
  - c. `getMoreResults` メソッドを呼び出して、さらに結果セットがあるかどうかを確認します。`getMoreResults` から `true` が返された場合は、手順 1 に進んで次の結果セットを取得します。

**例:** 結果セットの数と内容がわからない場合に結果セットを取得するためのコードを以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```

CallableStatement cstmt;
ResultSet rs;
...
boolean resultsAvailable = cstmt.execute(); // Call the stored procedure
while (resultsAvailable) { // Test for result sets 1
    ResultSet rs = cstmt.getResultSet(); // Get a result set 2a
    ... // Process the ResultSet
}

```



```

// as you would process
// a ResultSet from a table
resultsAvailable = pstmt.getMoreResults(); // Check for next result set 2c
// (Also closes the
// previous result set)
}

```

**JDBC アプリケーションでストアド・プロシージャから複数の結果セットを取得するときに結果セットを開いたままにしておく:**

`getMoreResults` メソッドには、次の `ResultSet` を開くときに現在の `ResultSet` を開いたままにしておくための形式があります。

結果セットを開いたままにするかどうかを指定するには、以下の手順を実行します。

`getMoreResults` を呼び出して、次の `ResultSet` をチェックするときは、以下の形式を使用します。

```
CallableStatement.getMoreResults(int current);
```

- 次の `ResultSet` をチェックするときに現在の `ResultSet` を開いたままにしておくには、`current` に `Statement.KEEP_CURRENT_RESULT` の値を指定します。
- 次の `ResultSet` をチェックするときに現在の `ResultSet` を閉じるには、`current` に `Statement.CLOSE_CURRENT_RESULT` の値を指定します。
- すべての `ResultSet` オブジェクトを閉じるには、`current` に `Statement.CLOSE_ALL_RESULTS` の値を指定します。

**例:** 最後の `ResultSet` を取得するまですべての `ResultSet` を開いたままにしておき、その後すべての `ResultSet` を閉じるコードを以下に示します。

```

CallableStatement pstmt;
...
boolean resultsAvailable = pstmt.execute(); // Call the stored procedure
if (resultsAvailable==true) { // Test for result set
    ResultSet rs1 = pstmt.getResultSet(); // Get a result set
    ... // Process ResultSet
    resultsAvailable = pstmt.getMoreResults(Statement.KEEP_CURRENT_RESULT);
    // Check for next result set
    // but do not close
    // previous result set
    if (resultsAvailable==true) { // Test for another result set
        ResultSet rs2 = pstmt.getResultSet(); // Get next result set
        ... // Process either ResultSet
    }
}
resultsAvailable = pstmt.getMoreResults(Statement.CLOSE_ALL_RESULTS);
// Close the result sets

```

## IBM Data Server Driver for JDBC and SQLJ 使用時の JDBC アプリケーションでの LOB

IBM Data Server Driver for JDBC and SQLJ は、表の BLOB、CLOB、DBCLOB 列のデータを更新および検索したり、BLOB パラメーターまたは CLOB パラメーターを指定してストアド・プロシージャやユーザー定義関数を呼び出したりするためのメソッドをサポートしています。



## IBM Data Server Driver for JDBC and SQLJ での連続ストリーミング

データ・ソースにより連続ストリーミングがサポートされる場合は、IBM Data Server Driver for JDBC and SQLJ で連続ストリーミングを使用して LOB または XML 列内のデータを検索できます。

DB2 for z/OS バージョン 9.1 以降では、LOB および XML オブジェクトの連続ストリーミングがサポートされます。DB2 Database for Linux, UNIX, and Windows バージョン 9.5 以降では、LOB の連続ストリーミングがサポートされます。

連続ストリーミングを使用すると、LOB または XML オブジェクトのサイズに基づいて、LOB または XML のデータを戻すのに最も効率のよいモードがデータ・ソースで動的に判別されます。

IBM Data Server Driver for JDBC and SQLJ バージョン 3.50 以降については、連続ストリーミングの動作は、DB2 Database for Linux, UNIX, and Windows バージョン 9.5 以降への接続の場合の LOB 検索のデフォルトです。

連続ストリーミングは IBM Data Server Driver for JDBC and SQLJ バージョン 3.1 以降でサポートされています。ただし、IBM Data Server Driver for JDBC and SQLJ バージョン 3.2 以降では、連続ストリーミングの動作は、DB2 for z/OS バージョン 9.1 以降への接続の場合の LOB および XML 検索のデフォルトです。

IBM Data Server Driver for JDBC and SQLJ の `progressiveStreaming` プロパティが `DB2BaseDataSource.YES` の場合は、アプリケーションでは連続ストリーミングの動作が使用されます。

連続ストリーミングを使用可能に設定すると、JDBC ドライバーが `streamBufferSize` プロパティで LOB をいつマテリアライズするかをコントロールできます。LOB または XML オブジェクトが `streamBufferSize` 値以下であるとき、オブジェクトはマテリアライズされます。

**重要:** 連続ストリーミングでは、LOB または XML の値を `ResultSet` からアプリケーション変数に取り出す場合、カーソルを移動するまで、または `ResultSet` 上のカーソルをクローズするまでは、そのアプリケーション変数の内容を操作できます。その後は、アプリケーション変数の内容は操作できなくなります。アプリケーション変数内の LOB に対して何らかのアクションを実行すると、`SQLException` を受け取ります。例えば、連続ストリーミングが有効で、以下のようなステートメントを実行するとします。

```
...
ResultSet rs = stmt.executeQuery("SELECT CLOBCOL FROM MY_TABLE");
rs.next(); // Retrieve the first row of the ResultSet
Clob clobFromRow1 = rs.getClob(1); // Put the CLOB from the first column of
// the first row in an application variable
String substr1Clob = clobFromRow1.getSubString(1,50);
// Retrieve the first 50 bytes of the CLOB
rs.next(); // Move the cursor to the next row.
// clobFromRow1 is no longer available.
// String substr2Clob = clobFromRow1.getSubString(51,100);
// This statement would yield an SQLException
Clob clobFromRow2 = rs.getClob(1); // Put the CLOB from the first column of
```

```
rs.close(); // the second row in an application variable
           // Close the ResultSet.
           // clobFromRow2 is also no longer available.
```

rs.next() を実行してカーソルを ResultSet の 2 番目の行に置いた後は、clobFromRow1 内の CLOB 値は使用できなくなります。同様に、rs.close() を実行して ResultSet をクローズした後、clobFromRow1 および clobFromRow2 内の値は使用できなくなります。

連続ストリーミングを無効として設定した場合、IBM Data Server Driver for JDBC and SQLJ が LOB を扱う方法は、fullyMaterializeLobData プロパティの値によって決まります。

連続ストリーミングの使用は、LOB または XML データ検索に推奨される方式です。

## IBM Data Server Driver for JDBC and SQLJ での LOB ロケータ

IBM Data Server Driver for JDBC and SQLJ では、LOB ロケータを使用して LOB 列のデータを検索できます。LOB ロケータは、データ・ソースが連続ストリーミングをサポートしない場合にのみ、使用してください。

JDBC で LOB ロケータを使用して、LOB 列からデータを検索するには、fullyMaterializeLobData プロパティを false に設定し、progressiveStreaming プロパティを NO (アプリケーション・プログラム内では DB2BaseDataSource.NO) に設定する必要があります。

fullyMaterializeLobData の効果は、次のように、データ・ソースが連続ロケータをサポートするかどうかによって異なります。

- データ・ソースが連続ロケータをサポートしない場合:

fullyMaterializeLobData の値が true の場合、行が取り出されるときに LOB データは JDBC ドライバー内で完全にマテリアライズされます。この値が false の場合、LOB データはストリームされます。ドライバーは内部でロケータを使用し、必要に応じて LOB データをチャンクごとに検索します。大量のデータを収容している LOB を検索する場合、この値を false に設定するように強くお勧めします。デフォルトは true です。

- データ・ソースが連続ストリーミングをサポートする場合:

progressiveStreaming プロパティが YES (アプリケーション・プログラム内では DB2BaseDataSource.YES) に設定されているか、または設定されていない場合、JDBC ドライバーは fullyMaterializeLobData の値を無視します。

fullyMaterializeLobData は、ストアド・プロシージャ・パラメータには影響を与えません。

他の言語の場合と同様、Java アプリケーションにおける LOB ロケータは、単一のデータ・ソースと関連付けられています。単一の LOB ロケータを使用して、2 つの異なるデータ・ソースの間でデータを移動させることはできません。LOB データを 2 つのデータ・ソース間で移動するには、最初のデータ・ソースにある表か

ら LOB データを検索するときそれをマテリアライズし、次いでそのデータを 2 番目のデータ・ソースにある表に挿入する必要があります。

## IBM Data Server Driver for JDBC and SQLJ 使用時の LOB 操作

IBM Data Server Driver for JDBC and SQLJ は、表の BLOB、CLOB、DBCLOB 列のデータを更新および検索したり、 BLOB パラメーターまたは CLOB パラメーターを指定してストアド・プロシージャやユーザー定義関数を呼び出したりするためのメソッドをサポートしています。

IBM Data Server Driver for JDBC and SQLJ で LOB データに関して実行できる操作は次のとおりです。

- BLOB 列または CLOB 列からデータを検索するには、BLOB 列または CLOB 列を以下の `ResultSet` メソッドの引数として指定します。

BLOB 列の場合:

- `getBinaryStream`
- `getBytes`

CLOB 列の場合:

- `getAsciiStream`
- `getCharacterStream`
- `getString`

- 更新可能な `ResultSet` で BLOB 列または CLOB 列を更新するには、以下の `ResultSet` メソッドを呼び出します。

BLOB 列の場合:

- `updateBinaryStream`
- `updateBlob`

CLOB 列の場合:

- `updateAsciiStream`
- `updateCharacterStream`
- `updateClob`

上にリストしたいずれかのメソッドで `length` パラメーターに `-1` を指定すると、IBM Data Server Driver for JDBC and SQLJ は、入力データがなくなるまでデータを読み取ります。

- 以下の `PreparedStatement` メソッドを使用すると、 BLOB 列または CLOB 列に対応するパラメーターの値が設定されます。

BLOB 列の場合:

- `setBytes`
- `setBinaryStream`
- `setObject` (*Object* パラメーターの値は `InputStream`)

CLOB 列の場合:

- `setString`
- `setAsciiStream`
- `setCharacterStream`

- setObject (*Object* パラメーターの値は Reader)

*length* に -1 を指定すると、IBM Data Server Driver for JDBC and SQLJ は、入力データがなくなるまでデータを読み取ります。

- CallableStatement.getString メソッドを使用して、JDBC CLOB パラメーターの値を検索します。

**制約事項:** IBM Data Server Driver for JDBC and SQLJ Type 2 接続では、DBCLOB OUT または INOUT パラメーターを持つストアード・プロシージャを呼び出すことはできません。

IBM Data Server Driver for JDBC and SQLJ バージョン 4.0 以降を使用する場合は、このほかに以下の操作を実行できます。

- ResultSet.updateXXX メソッドまたは PreparedStatement.setXXX メソッドで、BLOB または CLOB を更新します。*length* 値に指定できるのは、最大で 2 GB です。例えば、BLOB には以下のメソッドが定義されます。

```
ResultSet.updateBlob(int columnIndex, InputStream x, long length)
ResultSet.updateBlob(String columnLabel, InputStream x, long length)
ResultSet.updateBinaryStream(int columnIndex, InputStream x, long length)
ResultSet.updateBinaryStream(String columnLabel, InputStream x, long length)
PreparedStatement.setBlob(int columnIndex, InputStream x, long length)
PreparedStatement.setBlob(String columnLabel, InputStream x, long length)
PreparedStatement.setBinaryStream(int columnIndex, InputStream x, long length)
PreparedStatement.setBinaryStream(String columnLabel, InputStream x, long length)
```

- BLOB または CLOB の更新時に *length* パラメーターを指定せずに ResultSet.updateXXX メソッドまたは PreparedStatement.setXXX メソッドを実行すると、IBM Data Server Driver for JDBC and SQLJ は入力データがなくなるまでデータを読み取ります。以下に例を示します。

```
ResultSet.updateBlob(int columnIndex, InputStream x)
ResultSet.updateBlob(String columnLabel, InputStream x)
ResultSet.updateBinaryStream(int columnIndex, InputStream x)
ResultSet.updateBinaryStream(String columnLabel, InputStream x)
PreparedStatement.setBlob(int columnIndex, InputStream x)
PreparedStatement.setBlob(String columnLabel, InputStream x)
PreparedStatement.setBinaryStream(int columnIndex, InputStream x)
PreparedStatement.setBinaryStream(String columnLabel, InputStream x)
```

- Connection.createBlob メソッドまたは Connection.createClob メソッドを使用して、データを含まない Blob オブジェクトまたは Clob オブジェクトを作成します。
- Blob.getBinaryStream メソッドまたは Clob.getCharacterStream メソッドを使用して、連続ストリーミングまたはロケーターの使用中に、クライアント上の Blob オブジェクトまたは Clob オブジェクトをマテリアライズします。
- Blob.free メソッドまたは Clob.free メソッドで、Blob オブジェクトまたは Clob オブジェクトが保持するリソースを解放します。

## JDBC アプリケーションでの LOB 列データの検索または更新のための Java データ・タイプ

JDBC ドライバーで、LOB 列とともに使用されるパラメーターのデータ・タイプがすぐに判別できない場合は、LOB データ・タイプと互換性のあるパラメーターのデータ・タイプを選択する必要があります。

deferPrepares プロパティが true に設定され、IBM Data Server Driver for JDBC and SQLJ により PreparedStatement.setXXX 呼び出しが処理される場合、ドライバーにデータ・タイプの判別のための追加的な処理が必要になることがあります。この追加処理は、パフォーマンスに影響を与えるおそれがあります。

## BLOB 列の入力パラメーター

BLOB 列の入力パラメーター、または BLOB 列への入力に使用される入出力パラメーターの場合、以下の技法のいずれかを使用できます。

- 次のように、BLOB 列と完全に一致する java.sql.Blob 入力変数を使用します。

```
cstmt.setBlob(parmIndex, blobData);
```

- 次のように、ターゲット・データ・タイプとして BLOB を指定する CallableStatement.setObject 呼び出しを使用します。

```
byte[] byteData = {(byte)0x1a, (byte)0x2b, (byte)0x3c};  
cstmt.setObject(parmInd, byteData, java.sql.Types.BLOB);
```

- CallableStatement.setBinaryStream 呼び出しで、java.io.ByteArrayInputStream タイプの入力パラメーターを使用します。 java.io.ByteArrayInputStream オブジェクトは、BLOB データ・タイプと互換性があります。この呼び出しの場合、入力データの正確な長さを指定する必要があります。

```
java.io.ByteArrayInputStream byteStream =  
    new java.io.ByteArrayInputStream(byteData);  
int numBytes = byteData.length;  
cstmt.setBinaryStream(parmIndex, byteStream, numBytes);
```

## BLOB 列の出力パラメーター

BLOB 列の出力パラメーター、または BLOB 列からの出力に使用される入出力パラメーターの場合、次の技法を使用できます。

- CallableStatement.registerOutParameter 呼び出しを使用して、出力パラメーターを BLOB タイプに指定します。これにより、パラメーター値を BLOB データ・タイプと互換性のあるデータ・タイプのすべての変数に取り出すことができます。例えば、次のコードでは BLOB 値が byte[] 変数に取り出されます。

```
cstmt.registerOutParameter(parmIndex, java.sql.Types.BLOB);  
cstmt.execute();  
byte[] byteData = cstmt.getBytes(parmIndex);
```

## CLOB 列の入力パラメーター

CLOB 列の入力パラメーター、または CLOB 列への入力に使用される入出力パラメーターの場合、以下の技法のいずれかを使用できます。

- 次のように、CLOB 列と完全に一致する java.sql.Clob 入力変数を使用します。

```
cstmt.setClob(parmIndex, clobData);
```

- 次のように、ターゲット・データ・タイプとして CLOB を指定する CallableStatement.setObject 呼び出しを使用します。

```
String charData = "CharacterString";  
cstmt.setObject(parmInd, charData, java.sql.Types.CLOB);
```

- 以下のいずれかのタイプのストリーム入力パラメーターを使用します。
  - cstmt.setCharacterStream 呼び出しでの java.io.StringReader 入力パラメーター:

```
java.io.StringReader reader = new java.io.StringReader(charData);
cstmt.setCharacterStream(parmIndex, reader, charData.length);
```

- `cstmt.setAsciiStream` 呼び出しでの `java.io.ByteArrayInputStream` パラメーター (ASCII データ用):

```
byte[] charDataBytes = charData.getBytes("US-ASCII");
java.io.ByteArrayInputStream byteStream =
    new java.io.ByteArrayInputStream (charDataBytes);
cstmt.setAsciiStream(parmIndex, byteStream, charDataBytes.length);
```

これらの呼び出しでは、入力データの正確な長さを指定する必要があります。

- 次のように、`cstmt.setString` 呼び出しで `String` の入力パラメーターを使用します。

```
cstmt.setString(charData);
```

データの長さが 32 KB より大きく、JDBC ドライバーにこのパラメーターのデータ・タイプに関する `DESCRIBE` 情報がない場合は、JDBC ドライバーにより `CLOB` データ・タイプが入力データに割り当てられます。

- 次のように、`cstmt.setObject` 呼び出しで `String` の入力パラメーターを使用し、ターゲット・データ・タイプを `VARCHAR` または `LONGVARCHAR` で指定します。

```
cstmt.setObject(parmIndex, charData, java.sql.Types.VARCHAR);
```

データの長さが 32 KB より大きく、JDBC ドライバーにこのパラメーターのデータ・タイプに関する `DESCRIBE` 情報がない場合は、JDBC ドライバーにより `CLOB` データ・タイプが入力データに割り当てられます。

## CLOB 列の出力パラメーター

CLOB 列の出力パラメーター、または CLOB 列からの出力に使用される入出力パラメーターの場合、以下の技法のいずれかを使用できます。

- `CallableStatement.registerOutParameter` 呼び出しを使用して、出力パラメーターを `CLOB` タイプに指定します。これにより、パラメーター値を `CLOB` データ・タイプと互換性のあるデータ・タイプのすべての変数に取り出すことができます。例えば、次のコードでは `CLOB` 値が `String` 変数に取り出されます。

```
cstmt.registerOutParameter(parmIndex, java.sql.Types.CLOB);
cstmt.execute();
String charData = cstmt.getString(parmIndex);
```

- `CallableStatement.registerOutParameter` 呼び出しを使用して、出力パラメーターを `VARCHAR` または `LONGVARCHAR` タイプに指定します。

```
cstmt.registerOutParameter(parmIndex, java.sql.Types.VARCHAR);
cstmt.execute();
String charData = cstmt.getString(parmIndex);
```

この技法は、取り出されるデータの長さが 32 KB 以下であることが分かっている場合にのみ使用してください。32 KB を超える場合、データは切り捨てられます。



## IBM Data Server Driver for JDBC and SQLJ 使用時の JDBC での ROWID

DB2 for z/OS、DB2 for i5/OS、および IBM Informix Dynamic Server (IDS) では、データベース表内の列に対して ROWID データ・タイプがサポートされます。ROWID とは、表内の行を一意的に識別する値です。

以下の ResultSet メソッドを使用して、データを ROWID 列から取得することができます。

- `getRowId` (JDBC 4.0 以降)
- `getBytes`
- `getObject`

次の ResultSet メソッドを使用して、更新可能な ResultSet の ROWID 列を更新することができます。

- `updateRowId` (JDBC 4.0 以降)

`updateRowId` は、ターゲット・データベース・システムで ROWID 列の更新がサポートされる場合にのみ有効です。

JDBC 3.0 を使用している場合、`getObject` では、IBM Data Server Driver for JDBC and SQLJ により、IBM Data Server Driver for JDBC and SQLJ のみのクラスである `com.ibm.db2.jcc.DB2RowID` のインスタンスが返されます。

JDBC 4.0 を使用している場合、`getObject` では、IBM Data Server Driver for JDBC and SQLJ により、クラス `java.sql.RowId` のインスタンスが返されます。

以下の PreparedStatement メソッドを使用して、ROWID 列に関連付けられたパラメーターの値を設定することができます。

- `setRowId` (JDBC 4.0 以降)
- `setBytes`
- `setObject`

JDBC 3.0 を使用している場合、`setObject` では、パラメーターのターゲット・タイプとして、IBM Data Server Driver for JDBC and SQLJ のみのタイプである `com.ibm.db2.jcc.Types.ROWID` または `com.ibm.db2.jcc.DB2RowID` クラスのインスタンスを使用します。

JDBC 4.0 を使用している場合、`setObject` では、パラメーターのターゲット・タイプとして、タイプ `java.sql.Types.RowId` または `java.sql.ROWID` クラスのインスタンスを使用します。

以下の CallableStatement メソッドを使用して、ROWID 列をストアド・プロシージャ呼び出しからの出力パラメーターとして取得することができます。

- `getRowId` (JDBC 4.0 以降)
- `getObject`

ROWID 出力パラメーターで定義されたストアド・プロシージャを呼び出すには、そのパラメーターを `java.sql.Types.ROWID` タイプで登録します。



ROWID 値が有効な期間は、それらの ROWID 値が定義されたデータ・ソースによって異なります。 DatabaseMetaData.getRowIdLifetime メソッドを使用して、ROWID 値が有効な期間を判別します。データ・ソースに返される値が、以下の表にリストされています。

表 10. サポートされるデータ・ソースの DatabaseMetaData.getRowIdLifetime 値

データベース・サーバー	DatabaseMetaData.getRowIdLifetime
DB2 for z/OS	ROWID_VALID_TRANSACTION
DB2 Database for Linux, UNIX, and Windows	ROWID_UNSUPPORTED
DB2 for i5/OS	ROWID_VALID_FOREVER
IDS	ROWID_VALID_FOREVER

例: `PreparedStatement.setRowId` の `java.sql.RowId` ターゲット・タイプでの使用: `rowid` が `RowId` オブジェクトであると仮定します。パラメーター 1 を設定するには、次の形式の `setRowId` メソッドを使用します。

```
ps.setRowId(1, rid);
```

例: `PreparedStatement.setObject` の `com.ibm.db2.jcc.DB2RowID` ターゲット・タイプでの使用: `rowid` が `com.ibm.db2.jcc.DB2RowID` のインスタンスであると仮定します。パラメーター 1 を設定するには、次の形式の `setObject` メソッドを使用します。

```
ps.setObject (1, rowid);
```

例: ROWID 値をデータ・ソースから取得するための `ResultSet.getRowId` の使用: ROWID 値を結果セットの最初の列から `RowId` オブジェクトの `rowid` に取り出すには、次の形式の `ResultSet.getRowId` メソッドを使用します。

```
java.sql.RowId rowid = rs.getRowId(1);
```

例: `CallableStatement.registerOutParameter` の `java.sql.Types.ROWID` パラメーター・タイプでの使用: CALL ステートメントのパラメーター 1 を `java.sql.Types.ROWID` データ・タイプで登録するには、次の形式の `registerOutParameter` メソッドを使用します。

```
cs.registerOutParameter(1, java.sql.Types.ROWID)
```

## JDBC アプリケーションでの特殊タイプ

特殊タイプとは、組み込みの SQL データ・タイプとして内部的に表されるユーザー定義のデータ・タイプのことです。特殊タイプは、SQL ステートメントの `CREATE DISTINCT TYPE` を実行して作成します。

JDBC プログラムでは、特殊タイプは、`executeUpdate` メソッドを使用して `CREATE DISTINCT TYPE` ステートメントを実行することで作成することができます。また、`executeUpdate` を使用して、このタイプの列が含まれた表を作成することもできます。このタイプの列からデータを取得するか、またはこのタイプの列を更新する場合は、特殊タイプのベースとなる組み込みタイプに対応するデータ・タイプと共に、Java ID を使用します。

以下の例では、`INTEGER` タイプをベースにした特殊タイプを作成し、このタイプの列を持つ表を作成して、行をその表に挿入し、さらにその表から行を取得します。

```

Connection con;
Statement stmt;
ResultSet rs;
String empNumVar;
int shoeSizeVar;
...
stmt = con.createStatement();           // Create a Statement object
stmt.executeUpdate(
    "CREATE DISTINCT TYPE SHOESIZE AS INTEGER");
// Create distinct type
stmt.executeUpdate(
    "CREATE TABLE EMP_SHOE (EMPNO CHAR(6), EMP_SHOE_SIZE SHOESIZE)");
// Create table with distinct type
stmt.executeUpdate("INSERT INTO EMP_SHOE " +
    "VALUES ('000010', 6)");           // Insert a row
rs=stmt.executeQuery("SELECT EMPNO, EMP_SHOE_SIZE FROM EMP_SHOE");
// Create ResultSet for query
while (rs.next()) {
    empNumVar = rs.getString(1);       // Get employee number
    shoeSizeVar = rs.getInt(2);       // Get shoe size (use int
// because underlying type
// of SHOESIZE is INTEGER)
    System.out.println("Employee number = " + empNumVar +
        " Shoe size = " + shoeSizeVar);
}
rs.close();                           // Close ResultSet
stmt.close();                          // Close Statement

```

図 17. 特殊タイプの作成および使用

## JDBC アプリケーションでの ARRAY パラメーターを使用したストアード・プロシージャの呼び出し

IBM Data Server Driver for JDBC and SQLJ で実行される JDBC アプリケーションでは、ARRAY パラメーターを持つストアード・プロシージャを呼び出すことができます。

CallableStatement.setArray メソッドまたは CallableStatement setObject メソッドを使用して、java.sql.Array オブジェクトを ARRAY ストアード・プロシージャのパラメーターに割り当てることができます。

CallableStatement.registerOutParameter 呼び出しでパラメーター・タイプとして java.sql.Types.ARRAY を指定することによって、ストアード・プロシージャ呼び出し用の出力の ARRAY パラメーターを登録することができます。

ARRAY 出力パラメーターからデータを取り出すには、次の 2 つの方法があります。

- CallableStatement.getArray メソッドを使用して、データを java.sql.Array オブジェクトに取り出し、java.sql.Array.getArray メソッドを使用して、java.sql.Array オブジェクトの内容を Java 配列に取り出します。
- CallableStatement.getArray メソッドを使用して、データを java.sql.Array オブジェクトに取り出します。java.sql.Array.getResultSet() メソッドを使用して、データを

ResultSet オブジェクトに取り出します。ResultSet メソッドを使用して、配列のエレメントを取り出します。ResultSet の各行には、次の 2 つの列が含まれています。

- 配列の指標 (1 から始まります)
- 配列エレメント

**例:** ストアド・プロシージャ GET\_EMP\_DATA の入力パラメーター IN\_PHONE と出力パラメーター OUT\_PHONE が以下のように定義される配列だとします。

```
CREATE TYPE PHONENUMBERS AS VARCHAR(10) ARRAY[5]
```

GET\_EMP\_DATA をこの 2 つのパラメーターで呼び出します。

```
Connection con;
CallableStatement cstmt;
ResultSet rs;
java.sql.Array inPhoneData;
...
stmt = con.prepareCall("CALL GET_EMP_DATA(?,?)");
cstmt.setObject (1, inPhoneData); // Create a CallableStatement object
cstmt.registerOutParameter (2, java.sql.Types.ARRAY); // Set input parameter
// Register out parameters
cstmt.executeUpdate(); // Call the stored procedure
Array outPhoneData = cstmt.getArray(2);
// Get the output parameter array
System.out.println("Parameter values from GET_EMP_DATA call: ");
String [] outPhoneNums = (String [])outPhoneData.getArray();
// Retrieve output data from the JDBC Array object
// into a Java String array
for(int i=0; i<outPhoneNums.length; i++) {
    System.out.print(outPhoneNums[i]);
    System.out.println();
}
```

## JDBC アプリケーションでのセーブポイント

SQL セーブポイントは、作業単位内の特定の時点のデータおよびスキーマの状態を表します。セーブポイントの設定、セーブポイントの解放、およびデータとスキーマのセーブポイントによって表された状態へのリストアを行うための SQL ステートメントがあります。

IBM Data Server Driver for JDBC and SQLJ では、セーブポイントを使用するための以下のメソッドがサポートされます。

### **Connection.setSavepoint() または Connection.setSavepoint(String name)**

セーブポイントが設定されます。これらのメソッドにより、この後の releaseSavepoint または rollback 操作で使用される Savepoint オブジェクトが返されます。

これらのメソッドのいずれかを実行すると、DB2 により ON ROLLBACK RETAIN<sup>®</sup> CURSORS などの SAVEPOINT ステートメントの形式が実行されます。

### **Connection.releaseSavepoint(Savepoint savepoint)**

指定したセーブポイント、およびそれ以後に確立されたすべてのセーブポイントが解放されます。

### **Connection.rollback(Savepoint savepoint)**

作業が指定したセーブポイントまでロールバックされます。

### **DatabaseMetaData.supportsSavepoints()**

データ・ソースでセーブポイントがサポートされているかどうかを示されます。

以下の例は、セーブポイントの設定、セーブポイントへのロールバック、およびセーブポイントの解除の方法を示しています。

```
Connection con;
Statement stmt;
ResultSet rs;
String empNumVar;
int shoeSizeVar;
...
con.setAutoCommit(false);           // set autocommit OFF
stmt = con.createStatement();        // Create a Statement object
...                                  // Perform some SQL
con.commit();                         // Commit the transaction
stmt.executeUpdate("INSERT INTO EMP_SHOE " +
    "VALUES ('000010', 6)");          // Insert a row
Savepoint savept = con.setSavepoint(); // Create a savepoint
...
stmt.executeUpdate("INSERT INTO EMP_SHOE " +
    "VALUES ('000020', 10)");         // Insert another row
conn.rollback(savept);                // Roll back work to the point
                                        // after the first insert
...
con.releaseSavepoint(savept);         // Release the savepoint
stmt.close();                          // Close the Statement
```

図 18. JDBC アプリケーションでのセーブポイントの設定、ロールバック、および解除

## **JDBC アプリケーションでの自動生成キーの取得**

IBM Data Server Driver for JDBC and SQLJ を使用すれば、JDBC 3.0 のメソッドによって自動生成キーを表から取得できます。

ID 列は、データ・ソースが各行の数値を自動生成するための手段になる表列です。CREATE TABLE ステートメントまたは ALTER TABLE ステートメントで ID 列を定義するには、スケール 0 の厳密な数値タイプ (スケール 0 の SMALLINT、INTEGER、BIGINT、DECIMAL、またはそのいずれかのタイプに基づく特殊タイプ) の列を定義するときに AS IDENTITY 節を指定します。

表から自動生成キーを取得する操作を有効にするには、行を挿入するとき、自動生成キー値を取得できるように設定する必要があります。そのためには、Connection.prepareStatement、Statement.executeUpdate、Statement.execute のいずれかのメソッド呼び出しで 1 つのフラグを設定します。実行するステートメントは、INSERT ステートメント、または SELECT ステートメント内の INSERT でなければなりません。それ以外の場合は、JDBC ドライバーがそのフラグを設定するためのパラメーターを無視します。

**制約事項:** 自動生成キーの取得のために SQL ステートメントを準備して、その PreparedStatement オブジェクトをバッチ更新で使用することはできません。IBM Data Server Driver for JDBC and SQLJ バージョン 3.50 以降では、自動生成キーを

取得するために準備した `PreparedStatement` オブジェクトに対して `addBatch` メソッドまたは `executeBatch` メソッドを呼び出すと、`SQLException` がスローされます。

JDBC 3.0 のメソッドを使用して表から自動生成キーを取得するには、以下の手順を実行する必要があります。

1. 以下のいずれかのメソッドを使用して、自動生成キーを返すための設定を行います。

- `PreparedStatement.executeUpdate` メソッドを使用して行を挿入する計画がある場合は、以下のいずれかの形式の `Connection.prepareStatement` メソッドを呼び出して、`PreparedStatement` オブジェクトを作成します。

ID 列をサポートしているデータ・ソースの表であれば、以下の形式が有効です。`sql-statement` は、単一行の `INSERT` ステートメントでなければなりません。

```
Connection.prepareStatement(sql-statement,  
    Statement.RETURN_GENERATED_KEYS);
```

以下の形式は、データ・ソースが ID 列をサポートしていて、`SELECT` ステートメントの中で `INSERT` を使用する場合にのみ有効です。`sql-statement` は、単一行の `INSERT` ステートメントでも複数行の `INSERT` ステートメントでもかまいません。最初の形式の場合は、自動生成キーを取得する列の名前を指定します。2 番目の形式の場合は、自動生成キーを取得する列の、表の中の位置を指定します。

```
Connection.prepareStatement(sql-statement, String [] columnNames);  
Connection.prepareStatement(sql-statement, int [] columnIndexes);
```

- `Statement.executeUpdate` メソッドを使用して行を挿入する場合は、以下のいずれかの形式の `Statement.executeUpdate` メソッドを呼び出します。

ID 列をサポートしているデータ・ソースの表であれば、以下の形式が有効です。`sql-statement` は、単一行の `INSERT` ステートメントでなければなりません。

```
Statement.executeUpdate(sql-statement, Statement.RETURN_GENERATED_KEYS);
```

以下の形式は、データ・ソースが ID 列をサポートしていて、`SELECT` ステートメントの中で `INSERT` を使用する場合にのみ有効です。`sql-statement` は、単一行の `INSERT` ステートメントでも複数行の `INSERT` ステートメントでもかまいません。最初の形式の場合は、自動生成キーを取得する列の名前を指定します。2 番目の形式の場合は、自動生成キーを取得する列の、表の中の位置を指定します。

```
Statement.executeUpdate(sql-statement, String [] columnNames);  
Statement.executeUpdate(sql-statement, int [] columnIndexes);
```

2. `PreparedStatement.getGeneratedKeys` メソッドまたは `Statement.getGeneratedKeys` メソッドを呼び出して、自動生成キー値が含まれている `ResultSet` オブジェクトを取得します。

`ResultSet` に含まれている自動生成キーのデータ・タイプは、対応する列のデータ・タイプにかかわらず `DECIMAL` になります。

ID 列が含まれている表を作成し、その表に 1 つの行を挿入し、ID 列の自動生成キー値を取得するコードを以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

図 19. 自動生成キーを取得する例

```
import java.sql.*;
import java.math.*;
import com.ibm.db2.jcc.*;

Connection con;
Statement stmt;
ResultSet rs;
java.math.BigDecimal idColVar;
...
stmt = con.createStatement();           // Create a Statement object

stmt.executeUpdate(
    "CREATE TABLE EMP_PHONE (EMPNO CHAR(6), PHONENO CHAR(4), " +
    "IDENTCOL INTEGER GENERATED ALWAYS AS IDENTITY)");
// Create table with identity column
stmt.executeUpdate("INSERT INTO EMP_PHONE (EMPNO, PHONENO) " + // 1
    "VALUES ('000010', '5555')", // Insert a row
    Statement.RETURN_GENERATED_KEYS); // Indicate you want automatically
// generated keys
rs = stmt.getGeneratedKeys(); // Retrieve the automatically // 2
// generated key value in a ResultSet.
// Only one row is returned.
// Create ResultSet for query

while (rs.next()) {
    java.math.BigDecimal idColVar = rs.getBigDecimal(1);
// Get automatically generated key
// value
    System.out.println("automatically generated key value = " + idColVar);
}
rs.close(); // Close ResultSet
stmt.close(); // Close Statement
```

ID 列が含まれている表を作成し、複数行の INSERT ステートメントを使用してその表に 2 つの行を挿入し、ID 列の自動生成キー値を取得するコードを以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

図 20. 複数行の INSERT を実行してから自動生成キーを取得する例

```
import java.sql.*;
import java.math.*;
import com.ibm.db2.jcc.*;

Connection con;
Statement stmt;
ResultSet rs;
...
stmt = con.createStatement();

stmt.executeUpdate(
    "CREATE TABLE EMP_PHONE (EMPNO CHAR(6), PHONENO CHAR(4), " +
    "IDENTCOL INTEGER GENERATED ALWAYS AS IDENTITY)");
// Create table with identity column
String[] id_col = {"IDENTCOL"};
int updateCount = // 1
```

```

stmt.executeUpdate("INSERT INTO EMP_PHONE (EMPNO, PHONENO)" +
"VALUES ('000010', '5555'), ('000020', '5556')", id_col);
// Insert two rows
// Indicate you want automatically
// generated keys
rs = stmt.getGeneratedKeys(); // Retrieve the automatically generated key values in a ResultSet.
// Two rows are returned.
// Create ResultSet for query

while (rs.next()) {
int idColVar = rs.getInt(1); // Get automatically generated key
// values
System.out.println("automatically generated key value = " + idColVar);
}
stmt.close();
con.close();

```

## IBM Data Server Driver for JDBC and SQLJ のみのメソッド によって拡張クライアント情報をデータ・ソースに提供する

一連の IBM Data Server Driver for JDBC and SQLJ のみのメソッドによって、クライアントに関する追加情報をサーバーに提供できます。その情報は、アカウントティング、ワークロード管理、デバッグに使用できます。

アプリケーションが SQL を実行するなどして、データベース・サーバーにアクセスする操作を実行すると、拡張クライアント情報がデータベース・サーバーに送信されます。

IBM Data Server Driver for JDBC and SQLJ バージョン 4.0 では、IBM Data Server Driver for JDBC and SQLJ のみのメソッドは非推奨になっています。代わりに `java.sql.Connection.setClientInfo` を使用してください。

IBM Data Server Driver for JDBC and SQLJ のみのメソッドを以下の表にまとめます。

表 11. DB2 サーバーにクライアント情報を提供するメソッド

メソッド	提供できる情報
<code>setDB2ClientAccountingInformation</code>	アカウント情報
<code>setDB2ClientApplicationInformation</code>	接続を使用するアプリケーションの名前
<code>setDB2ClientDebugInfo</code>	統合デバッガーの CLIENT DEBUGINFO 接続属性
<code>setDB2ClientProgramId</code>	特定の SQL ステートメントに関連するプログラムを呼び出し元が識別するために役立つ呼び出し元指定ストリング
<code>setDB2ClientUser</code>	接続のためのユーザー名
<code>setDB2ClientWorkstation</code>	接続のためのクライアント・ワークステーション名

拡張クライアント情報を設定するには、以下の手順を実行します。

1. `Connection` を作成します。
2. `java.sql.Connection` オブジェクトを `com.ibm.db2.jcc.DB2Connection` にキャストします。



3. 82 ページの表 11 にまとめられているいずれかのメソッドを呼び出します。
4. SQL ステートメントを実行して、情報を DB2 サーバーに送信します。

これまでに取り上げた手順を実行して、DB2 サーバーにユーザー名とワークステーション名を渡すコードを以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```
public class ClientInfoTest {
    public static void main(String[] args) {
        String url = "jdbc:db2://sysmvs1.st1.ibm.com:5021/san_jose";
        try {
            Class.forName("com.ibm.db2.jcc.DB2Driver");
            String user = "db2adm";
            String password = "db2adm";
            Connection conn = DriverManager.getConnection(url,           1
                user, password);
            if (conn instanceof DB2Connection) {
                DB2Connection db2conn = (DB2Connection) conn;           2
                db2conn.setDB2ClientUser("Michael L Thompson");         3
                db2conn.setDB2ClientWorkstation("sjwkstn1");
                // Execute SQL to force extended client information to be sent
                // to the server
                conn.prepareStatement("SELECT * FROM SYSIBM.SYSDUMMY1"
                    + "WHERE 0 = 1").executeQuery();                     4
            }
        } catch (Throwable e) {
            e.printStackTrace();
        }
    }
}
```

図 21. DB2 サーバーに拡張クライアント情報を渡す例

## クライアント情報プロパティによって拡張クライアント情報をデータ・ソースに提供する

IBM Data Server Driver for JDBC and SQLJ バージョン 4.0 では、JDBC 4.0 のクライアント情報プロパティがサポートされています。それらのプロパティを使用して、クライアントに関する追加情報をサーバーに提供できます。その情報は、アカウントिंग、ワークロード管理、デバッグに使用できます。

アプリケーションが SQL を実行するなどして、データベース・サーバーにアクセスする操作を実行すると、拡張クライアント情報がデータベース・サーバーに送信されます。

アプリケーションでは、`Connection.getClientInfo` メソッドを使用して、データベース・サーバーからクライアント情報を取得したり、`DatabaseMetaData.getClientInfoProperties` メソッドを実行して、ドライバーがサポートしているクライアント情報を確認したりすることもできます。

IBM Data Server Driver for JDBC and SQLJ のみのメソッドは非推奨になっているので、代わりに JDBC 4.0 のクライアント情報プロパティを使用してください。

クライアント情報プロパティを設定するには、以下の手順を実行します。

1. Connection を作成します。
2. `java.sql.setClientInfo` メソッドを呼び出して、データベース・サーバーがサポートしているクライアント情報プロパティを設定します。
3. SQL ステートメントを実行して、情報をデータベース・サーバーに送信します。

これまでに取り上げた手順を実行して、DB2 サーバーにユーザー名とワークステーション名を渡すコードを以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```
public class ClientInfoTest {
    public static void main(String[] args) {
        String url = "jdbc:db2://sysmvs1.st1.ibm.com:5021/san_jose";
        try {
            Class.forName("com.ibm.db2.jcc.DB2Driver");
            String user = "db2adm";
            String password = "db2adm";
            Connection conn = DriverManager.getConnection(url,      1
                user, password);
            conn.setClientInfo("ClientUser", "Michael L Thompson"); 2
            conn.setClientInfo("ClientWorkstation", "sjwkstn1");
            // Execute SQL to force extended client information to be sent
            // to the server
            conn.prepareStatement("SELECT * FROM SYSIBM.SYSDUMMY1"
                + "WHERE 0 = 1").executeQuery();      3
        } catch (Throwable e) {
            e.printStackTrace();
        }
    }
}
```

図 22. DB2 サーバーに拡張クライアント情報を渡す例

## IBM Data Server Driver for JDBC and SQLJ によりサポートされるクライアント情報プロパティ

JDBC 4.0 には、データ・ソースへの接続に関する情報が入っている、クライアント情報プロパティが組み込まれています。 `DatabaseMetaData.getClientInfoProperties` メソッドは、IBM Data Server Driver for JDBC and SQLJ がサポートするクライアント情報プロパティのリストを返します。

`DatabaseMetaData.getClientInfoProperties` を呼び出すと、以下の列を含む結果セットが返されます。

- NAME
- MAX\_LEN
- DEFAULT\_VALUE
- DESCRIPTION

以下の表では、IBM Data Server Driver for JDBC and SQLJ が DB2 Database for Linux, UNIX, and Windows に対して返すクライアント情報プロパティ値をリストしています。

表 12. DB2 Database for Linux, UNIX, and Windows のクライアント情報プロパティ値

NAME	MAX_LEN	DEFAULT_VALUE	DESCRIPTION
ApplicationName	255	空ストリング	現在接続を使用しているアプリケーションの名前。
ClientAccountingInformation	255	空ストリング	接続に指定されているクライアント情報からの会計情報ストリングの値。
ClientHostname	255	DB2Connection.setDB2ClientWorkstation により設定される値。この値が設定されない場合、デフォルトはローカル・ホストのホスト名になります。	接続を使用しているアプリケーションが実行しているコンピューターのホスト名。
ClientUser	255	空ストリング	接続を使用しているアプリケーションを実行しているユーザーの名前。

以下の表では、IBM Data Server Driver for JDBC and SQLJ が DB2 for z/OS に対して戻すクライアント情報プロパティ値をリストしています。

表 13. DB2 for z/OS のクライアント情報プロパティ値

NAME	MAX_LEN	Type 4 接続の DEFAULT_VALUE	Type 2 接続の DEFAULT_VALUE	DESCRIPTION
ApplicationName	32	設定されている場合、clientProgramName プロパティ値。そうでない場合、「db2jcc_application」。	空ストリング	現在接続を使用しているアプリケーションの名前。
ClientAccountingInformation	200	以下の値の連結であるストリング。 <ul style="list-style-type: none"> <li>「JCCnnnnn」。ここで nnnnn はドライバー・レベルです (例えば 04000 など)。</li> <li>DB2Connection.setDB2ClientWorkstation により設定される値。この値が設定されない場合、デフォルトはローカル・ホストのホスト名になります。</li> <li>設定されている場合、applicationName プロパティ値。そうでない場合、20 のブランク。</li> <li>設定されている場合、clientUser プロパティ値。そうでない場合、8 つのブランク。</li> </ul>	空ストリング	接続に指定されているクライアント情報からの会計情報ストリングの値。
ClientHostname	18	DB2Connection.setDB2ClientWorkstation により設定される値。この値が設定されない場合、デフォルトはローカル・ホストのホスト名になります。	空ストリング	接続を使用しているアプリケーションが実行しているコンピューターのホスト名。
ClientUser	16	DB2Connection.setDB2ClientUser により設定される値。この値が設定されない場合、デフォルトはデータベースの接続に使用される現在のユーザー ID になります。	空ストリング	接続を使用しているアプリケーションを実行しているユーザーの名前。

## JDBC アプリケーションでのオプティミスティック・ロック

データ・ソースのオプティミスティック・ロックを利用する JDBC アプリケーションを作成することができます。

オプティミスティック・ロックとは、アプリケーションで SELECT 操作と UPDATE または DELETE 操作との間のロック解除に使用できる手法のことです。選択した行がそのアプリケーションによって更新または削除される前に変更された場合は、UPDATE または DELETE 操作は失敗します。オプティミスティック・ロックによって、指定されたリソースが他のトランザクションによって使用できない時間が最小限に抑えられます。

通常、アプリケーションでは以下のステップが実行されて、オプティミスティック・ロックが使用されます。

1. 行を表から選択する。
2. 表に対してロック解除する。
3. 選択した行が変更されていない場合は、更新する。

行が変更されたかどうかを検査するために、アプリケーションにより行の変更トークンが照会されます。行の変更トークンは、必ずしも行が変更されたかどうかの完全に正確な指標であるわけではありません。行の変更トークンは、行の変更タイム・スタンプ列を持つ表を作成した場合は完全に正確です。行の変更タイム・スタンプ列なしで表を作成するか、または表を変更して行の変更タイム・スタンプ列を追加した場合は、行の変更トークンに、行に対する更新が正確に反映されていないおそれがあります。つまり、行の変更トークンにより、行が変更されていないにもかかわらず行が変更されたと示されることがあります。この状態を *false negative* 状態と呼びます。

オプティミスティック・ロックを実行するための JDBC アプリケーションを作成する場合は、以下の同様の手順に従います。

1. 照会を準備して実行する。

オプティミスティック・ロック情報が必要かどうか、およびその情報に *false negative* を含めることができるかどうかを指示します。

2. `ResultSet` にオプティミスティック・ロック情報が含まれるかどうか、およびその情報で *false negative* の生成が可能かどうかを判別する。

オプティミスティック・ロック情報のタイプに基づいて、オプティミスティック・ロックを続行するかどうかを判定することができます。

3. 表に対してロック解除する。
4. 選択した行が行の変更トークンによって変更されていないと示される場合は、更新する。

以下のコードでは、JDBC アプリケーションでのオプティミスティック・ロックの実行方法を示しています。例内の数字は、上記の各手順に対応しています。

```
com.ibm.db2.jcc.DB2Statement s1 =
    (com.ibm.db2.jcc.DB2Statement)conn.createStatement();
ResultSet rs =
    ((com.ibm.db2.jcc.DB2Statement)s1).executeDB2OptimisticLockingQuery
    ("SELECT EMPNO, SALARY, FROM EMP WHERE EMP.LASTNAME = 'HAAS'",
    com.ibm.db2.jcc.DB2Statement.RETURN_OPTLOCK_COLUMN_NO_FALSE_NEGATIVES); 1
    // Indicate that you plan to do
    // optimistic locking, and that you
    // want optimistic locking information
    // that does not generate
    // false negatives
ResultSetMetaData rsmd = rs.getMetaData();
int optColumns = 2
```

```

        ((com.ibm.db2.jcc.DB2ResultSetMetaData)rsmd).getDB2OptimisticLockingColumns();
        // Retrieve the optimistic locking
        // information.
boolean optColumnsReturned = false;

if (optColumns == 0);           // If optimistic locking information is not
                                // returned, do not attempt to do
                                // optimistic locking.
else if (optColumns == 1);     // A value of 1 is never returned if
                                // RETURN_OPTLOCK_COLUMN_NO_FALSE_NEGATIVES
                                // is specified, because 1 indicates
                                // that there could be false negatives.
else if (optColumns == 2)     // If optimistic locking information is
    optColumnsReturned = true; // returned, and false negatives will not
                                // occur, try optimistic locking.

rs.next();                     // Retrieve the contents of the ResultSet
int emp_id = rs.getInt(1);
double salary = rs.getDouble(2);

long rowChangeToken = 0;
Object rid = null;
int type = -1;

if (optColumnsReturned) {
    rowChangeToken =           // Get the row change token.
        ((com.ibm.db2.jcc.DB2ResultSet)rs).getDB2RowChangeToken();
    rid = ((com.ibm.db2.jcc.DB2ResultSet)rs).getDB2RID();
        // Get the RID, which uniquely identifies
        // the row.
    int type = ((com.ibm.db2.jcc.DB2ResultSet)rs).getDB2RIDType ();
        // Get the data type of the RID.
}
// *****
// Release the locks or disconnect from the database.
// Perform some work on the retrieved data.
// Reconnect to the data source.
// *****
...
PreparedStatement s2 =
    conn.prepareStatement ("UPDATE EMP SET SALARY = ? " +
        "WHERE EMPNO = ? AND ROW CHANGE TOKEN FOR EMP = ? and " +
        "RID_BIT(EMP) = ?");
                                // Statement for updating the
                                // previously selected rows that
                                // have not changed.

s2.setDouble(1, salary+10000);
s2.setInt(2, emp_id);
                                // Set the new row values.

s2.setLong(3, rowChangeToken);
                                // Set the row change token of the
                                // previously retrieved row.

if (type == java.sql.Types.BIGINT)
    s2.setLong (4, ((Long)rid).longValue());
else if (type == java.sql.Types.VARBINARY)
    s2.setBytes (4, (byte[])rid);
                                // Set the RID of the previously
                                // retrieved row.
                                // Use the correct setXXX method
                                // for the data type of the RID.

int updateCount = s2.executeUpdate();
                                // Perform the update.
if (updateCount == 1);         // Update is successful.
else                           // Update failed.
...

```

3

## JDBC アプリケーションでの XML データ

JDBC アプリケーションでは、XML 列へのデータの保管、および XML 列からのデータを取り出しが可能です。

データベース表では、XML 組み込みデータ・タイプを使用して XML データをノードの構造化セットとしてツリー形式で列に保管します。

アプリケーションでは、XML データは直列化されたストリング形式になります。

JDBC アプリケーションでは以下を行うことができます。

- `setXXX` メソッドを使用して XML 文書全体を XML 列に保管します。
- `getXXX` メソッドを使用して XML 文書全体を XML 列から取り出します。
- `SQL XMLQUERY` 関数を使用してシーケンスをデータベース内のシリアルライズされたシーケンスに取り出してから、`getXXX` メソッドを使用してデータをアプリケーション変数に取り出すことによって、XML 列の文書からシーケンスを取り出します。
- ストリング '`XQUERY`' が前に付加されている XQuery 式を使用して、シーケンスの要素をデータベース内の結果表に取り出すことによって、XML 列の文書からシーケンスを取り出します。結果表の各行はシーケンス内のアイテムを表します。続いて、`getXXX` メソッドを使用してデータをアプリケーション変数に取り出します。
- `SQL XMLTABLE` 関数を使用して結果表を定義し、その表を取り出すことによって、XML 列の文書からユーザー定義表としてシーケンスを取り出します。続いて、`getXXX` メソッドを使用して、結果表からのデータをアプリケーション変数に取り出します。

JDBC 4.0 の `java.sql.SQLXML` オブジェクトを使用して、XML 列内のデータの取り出しおよび更新を行うことができます。 `ResultSetMetaData.getColumnTypeName` などのメタデータ・メソッドを呼び出すと、XML 列タイプの整数値 `java.sql.Types.SQLXML` が戻されます。

## JDBC アプリケーションでの XML 列の更新

データベース表の XML 列についてデータを更新または挿入する場合は、JDBC アプリケーション内の入力データは直列化されたストリング形式である必要があります。

次の表には、データの XML 列への書き込みに使用できるメソッドと対応する入力データ・タイプがリストされています。

表 14. XML 列の更新用のメソッドとデータ・タイプ

メソッド	入力データ・タイプ
<code>PreparedStatement.setAsciiStream</code>	<code>InputStream</code>
<code>PreparedStatement.setBinaryStream</code>	<code>InputStream</code>
<code>PreparedStatement.setBlob</code>	<code>Blob</code>
<code>PreparedStatement.setBytes</code>	<code>byte[]</code>
<code>PreparedStatement.setCharacterStream</code>	<code>Reader</code>
<code>PreparedStatement.setClob</code>	<code>Clob</code>
<code>PreparedStatement.setObject</code>	<code>byte[]</code> 、 <code>Blob</code> 、 <code>Clob</code> 、 <code>SQLXML</code> 、 <code>DB2Xml</code> (非推奨)、 <code>InputStream</code> 、 <code>Reader</code> 、 <code>String</code>
<code>PreparedStatement.setString</code>	<code>String</code>

XML データのエンコード方式は、データ自体から導出する (内部的にエンコードされた データという) か、または外部ソースから導出する (外部的にエンコードされ

た データという) ことができます。データベース・サーバーにバイナリー・データとして送信される XML データは、内部的にエンコードされたデータとして処理されます。データ・ソースに文字データとして送信される XML データは、外部的にエンコードされたデータとして処理されます。

Java アプリケーションの外部エンコード方式は、常に Unicode エンコード方式です。

外部的にエンコードされたデータに、内部エンコード方式を含めることができます。つまり、このデータはデータ・ソースに文字データとして送信されますが、このデータにエンコード方式の情報を含めることができます。データ・ソースにより、以下のように内部エンコード方式と外部エンコード方式の間の非互換性が処理されます。

- データ・ソースが DB2 Database for Linux, UNIX, and Windows の場合、外部エンコード方式と内部エンコード方式の間に互換性がなく、外部エンコード方式と内部エンコード方式が Unicode でない場合は、データベース・ソースによりエラーが生成されます。外部エンコード方式と内部エンコード方式が Unicode の場合は、データベース・ソースでは内部エンコード方式が無視されます。
- データベース・ソースが DB2 for z/OS の場合、データベース・ソースでは内部エンコード方式が無視されます。

XML 列のデータは UTF-8 エンコード方式で保管されます。データベース・ソースにより、内部エンコード方式または外部エンコード方式から UTF-8 へのデータの変換が処理されます。

**例:** 以下の例は、データを SQLXML オブジェクトから XML 列へ挿入する方法を示しています。データがストリング・データであるため、データベース・ソースにより外部的にエンコードされるデータとして処理されます。

```
public void insertSQLXML()
{
    Connection con = DriverManager.getConnection(url);
    SQLXML info = con.createSQLXML();
        // Create an SQLXML object
    PreparedStatement insertStmt = null;
    String infoData =
        "<customerinfo xmlns=\"http://posample.org\" \" +
        \"Cid=\"1000\" xmlns=\"http://posample.org\">...</customerinfo>";
    cid.setString(cidData);
        // Populate the SQLXML object
    int cid = 1000;
    try {
        sqls = "INSERT INTO CUSTOMER (CID, INFO) VALUES (?, ?)";
        insertStmt = con.prepareStatement(sqls);
        insertStmt.setInt(1, cid);
        insertStmt.setSQLXML(2, info);
            // Assign the SQLXML object value
            // to an input parameter
        if (insertStmt.executeUpdate() != 1) {
            System.out.println("insertSQLXML: No record inserted.");
        }
    }
    catch (IOException ioe) {
        ioe.printStackTrace();
    }
    catch (SQLException sqle) {
        System.out.println("insertSQLXML: SQL Exception: " +
            sqle.getMessage());
    }
}
```



```

        System.out.println("insertSQLXML: SQL State: " +
            sql.getSQLState());
        System.out.println("insertSQLXML: SQL Error Code: " +
            sql.getErrorCode());
    }
}

```

**例:** 以下の例は、データをファイルから XML 列へ挿入する方法を示しています。データはバイナリー・データとして挿入されるため、データベース・サーバーにより内部エンコード方式が使用されます。

```

public void insertBinStream()
{
    PreparedStatement insertStmt = null;
    String sqls = null;
    int cid = 0;
    ResultSet rs=null;
    Statement stmt=null;
    try {
        sqls = "INSERT INTO CUSTOMER (CID, INFO) VALUES (?, ?)";
        insertStmt = conn.prepareStatement(sqls);
        insertStmt.setInt(1, cid);
        File file = new File(fn);
        insertStmt.setBinaryStream(2,
            new FileInputStream(file), (int)file.length());
        if (insertStmt.executeUpdate() != 1) {
            System.out.println("insertBinStream: No record inserted.");
        }
    }
    catch (IOException ioe) {
        ioe.printStackTrace();
    }
    catch (SQLException sqle) {
        System.out.println("insertBinStream: SQL Exception: " +
            sqle.getMessage());
        System.out.println("insertBinStream: SQL State: " +
            sqle.getSQLState());
        System.out.println("insertBinStream: SQL Error Code: " +
            sqle.getErrorCode());
    }
}
}

```

## JDBC アプリケーションにおける XML データの取り出し

JDBC アプリケーションでは、`ResultSet.getXXX` または `ResultSet.getObject` メソッドを使用して XML 列からデータを取り出すことができます。

DB2 表の XML 列からデータを取り出す場合、出力データは直列化されたストリング・フォーマットになります。これは、XML 列の内容全体を取り出す場合、または列からシーケンスを取り出す場合に当てはまります。

XML データを取り出すには、以下の手法のいずれかを使用することができます。

- `ResultSet.getSQLXML` メソッドを使用してデータを取り出します。次に、`SQLXML.getXXX` メソッドを使用して、データを互換性のある出力データ・タイプに取り出します。

`SQLXML.getXXX` メソッドは、XML 宣言をエンコード仕様と共に出力データに追加します。

- `ResultSet.getObject` 以外の `ResultSet.getXXX` メソッドを使用して、互換性のあるデータ・タイプにデータを取り出します。
- `ResultSet.getObject` メソッドを使用してデータを取り出してから、それを `DB2Xml` タイプにキャストし、`DB2Xml` オブジェクトに割り当てます。次に、`DB2Xml.getDB2XXX` または `DB2Xml.getDB2XmlXXX` メソッドを使用して、互換性のある出力データ・タイプにデータを取り出します。

`DB2Xml.getDB2XmlXXX` メソッドは、XML 宣言をエンコード仕様とともに出力データに追加します。`DB2Xml.getDB2XXX` メソッドは、XML 宣言をエンコード仕様とともに出力データに追加しません。

この手法では、推奨されない `DB2Xml` オブジェクトが使用されます。前で説明した手法の使用をお勧めします。

次の表には、XML データを取り出すための `ResultSet` メソッドおよび対応する出力データ・タイプがリストされています。

表 15. XML データを取り出すための `ResultSet` メソッドおよびデータ・タイプ

メソッド	出力データ・タイプ
<code>ResultSet.getAsciiStream</code>	<code>InputStream</code>
<code>ResultSet.getBinaryStream</code>	<code>InputStream</code>
<code>ResultSet.getBytes</code>	<code>byte[]</code>
<code>ResultSet.getCharacterStream</code>	<code>Reader</code>
<code>ResultSet.getObject</code>	<code>DB2Xml</code>
<code>ResultSet.getSQLXML</code>	<code>SQLXML</code>
<code>ResultSet.getString</code>	<code>String</code>

次の表には、`java.sql.SQLXML` または `com.ibm.db2.jcc.DB2Xml` オブジェクトからのデータの取り出しに呼び出すことができるメソッド、および対応する出力データ・タイプと XML 宣言でのエンコード方式のタイプがリストされています。

表 16. `SQLXML` および `DB2Xml` メソッド、データ・タイプ、および追加されるエンコード仕様

メソッド	出力データ・タイプ	追加される XML 内部エンコード宣言のタイプ
<code>SQLXML.getBinaryStream</code>	<code>InputStream</code>	なし
<code>SQLXML.getCharacterStream</code>	<code>Reader</code>	なし
<code>SQLXML.getSource</code>	<code>Source</code>	なし
<code>SQLXML.getString</code>	<code>String</code>	なし
<code>DB2Xml.getDB2AsciiStream</code>	<code>InputStream</code>	なし
<code>DB2Xml.getDB2BinaryStream</code>	<code>InputStream</code>	なし
<code>DB2Xml.getDB2Bytes</code>	<code>byte[]</code>	なし
<code>DB2Xml.getDB2CharacterStream</code>	<code>Reader</code>	なし
<code>DB2Xml.getDB2String</code>	<code>String</code>	なし
<code>DB2Xml.getDB2XmlAsciiStream</code>	<code>InputStream</code>	US-ASCII
<code>DB2Xml.getDB2XmlBinaryStream</code>	<code>InputStream</code>	<code>getDB2XmlBinaryStream targetEncoding</code> パラメーターで指定

表 16. SQLXML および DB2Xml メソッド、データ・タイプ、および追加されるエンコード仕様 (続き)

メソッド	出力データ・タイプ	追加される XML 内部エンコード宣言のタイプ
DB2Xml.getDB2XmlBytes	byte[]	DB2Xml.getDB2XmlBytes <i>targetEncoding</i> パラメータで指定
DB2Xml.getDB2XmlCharacterStream	Reader	ISO-10646-UCS-2
DB2Xml.getDB2XmlString	String	ISO-10646-UCS-2

戻されるデータに対してアプリケーションが XMLSERIALIZE 関数を実行すると、関数の実行後に、そのデータには XML データ・タイプではなく XMLSERIALIZE 関数で指定されたデータ・タイプが含まれます。そのため、ドライバーは指定されたタイプとしてそのデータを処理し、内部エンコード宣言をすべて無視します。

**例:** 以下の例は、データを XML 列から SQLXML オブジェクトに取り出してから、SQLXML.getString メソッドを使用して、ストリングにデータを取り出す方法を示しています。

```
public void fetchToSQLXML()
{
    System.out.println(">> fetchToSQLXML: Get XML data as an SQLXML object " +
        "using getSQLXML");
    PreparedStatement selectStmt = null;
    String sqls = null, stringDoc = null;
    ResultSet rs = null;

    try{
        sqls = "SELECT info FROM customer WHERE cid = " + cid;
        selectStmt = conn.prepareStatement(sqls);
        rs = selectStmt.executeQuery();

        // Get metadata
        // Column type for XML column is the integer java.sql.Types.OTHER
        ResultSetMetaData meta = rs.getMetaData();
        String colType = meta.getColumnType(1);
        System.out.println("fetchToSQLXML: Column type = " + colType);
        while (rs.next()) {
            // Retrieve the XML data with getSQLXML.
            // Then write it to a string with
            // explicit internal ISO-10646-UCS-2 encoding.
            java.sql.SQLXML xml = rs.getSQLXML(1);
            System.out.println (xml.getString());
        }
        rs.close();
    }
    catch (SQLException sqle) {
        System.out.println("fetchToSQLXML: SQL Exception: " +
            sqle.getMessage());
        System.out.println("fetchToSQLXML: SQL State: " +
            sqle.getSQLState());
        System.out.println("fetchToSQLXML: SQL Error Code: " +
            sqle.getErrorCode());
    }
}
```

**例:** 以下の例は、データを XML 列からストリング変数に取り出す方法を示しています。

```
public void fetchToString()
{
    System.out.println(">> fetchToString: Get XML data " +
        "using getString");
}
```

```

PreparedStatement selectStmt = null;
String sqls = null, stringDoc = null;
ResultSet rs = null;

try{
    sqls = "SELECT info FROM customer WHERE cid = " + cid;
    selectStmt = conn.prepareStatement(sqls);
    rs = selectStmt.executeQuery();

    // Get metadata
    // Column type for XML column is the integer java.sql.Types.OTHER
    ResultSetMetaData meta = rs.getMetaData();
    String colType = meta.getColumnType(1);
    System.out.println("fetchToString: Column type = " + colType);

    while (rs.next()) {
        stringDoc = rs.getString(1);
        System.out.println("Document contents:");
        System.out.println(stringDoc);
    }
} catch (SQLException sqle) {
    System.out.println("fetchToString: SQL Exception: " +
        sqle.getMessage());
    System.out.println("fetchToString: SQL State: " +
        sqle.getSQLState());
    System.out.println("fetchToString: SQL Error Code: " +
        sqle.getErrorCode());
}
}

```

**例:** 以下の例は、データを XML 列から DB2Xml オブジェクトに取り出してから、DB2Xml.getDB2XmlString メソッドを使用して、ISO-10646-UCS-2 エンコード仕様で追加された XML 宣言を含むストリングにデータを取り出す方法を示しています。

```

public void fetchToDB2Xml()
{
    System.out.println(">> fetchToDB2Xml: Get XML data as a DB2XML object " +
        "using getObject");
    PreparedStatement selectStmt = null;
    String sqls = null, stringDoc = null;
    ResultSet rs = null;

    try{
        sqls = "SELECT info FROM customer WHERE cid = " + cid;
        selectStmt = conn.prepareStatement(sqls);
        rs = selectStmt.executeQuery();

        // Get metadata
        // Column type for XML column is the integer java.sql.Types.OTHER
        ResultSetMetaData meta = rs.getMetaData();
        String colType = meta.getColumnType(1);
        System.out.println("fetchToDB2Xml: Column type = " + colType);
        while (rs.next()) {
            // Retrieve the XML data with getObject, and cast the object
            // as a DB2Xml object. Then write it to a string with
            // explicit internal ISO-10646-UCS-2 encoding.
            com.ibm.db2.jcc.DB2Xml xml =
                (com.ibm.db2.jcc.DB2Xml) rs.getObject(1);
            System.out.println (xml.getDB2XmlString());
        }
        rs.close();
    }
    catch (SQLException sqle) {
        System.out.println("fetchToDB2Xml: SQL Exception: " +
            sqle.getMessage());
    }
}

```

```

        System.out.println("fetchToDB2Xml: SQL State: " +
            sqlc.getSQLState());
        System.out.println("fetchToDB2Xml: SQL Error Code: " +
            sqlc.getErrorCode());
    }
}

```

## Java アプリケーションでの XML パラメーターを使用したルーチンの呼び出し

SQL または外部ストアド・プロシージャ、および外部ユーザー定義関数には、XML パラメーターを組み込むことができます。

SQL プロシージャの場合、ストアド・プロシージャ定義内のそれらのパラメーターは XML タイプです。外部ストアド・プロシージャおよびユーザー定義関数の場合は、ルーチン定義内の XML パラメーターは XML AS CLOB タイプです。XML パラメーターのあるストアド・プロシージャまたはユーザー定義関数を呼び出す場合は、呼び出しステートメントで互換データ・タイプを使用する必要があります。

JDBC プログラムから XML 入力パラメーターのあるルーチンを呼び出すには、`java.sql.SQLXML` または `com.ibm.db2.jcc.DB2Xml` タイプのパラメーターを使用します。XML 出力パラメーターを登録するには、パラメーターを `java.sql.Types.SQLXML` または `com.ibm.db2.jcc.DB2Types.XML` タイプで登録します。( `com.ibm.db2.jcc.DB2Xml` and `com.ibm.db2.jcc.DB2Types.XML` タイプは推奨されません。)

**例:** 3 つの XML パラメーター (IN パラメーター、OUT パラメーター、および INOUT パラメーター) を使用するストアド・プロシージャを呼び出す JDBC プログラム。この例では JDBC 4.0 が必要です。

```

java.sql.SQLXML in_xml = xmlvar;
java.sql.SQLXML out_xml = null;
java.sql.SQLXML inout_xml = xmlvar;
// Declare an input, output, and
// input/output XML parameter

Connection con;
CallableStatement cstmt;
ResultSet rs;
...
stmt = con.prepareCall("CALL SP_xml(?,?,?)");
// Create a CallableStatement object
cstmt.setObject (1, in_xml); // Set input parameter
cstmt.registerOutParameter (2, java.sql.Types.SQLXML);
// Register out and input parameters
cstmt.registerOutParameter (3, java.sql.Types.SQLXML);
cstmt.executeUpdate(); // Call the stored procedure
System.out.println("Parameter values from SP_xml call: ");
System.out.println("Output parameter value ");
printString(out_xml.getString());
// Use the SQLXML.getString
// method getBytes to convert the
// value to a string for printing
System.out.println("Input/output parameter value ");
printString(inout_xml.getString());

```

SQLJ プログラムから XML パラメーターのあるルーチンを呼び出すには、`com.ibm.db2.jcc.DB2Xml` タイプのパラメーターを使用します。

例: 3 つの XML パラメーター (IN パラメーター、OUT パラメーター、および INOUT パラメーター) を使用するストアード・プロシージャを呼び出す SQLJ プログラム。この例では JDBC 4.0 が必要です。

```
java.sql.SQLXML in_xml = xmlvar;
java.sql.SQLXML out_xml = null;
java.sql.SQLXML inout_xml = xmlvar;
                                // Declare an input, output, and
                                // input/output XML parameter
...
#sql [myConnCtx] {CALL SP_xml(:IN in_xml,
                                :OUT out_xml,
                                :INOUT inout_xml)};
                                // Call the stored procedure
System.out.println("Parameter values from SP_xml call: ");
System.out.println("Output parameter value ");
printString(out_xml.getString());
                                // Use the SQLXML.getString method to
                                // convert the value to a string for printing
System.out.println("Input/output parameter value ");
printString(inout_xml.getString());
```

## XML スキーマの登録および除去の Java サポート

IBM Data Server Driver for JDBC and SQLJ により、XML スキーマとそのコンポーネントを登録および除去するための Java アプリケーション・プログラムを作成できるメソッドが提供されます。

これらのメソッドは、DB2 で提供される

SYSPROC.XSR\_REGISTER、SYSPROC.XSR\_ADDSCHEMADOC、SYSPROC.XSR\_COMPLETE、SYSPROC.XSR\_REMOVE、および SYSPROC.XSR\_UPDATE の各ストアード・プロシージャと同等です。JDBC メソッドは以下のとおりです。

### **DB2Connection.registerDB2XMLSchema**

XML スキーマを 1 つ以上の XML スキーマ文書を使用して DB2 に登録します。このメソッドには 2 つの形式があります。1 つは `InputStream` オブジェクトから入力される XML スキーマ文書用の形式で、もう 1 つは `String` 内の XML スキーマ文書用の形式です。

### **DB2Connection.deregisterDB2XMLObject**

XML スキーマ定義を DB2 から除去します。

### **DB2Connection.updateDB2XmlSchema**

登録済みの XML スキーマ内の XML スキーマ文書を登録済みの別の XML スキーマからの XML スキーマ文書で置き換えます。オプションで、内容がコピーされた XML スキーマをドロップします。

これらのメソッドを呼び出す前に、ベースとなるストアード・プロシージャを DB2 データベース・サーバーにインストールしておく必要があります。

例: XML スキーマの登録: 以下の例では、入力ストリームから読み取られた単一の XML スキーマ文書 (`customer.xsd`) を使用して、XML スキーマを DB2 に登録するために `registerDB2XmlSchema` を使用方法が示されています。登録済みのスキーマの SQL スキーマ名は `SYSXSR` です。 `xmlSchemaLocations` 値が `NULL` のため、DB2 では、この XML スキーマは `NULL` 以外の XML スキーマ・ロケーション値を提供する `DSN_XMLVALIDATE` の呼び出し時には検出されません。追加のプ

ロパティは登録されません。

図 23. 入力ストリームからの XML 文書を使用した XML スキーマの DB2 への登録の例

```
public static void registerSchema(
    Connection con,
    String schemaName)
    throws SQLException {
    // Define the registerDB2XmlSchema parameters
    String[] xmlSchemaNameQualifiers = new String[1];
    String[] xmlSchemaNames = new String[1];
    String[] xmlSchemaLocations = new String[1];
    InputStream[] xmlSchemaDocuments = new InputStream[1];
    int[] xmlSchemaDocumentsLengths = new int[1];
    java.io.InputStream[] xmlSchemaDocumentsProperties = new InputStream[1];
    int[] xmlSchemaDocumentsPropertiesLengths = new int[1];
    InputStream xmlSchemaProperties;
    int xmlSchemaPropertiesLength;
    //Set the parameter values
    xmlSchemaLocations[0] = "";
    FileInputStream fi = null;
    xmlSchemaNameQualifiers[0] = "SYSXSR";
    xmlSchemaNames[0] = schemaName;
    try {
        fi = new FileInputStream("customer.xsd");
        xmlSchemaDocuments[0] = new BufferedInputStream(fi);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
    try {
        xmlSchemaDocumentsLengths[0] = (int) fi.getChannel().size();
        System.out.println(xmlSchemaDocumentsLengths[0]);
    } catch (IOException e1) {
        e1.printStackTrace();
    }
    xmlSchemaDocumentsProperties[0] = null;
    xmlSchemaDocumentsPropertiesLengths[0] = 0;
    xmlSchemaProperties = null;
    xmlSchemaPropertiesLength = 0;
    DB2Connection ds = (DB2Connection) con;
    // Invoke registerDB2XmlSchema
    ds.registerDB2XmlSchema(
        xmlSchemaNameQualifiers,
        xmlSchemaNames,
        xmlSchemaLocations,
        xmlSchemaDocuments,
        xmlSchemaDocumentsLengths,
        xmlSchemaDocumentsProperties,
        xmlSchemaDocumentsPropertiesLengths,
        xmlSchemaProperties,
        xmlSchemaPropertiesLength,
        false);
}
```

例: XML スキーマの除去: 以下の例では、XML スキーマを DB2 から除去するために `deregisterDB2XmlObject` を使用する方法が示されています。登録済みのスキーマの SQL スキーマ名は `SYSXSR` です。

図 24. XML スキーマの DB2 からの除去の例

```
public static void deregisterSchema(
    Connection con,
    String schemaName)
```



```

throws SQLException {
// Define and assign values to the deregisterDB2XmlObject parameters
String xmlSchemaNameQualifier = "SYSXSR";
String xmlSchemaName = schemaName;
DB2Connection ds = (DB2Connection) con;
// Invoke deregisterDB2XmlObject
ds.deregisterDB2XmlObject(
    xmlSchemaNameQualifier,
    xmlSchemaName);
}

```

例: XML スキーマの更新: 以下の例では、XML スキーマの内容を別の XML スキーマの内容で更新するために `updateDB2XmlSchema` を使用する方法が示されています。コピーされたスキーマは、リポジトリ内に維持されます。登録済みの両方のスキーマの SQL スキーマ名は `SYSXSR` です。

図 25. XML スキーマの更新の例

```

public static void updateSchema(
    Connection con,
    String schemaNameTarget,
    String schemaNameSource)
throws SQLException {
// Define and assign values to the updateDB2XmlSchema parameters
String xmlSchemaNameQualifierTarget = "SYSXSR";
String xmlSchemaNameQualifierSource = "SYSXSR";
String xmlSchemaNameTarget = schemaNameTarget;
String xmlSchemaNameSource = schemaNameSource;
boolean dropSourceSchema = false;
DB2Connection ds = (DB2Connection) con;
// Invoke updateDB2XmlSchema
ds.updateDB2XmlSchema(
    xmlSchemaNameQualifierTarget,
    xmlSchemaNameTarget,
    xmlSchemaNameQualifierSource,
    xmlSchemaNameSource,
    dropSourceSchema);
}

```

---

## JDBC アプリケーションにおけるトランザクション制御

JDBC アプリケーションでは、他のタイプの SQL アプリケーションと同様に、トランザクション制御にトランザクションの明示的または暗黙的なコミットおよびロールバック、およびトランザクションの分離レベルの設定が関係します。

### IBM Data Server Driver for JDBC and SQLJ の分離レベル

IBM Data Server Driver for JDBC and SQLJ は多数の分離レベルをサポートしています。これらは、データベース・サーバーの分離レベルに対応しています。

JDBC 分離レベルは、`Connection.setTransactionIsolation` メソッドを使用して、JDBC プログラム内の作業単位に対して設定できます。デフォルトの分離レベルは `defaultIsolationLevel` プロパティで設定できます。

次の表は、`Connection.setTransactionIsolation` メソッドで指定できる *level* の値と、データベース・サーバーでそれに相当するレベルを示しています。

表 17. 同等の JDBC および DB2 分離レベル

JDBC 値	DB2 独立性レベル
TRANSACTION_SERIALIZABLE	反復可能読み取り
TRANSACTION_REPEATABLE_READ	読み取り固定
TRANSACTION_READ_COMMITTED	カーソル固定
TRANSACTION_READ_UNCOMMITTED	非コミット読み取り

## JDBC トランザクションのコミットまたはロールバック

JDBC では、トランザクションの明示的なコミットまたはロールバックのために、`commit` メソッドまたは `rollback` メソッドを使用します。

以下に例を示します。

```
Connection con;
...
con.commit();
```

自動コミット・モードがオンになっていると、データベース・マネージャーは、各 SQL ステートメントの完了後にコミット操作を実行します。自動コミット・モードをオンに設定するには、`Connection.setAutoCommit(true)` メソッドを呼び出します。自動コミット・モードをオフに設定するには、`Connection.setAutoCommit(false)` メソッドを呼び出します。自動コミット・モードがオンになっているかどうかを確認するには、`Connection.getAutoCommit` メソッドを呼び出します。

分散トランザクションに加わっている接続では、`setAutoCommit(true)` メソッドを呼び出せません。

自動コミットの状態を変更すると、データベース・マネージャーは、アプリケーションがトランザクションの境界に達していなくてもコミット操作を実行します。

接続が分散トランザクションに加わっていると、関連するアプリケーションは、`commit` メソッドまたは `rollback` メソッドを呼び出せません。

## デフォルトの JDBC 自動コミット・モード

デフォルトの自動コミット・モードは、JDBC アプリケーションが接続するデータ・ソースによって決まります。

### DB2 データ・ソースの自動コミットのデフォルト

DB2 データ・ソースへの接続の場合、デフォルトの自動コミット・モードは `true` です。

### IDS データ・ソースの自動コミットのデフォルト

IDS データ・ソースに対する接続の場合、デフォルトの自動コミット・モードはデータ・ソースのタイプによって決まります。次の表はデフォルトを示しています。

表 18. IDS データ・ソースのデフォルトの自動コミット・モード

データ・ソースのタイプ	デフォルトの自動コミット・モード (ローカル・トランザクション)	デフォルトの自動コミット・モード (グローバル・トランザクション)
ANSI 準拠データベース	true	false
ロギングを行わない非 ANSI 準拠データベース	false	適用外
ロギングを行う非 ANSI 準拠データベース	true	false

## IBM Data Server Driver for JDBC and SQLJ での例外と警告

JDBC アプリケーションでは、SQL エラーによって例外がスローされ、ユーザーは try/catch ブロックを使用してこの例外を処理します。SQL 警告では例外はスローされないため、SQL ステートメントの実行後に警告が発生したかどうかを確認するためのメソッドを呼び出す必要があります。

IBM Data Server Driver for JDBC and SQLJ では次のようなクラスおよびインターフェースが提供され、これによってエラーや警告の情報が得られます。

### SQLException

エラー処理のための SQLException クラス。すべての JDBC メソッドは、実行中にエラーが発生すると SQLException のインスタンスをスローします。JDBC 仕様に従って、SQLException オブジェクトには以下の情報が含まれます。

- エラー・コードを含む int 値。SQLException.getErrorCode はこの値を取得します。
- SQLSTATE を含む String オブジェクト、または NULL。SQLException.getSQLState はこの値を取得します。
- エラーについての記述を含む String オブジェクト、または NULL。SQLException.getMessage はこの値を取得します。
- 次の SQLException へのポインター、または NULL。SQLException.getNextException はこの値を取得します。

JDBC メソッドがスローする SQLException が 1 つだけの場合、その SQLException は、IBM Data Server Driver for JDBC and SQLJ がメソッドを処理したときに発生した基本の Java 例外によって引き起こされた可能性があります。この場合は SQLException がこの基礎となる例外をラップするので、SQLException.getCause メソッドによってエラーに関する情報を取り出すことができます。

### DB2Diagnosable

IBM Data Server Driver for JDBC and SQLJ 専用のインターフェース com.ibm.db2.jcc.DB2Diagnosable は、SQLException クラスを拡張します。DB2Diagnosable インターフェースにより、データ・ソースへのアクセス時に発生するエラーについてより多くの情報が得られます。JDBC ドライバーがエラーを検出した場合は、DB2Diagnosable によって得られる情報は、標準の SQLException クラ

スによる情報と同じです。しかしデータベース・サーバーがエラーを検出したときには、DB2Diagnosable は以下のメソッドを追加して、エラーに関する追加情報を提供します。

#### **getSqlca**

以下の情報を含む DB2Sqlca オブジェクトを返します。

- SQL エラー・コード
- SQLERRMC 値
- SQLERRP 値
- SQLERRD 値
- SQLWARN 値
- SQLSTATE

#### **getThrowable**

SQLException の原因となった java.lang.Throwable オブジェクトを返します。そういうオブジェクトが存在しない場合は NULL を返します。

#### **printTrace**

診断情報を印刷します。

## **SQLException サブクラス**

JDBC 4.0 以降では、以下の例外クラスをキャッチすることで、SQLException よりさらに詳しい情報を取得できます。

#### • SQLNonTransientException

SQLNonTransientException は、以前に失敗した SQL 操作を再試行しても正常に実行できない場合に、何らかの修正処置をとらない限りスローされます。

SQLNonTransientException クラスには以下のサブクラスがあります。

- SQLFeatureNotSupportedException
- SQLNonTransientConnectionException
- SQLDataException
- SQLIntegrityConstraintViolationException
- SQLInvalidAuthorizationSpecException
- SQLSyntaxException

#### • SQLTransientException

SQLTransientException は、以前に失敗した SQL 操作を再試行すれば成功する可能性がある場合に、アプリケーションからの介入を受けることなくスローされます。SQLTransientException がスローされたあとでも接続は引き続き有効です。

SQLTransientException クラスには以下のサブクラスがあります。

- SQLTransientConnectionException
- SQLTransientRollbackException
- SQLTimeoutException

#### • SQLRecoverableException

SQLRecoverableException がスローされるのは、以前に失敗した操作が、アプリケーションがリカバリー・ステップを実行してトランザクションを再試行すれば成功する可能性がある場合です。SQLRecoverableException がスローされると、接続は無効になります。

- `SQLClientInfoException`

`SQLClientInfoException` は、設定できないクライアント・プロパティが 1 つ以上ある場合に `Connection.setClientInfo` メソッドによってスローされます。

`SQLClientInfoException` は、設定できないプロパティを示します。

## BatchUpdateException

`BatchUpdateException` オブジェクトには、SQL ステートメントのバッチの実行中に発生するエラーに関する以下の項目が含まれています。

- エラーについての記述を含む `String` オブジェクト、または `NULL`
- 失敗した SQL ステートメントの `SQLSTATE` を含む `String` オブジェクト、または `NULL`
- エラー・コードを含む整数値、またはゼロ
- バッチ内の SQL ステートメントの更新カウンターの整数配列、または `null`
- `SQLException` オブジェクトへのポインター、または `NULL`

バッチ全体に対して、1 つの `BatchUpdateException` がスローされます。少なくとも 1 つの `SQLException` オブジェクトが、`BatchUpdateException` オブジェクトにチェーニングされます。`SQLException` オブジェクトは、それに対応するステートメントがバッチに追加される順序と同じ順序でチェーニングされます。`SQLException` オブジェクトをバッチ内のステートメントにマッチングできるように、個々の `SQLException` オブジェクトのエラー記述フィールドは次のストリングで始まっています。

Error for batch element #*n*:

*n* はバッチ内のステートメントの番号です。

バッチ実行中の SQL 警告は `BatchUpdateException` をスローしません。警告に関する情報を取得するには、`executeBatch` メソッドを実行したオブジェクトに対して `Statement.getWarnings` メソッドを使用します。これで、各 `SQLWarning` オブジェクトのエラーに関する記述、`SQLSTATE`、およびエラー・コードを取得できます。

## SQLWarning

SQL ステートメントが正の数の `SQLCODE` を戻す場合、および SQL ステートメントがゼロ以外の `SQLSTATE` を持つ 0 の `SQLCODE` を戻す場合、IBM Data Server Driver for JDBC and SQLJ は警告を累積します。

`getWarnings` を呼び出すことによって、`SQLWarning` オブジェクトを取得できます。

**重要:** `Statement.executeUpdate` または `PreparedStatement.executeUpdate` への呼び出しがどの行にも影響を与えない場合、IBM Data Server Driver for JDBC and SQLJ はエラー・コード +100 の `SQLWarning` を生成します。

`ResultSet.next` への呼び出しで行が戻されない場合、IBM Data Server Driver for JDBC and SQLJ は `SQLWarning` を生成しません。

汎用の `SQLWarning` オブジェクトには、以下の情報が含まれています。

- 警告についての記述を含む `String` オブジェクト、または `NULL`
- `SQLSTATE` を含む `String` オブジェクト、または `NULL`

- エラー・コードを含む int 値
- 次の SQLWarning へのポインター、または NULL

IBM Data Server Driver for JDBC and SQLJ の下では、SQLException オブジェクトと同様に、SQLWarning オブジェクトにも DB2 固有の情報を含めることができます。SQLWarning オブジェクトの DB2 固有の情報は、SQLException オブジェクトの DB2 固有の情報と同じです。

## IBM Data Server Driver for JDBC and SQLJ 使用時の SQLException の処理

すべての Java プログラムに共通する点ですが、JDBC アプリケーションの場合も、エラー処理は、try/catch ブロックで実行します。エラーの発生時にメソッドが例外をスローすると、catch ブロックのコードがその例外を処理します。

IBM Data Server Driver for JDBC and SQLJ で実行する JDBC プログラムで SQLException を処理するための基本手順は、以下のとおりです。

1. com.ibm.db2.jcc.DB2Diagnosable インターフェースと com.ibm.db2.jcc.DB2Sqlca クラスに対するアクセス権をプログラムに与えます。そのインターフェースとクラスへのすべての参照を完全に修飾するか、そのインターフェースとクラスをインポートできます。

```
import com.ibm.db2.jcc.DB2Diagnosable;
import com.ibm.db2.jcc.DB2Sqlca;
```

2. オプション: SQLException.getMessage 呼び出しから完全なメッセージ・テキストを取得する場合は、DB2 for z/OS または IBM Informix Dynamic Server (IDS) のデータ・ソースに接続するときに、setRetrieveMessagesFromServerOnGetMessage プロパティを true に設定します。
3. SQLException を生成できるコードを try ブロックに配置します。
4. catch ブロックのループの中で以下の手順を実行します。
  - a. 最後の SQLException を取得したかどうかをテストします。そうでない場合は、次の手順に進みます。
  - b. オプション: IDS データ・ソースで実行する SQL ステートメントの場合は、com.ibm.db2.jcc.DB2Statement.getIDSSQLStatementOffset メソッドを実行して、構文エラーがある列を判別します。

DB2Statement.getIDSSQLStatementOffset は、SQL ステートメントに含まれている最初の構文エラーのオフセットを返します。

- c. SQLException が DB2Diagnosable のインスタンスであるかどうかをテストして、IBM Data Server Driver for JDBC and SQLJ のみの情報が存在するかどうかを確認します。存在する場合は、以下のようになります。
  - 1) そのオブジェクトを DB2Diagnosable オブジェクトにキャストします。
  - 2) 省略可能: DB2Diagnosable.printStackTrace メソッドを呼び出して、すべての SQLException 情報を java.io.PrintWriter オブジェクトに書き出します。
  - 3) DB2Diagnosable.getThrowable メソッドを呼び出して、基礎になっている java.lang.Throwable が SQLException の原因かどうかを確認します。



- 4) DB2Diagnosable.getSqlca メソッドを呼び出して、DB2Sqlca オブジェクトを取得します。
  - 5) DB2Sqlca.getSqlCode メソッドを呼び出して、SQL エラー・コード値を取得します。
  - 6) DB2Sqlca.getSqlErrmc メソッドを呼び出して、すべての SQLERRMC 値が含まれているストリングを取得するか、DB2Sqlca.getSqlErrmcTokens メソッドを呼び出して、SQLERRMC 値を配列として取得します。
  - 7) DB2Sqlca.getSqlErrp メソッドを呼び出して、SQLERRP 値を取得します。
  - 8) DB2Sqlca.getSqlErrd メソッドを呼び出して、SQLERRD 値を配列として取得します。
  - 9) DB2Sqlca.getSqlWarn メソッドを呼び出して、SQLWARN 値を配列として取得します。
  - 10) DB2Sqlca.getSqlState メソッドを呼び出して、SQLSTATE 値を取得します。
  - 11) DB2Sqlca.getMessage メソッドを呼び出して、データ・ソースからエラー・メッセージ・テキストを取得します。
- d. SQLException.getNextException メソッドを呼び出して、次の SQLException を取得します。

IBM Data Server Driver for JDBC and SQLJ に用意されている SQLException から IBM Data Server Driver for JDBC and SQLJ 固有の情報を取得するコードを以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

図 26. IBM Data Server Driver for JDBC and SQLJ 使用時の SQLException の処理

```

import java.sql.*;           // Import JDBC API package
import com.ibm.db2.jcc.DB2Diagnosable; // Import packages for DB2
import com.ibm.db2.jcc.DB2Sqlca;    // SQLException support
java.io.PrintWriter printWriter;    // For dumping all SQLException
                                   // information
String url = "jdbc:db2://myhost:9999/myDB:" +
    "retrieveMessagesFromServerOnGetMessage=true;";
                                   // Set properties to retrieve full message
                                   // text

String user = "db2adm";
String password = "db2adm";
java.sql.Connection con =
    java.sql.DriverManager.getConnection (url, user, password)
                                   // Connect to a DB2 for z/OS data source

...
try {
    // Code that could generate SQLExceptions
    ...
} catch(SQLException sqle) {
    while(sqle != null) {           // Check whether there are more
                                   // SQLExceptions to process
//=====> Optional IBM Data Server Driver for JDBC and SQLJ-only
// error processing
        if (sqle instanceof DB2Diagnosable) {
            // Check if IBM Data Server Driver for JDBC and SQLJ-only

```



```

// information exists
com.ibm.db2.jcc.DB2Diagnosable diagnosable =
    (com.ibm.db2.jcc.DB2Diagnosable)sqlc;
diagnosable.printStackTrace (printWriter, "");
java.lang.Throwable throwable =
    diagnosable.getThrowable();
if (throwable != null) {
    // Extract java.lang.Throwable information
    // such as message or stack trace.
    ...
}
DB2Sqlca sqlca = diagnosable.getSqlca();
if (sqlca != null) {
    int sqlCode = sqlca.getSqlCode();
    String sqlErrmc = sqlca.getSqlErrmc();
    String[] sqlErrmcTokens = sqlca.getSqlErrmcTokens();
    String sqlErrp = sqlca.getSqlErrp();
    int[] sqlErrd = sqlca.getSqlErrd();
    char[] sqlWarn = sqlca.getSqlWarn();
    String sqlState = sqlca.getSqlState();
    String errMsg = sqlca.getMessage();

    System.err.println ("Server error message: " + errMsg);

    System.err.println ("----- SQLCA -----");
    System.err.println ("Error code: " + sqlCode);
    System.err.println ("SQLERRMC: " + sqlErrmc);
    if (sqlErrmcTokens != null) {
        for (int i=0; i< sqlErrmcTokens.length; i++) {
            System.err.println (" token " + i + ": " + sqlErrmcTokens[i]);
        }
    }
    System.err.println ( "SQLERRP: " + sqlErrp );
    System.err.println (
        "SQLERRD(1): " + sqlErrd[0] + "\n" +
        "SQLERRD(2): " + sqlErrd[1] + "\n" +
        "SQLERRD(3): " + sqlErrd[2] + "\n" +
        "SQLERRD(4): " + sqlErrd[3] + "\n" +
        "SQLERRD(5): " + sqlErrd[4] + "\n" +
        "SQLERRD(6): " + sqlErrd[5] );
    System.err.println (
        "SQLWARN1: " + sqlWarn[0] + "\n" +
        "SQLWARN2: " + sqlWarn[1] + "\n" +
        "SQLWARN3: " + sqlWarn[2] + "\n" +
        "SQLWARN4: " + sqlWarn[3] + "\n" +
        "SQLWARN5: " + sqlWarn[4] + "\n" +
        "SQLWARN6: " + sqlWarn[5] + "\n" +
        "SQLWARN7: " + sqlWarn[6] + "\n" +
        "SQLWARN8: " + sqlWarn[7] + "\n" +
        "SQLWARN9: " + sqlWarn[8] + "\n" +
        "SQLWARNA: " + sqlWarn[9] );
    System.err.println ("SQLSTATE: " + sqlState);
}
sqlc=sqlc.getNextException();
}
}

```

4c1

4c2

4c3

4c4

4c5

4c6

4c7

4c8

4c9

4c10

4c11

4d

## IBM Data Server Driver for JDBC and SQLJ 使用時の SQLWarning の処理

SQL エラーとは異なり、SQL 警告の場合は、JDBC メソッドから例外がスローされません。その代わりに、

Connection、Statement、PreparedStatement、CallableStatement、ResultSet の各クラスには、getWarnings メソッドが含まれています。SQL ステートメントの実行後にそのメソッドを呼び出して、SQL 警告が生成されているかどうかを確認する必要があります。

SQL 警告の情報を取得するための基本手順は、以下のとおりです。

1. オプション: データベース・サーバーに接続するときに、SQLWarning オブジェクトに影響するプロパティを設定します。

SQLWarning.getMessage の呼び出しを実行するときに、DB2 for z/OS または IBM Informix Dynamic Server (IDS) のデータ・ソースから完全なメッセージ・テキストを取得する場合は、setRetrieveMessagesFromServerOnGetMessage プロパティを true に設定します。

2. データベース・サーバーに接続するか SQL ステートメントを実行するメソッドを呼び出した直後に、getWarnings メソッドを呼び出して、SQLWarning オブジェクトを取得します。
3. ループの中で以下の手順を実行します。
  - a. SQLWarning オブジェクトが NULL かどうかをテストします。そうでない場合は、次の手順に進みます。
  - b. SQLWarning.getMessage メソッドを呼び出して、警告に関する記述を取得します。
  - c. SQLWarning.getSQLState メソッドを呼び出して、SQLSTATE 値を取得します。
  - d. SQLWarning.getErrorCode メソッドを呼び出して、エラー・コード値を取得します。
  - e. DB2 固有の警告情報を取得する場合は、SQLException に関する DB2 固有の情報を取得する場合と同じ手順を実行します。
  - f. SQLWarning.getNextWarning メソッドを呼び出して、次の SQLWarning を取得します。

汎用の SQLWarning 情報を取得するコードを以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```

String url = "jdbc:db2://myhost:9999/myDB:" +
    "retrieveMessagesFromServerOnGetMessage=true;";
// Set properties to retrieve full message
// text

String user = "db2adm";
String password = "db2adm";
java.sql.Connection con =
    java.sql.DriverManager.getConnection (url, user, password)
// Connect to a DB2 for z/OS data source

Statement stmt;
ResultSet rs;
SQLWarning sqlwarn;
...
stmt = con.createStatement(); // Create a Statement object
rs = stmt.executeQuery("SELECT * FROM EMPLOYEE");
// Get the result table from the query
sqlwarn = stmt.getWarnings(); // Get any warnings generated
while (sqlwarn != null) { // While there are warnings, get and
// print warning information
    System.out.println ("Warning description: " + sqlwarn.getMessage());
    System.out.println ("SQLSTATE: " + sqlwarn.getSQLState());
    System.out.println ("Error code: " + sqlwarn.getErrorCode());
    sqlwarn=sqlwarn.getNextWarning(); // Get next SQLWarning
}

```

図 27. `SQLWarning` を処理する例

## BatchUpdateException からの情報の取り出し

バッチに含まれている 1 つのステートメントの実行中にエラーが発生しても、処理は続行します。ただし、`executeBatch` から `BatchUpdateException` がスローされます。

`BatchUpdateException` から情報を取り出すには、以下の手順を実行します。

1. `BatchUpdateException.getUpdateCounts` メソッドを使用して、例外がスローされる前に、バッチに含まれている各 SQL ステートメントによって更新された行の数を確認します。

`getUpdateCount` からは、バッチに含まれている各ステートメントに対応するエレメントが配列として返されます。各エレメントには、以下のいずれかの値が入ります。

$n$  ステートメントによって更新された行の数。

### **Statement.SUCCESS\_NO\_INFO**

この値は、更新された行の数を確認できない場合に返されます。

### **Statement.EXECUTE\_FAILED**

この値は、ステートメントが正常に実行されなかった場合に返されます。

2. `SQLException` のメソッド `getMessage`、`getSQLState`、`getErrorCode` を使用して、最初のエラーの記述、`SQLSTATE`、エラー・コードを取得します。
3. `BatchUpdateException.getNextException` メソッドを使用して、チェーンになっている `SQLException` を取得します。

4. ループの中で、`getMessage`、`getSQLState`、`getErrorCode`、`getNextException` の各メソッド呼び出しを実行して、`SQLException` に関する情報を取得し、次の `SQLException` を取得します。

`BatchUpdateException` の各フィールドと、チェーンになっている `SQLException` オブジェクトを取得するコード断片を以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```
try {
    // Batch updates
} catch (BatchUpdateException buex) {
    System.err.println("Contents of BatchUpdateException:");
    System.err.println(" Update counts: ");
    int [] updateCounts = buex.getUpdateCounts();           1
    for (int i = 0; i < updateCounts.length; i++) {
        System.err.println(" Statement " + i + ":" + updateCounts[i]);
    }
    System.err.println(" Message: " + buex.getMessage());   2
    System.err.println(" SQLSTATE: " + buex.getSQLState());
    System.err.println(" Error code: " + buex.getErrorCode());
    SQLException ex = buex.getNextException();              3
    while (ex != null) {                                     4
        System.err.println("SQL exception:");
        System.err.println(" Message: " + ex.getMessage());
        System.err.println(" SQLSTATE: " + ex.getSQLState());
        System.err.println(" Error code: " + ex.getErrorCode());
        ex = ex.getNextException();
    }
}
```

図 28. `BatchUpdateException` の各フィールドの取得

---

## DB2 JDBC Type 2 ドライバー使用時の `SQLException` の処理

すべての Java プログラムに共通する点ですが、DB2 JDBC Type 2 ドライバーの使用時も、エラー処理は、`try/catch` ブロックで実行します。エラーの発生時にメソッドが例外をスローすると、`catch` ブロックのコードがその例外を処理します。

JDBC には、エラー処理のための `SQLException` クラスが用意されています。すべての JDBC メソッドは、実行中にエラーが発生すると `SQLException` のインスタンスをスローします。JDBC 仕様に従って、`SQLException` オブジェクトには以下の情報が含まれます。

- エラーについての記述を含む `String` オブジェクト、または `NULL`
- `SQLSTATE` を含む `String` オブジェクト、または `NULL`
- エラー・コードを含む `int` 値
- 次の `SQLException` へのポインター、または `NULL`

Linux、UNIX、Windows 用の DB2 JDBC Type 2 ドライバー (DB2 JDBC Type 2 ドライバー) で実行する JDBC プログラムで `SQLException` を処理するための基本手順は、以下のとおりです。

1. `SQLException` を生成できるコードを `try` ブロックに配置します。
2. `catch` ブロックのループの中で以下の手順を実行します。

- a. 最後の `SQLException` を取得したかどうかをテストします。そうでない場合は、次の手順に進みます。
- b. `SQLException` からエラー情報を取得します。
- c. `SQLException.getNextException` メソッドを呼び出して、次の `SQLException` を取得します。

DB2 JDBC Type 2 ドライバーに用意されている DB2 バージョンの `SQLException` を使用した `catch` ブロックのコードを以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```
import java.sql.*;           // Import JDBC API package
...
try {
    // Code that could generate SQLExceptions           1
    ...
} catch(SQLException sqle) {
    while(sqle != null) {           // Check whether there are more           2a
        System.out.println("Message: " + sqle.getMessage());           2b
        System.out.println("SQLSTATE: " + sqle.getSQLState());
        System.out.println("SQL error code: " + sqle.getErrorCode());
        sqle=sqle.getNextException();           // Retrieve next SQLException           2c
    }
}
```

図 29. IBM Data Server Driver for JDBC and SQLJ 使用時の `SQLException` の処理

## DB2 JDBC Type 2 ドライバー使用時の `SQLWarning` の処理

SQL エラーとは異なり、SQL 警告の場合は、JDBC メソッドから例外がスローされません。その代わりに、

`Connection`、`Statement`、`PreparedStatement`、`CallableStatement`、`ResultSet` の各クラスには、`getWarnings` メソッドが含まれています。SQL ステートメントの実行後にそのメソッドを呼び出して、SQL 警告が生成されているかどうかを確認する必要があります。

`getWarnings` を呼び出すことによって、`SQLWarning` オブジェクトを取得できます。

Linux、UNIX、Windows 用の DB2 JDBC Type 2 ドライバー (DB2 JDBC Type 2 ドライバー) は、汎用の `SQLWarning` オブジェクトを生成します。汎用の `SQLWarning` オブジェクトには、以下の情報が含まれています。

- 警告についての記述を含む `String` オブジェクト、または `NULL`
- `SQLSTATE` を含む `String` オブジェクト、または `NULL`
- エラー・コードを含む `int` 値
- 次の `SQLWarning` へのポインター、または `NULL`

SQL 警告の情報を取得するための基本手順は、以下のとおりです。

1. SQL ステートメントを実行するメソッドを呼び出した直後に、`getWarnings` メソッドを呼び出して、`SQLWarning` オブジェクトを取得します。
2. ループの中で以下の手順を実行します。
  - a. `SQLWarning` オブジェクトが `NULL` かどうかをテストします。そうでない場合は、次の手順に進みます。

- b. `SQLWarning.getMessage` メソッドを呼び出して、警告に関する記述を取得します。
- c. `SQLWarning.getSQLState` メソッドを呼び出して、`SQLSTATE` 値を取得します。
- d. `SQLWarning.getErrorCode` メソッドを呼び出して、エラー・コード値を取得します。
- e. `SQLWarning.getNextWarning` メソッドを呼び出して、次の `SQLWarning` を取得します。

汎用の `SQLWarning` 情報を取得するコードを以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```

Connection con;
Statement stmt;
ResultSet rs;
SQLWarning sqlwarn;
...
stmt = con.createStatement(); // Create a Statement object
rs = stmt.executeQuery("SELECT * FROM EMPLOYEE");
sqlwarn = stmt.getWarnings(); // Get the result table from the query
while (sqlwarn != null) { // Get any warnings generated
    // While there are warnings, get and
    // print warning information
    System.out.println ("Warning description: " + sqlwarn.getMessage());
    System.out.println ("SQLSTATE: " + sqlwarn.getSQLState());
    System.out.println ("Error code: " + sqlwarn.getErrorCode());
    sqlwarn=sqlwarn.getNextWarning(); // Get next SQLWarning
}

```

1  
2a  
2b  
2c  
2d  
2e

図 30. `SQLWarning` の処理

## IBM Data Server Driver for JDBC and SQLJ クライアント・リルート・サポート

DB2 の自動クライアント・リルート機能を使用することにより、クライアント・アプリケーションがサーバーとの通信の消失からリカバーして、最小限の中断で処理を続けることができます。JDBC および SQLJ クライアント・アプリケーションはそのサポートを利用できます。

サーバー・クラッシュが発生すると、そのサーバーに接続している各クライアントは通信エラーを受け取り、接続が終了してアプリケーション・エラーが発生します。可用性が重要である場合には、予備セットアップまたはフェイルオーバー・サポートが必要です。フェイルオーバーは、別のサーバーが失敗するときに、操作を引き継ぐサーバーの機能です。どちらの場合も、IBM Data Server Driver for JDBC and SQLJ クライアントは元のサーバーまたは新規サーバーへの接続を再確立しようとします。接続が再確立されると、アプリケーションはトランザクションの失敗を通知する `SQLException` を受信しますが、アプリケーションは次のトランザクションを続行できます。

IBM Data Server Driver for JDBC and SQLJ クライアント・リルート・サポートは、DB2 Database for Linux, UNIX, and Windows または DB2 for z/OS サーバー



に対する IBM Data Server Driver for JDBC and SQLJ Type 4 接続 で使用できません。これは、`javax.sql.DataSource`、`javax.sql.ConnectionPoolDataSource`、`javax.sql.XADataSource`、または `java.sql.DriverManager` インターフェースを使用して得られる接続に有効です。

`DataSource` または `ConnectionPoolDataSource` インターフェースの使用時にクライアント・リルート用に 1 次サーバーおよび代替サーバーを設定するには、`clientRerouteServerListJNDIName` プロパティを設定します。

`DriverManager` インターフェースに使用時にクライアント・リルート用に 1 次サーバーおよび代替サーバーを設定するには、`clientRerouteAlternateServerName` and `clientRerouteAlternatePortNumber` プロパティを設定します。

1 次サーバーおよび代替サーバー用に JDBC ドライバー・プロパティを設定しない場合、1 次サーバーおよび代替サーバーのアドレスが DNS 項目で設定されている場合、クライアント・リルートが行われる可能性があります。1 次サーバーおよび代替サーバーの情報が JDBC ドライバー・メモリー内に存在せず、JNDI が使用されていない場合、IBM Data Server Driver for JDBC and SQLJ は DNS 項目内で 1 次サーバーおよび代替サーバーの情報を検索します。

**制約事項:** `DriverManager` インターフェースで確立される接続用のクライアント・リルート・サポートには、以下の制限があります。

- 同じ URL およびプロパティを指定して接続を作成する場合にのみ、代替サーバー情報が `DriverManager` 接続間で共有されます。
- `DriverManager` 接続には `clientRerouteServerListJNDIName` プロパティまたは `clientRerouteServerListJNDIContext` プロパティを設定できません。
- クライアント・リルートはデフォルト接続 (`jdbc:default:connection`) には使用可能にされていません。

## クライアント・リルート用の DB2 for z/OS サーバーの構成

Sysplex ルーティングが DB2 for z/OS サーバー上で無効になっており、データ共有グループがメンバー固有のアクセス用に設定されている場合、IBM Data Server Driver for JDBC and SQLJ クライアント・リルート・サポートを使用できます。

DB2 データ共有グループで Sysplex ルーティングが使用できる場合は、DB2 for z/OS サーバーおよび Workload Manager for z/OS (WLM) によって、データ共有グループのすべてのメンバー間で処理が分散されます。DB2 データ共有グループのメンバーのいずれかが失敗すると、処理は自動的に他のメンバーに転送されます。この例では、IBM Data Server Driver for JDBC and SQLJ クライアント・リルート・サポートは必要ありません。

メンバー固有のアクセスの設定には、データ共有グループの 1 つ以上のメンバーを表すロケーションの別名を設定することが含まれます。JDBC または SQLJ クライアント・アプリケーションがクライアント・リルート機能を使用する前に、データ共有グループの少なくとも 2 つのメンバーを表すロケーションの別名が存在している必要があります。クライアント・アプリケーションは、ロケーション名を使用してグループ全体に接続する代わりに、ロケーションの別名を使用してそれらのメン



バーに接続します。1つのメンバーが失敗すると、その優先順位 (WLM のサーバー・ウェイト情報) に基づいてデータ共有グループの他のメンバーへの接続が試みられます。

IBM Data Server Driver for JDBC and SQLJ クライアント・リルート・サポートは、メンバー固有のアクセスに対してのみ有効です。これはグループ・アクセスに対しては有効ではありません。このため、メンバーの動的仮想 IP アドレス (DVIPA) のみを定義する必要があります。グループの DVIPA は定義しないでください。

データ共有グループをセットアップして、JDBC および SQLJ クライアント・アプリケーションがクライアント・リルート・サポートを活用できるようにするには、データベース管理者は以下のステップを実行する必要があります。

1. すべてのメンバーが着信 SQL 要求を listen する DRDA ポート番号を指定します。
2. 各メンバーごとに固有の再同期ポート番号を指定します。
3. DRDA リクエスターが接続可能なメンバーのサブセットを指定します。
4. RACF PassTickets が使用される場合、データ共有グループの汎用 LU 名を指定します。

詳しくは、<http://publib.boulder.ibm.com/infocenter/imzic> の Information Management Software for z/OS Solutions Information Center にあるトピック『TCP/IP サーバーとしてのデータ共有グループの構成』を参照してください。

## クライアント上での IBM Data Server Driver for JDBC and SQLJ クライアント・リルート操作

IBM Data Server Driver for JDBC and SQLJ クライアント・リルート・サポートは、`javax.sql.DataSource`、`javax.sql.ConnectionPoolDataSource`、`javax.sql.XADataSource`、または `java.sql.DriverManager` インターフェースを使用して得られる接続に有効です。

クライアント上でのクライアント・リルートは以下のように行われます。

1. データ・ソースに初めて接続する前に、IBM Data Server Driver for JDBC and SQLJ は 1 次サーバーおよび代替サーバーの情報を取得します。
  - a. `clientRerouteAlternateServerName` および `clientRerouteAlternatePortNumber` の `DataSource` プロパティが設定されている場合、IBM Data Server Driver for JDBC and SQLJ は 1 次サーバー値 `serverName` および `portNumber` とともに、これらの値を代替サーバー値としてメモリーにロードします。
  - b. `clientRerouteAlternateServerName` および `clientRerouteAlternatePortNumber` の `DataSource` プロパティが設定されておらず、JNDI ストアが `DB2BaseDataSource` にプロパティ `clientRerouteServerListJNDIName` を設定して構成されている場合、IBM Data Server Driver for JDBC and SQLJ は 1 次サーバーおよび代替サーバーの情報を JNDI ストアからメモリーにロードします。
  - c. `DataSource` プロパティが代替サーバーに設定されておらず、JNDI が構成されていない場合、IBM Data Server Driver for JDBC and SQLJ は DNS 表を

調べて、1 次サーバーおよび代替サーバーの情報を確認します。DNS 情報がある場合、IBM Data Server Driver for JDBC and SQLJ はそれらの値をメモリーにロードします。

- d. 1 次サーバーおよび代替サーバーの情報が使用可能でない場合、接続を確立できません。また、IBM Data Server Driver for JDBC and SQLJ は例外をスローします。
2. IBM Data Server Driver for JDBC and SQLJ は 1 次サーバー名とポート番号を使用してデータ・ソースに接続します。
3. 1 次サーバーへの接続が失敗する場合:
  - a. IBM Data Server Driver for JDBC and SQLJ は、1 次サーバーに再接続しようとしています。
  - b. 1 次サーバーへの再接続が失敗すると、IBM Data Server Driver for JDBC and SQLJ は代替サーバーに接続しようとしています。

1 次サーバーへの再接続のことをフェイルバック と呼びます。代替サーバーへの接続はフェイルオーバー と呼ばれます。

IBM Data Server Driver for JDBC and SQLJ は `maxRetriesForClientReroute` および `retryIntervalForClientReroute` プロパティを使用して、接続が再試行された回数および再試行の間の待機時間を判別します。1 次サーバーおよび代替サーバーへの接続の試行は、1 回の再試行としてカウントされます。

4. 初期接続中にフェイルオーバーが成功すると、ドライバーは `SQLWarning` を生成します。初期接続の後でフェイルオーバーに成功すると、ドライバーはエラー・コード -4498 の `SQLException` をアプリケーションにスローし、代替サーバーへの接続が自動的に再確立され、トランザクションが暗黙的にロールバックされたことをアプリケーションに通知します。その後、アプリケーションは、明示的ロールバックをあらかじめ行う必要なく、トランザクションを再試行できます。

代替サーバーへの接続が成功した後、`Connection` または `DataSource` にはサーバー名とポート番号を除いた元のプロパティが含まれます。さらに、元の接続で変更されたデータ・ソース特殊レジスタがあれば、フェイルオーバー接続で再確立されます。

接続の確立に代替サーバー情報が使用されたかどうかを判別するには、`DB2Connection.alternateWasUsedOnConnect` メソッドを呼び出します。データ・ソースがメモリー内のリストにある情報とは異なる 1 次サーバーおよび代替サーバー情報を戻す場合、メモリー内の 1 次サーバーおよび代替サーバー情報が更新されます。`DB2Connection.alternateWasUsedOnConnect` が戻す値は、`DB2BaseDataSource` の初期構成ではなく、メモリー内の最新の代替情報によって決まります。

## JNDI による IBM Data Server Driver for JDBC and SQLJ クライアント・リルート・サポート

`javax.sql.DataSource`、`javax.sql.ConnectionPoolDataSource`、または `javax.sql.XADataSource` インターフェースで接続する場合、

clientRerouteServerListJNDIName プロパティを設定することによりクライアント・リルート用に JNDI が構成されることを IBM Data Server Driver for JDBC and SQLJ に示します。

また、clientRerouteServerListJNDIContext プロパティを指定することもできます。このプロパティは、DB2ClientRerouteServerList インスタンスのバインディングおよび検索に使用される JNDI コンテキストを提供します。

JNDI によるクライアント・リルートの場合、メモリー内の 1 次サーバーおよび代替サーバーの情報が、javax.naming.Referenceable インターフェースをインプリメントする DB2ClientRerouteServerList クラスのインスタンスです。

DB2ClientRerouteServerList は、以下のプロパティを備えた、シリアライズ可能な Java Bean です。

プロパティ名	データ・タイプ
com.ibm.db2.jcc.DB2ClientRerouteServerList.alternateServerName	String[]
com.ibm.db2.jcc.DB2ClientRerouteServerList.alternatePortNumber	int[]
com.ibm.db2.jcc.DB2ClientRerouteServerList.primaryServerName	String[]
com.ibm.db2.jcc.DB2ClientRerouteServerList.primaryPortNumber	int[]

getXXX および setXXX メソッドは、プロパティごとに定義されます。

DataSource がクライアント・リルート代替情報を保管するために JNDI を使用するように構成されている場合、DataSource の標準サーバーおよびポート・プロパティは getConnection 要求には使用されません。その代わりに、1 次サーバー・アドレスが一時 clientRerouteServerList 情報から取得されます。JNDI のバインドまたは検索の失敗のために JNDI ストアが使用できない場合、IBM Data Server Driver for JDBC and SQLJ は DataSource の標準サーバーおよびポート・プロパティを使用して接続を試行します。JNDI のバインドまたは検索の失敗が発生したことを示すために、警告が累積されます。

フェイルオーバーの後:

- IBM Data Server Driver for JDBC and SQLJ は、更新されたサーバー情報を JNDI ストアに伝搬しようとします。
- DB2ClientRerouteServerList に指定された primaryServerName および primaryPortNumber の値が接続に使用されます。primaryServerName が指定されていない場合、DataSource インスタンスの serverName 値が使用されます。

DB2ClientRerouteServerList を持続させるためにストレージをセットアップするには、以下のステップに従ってください。

1. DB2ClientRerouteServerList のインスタンスを作成して、そのインスタンスを JNDI レジストリーにバインドします。

例:

```
// Create a starting context for naming operations
InitialContext registry = new InitialContext();
// Create a DB2ClientRerouteServerList object
```

```

DB2ClientRerouteServerList address = new DB2ClientRerouteServerList();

// Set the port number and server name for the primary server
address.setPrimaryPortNumber(50000);
address.setPrimaryServerName("mvs1.sj.ibm.com");

// Set the port number and server name for the alternate server
int[] port = {50002};
String[] server = {"mvs3.sj.ibm.com"};
address.setAlternatePortNumber(port);
address.setAlternateServerName(server);

registry.rebind("serverList", address);

```

2. DB2ClientRerouteServerList オブジェクトの JNDI 名を clientRerouteServerListJNDIName プロパティに割り当てます。

例:

```
datasource.setClientRerouteServerListJNDIName("serverList");
```

---

## JDBC アプリケーションでのデータ・ソースからの切断

データ・ソースへの接続を使用した作業が完了したら、データ・ソースへの接続を閉じることが必要です。そうすると、Connection オブジェクトのデータベースと JDBC リソースがすぐに解放されます。

データ・ソースへの接続を閉じるには、close メソッドを使用します。以下に例を示します。

```

Connection con;
...
con.close();

```

DB2 データ・ソースへの接続で、自動コミット・モードがオンになっていない場合に接続を閉じるには、接続が作業単位の境界に達している必要があります。

IBM Informix Dynamic Server データベースへの接続で、そのデータベースがロギングをサポートしており、自動コミット・モードがオンになっていない場合に接続を閉じるには、接続が作業単位の境界に達している必要があります。

---

## 第 4 章 SQLJ アプリケーション・プログラミング

SQLJ アプリケーションの作成は、他の言語での SQL アプリケーションの作成と多くの点で共通しています。

一般に、以下のことを行う必要があります。

- SQLJ と JDBC のメソッドが含まれている Java パッケージをインポートする。
- DB2 表へのデータの送信用またはこの表からのデータの取得用の変数を宣言する。
- データ・ソースに接続する。
- SQL ステートメントを実行する。
- SQL エラーおよび警告を処理する。
- データ・ソースへの接続を切断する。

実行する必要があるタスクは他の言語のものと同様ですが、これらのタスクを実行する方法およびその順序は多少異なります。

---

### 簡単な SQLJ アプリケーションの例

この簡単な SQLJ アプリケーションでは、JDBC アプリケーションに含める必要がある基本要素を示しています。

図 31. 簡単な SQLJ アプリケーション

```
import sqlj.runtime.*;           1
import java.sql.*;

#sql context EzSqljCtx;         3a
#sql iterator EzSqljNameIter (String LASTNAME); 4a

public class EzSqlj {
    public static void main(String args[])
        throws SQLException
    {
        EzSqljCtx ctx = null;
        String URLprefix = "jdbc:db2:";
        String url;
        url = new String(URLprefix + args[0]); // Location name is an input parameter

        String hvmgr="000010";      2
        String hvdeptno="A00";
        try {                        3b
            Class.forName("com.ibm.db2.jcc.DB2Driver");
        } catch (Exception e)
        {
            throw new SQLException("Error in EzSqlj: Could not load the driver");
        }
        try
        {
            System.out.println("About to connect using url: " + url);
            Connection con0 = DriverManager.getConnection(url); 3c
            // Create a JDBC Connection
            con0.setAutoCommit(false); // set autocommit OFF
            ctx = new EzSqljCtx(con0); 3d
        }
    }
}
```

```

try
{
    EzSqljNameIter iter;
    int count=0;

    #sql [ctx] iter =
        {SELECT LASTNAME FROM EMPLOYEE}; // Create result table of the SELECT 4b
    while (iter.next()) { // Retrieve rows from result table 4c
        System.out.println(iter.LASTNAME());
        count++;
    }
    System.out.println("Retrieved " + count + " rows of data");
}
catch( SQLException e ) 5
{
    System.out.println ("**** SELECT SQLException...");
    while(e!=null) {
        System.out.println ("Error msg: " + e.getMessage());
        System.out.println ("SQLSTATE: " + e.getSQLState());
        System.out.println ("Error code: " + e.getErrorCode());
        e = e.getNextException(); // Check for chained exceptions
    }
}
catch( Exception e )
{
    System.out.println("**** NON-SQL exception = " + e);
    e.printStackTrace();
}
try
{
    #sql [ctx] 4d
        {UPDATE DEPARTMENT SET MGRNO=:hvmgr
          WHERE DEPTNO=:hvdeptno}; // Update data for one department 6

    #sql [ctx] {COMMIT}; // Commit the update
}
catch( SQLException e )
{
    System.out.println ("**** UPDATE SQLException...");
    System.out.println ("Error msg: " + e.getMessage() + ". SQLSTATE=" +
        e.getSQLState() + " Error code=" + e.getErrorCode());
    e.printStackTrace();
}
catch( Exception e )
{
    System.out.println("**** NON-SQL exception = " + e);
    e.printStackTrace();
}
iter.close(); // Close the iterator 7
ctx.close();
}
catch(SQLException e)
{
    System.out.println ("**** SQLException ...");
    System.out.println ("Error msg: " + e.getMessage() + ". SQLSTATE=" +
        e.getSQLState() + " Error code=" + e.getErrorCode());
    e.printStackTrace();
}
catch(Exception e)
{
    System.out.println ("**** NON-SQL exception = " + e);
}

```

```

        e.printStackTrace();
    }
}

```

115 ページの図 31 の注:

- | 注                   | 説明  |
|---------------------|---|
| 1                   | これらのステートメントにより、JDBC コア API が含まれた <code>java.sql</code> パッケージ、および SQLJ API が含まれた <code>sqlj.runtime</code> パッケージがインポートされます。アクセスが必要になる可能性のある他のパッケージまたはクラスについては、『SQLJ サポート用の Java パッケージ』を参照してください。   |
| 2                   | <code>String</code> 変数の <code>hvmgr</code> と <code>hvdeptno</code> は、ホスト ID です。これらは、DB2 のホスト変数に相当します。詳細については、『SQLJ アプリケーションでの変数』を参照してください。  |
| 3a、3b、3c、<br>および 3d | これらのステートメントでは、使用可能な 3 つの技法のいずれかを使用して、データ・ソースに接続する方法が示されています。詳細については、『SQLJ を使用したデータ・ソースへの接続』を参照してください。   |
|                     | 手順 3b (JDBC ドライバーのロード) は、JDBC 4.0 を使用する場合は不要です。   |
| 4a、4b、4c、<br>および 4d | これらのステートメントでは、SQLJ で SQL ステートメントを実行する方法が示されています。ステートメント 4a では、SQLJ での SQL カーソルの宣言に相当するものが示されています。ステートメント 4b と 4c では、SQL FETCH の実行に相当することを SQLJ で行う 1 つの方法が示されています。ステートメント 4d では、SQL UPDATE の実行に相当することを SQLJ で行う方法が示されています。詳細については、『SQLJ アプリケーションにおける SQL ステートメント』を参照してください。 |
| 5                   | この <code>try/catch</code> ブロックでは、SQL エラー処理用の <code>SQLException</code> クラスの使用が示されています。SQL エラーの処理の詳細については、『SQLJ アプリケーションでの SQL エラーの処理』を参照してください。SQL 警告の処理の詳細については、『SQLJ アプリケーションでの SQL 警告の処理』を参照してください。  |
| 6                   | これはコメントの例です。SQLJ プログラムへのコメントの組み込みに関する規則については、『SQLJ アプリケーションでのコメント』を参照してください。  |
| 7                   | このステートメントにより、データ・ソースへの接続がクローズされます。『SQLJ アプリケーションでのデータ・ソースへの接続のクローズ』を参照してください。   |

---

## SQLJ を使用したデータ・ソースへの接続

SQLJ アプリケーションの場合も、他の DB2 アプリケーションの場合と同じく、SQL ステートメントを実行する前にデータ・ソースに接続する必要があります。

SQLJ プログラムでデータ・ソースに接続するには、6 つの技法のうちのいずれか 1 つを使用します。そのうち、JDBC の `DriverManager` インターフェースを使用する技法が 2 つ、JDBC の `DataSource` インターフェースを使用する技法が 2 つ、すでに作成されている接続を使用する技法が 1 つ、デフォルトの接続を使用する技法が 1 つあります。



## SQLJ の接続技法 1: JDBC DriverManager インターフェース

SQLJ の接続技法 1 では、接続を作成するための基礎的な手段として JDBC の DriverManager インターフェースを使用します。

SQLJ の接続技法 1 を使用するには、以下の手順を実行します。

1. SQLJ の接続宣言節 を実行します。

そうすると、接続コンテキスト・クラス が生成されます。最も簡単な形式の接続宣言節は、以下のとおりです。

```
#sql context context-class-name;
```

生成される接続コンテキスト・クラスの名前は、*context-class-name* になります。

2. Class.forName メソッドを呼び出して、JDBC ドライバーをロードします。

- IBM Data Server Driver for JDBC and SQLJ の場合は、以下のようして Class.forName を呼び出します。

```
Class.forName("com.ibm.db2.jcc.DB2Driver");
```

JDBC 4.0 ドライバーを使用している場合は、この手順は不要です。

- DB2 JDBC Type 2 ドライバーの場合は、以下のようして Class.forName を呼び出します。

```
Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
```

3. 手順 1 で作成した接続コンテキスト・クラスのコンストラクターを呼び出します。

そうすると、関連するデータ・ソースで実行する各 SQL ステートメントに指定する接続コンテキスト・オブジェクトが作成されます。コンストラクター呼び出しステートメントは、以下のいずれかの形式にする必要があります。

```
connection-context-class connection-context-object=  
new connection-context-class(String url, boolean autocommit);
```

```
connection-context-class connection-context-object=  
new connection-context-class(String url, String user,  
String password, boolean autocommit);
```

```
connection-context-class connection-context-object=  
new connection-context-class(String url, Properties info,  
boolean autocommit);
```

各パラメーターの意味は、以下のとおりです。

*url* データ・ソースに関連したロケーション名を指定する文字列。その引数は、『DriverManager インターフェースと IBM Data Server Driver for JDBC and SQLJ を使用したデータ・ソースへの接続』で指定されているいずれかの形式になります。形式は、使用する JDBC ドライバーによって異なります。

*user* と *password*

接続先のデータ・ソースで接続用のユーザー ID とパスワードが必要であれば、そのユーザー ID とパスワードを指定します。

*info*

接続用の一連のドライバー・プロパティを含んだタイプ `java.util.Properties` のオブジェクトを指定します。Linux、UNIX、Windows 用の DB2 JDBC

Type 2 ドライバー (DB2 JDBC Type 2 ドライバー) の場合は、user プロパティと password プロパティだけを指定します。IBM Data Server Driver for JDBC and SQLJ の場合は、『IBM Data Server Driver for JDBC and SQLJ のプロパティ』でリストされているいずれかのプロパティを指定できます。

#### *autocommit*

各ステートメントの後でデータベース・マネージャーから COMMIT を実行するかどうかを指定します。可能な値は、true または false です。false を指定した場合は、明示的なコミット操作を実行する必要があります。

接続技法 1 を使用してロケーション NEWYORK への接続を作成するためのコードを以下に示します。この接続ではユーザー ID とパスワードが必要ですが、自動コミットは必要ありません。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```
#sql context Ctx;           // Create connection context class Ctx      1
String userid="dbadm";     // Declare variables for user ID and password
String password="dbadm";
String empname;           // Declare a host variable
...
try {                       // Load the JDBC driver
    Class.forName("com.ibm.db2.jcc.DB2Driver");                       2
}
catch (ClassNotFoundException e) {
    e.printStackTrace();
}
Ctx myConnCtx=             3
    new Ctx("jdbc:db2://sysmvs1.st1.ibm.com:5021/NEWYORK",
        userid,password,false); // Create connection context object myConnCtx
                                // for the connection to NEWYORK
#sql [myConnCtx] {SELECT LASTNAME INTO :empname FROM EMPLOYEE
    WHERE EMPNO='000010'};
                                // Use myConnCtx for executing an SQL statement
```

図 32. 接続技法 1 によるデータ・ソース接続

## SQLJ の接続技法 2: JDBC DriverManager インターフェース

SQLJ の接続技法 2 では、接続を作成するための基礎的な手段として JDBC の DriverManager インターフェースを使用します。

SQLJ の接続技法 2 を使用するには、以下の手順を実行します。

1. SQLJ の接続宣言節を実行します。

そうすると、接続コンテキスト・クラスが生成されます。最も簡単な形式の接続宣言節は、以下のとおりです。

```
#sql context context-class-name;
```

生成される接続コンテキスト・クラスの名前は、*context-class-name* になります。

2. Class.forName メソッドを呼び出して、JDBC ドライバーをロードします。
  - IBM Data Server Driver for JDBC and SQLJ の場合は、以下のようにして Class.forName を呼び出します。

```
Class.forName("com.ibm.db2.jcc.DB2Driver");
```

JDBC 4.0 ドライバーを使用している場合は、この手順は不要です。

- DB2 JDBC Type 2 ドライバーの場合は、以下のようにして `Class.forName` を呼び出します。

```
Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
```

3. JDBC の `DriverManager.getConnection` メソッドを呼び出します。

そうすると、データ・ソースに接続するための JDBC 接続オブジェクトが作成されます。『`DriverManager` インターフェースと IBM Data Server Driver for JDBC and SQLJ を使用したデータ・ソースへの接続』で指定されているいずれかの形式の `getConnection` を使用できます。

パラメーター `url`、`user`、`password` の意味は、以下のとおりです。

`url` データ・ソースに関連したロケーション名を指定するストリング。その引数は、『`DriverManager` インターフェースと IBM Data Server Driver for JDBC and SQLJ を使用したデータ・ソースへの接続』で指定されているいずれかの形式になります。形式は、使用する JDBC ドライバーによって異なります。

`user` と `password`

接続先のデータ・ソースで接続用のユーザー ID とパスワードが必要であれば、そのユーザー ID とパスワードを指定します。

4. 手順 1 (119 ページ) で作成した接続コンテキスト・クラスのコンストラクターを呼び出します。

そうすると、関連するデータ・ソースで実行する各 SQL ステートメントに指定する接続コンテキスト・オブジェクトが作成されます。コンストラクター呼び出しステートメントは、以下の形式にする必要があります。

```
connection-context-class connection-context-object=  
new connection-context-class(Connection JDBC-connection-object);
```

`JDBC-connection-object` パラメーターは、手順 3 で作成した `Connection` オブジェクトです。

接続技法 2 を使用してロケーション `NEWYORK` への接続を作成するためのコードを以下に示します。この接続ではユーザー ID とパスワードが必要ですが、自動コミットは必要ありません。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```

#sql context Ctx;           // Create connection context class Ctx 1
String userid="dbadm";     // Declare variables for user ID and password
String password="dbadm";
String empname;           // Declare a host variable
...
try {                       // Load the JDBC driver
    Class.forName("com.ibm.db2.jcc.DB2Driver"); 2
}
catch (ClassNotFoundException e) {
    e.printStackTrace();
}
Connection jdbccon=       3
    DriverManager.getConnection("jdbc:db2://sysmvs1.st1.ibm.com:5021/NEWYORK",
        userid,password);
// Create JDBC connection object jdbccon
jdbccon.setAutoCommit(false); // Do not autocommit
Ctx myConnCtx=new Ctx(jdbccon); 4
// Create connection context object myConnCtx
// for the connection to NEWYORK
#sql [myConnCtx] {SELECT LASTNAME INTO :empname FROM EMPLOYEE
    WHERE EMPNO='000010'};
// Use myConnCtx for executing an SQL statement

```

図 33. 接続技法 2 によるデータ・ソース接続

## SQLJ の接続技法 3: JDBC DataSource インターフェース

SQLJ の接続技法 3 では、接続を作成するための基礎的な手段として JDBC の DataSource を使用します。

SQLJ の接続技法 3 を使用するには、以下の手順を実行します。

1. SQLJ の接続宣言節 を実行します。

そうすると、接続コンテキスト・クラス が生成されます。最も簡単な形式の接続宣言節は、以下のとおりです。

```
#sql context context-class-name;
```

生成される接続コンテキスト・クラスの名前は、*context-class-name* になります。

2. システム管理者が DataSource オブジェクトを別のプログラムで作成した場合は、以下の手順を実行します。そうでない場合は、DataSource オブジェクトを作成して、そのオブジェクトにプロパティを割り当てます。
  - a. 接続する必要があるデータ・ソースの論理名を取得します。
  - b. 次の手順で使用するコンテキストを作成します。
  - c. アプリケーション・プログラムで Java Naming and Directory Interface (JNDI) を使用して、論理データ・ソース名に関連した DataSource オブジェクトを取得します。
3. JDBC の DataSource.getConnection メソッドを呼び出します。

そうすると、データ・ソースに接続するための JDBC 接続オブジェクトが作成されます。以下のいずれかの形式の getConnection を使用できます。

```
getConnection();
getConnection(user, password);
```

パラメーター *user*、*password* の意味は、以下のとおりです。

#### *user* と *password*

接続先のデータ・ソースで接続用のユーザー ID とパスワードが必要であれば、そのユーザー ID とパスワードを指定します。

4. デフォルトの自動コミット・モードが適切でない場合は、JDBC の `Connection.setAutoCommit` メソッドを呼び出します。

そうすることによって、各ステートメントの後でデータベース・マネージャーから `COMMIT` を実行するかどうかを指定できます。そのメソッドの形式は、以下のとおりです。

```
setAutoCommit(boolean autocommit);
```

5. 手順 1 (121 ページ) で作成した接続コンテキスト・クラスのコンストラクターを呼び出します。

そうすると、関連するデータ・ソースで実行する各 SQL ステートメントに指定する接続コンテキスト・オブジェクトが作成されます。コンストラクター呼び出しステートメントは、以下の形式にする必要があります。

```
connection-context-class connection-context-object=  
new connection-context-class(Connection JDBC-connection-object);
```

*JDBC-connection-object* パラメーターは、手順 3 (121 ページ) で作成した `Connection` オブジェクトです。

接続技法 3 を使用して論理名 `jdbc/sampledb` のロケーションへの接続を作成するコードを以下に示します。この例では、システム管理者が JNDI 検索で利用できる `DataSource` オブジェクトを作成してデプロイしたことを想定しています。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```
import java.sql.*;  
import javax.naming.*;  
import javax.sql.*;  
...  
#sql context CtxSqlj;           // Create connection context class CtxSqlj 1  
Context ctx=new InitialContext();  
DataSource ds=(DataSource)ctx.lookup("jdbc/sampledb"); 2b  
Connection con=ds.getConnection(); 2c  
String empname;                // Declare a host variable 3  
...  
con.setAutoCommit(false);      // Do not autocommit 4  
CtxSqlj myConnCtx=new CtxSqlj(con); 5  
// Create connection context object myConnCtx  
#sql [myConnCtx] {SELECT LASTNAME INTO :empname FROM EMPLOYEE  
WHERE EMPNO='000010'};  
// Use myConnCtx for executing an SQL statement
```

図 34. 接続技法 3 によるデータ・ソース接続

## SQLJ の接続技法 4: JDBC DataSource インターフェース

SQLJ の接続技法 4 では、接続を作成するための基礎的な手段として JDBC の DataSource を使用します。この技法では、DataSource を JNDI に登録する必要があります。

SQLJ の接続技法 4 を使用するには、以下の手順を実行します。

1. システム管理者から、接続する必要があるデータ・ソースの論理名を取得します。
2. SQLJ の接続宣言節を実行します。

このタイプの接続では、接続宣言節を以下の形式にする必要があります。

```
#sql public static context context-class-name
with (dataSource="logical-name");
```

接続コンテキストは、`public`、`static` として宣言する必要があります。  
`logical-name` は、手順 1 で取得したデータ・ソース名です。

3. 手順 2 で作成した接続コンテキスト・クラスのコンストラクターを呼び出します。

そうすると、関連するデータ・ソースで実行する各 SQL ステートメントに指定する接続コンテキスト・オブジェクトが作成されます。コンストラクター呼び出しステートメントは、以下のいずれかの形式にする必要があります。

```
connection-context-class connection-context-object=
new connection-context-class();
```

```
connection-context-class connection-context-object=
new connection-context-class (String user,
String password);
```

パラメーター `user`、`password` の意味は、以下のとおりです。

### *user* と *password*

接続先のデータ・ソースで接続用のユーザー ID とパスワードが必要であれば、そのユーザー ID とパスワードを指定します。

接続技法 4 を使用して論理名 `jdbc/sampledb` のロケーションへの接続を作成するコードを以下に示します。この接続では、ユーザー ID とパスワードが必要です。

```
#sql public static context Ctx
with (dataSource="jdbc/sampledb"); 2
// Create connection context class Ctx
String userid="dbadm"; // Declare variables for user ID and password
String password="dbadm";

String empname; // Declare a host variable
...
Ctx myConnCtx=new Ctx(userid, password); 3
// Create connection context object myConnCtx
// for the connection to jdbc/sampledb
#sql [myConnCtx] {SELECT LASTNAME INTO :empname FROM EMPLOYEE
WHERE EMPNO='000010'};
// Use myConnCtx for executing an SQL statement
```

図 35. 接続技法 4 によるデータ・ソース接続

## SQLJ の接続技法 5: すでに作成されている接続の使用

SQLJ の接続技法 5 では、すでに作成されている接続を使用してデータ・ソースに接続します。

通常は、1 つのプログラムで接続コンテキスト・クラスを宣言し、接続コンテキストを作成し、その接続コンテキストをパラメーターとして他のプログラムに渡すのが一般的です。接続コンテキストを使用するプログラムは、渡された接続コンテキスト・オブジェクトを引数としてコンストラクターを呼び出します。

プログラム CtxGen.sqlj は、接続コンテキスト Ctx を宣言し、インスタンス oldCtx を作成します。

```
#sql context Ctx;
...
// Create connection context object oldCtx
```

プログラム test.sqlj は、oldCtx をパラメーターとして受け取り、その oldCtx を接続コンテキスト・コンストラクターの引数として使用します。

```
void useContext(sqlj.runtime.ConnectionContext oldCtx)
    // oldCtx was created in CtxGen.sqlj
{
    Ctx myConnCtx=
        new Ctx(oldCtx);           // Create connection context object myConnCtx
    // from oldCtx
    #sql [myConnCtx] {SELECT LASTNAME INTO :empname FROM EMPLOYEE
        WHERE EMPNO='000010'};
    // Use myConnCtx for executing an SQL statement
    ...
}
```

## SQLJ の接続技法 6: デフォルトの接続の使用

SQLJ の接続技法 6 では、デフォルトの接続を使用してデータ・ソースに接続します。この技法を使用するのは、データベース・スレッドを別のリソース・マネージャー (Java ストアード・プロシージャ環境など) によって制御する場合に限ってください。

デフォルトの接続を使用する場合は、接続コンテキスト・オブジェクトなしで SQL ステートメントを指定します。この技法を使用すると、プログラムで JDBC インターフェースを明示的に使用する場合以外は、JDBC ドライバーをロードする必要がありません。

デフォルトの接続コンテキストは、以下のいずれかになります。

- 論理名 jdbc/defaultDataSource にバインドされているデータ・ソースに関連した接続コンテキスト。
- `ConnectionContext.setDefaultContext` メソッドでデフォルトの接続コンテキストとして設定されている、明示的に作成された接続コンテキスト。この方法でデフォルトの接続コンテキストを作成することは、お勧めできません。

以下の SQLJ 実行節には接続コンテキストがないので、デフォルトの接続コンテキストが使用されます。



```
#sql {SELECT LASTNAME INTO :empname FROM EMPLOYEE
      WHERE EMPNO='000010'}; // Use default connection for
                          // executing an SQL statement
```

---

## SQLJ サポート用の Java パッケージ

SQLJ ステートメントを実行するか、または SQLJ プログラムで JDBC メソッドを呼び出す場合は、事前にそれらのステートメントをサポートする各種 Java パッケージのすべてまたは一部にアクセスできるようにしておく必要があります。

これは、パッケージまたは特定のクラスをインポートするか、または完全修飾クラス名を使用して実行できます。SQLJ プログラムには、以下のパッケージまたはクラスが必要になる場合があります。

### **sqlj.runtime**

SQLJ ランタイム API が含まれています。

### **java.sql**

コア JDBC API が含まれます。

### **com.ibm.db2.jcc**

JDBC および SQLJ のドライバー固有のインプリメンテーションが含まれています。

### **javax.naming**

Java Naming and Directory Interface (JNDI) 検索の実行用のメソッドが含まれています。

### **javax.sql**

DataSource オブジェクトの作成用のメソッドが含まれています。

---

## SQLJ アプリケーションでの変数

他の言語の DB2 プログラムでは、ホスト変数を使用してアプリケーション・プログラムと DB2 の間のデータの受け渡しを行います。SQLJ プログラムでは、ホスト変数またはホスト式を使用できます。

ホスト式はコロン (:) で始まります。コロンの後にオプション・パラメーター・モード ID (IN、OUT、または INOUT) が続き、その後に括弧で囲んだ式の節が続きます。

ホスト変数およびホスト式には大文字と小文字の区別があります。

複合式とは、単一値に評価する配列エレメントまたは Java 式です。SQLJ 節での複合式は括弧で囲む必要があります。

次の例は、ホスト式の使用法を表したものです。

例: Java ID の宣言および SELECT ステートメントでの Java ID の使用

この例では、#sql で始まるステートメントには他の言語の SELECT ステートメントと同じ機能があります。このステートメントは、従業員番号 000010 の従業員の姓を Java ID empname に割り当てます。

```
String empname;
...
#sql [ctxt]
{SELECT LASTNAME INTO :empname FROM EMPLOYEE WHERE EMPNO='000010'};
```

例: Java ID の宣言およびストアード・プロシージャ呼び出しでの Java ID の使用

この例では、#sql で始まるステートメントには他の言語の SQL CALL ステートメントと同じ機能があります。このステートメントは Java ID empno をストアード・プロシージャ A への入力パラメーターとして使用します。empno の前の値 IN は、empno が入力パラメーターであることを示します。CALL ステートメントでのパラメーターの場合、IN がデフォルトです。パラメーターが使用される方法を示す明示的またはデフォルトの修飾子 (IN、OUT、または INOUT) は、ストアード・プロシージャの CREATE PROCEDURE ステートメントで指定したパラメーター定義の対応する値と一致しなければなりません。

```
String empno = "0000010";
...
#sql [ctxt] {CALL A (:IN empno)};
```

例: ホスト ID としての複合式の使用

この例は、複合式 (((int)yearsEmployed++/5)\*500) をホスト式として使用します。

```
#sql [ctxt] {UPDATE EMPLOYEE
SET BONUS=:(((int)yearsEmployed++/5)*500) WHERE EMPNO=:empID};
```

SQLJ は複合ホスト式を処理する際に以下のアクションを実行します。

- ステートメント内の各ホスト式を、それぞれの値をデータベースに割り当てる前に左から右に評価します。
- 副次作用 (postfix 演算子を指定した演算など) を、通常の Java 規則に従って評価します。すべてのホスト式の値が完全に評価されてから、それらの値が DB2 に渡されます。
- 丸めおよび切り捨てに Java 規則を使用します。

従って、UPDATE ステートメントが実行される前に yearsEmployed の値が 6 である場合、UPDATE ステートメントによって列 BONUS に割り当てられる値は ((int)6/5)\*500、または 500 です。500 が BONUS に割り当てられた後、yearsEmployed の値が増やされます。

**変数名に関する制限:** SQLJ プログラムでは 2 つのストリングに特殊な意味があります。それらのストリングを SQLJ プログラムで使用する場合は以下の制限を守ってください。

- ストリング `__sJT_` は SQLJ によって生成される変数名のために予約された接頭部です。以下のタイプの名前を `__sJT_` で始めないでください。
  - ホスト式の名前
  - 実行可能 SQL ステートメントを含むブロックで宣言されている Java 変数名
  - 実行可能 SQL ステートメントを含むメソッドのパラメーターの名前
  - 実行可能 SQL ステートメントを含むクラス内のフィールドの名前、あるいは実行可能 SQL ステートメントを含むサブクラスまたは囲みクラスを持つクラス内のフィールドの名前

- スtring `_SJ` は SQLJ によって生成されるリソース・ファイルおよびクラスのために予約された接尾部です。クラス名および入力ソース・ファイル名では String `_SJ` を使用しないでください。

---

## SQLJ アプリケーションでのコメント

SQLJ プログラムで説明を記述するには、コメントを組み込む必要があります。これを行うには、Java のコメントを使用します。Java でのコメントは、`/* */` または `//` で示します。

Java のコメントは、SQLJ 節の外側に組み込みますが、Java 言語で許可される場所であればどこにでも組み込むことができます。SQLJ 節内で、Java のコメントを使用できるのは、以下の場所です。

- ホスト式 (`/* */` または `//`) 内。
- データ・ソースで SQL ステートメント内のコメント (`/* */` または `--`) がサポートされる場合は、実行可能節中の SQL ステートメント内。

SQL ステートメント内の `/*` および `*/` の組みは、ネストすることができます。

---

## SQLJ アプリケーションでの SQL ステートメントの実行

従来の SQL プログラムでは SQL ステートメントを実行して、表の作成、表へのデータの挿入、更新、削除、またはマージ、表からのデータの取得、ストアード・プロシージャの呼び出し、トランザクションのコミットまたはロールバックを行います。SQLJ プログラムでは、さらにこれらのステートメントを SQLJ 実行可能節内で実行します。

実行可能節は、以下のいずれかの一般形式にすることができます。

```
#sql [connection-context] {sql-statement};  
#sql [connection-context,execution-context] {sql-statement};  
#sql [execution-context] {sql-statement};
```

### execution-context の指定

実行可能節では、常に明示接続コンテキストを指定する必要があります。ただし、1 つの例外として、FETCH ステートメントには明示接続コンテキストを指定しません。特定の場合に限り、実行コンテキストを組み込みます。実行コンテキストが必要となる場合については、『SQLJ での SQL ステートメントの実行の制御』を参照してください。

### connection-context の指定

実行可能節では、接続コンテキストを明示的に指定しない場合、実行可能節はデフォルト接続コンテキストを使用します。

## SQLJ アプリケーションでの DB2 オブジェクトの作成および変更

SQLJ 実行可能節を使用して、データ定義ステートメント

(CREATE、ALTER、DROP、GRANT、REVOKE)、INSERT ステートメント、検索または位置指定の UPDATE ステートメント、検索または位置指定の DELETE ステートメントを実行できます。

INSERT、検索の UPDATE、検索の DELETE の実行可能ステートメントを以下に示します。

```
#sql [myConnCtx] {INSERT INTO DEPARTMENT VALUES
("X00","Operations 2","000030","E01",NULL)};
#sql [myConnCtx] {UPDATE DEPARTMENT
SET MGRNO="000090" WHERE MGRNO="000030"};
#sql [myConnCtx] {DELETE FROM DEPARTMENT
WHERE DEPTNO="X00"};
```

## SQLJ アプリケーションでの位置指定 UPDATE および DELETE 操作の実行

SQLJ で位置指定の UPDATE と DELETE を実行する操作は、他の言語の DB2 アプリケーションの場合と同じく、結果表から行を取得する操作を拡張した操作になります。

基本手順は、以下のとおりです。

### 1. イテレーターを宣言します。

イテレーターには、位置指定と名前指定があります。位置指定の UPDATE 操作または DELETE 操作では、イテレーターを更新可能として宣言する必要があります。そのためには、宣言の中に以下の節を組み込まなければなりません。

#### **implements sqlj.runtime.ForUpdate**

この節を指定すると、生成されたイテレーター・クラスに更新可能なイテレーターを使用するためのメソッドが組み込まれます。位置指定の UPDATE 操作または DELETE 操作を組み込んだプログラムでは、この節が必須になります。

#### **with (updateColumns="column-list")**

この節では、イテレーターが更新する結果表の各列のコマ区切りリストを指定します。この節はオプションです。

イテレーターは `public` として宣言する必要があるため、`public` イテレーターの宣言と使用を同じファイルに記述する場合も別々のファイルに記述する場合も、以下の規則に従わなければなりません。

ファイルの中でイテレーターを単独で宣言する場合は、そのイテレーターに対するアドレス可能性があり、生成されたクラスをインポートする SQLJ ソース・ファイルから、そのイテレーターを使用してデータを取得し、位置指定の UPDATE ステートメントまたは DELETE ステートメントを実行できます。位置指定の UPDATE ステートメントまたは DELETE ステートメントを実行するための許可 ID は、ステートメントを静的に実行するか動的に実行するかによって異なります。ステートメントを静的に実行する場合の許可 ID は、そのステートメントが含まれているプランまたはパッケージの所有者になります。ステートメントを動的に実行する場合の許可 ID は、有効な DYNAMICRULES の動作によって決まります。IBM Data Server Driver for JDBC and SQLJ の場合、その動作は常に DYNAMICRULES BIND になります。

### 2. 接続の自動コミット・モードを使用不可にします。

自動コミット・モードが使用可能になっていると、位置指定の UPDATE ステートメントが実行されるたびに COMMIT 操作が実行され、イテレーターに `with`

(holdability=true) 属性が設定されている場合以外はイテレーターが破棄されることとなります。したがって、イテレーターを使用する作業が完了するまで COMMIT 操作が実行されないようにするには、自動コミットをオフにしなければなりません。各更新操作の後に COMMIT を実行する場合は、各 COMMIT 操作の後にイテレーターが破棄されないようにするために、イテレーターを with (holdability=true) で宣言するという代替方法を使用してください。

3. イテレーター・クラスのインスタンスを作成します。

更新可能でないイテレーターの場合と同じ手順です。

4. SELECT の結果表をイテレーターのインスタンスに割り当てます。

更新可能でないイテレーターの場合と同じ手順です。FOR UPDATE 節が含まれている SELECT ステートメントは、使用できません。

5. 行を取得して更新します。

位置指定イテレーターの場合は、そのために以下のアクションをループの中で実行します。

- a. 実行可能節で FETCH ステートメントを実行して現在行を取得します。
- b. イテレーターが結果表の行を参照しているかどうかをテストするために、PositionedIterator.endFetch メソッドを呼び出します。
- c. イテレーターが結果表の行を参照していれば、実行可能節で SQL の UPDATE... WHERE CURRENT OF :iterator-object ステートメントを実行して、現在行の各列を更新します。現在行を削除する場合は、実行可能節で SQL の DELETE ... WHERE CURRENT OF :iterator-object ステートメントを実行します。

名前指定イテレーターの場合は、そのために以下のアクションをループの中で実行します。

- a. next メソッドを呼び出して、イテレーターを前方へ移動します。
- b. イテレーターが結果表の行を参照しているかどうかをテストするために、next から true が返されるかどうかをチェックします。
- c. 実行可能節で SQL の UPDATE... WHERE CURRENT OF :iterator-object ステートメントを実行して、現在行の各列を更新します。現在行を削除する場合は、実行可能節で SQL の DELETE ... WHERE CURRENT OF iterator-object ステートメントを実行します。

6. イテレーターを閉じます。

そのためには、close メソッドを使用します。

位置指定イテレーターを宣言し、そのイテレーターを位置指定の UPDATE で使用するコードを以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

まず、1 つのファイルで位置指定イテレーター UpdByPos を宣言し、そのイテレーターで列 SALARY を更新することを指定します。

```
import java.math.*; // Import this class for BigDecimal data type
#sql public iterator UpdByPos implements sqlj.runtime.ForUpdate
    with(updateColumns="SALARY") (String, BigDecimal);
```

1

図 36. 位置指定の UPDATE で使用する位置指定イテレーターを宣言する例

次に、別のファイルで、位置指定の UPDATE のために UpdByPos を使用します。そのためのコード断片は、以下のとおりです。

```
import sqlj.runtime.*; // Import files for SQLJ and JDBC APIs
import java.sql.*;
import java.math.*; // Import this class for BigDecimal data type
import UpdByPos; // Import the generated iterator class that
                // was created by the iterator declaration clause
                // for UpdByName in another file
#sql context HSCTX; // Create a connection context class HSCTX
public static void main (String args[])
{
    try {
        Class.forName("com.ibm.db2.jcc.DB2Driver");
    }
    catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
    Connection HSjdbccon=
    DriverManager.getConnection("jdbc:db2:SANJOSE");
    // Create a JDBC connection object
    HSjdbccon.setAutoCommit(false);
    // Set autocommit off so automatic commits
    // do not destroy the cursor between updates
    HSCTX myConnCtx=new HSCTX(HSjdbccon);
    // Create a connection context object
    UpdByPos upditer; // Declare iterator object of UpdByPos class
    String enum; // Declares host variable to receive EMPNO
    BigDecimal sal; // and SALARY column values
    #sql [myConnCtx]
        upditer = {SELECT EMPNO, SALARY FROM EMPLOYEE
            WHERE WORKDEPT='D11'};
    // Assign result table to iterator object
    #sql {FETCH :upditer INTO :enum,:sal};
    // Move cursor to next row
    while (!upditer.endFetch())
    // Check if on a row
    {
        #sql [myConnCtx] {UPDATE EMPLOYEE SET SALARY=SALARY*1.05
            WHERE CURRENT OF :upditer};
        // Perform positioned update
        System.out.println("Updating row for " + enum);
        #sql {FETCH :upditer INTO :enum,:sal};
        // Move cursor to next row
    }
    upditer.close(); // Close the iterator
    #sql [myConnCtx] {COMMIT};
    // Commit the changes
    myConnCtx.close(); // Close the connection context
}
```

2

3

4

5a

5b

5c

6

図 37. 位置指定イテレーターで位置指定の UPDATE を実行する例

名前指定イテレーターを宣言し、そのイテレーターを位置指定の UPDATE で使用するコードを以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。



まず、1つのファイルで名前指定イテレーター `UpdByName` を宣言し、そのイテレーターで列 `SALARY` を更新することを指定します。

```
import java.math.*;           // Import this class for BigDecimal data type
#sql public iterator UpdByName implements sqlj.runtime.ForUpdate 1
    with(updateColumns="SALARY") (String EmpNo, BigDecimal Salary);
```

図 38. 位置指定の `UPDATE` で使用する名前指定イテレーターを宣言する例

次に、別のファイルで、位置指定の `UPDATE` のために `UpdByName` を使用します。そのためのコード断片は、以下のとおりです。

```
import sqlj.runtime.*;        // Import files for SQLJ and JDBC APIs
import java.sql.*;
import java.math.*;          // Import this class for BigDecimal data type
import UpdByName;            // Import the generated iterator class that
                              // was created by the iterator declaration clause
                              // for UpdByName in another file
#sql context HSCTX;          // Create a connection context class HSCTX
public static void main (String args[])
{
    try {
        Class.forName("com.ibm.db2.jcc.DB2Driver");
    }
    catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
    Connection HSjdbccon=
    DriverManager.getConnection("jdbc:db2:SANJOSE");
    HSjdbccon.setAutoCommit(false); // Create a JDBC connection object
    HSCTX myConnCtx=new HSCTX(HSjdbccon); // Set autocommit off so automatic commits 2
    UpdByName upditer; // do not destroy the cursor between updates
    String enum; // Create a connection context object
    #sql [myConnCtx] // Declare iterator object of UpdByName class
        upditer = {SELECT EMPNO, SALARY FROM EMPLOYEE 3
                    WHERE WORKDEPT='D11'}; // Declare host variable to receive EmpNo
    while (upditer.next()) // column values
    { // Assign result table to iterator object 4
        enum = upditer.EmpNo(); // Get employee number from current row
        #sql [myConnCtx] // Move cursor to next row and 5a,5b
            {UPDATE EMPLOYEE SET SALARY=SALARY*1.05
              WHERE CURRENT OF :upditer}; // Perform positioned update
        System.out.println("Updating row for " + enum);
    }
    upditer.close(); // Close the iterator 6
    #sql [myConnCtx] {COMMIT}; // Commit the changes
    myConnCtx.close(); // Close the connection context
}
```

図 39. 名前指定イテレーターで位置指定の `UPDATE` を実行する例



## SQLJ アプリケーションでの、位置指定 UPDATE または DELETE 操作の変数としてのイテレーターの受け渡し

SQLJ はイテレーターを変数としてメソッド間で渡すことを許可します。

位置指定 UPDATE または DELETE ステートメントに使用されるイテレーターは、実行時にのみ識別できます。同じ SQLJ 位置指定 UPDATE または DELETE ステートメントを別のイテレーターと共に実行時に使用できます。SQLJ アプリケーションをプログラム準備処理の一部としてカスタマイズする際に `-staticpositioned` に値 `YES` を指定する場合、SQLJ カスタマイザーは位置指定 UPDATE または DELETE ステートメントを静的に実行する準備を行います。この場合、カスタマイザーはどのイテレーターがどの位置指定 UPDATE または DELETE ステートメントに属するのかを判別する必要があります。SQLJ カスタマイザーは、イテレーター・データ・タイプを UPDATE または DELETE ステートメント内のデータ・タイプにマッピングさせることによってこれを行います。ただし、UPDATE または DELETE ステートメント内の表からイテレーター・クラスへの固有のマッピングが存在しない場合、SQLJ カスタマイザーはどのイテレーターと UPDATE または DELETE ステートメントが組になるのかを正確には判別できません。SQLJ カスタマイザーはイテレーターと UPDATE または DELETE ステートメントを任意のペアにしなればならず、その結果として SQL エラーが生じることもあります。以下のコード・フラグメントはこの点を示しています。

```
#sql iterator GeneralIter implements sqlj.runtime.ForUpdate
( String );

public static void main ( String args[] )
{
...
    GeneralIter iter1 = null;
    #sql [ctxt] iter1 = { SELECT CHAR_COL1 FROM TABLE1 };

    GeneralIter iter2 = null;
    #sql [ctxt] iter2 = { SELECT CHAR_COL2 FROM TABLE2 };
...

    doUpdate ( iter1 );
}

public static void doUpdate ( GeneralIter iter )
{
    #sql [ctxt] { UPDATE TABLE1 ... WHERE CURRENT OF :iter };
}
```

図 40. 失敗する静的な位置指定 UPDATE

この例では、1 つのイテレーターのみ定義されます。そのイテレーターの 2 つのインスタンスが定義され、それぞれが、異なる表からデータを検索する異なる SELECT ステートメントに関連付けられます。カスタマイズおよび `-staticpositioned YES` とのバインドの際に、SQLJ は 2 つの `DECLARE CURSOR` ステートメント (各 SELECT ステートメントに対して 1 つずつ) を作成し、UPDATE ステートメントをそれぞれのカーソルにバインドしようとしています。しかし、バインド処理は、

UPDATE TABLE1 ... WHERE CURRENT OF :iter が SELECT CHAR\_COL2 FROM TABLE2 のカーソルにバインドされるとき、UPDATE の表がカーソルの表と一致しないために、SQLCODE -509 で失敗します。

132 ページの図 40 のようなプログラムでバインド時のエラーを避けるには、バインド・オプション `SQLERROR(CONTINUE)` を指定します。しかし、この手法には、プログラム内に SQL エラーがある場合でも、DB2 データベース・マネージャーがパッケージを作成してしまうという欠点があります。より優れた手法は、位置指定 UPDATE または DELETE ステートメント内の表とイテレーター・クラスの間 1 対 1 のマッピングが存在するようにプログラムを作成することです。図 41 はこれを行う方法の例を示しています。

```
#sql iterator Table2Iter(String);
#sql iterator Table1Iter(String);
public static void main ( String args[] )
{
    ...
    Table2Iter iter2 = null;
    #sql [ctxt] iter2 = { SELECT CHAR_COL2 FROM TABLE2 };

    Table1Iter iter1 = null;
    #sql [ctxt] iter1 = { SELECT CHAR_COL1 FROM TABLE1 };
    ...

    doUpdate(iter1);
}

public static void doUpdate ( Table1Iter iter )
{
    ...
    #sql [ctxt] { UPDATE TABLE1 ... WHERE CURRENT OF :iter };
    ...
}

public static void doUpdate ( Table2Iter iter )
{
    ...
    #sql [ctxt] { UPDATE TABLE2 ... WHERE CURRENT OF :iter };
    ...
}
```

図 41. 成功する静的な位置指定 UPDATE

このコーディング方法を使用すると、各イテレーター・クラスは 1 つの表にのみ関連付けられます。そのため、DB2 バインド処理では、常に位置指定 UPDATE ステートメントを有効なイテレーターに関連付けることができます。

## SQLJ アプリケーションでのバッチ更新の実行

IBM Data Server Driver for JDBC and SQLJ は、SQLJ のバッチ更新をサポートしています。表の各行を 1 行ずつ更新する代わりにバッチ更新を使用すれば、SQLJ によって一群の更新を同時に実行できます。

バッチ更新には、以下のタイプのステートメントを組み込みます。

- 検索の INSERT、UPDATE、DELETE の各ステートメント
- CREATE、ALTER、DROP、GRANT、REVOKE の各ステートメント
- 入力パラメーターだけを指定した CALL ステートメント

JDBC の場合とは異なり、SQLJ では、入力パラメーターまたはホスト式を含んだステートメントを組み込んだ異種バッチを使用できます。したがって、SQLJ バッチでは、以下の項目を自由に組み合わせることができます。

- 同じステートメントの各インスタンス
- 別々のステートメント
- 異なる数の入力パラメーターまたはホスト式を含んだステートメント
- 異なるデータ・タイプの入力パラメーターまたはホスト式を含んだステートメント
- 入力パラメーターまたはホスト式がないステートメント

バッチに含まれている 1 つのステートメントの実行中にエラーが発生しても、残りのステートメントは実行され、バッチ内のすべてのステートメントが実行されてから、`BatchUpdateException` がスローされます。`BatchUpdateException` を処理する方法については、『`BatchUpdateException` からの情報の取り出し』を参照してください。

警告に関する情報を取得するには、`executeBatch` メソッドを実行したオブジェクトに対して `Statement.getWarnings` メソッドを使用します。これで、各 `SQLWarning` オブジェクトのエラーに関する記述、`SQLSTATE`、およびエラー・コードを取得できます。

バッチに追加できないステートメントがプログラムに含まれているために、バッチが暗黙的に実行される場合は、バッチが実行されてから、その新しいステートメントが処理されます。バッチの実行中にエラーが発生すると、バッチ実行の原因になったステートメントは実行されません。

ステートメント・バッチの作成、実行、削除の基本手順は、以下のとおりです。

1. 接続の `AutoCommit` を使用不可にします。

そのようにしておくこと、バッチの実行中にエラーが発生した場合に、既に実行されたステートメントの変更内容をコミットするかどうかを自分で制御できます。

2. 実行コンテキストを取得します。

バッチの中で実行するすべてのステートメントは、その実行コンテキストを使用します。

3. `ExecutionContext.setBatching(true)` メソッドを呼び出して、バッチを作成します。

手順 2 で作成した実行コンテキストに関連した後続のバッチ可能ステートメントは、その後の実行時にバッチに追加されます。

バッチ互換でないステートメント・セットを並列的にバッチに組み込む場合は、バッチ互換ステートメントの各セットごとに実行コンテキストを作成する必要があります。

4. バッチに含める SQL ステートメントの `SQLJ` 実行可能節を組み込みます。

それらの節には、手順 2 で作成した実行コンテキストを組み込む必要があります。

SQLJ 実行可能節に入力パラメーターまたはホスト式が含まれている場合は、バッチの中にそのステートメントを複数回組み込んで、それぞれに入力パラメーターまたはホスト式の別々の値を適用することもできます。

ステートメントが既存のバッチに追加されたかどうか、ステートメントが新しいバッチの最初のステートメントかどうか、ステートメントが実行されたのはバッチの中か外かを確認するには、`ExecutionContext.getUpdateCount` メソッドを呼び出します。このメソッドは、以下のいずれかの値を返します。

#### **ExecutionContext.ADD\_BATCH\_COUNT**

ステートメントが既存のバッチに追加された場合は、この定数が返されません。

#### **ExecutionContext.NEW\_BATCH\_COUNT**

ステートメントが新しいバッチの最初のステートメントの場合は、この定数が返されます。

#### **ExecutionContext.EXEC\_BATCH\_COUNT**

ステートメントがバッチに含まれている状態でバッチが実行された場合は、この定数が返されます。

#### **その他の整数**

この値は、ステートメントにより更新された行の数です。ステートメントがバッチに追加されない状態で実行された場合は、この値が返されます。

### 5. 明示的または暗黙的にバッチを実行します。

- バッチを明示的に実行するには、`ExecutionContext.executeBatch` メソッドを呼び出します。

`executeBatch` からは、バッチに含まれている各ステートメントによって更新された行の数を含んだ整数配列が返されます。配列内の各エレメントの順序は、各ステートメントをバッチに追加した順序に対応します。

- バッチは、以下の状況では暗黙的に実行されます。
  - 既にバッチに含まれているステートメントとの互換性のないバッチ可能ステートメントをプログラムに組み込んだ場合。この場合、SQLJ は、既にバッチに含まれているステートメントを実行し、非互換ステートメントを組み込んだ新しいバッチを作成します。SQLJ は、バッチに含まれているステートメントとの互換性のないステートメントも実行します。
  - バッチ可能でないステートメントをプログラムに組み込んだ場合。この場合、SQLJ は、既にバッチに含まれているステートメントを実行します。SQLJ は、バッチ可能でないステートメントも実行します。
  - `ExecutionContext.setBatchLimit(n)` メソッドを呼び出した後に、バッチにステートメントを追加して、バッチ内のステートメントの数が  $n$  以上になる場合。 $n$  は、以下のいずれかの値になります。

#### **ExecutionContext.UNLIMITED\_BATCH**

この定数を指定した場合は、バッチ可能ではあっても非互換のステートメント、またはバッチ可能でないステートメントを SQLJ が検出した場合にのみ、バッチが暗黙的に実行されます。この値を設定することは、`setBatchLimit` を呼び出さないことと同じです。

### ExecutionContext.AUTO\_BATCH

この定数を指定した場合は、バッチ内のステートメントの数が SQLJ によって設定されている数に達した場合に、バッチが暗黙的に実行されます。

#### 正の整数

この数のステートメントがバッチに追加されると、SQLJ によってバッチが暗黙的に実行されます。ただし、バッチ可能ではあっても非互換のステートメント、またはバッチ可能でないステートメントを SQLJ が検出した場合は、その数のステートメントが追加される前にバッチが実行される可能性があります。

暗黙的に実行されたバッチで更新された行数を確認するには、`ExecutionContext.getBatchUpdateCounts` メソッドを呼び出します。

`getBatchUpdateCounts` からは、バッチに含まれている各ステートメントによって更新された行数を含んだ整数配列が返されます。配列内の各エレメントの順序は、各ステートメントをバッチに追加した順序に対応します。それぞれの配列エレメントには、以下のいずれかの値が入ります。

- 2 この値は、SQL ステートメントは正常に実行されたが、更新された行数は判別できなかったことを示します。
- 3 この値は SQL ステートメントが失敗したことを示します。

#### その他の整数

この値は、ステートメントにより更新された行数です。

6. オプションとして、すべてのステートメントをバッチに追加した後に、バッチ処理を使用不可にすることもできます。

そのためには、`ExecutionContext.setBatching(false)` メソッドを呼び出します。バッチ処理を使用不可にしても、暗黙的または明示的にバッチを実行することは可能ですが、バッチにさらにステートメントを追加することができなくなります。バッチが既に存在する場合に、バッチ互換ステートメントをバッチに追加する代わりにそのまま実行することが望ましい状況では、バッチ処理を使用不可にすると便利です。

バッチを実行しないでクリアする場合は、`ExecutionContext.cancel` メソッドを呼び出します。

7. バッチ実行が暗黙的だった場合は、最後に明示的な `executeBatch` を実行して、すべてのステートメントを確実に実行するようにします。

バッチの中で `UPDATE` を実行して、すべてのマネージャー (管理職) に昇給を適用するコード断片を以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```

#sql iterator GetMgr(String);           // Declare positioned iterator
{
  GetMgr deptiter;                     // Declare object of GetMgr class
  String mgrnum = null;                 // Declare host variable for manager number
  int raise = 400;                     // Declare raise amount
  int currentSalary;                   // Declare current salary
  String url, username, password;      // Declare url, user ID, password
  ...
  TestContext c1 = new TestContext (url, username, password, false); 1
  ExecutionContext ec = new ExecutionContext();                        2
  ec.setBatching(true);                                              3

#sql [c1] deptiter =
  {SELECT MGRNO FROM DEPARTMENT};
                                     // Assign the result table of the SELECT
                                     // to iterator object deptiter
#sql {FETCH :deptiter INTO :mgrnum};
                                     // Retrieve the first manager number
while (!deptiter.endFetch()) {      // Check whether the FETCH returned a row
#sql [c1]
  {SELECT SALARY INTO :currentSalary FROM EMPLOYEE
   WHERE EMPNO=:mgrnum};
#sql [c1, ec]                               4
  {UPDATE EMPLOYEE SET SALARY=:(currentSalary+raise)
   WHERE EMPNO=:mgrnum};
#sql {FETCH :deptiter INTO :mgrnum };
                                     // Fetch the next row
}
ec.executeBatch();                    5
ec.setBatching(false);                6
#sql [c1] {COMMIT};
deptiter.close();                     // Close the iterator
ec.close();                            // Close the execution context
c1.close();                            // Close the connection
}

```

図 42. バッチ更新の実行例

## SQLJ アプリケーションでのデータの検索

SQLJ アプリケーションは結果セット・イテレーター を使用して、結果セットを検索します。カーソルと同様に、結果セット・イテレーターはスクロール不可にもスクロール可能にもできます。

他の言語の DB2 アプリケーションと同様に、SQLJ アプリケーションで表から単一行を検索する場合、その行のみを含む結果表を定義する、WHERE 節を指定した SELECT INTO ステートメントを作成できます。

```

#sql [myConnCtx] {SELECT DEPTNO INTO :hvdeptno
  FROM DEPARTMENT WHERE DEPTNAME="OPERATIONS"};

```

しかし、使用するほとんどの SELECT ステートメントでは、多数の行を含む結果表が作成されます。他の言語の DB2 アプリケーションでは、カーソルを使用して個々の行を結果表から選択します。そのカーソルはスクロール不可にすることができます。つまり、カーソルを使用して行をフェッチする場合、カーソルを結果表の最初から最後まで連続的に移動させます。あるいは、カーソルをスクロール可能にすることもできます。この場合、カーソルを使用して行をフェッチするときは、カーソルを順方向、逆方向、または結果表の任意の行に移動させることができます。



このトピックでは、スクロール不可イテレーターの使用方法を解説します。スクロール可能イテレーターの使用については、『SQLJ アプリケーションでのスクロール可能イテレーターの使用』を参照してください。

結果セット・イテレーターとは、結果表から行を取得するために使用する Java オブジェクトです。カーソルとは異なり、結果セット・イテレーターはパラメーターとしてメソッドに渡すことができます。

結果セット・イテレーターを使用する基本ステップは次のとおりです。

1. イテレーターを宣言します。それがイテレーター・クラスになります。
2. イテレーター・クラスのインスタンスを定義します。
3. SELECT の結果表をイテレーターのインスタンスに割り当てます。
4. 行を取得します。
5. イテレーターを閉じます。

イテレーターには、位置指定イテレーター と名前指定イテレーター という 2 つのタイプがあります。位置指定イテレーターはインターフェース `sqlj.runtime.PositionedIterator` を拡張します。位置指定イテレーターは結果表の列を、結果表内での位置で識別します。名前指定イテレーターはインターフェース `sqlj.runtime.NamedIterator` を拡張します。名前指定イテレーターは結果表の列を、結果表の列名で識別します。

## SQLJ アプリケーションでの名前指定イテレーターの使用

名前指定イテレーターを使用して、結果表の各列を名前で参照できます。

名前指定イテレーターを使用するための手順は、以下のとおりです。

1. イテレーターを宣言します。

結果セット・イテレーターを宣言するには、イテレーター宣言節を使用します。そうすると、イテレーターと同じ名前のイテレーター・クラスが作成されます。名前指定イテレーターの場合は、イテレーター宣言節で以下の情報を指定します。

- イテレーターの名前
- 列名と Java データ・タイプのリスト
- Java クラス宣言のための情報 (イテレーターが `public` か `static` か、など)
- 一連の属性 (イテレーターが保持可能かどうか、各列が更新可能かどうか、など)

照会の名前指定イテレーターを宣言するときには、イテレーターの各列の名前を指定します。指定する名前は、照会の結果表の各列の名前と一致している必要があります。イテレーターの列名と結果表の列名の大/小文字だけが違っている場合は、一致する名前と見なされます。イテレーター宣言節から生成される名前指定イテレーター・クラスには、`accessor` メソッドが含まれています。イテレーターの各列ごとに 1 つの `accessor` メソッドが存在します。各 `accessor` メソッドの名前は、対応するイテレーター列名と同じです。その `accessor` メソッドを使用して、結果表の各列のデータを取得します。



イテレーターには、対応する DB2 列のデータ・タイプに近い Java データ・タイプを指定する必要があります。Java データ・タイプと DB2 データ・タイプの最適な対応関係のリストについては、『Java、JDBC、および SQL のデータ・タイプ』を参照してください。

イテレーターを宣言するには、いくつかの方法があります。ただし、各イテレーターは Java クラスが基礎になっているので、イテレーターを宣言するときには、基礎になっているクラスが Java の規則に準拠していることを確認する必要があります。例えば、*with* 節が含まれているイテレーターは、`public` として宣言しなければなりません。したがって、`public` にしなければならないイテレーターは、`public` クラスを使用できる場所でのみ宣言できる、ということになります。イテレーターを宣言するためのいくつかの代替方法を以下のリストにまとめます。

- ソース・ファイルの中で単独で `public` として宣言する方法

この方法の場合は、他のコード・モジュールの中でもイテレーター宣言を使用できるので、すべての SQLJ アプリケーションで有効なイテレーターを用意できます。さらに、同じソース・ファイルの中に他の最上位クラスや `public` クラスがあるかどうかを気にする必要はありません。

- ソース・ファイルの中で他の最上位クラス定義と一緒に最上位クラスとして宣言する方法

Java では、1 つのコード・モジュールの中に `public` の最上位クラスを 1 つしか宣言できません。したがって、イテレーターを `public` として宣言しなければならない場合 (イテレーターに *with* 節を組み込む場合など) は、コード・モジュール内の他のクラスを `public` として宣言することはできません。

- 別のクラスの中にネストした `static` クラスとして宣言する方法

この代替方法の場合は、イテレーター宣言と他のクラス宣言を同じソース・ファイルの中で組み合わせ、イテレーターと他のクラスを `public` として宣言し、イテレーター・クラスを他のコード・モジュールやパッケージから参照できるようにすることが可能になります。ただし、ネスト元のクラスの外からイテレーターを参照する場合は、ネスト元のクラスの名前を付けた完全修飾イテレーター名を指定する必要があります。

- 別のクラスの `inner` クラスとして宣言する方法

この方法でイテレーターを宣言する場合は、ネスト元のクラスのインスタンスの中でしかイテレーターのインスタンスを生成できません。ただし、ファイルの中でイテレーターと他のクラスを `public` として宣言することは可能です。

イテレーターを `inner` クラスとして宣言した場合は、JDBC の `ResultSet` をイテレーターにキャストできません。この制限は、`static` のネスト・クラスとして宣言したイテレーターには当てはまりません。`ResultSet` をイテレーターにキャストする方法の詳細については、『同じアプリケーションでの SQLJ と JDBC の使用』を参照してください。

2. イテレーター・クラスのインスタンスを作成します。

結果表から行を取得するための名前指定イテレーター・クラスのオブジェクトを宣言します。

3. SELECT の結果表をイテレーターのインスタンスに割り当てます。

SELECT の結果表をイテレーターに割り当てるには、SQLJ の 代入節 を使用します。名前指定イテレーターの代入節の形式は、以下のとおりです。

```
#sql context-clause iterator-object={select-statement};
```

詳細については、『SQLJ 代入節』と『SQLJ コンテキスト節』を参照してください。

4. 行を取得します。

そのためには、ループの中で accessor メソッドを呼び出します。accessor メソッドの名前は、対応するイテレーターの列の名前と同じです。accessor メソッドにパラメーターはありません。accessor メソッドからは、結果表の現在行の対応する列の値が返されます。結果表の中でカーソルを前方へ移動するには、NamedIterator.next() メソッドを使用します。

すべての行を取得できたかどうかをテストするには、next メソッドを呼び出したときに返される値をチェックします。次の行がない場合は、next から false の値の boolean が返されます。

5. イテレーターを閉じます。

そのためには、NamedIterator.close メソッドを使用します。

名前指定イテレーターを宣言して使用するためのコードを以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```
#sql iterator ByName(String LastName, Date HireDate); 1
// Declare named iterator ByName
{
  ByName nameiter; 2
  // Declare object of ByName class
  #sql [ctxt]
  nameiter={SELECT LASTNAME, HIREDATE FROM EMPLOYEE}; 3
  // Assign the result table of the SELECT
  // to iterator object nameiter
  while (nameiter.next()) 4
  // Move the iterator through the result
  // table and test whether all rows retrieved
  {
    System.out.println( nameiter.LastName() + " was hired on "
      + nameiter.HireDate()); 5
    // Use accessor methods LastName and
    // HireDate to retrieve column values
  }
  nameiter.close(); 5
  // Close the iterator
}
```

図 43. 名前指定イテレーターの使用例

## SQLJ アプリケーションでの位置指定イテレーターの使用

位置指定イテレーターを使用すれば、結果表の各列を結果セット内の位置で参照できます。

位置指定イテレーターを使用するための手順は、以下のとおりです。

## 1. イテレーターを宣言します。

結果セット・イテレーターを宣言するには、イテレーター宣言節を使用します。そうすると、イテレーターと同じ名前、同じ属性のイテレーター・クラスが作成されます。位置指定イテレーターの場合は、イテレーター宣言節で以下の情報を指定します。

- イテレーターの名前
- Java データ・タイプのリスト
- Java クラス宣言のための情報 (イテレーターが `public` か `static` か、など)
- 一連の属性 (イテレーターが保持可能かどうか、各列が更新可能かどうか、など)

データ・タイプ宣言は、結果表の列に対応しており、その宣言のことを結果セット・イテレーターの列といいます。結果セット・イテレーターの列は、左から右の順で結果表の列に対応しています。例えば、イテレーター宣言節に 2 つのデータ・タイプ宣言があれば、最初のデータ・タイプ宣言が結果表の最初の列に対応し、2 番目のデータ・タイプ宣言が結果表の 2 番目の列に対応する、という具合になります。

イテレーターには、対応する DB2 列のデータ・タイプに近い Java データ・タイプを指定する必要があります。Java データ・タイプと DB2 データ・タイプの最適な対応関係のリストについては、『Java、JDBC、および SQL のデータ・タイプ』を参照してください。

イテレーターを宣言するには、いくつかの方法があります。ただし、各イテレーターは Java クラスが基礎になっているので、イテレーターを宣言するときには、基礎になっているクラスが Java の規則に準拠していることを確認する必要があります。例えば、`with` 節が含まれているイテレーターは、`public` として宣言しなければなりません。したがって、`public` にしなければならないイテレーターは、`public` クラスを使用できる場所でのみ宣言できる、ということになります。イテレーターを宣言するためのいくつかの代替方法を以下のリストにまとめます。

- ソース・ファイルの中で単独で `public` として宣言する方法

これは、イテレーターを宣言するための最も汎用的な方法です。この方法の場合は、他のコード・モジュールの中でもイテレーター宣言を使用できるので、すべての SQLJ アプリケーションで有効なイテレーターを用意できます。さらに、同じソース・ファイルの中に他の最上位クラスや `public` クラスがあるかどうかを気にする必要はありません。

- ソース・ファイルの中で他の最上位クラス定義と一緒に最上位クラスとして宣言する方法

Java では、1 つのコード・モジュールの中に `public` の最上位クラスを 1 つしか宣言できません。したがって、イテレーターを `public` として宣言しなければならない場合 (イテレーターに `with` 節を組み込む場合など) は、コード・モジュール内の他のクラスを `public` として宣言することはできません。

- 別のクラスの中にネストした `static` クラスとして宣言する方法

この代替方法の場合は、イテレーター宣言と他のクラス宣言を同じソース・ファイルの中で組み合わせ、イテレーターと他のクラスを `public` として宣言し、イテレーター・クラスを他のコード・モジュールやパッケージから参照できるようにすることが可能になります。ただし、ネスト元のクラスの外からイテレーターを参照する場合は、ネスト元のクラスの名前を付けた完全修飾イテレーター名を指定する必要があります。

- 別のクラスの `inner` クラスとして宣言する方法

この方法でイテレーターを宣言する場合は、ネスト元のクラスのインスタンスの中でしかイテレーターのインスタンスを生成できません。ただし、ファイルの中でイテレーターと他のクラスを `public` として宣言することは可能です。

イテレーターを `inner` クラスとして宣言した場合は、JDBC の `ResultSet` をイテレーターにキャストできません。この制限は、`static` のネスト・クラスとして宣言したイテレーターには当てはまりません。`ResultSet` をイテレーターにキャストする方法の詳細については、『同じアプリケーションでの SQLJ と JDBC の使用』を参照してください。

2. イテレーター・クラスのインスタンスを作成します。

結果表から行を取得するための位置指定イテレーター・クラスのオブジェクトを宣言します。

3. `SELECT` の結果表をイテレーターのインスタンスに割り当てます。

`SELECT` の結果表をイテレーターに割り当てるには、SQLJ の 代入節 を使用します。位置指定イテレーターの代入節の形式は、以下のとおりです。

```
#sql context-clause iterator-object={select-statement};
```

4. 行を取得します。

そのためには、ループの実行可能節の中で `FETCH` ステートメントを実行します。この `FETCH` ステートメントは、他の言語の `FETCH` ステートメントと同じ形式になります。

すべての行を取得できたかどうかをテストするには、各 `FETCH` の後で `PositionedIterator.endFetch` メソッドを呼び出します。取得する行がなくて `FETCH` が失敗した場合は、`endFetch` から `true` の値の `boolean` が返されます。

5. イテレーターを閉じます。

そのためには、`PositionedIterator.close` メソッドを使用します。

位置指定イテレーターを宣言して使用するためのコードを以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応していません。

```

#sql iterator ByPos(String,Date); // Declare positioned iterator ByPos 1
{
  ByPos positer;                // Declare object of ByPos class 2
  String name = null;           // Declare host variables
  Date hrdate;
  #sql [ctxt] positer =
    {SELECT LASTNAME, HIREDATE FROM EMPLOYEE};                3
    // Assign the result table of the SELECT
    // to iterator object positer
  #sql {FETCH :positer INTO :name, :hrdate };                4
    // Retrieve the first row
  while (!positer.endFetch()) // Check whether the FETCH returned a row
  { System.out.println(name + " was hired in " +
    hrdate);
    #sql {FETCH :positer INTO :name, :hrdate };
    // Fetch the next row
  }
  positer.close(); // Close the iterator 5
}

```

図 44. 位置指定イテレーターの使用例

## SQLJ アプリケーションにおける同じ SQL ステートメントの複数のオープン・イテレーター

IBM Data Server Driver for JDBC and SQLJ により、ご使用のアプリケーションは SQLJ アプリケーション内の単一 SQL ステートメントの複数のイテレーターを同時にオープンできます。この機能によって、1 つのイテレーターを使用して表である操作を実行しながら、別のイテレーターを使用して同じ表で別の操作を実行できます。

アプリケーションで同時にオープンしているイテレーターを使用する場合、それが不要になったら Java ヒープでの過剰なストレージ消費を防ぐためにそれらのイテレーターをクローズする必要があります。

以下の例では、単一の SQL ステートメントで同時にオープンしているイテレーターがない場合と、単一の SQL ステートメントで同時にオープンしているイテレーターがある場合に、表に対して同じ操作を実行する方法を示しています。これらの例では、次のイテレーター宣言を使用します。

```

import java.math.*;
#sql public iterator MultiIter(String EmpNo, BigDecimal Salary);

```

単一の SQL ステートメントで複数のイテレーターを同時にオープンできない場合、特定の従業員番号の従業員および給与の値を選択するには、144 ページの図 45 に示されるように、従業員番号ごとに異なる SQL ステートメントを定義する必要があります。

```

MultiIter iter1 = null;           // Iterator instance for retrieving
                                  // data for first employee
String EmpNo1 = "000100";        // Employee number for first employee
#sql [ctx] iter2 =
    {SELECT EMPNO, SALARY FROM EMPLOYEE WHERE EMPNO = :EmpNo1};
                                  // Assign result table to first iterator
MultiIter iter2 = null;          // Iterator instance for retrieving
                                  // data for second employee
String EmpNo2 = "000200";        // Employee number for second employee
#sql [ctx] iter2 =
    {SELECT EMPNO, SALARY FROM EMPLOYEE WHERE EMPNO = :EmpNo2};
                                  // Assign result table to second iterator

// Process with iter1
// Process with iter2
iter1.close();                    // Close the iterators
iter2.close();

```

図 45. 異なる SQL ステートメントでイテレーターを使用して表に対する操作を同時に実行する例

図 46 では、単一の SQL ステートメントで複数のイテレーターを同時にオープンできる場合に、同じ操作を実行する方法を示しています。

```

...
MultiIter iter1 = openIter("000100"); // Invoke openIter to assign the result table
                                          // (for employee 100) to the first iterator
MultiIter iter2 = openIter("000200"); // Invoke openIter to assign the result
                                          // table to the second iterator
                                          // iter1 stays open when iter2 is opened

// Process with iter1
// Process with iter2
...
iter1.close();                          // Close the iterators
iter2.close();
...
public MultiIter openIter(String EmpNo)
    // Method to assign a result table
    // to an iterator instance
{
    MultiIter iter;
    #sql [ctx] iter =
        {SELECT EMPNO, SALARY FROM EMPLOYEE WHERE EMPNO = :EmpNo};
    return iter;                          // Method returns an iterator instance
}

```

図 46. 同じ SQL ステートメントでイテレーターを使用して表に対する操作を同時に実行する例

## SQLJ アプリケーションにおけるイテレーターの複数のオープン・インスタンス

イテレーターの複数インスタンスを単一の SQLJ アプリケーションで同時にオープンできます。この機能の応用の 1 つは、ホスト式を使用するイテレーターの複数インスタンスをオープンすることです。それぞれのインスタンスは様々なホスト式の値のセットを使用できます。



以下の例は、イテレーターの 2 つのインスタンスが同時にオープンしているアプリケーションを示しています。

```
...
ResultSet myFunc(String empid) // Method to open an iterator and get a resultSet
{
    MyIter iter;
    #sql iter = {SELECT * FROM EMPLOYEE WHERE EMPNO = :empid};
    return iter.getResultSet();
}

// An application can call this method to get a resultSet for each
// employee ID. The application can process each resultSet separately.
...
ResultSet rs1 = myFunc("000100"); // Get employee record for employee ID 000100
...
ResultSet rs2 = myFunc("000200"); // Get employee record for employee ID 000200
```

図 47. 単一アプリケーションでイテレーターの複数インスタンスをオープンする例

その他のイテレーターの場合と同様に、このイテレーターを最後に使用した後は、過剰なストレージ消費を防ぐために必ずクローズする必要があります。

## SQLJ アプリケーションでのスクロール可能イテレーターの使用

結果表の中では、1 行ずつ前方へ移動する機能だけでなく、後方へ移動したり、特定の行に直接移動したりする機能があれば便利です。IBM Data Server Driver for JDBC and SQLJ には、そのための機能が用意されています。

前方への移動、後方への移動、特定の行への移動が可能なイテレーターのことをスクロール可能イテレーター といいます。SQLJ のスクロール可能イテレーターは、SCROLL として宣言したデータベース・カーソルの結果表と等価です。

両方向スクロール・カーソルの場合と同じく、スクロール可能イテレーターにも、インセンシティブ とセンシティブ があります。センシティブのスクロール可能イテレーターには、静的 と動的 があります。インセンシティブとは、イテレーターを開いた後の基礎表の変更がイテレーターから不可視になる、という意味です。インセンシティブ・イテレーターは、読み取り専用です。センシティブとは、イテレーターまたは他のプロセスが基礎表に加えた変更がイテレーターから可視になる、という意味です。アセンシティブとは、カーソルが読み取り専用の場合に、インセンシティブ・カーソルとして動作する、という意味です。読み取り専用カーソルでなければ、センシティブ・カーソルとして動作します。

スクロール可能イテレーターが静的であれば、結果表のサイズと各行の順序は、イテレーターを開いた後も変わりません。したがって、結果表に行を挿入することはできませんし、結果表の行を削除すると、削除ホールが発生します。結果表の行を更新して、行が結果表の資格を満たさなくなると、更新ホールが発生します。ホールからフェッチすると、SQLException になります。

スクロール可能イテレーターが動的であれば、結果表のサイズと各行の順序は、イテレーターを開いた後に変わる可能性があります。同じアプリケーション・プロセスで実行される INSERT ステートメントや DELETE ステートメントで挿入されたり削除されたりした行は、すぐに可視になります。他のアプリケーション・プロセ



スで実行される INSERT ステートメントや DELETE ステートメントで挿入されたり削除されたりした行は、変更のコミット後に可視になります。

**重要:** DB2 Database for Linux, UNIX, and Windows サーバーは、動的な両方向スクロール・カーソルをサポートしていません。SQLJ アプリケーションで動的なスクロール可能イテレーターを使用できるのは、そのアプリケーションがバージョン 9 以降の DB2 for z/OS サーバーのデータにアクセスする場合に限られます。

スクロール可能イテレーターを作成して使用するには、以下の手順を実行する必要があります。

1. 以下の節を組み込んだイテレーター宣言節を指定します。

- `implements sqlj.runtime.Scrollable`

イテレーターをスクロール可能として指定します。

- `with (sensitivity=INSENSITIVE|SENSITIVE|ASENSITIVE)` または `with (sensitivity=SENSITIVE, dynamic=true|false)`

`sensitivity=INSENSITIVE|SENSITIVE|ASENSITIVE` では、基礎表の更新操作や削除操作をイテレーターから可視にするかどうかを指定します。デフォルトの `sensitivity` は `INSENSITIVE` です。

`dynamic=true|false` では、イテレーターを開いた後に、結果表のサイズと各行の順序を変更可能にするかどうかを指定します。`dynamic` のデフォルト値は `false` です。

イテレーターには、位置指定イテレーターと名前指定イテレーターがあります。例えば、以下のイテレーター宣言節では、位置指定、センシティブ、動的、スクロール可能のイテレーターを指定しています。

```
#sql public iterator ByPos
  implements sqlj.runtime.Scrollable
  with (sensitivity=SENSITIVE, dynamic=true) (String);
```

以下のイテレーター宣言節では、名前指定、インセンシティブ、スクロール可能のイテレーターを指定しています。

```
#sql public iterator ByName
  implements sqlj.runtime.Scrollable
  with (sensitivity=INSENSITIVE) (String EmpNo);
```

**制限:** DB2 Database for Linux, UNIX, and Windows サーバーでは、スクロール可能イテレーターを使用して、表から以下のデータ・タイプの列を選択することができません。

- `LONG VARCHAR`
- `LONG VARGRAPHIC`
- `BLOB`
- `CLOB`
- このリスト内の前述のデータ・タイプのいずれかに基づいた特殊タイプ
- 構造化タイプ

2. イテレーター・クラスのインスタンスであるイテレーター・オブジェクトを作成します。

3. 初期のフェッチ方向に関するヒントを SQLJ ランタイム環境に与えるには、`setFetchDirection(int direction)` メソッドを使用します。`direction` は、`FETCH_FORWARD` と `FETCH_REVERSE` のいずれかになります。`setFetchDirection` を呼び出さない場合のフェッチ方向は、`FETCH_FORWARD` です。
4. アクセスする各行ごとに以下のようにします。

名前指定イテレーターの場合は、以下の手順を実行します。

- a. 以下の表にまとめられているメソッドのいずれかを使用して、カーソルの位置を指定します。

表 19. 両方向スクロール・カーソルの位置を指定するための `sqlj.runtime.Scrollable` メソッド

メソッド	カーソルの位置
<code>first</code>	結果表の最初の行
<code>last</code>	結果表の最後の行
<code>previous<sup>1</sup></code>	結果表の前の行
<code>next</code>	結果表の次の行
<code>absolute(int n)<sup>2</sup></code>	$n > 0$ の場合は、結果表の行 $n$ 。 $n < 0$ の場合は、結果表の行 $m+n+1$ ( $m$ は結果表の行数)。
<code>relative(int n)<sup>3</sup></code>	$n > 0$ の場合、現在行の $n$ 行後ろにある行。 $n < 0$ の場合、現在行の $n$ 行前にある行。 $n = 0$ の場合は現在行。
<code>afterLast</code>	結果表の最後の行の後
<code>beforeFirst</code>	結果表の最初の行の前

注:

1. カーソルが結果表の最後の行の後にあれば、このメソッドによってカーソルはその最後の行に配置されます。
2.  $n$  の絶対値が結果表の行数より大きい場合、 $n$  が正であれば、このメソッドによってカーソルは最後の行の後に配置され、 $n$  が負であれば最初の行の前に配置されます。
3.  $m$  が結果表に含まれる行数、 $x$  が結果表内の現在行の番号とします。  $n > 0$  であり、 $x+n > m$  である場合、イテレーターは最後の行の後ろに置かれます。  $n < 0$  であり、 $x+n < 1$  である場合、イテレーターは最初の行の前に置かれます。

- b. 現行カーソル位置を知る必要がある場合には、`getRow`、`isFirst`、`isLast`、`isBeforeFirst`、または `isAfterLast` メソッドを使用してこの情報を取得します。

現在のフェッチ方向を確認する必要がある場合は、`getFetchDirection` メソッドを呼び出します。

- c. `accessor` メソッドを使用して、結果表の現在行を取得します。
- d. イテレーターまたは他の手段による更新操作または削除操作が結果表の中で可視になっている場合は、`getWarnings` メソッドを呼び出して、現在行がホールになっているかどうかをチェックします。

位置指定イテレーターの場合は、以下の手順を実行します。

- a. フェッチ・オリエンテーション節を指定した `FETCH` ステートメントを使用して、イテレーターの位置を指定し、結果表の現在行を取得します。カーソルの位置を指定するために使用できる節を 148 ページの表 20 にまとめます。

表 20. 両方向スクロール・カーソルの位置を指定するための *FETCH* 節

メソッド	カーソルの位置
FIRST	結果表の最初の行
LAST	結果表の最後の行
PRIOR <sup>1</sup>	結果表の前の行
NEXT	結果表の次の行
ABSOLUTE( <i>n</i> ) <sup>2</sup>	<i>n</i> >0 の場合は、結果表の行 <i>n</i> 。 <i>n</i> <0 の場合は、結果表の行 <i>m+n+1</i> ( <i>m</i> は結果表の行数)。
RELATIVE( <i>n</i> ) <sup>3</sup>	<i>n</i> >0 の場合、現在行の <i>n</i> 行後ろにある行。 <i>n</i> <0 の場合、現在行の <i>n</i> 行前にある行。 <i>n</i> =0 の場合は現在行。
AFTER <sup>4</sup>	結果表の最後の行の後
BEFORE <sup>4</sup>	結果表の最初の行の前

注:

- カーソルが結果表の最後の行の後にあれば、このメソッドによってカーソルはその最後の行に配置されます。
- n* の絶対値が結果表の行数より大きい場合、*n* が正であれば、このメソッドによってカーソルは最後の行の後に配置され、*n* が負であれば最初の行の前に配置されます。
- m* が結果表に含まれる行数、*x* が結果表内の現在行の番号とします。 *n*>0 であり、*x+n>m* である場合、イテレーターは最後の行の後ろに置かれます。 *n*<0 であり、*x+n<1* である場合、イテレーターは最初の行の前に置かれます。
- ホスト式には値が割り当てられません。

b. イテレーターまたは他の手段による更新操作または削除操作が結果表の中で可視になっている場合は、`getWarnings` メソッドを呼び出して、現在行がホールになっているかどうかをチェックします。

- `close` メソッドを呼び出してイテレーターを閉じます。

名前指定イテレーターを使用して、従業員表のすべての行から従業員番号とラストネーム (姓) を逆順で取得するコードを以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```
#sql iterator ScrollIter implements sqlj.runtime.Scrollable      1
    (String EmpNo, String LastName);
{
    ScrollIter scrliter;                                          2
    #sql [ctxt]
    scrliter={SELECT EMPNO, LASTNAME FROM EMPLOYEE};
    scrliter.afterLast();
    while (scrliter.previous()                                    4a
    {
        System.out.println(scrliter.EmpNo() + " "              4c
            + scrliter.LastName());
    }
    scrliter.close();                                           5
}
```

図 48. スクロール可能イテレーターの使用例

## SQLJ アプリケーションでのストアード・プロシージャの呼び出し

ストアード・プロシージャを呼び出すには、SQL の CALL ステートメントが含まれている実行可能節を使用します。

ホスト ID パラメーターを指定した CALL ステートメントを実行することも可能です。リテラル・パラメーターを指定した CALL ステートメントを実行できるのは、その CALL ステートメントを実行する DB2 サーバーが CALL ステートメントの動的実行をサポートしている場合に限られます。

ストアード・プロシージャを呼び出すための基本手順は、以下のとおりです。

1. 入力 (IN または INOUT) パラメーターに値を割り当てます。
2. ストアード・プロシージャを呼び出します。
3. 出力 (OUT または INOUT) パラメーターを処理します。
4. ストアード・プロシージャが複数の結果セットを返す場合は、それらの結果セットを取得します。

3 つの入力パラメーターと 3 つの出力パラメーターがあるストアード・プロシージャを呼び出すコードを以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```
String FirstName="TOM";           // Input parameters 1
String LastName="NARISINST";
String Address="IBM";
int CustNo;                       // Output parameters
String Mark;
String MarkErrorText;
...
#sql [myConnCtx] {CALL ADD_CUSTOMER(:IN FirstName, 2
                                     :IN LastName,
                                     :IN Address,
                                     :OUT CustNo,
                                     :OUT Mark,
                                     :OUT MarkErrorText)};
                                     // Call the stored procedure
System.out.println("Output parameters from ADD_CUSTOMER call: ");
System.out.println("Customer number for " + LastName + ": " + CustNo); 3
System.out.println(Mark);
If (MarkErrorText != null)
    System.out.println(" Error messages:" + MarkErrorText);
```

図 49. SQLJ アプリケーションでストアード・プロシージャを呼び出す例

### SQLJ アプリケーションでのストアード・プロシージャの複数の結果セットの検索

ストアード・プロシージャには、1 つ以上の結果セットを呼び出し側プログラムに戻すものがあります。呼び出し側プログラムはそれらの結果セットの内容を検索する必要があります。

それらの結果セットから行を取得するには、以下のステップを実行します。

1. ストアド・プロシージャの結果セットを取得するために実行コンテキストを取得します。
2. 実行コンテキストをストアド・プロシージャの CALL ステートメントと関連付けます。

最後の結果セットを取得して処理するまで、この実行コンテキストを他の目的では使用しないでください。

3. 結果セットごとに、次のことを実行します。
  - a. `ExecutionContext` メソッド `getNextResultSet` を使用して結果セットを取得します。
  - b. 結果セットの内容がわからない場合、`ResultSetMetaData` メソッドを使用してこの情報を取得します。
  - c. `SQLJ` 結果セット・イテレーターまたは `JDBC ResultSet` を使用して、結果セットから行を取得します。

結果セットは、カーソルがストアド・プロシージャでオープンされるのと同じ順序で呼び出し側プログラムに戻されます。検索する結果セットがなくなると、`getNextResultSet` は `NULL` 値を戻します。

`getNextResultSet` には次の 2 つの形式があります。

```
getNextResultSet();  
getNextResultSet(int current);
```

最初の形式である `getNextResultSet` を呼び出すと、`SQLJ` は現在オープンしている結果セットをクローズし、次の結果セットに進みます。2 番目の形式である `getNextResultSet` を呼び出すと、`SQLJ` が次の結果セットに進む前に現在オープンしている結果セットで行う事柄を、`current` の値で指示します。

#### **java.sql.Statement.CLOSE\_CURRENT\_RESULT**

次の `ResultSet` オブジェクトが戻されるときに、現在の `ResultSet` オブジェクトをクローズすることを指定します。

#### **java.sql.Statement.KEEP\_CURRENT\_RESULT**

次の `ResultSet` オブジェクトが戻されるときに、現在の `ResultSet` オブジェクトをオープンしたままにすることを指定します。

#### **java.sql.Statement.CLOSE\_ALL\_RESULTS**

次の `ResultSet` オブジェクトが戻されるときに、オープンしているすべての `ResultSet` オブジェクトをクローズすることを指定します。

以下のコードは、複数の結果セットを戻すストアド・プロシージャを呼び出します。この例の場合、戻される結果セットの数またはそれらの結果セットの内容を呼び出し元が認識していないことを想定しています。また、`autoCommit` が `false` であることも想定しています。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```

ExecutionContext execCtx=myConnCtx.getExecutionContext();
#sql [myConnCtx, execCtx] {CALL MULTRSSP()};
// MULTRSSP returns multiple result sets
ResultSet rs;
while ((rs = execCtx.getNextResultSet()) != null)
{
    ResultSetMetaData rsmeta=rs.getMetaData();
    int numcols=rsmeta.getColumnCount();
    while (rs.next())
    {
        for (int i=1; i<=numcols; i++)
        {
            String colval=rs.getString(i);
            System.out.println("Column " + i + "value is " + colval);
        }
    }
}

```

1  
2

3a

3b

3c

図 50. ストアード・プロシージャの結果セットの検索

## IBM Data Server Driver for JDBC and SQLJ 使用時の SQLJ アプリケーションでの LOB

IBM Data Server Driver for JDBC and SQLJを使用すると、LOB データの Clob または Blob ホスト式への取り出しや、Clob または Blob ホスト式で CLOB、BLOB、DBCLOB 列の更新ができます。また、Clob または Blob データ・タイプのイテレーターを宣言して、CLOB、BLOB、または DBCLOB 列のデータを検索することができます。

**LOB データの検索または更新:** BLOB 列のデータを検索するには、データ・タイプ Blob または byte[] を含むイテレーターを宣言します。CLOB または DBCLOB 列のデータを検索するには、対応する列に Clob データ・タイプがあるイテレーターを宣言します。

BLOB 列のデータを更新するには、データ・タイプが Blob のホスト式を使用します。CLOB 列または DBCLOB 列のデータを更新するには、データ・タイプが Clob のホスト式を使用します。

**連続ストリーミングまたは LOB ロケーター:** SQLJ アプリケーションでは、連続ストリーミングまたは LOB ロケーターを、JDBC アプリケーションで使用すると同じ方法で使用できます。

### SQLJ アプリケーションでの LOB 列データの検索および更新のための Java データ・タイプ

deferPrepares プロパティが true に設定されており、IBM Data Server Driver for JDBC and SQLJ がホスト式を含むカスタマイズされていない SQLJ ステートメントを処理する場合、ドライバーはデータ・タイプを判別するための追加の処理を行う必要があるかもしれません。この追加処理は、パフォーマンスに影響を与えるおそれがあります。

JDBC ドライバーで、LOB 列とともに使用されるパラメーターのデータ・タイプがすぐに判別できない場合は、LOB データ・タイプと互換性のあるパラメーターのデータ・タイプを選択する必要があります。

## BLOB 列の入力パラメーター

BLOB 列の入力パラメーターの場合、以下の手法のいずれかを使用できます。

- 次のように、BLOB 列と完全に一致する `java.sql.Blob` 入力変数を使用します。

```
java.sql.Blob blobData;  
#sql {CALL STORPROC(:IN blobData)};
```

`java.sql.Blob` 入力変数を使用するには、その前に `java.sql.Blob` オブジェクトを作成し、それからそのオブジェクトにデータを入力する必要があります。

- タイプ `sqlj.runtime.BinaryStream` の入力パラメーターを使用します。  
`sqlj.runtime.BinaryStream` オブジェクトは BLOB データ・タイプと互換性があります。この呼び出しの場合、入力データの正確な長さを指定する必要があります。

```
java.io.ByteArrayInputStream byteStream =  
    new java.io.ByteArrayInputStream(byteData);  
int numBytes = byteData.length;  
sqlj.runtime.BinaryStream binStream =  
    new sqlj.runtime.BinaryStream(byteStream, numBytes);  
#sql {CALL STORPROC(:IN binStream)};
```

この手法は入出力パラメーターには使用できません。

## BLOB 列の出力パラメーター

BLOB 列の出力パラメーターまたは入出力パラメーターの場合、以下の手法を使用できます。

- 出力パラメーターまたは入出力変数を `java.sql.Blob` データ・タイプで宣言します。

```
java.sql.Blob blobData = null;  
#sql CALL STORPROC (:OUT blobData)};
```

```
java.sql.Blob blobData = null;  
#sql CALL STORPROC (:INOUT blobData)};
```

## CLOB 列の入力パラメーター

CLOB 列の入力パラメーターの場合、以下の手法のいずれかを使用できます。

- 次のように、CLOB 列と完全に一致する `java.sql.Clob` 入力変数を使用します。

```
#sql CALL STORPROC(:IN clobData)};
```

`java.sql.Clob` 入力変数を使用するには、その前に `java.sql.Clob` オブジェクトを作成し、それからそのオブジェクトにデータを入力する必要があります。

- 以下のいずれかのタイプのストリーム入力パラメーターを使用します。

```
- sqlj.runtime.CharacterStream 入力パラメーター:  
java.lang.String charData;  
java.io.StringReader reader = new java.io.StringReader(charData);  
sqlj.runtime.CharacterStream charStream =  
    new sqlj.runtime.CharacterStream (reader, charData.length);  
#sql {CALL STORPROC(:IN charStream)};
```



- Unicode UTF-16 データ用の `sqlj.runtime.UnicodeStream` パラメーター:

```
byte[] charDataBytes = charData.getBytes("UnicodeBigUnmarked");
java.io.ByteArrayInputStream byteStream =
    new java.io.ByteArrayInputStream(charDataBytes);
sqlj.runtime.UnicodeStream uniStream =
    new sqlj.runtime.UnicodeStream(byteStream, charDataBytes.length);
#sql {CALL STORPROC(:IN uniStream)};
```

- ASCII データ用の `sqlj.runtime.AsciiStream` パラメーター:

```
byte[] charDataBytes = charData.getBytes("US-ASCII");
java.io.ByteArrayInputStream byteStream =
    new java.io.ByteArrayInputStream(charDataBytes);
sqlj.runtime.AsciiStream asciiStream =
    new sqlj.runtime.AsciiStream(byteStream, charDataBytes.length);
#sql {CALL STORPROC(:IN asciiStream)};
```

これらの呼び出しでは、入力データの正確な長さを指定する必要があります。この手法は入出力パラメーターには使用できません。

- `java.lang.String` 入力パラメーターを使用します。

```
java.lang.String charData;
#sql {CALL STORPROC(:IN charData)};
```

### CLOB 列の出力パラメーター

CLOB 列の出力パラメーターまたは入出力パラメーターの場合、以下の手法のいずれかを使用できます。

- `java.sql.Clob` 出力変数を使用します。これは、CLOB 列に完全一致します。

```
java.sql.Clob clobData = null;
#sql CALL STORPROC(:OUT clobData)};
```

- `java.lang.String` 出力変数を使用します。

```
java.lang.String charData = null;
#sql CALL STORPROC(:OUT charData)};
```

この技法は、取り出されるデータの長さが 32 KB 以下であることが分かっている場合にのみ使用してください。32 KB を超える場合、データは切り捨てられます。

### DBCLOB 列の出力パラメーター

ストアード・プロシージャ用の DBCLOB 出力パラメーターまたは入出力パラメーターはサポートされていません。

## 同じアプリケーションでの SQLJ および JDBC

SQLJ 節と JDBC 呼び出しを単一プログラムで結合できます。

これを効率的に行うには、以下の事柄を行えなければなりません。

- JDBC Connection を使用して SQLJ ConnectionContext を作成するか、JDBC Connection を SQLJ ConnectionContext から取得します。
- SQLJ イテレーターを使用して JDBC ResultSet からデータを取り出すか、JDBC ResultSet を SQLJ イテレーターから生成します。

**JDBC Connection からの SQLJ ConnectionContext の作成:** これを行うには、以下のようになります。

1. SQLJ 接続宣言節を実行して、`ConnectionContext` クラスを作成します。
2. ドライバーをロードするか、または `DataSource` インスタンスを取得します。
3. `JDBC DriverManager.getConnection` または `DataSource.getConnection` メソッドを呼び出して、`JDBC Connection` を取得します。
4. 引数として `Connection` を使用して `ConnectionContext` コンストラクターを呼び出して、`ConnectionContext` オブジェクトを作成します。

**SQLJ `ConnectionContext` からの `JDBC Connection` の取得:** これを行うには、以下のようになります。

1. SQLJ 接続宣言節を実行して、`ConnectionContext` クラスを作成します。
2. ドライバーをロードするか、または `DataSource` インスタンスを取得します。
3. 引数としてドライバーの URL およびその他の必要なパラメーターを使用して `ConnectionContext` コンストラクターを呼び出して、`ConnectionContext` オブジェクトを作成します。
4. `JDBC ConnectionContext.getConnection` メソッドを呼び出して、`JDBC Connection` オブジェクトを作成します。

SQLJ 接続について詳しくは、『SQLJ を使用したデータ・ソースへの接続』を参照してください。

**SQLJ イテレーターを使用した `JDBC` 結果セットの検索:** *iterator conversion statement* を使用して、`JDBC` 結果セットを SQLJ イテレーターとして操作します。イテレーター変換ステートメントの一般的な形式は、次のとおりです。

```
#sql iterator={CAST :result-set};
```

結果セットをイテレーターに正常にキャストするには、その前にイテレーターが以下の規則に準拠している必要があります。

- イテレーターは `public` と宣言される必要があります。
- イテレーターが位置指定イテレーターの場合、結果セット内の列の数がイテレーター内の列の数に一致しなければなりません。さらに、結果セット内の列ごとのデータ・タイプがイテレーター内の対応する列のデータ・タイプに一致する必要があります。
- イテレーターが名前指定イテレーターの場合、各 `accessor` メソッドの名前が結果セット内の列の名前に一致しなければなりません。さらに、`accessor` メソッドが戻すオブジェクトのデータ・タイプが結果セット内の対応する列のデータ・タイプに一致する必要があります。

155 ページの図 51 のコードは、`JDBC` 呼び出しを使用して照会を作成および実行し、*iterator conversion* ステートメントを実行して `JDBC` 結果セットを SQLJ イテレーターに変換し、イテレーターを使用して結果表から行を取得します。

```

#sql public iterator ByName(String LastName, Date HireDate); 1
public void HireDates(ConnectionContext connCtx, String whereClause)
{
    ByName nameiter;           // Declare object of ByName class
    Connection conn=connCtx.getConnection();
                                // Create JDBC connection
    Statement stmt = conn.createStatement(); 2
    String query = "SELECT LASTNAME, HIREDATE FROM EMPLOYEE";
    query+=whereClause; // Build the query
    ResultSet rs = stmt.executeQuery(query); 3
    #sql [connCtx] nameiter = {CAST :rs}; 4
    while (nameiter.next())
    {
        System.out.println( nameiter.LastName() + " was hired on "
            + nameiter.HireDate());
    }
    nameiter.close(); 5
    stmt.close();
}

```

図 51. SQLJ イテレーターへの JDBC 結果セットの変換

図 51 の注:

注	説明
1	この SQLJ 節は、名前指定イテレーター・クラス ByName を作成します。このクラスには、結果表列 LASTNAME および HIREDATE からのデータを戻す accessor メソッド LastName() および HireDate() があります。
2	このステートメントおよび次の 2 つのステートメントは、JDBC を使用して動的実行のために照会を作成し、準備します。
3	この JDBC ステートメントは SELECT ステートメントを実行し、その結果表を結果セット rs に割り当てます。
4	このイテレーター変換節は JDBC ResultSet rs を SQLJ イテレーター nameiter に変換し、次のステートメントは nameiter を使用して結果表から値を検索します。
5	nameiter.close() メソッドは、SQLJ イテレーターおよび JDBC ResultSet rs をクローズします。

**SQLJ イテレーターからの JDBC ResultSet の生成:** getResultSet メソッドを使用して、SQLJ イテレーターから JDBC ResultSet を生成します。すべての SQLJ イテレーターには getResultSet メソッドがあります。イテレーターを結果セットに変換した後、結果セットのみを使用して行をフェッチする必要があります。

156 ページの図 52 のコードは、照会用の位置指定イテレーターを生成し、そのイテレーターを結果セットに変換し、さらに JDBC メソッドを使用して表から行をフェッチします。

```

#sql iterator EmpIter(String, java.sql.Date);
{
...
    EmpIter iter=null;
    #sql [connCtx] iter=
        {SELECT LASTNAME, HIREDATE FROM EMPLOYEE};
    ResultSet rs=iter.getResultSet();
    while (rs.next())
    { System.out.println(rs.getString(1) + " was hired in " +
        rs.getDate(2));
    }
    rs.close();
}

```

図 52. JDBC ResultSet への SQLJ イテレーターの変換

図 52 の注:

注	説明
1	この SQLJ 節は SELECT ステートメントを実行し、その SELECT ステートメントの結果表を含むイテレーター・オブジェクトを構成し、そのイテレーター・オブジェクトを変数 iter に割り当てます。
2	getResultSet() メソッドはイテレーター iter を ResultSet rs に変換します。
3	JDBC getString() および getDate() メソッドは ResultSet から値を検索します。next() メソッドは ResultSet 内の次の行にカーソルを移動します。
4	rs.close() メソッドは、ResultSet と同時に SQLJ イテレーターをクローズします。

**SQLJ アプリケーションでの JDBC ResultSet の使用の規則および制限:** JDBC 結果セットを含む SQLJ アプリケーションを作成する場合、以下の規則および制限を守ってください。

- ResultSet とイテレーターが異なる保持可能性属性を持つ場合、ResultSet を SQLJ イテレーターにキャストできません。

COMMIT 操作後も JDBC ResultSet または SQLJ イテレーターをオープンしたままにすることができます。JDBC ResultSet の場合、この特性は IBM Data Server Driver for JDBC and SQLJ プロパティ resultSetHoldability によって制御されます。SQLJ イテレーターの場合、この特性はイテレーター宣言の with holdability パラメーターによって制御されます。保持可能性を持つ ResultSet を保持可能性を持たない SQLJ イテレーターにキャストしたり、保持可能性を持たない ResultSet を保持可能性を持つ SQLJ イテレーターにキャストしたりすることはサポートされていません。

- プログラムの終了時に、生成された ResultSet オブジェクトまたは基礎となるイテレーターをクローズします。

ResultSet オブジェクトの生成元のイテレーター・オブジェクトをクローズすると、ResultSet オブジェクトもクローズします。生成された ResultSet オブジェクトをクローズすると、イテレーター・オブジェクトもクローズします。一般に、最後に使用されるオブジェクトをクローズするのが最善です。

- スクロール可能イテレーターおよびスクロール可能で更新可能な ResultSet をサポートする IBM Data Server Driver for JDBC and SQLJ の場合、以下の制限が適用されます。

- スクロール可能イテレーターには、その基礎となる JDBC ResultSet と同じ制限があります。
- 更新可能な SQLJ イテレーターに、更新できない JDBC ResultSet をキャストすることはできません。

## SQLJ での SQL ステートメントの実行の制御

SQL ステートメントの実行を制御したりモニターしたりするには、SQLJ の ExecutionContext クラスから特定のメソッドを選択して使用します。

ExecutionContext メソッドを使用するには、以下の手順を実行します。

1. 接続コンテキストからデフォルトの実行コンテキストを取得します。

実行コンテキストを取得するには、2 つの方法があります。

- 接続コンテキストからデフォルトの実行コンテキストを取得します。以下に例を示します。

```
ExecutionContext execCtx = connCtx.getExecutionContext();
```

- ExecutionContext のコンストラクターを呼び出して、新しい実行コンテキストを作成します。以下に例を示します。

```
ExecutionContext execCtx=new ExecutionContext();
```

2. 実行コンテキストを SQL ステートメントに関連付けます。

そのためには、SQL ステートメントを組み込んだ実行節の接続コンテキストの後に実行コンテキストを指定します。

3. ExecutionContext メソッドを呼び出します。

ExecutionContext メソッドの中には、関連する SQL ステートメントの実行前に適用できるものもあれば、関連する SQL ステートメントの実行後にしか適用できないものもあります。

例えば、getUpdateCount メソッドは、DELETE ステートメントの実行後に、その DELETE ステートメントによって削除された行の数をカウントするために使用できます。

実行コンテキストを取得してから、その実行コンテキストで getUpdateCount メソッドを使用して、DELETE ステートメントによって削除された行の数を確認するためのコードを以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```
ExecutionContext execCtx=new ExecutionContext();
#sql [connCtx, execCtx] {DELETE FROM EMPLOYEE WHERE SALARY > 10000};
System.out.println("Deleted " + execCtx.getUpdateCount() + " rows");
```

1  
2  
3

## IBM Data Server Driver for JDBC and SQLJ 使用時の SQLJ での ROWID

DB2 for z/OS および DB2 for i5/OS は、表内の列に対して ROWID データ・タイプをサポートします。ROWID とは、表内の行を一意的に識別する値です。

SQLJ プログラムで ROWID を使用する場合、それらのプログラムをカスタマイズする必要があります。

JDBC 4.0 には、イテレーターおよび CALL ステートメント・パラメーターで使用できるインターフェース `java.sql.RowId` が含まれています。JDBC 4.0 がない場合、IBM Data Server Driver for JDBC and SQLJ 専用のクラス `com.ibm.db2.jcc.DB2RowID` を使用できます。イテレーターの場合、`byte[]` オブジェクト・タイプを使用して ROWID 値を検索することもできます。

以下のコーディング例は、ROWID 列から値を選択するために使用するイテレーターを示したものです。

```
#sql iterator PosIter(int,String,java.sql.RowId);
                                // Declare positioned iterator
                                // for retrieving ITEM_ID (INTEGER),
                                // ITEM_FORMAT (VARCHAR), and ITEM_ROWID (ROWID)
                                // values from table ROWIDTAB
{
    PosIter positrowid;        // Declare object of PosIter class
    java.sql.RowId rowid = null;
    int id = 0;
    String i_fmt = null;

                                // Declare host expressions
    #sql [ctxt] positrowid =
        {SELECT ITEM_ID, ITEM_FORMAT, ITEM_ROWID FROM ROWIDTAB
         WHERE ITEM_ID=3};
                                // Assign the result table of the SELECT
                                // to iterator object positrowid
    #sql {FETCH :positrowid INTO :id, :i_fmt, :rowid};
                                // Retrieve the first row
    while (!positrowid.endFetch())
        // Check whether the FETCH returned a row
        {System.out.println("Item ID " + id + " Item format " +
            i_fmt + " Item ROWID ");
         printBytes(rowid.getBytes());
                                // Use the getBytes method to
                                // convert the value to bytes for printing
         #sql {FETCH :positrowid INTO :id, :i_fmt, :rowid};
                                // Retrieve the next row
        }
    positrowid.close();        // Close the iterator
}
```

図 53. イテレーターを使用した ROWID 値の検索の例

以下のコーディング例は、3 つの ROWID パラメーター、すなわち IN パラメーター、OUT パラメーター、および INOUT パラメーターを含むストアード・プロシージャの呼び出しを示したものです。

```

java.sql.RowId in_rowid = rowid;
java.sql.RowId out_rowid = null;
java.sql.RowId inout_rowid = rowid;
// Declare an input, output, and
// input/output ROWID parameter
...
#sql [myConnCtx] {CALL SP_ROWID(:IN in_rowid,
                             :OUT out_rowid,
                             :INOUT inout_rowid)};
// Call the stored procedure
System.out.println("Parameter values from SP_ROWID call: ");
System.out.println("Output parameter value ");
printBytes(out_rowid.getBytes());
// Use the getBytes method to
// convert the value to bytes for printing
System.out.println("Input/output parameter value ");
printBytes(inout_rowid.getBytes());

```

図 54. ROWID パラメーターを指定してストアード・プロシージャを呼び出す例

## SQLJ アプリケーションにおける特殊タイプ

SQLJ プログラムでは、特殊タイプは実行可能節で CREATE DISTINCT TYPE ステートメントを使用して作成することができます。

また、実行可能節で CREATE TABLE を使用して、このタイプの列を含む表を作成することもできます。このタイプの列からデータを取得するか、またはこのタイプの列を更新する場合は、特殊タイプのベースとなる組み込みタイプに対応するデータ・タイプと共に、Java ID を使用します。

以下の例では、INTEGER タイプをベースにした特殊タイプを作成し、このタイプの列を持つ表を作成して、行をその表に挿入し、さらにその表から行を取得します。

```

String empNumVar;
int shoeSizeVar;
...
#sql [myConnCtx] {CREATE DISTINCT TYPE SHOESIZE AS INTEGER WITH COMPARISONS};
// Create distinct type
#sql [myConnCtx] {COMMIT}; // Commit the create
#sql [myConnCtx] {CREATE TABLE EMP_SHOE
  (EMPNO CHAR(6), EMP_SHOE_SIZE SHOESIZE)};
// Create table using distinct type
#sql [myConnCtx] {COMMIT}; // Commit the create
#sql [myConnCtx] {INSERT INTO EMP_SHOE
  VALUES('000010',6)}; // Insert a row in the table
#sql [myConnCtx] {COMMIT}; // Commit the INSERT
#sql [myConnCtx] {SELECT EMPNO, EMP_SHOE_SIZE
  INTO :empNumVar, :shoeSizeVar
  FROM EMP_SHOE}; // Retrieve the row
System.out.println("Employee number: " + empNumVar +
  " Shoe size: " + shoeSizeVar);

```

図 55. 特殊タイプの定義および使用



## ARRAY パラメーターを持つストアード・プロシージャの、SQLJ アプリケーションでの呼び出し

IBM Data Server Driver for JDBC and SQLJ で実行される SQLJ アプリケーションは、ARRAY パラメーターを持つストアード・プロシージャを呼び出すことができます。

java.sql.Array オブジェクトは、ストアード・プロシージャで IN、OUT、または INOUT パラメーターとして使用できます。

DB2Connection.createArrayOf メソッドを使用すると、ARRAY 入力ストアード・プロシージャ・パラメーターに値が割り当てられます。

ARRAY 出力ストアード・プロシージャ・パラメーターからデータを取り出すには、次の 2 つの方法があります。

- java.sql.Array.getArray メソッドを使用して出力パラメーターの内容を Java 配列に取り出します。
- java.sql.Array.getResultSet メソッドを使用して出力パラメーター・データを ResultSet オブジェクトに取り出します。次に ResultSet メソッドを使用して配列のエレメントを取り出します。ResultSet の各行には、次の 2 つの列が含まれています。
  - 配列の指標 (1 から始まります)
  - 配列エレメント

配列エレメントは、getObject メソッドで ResultSet から取り出す必要があります。

**例:** ストアード・プロシージャ GET\_EMP\_DATA の入力パラメーター IN\_PHONE と出力パラメーター OUT\_PHONE が以下のように定義される配列だとします。

```
CREATE TYPE PHONENUMBERS AS VARCHAR(10) ARRAY[5]
```

GET\_EMP\_DATA をこの 2 つのパラメーターで呼び出します。

```
Connection con;
String type = "CHAR";
String [] contents = {"1234", "5678", "9101"};
...
com.ibm.db2.jcc.DB2Connection db2con = (com.ibm.db2.jcc.DB2Connection) con;
// Cast the Connection as a DB2Connection
// so you can use the
// DB2Connection.createArrayOf method
java.sql.Array inPhoneData = db2con.createArrayOf(type, contents);
java.sql.Array outPhoneData;
try {
    #sql [db2con] {CALL GET_EMP_DATA(:IN inPhoneData, :OUT outPhoneData ) };
}
catch( SQLException e )
{
    throw e;
}
ResultSet rs = outPhoneData.getResultSet();
while (rs.next()) {
    String phoneNum = rs.getString(2); // Get phone number
    System.out.println("Phone number = " + phoneNum);
}
```

## SQLJ アプリケーションにおけるセーブポイント

IBM Data Server Driver for JDBC and SQLJ では、任意の形式の SQL SAVEPOINT ステートメントを SQLJ プログラムに組み込むことができます。

SQL セーブポイントは、作業単位内の特定の時点のデータおよびスキーマの状態を表します。セーブポイントを設定したり、セーブポイントを解除したり、データおよびスキーマをセーブポイントが表す状態へリストアするための SQL ステートメントがあります。

以下の例は、セーブポイントの設定、セーブポイントへのロールバック、およびセーブポイントの解除の方法を示しています。

図 56. SQLJ アプリケーションでのセーブポイントの設定、ロールバック、および解除

```
#sql context Ctx;           // Create connection context class Ctx
String empNumVar;
int shoeSizeVar;
...
try {                       // Load the JDBC driver
    Class.forName("com.ibm.db2.jcc.DB2Driver");
}
catch (ClassNotFoundException e) {
    e.printStackTrace();
}
Connection jdbccon=
    DriverManager.getConnection("jdbc:db2://sysmv1.stl.ibm.com:5021/NEWYORK",
        userid,password);
                                // Create JDBC connection object jdbccon
jdbccon.setAutoCommit(false); // Do not autocommit
Ctx ctxt=new Ctx(jdbccon);
                                // Create connection context object myConnCtx
                                // for the connection to NEWYORK
...
                                // Perform some SQL
#sql [ctxt] {COMMIT};          // Commit the transaction
                                // Commit the create

#sql [ctxt]
    {INSERT INTO EMP_SHOE VALUES ('000010', 6)};
                                // Insert a row

#sql [ctxt]
    {SAVEPOINT SVPT1 ON ROLLBACK RETAIN CURSORS};
                                // Create a savepoint

...
#sql [ctxt]
    {INSERT INTO EMP_SHOE VALUES ('000020', 10)};
                                // Insert another row
#sql [ctxt] {ROLLBACK TO SAVEPOINT SVPT1};
                                // Roll back work to the point
                                // after the first insert

...
#sql [ctxt] {RELEASE SAVEPOINT SVPT1};
                                // Release the savepoint
ctx.close();                   // Close the connection context
```

---

## SQLJ アプリケーションでの XML データ

SQLJ アプリケーションでは、XML 列へのデータの保管、および XML 列からのデータの取り出しが可能です。

DB2 表では、XML 組み込みデータ・タイプを使用して XML データをノードの構造化セットとしてツリー形式で列に保管します。

アプリケーションでは、XML データは直列化されたストリング形式になります。

SQLJ アプリケーションでは以下を行うことができます。

- INSERT または UPDATE ステートメントを使用して XML 文書全体を XML 列に保管します。
- 単一行の SELECT ステートメントまたはイテレーターを使用して XML 文書全体を XML 列から取得します。
- SQL XMLQUERY 関数を使用してシーケンスをデータベース内に取り出してから、単一行の SELECT ステートメントまたはイテレーターを使用して、シリアル化された XML ストリング・データをアプリケーション変数に取り出すことによって、XML 列の文書からシーケンスを取り出します。
- ストリング 'XQUERY' が前に付加されている XQuery 式を使用して、シーケンスの要素をデータベース内の結果表に取り出すことによって、XML 列の文書からシーケンスを取り出します。結果表の各行はシーケンス内のアイテムを表します。続いて、単一行の SELECT ステートメントまたはイテレーターを使用してデータをアプリケーション変数に取り出します。
- SQL XMLTABLE 関数を使用して結果表を定義し、その表を取り出すことによって、XML 列の文書からユーザー定義表としてシーケンスを取り出します。続いて、単一行の SELECT ステートメントまたはイテレーターを使用して、データを結果表からアプリケーション変数に取り出します。

JDBC 4.0 の `java.sql.SQLXML` オブジェクトを使用して、XML 列内のデータの取り出しおよび更新を行うことができます。 `ResultSetMetaData.getColumnTypeName` などのメタデータ・メソッドを呼び出すと、XML 列タイプの整数値 `java.sql.Types.SQLXML` が戻されます。

## SQLJ アプリケーションでの XML 列の更新

表の XML 列について SQLJ アプリケーションでデータを更新または挿入する場合は、入力データは直列化されたストリング形式である必要があります。

XML 列の更新に使用できるホスト式のデータ・タイプは以下のとおりです。

- `java.sql.SQLXML` (SDK for Java バージョン 6 以降、および IBM Data Server Driver for JDBC and SQLJ バージョン 4.0 以降が必要)
- `com.ibm.db2.jcc.DB2Xml` (非推奨)
- `String`
- `byte`
- `Blob`
- `Clob`
- `sqlj.runtime.AsciiStream`
- `sqlj.runtime.BinaryStream`
- `sqlj.runtime.CharacterStream`

ストリーム・タイプの場合、ストリームの長さを JDBC ドライバーに受け渡すために、`java.io.typeInputStream` ホスト式ではなく、`sqlj.runtime.typeStream` ホスト式を使用する必要があります。

XML データのエンコード方式は、データ自体から導出する (内部的にエンコードされた データという) か、または外部ソースから導出する (外部的にエンコードされた データという) ことができます。データベース・サーバーにバイナリー・データとして送信される XML データは、内部的にエンコードされたデータとして処理されます。データ・ソースに文字データとして送信される XML データは、外部的にエンコードされたデータとして処理されます。外部エンコード方式は JVM のデフォルトのエンコード方式です。

Java アプリケーションの外部エンコード方式は、常に Unicode エンコード方式です。

外部的にエンコードされたデータに、内部エンコード方式を含めることができます。つまり、このデータはデータ・ソースに文字データとして送信されますが、このデータにエンコード方式の情報を含めることができます。データ・ソースにより、以下のように内部エンコード方式と外部エンコード方式の間の非互換性が処理されます。

- データ・ソースが DB2 Database for Linux, UNIX, and Windows の場合、外部エンコード方式と内部エンコード方式の間に互換性がなく、外部エンコード方式と内部エンコード方式が Unicode でない場合は、データ・ソースによりエラーが生成されます。外部エンコード方式と内部エンコード方式が Unicode の場合は、データ・ソースでは内部エンコード方式が無視されます。
- データ・ソースが DB2 for z/OS の場合は、データ・ソースでは内部エンコード方式が無視されます。

XML 列のデータは UTF-8 エンコード方式で保管されます。

**例:** 次のステートメントを使用して、データを String ホスト式 xmlString から表の XML 列に挿入するとします。xmlString は文字タイプのため、内部エンコード方式が指定されているかどうかに関係なく、外部エンコード方式が使用されます。

```
#sql [ctx] {INSERT INTO CUSTACC VALUES (1, :xmlString)};
```

**例:** データを xmlString から CP500 エンコード方式のバイト配列にコピーするとします。データには、CP500 のエンコード方式宣言を持つ XML 宣言が含まれています。次に、byte[] ホスト式から表の XML 列にデータを挿入します。

```
byte[] xmlBytes = xmlString.getBytes("CP500");  
#sql[ctx] {INSERT INTO CUSTACC VALUES (4, :xmlBytes)};
```

バイト・ストリングは、内部的にエンコードされたデータとみなされます。データはその内部コード化スキームから UTF-8 に変換され、必要な場合はデータ・ソース上に階層形式で保管されます。

**例:** データを xmlString から US-ASCII エンコード方式のバイト配列にコピーするとします。sqlj.runtime.AsciiStream ホスト式を構成し、データを sqlj.runtime.AsciiStream ホスト式からデータ・ソース上にある表の XML 列に挿入します。

```
byte[] b = xmlString.getBytes("US-ASCII");  
java.io.ByteArrayInputStream xmlAsciiInputStream =  
    new java.io.ByteArrayInputStream(b);  
sqlj.runtime.AsciiStream sqljXmlAsciiStream =  
    new sqlj.runtime.AsciiStream(xmlAsciiInputStream, b.length);  
#sql[ctx] {INSERT INTO CUSTACC VALUES (4, :sqljXmlAsciiStream)};
```

sqljXmlAsciiStream はストリーム・タイプであるため、内部エンコード方式が使用されます。データはその内部エンコード方式から UTF-8 エンコード方式に変換され、データ・ソース上に階層形式で保管されます。

**例: sqlj.runtime.CharacterStream ホスト式:** sqlj.runtime.CharacterStream ホスト式を構成し、データを sqlj.runtime.CharacterStream ホスト式から表の XML 列に挿入します。

```
java.io.StringReader xmlReader =
    new java.io.StringReader(xmlString);
sqlj.runtime.CharacterStream sqljXmlCharacterStream =
    new sqlj.runtime.CharacterStream(xmlReader, xmlString.length());
#sql [ctx] {INSERT INTO CUSTACC VALUES (4, :sqljXmlCharacterStream)};
```

sqljXmlCharacterStream は文字タイプのため、内部エンコード方式が指定されているかどうかに関係なく、外部エンコード方式が使用されます。

**例: 文書を XML 列から java.sql.SQLXML ホスト式に取り出し、このデータを表の XML 列に挿入します。**

```
java.sql.ResultSet rs = s.executeQuery ("SELECT * FROM CUSTACC");
rs.next();
java.sql.SQLXML xmlObject = (java.sql.SQLXML)rs.getObject(2);
#sql [ctx] {INSERT INTO CUSTACC VALUES (6, :xmlObject)};
```

データを取り出した後、このデータは UTF-8 エンコード方式のままなので、このデータを他の XML 列に挿入する場合も変換は実行されません。

**例: 文書を XML 列から com.ibm.db2.jcc.DB2Xml ホスト式に取り出し、このデータを表の XML 列に挿入します。**

```
java.sql.ResultSet rs = s.executeQuery ("SELECT * FROM CUSTACC");
rs.next();
com.ibm.db2.jcc.DB2Xml xmlObject = (com.ibm.db2.jcc.DB2Xml)rs.getObject(2);
#sql [ctx] {INSERT INTO CUSTACC VALUES (6, :xmlObject)};
```

データを取り出した後、このデータは UTF-8 エンコード方式のままなので、このデータを他の XML 列に挿入する場合も変換は実行されません。

## SQLJ アプリケーションにおける XML データの取り出し

SQLJ アプリケーションで、データをデータベース表の XML 列から取り出す場合は、出力データは明示的または暗黙的に直列化される必要があります。

データの XML 列からの取り出しに使用できるホスト式またはイテレーターのデータ・タイプは以下のとおりです。

- java.sql.SQLXML (SDK for Java バージョン 6 以降、および IBM Data Server Driver for JDBC and SQLJ バージョン 4.0 以降が必要)
- com.ibm.db2.jcc.DB2Xml (非推奨)
- String
- byte[]
- sqlj.runtime.AsciiStream
- sqlj.runtime.BinaryStream
- sqlj.runtime.CharacterStream

アプリケーションで、データの取り出しの前に XMLSERIALIZE 関数が呼び出されない場合は、データは UTF-8 から文字データ・タイプ用の外部アプリケーション・エンコード方式か、またはバイナリー・データ・タイプ用の内部エンコード方式に変換されます。XML 宣言は追加されません。ホスト式が `java.sql.SQLXML` または `com.ibm.db2.jcc.DB2Xml` タイプのオブジェクトの場合は、追加のメソッドを呼び出して、データをこのオブジェクトから取り出す必要があります。呼び出すメソッドにより、出力データのエンコード方式、およびエンコード方式の指定を含む XML 宣言が追加されるかどうかが決まります。

次の表には、`java.sql.SQLXML` または `com.ibm.db2.jcc.DB2Xml` オブジェクトからのデータの取り出しに呼び出すことができるメソッド、および対応する出力データ・タイプと XML 宣言でのエンコード方式のタイプがリストされています。

表 21. SQLXML および DB2Xml メソッド、データ・タイプ、および追加されるエンコード仕様

メソッド	出力データ・タイプ	追加される XML 内部エンコード宣言のタイプ
<code>SQLXML.getBinaryStream</code>	<code>InputStream</code>	なし
<code>SQLXML.getCharacterStream</code>	<code>Reader</code>	なし
<code>SQLXML.getSource</code>	<code>Source</code>	なし
<code>SQLXML.getString</code>	<code>String</code>	なし
<code>DB2Xml.getDB2AsciiStream</code>	<code>InputStream</code>	なし
<code>DB2Xml.getDB2BinaryStream</code>	<code>InputStream</code>	なし
<code>DB2Xml.getDB2Bytes</code>	<code>byte[]</code>	なし
<code>DB2Xml.getDB2CharacterStream</code>	<code>Reader</code>	なし
<code>DB2Xml.getDB2String</code>	<code>String</code>	なし
<code>DB2Xml.getDB2XmlAsciiStream</code>	<code>InputStream</code>	US-ASCII
<code>DB2Xml.getDB2XmlBinaryStream</code>	<code>InputStream</code>	<code>getDB2XmlBinaryStream targetEncoding</code> パラメーターで指定
<code>DB2Xml.getDB2XmlBytes</code>	<code>byte[]</code>	<code>DB2Xml.getDB2XmlBytes targetEncoding</code> パラメーターで指定
<code>DB2Xml.getDB2XmlCharacterStream</code>	<code>Reader</code>	ISO-10646-UCS-2
<code>DB2Xml.getDB2XmlString</code>	<code>String</code>	ISO-10646-UCS-2

戻されるデータに対してアプリケーションが XMLSERIALIZE 関数を実行すると、関数の実行後に、そのデータには XML データ・タイプではなく XMLSERIALIZE 関数で指定されたデータ・タイプが含まれます。そのため、ドライバーは指定されたタイプとしてそのデータを処理し、内部エンコード宣言をすべて無視します。

**例:** データを XML 列から String ホスト式に取り出します。

```
#sql iterator XmlStringIter (int, String);
#sql [ctx] siter = {SELECT C1, CADOC from CUSTACC};
#sql {FETCH :siter INTO :row, :outString};
```

String タイプは文字タイプであるため、データは UTF-8 から外部エンコード方式に変換され (これがデフォルトの JVM エンコード方式)、XML 宣言なしで返されます。

**例:** データを XML 列から byte[] ホスト式に取り出します。



```
#sql iterator XmlByteArrayIter (int, byte[]);
XmlByteArrayIter biter = null;
#sql [ctx] biter = {SELECT c1, CADOC from CUSTACC};
#sql {FETCH :biter INTO :row, :outBytes};
```

byte[] タイプはバイナリー・タイプであるため、UTF-8 エンコード方式からのデータ変換は実行されず、データは XML 宣言なしで返されます。

**例:** 文書を XML 列から java.sql.SQLXML ホスト式に取り出しますが、バイナリー・ストリームのデータが必要な場合です。

```
#sql iterator SqlXmlIter (int, java.sql.SQLXML);
SqlXmlIter SQLXMLiter = null;
java.sql.SQLXML outSqlXml = null;
#sql [ctx] SqlXmlIter = {SELECT c1, CADOC from CUSTACC};
#sql {FETCH :SqlXmlIter INTO :row, :outSqlXml};
java.io.InputStream XmlStream = outSqlXml.getBinaryStream();
```

FETCH ステートメントでは、データが UTF-8 エンコード方式で SQLXML オブジェクトに取り出されます。SQLXML.getBinaryStream では、データがバイナリー・ストリームで保管されます。

**例:** 文書を XML 列から com.ibm.db2.jcc.DB2Xml ホスト式に取り出しますが、UTF-8 の内部エンコード方式の指定が含まれた XML 宣言を持つバイト・ストリングとしてデータが必要な場合です。

```
#sql iterator DB2XmlIter (int, com.ibm.db2.jcc.DB2Xml);
DB2XmlIter db2xmliter = null;
com.ibm.db2.jcc.DB2Xml outDB2Xml = null;
#sql [ctx] db2xmliter = {SELECT c1, CADOC from CUSTACC};
#sql {FETCH :db2xmliter INTO :row, :outDB2Xml};
byte[] byteArray = outDB2XML.getDB2XmlBytes("UTF-8");
```

FETCH ステートメントでは、データが UTF-8 エンコード方式で DB2Xml オブジェクトに取り出されます。UTF-8 引数のある getDB2XmlBytes メソッドでは、UTF-8 エンコード方式の指定を含む XML 宣言が追加され、データがバイト配列で保管されます。

---

## SDK for Java バージョン 5 の機能の SQLJ での利用

SQLJ アプリケーションで、SDK for Java バージョン 5 で導入されたいくつかの機能を使用することができます。

### 静的インポート

import static 構文を使用すれば、静的メンバーが属しているクラスの名前でそれらのメンバーを修飾せずに、静的メンバーにアクセスすることができます。つまり SQLJ アプリケーションの場合、静的メンバーを修飾せずにホスト式でそれらのメンバーを使用することができます。

**例:** 次の形式のホスト式を宣言したいとします。

```
double r = cos(PI * E);
```

cos、PI、および E は java.lang.Math クラスのメンバーです。cos、PI、および E を明示的に修飾しないで r を宣言するには、次の import static ステートメントをプログラム内に含めます。



```
import static java.lang.Math.*;
```

## アノテーション

Java アノテーションとは、メタデータを Java プログラムに追加するための手段のことで、これらの Java プログラムがツールとライブラリーによって処理される方法に影響を与える可能性もあります。アノテーションは、インターフェース宣言と同様なアノテーション型宣言で宣言されます。Java アノテーションは、以下のタイプのクラスまたはインターフェースに指定することができます。

- クラス宣言
- インターフェース宣言
- ネスト・クラス宣言
- ネスト・インターフェース宣言

Java アノテーションは SQLJ プログラムに直接組み込むことはできませんが、アノテーションを Java ソース・コードに組み込んだ後で、このソース・コードを SQLJ プログラムに組み込むことができます。

**例:** 次のマーカー・アノテーションを MyAnnot.java というプログラム内で宣言するとします。

```
public @interface MyAnot { }
```

次のマーカー・アノテーションも MyAnnot2.java というプログラム内で宣言します。

```
public @interface MyAnot2 { }
```

これで、これらのアノテーションを SQLJ プログラム内で以下のように使用することができます。

```
// Class annotations
@MyAnot2 public @MyAnot class TestAnnotation
{
    // Field annotation
    @MyAnot
    private static final int field1 = 0;
    // Constructor annotation
    @MyAnot2 public @MyAnot TestAnnotation () { }
    // Method annotation
    @MyAnot
    public static void main (String a[])
    {
        TestAnnotation TestAnnotation_o = new TestAnnotation();
        TestAnnotation_o.runThis();
    }
    // Inner class annotation
    public static @MyAnot class TestAnotherInnerClass { }
    // Inner interface annotation
    public static @MyAnot interface TestAnotInnerInterface { }
}
```

## 列挙型

列挙型とは、一連の順序付けられた値で構成されるデータ型のことです。SDK for Java バージョン 5 では、列挙型用の enum 型が導入されました。

Java の enum 型は SQLJ プログラムに直接組み込むことはできませんが、enum は次の場所に組み込むことができます。

- SQLJ プログラムに組み込む Java ソース・ファイル (.java ファイル) 内
- SQLJ のクラス宣言内

**例:** 以下の TestEnum.sqlj クラス宣言に enum 型が組み込まれています。

```
public class TestEnum2
{
    public enum Color {
        RED,ORANGE,YELLOW,GREEN,BLUE,INDIGO,VIOLET}
    Color color = null;
    switch (color) {
case RED:
    System.out.println("Red is at one end of the spectrum.");
    #sql[ctx] { INSERT INTO MYTABLE VALUES (:color) };
    break;
case VIOLET:
    System.out.println("Violet is on the other end of the spectrum.");
    break;
case ORANGE:
case YELLOW:
case GREEN:
case BLUE:
case INDIGO:
    System.out.println("Everything else is in the middle.");
    break;
}
}
```

## 総称

総称を Java プログラム内で使用して、型を Java コレクションに割り当てることができます。SQLJ 変換プログラムでは、Java の総称構文が許容されます。SQLJ プログラムで使用できる総称の例は以下のとおりです。

- List オブジェクトの List:

```
List <List<String>> strList2 = new ArrayList<List<String>>();
```
- キーと値の組みに String 型を持つ HashMap:

```
Map <String,String> map = new HashMap<String,String>();
```
- 任意の型のエレメントを持つ List を使用するメソッド:

```
public void mthd(List <?> obj) {
    ...
}
```

総称は SQLJ ホスト変数内で使用できますが、総称の使用の意義は、SQLJ 変換プログラムではそれらのホスト変数の型を判別できないため限られています。

## 拡張された for ループ

拡張された for を使用すれば、一連の命令をコレクションまたは配列の各メンバーに対して実行することを指定できます。イテレーターを、拡張された for ループ内でホスト式で使用することができます。

**例:** 配列 names 内の項目ごとに表 TAB に対して INSERT を実行します。

```
String[] names = {"ABC","DEF","GHI"};
for (String n : names)
{
    #sql {INSERT INTO TAB (VARCHARCOL) VALUES (:n) };
}
```

## 可変引数

可変引数により、任意の数の値をメソッドに受け渡すことが容易になります。メソッド宣言の最後の引数の位置にある可変引数により、最後の引数が配列または一続きの引数であることが示されます。SQLJ プログラムでは、受け渡されたホスト式内の引数を使用することができます。

**例:** 任意の数の Object 型のパラメーターを、各パラメーター値を表 TAB に挿入するメソッドに受け渡します。

```
public void runThis(Object... objects) throws SQLException
{
    for (Object obj : objects)
    {
        #sql { INSERT INTO TAB (VARCHARCOL) VALUES(:obj) };
    }
}
```

---

## SQLJ アプリケーションにおけるトランザクション制御

SQLJ アプリケーションでは、他のタイプの SQL アプリケーションと同様に、トランザクション制御にトランザクションの明示的または暗黙的なコミットおよびロールバック、およびトランザクションの分離レベルの設定が関係します。

### SQLJ トランザクションの分離レベルの設定

SQLJ プログラムの作業単位の分離レベルを設定するには、SET TRANSACTION ISOLATION LEVEL 節を使用します。

SET TRANSACTION ISOLATION LEVEL 節に指定できる値と、それぞれの値に対応する DB2 での等価値を以下の表にまとめます。

表 22. SQLJ と DB2 の分離レベルの等価値

SET TRANSACTION 値	DB2 の分離レベル
SERIALIZABLE	反復可能読み取り
REPEATABLE READ	読み取り固定
READ COMMITTED	カーソル固定
READ UNCOMMITTED	非コミット読み取り

分離レベルは、基礎になっている JDBC 接続と SQLJ 接続の両方に影響を及ぼします。

### SQLJ トランザクションのコミットまたはロールバック

SQLJ 接続の自動コミットを使用不可にした場合は、明示的なコミット操作またはロールバック操作を実行する必要があります。

そのためには、SQL の COMMIT ステートメントまたは ROLLBACK ステートメントを含んだ実行節を使用します。

SQLJ プログラムでトランザクションをコミットするには、以下のようなステートメントを使用します。

```
#sql [myConnCtx] {COMMIT};
```

SQLJ プログラムでトランザクションをロールバックするには、以下のようなステートメントを使用します。

```
#sql [myConnCtx] {ROLLBACK};
```

---

## SQLJ アプリケーションで SQL のエラーと警告を処理する

SQLJ 節は、SQL エラーが発生すると `SQLException` をスローしますが、ほとんどの SQL 警告では `SQLException` をスローしません。

SQLJ は、以下の状況で `SQLException` を生成します。

- SQL ステートメントが負の SQL エラー・コードを返したとき
- `SELECT INTO SQL` ステートメントが +100 の SQL エラー・コードを返したとき

その他の SQL 警告については、明示的に確認する必要があります。

- SQL エラー処理のために、SQLJ ステートメントの周りに `try/catch` ブロックを組み込みます。
- SQL 警告処理のために、各 SQLJ ステートメントの後に `getWarnings` メソッドを呼び出します。

## SQLJ アプリケーションでの SQL エラーの処理

SQLJ の節では、エラー処理のために JDBC クラス `java.sql.SQLException` を使用します。

SQLJ アプリケーションで SQL エラーを処理するには、以下の手順を実行します。

1. `java.sql.SQLException` クラスをインポートします。
2. Java エラー処理の `try/catch` ブロックを使用して、SQL エラー発生時のプログラム・フローを変更します。
3. `SQLException` からエラー情報を取得します。

`getErrorCode` メソッドによって SQL エラー・コードを、`getSQLState` メソッドによって `SQLSTATE` をそれぞれ取得できます。

IBM Data Server Driver for JDBC and SQLJ を使用している場合は、`SQLException` を `DB2Diagnosable` オブジェクトにキャストすることによって、例外に関する追加情報を取得できます (JDBC アプリケーションで追加情報を取得する場合と同じです)。

Linux、UNIX、Windows 用の DB2 JDBC Type 2 ドライバー (DB2 JDBC Type 2 ドライバー) の場合は、標準の `SQLException` を使用して SQL エラー情報を取得します。

`SELECT` ステートメントが失敗したときに発生する SQL エラーを出力するコードを以下に示します。

```
try {
    #sql [ctxt] {SELECT LASTNAME INTO :empname
                FROM EMPLOYEE WHERE EMPNO='000010'};
```

```

}
catch(SQLException e) {
    System.out.println("Error code returned: " + e.getErrorCode());
}

```

## SQLJ アプリケーションでの SQL 警告の処理

SELECT INTO ステートメントの +100 の SQL エラー・コードを除いて、DB2 の警告では SQLException がスローされません。DB2 の警告を処理するには、java.sql.SQLWarning クラスに対するアクセス権をプログラムに与える必要があります。

警告に関する DB2 固有の情報を取得するには、com.ibm.db2.jcc.DB2Diagnosable インターフェースと com.ibm.db2.jcc.DB2Sqlca クラスに対するアクセス権もプログラムに与える必要があります。その後、以下の手順を実行します。

1. SQL 節の実行コンテキストをセットアップします。実行コンテキストのセットアップ方法については、『SQLJ での SQL ステートメントの実行の制御』を参照してください。
2. DB2 の警告をチェックするには、SQLJ 節を実行した後に getWarnings メソッドを呼び出します。

getWarnings からは、SQL ステートメントが生成する最初の SQLWarning オブジェクトが返されます。後続の SQLWarning オブジェクトは、最初のオブジェクトにチェーンングされます。

3. IBM Data Server Driver for JDBC and SQLJ 使用時に、SQLWarning オブジェクトから DB2 固有の情報を取得するには、『IBM Data Server Driver for JDBC and SQLJ 使用時の SQLException の処理』の手順を実行します。

実行コンテキスト execCtx を使用する SQL 節の SQLWarning オブジェクトを取得する例を以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```

ExecutionContext execCtx=myConnCtx.getExecutionContext(); 1
// Get default execution context from
// connection context

SQLWarning sqlWarn;
...
#sql [myConnCtx,execCtx] {SELECT LASTNAME INTO :empname
FROM EMPLOYEE WHERE EMPNO='000010'};
if ((sqlWarn = execCtx.getWarnings()) != null) 2
System.out.println("SQLWarning " + sqlWarn);

```

---

## SQLJ アプリケーションでのデータ・ソースへの接続のクローズ

データ・ソースへの接続を使用した作業が完了したら、データ・ソースへの接続を閉じる必要があります。そうすれば、接続コンテキスト・オブジェクトの DB2 と SQLJ のリソースがすぐに解放されます。

データ・ソースへの接続を閉じるには、いずれかの ConnectionContext.close メソッドを使用します。

- ConnectionContext.close() または ConnectionContext.close(ConnectionContext.CLOSE\_CONNECTION) を実行すると、データ・ソースへの接続と接続コンテキストの両方が閉じます。

- `ConnectionContext.close(ConnectionContext.KEEP_CONNECTION)` を実行すると、接続コンテキストは閉じますが、データ・ソースへの接続は閉じません。

データ・ソースへの接続を閉じないで接続コンテキストだけを閉じるコードを以下に示します。

```
...
ctx = new EzSqljctx(con0);           // Create a connection context object
                                     // from JDBC connection con0
...
EzSqljctx.close(ConnectionContext.KEEP_CONNECTION);
                                     // Close the connection context but keep
                                     // the connection to the data source open
```

## 第 5 章 IBM Data Server Driver for JDBC and SQLJ 使用時のセキュリティ

IBM Data Server Driver for JDBC and SQLJ を使用する場合、`securityMechanism` プロパティの値を指定してセキュリティ・メカニズムを選択します。

このプロパティは以下の方法のいずれかで設定できます。

- `DriverManager` インターフェースを使用する場合、`java.util.Properties` パラメーターを含む `getConnection` メソッドの形式を呼び出す前に、`java.util.Properties` オブジェクトで `securityMechanism` を設定します。
- `DataSource` インターフェースを使用しており、独自の `DataSource` オブジェクトを作成およびデプロイしている場合、`DataSource` オブジェクトを作成した後で `DataSource.setSecurityMechanism` メソッドを呼び出します。

`DB2Connection.getDB2SecurityMechanism` メソッドを呼び出すことにより、接続に有効なセキュリティ・メカニズムを判別できます。

以下の表は、IBM Data Server Driver for JDBC and SQLJ がサポートするセキュリティ・メカニズム、およびそれらのセキュリティ・メカニズムをサポートするデータ・ソースをリストしています。

表 23. IBM Data Server Driver for JDBC and SQLJ セキュリティ・メカニズムのデータベース・サーバー・サポート

セキュリティ・メカニズム	サポート		
	DB2 Database for Linux, UNIX, and Windows	DB2 for z/OS	IBM Informix Dynamic Server
ユーザー ID およびパスワード	はい	はい	はい
ユーザー ID のみ	はい	はい	はい
ユーザー ID および暗号化されたパスワード	はい	はい	はい
暗号化されたユーザー ID	はい	はい	いいえ
暗号化されたユーザー ID および暗号化されたパスワード	はい	はい	はい
暗号化されたユーザー ID および暗号化されたセキュリティ・センシティブ・データ	いいえ	はい	いいえ
暗号化されたユーザー ID、暗号化されたパスワード、暗号化されたセキュリティ・センシティブ・データ	はい	はい	いいえ
Kerberos <sup>1</sup>	はい	はい	いいえ
プラグイン <sup>1</sup>	はい	いいえ	いいえ

注:

1. IBM Data Server Driver for JDBC and SQLJ Type 4 接続にのみ使用可能。



以下の表は、IBM Data Server Driver for JDBC and SQLJ がサポートするセキュリティ・メカニズム、および各セキュリティ・メカニズムを指定するために securityMechanism プロパティに指定する必要がある値をリストしています。

デフォルトのセキュリティ・メカニズムは CLEAR\_TEXT\_PASSWORD\_SECURITY です。サーバーが、CLEAR\_TEXT\_PASSWORD\_SECURITY をサポートしない一方で、ENCRYPTED\_USER\_AND\_PASSWORD\_SECURITY をサポートするときは、IBM Data Server Driver for JDBC and SQLJ ドライバーはセキュリティ・メカニズムを ENCRYPTED\_USER\_AND\_PASSWORD\_SECURITY に更新し、サーバーへの接続を試みます。リクエスターとサーバー間でこれ以外のセキュリティ・メカニズム・サポートの不一致がある場合には、エラーになります。

表 24. IBM Data Server Driver for JDBC and SQLJ によってサポートされるセキュリティ・メカニズム

セキュリティ・メカニズム	securityMechanism プロパティ値
ユーザー ID およびパスワード	DB2BaseDataSource.CLEAR_TEXT_PASSWORD_SECURITY
ユーザー ID のみ	DB2BaseDataSource.USER_ONLY_SECURITY
ユーザー ID および暗号化されたパスワード	DB2BaseDataSource.ENCRYPTED_PASSWORD_SECURITY
暗号化されたユーザー ID	DB2BaseDataSource.ENCRYPTED_USER_ONLY_SECURITY
暗号化されたユーザー ID および暗号化されたパスワード	DB2BaseDataSource.ENCRYPTED_USER_AND_PASSWORD_SECURITY
暗号化されたユーザー ID および暗号化されたセキュリティ・センシティブ・データ	DB2BaseDataSource.ENCRYPTED_USER_AND_DATA_SECURITY
暗号化されたユーザー ID、暗号化されたパスワード、暗号化されたセキュリティ・センシティブ・データ	DB2BaseDataSource.ENCRYPTED_USER_PASSWORD_AND_DATA_SECURITY
Kerberos	DB2BaseDataSource.KERBEROS_SECURITY
プラグイン	DB2BaseDataSource.PLUGIN_SECURITY

以下の表は、考えられる DB2 Database for Linux, UNIX, and Windows サーバー認証タイプ、および互換性のある IBM Data Server Driver for JDBC and SQLJ securityMechanism プロパティ値を示しています。

表 25. 互換性のある DB2 Database for Linux, UNIX, and Windows サーバー認証タイプおよび IBM Data Server Driver for JDBC and SQLJ securityMechanism 値

DB2 Database for Linux, UNIX, and Windows	
サーバー認証タイプ	securityMechanism 設定
CLIENT	USER_ONLY_SECURITY
SERVER	CLEAR_TEXT_PASSWORD_SECURITY
SERVER_ENCRYPT	CLEAR_TEXT_PASSWORD_SECURITY、 ENCRYPTED_PASSWORD_SECURITY、または ENCRYPTED_USER_AND_PASSWORD_SECURITY
DATA_ENCRYPT	ENCRYPTED_USER_PASSWORD_AND_DATA_SECURITY
KERBEROS	KERBEROS_SECURITY または PLUGIN_SECURITY <sup>2</sup>

表 25. 互換性のある DB2 Database for Linux, UNIX, and Windows サーバー認証タイプおよび IBM Data Server Driver for JDBC and SQLJ securityMechanism 値 (続き)

DB2 Database for Linux, UNIX, and Windows サーバー認証タイプ	securityMechanism 設定
KRB_SERVER_ENCRYPT	KERBEROS_SECURITY、PLUGIN_SECURITY <sup>1</sup> 、 ENCRYPTED_PASSWORD_SECURITY、または ENCRYPTED_USER_AND_PASSWORD_SECURITY
GSSPLUGIN	PLUGIN_SECURITY <sup>1</sup> または KERBEROS_SECURITY
GSS_SERVER_ENCRYPT <sup>3</sup>	CLEAR_TEXT_PASSWORD_SECURITY、 ENCRYPTED_PASSWORD_SECURITY、 ENCRYPTED_USER_AND_PASSWORD_SECURITY、 PLUGIN_SECURITY、または KERBEROS_SECURITY
注:	
1. PLUGIN_SECURITY の場合、プラグインは Kerberos プラグインでなければなりません。	
2. PLUGIN_SECURITY の場合、サーバーでのプラグインのうちの 1 つが、Kerberos をサポートするプラグインとして自らを識別します。	
3. GSS_SERVER_ENCRYPT は GSSPLUGIN と SERVER_ENCRYPT の組み合わせです。	

## IBM Data Server Driver for JDBC and SQLJ 使用時のユーザー ID およびパスワード・セキュリティ

IBM Data Server Driver for JDBC and SQLJ を使用する場合、使用可能なセキュリティの方法の 1 つが、ユーザー ID およびパスワード・セキュリティです。

JDBC 接続で、ユーザー ID およびパスワード・セキュリティを指定する場合、以下のいずれかの技法を使用します。

**DriverManager インターフェースの場合:** DriverManager.getConnection 呼び出しで、ユーザー ID およびパスワードを直接指定することができます。以下に例を示します。

```
import java.sql.*;          // JDBC base
...
String id = "dbadm";       // Set user ID
String pw = "dbadm";      // Set password
String url = "jdbc:db2://mvs1.sj.ibm.com:5021/san_jose";
                          // Set URL for the data source

Connection con = DriverManager.getConnection(url, id, pw);
                          // Create connection
```

別の方法として、ユーザー ID とパスワードを URL スtring に直接設定することができます。以下に例を示します。

```
import java.sql.*;          // JDBC base
...
String url =
    "jdbc:db2://mvs1.sj.ibm.com:5021/san_jose:user=dbadm;password=dbadm";
                          // Set URL for the data source
Connection con = DriverManager.getConnection(url);
                          // Create connection
```

別の方法として、Properties オブジェクトに user および password プロパティを設定し、Properties オブジェクトをパラメーターとして組み込んだフォームの getConnection メソッドを呼び出して、ユーザー ID およびパスワードを設定できます。オプションで、securityMechanism プロパティを設定して、ユーザー ID およびパスワード・セキュリティを使用していることを示すことができます。以下に例を示します。

```
import java.sql.*;                // JDBC base
import com.ibm.db2.jcc.*;        // IBM Data Server Driver for JDBC
                                  // and SQLJ implementation of JDBC
...
Properties properties = new java.util.Properties();
                                  // Create Properties object
properties.put("user", "dbadm");  // Set user ID for the connection
properties.put("password", "dbadm"); // Set password for the connection
properties.put("securityMechanism",
    new String(" " + com.ibm.db2.jcc.DB2BaseDataSource.CLEAR_TEXT_PASSWORD_SECURITY +
    ""));
                                  // Set security mechanism to
                                  // user ID and password
String url = "jdbc:db2://mvs1.sj.ibm.com:5021/san_jose";
                                  // Set URL for the data source
Connection con = DriverManager.getConnection(url, properties);
                                  // Create connection
```

**DataSource インターフェースの場合:** DataSource.getConnection を呼び出して、ユーザー ID およびパスワードを直接指定することができます。以下に例を示します。

```
import java.sql.*;                // JDBC base
import com.ibm.db2.jcc.*;        // IBM Data Server Driver for JDBC
                                  // and SQLJ implementation of JDBC
...
Context ctx=new InitialContext(); // Create context for JNDI
DataSource ds=(DataSource)ctx.lookup("jdbc/sampled");
                                  // Get DataSource object
String id = "dbadm";             // Set user ID
String pw = "dbadm";            // Set password
Connection con = ds.getConnection(id, pw);
                                  // Create connection
```

DataSource オブジェクトを作成してデプロイする場合、DataSource オブジェクトの作成後に、DataSource.setUser および DataSource.setPassword メソッドを呼び出して、ユーザー ID およびパスワードを設定できます。オプションで、DataSource.setSecurityMechanism メソッド・プロパティを呼び出して、ユーザー ID およびパスワード・セキュリティを使用していることを示すことができます。以下に例を示します。

```
...
com.ibm.db2.jcc.DB2SimpleDataSource ds = // Create DB2SimpleDataSource object
    new com.ibm.db2.jcc.DB2SimpleDataSource();
ds.setDriverType(4);                // Set driver type
ds.setDatabaseName("san_jose");     // Set location
ds.setServerName("mvs1.sj.ibm.com"); // Set server name
ds.setPortNumber(5021);            // Set port number
ds.setUser("dbadm");               // Set user ID
ds.setPassword("dbadm");           // Set password
ds.setSecurityMechanism(
    com.ibm.db2.jcc.DB2BaseDataSource.CLEAR_TEXT_PASSWORD_SECURITY);
                                  // Set security mechanism to
                                  // user ID and password
```

---

## IBM Data Server Driver for JDBC and SQLJ 使用時のユーザー ID のみのセキュリティー

IBM Data Server Driver for JDBC and SQLJ を使用する場合、使用可能なセキュリティーの方法の 1 つが、ユーザー ID のみのセキュリティーです。

JDBC 接続で、ユーザー ID セキュリティーを指定する場合、以下のいずれかの技法を使用します。

**DriverManager インターフェースの場合:** Properties オブジェクトに、user および securityMechanism プロパティーを設定し、Properties オブジェクトをパラメーターとして組み込んだ形式の getConnection メソッドを呼び出して、ユーザー ID およびセキュリティーのメカニズムを設定します。以下に例を示します。

```
import java.sql.*;           // JDBC base
import com.ibm.db2.jcc.*;    // IBM Data Server Driver
                               // for JDBC and SQLJ
                               // implementation of JDBC
...
Properties properties = new Properties();
                               // Create a Properties object
properties.put("user", "db2adm"); // Set user ID for the connection
properties.put("securityMechanism",
    new String("" + com.ibm.db2.jcc.DB2BaseDataSource.USER_ONLY_SECURITY + ""));
                               // Set security mechanism to
                               // user ID only
String url = "jdbc:db2://mvs1.sj.ibm.com:5021/san_jose";
                               // Set URL for the data source
Connection con = DriverManager.getConnection(url, properties);
                               // Create the connection
```

**DataSource インターフェースの場合:** DataSource オブジェクトを作成してデプロイする場合、DataSource オブジェクトの作成後に、DataSource.setUser および DataSource.setSecurityMechanism メソッドを呼び出して、ユーザー ID およびセキュリティーのメカニズムを設定することができます。以下に例を示します。

```
import java.sql.*;           // JDBC base
import com.ibm.db2.jcc.*;    // IBM Data Server Driver
                               // for JDBC and SQLJ
                               // implementation of JDBC
...
com.ibm.db2.jcc.DB2SimpleDataSource db2ds =
    new com.ibm.db2.jcc.DB2SimpleDataSource();
                               // Create DB2SimpleDataSource object
db2ds.setDriverType(4);      // Set the driver type
db2ds.setDatabaseName("san_jose"); // Set the location
db2ds.setServerName("mvs1.sj.ibm.com");
                               // Set the server name
db2ds.setPortNumber(5021);  // Set the port number
db2ds.setUser("db2adm");    // Set the user ID
db2ds.setSecurityMechanism(
    com.ibm.db2.jcc.DB2BaseDataSource.USER_ONLY_SECURITY);
                               // Set security mechanism to
                               // user ID only
```

---

## 暗号化されたパスワード・セキュリティ、暗号化されたユーザー ID セキュリティ、または IBM Data Server Driver for JDBC and SQLJ 使用時の暗号化されたユーザー ID とパスワード・セキュリティ

IBM Data Server Driver for JDBC and SQLJ は、暗号化されたパスワード・セキュリティ、暗号化されたユーザー ID のセキュリティ、またはデータ・ソースにアクセスするための暗号化されたユーザー ID およびパスワードのセキュリティをサポートします。

暗号化パスワード・セキュリティ、暗号化されたユーザー ID セキュリティ、または暗号化ユーザー ID と暗号化パスワード・セキュリティを使用する場合、IBM Java Cryptography Extension (JCE) がクライアントで有効になっている必要があります。IBM JCE は、IBM SDK for Java バージョン 1.4.2 以降でその一部として含まれています。

IBM JCE は、IBM Data Server Driver for JDBC and SQLJ ドライバーから DB2 Database for Linux, UNIX, and Windows サーバーへの 256 ビットの暗号化されたクライアント/サーバー通信を使用するために必要です。

暗号化されたユーザー ID セキュリティまたは暗号化されたユーザー ID とパスワード・セキュリティに加えて、暗号化されたセキュリティ・センシティブ・データも使用することができます。セキュリティ・センシティブ・データの暗号化は、ENCRYPTED\_USER\_AND\_DATA\_SECURITY または ENCRYPTED\_USER\_PASSWORD\_AND\_DATA\_SECURITY securityMechanism 値によって指定できます。ENCRYPTED\_USER\_AND\_DATA\_SECURITY は、DB2 for z/OS サーバーへの接続にのみ有効です。

セキュリティ・センシティブ・データの暗号化を指定すると、DB2 for z/OS または DB2 Database for Linux, UNIX, and Windows データベース・サーバーは、以下のデータを暗号化します。

- 準備中、実行中、またはパッケージにバインドされている SQL ステートメント
- パラメーター情報の入出力
- 結果セット
- LOB データ
- XML データ
- 記述操作の結果

暗号化されたセキュリティ・センシティブ・データを使用する前に、z/OS 統合暗号化サービス機能 (Integrated Cryptographic Services Facility) を z/OS オペレーティング・システムにインストールし、使用可能にする必要があります。

JDBC 接続で、暗号化されたユーザー ID または暗号化されたパスワード・セキュリティを指定する場合、以下のいずれかの技法を使用します。

**DriverManager インターフェースの場合:** Properties オブジェクトに、user、password、および securityMechanism プロパティを設定し、Properties オブジェクトをパラメーターとして組み込んだフォームの getConnection メソッドを呼び出して、ユーザー ID、パスワード、およびセキュリティのメカニズムを設定します。例えば、次のようなコードを使用して、ユーザー ID および暗号化されたパスワード・セキュリティ・メカニズムを設定します。

```

import java.sql.*;           // JDBC base
import com.ibm.db2.jcc.*;   // IBM Data Server Driver for JDBC
                             // and SQLJ implementation of JDBC
...
Properties properties = new Properties(); // Create a Properties object
properties.put("user", "dbadm");        // Set user ID for the connection
properties.put("password", "dbadm");    // Set password for the connection
properties.put("securityMechanism",
  new String("" + com.ibm.db2.jcc.DB2BaseDataSource.ENCRYPTED_PASSWORD_SECURITY +
  ""));
                             // Set security mechanism to
                             // user ID and encrypted password
String url = "jdbc:db2://mvs1.sj.ibm.com:5021/san_jose";
                             // Set URL for the data source
Connection con = DriverManager.getConnection(url, properties);
                             // Create the connection

```

**DataSource インターフェースの場合:** DataSource オブジェクトを作成してデプロイする場合、DataSource オブジェクトの作成後に、DataSource.setUser、DataSource.setPassword、および DataSource.setSecurityMechanism メソッドを呼び出して、ユーザー ID、パスワード、およびセキュリティーのメカニズムを設定することができます。例えば、次のようなコードを使用して、暗号化されたユーザー ID および暗号化されたパスワード・セキュリティー・メカニズムを設定します。

```

import java.sql.*;           // JDBC base
import com.ibm.db2.jcc.*;   // IBM Data Server Driver for JDBC
                             // and SQLJ implementation of JDBC
...
com.ibm.db2.jcc.DB2SimpleDataSource ds =
  new com.ibm.db2.jcc.DB2SimpleDataSource();
                             // Create the DataSource object
ds.setDriverType(4);        // Set the driver type
ds.setDatabaseName("san_jose"); // Set the location
ds.setServerName("mvs1.sj.ibm.com");
                             // Set the server name
ds.setPortNumber(5021);    // Set the port number
ds.setUser("db2adm");      // Set the user ID
ds.setPassword("db2adm");  // Set the password
ds.setSecurityMechanism(
  com.ibm.db2.jcc.DB2BaseDataSource.ENCRYPTED_PASSWORD_SECURITY);
                             // Set security mechanism to
                             // User ID and encrypted password

```

---

## IBM Data Server Driver for JDBC and SQLJ 使用時の Kerberos セキュリティー

Kerberos セキュリティー用の JDBC サポートは、IBM Data Server Driver for JDBC and SQLJ Type 4 接続でのみ使用可能です。

Kerberos セキュリティーの JDBC サポートを使用可能にするには、Software Development Kit (SDK) for Java の以下のコンポーネントも使用可能にする必要があります。

- Java Cryptography Extension
- Java Generic Security Service (JGSS)
- Java Authentication and Authorization Service (JAAS)

これらのコンポーネントを使用可能にする方法については、ご使用の SDK for Java の資料を参照してください。



接続に Kerberos セキュリティーを指定する方法には、以下の 3 つがあります。

- ユーザー ID とパスワードを使用する
- ユーザー ID もパスワードも使用しない
- 委任証明書を使用する

## ユーザー ID とパスワードを使用する Kerberos セキュリティー

この場合、Kerberos により、指定されたユーザー ID とパスワードが使用されてチケット許可チケット (TGT) が入手され、これによってデータベース・サーバーから認証を受けることができます。

user、password、kerberosServerPrincipal、および securityMechanism プロパティを設定する必要があります。kerberosServerPrincipal プロパティには、データベース・サーバーにより Kerberos 鍵配布センター (KDC) に登録されるプリンシパル名を指定します。

**DriverManager インターフェースの場合:** Properties オブジェクトに、user、password、kerberosServerPrincipal、および securityMechanism プロパティを設定し、Properties オブジェクトをパラメーターとして組み込んだ形式の getConnection メソッドを呼び出して、ユーザー ID、パスワード、Kerberos サーバーおよびセキュリティのメカニズムを設定します。例えば、次のようなコードを使用して、ユーザー ID およびパスワードで Kerberos セキュリティー・メカニズムを設定します。

```
import java.sql.*;           // JDBC base
import com.ibm.db2.jcc.*;   // IBM Data Server Driver for JDBC
                             // and SQLJ implementation of JDBC
...
Properties properties = new Properties(); // Create a Properties object
properties.put("user", "db2adm");      // Set user ID for the connection
properties.put("password", "db2adm");  // Set password for the connection
properties.put("kerberosServerPrincipal",
    "sample/srvlsj.ibm.com@SRVLSJ.SJ.IBM.COM");
                                     // Set the Kerberos server
properties.put("securityMechanism",
    new String(" +
    com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY + "));
                                     // Set security mechanism to
                                     // Kerberos
String url = "jdbc:db2://mvs1.sj.ibm.com:5021/san_jose";
                                     // Set URL for the data source
Connection con = DriverManager.getConnection(url, properties);
                                     // Create the connection
```

**DataSource インターフェースの場合:** DataSource オブジェクトを作成してデプロイする場合、DataSource オブジェクトの作成後に、

DataSource.setKerberosServerPrincipal および DataSource.setSecurityMechanism メソッドを呼び出して、Kerberos サーバーおよびセキュリティのメカニズムを設定します。以下に例を示します。

```
import java.sql.*;           // JDBC base
import com.ibm.db2.jcc.*;   // IBM Data Server Driver for JDBC
                             // and SQLJ implementation of JDBC
...
com.ibm.db2.jcc.DB2SimpleDataSource db2ds =
    new com.ibm.db2.jcc.DB2SimpleDataSource();
                                     // Create the DataSource object
db2ds.setDriverType(4);             // Set the driver type
db2ds.setDatabaseName("san_jose"); // Set the location
```



```

db2ds.setUser("db2adm");           // Set the user
db2ds.setPassword("db2adm");       // Set the password
db2ds.setServerName("mvs1.sj.ibm.com"); // Set the server name
db2ds.setPortNumber(5021);         // Set the port number
db2ds.setKerberosServerPrincipal(
    "sample/srvlsj.ibm.com@SRVLSJ.SJ.IBM.COM"); // Set the Kerberos server
db2ds.setSecurityMechanism(
    com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY); // Set security mechanism to
// Kerberos

```

## ユーザー ID もパスワードを使用しない Kerberos セキュリティー

この場合、Kerberos のデフォルト証明書キャッシュにチケット許可チケット (TGT) が含まれている必要があります、これによってデータベース・サーバーから認証を受けることができます。

kerberosServerPrincipal および securityMechanism プロパティーを設定する必要があります。

**DriverManager インターフェースの場合:** Properties オブジェクトに、kerberosServerPrincipal および securityMechanism プロパティーを設定し、Properties オブジェクトをパラメーターとして組み込んだ形式の getConnection メソッドを呼び出して、Kerberos サーバーおよびセキュリティーのメカニズムを設定します。例えば、次のようなコードを使用して、ユーザー ID およびパスワードなしで Kerberos セキュリティー・メカニズムを設定します。

```

import java.sql.*;           // JDBC base
import com.ibm.db2.jcc.*;    // IBM Data Server Driver for JDBC
// and SQLJ implementation of JDBC
...
Properties properties = new Properties(); // Create a Properties object
properties.put("kerberosServerPrincipal",
    "sample/srvlsj.ibm.com@SRVLSJ.SJ.IBM.COM"); // Set the Kerberos server
properties.put("securityMechanism",
    new String("" +
    com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY + "")); // Set security mechanism to
// Kerberos
String url = "jdbc:db2://mvs1.sj.ibm.com:5021/san_jose"; // Set URL for the data source
Connection con = DriverManager.getConnection(url, properties); // Create the connection

```

**DataSource インターフェースの場合:** DataSource オブジェクトを作成してデプロイする場合、DataSource オブジェクトの作成後に、DataSource.setKerberosServerPrincipal および DataSource.setSecurityMechanism メソッドを呼び出して、Kerberos サーバーおよびセキュリティーのメカニズムを設定します。以下に例を示します。

```

import java.sql.*;           // JDBC base
import com.ibm.db2.jcc.*;    // IBM Data Server Driver for JDBC
// and SQLJ implementation of JDBC
...
DB2SimpleDataSource db2ds =
    new com.ibm.db2.jcc.DB2SimpleDataSource(); // Create the DataSource object
db2ds.setDriverType(4); // Set the driver type

```

```

db2ds.setDatabaseName("san_jose");           // Set the location
db2ds.setServerName("mvs1.sj.ibm.com");      // Set the server name
db2ds.setPortNumber(5021);                  // Set the port number
db2ds.setKerberosServerPrincipal(
    "sample/srv1sj.ibm.com@SRVLSJ.SJ.IBM.COM"); // Set the Kerberos server
db2ds.setSecurityMechanism(
    com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY); // Set security mechanism to
                                                            // Kerberos

```

## 別のプリンシパルからの委任証明書を使用した Kerberos セキュリティー

この場合、別のプリンシパルから受け渡される委任証明書を使用して、データベース・サーバーから認証を受けます。

kerberosServerPrincipal、gssCredential、および securityMechanism プロパティを設定する必要があります。

**DriverManager インターフェースの場合:** Properties オブジェクトに、kerberosServerPrincipal および securityMechanism プロパティを設定して、Kerberos サーバー、委任証明書、およびセキュリティのメカニズムを設定します。次に、Properties オブジェクトをパラメーターとして組み込んだ形式の getConnection メソッドを呼び出します。例えば、次のようなコードを使用して、ユーザー ID およびパスワードなしで Kerberos セキュリティー・メカニズムを設定します。

```

import java.sql.*;           // JDBC base
import com.ibm.db2.jcc.*;    // IBM Data Server Driver for JDBC
                              // and SQLJ implementation of JDBC
...
Properties properties = new Properties(); // Create a Properties object
properties.put("kerberosServerPrincipal",
    "sample/srv1sj.ibm.com@SRVLSJ.SJ.IBM.COM"); // Set the Kerberos server
properties.put("gssCredential",delegatedCredential); // Set the delegated credential
properties.put("securityMechanism",
    new String("" +
        com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY + "")); // Set security mechanism to
                                                                    // Kerberos
String url = "jdbc:db2://mvs1.sj.ibm.com:5021/san_jose"; // Set URL for the data source
Connection con = DriverManager.getConnection(url, properties); // Create the connection

```

**DataSource インターフェースの場合:** DataSource オブジェクトを作成してデプロイする場合、DataSource オブジェクトの作成後に、DataSource.setKerberosServerPrincipal、DataSource.setGssCredential、および DataSource.setSecurityMechanism メソッドを呼び出して、Kerberos サーバー、委任証明書、およびセキュリティのメカニズムを設定します。以下に例を示します。

```

DB2SimpleDataSource db2ds = new com.ibm.db2.jcc.DB2SimpleDataSource(); // Create the DataSource object
db2ds.setDriverType(4); // Set the driver type
db2ds.setDatabaseName("san_jose"); // Set the location
db2ds.setServerName("mvs1.sj.ibm.com"); // Set the server name
db2ds.setPortNumber(5021); // Set the port number

```

```

db2ds.setKerberosServerPrincipal(
    "sample/srv1sj.ibm.com@SRVLSJ.SJ.IBM.COM");
// Set the Kerberos server
db2ds.setGssCredential(delegatedCredential);
// Set the delegated credential
db2ds.setSecurityMechanism(
    com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY);
// Set security mechanism to
// Kerberos

```

---

## IBM Data Server Driver for JDBC and SQLJ のセキュリティー・プラグインのサポート

独自の認証メカニズムをロード可能なライブラリーまたはプラグインの形式で作成することができます。この認証メカニズムは、DB2 Database for Linux, UNIX, and Windows でユーザー認証を実行するためにロードされます。Java でのセキュリティー・プラグインの開発をサポートするために、IBM Data Server Driver for JDBC and SQLJ ではセキュリティー・プラグインのサポートが提供されます。

IBM Data Server Driver for JDBC and SQLJ のセキュリティー・プラグインのサポートは、DB2 Database for Linux, UNIX, and Windows サーバーに対する IBM Data Server Driver for JDBC and SQLJ Type 4 接続でのみ使用可能です。

プラグイン・セキュリティーを使用するには、セキュリティー・プラグインがクライアント上に必要で、またサーバー上には別のプラグインが必要です。

セキュリティー・プラグインには、以下のものを組み込む必要があります。

- 抽象クラス `com.ibm.db2.jcc.DB2JCCPlugin` を拡張するクラス

抽象クラス `com.ibm.db2.jcc.DB2JCCPlugin` に IBM Data Server Driver for JDBC and SQLJ が提供されます。

- クラス `com.ibm.db2.jcc.DB2JCCPlugin` 内の `com.ibm.db2.jcc.DB2JCCPlugin.getTicket` メソッド

このメソッドでは、ユーザーの Kerberos チケットが検索され、セキュリティー・コンテキスト情報がバイト配列で返されます。このバイト配列の情報は、IBM Data Server Driver for JDBC and SQLJ による DB2 データベース・サーバーへのアクセスに使用されます。

- `org.ietf.jgss.GSSContext` および `org.ietf.jgss.GSSCredential` インターフェースで定義されるいくつかのメソッドのインプリメンテーション

これらのメソッドのインプリメンテーションは、Generic Security Service Application Program Interface, Version 2 (IETF RFC2743) および Generic Security Service API Version 2: Java-Bindings (IETF RFC2853) 仕様に準拠している必要があります。プラグインでは、以下の各メソッドをインプリメントして呼び出す必要があります。

### **GSSContext.dispose**

コンテキスト・オブジェクトに保管されているシステム・リソースおよび暗号情報を解放し、コンテキストを無効にします。

### **GSSContext.getCredDelegState**

信用証明情報の委任がコンテキストで使用可能かどうかを判別します。

### **GSSContext.getMutualAuthState**

相互認証がコンテキストで使用可能かどうかを判別します。

### **GSSContext.initSecContext**

コンテキスト作成フェーズを開始し、ピアの `acceptSecContext` メソッドによって生成されるすべてのトークンを処理します。

### **GSSContext.requestCredDeleg**

コンテキストが確立される際に、イニシエーターの信用証明情報がアクセプターに委任されることを要求します。

### **GSSContext.requestMutualAuth**

コンテキストが確立される際に、相互認証を要求します。

### **GSSCredential.dispose**

`GSSCredential` オブジェクトに含まれる機密情報を解放します。

Java セキュリティー・プラグインの作成に役立つ、以下の 2 つの Java プラグイン・サンプルが `sqllib/samples/java/jdbc` に提供されています。

### **JCCSimpleGSSPlugin.java**

サーバー用の GSS-API プラグインのインプリメンテーション。これにより、ユーザー ID とパスワードのチェックが実行されます。このサンプルは、C 言語のサンプル・プログラム `gssapi_simple.c` の Java バージョンです。

### **JCCKerberosPlugin.java**

クライアント用の Kerberos セキュリティー・プラグイン。このサンプルは、C 言語のサンプル・プログラム `IBMkrb5.c` の Java バージョンです。

アプリケーション・プログラムで JDBC プラグイン・セキュリティが使用されて接続が取得される場合、以下の `Connection` または `DataSource` プロパティーを設定する必要があります。

表 26. Java セキュリティー・プラグインを使用するための `Connection` または `DataSource` プロパティーの設定

プロパティー	設定
<code>com.ibm.db2.jcc.DB2BaseDataSource.user</code>	<code>Connection</code> を取得する際のユーザー ID
<code>com.ibm.db2.jcc.DB2BaseDataSource.password</code>	ユーザー ID のパスワード
<code>com.ibm.db2.jcc.DB2BaseDataSource.securityMechanism</code>	<code>com.ibm.db2.jcc.DB2BaseDataSource.PLUGIN_SECURITY</code>
<code>com.ibm.db2.jcc.DB2BaseDataSource.pluginName</code>	サーバー・サイド・セキュリティ・プラグインのプラグイン・モジュールの名前
<code>com.ibm.db2.jcc.DB2BaseDataSource.plugin</code>	クライアント・サイド・セキュリティ・プラグインのプラグイン・オブジェクト

例: 下のコードでは、GSS-API プラグイン・セキュリティを使用する接続のプロパティーが設定されます。この接続では、クライアント・サイドでサンプル・プラグインの `JCCSimpleGSSPlugin` が使用され、サーバー・サイドでサンプル・プラグインの `gssapi_simple` が使用されます。

```
java.util.Properties properties = new java.util.Properties();
properties.put("user", "db2admin");
properties.put("password", "admindb2");
properties.put("pluginName", "gssapi_simple");
properties.put("securityMechanism",
    new String(""+com.ibm.db2.jcc.DB2BaseDataSource.PLUGIN_SECURITY+""));
```

```
com.ibm.db2.jcc.DB2JCCPlugin plugin =
    new com.ibm.db2.jcc.samples.plugins.JCCSimpleGSSPlugin();
properties.put("plugin", plugin);
Connection con = java.sql.DriverManager.getConnection(url,
    properties);
```

---

## IBM Data Server Driver for JDBC and SQLJ のトラステッド・コンテキストのサポート

IBM Data Server Driver for JDBC and SQLJ では、Java プログラム内でのトラステッド接続の確立および使用を可能にするメソッドが提供されます。

トラステッド接続は、DB2 Database for Linux, UNIX, and Windows と DB2 for z/OS 用の IBM Data Server Driver for JDBC and SQLJ Type 4 接続、および DB2 for z/OS 用の IBM Data Server Driver for JDBC and SQLJ Type 2 接続でサポートされます。

3 層アプリケーション・モデルは、データベース・サーバー、WebSphere Application Server などのミドルウェア・サーバー、およびエンド・ユーザーで構成されています。このモデルでは、ミドルウェア・サーバーがエンド・ユーザーに代わってデータベース・サーバーにアクセスします。トラステッド・コンテキストのサポートにより、ミドルウェア・サーバーがエンド・ユーザーに代わって任意のデータベース要求を実行する際に、エンド・ユーザーのデータベース ID とデータベース特権が使用されます。

トラステッド・コンテキストとは、データベース管理者が定義し、システムの許可 ID およびトラスト属性のセットが含まれたオブジェクトのことです。現行の DB2 データベース・サーバーでは、データベース接続がサポートされているコンテキストの唯一のタイプです。トラスト属性により、接続がトラステッド接続と見なされるために必要な接続の特性のセットが識別されます。データベース接続とトラステッド・コンテキストの関係は、データベース・サーバーへの接続が最初に作成されたときに確立され、この関係はデータベース接続が存続する間持続します。

トラステッド・コンテキストが定義され、DB2 データベース・サーバーへの初期トラステッド接続が確立されると、ミドルウェア・サーバーでは、データベース・サーバーで新規ユーザーの再認証を行うことなく、このデータベース接続を異なるユーザーの下で使用することができます。

セキュリティー・ブリーチ (抜け穴) へのぜい弱性を避けるために、これらのトラステッド・メソッドを使用するアプリケーション・サーバーでは、非トラステッド接続メソッドを使用しないでください。

DB2ConnectionPoolDataSource クラスでは `getDB2TrustedPooledConnection` メソッドのいくつかのバージョンが提供され、また DB2XADataSource クラスでは `getDB2TrustedXAConnection` メソッドのいくつかのバージョンが提供され、これらによってアプリケーション・サーバーでの初期トラステッド接続の確立が可能になります。受け渡す接続プロパティーのタイプ、および Kerberos セキュリティーを使用するかどうかに基づいて、メソッドを選択します。アプリケーション・サーバーでこれらのメソッドの 1 つが呼び出されると、IBM Data Server Driver for JDBC and SQLJ により以下の 2 つの要素を持つ `Object[]` 配列が返されます。

- 最初の要素には、初期接続用の接続インスタンスが含まれています。

- 2 番目のエレメントには、接続インスタンス用の固有の cookie が含まれています。この cookie は JDBC ドライバーによって生成され、その後の接続の再利用時に認証用に使用されます。

DB2PooledConnection クラスでは getDB2Connection メソッドのいくつかのバージョンが提供され、また DB2Connection クラスでは reuseDB2Connection メソッドのいくつかのバージョンが提供されます。これらによって、新規ユーザーの代わりにアプリケーション・サーバーで既存のトラステッド接続の再利用が可能になります。アプリケーション・サーバーでは、これらのメソッドの 1 つが使用されて以下の項目が新規ユーザーに受け渡されます。

- 初期接続からの cookie
- 再利用される接続用の新規接続プロパティ

JDBC ドライバーは、提供された cookie が基礎となるトラステッド物理接続の cookie と一致することを調べて、接続要求がそのトラステッド物理接続を確立したアプリケーション・サーバーから発信されていることを確認します。cookie が一致する場合、新規プロパティを持つ接続が使用可能になり、この新規ユーザーが即時使用することができます。

**例:** 初期トラステッド接続の取得:

```
// Create a DB2ConnectionPoolDataSource instance
com.ibm.db2.jcc.DB2ConnectionPoolDataSource dataSource =
    new com.ibm.db2.jcc.DB2ConnectionPoolDataSource();
// Set properties for this instance
dataSource.setDatabaseName ("STLEC1");
dataSource.setServerName ("v7ec167.svl.ibm.com");
dataSource.setDriverType (4);
dataSource.setPortNumber(446);
java.util.Properties properties = new java.util.Properties();
// Set other properties using
// properties.put("property", "value");
// Supply the user ID and password for the connection
String user = "user";
String password = "password";
// Call getDB2TrustedPooledConnection to get the trusted connection
// instance and the cookie for the connection
Object[] objects = dataSource.getDB2TrustedPooledConnection(
    user,password, properties);
```

**例:** 既存のトラステッド接続の再利用:

```
// The first item that was obtained from the previous getDB2TrustedPooledConnection
// call is a connection object. Cast it to a PooledConnection object.
javax.sql.PooledConnection pooledCon =
    (javax.sql.PooledConnection)objects[0];
properties = new java.util.Properties();
// Set new properties for the reused object using
// properties.put("property", "value");
// The second item that was obtained from the previous getDB2TrustedPooledConnection
// call is the cookie for the connection. Cast it as a byte array.
byte[] cookie = ((byte[])objects[1]);
// Supply the user ID for the new connection.
String newuser = "newuser";
// Supply the name of a mapping service that maps a workstation user
// ID to a z/OS RACF ID
String userRegistry = "registry";
// Do not supply any security token data to be traced.
byte[] userSecTkn = null;
// Do not supply a previous user ID.
String originalUser = null;
```



```
// Call getDB2Connection to get the connection object for the new
// user.
java.sql.Connection con =
    ((com.ibm.db2.jcc.DB2PooledConnection)pooledCon).getDB2Connection(
        cookie,newuser,password,userRegistry,userSecTkn,originalUser,properties);
```

---

## SSL 用の IBM Data Server Driver for JDBC and SQLJ サポート

IBM Data Server Driver for JDBC and SQLJ では、Java Secure Socket Extension (JSSE) を介した Security Socket Layer (SSL) がサポートされます。

DB2 for z/OS バージョン 9.1 以降、または DB2 Database for Linux, UNIX, and Windows バージョン 9.1 フィックスパック 2 以降用の IBM Data Server Driver for JDBC and SQLJ Type 4 接続を使用している場合は、Java アプリケーション内で SSL サポートを使用することができます。

IBM Data Server Driver for JDBC and SQLJ では、DB2BaseDataSource.sslConnection プロパティを true に設定することで SSL サポートを有効にすることができます。これにより、IBM Data Server Driver for JDBC and SQLJ は SSL ソケットを使用して DB2 データベース・サーバーに接続できます。

クライアント・アプリケーションで SSL を使用する前に、クライアント・アプリケーションのユーザーは以下を行う必要があります。

- ホスト・システムのセキュリティ管理者から証明書を取得し、Java keytool ユーティリティを使用してその証明書をトラストストアにインポートします。

例えば、DB2 データベース・サーバーから証明書を取得して、jcc.cacert というファイルに保管したとします。ファイル jcc.cacert から証明書を読み取り、それを cacerts というトラストストアに保管するには、以下の keytool ユーティリティ・ステートメントを発行します。

```
keytool -import -file jcc.cacert -keystore cacerts
```

- 以下の Java システム・プロパティを設定します。

**javax.net.ssl.trustStore**

トラストストアを指定します。

**javax.net.ssl.trustStorePassword**

トラストストア・パスワードを指定します。

IBM Data Server Driver for JDBC and SQLJ は IBM FIPS に準拠した SSL 用のプロバイダー、または SunJSSE プロバイダーを使用できます。IBM Data Server Driver for JDBC and SQLJ を FIPS に準拠したモードで使用するには、FIPS 認証プロバイダー IBMJSSEFIPSProvider および IBMJCEFIPS を使用するか、あるいは IBMJSSE2 プロバイダーで FIPS モードを有効にします。

IBMJSSEFIPSProvider および IBMJCEFIPS プロバイダーを java.security ファイルで使用するには、以下の行を追加します。

```
security.provider.n=com.ibm.fips.jsse.IBMJSSEFIPSProvider
security.provider.n=com.ibm.crypto.fips.provider.IBMJCEFIPS
```

```
ssl.SocketFactory.provider=com.ibm.fips.jsse.JSSESocketFactory
ssl.ServerSocketFactory.provider= com.ibm.fips.jsse.JSSEServerSocketFactory
```



$n$  は設定順序です。以前に指定されたプロバイダーの優先順序は、`java.security` ファイル内の非 FIPS プロバイダーの優先順序よりも高く ( $n$  の値がより小さい) なければなりません。

IBMJSSE2 プロバイダーで FIPS モードを有効にするには、以下のステップに従います。

1. IBMJSSE2 FIPS システム・プロパティを以下のように設定して、FIPS モードを有効にします。

```
com.ibm.jsse2.JSSEFIPS=true
```

2. セキュリティー・プロパティを以下のように設定して、すべての JSSE コードが IBMJSSE2 プロバイダーを使用するようにします。

```
ssl.SocketFactory.provider=com.ibm.jsse2.SSLSocketFactoryImpl  
ssl.SocketFactory.provider=com.ibm.jsse2.SSLServerSocketFactoryImpl
```

3. IBMJCEFIPS 暗号プロバイダーを追加します。

これを行うには、`com.ibm.crypto.fips.provider.IBMJCEFIPS` をプロバイダー・リストの IBMJCE プロバイダーの前に追加します。IBMJCE プロバイダーを除去しないでください。IBMJCE プロバイダーは鍵ストアのサポートに必要です。

IBMJSSE2 プロバイダーを FIPS モードで実行するには、アプリケーションに変更を加える必要はありません。

#### 制約事項:

- 非 FIPS モードの JSSE アプリケーションは FIPS モードの JVM では実行できません。
- IBMJSSE2 プロバイダーが FIPS モードで実行する場合、ハードウェアの暗号化を使用することはできません。

---

## IBM Data Server Driver for JDBC and SQLJ 使用時の SQLJ アプリケーション準備のセキュリティー

SQLJ アプリケーションの準備中にセキュリティーを提供する 2 つの方法があります。ユーザーにアプリケーションのカスタマイズのみ許可するという方法と、カスタマイズ中に特定の表のセットに対するアクセスを制限するという方法です。

### ユーザーにカスタマイズのみ許可する

一群のユーザーに SQLJ アプリケーションのカスタマイズを許可し、それらのアプリケーションのバインドまたは実行を許可しないようにするには、以下のいずれかの手法を使用できます。

- **カスタマイズ専用のデータベース・システムを作成します (推奨ソリューション):** 以下のステップを実行してください。
  1. 新規のデータベース・マネージャー・インスタンスを作成します。これはカスタマイズ専用システムです。
  2. カスタマイズ専用システムでは、SQLJ アプリケーションがアクセスするすべての表およびビューを定義します。表またはビューの定義は、アプリケーションがバインドされて実行するデータベース・マネージャー・インスタンス (「バインドおよび実行」システム) 上の定義と同じでなければなりません。表

またはビューに対して DESCRIBE ステートメントを実行した場合、カスタマイズ専用システムと「バインドおよび実行」システムでは同じ結果にならないかもしれません。

3. カスタマイズ専用システムでは、必要な表またはビュー特権を、SQLJ アプリケーションをカスタマイズするユーザーに付与します。
  4. カスタマイズ専用システムでは、ユーザーは `-compile=true` オプションを指定して `sqlj` コマンドを実行し、プログラム用の Java バイト・コードおよびシリアルライズド・プロファイルを作成します。次に、`-automaticbind NO` オプションを指定して `db2sqljcustomize` コマンドを実行し、カスタマイズ済みシリアルライズド・プロファイルを作成します。
  5. `java` バイトコード・ファイルおよびカスタマイズ済みシリアルライズド・プロファイルを「バインドおよび実行」システムにコピーします。
  6. 「バインドおよび実行」システムでパッケージをバインドする権限を持つユーザーは、カスタマイズ専用システムからコピーされたカスタマイズ済みシリアルライズド・プロファイルに対して `db2sqljbind` コマンドを実行します。
- **ストアード・プロシージャを使用してカスタマイズを行います:** エンド・ユーザーのためにシリアルライズド・プロファイルをカスタマイズし、SQLJ アプリケーションのパッケージをバインドする Java ストアード・プロシージャを作成します。この Java ストアード・プロシージャは、`DYNAMICRULES` オプションのいずれかを指定してバインドされた JDBC ドライバー・パッケージを使用する必要があります。このオプションを指定すると、動的 SQL はエンド・ユーザーの許可 ID とは別のユーザー ID で実行されます。例えば、`DYNAMICRULES` オプションの `DEFINEBIND` または `DEFINERUN` を使用して、Java ストアード・プロシージャの作成者の許可 ID で動的 SQL を実行する場合があります。その場合、ストアード・プロシージャでの `EXECUTE` 権限を、SQLJ カスタマイズを実行する必要があるユーザーに付与する必要があります。

ストアード・プロシージャは以下の処理を実行します。

1. コンパイルされた SQLJ プログラムおよびシリアルライズド・プロファイルを BLOB 入力パラメーターで受け取ります。
  2. 入力パラメーターをそのファイル・システムにコピーします。
  3. `db2sqljcustomize` を実行して、シリアルライズド・プロファイルをカスタマイズし、SQLJ プログラム用のパッケージをバインドします。
  4. カスタマイズ済みのシリアルライズド・プロファイルを出力パラメーターで戻します。
- **スタンドアロン・プログラムを使用してカスタマイズを行います:** この手法では、エンド・ユーザーのためにシリアルライズド・プロファイルをカスタマイズし、SQLJ アプリケーションのパッケージをバインドする Java ストアード・プロシージャと同じステップを実行するプログラムを作成することを伴います。ただし、プログラムをストアード・プロシージャとして実行する代わりに、プログラムをライブラリー・サーバーの下でスタンドアロン・プログラムとして実行します。

## カスタマイズ中の表アクセスの制限

シリアルライズド・プロファイルをカスタマイズする場合、オンライン検査を実行して、プログラムがアクセスする表列のデータ・タイプおよび長さに関するアプリケ

ーション・プログラム情報を提供する必要があります。デフォルトでは、カスタマイズにはオンライン検査が含まれています。

オンライン検査では、シリアライズド・プロファイルをカスタマイズするユーザーが、SQLJ プログラム内の SQL ステートメントに対してPREPARE および DESCRIBE ステートメントを実行する許可を持っている必要があります。その許可には、SQL ステートメントでアクセスされる表およびビューに対する SELECT 特権が含まれます。SQL ステートメントに未修飾の表名が含まれる場合、オンライン検査中に使用される修飾子は、db2sqljcustomize -qualifier パラメーターの値になります。そのため、SQLJ アプリケーション内の未修飾名を持つ表およびビューをオンライン検査する場合、-qualifier パラメーターの値に一致する修飾子を持つ表およびビューに対してのみ SELECT 特権を付与できます。

---

## 第 6 章 DB2 JDBC Type 2 ドライバー使用時のセキュリティ

Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバー (DB2 JDBC Type 2 ドライバー) では、ユーザー ID およびパスワード・セキュリティがサポートされます。

ユーザー ID とパスワードを設定するか、あるいはそれらを両方とも設定しないようにしなければなりません。ユーザー ID とパスワードを設定しない場合、ドライバーは、オペレーティング・システムに現在ログオンしているユーザーのユーザー ID とパスワードを使用します。

JDBC 接続で、ユーザー ID およびパスワード・セキュリティを指定する場合、以下のいずれかの技法を使用します。

**DriverManager インターフェースの場合:** DriverManager.getConnection 呼び出しで、ユーザー ID およびパスワードを直接指定することができます。以下に例を示します。

```
import java.sql.*;          // JDBC base
...
String id = "db2adm";      // Set user ID
String pw = "db2adm";     // Set password
String url = "jdbc:db2:toronto";
                          // Set URL for the data source
Connection con = DriverManager.getConnection(url, id, pw);
                          // Create connection
```

別の方法として、Properties オブジェクトに user および password プロパティを設定し、Properties オブジェクトをパラメーターとして組み込んだフォームの getConnection メソッドを呼び出して、ユーザー ID およびパスワードを設定できます。以下に例を示します。

```
import java.sql.*;          // JDBC base
import COM.ibm.db2.jdbc.*; // DB2 JDBC Type 2 driver
                          // implementation of JDBC
...
Properties properties = new java.util.Properties();
                          // Create Properties object
properties.put("user", "db2adm"); // Set user ID for the connection
properties.put("password", "db2adm"); // Set password for the connection
String url = "jdbc:db2:toronto";
                          // Set URL for the data source
Connection con = DriverManager.getConnection(url, properties);
                          // Create connection
```

**DataSource インターフェースの場合:** DataSource.getConnection 呼び出しで、ユーザー ID およびパスワードを直接指定することができます。以下に例を示します。

```
import java.sql.*;          // JDBC base
import COM.ibm.db2.jdbc.*; // DB2 JDBC Type 2 driver
                          // implementation of JDBC
...
Context ctx=new InitialContext(); // Create context for JNDI
DataSource ds=(DataSource)ctx.lookup("jdbc/sampledb");
                          // Get DataSource object
String id = "db2adm";      // Set user ID
```

```
String pw = "db2adm"; // Set password
Connection con = ds.getConnection(id, pw); // Create connection
```

DataSource オブジェクトを作成してデプロイする場合、 DataSource オブジェクトの作成後に、 DataSource.setUser および DataSource.setPassword メソッドを呼び出して、ユーザー ID およびパスワードを設定できます。以下に例を示します。

```
import java.sql.*; // JDBC base
import COM.ibm.db2.jdbc.*; // DB2 JDBC Type 2 driver
// implementation of JDBC
...
DB2DataSource db2ds = new DB2DataSource(); // Create DataSource object
db2ds.setDatabaseName("toronto"); // Set location
db2ds.setUser("db2adm"); // Set user ID
db2ds.setPassword("db2adm"); // Set password
```

---

## 第 7 章 Java データベース・アプリケーションの構築

JDBC と SQLJ のデータベース・アプリケーションは、手動で構築できます。別の方法として、Java makefile を使用して JDBC アプリケーションを構築したり、DB2 Database for Linux, UNIX, and Windows に用意されている bldsqlj 構築ファイルを使用して SQLJ アプリケーションを構築したりすることも可能です。

---

### JDBC アプレットの構築

JDBC アプリケーションを構築するには、Java makefile を使用するか、または手動で javac コマンドを実行することができます。

以下のステップは、Applt.java サンプル JDBC アプレットを構築して実行する方法を示しています。

1. 以下のコマンドで、Applt.java をコンパイルして、ファイル Applt.class を作成します。

```
javac Applt.java
```

2. 作業ディレクトリーが、Web ブラウザーまたは (使用する場合は) Java アプレット・ビューアーからアクセス可能であることを確認します。ディレクトリーがアクセスできない場合は、以下のファイルをアクセス可能なディレクトリーにコピーします。
  - Applt.html
  - Applt.class
3. sqllib¥java¥db2jcc.jar (Windows の場合) または sqllib/java/db2jcc.jar (UNIX の場合) を、Applt.class および Applt.html と同じディレクトリーにコピーします。

JDBC 4.0 の機能を使用する場合は、db2jcc.jar ではなく db2jcc4.jar をコピーします。

4. IBM Data Server Driver for JDBC and SQLJ を使用する場合、ファイルに記載されている手順に従って、Applt.html ファイルを変更してこのドライバーで接続します。TCP/IP ポート番号には、データベース・ポート番号 50000 を使用する必要があります。
5. このアプレットを実行するには、Web サーバーが DB2 マシン上にインストール済みで実行中であることを確認するか、または以下のコマンドをクライアント・マシンの作業ディレクトリーで入力して、SDK for Java に付属しているアプレット・ビューアーを使用することができます。

```
appletviewer Applt.html
```

---

### JDBC アプリケーションの構築

JDBC アプリケーションを構築するには、Java makefile を使用するか、または手動で javac コマンドを実行することができます。

以下のステップは、DbInfo サンプル JDBC アプリケーションを構築して実行する方法を示しています。

1. 以下のコマンドで、DbInfo.java をコンパイルして、ファイル DbInfo.class を作成します。

```
javac DbInfo.java
```

2. Java アプリケーションを UNIX 上の 64 ビットの DB2 インスタンスで実行しているが、Software Development Kit for Java が 32 ビットである場合、アプリケーションを実行する前に DB2 ライブラリー・パスを変更する必要があります。例えば、AIX 上では以下のようにします。

- bash または Korn シェルを使用している場合:

```
export LIBPATH=$HOME/sql1lib/lib32
```

- C シェルを使用している場合:

```
setenv LIBPATH $HOME/sql1lib/lib32
```

3. Java インタープリターをアプリケーション上で、以下のコマンドで実行します。

```
java DbInfo
```

---

## JDBC ルーチンの構築

JDBC ルーチンを構築するには、Java makefile または javac コマンドを使用できます。これらのルーチンを構築した後で、それらをカタログする必要があります。

以下のステップは、これらのルーチンを構築して実行する方法を示しています。

- SpServer のサンプル JDBC ストアード・プロシージャ
- UDFsrv のサンプル・ユーザー定義関数。これには SQL ステートメントが入っていません。
- UDFsqlsv のサンプル・ユーザー定義関数。これには SQL ステートメントが入っています。
- SpServer.java ストアード・プロシージャをサーバー上で構築して実行するには、コマンド行から以下のようにします。

1. 以下のコマンドで、SpServer.java をコンパイルして、ファイル SpServer.class を作成します。

```
javac SpServer.java
```

2. SpServer.class を、Windows オペレーティング・システム上では sql1lib¥function ディレクトリーに、UNIX 上では sql1lib/function ディレクトリーにコピーします。
3. サーバーで spcat スクリプトを実行してルーチンをカタログします。spcat スクリプトは、サンプル・データベースに接続し、ルーチンが既にカタログ済みであれば SpDrop.db2 を呼び出してルーチンをアンカタログし、次に SpCreate.db2 を呼び出してそのルーチンをカタログし、そして最後にデータベースへの接続を切断します。また、SpDrop.db2 スクリプトと SpCreate.db2 スクリプトは、個別に実行することもできます。
4. データベースを一度停止してから再始動し、新しいクラス・ファイルが認識されるようにします。必要であれば、クラス・ファイルのファイル・モードを「読み取り」に設定して、fenced ユーザーが読み取ることができるようにします。



5. SpClient クライアント・アプリケーションをコンパイルして実行し、ストアード・プロシージャ・クラスにアクセスします。
- UDFsrv.java ユーザー定義関数プログラム (SQL ステートメントがないユーザー定義関数) をサーバー上で構築して実行するには、コマンド行から以下のようになります。

1. 以下のコマンドで、UDFsrv.java をコンパイルして、ファイル UDFsrv.class を作成します。

```
javac UDFsrv.java
```

2. UDFsrv.class を、Windows オペレーティング・システム上では sqllib¥function ディレクトリーに、UNIX 上では sqllib/function ディレクトリーにコピーします。
3. UDFsrv を呼び出すクライアント・プログラムをコンパイルして実行します。

UDFsrv ライブラリーにアクセスするには、UDFcli.java JDBC アプリケーションまたは UDFcli.sqlj SQLJ クライアント・アプリケーションを使用できます。どちらのバージョンのクライアント・プログラムにも、ユーザー定義関数をデータベースで登録するために使用する CREATE FUNCTION SQL ステートメントと、ユーザー定義関数を使用する SQL ステートメントが含まれています。

- UDFsqlsv.java ユーザー定義関数プログラム (SQL ステートメントを含むユーザー定義関数) をサーバー上で構築して実行するには、コマンド行から以下のようになります。

1. 以下のコマンドで、UDFsqlsv.java をコンパイルして、ファイル UDFsqlsv.class を作成します。

```
javac UDFsqlsv.java
```

2. UDFsqlsv.class を、Windows オペレーティング・システム上では sqllib¥function ディレクトリーに、UNIX 上では sqllib/function ディレクトリーにコピーします。
3. UDFsqlsv を呼び出すクライアント・プログラムをコンパイルして実行します。

UDFsqlsv ライブラリーにアクセスするには、UDFsqlcl.java JDBC アプリケーションを使用できます。クライアント・プログラムには、ユーザー定義関数をデータベースで登録するために使用する CREATE FUNCTION SQL ステートメントと、ユーザー定義関数を使用する SQL ステートメントが含まれています。

---

## SQLJ アプレットの構築

SQLJ アプレットを構築するには、Java makefile または bldsqlj ビルド・ファイルを使用できます。

以下のステップは、App1t サンプル SQLJ アプレットを構築して実行する方法を示しています。これらのステップは、ビルド・ファイル bldsqlj (UNIX) または bldsqlj.bat (Windows) を使用します。これには SQLJ アプレットまたはアプリケーションのいずれかを構築するためのコマンドが含まれています。

ビルド・ファイルは最大で 6 つのパラメーターを取ります。これは UNIX 上では \$1、\$2、\$3、\$4、\$5、および \$6、Windows 上では %1、%2、%3、%4、%5、および %6 です。最初のパラメーターはプログラムの名前を指定します。2 番目のパラメーターはデータベース・インスタンスのユーザー ID を指定し、3 番目のパラメーターはパスワードを指定します。4 番目のパラメーターはサーバー名を指定します。5 番目のパラメーターはポート番号を指定します。6 番目のパラメーターはデータベース名を指定します。最初のパラメーター (プログラム名) を除く他のすべてについては、デフォルト値を使用できます。デフォルトのパラメーター値の使用について詳しくは、ビルド・ファイルを参照してください。

1. アプレットをこのコマンドで構築するには、以下のようにします。

```
bldsqlj Applt <userid> <password> <server_name> <port_number> <db_name>
```

2. 作業ディレクトリーが、Web ブラウザーまたは (使用する場合は) Java アプレット・ビューアーからアクセス可能であることを確認します。ディレクトリーがアクセスできない場合は、以下のファイルをアクセス可能なディレクトリーにコピーします。

- Applt.html
- Applt.class
- Applt\_Cursor1.class
- Applt\_Cursor2.class
- Applt\_SJProfileKeys.class
- Applt\_SJProfile0.ser

3. sqllib¥java¥db2jcc.jar (Windows の場合) または sqllib/java/db2jcc.jar (UNIX の場合) を、Applt.class および Applt.html と同じディレクトリーにコピーします。

JDBC 4.0 の機能を使用する場合は、db2jcc.jar ではなく db2jcc4.jar をコピーします。

4. IBM Data Server Driver for JDBC and SQLJ を使用する場合は、ファイルに記載されている手順に従って、Applt.html ファイルを変更してこのドライバで接続します。TCP/IP ポート番号には、データベース・ポート番号 50000 を使用する必要があります。
5. このアプレットを実行するには、Web サーバーが DB2 マシン上にインストール済みで実行中であることを確認するか、または以下のコマンドをクライアント・マシンの作業ディレクトリーで入力して、SDK for Java に付属しているアプレット・ビューアーを使用することができます。

```
appletviewer Applt.html
```

---

## SQLJ アプリケーションの構築

SQLJ アプリケーションを構築するには、Java makefile または bldsqlj ビルド・ファイルを使用できます。

以下のステップは、TbMod サンプル SQLJ アプリケーションを構築して実行する方法を示しています。これらのステップは、ビルド・ファイル bldsqlj (UNIX) または bldsqlj.bat (Windows) を使用します。これには SQLJ アプレットまたはアプリケーションのいずれかを構築するためのコマンドが含まれています。

ビルド・ファイルは最大で 6 つのパラメーターを取ります。これは UNIX 上では \$1、\$2、\$3、\$4、\$5、および \$6、Windows 上では %1、%2、%3、%4、%5、および %6 です。最初のパラメーターはプログラムの名前を指定します。2 番目のパラメーターはデータベース・インスタンスのユーザー ID を指定し、3 番目のパラメーターはパスワードを指定します。4 番目のパラメーターはサーバー名を指定します。5 番目のパラメーターはポート番号を指定します。6 番目のパラメーターはデータベース名を指定します。最初のパラメーター (プログラム名) を除く他のすべてについては、デフォルト値を使用できます。デフォルトのパラメーター値の使用については詳しくは、ビルド・ファイルを参照してください。

1. アプリケーションをこのコマンドで構築するには、以下のようにします。

```
blidsqlj TbMod <userid> <password> <server_name> <port_number> <db_name>
```

2. Java アプリケーションを UNIX 上の 64 ビットの DB2 インスタンスで実行しているが、Software Development Kit for Java が 32 ビットである場合、アプリケーションを実行する前に DB2 ライブラリー・パスを変更する必要があります。例えば、AIX 上では以下のようにします。

- bash または Korn シェルを使用している場合:

```
export LIBPATH=$HOME/sql1lib/lib32
```

- C シェルを使用している場合:

```
setenv LIBPATH $HOME/sql1lib/lib32
```

3. Java インタープリターをアプリケーション上で、以下のコマンドで実行します。

```
java TbMod
```

4. And finally, this.

Here's an example... Insert tab A into slot B.

Now, you too can do this...

---

## Java アプレットに関する考慮事項

DB2 データベースには、Java アプレットを使用してアクセスすることができます。

アプレットを使用する場合は、以下の点に注意してください。

- いくつかの Java クラスから構成される比較的大きな JDBC または SQLJ アプレットの場合は、そのすべてのクラスを単一の JAR ファイル内にパッケージ化することができます。また、SQLJ アプレットの場合、直列化プロファイルそのクラスと共にパッケージ化する必要もあります。このパッケージ化を行う場合は、JAR ファイルを "applet" タグ内の archive パラメーターに追加します。詳細については、ご使用の Java 用ソフトウェア開発キットの資料を参照してください。

SQLJ アプレットの場合、一部のブラウザには、アプレットに関連付けられたリソース・ファイルからシリアライズド・オブジェクトを読み込むためのサポートがまだありません。このようなブラウザで付属のサンプル・アプレット Applet を読み込もうとすると、次のエラー・メッセージが表示されます。

```
java.lang.ClassNotFoundException: Applet_SJProfile0
```

この回避策として、直列化プロファイルを Java クラス形式で保管されたプロファイルに変換するユーティリティがあります。このユーティリティは、`sqlj.runtime.profile.util.SerProfileToClass` という名前の Java クラスです。このユーティリティでは、直列化プロファイルのリソース・ファイルが入力として読み込まれ、このプロファイルが含まれた Java クラスが出力として生成されます。プロファイルは以下のいずれかのコマンドを使用して変換できます。

```
profconv Applt_SJProfile0.ser
```

または

```
java sqlj.runtime.profile.util.SerProfileToClass Applt_SJProfile0.ser
```

このコマンドの結果、クラス `Applt_SJProfile0.class` が生成されます。アプレットで使用される `.ser` 形式のすべてのプロファイルを `.class` 形式のプロファイルに置き換えれば、この問題は解消されるはずです。

- `db2jcc.jar` というファイルを、ご使用の Web サイトから読み込まれる可能性があるいくつかのアプレットによって共有される特定のディレクトリーに配置することができます。 `db2jcc.jar` は、IBM Data Server Driver for JDBC and SQLJ を使用するアプレットまたは SQLJ アプレットのためのものです。このファイルは、Windows オペレーティング・システムの場合には `sql1lib¥java` ディレクトリーに、UNIX の場合には `sql1lib/java` ディレクトリーにあります。このディレクトリーを特定するために、`codebase` パラメーターを HTML ファイル内の "applet" タグに追加する必要がある場合もあります。詳細については、ご使用の Software Development Kit for Java の資料を参照してください。

JDBC 4.0 の機能を使用する場合は、`db2jcc.jar` ではなく `db2jcc4.jar` をコピーしてください。

- JDBC アプレット・サーバー (リスナー) である `db2jd` には、サーバーを堅固にするためのシグナル処理が備わっています。このため、CTRL-C キー・シーケンスを使用して、`db2jd` を終了することはできません。したがって、リスナーを終了させる唯一の方法は、`kill -9` (UNIX の場合) または「タスク マネージャ」 (Windows の場合) を使用してこのプロセスを強制終了することです。

---

## UNIX の SQLJ アプリケーションおよびアプレット・オプション

`bldsqlj` ビルド・スクリプトは、UNIXオペレーティング・システムで SQLJ アプリケーションおよびアプレットを構築します。 `bldsqlj` は SQLJ 変換プログラムおよびカスタマイザー・オプションのセットを指定します。

**推奨:** UNIX プラットフォームで SQLJ アプリケーションおよびアプレットを構築する際に `bldsqlj` が使用するものと同じ SQLJ 変換プログラムおよびカスタマイザー・オプションを使用してください。

`bldsqlj` が含んでいるオプションは、以下のとおりです。

**sqlj** SQLJ 変換プログラム (プログラムのコンパイルも行います)。

**"\${progname}.sqlj"**

SQLJ ソース・ファイル。拡張子が入力ファイル名に含まれていた場合、`progname=${1%.sqlj}` コマンドはその拡張子を除去します。そのため、拡張子を再び追加しても重複することはありません。

### **db2sqljcustomize**

SQLJ プロファイル・カスタマイザー。

**-url** データベース接続を確立するための JDBC URL を指定します  
(jdbc:db2://servername:50000/sample など)。

**-user** ユーザー ID を指定します。

### **-password**

パスワードを指定します。

### **"\${progname}\_SJProfile0"**

プログラムのシリアライズド・プロファイルを指定します。

---

## Windows の SQLJ アプリケーションおよびアプレット・オプション

bldsqlj.bat バッチ・ファイルは、Windows オペレーティング・システムで SQLJ アプリケーションおよびアプレットを構築します。bldsqlj.bat は SQLJ 変換プログラムおよびカスタマイザー・オプションのセットを指定します。

**推奨:** Windows オペレーティング・システムで SQLJ アプリケーションおよびアプレットを構築する際に bldsqlj.bat が使用するものと同じ SQLJ 変換プログラムおよびカスタマイザー・オプションを使用してください。

bldsqlj.bat が含んでいるオプションは、以下のとおりです。

**sqlj** SQLJ 変換プログラム (プログラムのコンパイルも行います)。

### **%1.sqlj**

SQLJ ソース・ファイル。

### **db2sqljcustomize**

SQLJ プロファイル・カスタマイザー。

**-url** データベース接続を確立するための JDBC URL を指定します  
(jdbc:db2://servername:50000/sample など)。

**-user** ユーザー ID を指定します。

### **-password**

パスワードを指定します。

### **%1\_SJProfile0**

プログラムのシリアライズド・プロファイルを指定します。

---

## SQL ルーチンの構築

SQLJ ルーチンを構築するには、Java makefile または bldsqljs ビルド・ファイルを使用できます。これらのルーチンを構築した後で、それらをカタログする必要があります。

以下のステップは、SpServer サンプル SQLJ ストアード・プロシージャを構築して実行する方法を示しています。これらのステップは、ビルド・ファイル bldsqljs (UNIX) または bldsqljs.bat (Windows) を使用します。これには SQLJ アプレットまたはアプリケーションのいずれかを構築するためのコマンドが含まれています。

ビルド・ファイルは最大で 6 つのパラメーターを取ります。これは UNIX 上では \$1、\$2、\$3、\$4、\$5、および \$6、Windows 上では %1、%2、%3、%4、%5、および %6 です。最初のパラメーターはプログラムの名前を指定します。2 番目のパラメーターはデータベース・インスタンスのユーザー ID を指定し、3 番目のパラメーターはパスワードを指定します。4 番目のパラメーターはサーバー名を指定します。5 番目のパラメーターはポート番号を指定します。6 番目のパラメーターはデータベース名を指定します。最初のパラメーター (プログラム名) を除く他のすべてについては、デフォルト値を使用できます。デフォルトのパラメーター値の使用について詳しくは、ビルド・ファイルを参照してください。

1. ストアード・プロシージャー・アプリケーションをこのコマンドで構築するには、以下のようにします。

```
bldsqljs SpServer <userid> <password> <server_name> <port_number> <db_name>
```

2. ストアード・プロシージャーを以下のコマンドでカタログします。

```
spcat
```

このスクリプトは、サンプル・データベースに接続し、ルーチンが既にカタログ済みであれば SpDrop.db2 を呼び出してルーチンをアンカタログし、次に SpCreate.db2 を呼び出してそのルーチンをカタログし、そして最後にデータベースへの接続を切断します。また、SpDrop.db2 スクリプトと SpCreate.db2 スクリプトは、個別に実行することもできます。

3. データベースを一度停止してから再始動し、新しいクラス・ファイルが認識されるようにします。必要であれば、クラス・ファイルのファイル・モードを読み取りに設定して、fenced ユーザーが読み取ることができるようにします。
4. SpClient クライアント・アプリケーションをコンパイルして実行し、ストアード・プロシージャー・クラスにアクセスします。SpClient は、アプリケーション・ビルド・ファイルの bldsqlj (UNIX) または bldsqlj.bat (Windows) で構築できます。

---

## UNIX の SQLJ ルーチン・オプション

bldsqljs ビルド・スクリプトは、UNIXオペレーティング・システムで SQLJ ルーチンを構築します。bldsqljs は SQLJ 変換プログラムおよびカスタマイザー・オプションのセットを指定します。

**推奨:** UNIX プラットフォームで SQLJ ルーチンを構築する際に bldsqljs が使用するものと同じ SQLJ 変換プログラムおよびカスタマイザー・オプションを使用してください。

bldsqljs が含んでいるオプションは、以下のとおりです。

**sqlj** SQLJ 変換プログラム (プログラムのコンパイルも行います)。

**"\${progname}.sqlj"**

SQLJ ソース・ファイル。拡張子が入力ファイル名に含まれていた場合、progname=\${1%.sqlj} コマンドはその拡張子を除去します。そのため、拡張子を再び追加しても重複することはありません。

**db2sqljcustomize**

SQLJ プロファイル・カスタマイザー。



- url** データベース接続を確立するための JDBC URL を指定します (jdbc:db2://servername:50000/sample など)。
- user** ユーザー ID を指定します。
- password**  
パスワードを指定します。
- "\${progname}\_SJProfile0"**  
プログラムのシリアルライズド・プロファイルを指定します。

---

## Windows の SQLJ ルーチン・オプション

bldsqljs.bat バッチ・ファイルは、Windows オペレーティング・システムで SQLJ ルーチンを構築します。bldsqljs.bat は SQLJ 変換プログラムおよびカスタマイザー・オプションのセットを指定します。

**推奨:** Windows オペレーティング・システムで SQLJ ルーチンを構築する際に bldsqljs.bat が使用するものと同じ SQLJ 変換プログラムおよびカスタマイザー・オプションを使用してください。

以下の SQLJ 変換プログラムおよびカスタマイザー・オプションが、Windows オペレーティング・システム上の bldsqljs.bat バッチ・ファイルで使用されます。これらのオプションは、DB2 が SQLJ ルーチン (ストアード・プロシージャおよびユーザー定義関数) の構築に使用するように推奨しているものです。

**sqlj** SQLJ 変換プログラム (プログラムのコンパイルも行います)。

**%1.sqlj**

SQLJ ソース・ファイル。

**db2sqljcustomize**

DB2 for Java プロファイル・カスタマイザー。

**-url** データベース接続を確立するための JDBC URL を指定します (jdbc:db2://servername:50000/sample など)。

**-user** ユーザー ID を指定します。

**-password**

パスワードを指定します。

**%1\_SJProfile0**

プログラムのシリアルライズド・プロファイルを指定します。





---

## 第 8 章 IBM Data Server Driver for JDBC and SQLJ での問題診断

IBM Data Server Driver for JDBC and SQLJ での SQLJ または JDBC 問題を診断するためのデータを取得するには、トレース・データを収集し、そのトレース・データをフォーマットするユーティリティを実行します。

トレースおよび診断ユーティリティは、IBM ソフトウェア・サポートの指示の下でのみ実行してください。

### JDBC トレース・データの収集

以下の手順のいずれかを使用して、トレースを開始します。

手順 1: DB2 for Linux, UNIX and Windows 用の IBM Data Server Driver for JDBC and SQLJ Type 4 接続 または IBM Data Server Driver for JDBC and SQLJ Type 2 接続の場合、推奨されるメソッドは、IBM Data Server Driver for JDBC and SQLJ 構成プロパティ・ファイルで `db2.jcc.override.traceFile` プロパティまたは `db2.jcc.override.traceDirectory` プロパティを設定して、トレースを開始することです。

手順 2:

1. `DataSource` インターフェースを使用してデータ・ソースに接続する場合、`DB2BaseDataSource.setTraceLevel` メソッドを呼び出して、必要なトレースのタイプを設定します。デフォルトのトレース・レベルは `TRACE_ALL` です。複数のトレースのタイプを指定する方法については、『IBM Data Server Driver for JDBC and SQLJ のプロパティ』を参照してください。
2. `DB2BaseDataSource.setJccLogWriter` メソッドを呼び出して、トレースの宛先を指定し、トレースをオンにします。

手順 3:

`DataSource` インターフェースを使用してデータ・ソースに接続する場合、`javax.sql.DataSource.setLogWriter` メソッドを呼び出して、トレースをオンにします。このメソッドを使用する場合、使用可能なトレース・レベルは `TRACE_ALL` のみです。

`DriverManager` インターフェースを使用してデータ・ソースに接続する場合、以下の手順を実行して、トレースを開始します。

1. `traceLevel` プロパティを必要なトレースのタイプの `info` パラメーターまたは `url` パラメーターで設定して、`DriverManager.getConnection` メソッドを呼び出します。デフォルトのトレース・レベルは `TRACE_ALL` です。複数のトレースのタイプを指定する方法については、『IBM Data Server Driver for JDBC and SQLJ のプロパティ』を参照してください。
2. `DriverManager.setLogWriter` メソッドを呼び出して、トレースの宛先を指定し、トレースをオンにします。

接続が確立された後、トレースをオフにしたりオンに戻したり、トレースの宛先を変更したり、`DB2Connection.setJccLogWriter` メソッドを使用してトレース・レベルを変更したりできます。トレースをオフにするには、`logWriter` 値を `null` に設定します。

`logWriter` プロパティはタイプ `java.io.PrintWriter` のオブジェクトです。アプリケーションが `java.io.PrintWriter` オブジェクトを処理できない場合、`traceFile` プロパティを使用してトレース出力の宛先を指定できます。`traceFile` プロパティを使用するには、`logWriter` プロパティを `null` に設定し、`traceFile` プロパティをドライバーがトレース・データを書き込むファイルの名前に設定します。このファイルおよびこのファイルが置かれているディレクトリは書き込み可能でなければなりません。ファイルが既に存在している場合、ドライバーがこのファイルを上書きします。

手順 4: `DriverManager` インターフェースを使用している場合、ドライバーのロード時に URL の一部として `traceFile` および `traceLevel` プロパティを指定します。以下に例を示します。

```
String url = "jdbc:db2://sysmvs1.st1.ibm.com:5021/san_jose" +
":traceFile=/u/db2p/jcctrace;" +
"traceLevel=" + com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRDA_FLOWS + ";;";
```

手順 5: `DB2TraceManager` メソッドを使用します。`DB2TraceManager` クラスは、すべてのタイプのログ書き込みプログラムによるトレースを中断および再開する機能を提供します。

構成プロパティを使用してトレースを開始する例: 構成パラメーターを使用してトレース・データを収集する完全な例については、『構成プロパティを使用して JDBC トレースを開始する例』を参照してください。

トレース例プログラム: IBM Data Server Driver for JDBC and SQLJ でのトレース用のプログラムの完全な例については、『IBM Data Server Driver for JDBC and SQLJ 使用時のトレース・プログラムの例』を参照してください。

## カスタマイズまたはバインド時の SQLJ トレース・データの収集

SQLJ のカスタマイズまたはバインド処理時に問題を診断するためにトレース・データを収集するには、`db2sqljcustomize` または `db2sqljbind` バインド・ユーティリティーの実行時に `-tracelevel` および `-tracefile` オプションを指定します。

## SQLJ シリアライズド・プロファイルに関する情報のフォーマット

`profp` ユーティリティーは、シリアライズド・プロファイル内の各 SQLJ 節に関する情報をフォーマットします。`profp` ユーティリティーのフォーマットは次のとおりです。

▶▶—`profp—serialized-profile-name`—▶▶

エラーが発生している、接続用のシリアライズド・プロファイルに対して `profp` ユーティリティーを実行します。例外がスローされる場合、Java スタック・トレースが生成されます。例外がスタック・トレースからスローされたときに、どのシリアライズド・プロファイルが使用されていたかを判別できます。

## SQLJ カスタマイズ済みシリアルライズド・プロファイルに関する情報のフォーマット

db2sqljprint ユーティリティーは、IBM Data Server Driver for JDBC and SQLJ 用にカスタマイズされたシリアルライズド・プロファイル内の各 SQLJ 節に関する情報をフォーマットします。

エラーが発生している、接続用のカスタマイズ済みシリアルライズド・プロファイルに対して db2sqljprint ユーティリティーを実行します。

---

## 構成プロパティを使用して JDBC トレースを開始する例

JDBC アプリケーションを変更せずに、そのアプリケーションのトレースを制御できます。

IBM Data Server Driver for JDBC and SQLJ Type 4 接続を使用する Test.java という名前のプログラムのトレース・データを収集すると仮定します。Test.java はトレースを行いませんが、プログラムを変更するつもりはありません。そのため、構成プロパティを使用してトレースを使用可能にします。トレース出力に以下の特性を持たせることにします。

- 同じ DataSource での各接続のトレース情報は別個のトレース・ファイルに書き込まれます。出力は /Trace という名前のディレクトリに入れられます。
- 各トレース・ファイルの名前は jccTrace1 で始まります。
- 同じ名前のトレース・ファイルが既に存在する場合、トレース・データがそれらに追加されます。

Test1.java にはトレースを行うコードが含まれていませんが、今後トレースを行うようにアプリケーションが変更される場合にプログラム内の設定が構成プロパティの設定より優先するように、その構成プロパティを設定します。これを行うには、db2.jcc.override ではなく、db2.jcc で始まる構成プロパティの設定を使用します。

構成プロパティ設定は次のようになります。

- db2.jcc.traceDirectory=/Trace
- db2.jcc.traceFile=jccTrace1
- db2.jcc.traceFileAppend=true

トレース設定がスタンドアロン・プログラム Test1.java にのみ適用されるようにします。そのため、そのような設定のファイルを作成し、-Ddb2.jcc.propertiesFile オプションを指定して Java プログラムを呼び出すときにそのファイルを参照するようにします。その設定を含むファイルが /Test/jcc.properties であるとし、Test1.java の実行時にトレースを使用可能にするには、次のようなコマンドを発行します。

```
java -Ddb2.jcc.propertiesFile=/Test/jcc.properties Test1
```

Test1.java が 1 つの DataSource に 2 つの接続を作成するとします。プログラムは logWriter オブジェクトを定義しません。そのため、ドライバーはトレース出力用にグローバル logWriter オブジェクトを作成します。プログラムが完了すると、以下のファイルにトレース・データが含まれています。

- /Trace/jccTrace1\_global\_0

## IBM Data Server Driver for JDBC and SQLJ 使用時のトレース・プログラムの例

トレース用のメソッドを組み込んでいる単一のクラスを DriverManager インターフェースおよび DataSource インターフェースの下に作成することができます。

以下の例は、そのようなクラスを示しています。この例では、IBM Data Server Driver for JDBC and SQLJ Type 4 接続が使用されています。

図 57. IBM Data Server Driver for JDBC and SQLJ 使用時のトレースの例

```
public class TraceExample
{
    public static void main(String[] args)
    {
        sampleConnectUsingSimpleDataSource();
        sampleConnectWithURLUsingDriverManager();
    }

    private static void sampleConnectUsingSimpleDataSource()
    {
        java.sql.Connection c = null;
        java.io.PrintWriter printWriter =
            new java.io.PrintWriter(System.out, true);
                                                // Prints to console, true means
                                                // auto-flush so you don't lose trace
        try {
            javax.sql.DataSource ds =
                new com.ibm.db2.jcc.DB2SimpleDataSource();
            ((com.ibm.db2.jcc.DB2BaseDataSource) ds).setServerName("sysmvs1.st1.ibm.com");
            ((com.ibm.db2.jcc.DB2BaseDataSource) ds).setPortNumber(5021);
            ((com.ibm.db2.jcc.DB2BaseDataSource) ds).setDatabaseName("san_jose");
            ((com.ibm.db2.jcc.DB2BaseDataSource) ds).setDriverType(4);

            ds.setLogWriter(printWriter);    // This turns on tracing

            // Refine the level of tracing detail
            ((com.ibm.db2.jcc.DB2BaseDataSource) ds).
                setTraceLevel(com.ibm.db2.jcc.DB2SimpleDataSource.TRACE_CONNECTS |
                    com.ibm.db2.jcc.DB2SimpleDataSource.TRACE_DRDA_FLOWS);

            // This connection request is traced using trace level
            // TRACE_CONNECTS | TRACE_DRDA_FLOWS
            c = ds.getConnection("myname", "mypass");

            // Change the trace level to TRACE_ALL
            // for all subsequent requests on the connection
            ((com.ibm.db2.jcc.DB2Connection) c).setJccLogWriter(printWriter,
                com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL);
            // The following INSERT is traced using trace level TRACE_ALL
            java.sql.Statement s1 = c.createStatement();
            s1.executeUpdate("INSERT INTO sampleTable(sampleColumn) VALUES(1)");
            s1.close();

            // This code disables all tracing on the connection
            ((com.ibm.db2.jcc.DB2Connection) c).setJccLogWriter(null);

            // The following INSERT statement is not traced
            java.sql.Statement s2 = c.createStatement();
```

```

s2.executeUpdate("INSERT INTO sampleTable(sampleColumn) VALUES(1)");
s2.close();

c.close();
}
catch(java.sql.SQLException e) {
    com.ibm.db2.jcc.DB2ExceptionFormatter.printStackTrace(e,
        printWriter, "[TraceExample]");
}
finally {
    cleanup(c, printWriter);
    printWriter.flush();
}
}

// If the code ran successfully, the connection should
// already be closed. Check whether the connection is closed.
// If so, just return.
// If a failure occurred, try to roll back and close the connection.

private static void cleanup(java.sql.Connection c,
    java.io.PrintWriter printWriter)
{
    if(c == null) return;

    try {
        if(c.isClosed()) {
            printWriter.println("[TraceExample] " +
                "The connection was successfully closed");
            return;
        }

        // If we get to here, something has gone wrong.
        // Roll back and close the connection.
        printWriter.println("[TraceExample] Rolling back the connection");
        try {
            c.rollback();
        }
        catch(java.sql.SQLException e) {
            printWriter.println("[TraceExample] " +
                "Trapped the following java.sql.SQLException while trying to roll back:");
            com.ibm.db2.jcc.DB2ExceptionFormatter.printStackTrace(e, printWriter,
                "[TraceExample]");
            printWriter.println("[TraceExample] " +
                "Unable to roll back the connection");
        }
        catch(java.lang.Throwable e) {
            printWriter.println("[TraceExample] Trapped the " +
                "following java.lang.Throwable while trying to roll back:");
            com.ibm.db2.jcc.DB2ExceptionFormatter.printStackTrace(e,
                printWriter, "[TraceExample]");
            printWriter.println("[TraceExample] Unable to " +
                "roll back the connection");
        }
    }

    // Close the connection
    printWriter.println("[TraceExample] Closing the connection");
    try {
        c.close();
    }
    catch(java.sql.SQLException e) {
        printWriter.println("[TraceExample] Exception while " +
            "trying to close the connection");
        printWriter.println("[TraceExample] Deadlocks could " +
            "occur if the connection is not closed.");
        com.ibm.db2.jcc.DB2ExceptionFormatter.printStackTrace(e, printWriter,
            "[TraceExample]");
    }
}

```

```

    }
    catch(java.lang.Throwable e) {
        printWriter.println("[TraceExample] Throwable caught " +
            "while trying to close the connection");
        printWriter.println("[TraceExample] Deadlocks could " +
            "occur if the connection is not closed.");
        com.ibm.db2.jcc.DB2ExceptionFormatter.printTrace(e, printWriter,
            "[TraceExample]");
    }
}
catch(java.lang.Throwable e) {
    printWriter.println("[TraceExample] Unable to " +
        "force the connection to close");
    printWriter.println("[TraceExample] Deadlocks " +
        "could occur if the connection is not closed.");
    com.ibm.db2.jcc.DB2ExceptionFormatter.printTrace(e, printWriter,
        "[TraceExample]");
}
}
private static void sampleConnectWithURLUsingDriverManager()
{
    java.sql.Connection c = null;

    // This time, send the printWriter to a file.
    java.io.PrintWriter printWriter = null;
    try {
        printWriter =
            new java.io.PrintWriter(
                new java.io.BufferedOutputStream(
                    new java.io.FileOutputStream("/temp/driverLog.txt"), 4096), true);
    }
    catch(java.io.FileNotFoundException e) {
        java.lang.System.err.println("Unable to establish a print writer for trace");
        java.lang.System.err.flush();
        return;
    }

    try {
        Class.forName("com.ibm.db2.jcc.DB2Driver");
    }
    catch(ClassNotFoundException e) {
        printWriter.println("[TraceExample] " +
            "IBM Data Server Driver for JDBC and SQLJ type 4 connectivity " +
            "is not in the application classpath. Unable to load driver.");
        printWriter.flush();
        return;
    }

    // This URL describes the target data source for Type 4 connectivity.
    // The traceLevel property is established through the URL syntax,
    // and driver tracing is directed to file "/temp/driverLog.txt"
    // The traceLevel property has type int. The constants
    // com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRDA_FLOWS and
    // com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTS represent
    // int values. Those constants cannot be used directly in the
    // first getConnection parameter. Resolve the constants to their
    // int values by assigning them to a variable. Then use the
    // variable as the first parameter of the getConnection method.
    String databaseURL =
        "jdbc:db2://sysmvs1.st1.ibm.com:5021" +
        "/sample:traceFile=/temp/driverLog.txt;traceLevel=" +
        (com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRDA_FLOWS |
        com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTS) + ";";

    // Set other properties
    java.util.Properties properties = new java.util.Properties();
    properties.setProperty("user", "myname");

```



```

properties.setProperty("password", "mypass");

try {
    // This connection request is traced using trace level
    // TRACE_CONNECTS | TRACE_DRDA_FLOWS
    c = java.sql.DriverManager.getConnection(databaseURL, properties);

    // Change the trace level for all subsequent requests
    // on the connection to TRACE_ALL
    ((com.ibm.db2.jcc.DB2Connection) c).setJccLogWriter(printWriter,
        com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL);

    // The following INSERT is traced using trace level TRACE_ALL
    java.sql.Statement s1 = c.createStatement();
    s1.executeUpdate("INSERT INTO sampleTable(sampleColumn) VALUES(1)");
    s1.close();

    // Disable all tracing on the connection
    ((com.ibm.db2.jcc.DB2Connection) c).setJccLogWriter(null);

    // The following SQL insert code is not traced
    java.sql.Statement s2 = c.createStatement();
    s2.executeUpdate("insert into sampleTable(sampleColumn) values(1)");
    s2.close();

    c.close();
}
catch(java.sql.SQLException e) {
    com.ibm.db2.jcc.DB2ExceptionFormatter.printStackTrace(e, printWriter,
        "[TraceExample]");
}
finally {
    cleanup(c, printWriter);
    printWriter.flush();
}
}
}

```



---

## 第 9 章 IBM Data Server Driver for JDBC and SQLJ のシステム・モニター

IBM Data Server Driver for JDBC and SQLJ の使用時にアプリケーションのパフォーマンスをモニターする作業のために、このドライバーには、接続に関する情報を収集するためのメソッドが 2 つ用意されています。

その情報は、以下のとおりです。

### コア・ドライバー時間

システム・モニターが有効になっている間に収集されたモニター対象の API の経過時間の合計 (マイクロ秒単位)。一般にモニター対象になるのは、ネットワーク I/O またはデータベース・サーバー対話が発生する可能性のある API だけです。

### ネットワーク I/O 時間

システム・モニターが有効になっている間に収集されたネットワーク I/O の経過時間の合計 (マイクロ秒単位)。

### サーバー時間

システム・モニターが有効になっている間に収集されたすべてのレポート対象データベース・サーバーの経過時間の合計 (マイクロ秒単位)。

現時点で、IBM Informix Dynamic Server データベースはこの機能をサポートしていません。

### アプリケーション時間

アプリケーション、JDBC ドライバー、ネットワーク I/O、データベース・サーバーの経過時間の合計 (ミリ秒単位)。

2 つのメソッドは、以下のとおりです。

- `DB2SystemMonitor` インターフェース
- `TRACE_SYSTEM_MONITOR` トレース・レベル

`DB2SystemMonitor` インターフェースを使用してシステム・モニター・データを収集するには、以下の基本手順を実行します。

1. `DB2Connection.getDB2SystemMonitor` メソッドを呼び出して、`DB2SystemMonitor` オブジェクトを作成します。
2. `DB2SystemMonitor.enable` メソッドを呼び出して、接続の `DB2SystemMonitor` オブジェクトを使用可能にします。
3. `DB2SystemMonitor.start` メソッドを呼び出して、システム・モニターを開始します。
4. モニター対象のアクティビティーが完了したら、`DB2SystemMonitor.stop` を呼び出して、システム・モニターを停止します。

5.

`DB2SystemMonitor.getCoreDriverTimeMicros`、`DB2SystemMonitor.getNetworkIOTimeMicros`、`DB2SystemMonitor.getServerTimeMicros`、`DB2SystemMonitor.getApplicationTimeMillis` のいずれかのメソッドを呼び出して、経過時間データを取得します。

例えば、各タイプの経過時間データを収集するコードを以下に示します。選択されたステートメントの右にある番号は、以前に説明されたステップに対応しています。

```

import java.sql.*;
import com.ibm.db2.jcc.*;
public class TestSystemMonitor
{
    public static void main(String[] args)
    {
        String url = "jdbc:db2://sysmvs1.svl.ibm.com:5021/san_jose";
        String user="db2adm";
        String password="db2adm";
        try
        {
            // Load the IBM Data Server Driver for JDBC and SQLJ
            Class.forName("com.ibm.db2.jcc.DB2Driver");
            System.out.println("**** Loaded the JDBC driver");

            // Create the connection using the IBM Data Server Driver for JDBC and SQLJ
            Connection conn = DriverManager.getConnection (url,user,password);
            // Commit changes manually
            conn.setAutoCommit(false);
            System.out.println("**** Created a JDBC connection to the data source");
            DB2SystemMonitor systemMonitor = 1
                ((DB2Connection)conn).getDB2SystemMonitor();
            systemMonitor.enable(true); 2
            systemMonitor.start(DB2SystemMonitor.RESET_TIMES); 3
            Statement stmt = conn.createStatement();
            int numUpd = stmt.executeUpdate(
                "UPDATE EMPLOYEE SET PHONENO='4657' WHERE EMPNO='000010'");
            systemMonitor.stop(); 4
            System.out.println("Server elapsed time (microseconds)="
                + systemMonitor.getServerTimeMicros()); 5
            System.out.println("Network I/O elapsed time (microseconds)="
                + systemMonitor.getNetworkIOTimeMicros());
            System.out.println("Core driver elapsed time (microseconds)="
                + systemMonitor.getCoreDriverTimeMicros());
            System.out.println("Application elapsed time (milliseconds)="
                + systemMonitor.getApplicationTimeMillis());
            conn.rollback();
            stmt.close();
            conn.close();
        }
        // Handle errors
        catch(ClassNotFoundException e)
        {
            System.err.println("Unable to load the driver, " + e);
        }
        catch(SQLException e)
        {
            System.out.println("SQLException: " + e);
            e.printStackTrace();
        }
    }
}

```

図 58. *DB2SystemMonitor* メソッドを使用してシステム・モニター・データを収集する例

トレース・メソッドを使用してシステム・モニター情報を収集するには、構成プロパティー、または *Connection*、*DataSource* のいずれかのプロパティーを使用して JDBC トレースを開始します。 *traceLevel* プロパティーを設定するときに、*TRACE\_SYSTEM\_MONITOR* を組み込んでください。以下に例を示します。

```

String url = "jdbc:db2://sysmvs1.stl.ibm.com:5021/san_jose" +
    ":traceFile=/u/db2p/jcctrace;" +
    "traceLevel=" + com.ibm.db2.jcc.DB2BaseDataSource.TRACE_SYSTEM_MONITOR + ";";

```

システム・モニター情報のトレース・レコードは、以下のようになります。

```
[jcc][SystemMonitor:start]
```

```
...
```

```
[jcc][SystemMonitor:stop] core: 565.67ms | network: 211.695ms | server: 207.771ms
```

---

## IBM Data Server Driver for JDBC and SQLJ のリモート・トレース・コントローラー

IBM Data Server Driver for JDBC and SQLJ では、IBM Data Server Driver for JDBC and SQLJ のトレースを動的に制御するための機能が提供されます。

このリモート・トレース・コントローラーを使用すれば、複数のドライバー・インスタンス用のものと同様な以下の操作を実行することができます。

- トレースの開始、停止、または再開
- 出力トレース・ファイルまたはディレクトリーの場所の変更
- トレース・レベルの変更

リモート・トレース・コントローラーでは、Java Standard Edition バージョン 6 以降でその一部となっている Java Management Extensions (JMX) アーキテクチャが使用されます。JMX は、以下のものから構成されています。

- 組み込みの管理ユーティリティー一式 (これらを使用すると、Java Monitoring and Management Console (JConsole) などの管理コンソールからモニターを実行することができます)。
- 同じ機能を実行するためのアプリケーションを作成できる API 一式。

### リモート・トレース・コントローラーの使用可能化

リモート・トレース・コントローラーを使用可能にするには、IBM Data Server Driver for JDBC and SQLJ で Java Management Extensions (JMX) を使用可能にし、クライアントから JMX エージェントを使用できるようにする必要があります。

リモート・トレース・コントローラーでは、Java Standard Edition バージョン 6 以降が必要です。

リモート・トレース・コントローラーを使用可能にする手順は、以下のとおりです。

1. IBM Data Server Driver for JDBC and SQLJ で JMX を使用可能にするために、db2.jcc.jmxEnabled グローバル構成プロパティを true または yes に設定します。

例えば、DB2JccConfiguration.properties に以下のストリングを組み込みます。

```
db2.jcc.jmxEnabled=true
```

2. ローカル・クライアントまたはリモート・クライアントから JMX エージェント (プラットフォームの MBean サーバー) を使用できるようにします。
  - ローカル・クライアントの場合は、以下のようになります。

JVM の始動時に、モニター機能と管理機能が自動的に使用可能になります。アプリケーションを開始した後に、JConsole などの JMX クライアントを使用して、Java プロセスにローカルで接続できます。

- リモート・クライアントの場合は、以下のいずれかの方法を使用します。
  - すぐに使用可能な JMX エージェントを使用します。

すぐに使用可能な管理機能では、JMX 組み込み管理ユーティリティーを使用します。すぐに使用可能な管理機能を有効にするには、いくつかの Java システム・プロパティーを設定する必要があります。少なくとも以下のプロパティーを設定しなければなりません。

```
com.sun.management.jmxremote.port=portNum
```

さらに、認証と SSL が正しく構成されていることを確認してください。

すぐに使用可能な管理機能を有効にするための詳細については、以下の URL を参照してください。

```
http://java.sun.com/javase/6/docs/technotes/guides/management/agent.html
```

- JMX エージェントを作成します。この技法については、以下の資料も参照できます。

```
http://java.sun.com/javase/6/docs/technotes/guides/management/agent.html
```

以下の例では、MyCustomJMXAuthenticator オブジェクトを使用して、PlatformMBeanServer の RMI コネクター・サーバーを作成します。MyCustomJMXAuthenticator クラスでは、JMXAuthenticator インターフェースを実装して、リモート証明書を JAAS Subject に変換する方法を定義します。

```
...
HashMap<String> env = new HashMap<String>();
env.put(JMXConnectorServer.AUTHENTICATOR, new MyCustomJMXAuthenticator());
env.put("jmx.remote.x.access.file", "my.access.file");

MBeanServer mbs =
    java.lang.management.ManagementFactory.getPlatformMBeanServer();
JMXServiceURL url =
    new JMXServiceURL("service:jmx:rmi:///jndi/rmi://:9999/jmxrmi");

JMXConnectorServer cs =
    JMXConnectorServerFactory.newJMXConnectorServer(url, env, mbs);
cs.start();
...
public class MyCustomJMXAuthenticator implements JMXAuthenticator {

    public Subject authenticate(Object credentials) {
        // the hash contains username, password, etc...
        Hashtable <String> credentialsHash
            = (Hashtable <String>) credentials;

        ...
        // Authenticate using the provided credentials
        ...
        if (authentication-successful) {
            return new Subject(true,
                Collections.singleton
                    (new JMXPrincipal(credentialsHash.get("username"))),
                Collections.EMPTY_SET,
                Collections.EMPTY_SET);
        }
        throw new SecurityException("Invalid credentials");
    }
}
```



## リモート・トレース・コントローラーへのアクセス

リモート・トレース・コントローラーにアクセスするには、すぐに使用可能な管理ツールを使用するか、アプリケーションを使用します。

すぐに使用可能な管理機能は、JConsole などの JMX 準拠の管理クライアントから使用できます。JConsole は、Java Standard Edition バージョン 6 に組み込まれています。すぐに使用可能な管理機能を JConsole から使用するための詳細については、以下の URL を参照してください。

<http://java.sun.com/javase/6/docs/technotes/guides/management/jconsole.html>

リモート・トレース・コントローラーにアクセスするアプリケーションでは、リモート・トレース・コントローラーが管理対象 Bean (MBean) になります。JMX は、JMX エージェントによってリソースを管理します。JMX エージェントは、MBean サーバーです。それぞれの MBean は、1 つのリソースに対応します。すべての MBean には名前があり、その名前は、クラス `javax.management.ObjectName` のオブジェクトによって定義します。ObjectName オブジェクトを使用して、MBean を MBeanServer に登録したり、MBeanServer から MBean を取得したりすることも可能です。

MBean 名は、ドメインとキー・プロパティーという 2 つの部分からなっています。IBM Data Server Driver for JDBC and SQLJ のリモート・トレース・コントローラーの ObjectName の場合、ドメインは `com.ibm.db2.jcc`、キー・プロパティーは `name=DB2TraceManager` になります。

リモート・トレース・コントローラーにアクセスするアプリケーションには、以下の手順を組み込む必要があります。

1. MBean サーバーに対するリモート・メソッド呼び出し (RMI) 接続を確立します。
2. MBean サーバーでリモート・トレース・コントローラーに対する検索を実行します。
3. MBean に対するトレース操作を呼び出します。

MBean を操作するには、以下の方法があります。

- MBean プロキシを使用する方法
- プロキシを使用しないで MBeanServerConnection を使用する方法

**例: プロキシを使用しないでリモート・トレース・コントローラーにアクセスする方法:** この例では、MBeanServerConnection から MBean に直接アクセスする方法を示します。この方法は、JMX クライアント・アプリケーションでインターフェース定義を一致させる必要がないので非常に汎用的です。

```
Hashtable<String> env = new Hashtable<String>();
env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.sun.jndi.fscontext.ReffSContextFactory");

try {
    System.out.println ("");
    System.out.println ("-----");
    System.out.println ("Establish an RMI connection to an MBeanServer");
    System.out.println ("-----");
    JMXServiceURL url =
        new JMXServiceURL ("service:jmx:rmi:///jndi/rmi://localhost:9999/jmxrmi");
```

```

JMXConnector jmx = JMXConnectorFactory.connect (url, env);
MBeanServerConnection mbsc = jmx.getMBeanServerConnection();

System.out.println ("");
System.out.println ("-----");
System.out.println ("Processing MBean");
System.out.println ("-----");
String objectNameString = "com.ibm.db2.jcc:name=DB2TraceManager";
ObjectName name = new ObjectName(objectNameString);
System.out.println ("ObjectName="+objectNameString);

System.out.println ("");
System.out.println ("-----");
System.out.println ("Print all attributes of the MBean");
System.out.println ("-----");

System.out.println(
    "TraceDirectory = "+mbsc.getAttribute (name, "TraceDirectory"));
System.out.println(
    "TraceFile      = "+mbsc.getAttribute (name, "TraceFile"));
System.out.println(
    "TraceFileAppend = "+mbsc.getAttribute (name, "TraceFileAppend"));
System.out.println(
    "TraceLevel     = "+mbsc.getAttribute (name, "TraceLevel"));

System.out.println ("");
System.out.println ("-----");
System.out.println ("Invoke some operations on the MBean");
System.out.println ("-----");
System.out.print ("Invoking suspendTrace()...");
mbsc.invoke (name, "suspendTrace", null , null);
System.out.println ("success");

System.out.print ("Invoking resumeTrace()...");
mbsc.invoke (name, "resumeTrace", null , null);
System.out.println ("success");
}
catch (Exception e) {
    System.out.println ("failure");
    e.printStackTrace ();
}
}

```

**例: プロキシを使用してリモート・トレース・コントローラーにアクセスする方法:** この例では、MBean のプロキシを作成する方法を示します。このプロキシでは、com.ibm.db2.jcc.mx.DB2TraceManagerMXBean インターフェースを実装します。アプリケーションは、そのプロキシに対して呼び出しを直接実行し、基礎になっているプロキシ実装がリモート MBean サーバーに対する MBean 操作を呼び出します。

```

Hashtable<String> env = new Hashtable<String>();
env.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.fscontext.ReffSContextFactory");

try {
    System.out.println ("");
    System.out.println ("-----");
    System.out.println ("Establish an RMI connection to an MBeanServer");
    System.out.println ("-----");
    JMXServiceURL url =
        new JMXServiceURL ("service:jmx:rmi:///jndi/rmi://localhost:9999/jmxrmi");
    JMXConnector jmx = JMXConnectorFactory.connect (url, env);
    MBeanServerConnection mbsc = jmx.getMBeanServerConnection();

    System.out.println ("");
    System.out.println ("-----");
}

```

```

System.out.println ("Processing MBean");
System.out.println ("-----");
String objectNameString = "com.ibm.db2.jcc:name=DB2TraceManager";
ObjectName name = new ObjectName(objectNameString);
System.out.println ("ObjectName="+objectNameString);

System.out.println ("");
System.out.println ("-----");
System.out.println ("Print all attributes of the MBean");
System.out.println ("-----");
com.ibm.db2.jcc.mx.DB2TraceManagerMBean mbeanProxy =
    JMX.newMBeanProxy(mbsc, name,
        com.ibm.db2.jcc.mx.DB2TraceManagerMBean.class, true);
System.out.println ("TraceDirectory = "+mbeanProxy.getTraceDirectory ());
System.out.println ("TraceFile      = "+mbeanProxy.getTraceFile ());
System.out.println ("TraceFileAppend = "+mbeanProxy.getTraceFileAppend ());
System.out.println ("TraceLevel     = "+mbeanProxy.getTraceLevel ());
System.out.println ("");
System.out.println ("-----");
System.out.println ("Invoke some operations on the MBean");
System.out.println ("-----");
System.out.print ("Invoking suspendTrace()...");
mbeanProxy.suspendTrace();
System.out.println ("success");
System.out.print ("Invoking resumeTrace()...");
mbeanProxy.resumeTrace();
System.out.println ("success");
}
catch (Exception e) {
    System.out.println ("failure");
    e.printStackTrace ();
}
}

```

---

## 第 10 章 Java 2 Platform, Enterprise Edition

Java 2 Platform, Enterprise Edition (J2EE) を使用すれば、下に述べるような複数層サービスの開発に伴うコストと複雑さが低減し、企業の要件に基づいて迅速にデプロイして容易に拡張できるサービスを実現することができます。

今日のグローバルなビジネス環境においては、組織は活動範囲を広げ、コストを削減し、また応答時間を短縮するために、顧客、従業員、サプライヤー、および他のビジネス・パートナーが容易にアクセスできるサービスを提供する必要があります。これらのサービスには、以下の特性が備わっている必要があります。

- 高可用性 - グローバルなビジネス環境の要件を満たすため
- セキュリティー - ユーザーのプライバシーと企業の保全性を保護するため
- 信頼性と拡張性 - 商取引が正確かつ迅速に処理されるため

これらのサービスは、多くの場合、各層が特定の目的を満たす複数層アプリケーションによって提供されます。

J2EE では、以下のような要素として提供されている標準アーキテクチャーを定義することにより、これらの利点を実現されます。

- J2EE Application Model - 複数層のシン・クライアント・サービスを開発するための標準アプリケーション・モデル
- J2EE Platform - J2EE アプリケーションをホストするための標準のプラットフォーム
- J2EE Compatibility Test Suite - 特定の J2EE Platform 製品の J2EE Platform 標準への準拠の検査
- J2EE Reference Implementation - J2EE 機能のデモおよび J2EE Platform の作動可能な定義の提供

---

### Java 2 Platform, Enterprise Edition のアプリケーション・コンポーネントのサポート

Java 2 Platform, Enterprise Edition (J2EE) では、J2EE アプリケーションをホストするためのランタイム環境が提供されます。

このランタイム環境では、J2EE 製品でサポートされる必要がある以下の 4 つのアプリケーション・コンポーネント・タイプが定義されます。

- アプリケーション・クライアントは、通常デスクトップ・コンピューター上で実行する GUI プログラムである Java プログラミング言語プログラムです。アプリケーション・クライアントでは、J2EE 中間層の機能のすべてにアクセスできません。
- アプレットは、通常 Web ブラウザーで実行される GUI コンポーネントですが、アプレット・プログラミング・モデルをサポートしている他のさまざまなアプリケーションや装置でも実行できます。

- サーブレット、JavaServer Pages (JSP)、フィルター、および Web イベント・リスナーは、通常 Web サーバーで実行され、Web クライアントからの HTTP 要求に応答できます。サーブレット、JSP、およびフィルターは、アプリケーションのユーザー・インターフェースである HTML ページを生成するために使用できます。また、他のアプリケーション・コンポーネントによって取り込まれる XML や他のフォーマットのデータも生成できます。サーブレット、JSP テクノロジーによって作成されたページ、Web フィルター、および Web イベント・リスナーは、この仕様では Web コンポーネント と総称されています。Web アプリケーションは、Web コンポーネントと HTML ページなどのその他のデータから構成されます。
- Enterprise JavaBeans™ (EJB) コンポーネントは、トランザクションをサポートしている管理下環境で実行されます。エンタープライズ Bean には、通常 J2EE アプリケーション用のビジネス・ロジックが含まれています。

上記の各アプリケーション・コンポーネントは、デプロイ方法や管理方法に基づいて 3 つのカテゴリに分類することができます。

- J2EE サーバー上でデプロイ、管理、および実行されるコンポーネント。
- J2EE サーバー上でデプロイおよび管理されるが、クライアント・マシンにロードされて実行されるコンポーネント。
- デプロイメント方法と管理方法がこの仕様では完全に定義されていないコンポーネント。アプリケーション・クライアントはこのカテゴリに含めることができます。

これらのコンポーネントのランタイム・サポートは、コンテナーによって提供されます。

---

## Java 2 Platform, Enterprise Edition コンテナー

コンテナーでは、ベースとなる Java 2 Platform, Enterprise Edition (J2EE) API のフェデレーテッド・ビューを、アプリケーション・コンポーネントに提供します。

標準的な J2EE 製品では、アプリケーション・コンポーネント・タイプごとに 1 つのコンテナー (アプリケーション・クライアント・コンテナー、アプレット・コンテナー、Web コンテナー、およびエンタープライズ Bean コンテナー) が備わっています。また、コンテナー・ツールでは、アプリケーション・コンポーネントをデプロイメント用にパッケージするためのファイル形式も認識されます。

仕様によると、これらのコンテナーは、Java 互換のランタイム環境を提供する必要があります。この仕様では、各 J2EE 製品がサポートする必要がある一連の標準サービスが定義されています。これらの標準サービスは以下のとおりです。

- HTTP Service
- HTTPS Service
- Java Transaction API
- Remote Invocation Method
- Java IDL
- JDBC API
- Java Message Service
- Java Naming and Directory Interface

- JavaMail
- JavaBeans Activation Framework
- Java API for XML Parsing
- Connector Architecture
- Java Authentication and Authorization Service

---

## Java 2 Platform, Enterprise Edition Server

Java 2 Platform, Enterprise Edition (J2EE) コンテナのベースとなるものは、このコンテナがその一部を構成しているサーバーです。

通常 J2EE Product Provider により、J2EE のサーバー・サイド機能がインプリメントされます。また、J2EE クライアント機能は、通常 J2SE テクノロジーに基づいて構築されます。

IBM WebSphere Application Server は、J2EE に準拠したサーバーです。

---

## Java 2 Platform, Enterprise Edition のデータベース要件

Java 2 Platform, Enterprise Edition では、ビジネス・データの保管用に JDBC API を介してアクセス可能なデータベースが必要です。

このデータベースには、Web コンポーネント、エンタープライズ Bean、およびアプリケーション・クライアント・コンポーネントからアクセスすることができます。また、アプレットからアクセスできる必要はありません。

---

## Java Naming and Directory Interface (JNDI)

JNDI を使用すれば、Java プラットフォーム・ベースのアプリケーションによる複数の命名およびディレクトリー・サービスへのアクセスが可能になります。

これは、Java Enterprise アプリケーション・プログラミング・インターフェース (API) セットの一部です。JNDI により、開発者は複数の異なる命名およびディレクトリー・サービスに対応可能なポータブル・アプリケーションを作成できます。これらの使用可能なサービスには、ファイル・システム、Lightweight Directory Access Protocol (LDAP) や Novell Directory Services などのディレクトリー・サービス、および Common Object Request Broker Architecture (CORBA)、Java Remote Method Invocation (RMI)、Enterprise JavaBeans (EJB) などの分散オブジェクト・システムがあります。

JNDI API には次の 2 つの部分があります。ひとつは、アプリケーション・コンポーネントにより命名およびディレクトリー・サービスへのアクセスに使用されるアプリケーション・レベル・インターフェースであり、もうひとつは、命名およびディレクトリー・サービスのプロバイダーにアタッチするためのサービス・プロバイダー・インターフェースです。

---

## Java トランザクション管理

Java 2 Platform, Enterprise Edition (J2EE) では、分散トランザクション管理用のアプリケーション・プログラミングが容易になります。

J2EE には、Java Transaction API (JTA) および Java Transaction Service (JTS) という 2 つの仕様による分散トランザクションのサポートが組み込まれています。JTA はインプリメンテーションやプロトコルに依存しない高水準 API で、アプリケーションやアプリケーション・サーバーによるトランザクションへのアクセスを可能にします。また、JTA は常に使用可能になっています。

IBM Data Server Driver for JDBC and SQLJ および Linux、UNIX、Windows 用の DB2 JDBC Type 2 ドライバーでは、JTA および JTS の各仕様がインプリメントされています。

IBM Data Server Driver for JDBC and SQLJ Type 4 接続の場合、分散トランザクションは DB2 Database for Linux, UNIX, and Windows、DB2 for z/OS、および DB2 for i5/OS の各サーバーでサポートされています。

JTA では、トランザクション・マネージャーと分散トランザクション・システムに関連する部分 (リソース・マネージャー、アプリケーション・サーバー、およびトランザクション・アプリケーション) との間の標準の Java インターフェースが指定されます。

JTS では、JTA をサポートし、OMG Object Transaction Service (OTS) 1.1 仕様の Java マッピングをこの API の下のレベルでインプリメントするトランザクション・マネージャーのインプリメンテーションが指定されます。JTS では、IIOP を使用してトランザクションが伝搬されます。

JTA および JTS により、アプリケーション用の J2EE サーバーでは、トランザクション管理の負担がコンポーネント開発者から取り除かれます。開発者は、設計時またはデプロイメント時にデプロイメント記述子内で宣言ステートメントを使用して、EJB テクノロジー・ベースのコンポーネントのトランザクション・プロパティを定義できます。アプリケーション・サーバーにより、トランザクション管理の責任が引き受けられます。

DB2 と WebSphere Application Server の環境では、WebSphere Application Server がトランザクション・マネージャーの役割を担い、DB2 がリソース・マネージャーとして機能します。WebSphere Application Server および DB2 による整合分散トランザクションの提供が可能になるように、WebSphere Application Server により JTS と JTA の一部がインプリメントされ、また JDBC ドライバーでも JTA の一部がインプリメントされます。

WebSphere Application Server 環境では、JDBC ドライバーによりこの環境が自動的に検出されるため、DB2 を JTA 対応に構成する必要はありません。

DB2 JDBC Type 2 Driver では、次の 2 つの DataSource クラスが用意されています。

- `COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource`
- `COM.ibm.db2.jdbc.DB2XADataSource`

IBM Data Server Driver for JDBC and SQLJ では、次の 2 つの DataSource クラスが用意されています。

- `com.ibm.db2.jcc.DB2ConnectionPoolDataSource`
- `com.ibm.db2.jcc.DB2XADataSource`



WebSphere Application Server では、データベースへのプール接続が提供されます。アプリケーションが分散トランザクションに関連する場合は、WebSphere Application Server で DB2 データ・ソースを定義する際に `com.ibm.db2.jdbc.DB2XADataSource` クラスを使用する必要があります。

WebSphere Application Server を DB2 と共に使用するよう構成する方法については、次のサイトにある WebSphere Application Server インフォメーション・センターを参照してください。

<http://www.ibm.com/software/webservers/appserv/library.html>

## JTA メソッドを使用する分散トランザクションの例

分散トランザクションには通常、様々な製造元からのデータ・ソースを含むことができる同じデータ・ソースまたは異なるデータ・ソースへの複数の接続が含まれます。

分散トランザクションについて説明する最善の方法は、それをローカル・トランザクションと比較することです。ローカル・トランザクションでは、JDBC アプリケーションはデータベースへの変更を永久的なものとし、以下のいずれかの方法で作業単位の終わりを示します。

- 1 つ以上の SQL ステートメントを実行した後で `Connection.commit` または `Connection.rollback` メソッドを呼び出す。
- 毎回の SQL ステートメントの後で変更をコミットするために、アプリケーションの開始時に `Connection.setAutoCommit(true)` メソッドを呼び出す。

図 59 では、ローカル・トランザクションを実行するコードを概説しています。

```
con1.setAutoCommit(false); // Set autocommit off
// execute some SQL
...
con1.commit();             // Commit the transaction
// execute some more SQL
...
con1.rollback();          // Roll back the transaction
con1.setAutoCommit(true); // Enable commit after every SQL statement
...
// Execute some more SQL, which is automatically committed after
// every SQL statement.
```

図 59. ローカル・トランザクションの例

対照的に、分散トランザクションに関与するアプリケーションは、分散トランザクション内で `Connection.commit`、`Connection.rollback`、または `Connection.setAutoCommit(true)` メソッドを呼び出すことができません。分散トランザクションでは、`Connection.commit` または `Connection.rollback` メソッドはトランザクション境界を示しません。その代わりに、アプリケーションはアプリケーション・サーバーにトランザクション境界を管理させます。

224 ページの図 60 は、分散トランザクションを使用するアプリケーションを示しています。例の中のコードが実行している間、アプリケーション・サーバーはこの同じ分散トランザクションの一部である他の EJB も実行します。すべての EJB が `utx.commit()` を呼び出した場合、分散トランザクション全体がアプリケーション・サーバーによってコミットされます。いずれかの EJB が失敗した場合、アプリケーション

オン・サーバーは分散トランザクションに関連するすべての EJB によって行われたすべての作業をロールバックします。

```
javax.transaction.UserTransaction utx;
// Use the begin method on a UserTransaction object to indicate
// the beginning of a distributed transaction.
utx.begin();
...
// Execute some SQL with one Connection object.
// Do not call Connection methods commit or rollback.
...
// Use the commit method on the UserTransaction object to
// drive all transaction branches to commit and indicate
// the end of the distributed transaction.

utx.commit();
...
```

図 60. アプリケーション・サーバーで実行する分散トランザクションの例

図 61 は、JTA メソッドを使用して分散トランザクションを実行するプログラムを示しています。このプログラムはトランザクション・マネージャーおよびトランザクション・アプリケーションとして機能します。2 つの異なるデータ・ソースへの 2 つの接続は、単一の分散トランザクションの下で SQL 作業を行います。

図 61. JTA を使用する分散トランザクションの例

```
class XASample
{
    javax.sql.XADataSource xaDS1;
    javax.sql.XADataSource xaDS2;
    javax.sql.XAConnection xaconn1;
    javax.sql.XAConnection xaconn2;
    javax.transaction.xa.XAResource xares1;
    javax.transaction.xa.XAResource xares2;
    java.sql.Connection conn1;
    java.sql.Connection conn2;

    public static void main (String args []) throws java.sql.SQLException
    {
        XASample xat = new XASample();
        xat.runThis(args);
    }
    // As the transaction manager, this program supplies the global
    // transaction ID and the branch qualifier. The global
    // transaction ID and the branch qualifier must not be
    // equal to each other, and the combination must be unique for
    // this transaction manager.
    public void runThis(String[] args)
    {
        byte[] gtrid = new byte[] { 0x44, 0x11, 0x55, 0x66 };
        byte[] bqual = new byte[] { 0x00, 0x22, 0x00 };
        int rc1 = 0;
        int rc2 = 0;

        try
        {
            javax.naming.InitialContext context = new javax.naming.InitialContext();
            /*
             * Note that javax.sql.XADataSource is used instead of a specific
             * driver implementation such as com.ibm.db2.jcc.DB2XADataSource.
            */
        }
    }
}
```

```

    */
xaDS1 = (javax.sql.XADataSource)context.lookup("checkingAccounts");
xaDS2 = (javax.sql.XADataSource)context.lookup("savingsAccounts");

// The XADataSource contains the user ID and password.
// Get the XAConnection object from each XADataSource
xaconn1 = xaDS1.getXAConnection();
xaconn2 = xaDS2.getXAConnection();

// Get the java.sql.Connection object from each XAConnection
conn1 = xaconn1.getConnection();
conn2 = xaconn2.getConnection();

// Get the XAResource object from each XAConnection
xares1 = xaconn1.getXAResource();
xares2 = xaconn2.getXAResource();
// Create the Xid object for this distributed transaction.
// This example uses the com.ibm.db2.jcc.DB2Xid implementation
// of the Xid interface. This Xid can be used with any JDBC driver
// that supports JTA.
javax.transaction.xa.Xid xid1 =
    new com.ibm.db2.jcc.DB2Xid(100, gtrid, bqual);

// Start the distributed transaction on the two connections.
// The two connections do NOT need to be started and ended together.
// They might be done in different threads, along with their SQL operations.
xares1.start(xid1, javax.transaction.xa.XAResource.TMNOFLAGS);
xares2.start(xid1, javax.transaction.xa.XAResource.TMNOFLAGS);
...
// Do the SQL operations on connection 1.
// Do the SQL operations on connection 2.
...
// Now end the distributed transaction on the two connections.
xares1.end(xid1, javax.transaction.xa.XAResource.TMSUCCESS);
xares2.end(xid1, javax.transaction.xa.XAResource.TMSUCCESS);

// If connection 2 work had been done in another thread,
// a thread.join() call would be needed here to wait until the
// connection 2 work is done.

try
{ // Now prepare both branches of the distributed transaction.
  // Both branches must prepare successfully before changes
  // can be committed.
  // If the distributed transaction fails, an XAException is thrown.
  rc1 = xares1.prepare(xid1);
  if(rc1 == javax.transaction.xa.XAResource.XA_OK)
  { // Prepare was successful. Prepare the second connection.
    rc2 = xares2.prepare(xid1);
    if(rc2 == javax.transaction.xa.XAResource.XA_OK)
    { // Both connections prepared successfully and neither was read-only.
      xares1.commit(xid1, false);
      xares2.commit(xid1, false);
    }
    else if(rc2 == javax.transaction.xa.XAException.XA_RDONLY)
    { // The second connection is read-only, so just commit the
      // first connection.
      xares1.commit(xid1, false);
    }
  }
}
else if(rc1 == javax.transaction.xa.XAException.XA_RDONLY)
{ // SQL for the first connection is read-only (such as a SELECT).
  // The prepare committed it. Prepare the second connection.
  rc2 = xares2.prepare(xid1);
  if(rc2 == javax.transaction.xa.XAResource.XA_OK)
  { // The first connection is read-only but the second is not.
    // Commit the second connection.

```

```

        xares2.commit(xid1, false);
    }
    else if(rc2 == javax.transaction.xa.XAException.XA_RDONLY)
    { // Both connections are read-only, and both already committed,
      // so there is nothing more to do.
    }
}
}
    catch (javax.transaction.xa.XAException xae)
{ // Distributed transaction failed, so roll it back.
  // Report XAException on prepare/commit.
  System.out.println("Distributed transaction prepare/commit failed. " +
    "Rolling it back.");
  System.out.println("XAException error code = " + xae.errorCode);
  System.out.println("XAException message = " + xae.getMessage());
  xae.printStackTrace();
  try
  {
    xares1.rollback(xid1);
  }
  catch (javax.transaction.xa.XAException xae1)
  { // Report failure of rollback.
    System.out.println("distributed Transaction rollback xares1 failed");
    System.out.println("XAException error code = " + xae1.errorCode);
    System.out.println("XAException message = " + xae1.getMessage());
  }
  try
  {
    xares2.rollback(xid1);
  }
  catch (javax.transaction.xa.XAException xae2)
  { // Report failure of rollback.
    System.out.println("distributed Transaction rollback xares2 failed");
    System.out.println("XAException error code = " + xae2.errorCode);
    System.out.println("XAException message = " + xae2.getMessage());
  }
}

try
{
  conn1.close();
  xaconn1.close();
}
catch (Exception e)
{
  System.out.println("Failed to close connection 1: " + e.toString());
  e.printStackTrace();
}
try
{
  conn2.close();
  xaconn2.close();
}
catch (Exception e)
{
  System.out.println("Failed to close connection 2: " + e.toString());
  e.printStackTrace();
}
}
catch (java.sql.SQLException sqe)
{
  System.out.println("SQLException caught: " + sqe.getMessage());
  sqe.printStackTrace();
}
catch (javax.transaction.xa.XAException xae)
{
  System.out.println("XA error is " + xae.getMessage());
  xae.printStackTrace();
}

```

```

    }
    catch (javax.naming.NamingException nme)
    {
        System.out.println(" Naming Exception: " + nme.getMessage());
    }
}
}

```

**推奨事項:** パフォーマンスを向上させるには、別の分散トランザクションまたはローカル・トランザクションを開始する前に、分散トランザクションを完了してください。

## XAResource インスタンスのトランザクション・タイムアウト値の設定

`XAResource.setTransactionTimeout` メソッドを使用して、分散トランザクションのターゲットになっている DB2 データベースでデッドロックの回数を減らします。

準備ができなくても終了する DB2 Database for Linux, UNIX, and Windows の分散トランザクションは、未確定トランザクションではありません。したがって、トランザクション・マネージャーは、そのトランザクションを回復できませんし、DB2 リソース・マネージャーも、そのトランザクションを未確定トランザクションのリストに入れません。DB2 リソース・マネージャーは、そのトランザクションをすぐにロールバックしないで、すべてのデータベース接続が解放されるまで待機します。その間はアクティビティーがないまま、そのトランザクションがデータベースに対するロックを保持することになります。トランザクション・マネージャーがすべてのデータベース接続を切断してロールバックを可能にしなければ、終了したトランザクションがいつまでもデータベース・レコードをロックし続けてしまいます。ロックされているレコードに別のアプリケーションがアクセスしようとする、デッドロックが発生する可能性があります。

分散トランザクションと IBM Data Server Driver for JDBC and SQLJ Type 4 接続を使用する Java アプリケーションでは、`XAResource.setTransactionTimeout` メソッドを呼び出してトランザクションのタイムアウト値を設定することによって、トランザクションがデータベースのロックをいつまでも保持する事態を回避できます。そのためには、以下の手順を実行します。

1. DB2 Database for Linux, UNIX, and Windows インスタンスで以下のコマンドを実行して、インスタンスがタイムアウト値をチェックするようにします。

```
DB2 UPDATE DBM CFG USING RESYNC_INTERVAL seconds
```

*seconds* は、トランザクションに設定する最小タイムアウト値より小さい値にする必要があります。

2. アプリケーションで `XAResource` オブジェクトを作成してから、`XAResource.setTransactionTimeout` メソッドを呼び出してタイムアウト値を設定します。

現在のタイムアウト値をチェックするには、`XAResource.getTransactionTimeout` を呼び出します。

---

## Enterprise Java Beans

Enterprise Java Beans アーキテクチャーは、コンポーネント・ベースの分散ビジネス・アプリケーションを開発およびデプロイするためのコンポーネント・アーキテクチャーです。

Enterprise Java Beans アーキテクチャーを使用して作成されたアプリケーションは、一度作成すれば、Enterprise Java Beans 仕様をサポートするすべてのサーバー・プラットフォームにデプロイすることができます。Java 2 Platform, Enterprise Edition (J2EE) アプリケーションによって、セッション Bean とエンティティ Bean が含まれた Enterprise Java Beans (EJB) が使用され、サーバー・サイドのビジネス・コンポーネントがインプリメントされます。

セッション Bean はビジネス・サービスを表し、ユーザー間では共有されません。エンティティ Bean とは、永続データを表すマルチユーザーの分散トランザクション・オブジェクトです。EJB アプリケーションのトランザクション境界は、コンテナ管理トランザクションまたは Bean 管理トランザクションを指定することによって設定できます。

サンプル・プログラムの AccessEmployee.ear により、Enterprise Java Beans が使用され、データ・ソースにアクセスする J2EE アプリケーションがインプリメントされます。このサンプルは SQLLIB/samples/websphere ディレクトリーにあります。

この EJB サンプル・アプリケーションにより、次の 2 つのビジネス・サービスが提供されます。1 つのサービスにより、ユーザーは従業員の従業員番号でその従業員に関する情報 (**sample** データベースの EMPLOYEE 表に保管されている) にアクセスできます。もう 1 つのサービスにより、ユーザーは従業員番号のリストを検索して、従業員データの照会に使用する従業員番号を取得できます。

下のサンプルでは EJB が使用されて、データ・ソースにアクセスする J2EE アプリケーションがインプリメントされます。このサンプルでは、共通で使用される GUI アーキテクチャーの Model-View-Controller (MVC) アーキテクチャーが使用されます。JSP は、View (表示コンポーネント) のインプリメントに使用されます。サーブレットは、このサンプルで Controller (コントローラー) として機能します。サーブレットによりワークフローが制御され、ユーザーの要求が EJB によってインプリメントされた Model (モデル) に委任されます。このサンプルの Model コンポーネントは、2 つの EJB (1 つのセッション Bean と 1 つのエンティティ Bean) によって構成されます。コンテナ管理パーシスタンス (CMP) Bean である Employee により、サンプル・データベースの EMPLOYEE 表にある永続データを表す分散トランザクション・オブジェクトが表されます。「コンテナ管理パーシスタンス」という用語は、エンティティ Bean で必要なすべてのデータベース・アクセスが EJB コンテナによって処理されることを意味します。この Bean のコードには、データベース・アクセス (SQL) の呼び出しが含まれていません。このため、この Bean のコードは特定の永続ストレージ機構 (データベース) に束縛されません。セッション Bean の AccessEmployee はエンティティ Bean の Façade (ファサード) として機能し、統一的なクライアント・アクセス戦略が提供されます。この Façade 設計により、EJB クライアントとエンティティ Bean 間のネットワーク・トラフィックが低減し、エンティティ Bean が EJB クライアントから直接アクセスされる場合よりも分散トランザクションが効率的になります。データベー

ス・サーバーへのアクセスは、セッション Bean またはエンティティ Bean から提供できます。このサンプル・アプリケーションの 2 つのサービスでは、データベース・サーバーへのアクセス方法が両方とも示されています。次の最初のサービスでは、エンティティ Bean が使用されています。

```
//=====
// This method returns an employee's information by
// interacting with the entity bean located by the
// provided employee number
public EmployeeInfo getEmployeeInfo(String empNo)
throws java.rmi.RemoteException
{
    Employee employee = null;
    try
    {
        employee = employeeHome.findByPrimaryKey(new EmployeeKey(empNo));
        EmployeeInfo empInfo = new EmployeeInfo(empNo);
        //set the employee's information to the dependent value object
        empInfo.setEmpno(employee.getEmpno());
        empInfo.setFirstName (employee.getFirstName());
        empInfo.setMidInit(employee.getMidInit());
        empInfo.setLastName(employee.getLastName());
        empInfo.setWorkDept(employee.getWorkDept());
        empInfo.setPhoneNo(employee.getPhoneNo());
        empInfo.setHireDate(employee.getHireDate());
        empInfo.setJob(employee.getJob());
        empInfo.setEdLevel (employee.getEdLevel());
        empInfo.setSex(employee.getSex());
        empInfo.setBirthDate(employee.getBirthDate());
        empInfo.setSalary(employee.getSalary());
        empInfo.setBonus(employee.getBonus());
        empInfo.setComm(employee.getComm());
        return empInfo;
    }
    catch (java.rmi.RemoteException rex)
    {
        .....
    }
}
```

従業員番号を表示する 2 番目のサービスでは、セッション Bean の AccessEmployee によりデータベース表が直接アクセスされています。

```
//=====
* Get the employee number list.
* @return Collection
*/
public Collection getEmpNoList()
{
    ResultSet rs = null;
    PreparedStatement ps = null;
    Vector list = new Vector();
    DataSource ds = null;
    Connection con = null;
    try
    {
        ds = getDataSource();
        con = ds.getConnection();
        String schema = getEnvProps(DBSchema);
        String query = "Select EMPNO from " + schema + ".EMPLOYEE";
        ps = con.prepareStatement(query);
        ps.executeQuery();
        rs = ps.getResultSet();
        EmployeeKey pk;
        while (rs.next())
        {
            pk = new EmployeeKey();
            pk.employeeId = rs.getString(1);
        }
    }
}
```



```
list.addElement(pk.employeeId);  
}  
rs.close();  
return list;
```

---

## 第 11 章 JDBC および SQLJ 接続プールのサポート

接続プールは、JDBC の DataSource サポートの一部であり、IBM Data Server Driver for JDBC and SQLJ によってサポートされます。

IBM Data Server Driver for JDBC and SQLJ では、WebSphere Application Server または他のアプリケーション・サーバーによって使用されるプール接続のファクトリーが提供されます。このプールは、実際にはアプリケーション・サーバーにより実行されます。接続プールは、JDBC または SQLJ アプリケーションには全く認識されません。

接続プールとは、DB2 スレッドに相当する物理データ・ソース接続をキャッシュするためのフレームワークです。JDBC により物理データ・ソース接続が再利用されると、`java.sql.Connection` オブジェクトの作成とそれに続く終了に必要な負荷のかかる操作が最小化されます。

接続プールがない場合、各 `java.sql.Connection` オブジェクトがデータ・ソースへの物理接続を表します。アプリケーションによりデータ・ソースへの接続が確立されると、DB2 によりデータ・ソースへの新規の物理接続が作成されます。アプリケーションにより `java.sql.Connection.close` メソッドが呼び出されると、DB2 により、このデータ・ソースへの物理接続が終了されます。

これとは対照的に、接続プールでは、`java.sql.Connection` オブジェクトが物理データ・ソース接続を一時的および論理的に表すものです。物理データ・ソース接続は、論理 `java.sql.Connection` インスタンスによって連続的に再利用されます。アプリケーションでは、接続プール・サポートがない場合に `java.sql.Connection` オブジェクトを使用するときと全く同様に、論理 `java.sql.Connection` オブジェクトを使用できます。

接続プールでは、JDBC アプリケーションにより `DataSource.getConnection` メソッドが呼び出されると、データ・ソースで適切な物理接続が存在するかどうか判定されます。適切な物理接続が既存する場合、データ・ソースにより `java.sql.Connection` インスタンスがアプリケーションに返されます。JDBC アプリケーションにより `java.sql.Connection.close` メソッドが呼び出されても、JDBC では物理データ・ソース接続はクローズされません。その代わりに、JDBC では `Statement` オブジェクトや `ResultSet` オブジェクトなどの JDBC リソースのみがクローズされます。データ・ソースにより、物理接続が再利用のために接続プールに返されます。

接続プールは、同種 または異種 にすることができます。

同種プールの場合、1 つの接続プールからのすべての `Connection` オブジェクトに同じプロパティがあります。DataSource で作成された最初の論理 `Connection` には、DataSource に定義されたプロパティがあります。ただし、アプリケーションでそれらのプロパティを変更できます。Connection が接続プールに返されると、アプリケーション・サーバーまたはプール・モジュールにより、これらのプロパティは元の値にリセットされます。ただし、アプリケーション・サーバーまたはプール・モジュールにより、変更されたプロパティがリセットされない場合があります。

ます。 JDBC ドライバーでは、これらのプロパティは変更されません。したがって、アプリケーション・サーバーまたはプール・モジュールの設計に応じて、再利用される論理 `Connection` が、`DataSource` に定義されたものと同じプロパティを持つこともあれば、異なるプロパティを持つこともあります。

異種プールでは、異なるプロパティの `Connection` オブジェクトが、同じ接続プールを共有することができます。

---

## 第 12 章 JDBC 接続コンセントレーターおよび Sysplex ワークロード・バランシング

IBM Data Server Driver for JDBC and SQLJ Type 4 接続を使用して DB2 for z/OS サーバーにアクセスする Java アプリケーションでは、接続コンセントレーターと Sysplex ワークロード・バランシング機能を利用することができます。

IBM Data Server Driver for JDBC and SQLJ の接続コンセントレーターと Sysplex ワークロード・バランシング機能は、DB2 Connect の接続コンセントレーターと Sysplex ワークロード・バランシング機能に類似しています。

IBM Data Server Driver for JDBC and SQLJ の接続コンセントレーター機能により、DB2 for z/OS データ・ソースによる多数のクライアント・アプリケーションのサポートに必要なリソースを削減することができます。IBM Data Server Driver for JDBC and SQLJ の接続コンセントレーター機能によって、多数の接続オブジェクトが同じ物理接続を使用することができ、これによりデータ・ソースへの物理接続の総数が減少します。

IBM Data Server Driver for JDBC and SQLJ の Sysplex ワークロード・バランシングにより、データ共有グループの可用性を高めることができます。Sysplex ワークロード・バランシングが使用可能になっている場合は、ドライバーによりデータ共有グループのメンバーについての状況情報が頻繁に取得されます。ドライバーによりこの情報が使用され、次のトランザクションがどのデータ共有メンバーに転送されるべきかが判断されます。Sysplex ワークロード・バランシングを使用すれば、DB2 for z/OS サーバーおよび Workload Manager for z/OS (WLM) により、作業がデータ共有グループの各メンバー間で効率よく分散され、またデータ共有グループのあるメンバーに障害が発生した場合は、その作業が別のメンバーに転送されます。

IBM Data Server Driver for JDBC and SQLJ はトランスポート・オブジェクト およびグローバル・トランスポート・オブジェクト・プールを使用して、接続コンセントレーターおよび Sysplex ワークロード・バランシングをサポートします。データベース・ソースへの各物理接続ごとに、トランスポート・オブジェクトが 1 つあります。接続コンセントレーターと Sysplex ワークロード・バランシングを使用可能にした場合は、トランスポート・オブジェクトの最大数を設定することによって、任意の時点でデータベース・ソースへの物理接続の最大数を設定します。

ドライバー・レベルでは、IBM Data Server Driver for JDBC and SQLJ 構成プロパティーを使用して、トランスポート・オブジェクトの数に対する限度を設定します。

接続レベルでは、DataSource プロパティーを使用して、IBM Data Server Driver for JDBC and SQLJ の接続コンセントレーターと Sysplex ワークロード・バランシングを使用可能および使用不可にし、トランスポート・オブジェクトの数に対する限度を設定します。これらのプロパティーは、DataSource インターフェースまたは DriverManager インターフェースを使用して接続を取得する際に設定することができます。

次のいずれかの方法で、グローバル・トランスポート・オブジェクト・プールをモニターすることができます。

- IBM Data Server Driver for JDBC and SQLJ の構成プロパティを使用して開始するトレースの使用
- アプリケーション・プログラミング・インターフェースの使用

---

## IBM Data Server Driver for JDBC and SQLJ 接続コンセントレーターを使用可能にする例

IBM Data Server Driver for JDBC and SQLJ の接続コンセントレーター機能と Sysplex ワークロード・バランシング機能を WebSphere Application Server と一緒に使用するには、まずそれらの機能を構成する必要があります。

サーバー要件:

- WLM for z/OS
- DB2 UDB for OS/390 and z/OS バージョン 7 以降 (データ共用セットアップ)

データ共用グループのすべてのメンバーで特殊レジスターのデフォルト値を統一する必要があります。IBM Data Server Driver for JDBC and SQLJ は、データ共用グループの各メンバーのロード・バランスを調整するときに、ユーザーの接続を 1 つのメンバーから別のメンバーに移動するからです。ユーザーが元のデータ共用メンバーで特殊レジスターの値を設定していれば、ドライバは、すべての特殊レジスターをデフォルト値にリセットしてから、特殊レジスターの変更内容を新しいメンバーに適用します。ところが、IBM Data Server Driver for JDBC and SQLJ は、すべてのメンバーのデフォルト値を確認できるわけではありません。2 つのメンバーのデフォルト値がそれぞれ異なっていれば、どちらのメンバーに対して SQL ステートメントを実行したかによって、結果が異なる場合があります。

クライアント要件:

- IBM Data Server Driver for JDBC and SQLJ (フィックスパック 10 レベル)
- WebSphere Application Server バージョン 5.1 以降

IBM Data Server Driver for JDBC and SQLJ の接続コンセントレーター機能と Sysplex ワークロード・バランシング機能を WebSphere Application Server と一緒に使用することを可能にするための手順の例を次に示します。ここで指定する値を推奨値として示す意図はありません。以下のような要因に基づいて値を判別する必要があります。

- システム・リソースの可用性
  - 使用可能な物理接続の数
  - 接続オブジェクトと転送オブジェクトの望ましい比率
1. IBM Data Server Driver for JDBC and SQLJ が接続コンセントレーターと Sysplex ワークロード・バランシングをサポートできる正しいレベルであるかどうかを確認します。そのための手順は、以下のとおりです。
    - a. コマンド行プロセッサで以下のコマンドを実行します。

```
java com.ibm.db2.jcc.DB2Jcc -version
```
    - b. 出力で以下のような行を見つけて、*nnn* が 2.7 以降になっているかどうかを確認します。

2. IBM Data Server Driver for JDBC and SQLJ の構成プロパティーを設定し、このドライバーの下で作成されたすべての DataSource インスタンスまたは Connection インスタンスで接続コンцентрーターまたは Sysplex ワークロード・バランシングを使用可能にします。DB2JccConfiguration.properties ファイルの構成プロパティーを設定する手順は、以下のとおりです。
  - a. DB2JccConfiguration.properties ファイルを作成するか、既存の DB2JccConfiguration.properties ファイルを編集します。
  - b. 以下の構成プロパティーを設定します。
    - db2.jcc.minTransportObjects
    - db2.jcc.maxTransportObjects
    - db2.jcc.maxTransportObjectWaitTime
    - db2.jcc.dumpPool
    - db2.jcc.dumpPoolStatisticsOnScheduleFile
 まずは以下のような設定で始めてみてください。
 

```
db2.jcc.minTransportObjects=0
db2.jcc.maxTransportObjects=1500
db2.jcc.maxTransportObjectWaitTime=-1
db2.jcc.dumpPool=0
db2.jcc.dumpPoolStatisticsOnScheduleFile=/home/WAS/logs/srv1/poolstats
```
  - c. DB2JccConfiguration.properties のディレクトリー・パスを WebSphere Application Server IBM Data Server Driver for JDBC and SQLJ のクラスパスに追加します。
3. IBM Data Server Driver for JDBC and SQLJ のデータ・ソース・プロパティーを設定し、接続コンцентрーターまたは Sysplex ワークロード・バランシングを使用可能にします。

WebSphere Application Server 管理コンソールで、アプリケーションからデータ・ソースに接続するために使用するデータ・ソース・プロパティーを設定します。

- enableSysplexWLB
- enableConnectionConcentrator
- maxTransportObjects

接続コンцентрーター機能と Sysplex ワークロード・バランシング機能の両方を使用するとしましょう。まずは以下のような設定で始めてみてください。

表 27. IBM Data Server Driver for JDBC and SQLJ の接続コンцентрーターと Sysplex ワークロード・バランシングのためのデータ・ソース・プロパティー設定の例

プロパティー	設定
enableSysplexWLB	true <sup>1</sup>
maxTransportObjects	100

注:

1. enableSysplexWLB が true に設定されていれば、enableConnectionConcentrator もデフォルトで true に設定されます。

4. WebSphere Application Server を再始動します。

---

## IBM Data Server Driver for JDBC and SQLJ 接続コンセントレーター および Sysplex ワークロード・บาลランシングのモニター技法

IBM Data Server Driver for JDBC and SQLJ の接続コンセントレーターと Sysplex ワークロード・บาลランシングをモニターするには、グローバル・トランスポート・オブジェクト・プールをモニターする必要があります。

次のいずれかの方法で、グローバル・トランスポート・オブジェクト・プールをモニターすることができます。

- IBM Data Server Driver for JDBC and SQLJ の構成プロパティを設定して開始するトレースの使用
- アプリケーション・プログラミング・インターフェースの使用

### グローバル・トランスポート・オブジェクト・プールのモニターのための構成プロパティ

db2.jcc.dumpPool、db2.jcc.dumpPoolStatisticsOnSchedule、および db2.jcc.dumpPoolStatisticsOnScheduleFile 構成プロパティにより、グローバル・トランスポート・オブジェクト・プールのトレースが制御されます。

例えば、以下の一組の構成プロパティの設定では、Sysplex エラー・メッセージとダンプ・プール・エラー・メッセージが 60 秒ごとに /home/WAS/logs/srv1/poolstats というファイルに書き込まれます。

```
db2.jcc.dumpPool=DUMP_SYSPLEX_MSG|DUMP_POOL_ERROR
db2.jcc.dumpPoolStatisticsOnSchedule=60
db2.jcc.dumpPoolStatisticsOnScheduleFile=/home/WAS/logs/srv1/poolstats
```

プール統計ファイル内の項目は次のようになります。

```
time Scheduled PoolStatistics npr:2575 nsr:2575 lwroc:439 hwroc:1764 coc:372
aoc:362 rmoc:362 nbr:2872 tbt:857520 tpo:10
```

各フィールドの意味は次のとおりです。

#### npr

プールの作成時以降にプールに対して IBM Data Server Driver for JDBC and SQLJ から出された要求の総数。

#### nsr

プールの作成時以降にプールに対して IBM Data Server Driver for JDBC and SQLJ から出されて正常に完了した要求の数。正常に完了した要求とは、プールからオブジェクトが返されたことを意味します。

#### lwroc

再利用されたがプール内に存在しなかったオブジェクトの数。これは、接続オブジェクトによってトランスポート・オブジェクトがトランザクション境界で解放された場合に発生することがあります。この接続オブジェクトで、そのトランスポート・オブジェクトが後で必要になった際に、元のトランスポート・オブジェクトが他のどの接続オブジェクトによってもまだ使用されていない場合は、この接続オブジェクトでそのトランスポート・オブジェクトを使用することができます。



**hwroc**

プールから再利用されたオブジェクトの数。

**coc**

プールの作成時以降に IBM Data Server Driver for JDBC and SQLJ により作成されたオブジェクトの数。

**aooc**

db2.jcc.maxTransportObjectIdleTime で指定されていたアイドル時間を超過したためにプールから削除されたオブジェクトの数。

**rmoc**

プールの作成時以降にプールから削除されたオブジェクトの数。

**nbr**

プールに対して IBM Data Server Driver for JDBC and SQLJ から出されたが、プールがその最大容量に達したためプールによってブロックされた要求の数。ブロックされた要求が正常に完了する可能性があるのは、db2.jcc.maxTransportObjectWaitTime を超過して例外がスローされる前にオブジェクトがプールに返された場合です。

**tbt** プールによってブロックされた要求のミリ秒単位の合計時間。アプリケーションによって複数のスレッドが使用される場合は、この時間は、アプリケーションの経過実行時間よりはるかに長くなる可能性があります。

**sbt**

1 つのスレッドがプールからのトランスポート・オブジェクトの取得を待機したミリ秒単位の最短時間。この時間が 1 ミリ秒未満の場合、このフィールドの値はゼロになります。

**lbt** 1 つのスレッドがプールからのトランスポート・オブジェクトの取得を待機したミリ秒単位の最長時間。

**abt**

各スレッドがプールからのトランスポート・オブジェクトの取得を待機したミリ秒単位の平均時間。この値は tbt/nbr です。

**tpo**

プール内に現在存在するオブジェクトの数。

## グローバル・トランスポート・オブジェクト・プールのモニターのためのアプリケーション・プログラミング・インターフェース

グローバル・トランスポート・オブジェクト・プールに関する統計を収集するためのアプリケーションを作成することができます。このようなアプリケーションでは、オブジェクトが DB2PoolMonitor クラスに作成され、プールに関する情報を取得するためにメソッドが呼び出されます。

例えば、次のコードでは、グローバル・トランスポート・オブジェクト・プールのモニター用のオブジェクトが作成されます。

```
import com.ibm.db2.jcc.DB2PoolMonitor;  
DB2PoolMonitor transportObjectPoolMonitor =  
    DB2PoolMonitor.getPoolMonitor (DB2PoolMonitor.TRANSPORT_OBJECT);
```

DB2PoolMonitor オブジェクトの作成後に、DB2PoolMonitor クラス内のメソッドを使用してプールをモニターすることができます。

## 第 13 章 JDBC および SQLJ の参照情報

JDBC および SQLJ の IBM のインプリメンテーションでは、JDBC および SQLJ アプリケーションを開発するための数多くのアプリケーション・プログラミング・インターフェース、プロパティ、およびコマンドが提供されています。

### Java アプリケーションでデータベース・データ・タイプにマップするデータ・タイプ

効果的な JDBC および SQLJ プログラムを作成するには、Java データ・タイプと表列データ・タイプの間で最適なマッピングを使用する必要があります。

以下の表は、DB2 Database for Linux, UNIX, and Windows、DB2 for z/OS、または IBM Informix Dynamic Server (IDS) システムの場合の、Java データ・タイプの JDBC およびデータベース・データ・タイプへのマッピングを要約したものです。

#### 表列を更新するためのデータ・タイプ

以下の表は、JDBC プログラム内の `PreparedStatement.setXXX` または `ResultSet.updateXXX` メソッド、および SQLJ プログラム内の入力ホスト式の場合の、Java データ・タイプのデータベース・データ・タイプへのマッピングを要約したものです。複数の Java データ・タイプがリストされている場合、最初のデータ・タイプが推奨されるデータ・タイプです。

表 28. データベース表を更新するための、Java データ・タイプのデータベース・サーバー・データ・タイプへのマッピング

Java データ・タイプ	データベース・データ・タイプ
short	SMALLINT
boolean <sup>1</sup> 、byte <sup>1</sup> 、java.lang.Boolean	SMALLINT
int、java.lang.Integer	INTEGER
long、java.lang.Long	BIGINT
float、java.lang.Float	REAL
double、java.lang.Double	DOUBLE
java.math.BigDecimal	DECIMAL( <i>p,s</i> ) <sup>2</sup>
java.math.BigDecimal	DECFLOAT( <i>n</i> ) <sup>3、4</sup>
java.lang.String	CHAR( <i>n</i> ) <sup>5</sup>
java.lang.String	GRAPHIC( <i>m</i> ) <sup>6</sup>
java.lang.String	VARCHAR( <i>n</i> ) <sup>7</sup>
java.lang.String	VARGRAPHIC( <i>m</i> ) <sup>8</sup>
java.lang.String	CLOB <sup>9</sup>
java.lang.String	XML
byte[]	CHAR( <i>n</i> ) FOR BIT DATA <sup>5</sup>
byte[]	VARCHAR( <i>n</i> ) FOR BIT DATA <sup>7</sup>
byte[]	BINARY( <i>n</i> ) <sup>5</sup>

表 28. データベース表を更新するための、Java データ・タイプのデータベース・サーバー・データ・タイプへのマッピング (続き)

Java データ・タイプ	データベース・データ・タイプ
byte[]	VARBINARY( <i>n</i> ) <sup>7</sup>
byte[]	BLOB <sup>9</sup>
byte[]	ROWID
byte[]	XML
java.sql.Blob	BLOB
java.sql.Blob	XML
java.sql.Clob	CLOB
java.sql.Clob	DBCLOB <sup>9</sup>
java.sql.Clob	XML
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP
java.io.ByteArrayInputStream	BLOB
java.io.StringReader	CLOB
java.io.ByteArrayInputStream	CLOB
java.io.InputStream	XML
com.ibm.db2.jcc.DB2RowID (非推奨)	ROWID
java.sql.RowId	ROWID
com.ibm.db2.jcc.DB2Xml (非推奨)	XML
java.sql.SQLXML	XML

**注:**

1. データベース・サーバーには、Java ブール値またはバイト・データ・タイプに厳密に相当するものではありませんが、最も近いものが SMALLINT です。
2. *p* は 10 進数精度で、*s* は表列のスケールです。

java.math.BigDecimal 列が DECIMAL 列にマップするように金融アプリケーションを設計するとします。DECIMAL 列の精度およびスケールがわかっている場合、DECIMAL 列のデータを java.math.BigDecimal 変数の持つデータで更新すると、他のデータ・タイプの組み合わせを使用する場合よりもパフォーマンスが向上します。

3. *n*=16 または *n*=34。
4. DECFLOAT は、DB2 バージョン 9.1 for z/OS 以降のデータベース・サーバー、または DB2 V9.5 for Linux, UNIX, and Windows 以降のデータベース・サーバーへの接続に有効です。DECFLOAT を使用するには、SDK for Java バージョン 5 (1.5) 以降が必要です。
5. *n*≤254。
6. *m*≤127。
7. *n*≤32672。
8. *m*≤16336。
9. このマッピングは、データベース・サーバーが列のデータ・タイプを判別できる場合にのみ有効です。

## 表列を検索するためのデータ・タイプ

以下の表は、JDBC プログラム内の `ResultSet.getXXX` メソッド、および SQLJ プログラム内のイテレータの場合の、DB2 または IDS データ・タイプの Java データ・タイプへのマッピングを要約したものです。この表では、Java 数値ラッパー・オブジェクト・タイプはリストしていません。これは、`ResultSet.getObject` を使用して検索されます。

表 29. データベース・サーバー表からデータを検索するための、データベース・サーバー・データ・タイプの Java データ・タイプへのマッピング

SQL データ・タイプ	推奨される Java データ・タイプ または Java オブジェクト・タイプ	その他のサポートされる Java データ・タイプ
SMALLINT	short	byte, int, long, float, double, java.math.BigDecimal, boolean, java.lang.String
INTEGER	int	short, byte, long, float, double, java.math.BigDecimal, boolean, java.lang.String
BIGINT	long	int, short, byte, float, double, java.math.BigDecimal, boolean, java.lang.String
DECIMAL( <i>p,s</i> ) or NUMERIC( <i>p,s</i> )	java.math.BigDecimal	long, int, short, byte, float, double, boolean, java.lang.String
DECFLOAT( <i>n</i> ) <sup>1,2</sup>	java.math.BigDecimal	long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.lang.String
REAL	float	long, int, short, byte, double, java.math.BigDecimal, boolean, java.lang.String
DOUBLE	double	long, int, short, byte, float, java.math.BigDecimal, boolean, java.lang.String
CHAR( <i>n</i> )	java.lang.String	long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.io.InputStream, java.io.Reader
VARCHAR( <i>n</i> )	java.lang.String	long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.io.InputStream, java.io.Reader
CHAR( <i>n</i> ) FOR BIT DATA	byte[]	java.lang.String, java.io.InputStream, java.io.Reader
VARCHAR( <i>n</i> ) FOR BIT DATA	byte[]	java.lang.String, java.io.InputStream, java.io.Reader
BINARY( <i>n</i> )	byte[]	なし

表 29. データベース・サーバー表からデータを検索するための、データベース・サーバー・データ・タイプの Java データ・タイプへのマッピング (続き)

SQL データ・タイプ	推奨される Java データ・タイプ または Java オブジェクト・タイプ	その他のサポートされる Java データ・タイプ
VARBINARY( <i>n</i> )	byte[]	なし
GRAPHIC( <i>m</i> )	java.lang.String	long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.io.InputStream, java.io.Reader
VARGRAPHIC( <i>m</i> )	java.lang.String	long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.io.InputStream, java.io.Reader
CLOB( <i>n</i> )	java.sql.Clob	java.lang.String
BLOB( <i>n</i> )	java.sql.Blob	byte[] <sup>3</sup>
DBCLOB( <i>m</i> )	厳密な対応なし。java.sql.Clob を使用。	
ROWID	java.sql.RowId	byte[], com.ibm.db2.jcc.DB2RowID (非推奨)
XML	java.sql.SQLXML	byte[], java.lang.String, java.io.InputStream, java.io.Reader
DATE	java.sql.Date	java.sql.String, java.sql.Timestamp
TIME	java.sql.Time	java.sql.String, java.sql.Timestamp
TIMESTAMP	java.sql.Timestamp	java.sql.String, java.sql.Date, java.sql.Time, java.sql.Timestamp

注:

1.  $n=16$  または  $n=34$ 。
2. DECFLOAT は、DB2 バージョン 9.1 for z/OS 以降のデータベース・サーバー、または DB2 V9.5 for Linux, UNIX, and Windows 以降のデータベース・サーバーへの接続に有効です。DECFLOAT を使用するには、SDK for Java バージョン 5 (1.5) 以降が必要です。
3. このマッピングは、データベース・サーバーが列のデータ・タイプを判別できる場合にのみ有効です。

## ストアド・プロシージャおよびユーザー定義関数を呼び出すためのデータ・タイプ

以下の表は、ユーザー定義関数およびストアド・プロシージャ・パラメーターを呼び出すための、Java データ・タイプの JDBC データ・タイプおよび DB2 または IDS データ・タイプへのマッピングを要約したものです。Java データ・タイプの JDBC データ・タイプへのマッピングは、JDBC プログラム内の CallableStatement.registerOutParameter メソッドを対象としています。Java データ・タイプのデータベース・サーバー・データ・タイプへのマッピングは、ストアド・プロシージャまたはユーザー定義関数呼び出しでのパラメーターを対象としています。

複数の Java データ・タイプが以下の表でリストされている場合、最初のデータ・タイプが推奨されるデータ・タイプです。

表 30. ストアド・プロシージャおよびユーザー定義関数を呼び出すための Java、JDBC、および SQL データ・タイプのマッピング

Java データ・タイプ	JDBC データ・タイプ	SQL データ・タイプ
boolean <sup>1</sup>	BIT	SMALLINT
byte <sup>1</sup>	TINYINT	SMALLINT
short、 java.lang.Short	SMALLINT	SMALLINT
int、 java.lang.Integer	INTEGER	INTEGER
long	BIGINT	BIGINT
float、 java.lang.Float	REAL	REAL
float、 java.lang.Float	FLOAT	REAL
double、 java.lang.Double	DOUBLE	DOUBLE
java.math.BigDecimal	NUMERIC	DECIMAL
java.math.BigDecimal	DECIMAL	DECIMAL
java.lang.String	CHAR	CHAR
java.lang.String	CHAR	GRAPHIC
java.lang.String	VARCHAR	VARCHAR
java.lang.String	VARCHAR	VARGRAPHIC
java.lang.String	LONGVARCHAR	VARCHAR
java.lang.String	VARCHAR	CLOB
java.lang.String	LONGVARCHAR	CLOB
java.lang.String	CLOB	CLOB
byte[]	BINARY	CHAR FOR BIT DATA
byte[]	VARBINARY	VARCHAR FOR BIT DATA
byte[]	BINARY	BINARY
byte[]	VARBINARY	VARBINARY
byte[]	LONGVARBINARY	VARCHAR FOR BIT DATA
byte[]	VARBINARY	BLOB <sup>3</sup>
byte[]	LONGVARBINARY	BLOB <sup>3</sup>
java.sql.Date	DATE	DATE
java.sql.Time	TIME	TIME
java.sql.Timestamp	TIMESTAMP	TIMESTAMP
java.sql.Blob	BLOB	BLOB
java.sql.Clob	CLOB	CLOB
java.sql.Clob	CLOB	DBCLOB
java.io.ByteArrayInputStream	なし	BLOB
java.io.StringReader	なし	CLOB
java.io.ByteArrayInputStream	なし	CLOB
com.ibm.db2.jcc.DB2RowID (非推奨)	com.ibm.db2.jcc.DB2Types.ROWID	ROWID



表 30. ストアド・プロシージャおよびユーザー定義関数を呼び出すための Java、JDBC、および SQL データ・タイプのマッピング (続き)

Java データ・タイプ	JDBC データ・タイプ	SQL データ・タイプ
java.sql.RowId	java.sql.Types.ROWID	ROWID
com.ibm.db2.jcc.DB2Xml (非推奨)	com.ibm.db2.jcc.DB2Types.XML	XML AS CLOB
java.sql.SQLXML	java.sql.Types.SQLXML	XML AS CLOB
java.sql.Array <sup>4</sup>	java.sql.Types.ARRAY	ARRAY

注:

- SMALLINT パラメーターで定義されたストアド・プロシージャまたはユーザー定義関数は、ブール・パラメーターまたはバイト・パラメーターで呼び出すことができます。ただし、これは推奨されていません。
- Java ルーチンの DECFLOAT パラメーターは、DB2 バージョン 9.1 for z/OS 以降のデータベース・サーバーへの接続にのみ有効です。Java ルーチンの DECFLOAT パラメーターは、for Linux, UNIX, and Windows への接続にはサポートされていません。DECFLOAT を使用するには、SDK for Java バージョン 5 (1.5) 以降が必要です。
- このマッピングは、データベース・サーバーが列のデータ・タイプを判別できる場合にのみ有効です。
- ARRAY パラメーターはストアド・プロシージャにのみサポートされます。

## Java ストアド・プロシージャおよびユーザー定義関数でのデータ・タイプ

以下の表は、CREATE PROCEDURE または CREATE FUNCTION ステートメント内の SQL パラメーターのデータ・タイプを、対応する Java ストアド・プロシージャまたはユーザー定義関数メソッド内のデータ・タイプにマッピングすることについて要約したものです。

DB2 Database for Linux, UNIX, and Windows の場合、複数の Java データ・タイプが SQL データ・タイプについてリストされている場合は、最初の Java データ・タイプのみ有効です。

DB2 for z/OS の場合、複数の Java データ・タイプがリストされており、最初のデータ・タイプ以外のデータ・タイプをメソッド・パラメーターとして使用する場合は、そのメソッド・パラメーターの Java データ・タイプを指定する CREATE PROCEDURE または CREATE FUNCTION ステートメントの EXTERNAL 節にメソッド・シグニチャーを含める必要があります。

表 31. CREATE PROCEDURE または CREATE FUNCTION ステートメントの SQL データ・タイプを、対応する Java ストアド・プロシージャまたはユーザー定義関数プログラム内のデータ・タイプにマッピングする

CREATE PROCEDURE または CREATE FUNCTION での SQL データ・タイプ	Java ストアド・プロシージャまたはユーザー定義関数メソッドでのデータ・タイプ <sup>1</sup>
SMALLINT	short, java.lang.Integer
INTEGER	int, java.lang.Integer
BIGINT	long, java.lang.Long
REAL	float, java.lang.Float
DOUBLE	double, java.lang.Double
DECIMAL	java.math.BigDecimal
DECFLOAT <sup>2</sup>	java.math.BigDecimal
CHAR	java.lang.String

表 31. CREATE PROCEDURE または CREATE FUNCTION ステートメントの SQL データ・タイプを、対応する Java ストアド・プロシージャまたはユーザー定義関数プログラム内のデータ・タイプにマッピングする (続き)

CREATE PROCEDURE または CREATE FUNCTION での SQL データ・タイプ	Java ストアド・プロシージャまたはユーザー定義関数メソッドでのデータ・タイプ <sup>1</sup>
VARCHAR	java.lang.String
CHAR FOR BIT DATA	byte[]
VARCHAR FOR BIT DATA	byte[]
BINARY	byte[]
VARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
BLOB	java.sql.Blob
CLOB	java.sql.Clob
DBCLOB	java.sql.Clob
ROWID	java.sql.Types.ROWID
XML AS CLOB	java.sql.Types.SQLXML

注:

1. DB2 Database for Linux, UNIX, and Windows サーバー上のストアド・プロシージャまたはユーザー定義関数の場合、最初のデータ・タイプのみ有効です。
2. Java ルーチンの DECFLOAT パラメーターは、DB2 バージョン 9.1 for z/OS 以降のデータベース・サーバーへの接続にのみ有効です。Java ルーチンの DECFLOAT パラメーターは、for Linux, UNIX, and Windows への接続にはサポートされていません。DECFLOAT を使用するには、SDK for Java バージョン 5 (1.5) 以降が必要です。

## IBM Data Server Driver for JDBC and SQLJ のプロパティー

IBM Data Server Driver for JDBC and SQLJ のプロパティーは、特定のデータ・ソースへの接続の作成方法を定義します。ほとんどのプロパティーを、DataSource オブジェクトまたは Connection オブジェクトに関して設定できます。

### プロパティーを設定する方法

プロパティーは以下の方法のいずれかで設定できます。

- setXXX メソッドを使用する。ここで XXX は、先頭文字が大文字になっている非修飾プロパティー名です。

プロパティーは、com.ibm.db2.jcc.DB2BaseDataSource から継承される、以下の IBM Data Server Driver for JDBC and SQLJ 固有のインプリメンテーションに適用可能です。

- com.ibm.db2.jcc.DB2SimpleDataSource
- com.ibm.db2.jcc.DB2ConnectionPoolDataSource
- com.ibm.db2.jcc.DB2XADataSource
- DriverManager.getConnection 呼び出しの info パラメーターの java.util.Properties 値。

- DriverManager.getConnection 呼び出しの url パラメーターの java.lang.String 値。

int データ・タイプのプロパティには、事前定義定数フィールド値を持つものがあります。定数フィールド値は、url パラメーターで使用する前に、整数値に解決する必要があります。例えば、com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_ALL を url パラメーターに使用することはできません。しかし、com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_ALL を含む URL スtring を作成して、その URL スtring を String 変数に割り当てることができます。その後、以下のようにして、その String 変数を url パラメーターで使用できます。

```
String url =
    "jdbc:db2://sysmvs1.st1.ibm.com:5021" +
    "user=dbadm;password=dbadm;" +
    "traceLevel=" +
    (com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL) + ";";

Connection con =
    java.sql.DriverManager.getConnection(url);
```

## サポートされるすべてのデータベース製品に共通の IBM Data Server Driver for JDBC and SQLJ のプロパティ

大半の IBM Data Server Driver for JDBC and SQLJ のプロパティは、ドライバーがサポートするすべてのデータベース製品に適用されます。

特に注記されていない限り、すべてのプロパティは com.ibm.db2.jcc.DB2BaseDataSource にあります。

それらのプロパティは以下のとおりです。

### blockingReadConnectionTimeout

接続ソケットの読み取りがタイムアウトになるまでの秒単位の時間。このプロパティは IBM Data Server Driver for JDBC and SQLJ Type 4 接続にのみ適用され、接続が正常に確立した後でデータ・ソースに送信されるすべての要求に影響を与えます。デフォルトは 0 です。0 の値は、タイムアウトがないことを意味します。

### databaseName

データ・ソースの名前を指定します。この名前は、接続 URL のデータベース部分として使用されます。名前は、IBM Data Server Driver for JDBC and SQLJ Type 4 接続か IBM Data Server Driver for JDBC and SQLJ Type 2 接続のどちらが使用されるかによって異なります。

IBM Data Server Driver for JDBC and SQLJ Type 4 接続の場合:

- 接続先が DB2 for z/OS サーバーの場合、databaseName 値は、インストール中に定義される DB2 ロケーション名です。この値の中の文字はすべて大文字でなければなりません。サーバーで以下の SQL ステートメントを実行することにより、ロケーション名を判別できます。

```
SELECT CURRENT SERVER FROM SYSIBM.SYSDUMMY1;
```

- 接続先が DB2 Database for Linux, UNIX, and Windows サーバーの場合、`databaseName` 値は、インストール中に定義されたデータベース名になります。
- 接続を IDS サーバーに対して行う場合、`database` は、データベース名です。この名前は大小文字を区別しません。サーバーにより、名前は小文字に変換されます。
- 接続先が IBM Cloudscape サーバーの場合、`databaseName` 値は、データベースを含むファイルの完全修飾名です。この名前は、二重引用符 (") で囲まなければならない。以下に例を示します。

```
"c:/databases/testdb"
```

このプロパティが設定されていない場合、ローカル・サイトへの接続が行われます。

IBM Data Server Driver for JDBC and SQLJ Type 2 接続の場合:

- `serverName` 接続プロパティの値がヌルの場合、`databaseName` 値は、インストール中に定義されるデータベース名になります。`serverName` プロパティの値がヌルではない場合、`databaseName` 値はデータベース別名になります。

#### defaultIsolationLevel

新規接続用のデフォルトのトランザクション分離レベルを指定します。このプロパティのデータ・タイプは `int` です。`defaultIsolationLevel` が `DataSource` で設定されている場合、その `DataSource` から作成されるすべての接続には、`defaultIsolationLevel` によって指定されたデフォルトの分離レベルがあります。

DB2 データ・ソースの場合、デフォルトは `java.sql.Connection.TRANSACTION_READ_COMMITTED` です。

IBM Informix Dynamic Server (IDS) データベースの場合、デフォルトはデータ・ソースのタイプによって決まります。次の表はデフォルトを示しています。

表 32. IDS データベースのデフォルトの分離レベル

データ・ソースのタイプ	デフォルトの分離レベル
ロギングを行う ANSI 準拠データベース	<code>java.sql.Connection.TRANSACTION_SERIALIZABLE</code>
ロギングを行わないデータベース	<code>java.sql.Connection.TRANSACTION_READ_UNCOMMITTED</code>
ロギングを行う非 ANSI 準拠データベース	<code>java.sql.Connection.TRANSACTION_READ_COMMITTED</code>

#### deferPrepares

`Connection.prepareStatement` メソッドの呼び出しにより、データ・ソースに対して SQL ステートメントを直ちに準備するか、または `PreparedStatement.execute` メソッドが実行されるまでステートメントの準備を据え置かかを指定します。このプロパティのデータ・タイプは `boolean` です。

`deferPrepares` は、DB2 Database for Linux, UNIX, and Windows に対する IBM Data Server Driver for JDBC and SQLJ Type 2 接続、および IBM Data Server Driver for JDBC and SQLJ Type 4 接続についてサポートされます。

可能な値は以下のとおりです。

- true** データ・ソースに対するステートメントの準備は、`PreparedStatement.execute` メソッドが実行されるまで行われません。これはデフォルトです。
- false** データ・ソースに対するステートメントの準備は、`Connection.prepareStatement` メソッドが実行されると行われます。

準備操作を据え置くと、ネットワークの遅延を減らすことができます。ただし、準備操作を据え置く場合、入力データ・タイプが表列のタイプと一致していることを確認する必要があります。

#### 説明

データ・ソースの説明。このプロパティのデータ・タイプは `String` です。

#### **downgradeHoldCursorsUnderXa**

XA 接続のもとで保留カーソルをオープンできるかどうかを指定します。

`downgradeHoldCursorsUnderXa` は以下に適用されます。

- DB2 for z/OS サーバーに対する IBM Data Server Driver for JDBC and SQLJ Type 4 接続
- DB2 Database for Linux, UNIX, and Windowsサーバーに対する IBM Data Server Driver for JDBC and SQLJ Type 4 接続または IBM Data Server Driver for JDBC and SQLJ Type 2 接続

デフォルトは `false` です。つまり、XA 接続のもとで保留カーソルをオープンできません。

`downgradeHoldCursorsUnderXa` を `true` に設定した場合、XA 接続のもとで保留カーソルをオープンできます。ただし、XA 接続のカーソルには、次のような制限があります。

- XA 接続のもとでオープンされた保留カーソルは、`WITH HOLD` 動作を行いません。XA 接続のもとでオープンされた保留カーソルは、`XA End` でクローズされます。
- ローカル・トランザクションで `XA Start` よりも前にオープンしているカーソルは、`XA Start` でクローズされます。

#### **driverType**

`DataSource` インターフェースでは、接続に使用するドライバーを判別します。このプロパティのデータ・タイプは `int` です。有効な値は 2 または 4 です。2 がデフォルトです。

#### **fullyMaterializeLobData**

`FETCH` 操作のときにドライバーが `LOB` ロケーターを検索するかどうかを示します。このプロパティのデータ・タイプは `boolean` です。

`fullyMaterializeLobData` の効果は、次のように、データ・ソースが連続ストリーミングをサポートするかどうかによって異なります。

- データ・ソースが連続ストリーミングをサポートしない場合:

`fullyMaterializeLobData` の値が `true` の場合、行が取り出されるときに `LOB` データは `JDBC` ドライバー内で完全にマテリアライズされます。この値が `false` の場合、`LOB` データはストリームされます。ドライバーは内部でロケーターを使用し、必要に応じて `LOB` データをチャンクごとに検索します。

大量のデータを収容している LOB を検索する場合、この値を `false` に設定するように強くお勧めします。デフォルトは `true` です。

- データ・ソースが連続ストリーミングをサポートする場合:

`progressiveStreaming` プロパティが `DB2BaseDataSource.YES` または `DB2BaseDataSource.NOT_SET` に設定されていると、JDBC ドライバーは `fullyMaterializeLobData` の値を無視します。

このプロパティは、ストアード・プロシージャのパラメーターや、両方向スクロール・カーソルを使用して取り出される LOB には影響しません。LOB ストアード・プロシージャ・パラメーターは常に完全にマテリアライズされます。両方向スクロール・カーソルを使用してフェッチされる LOB は、連続ストリーミングが無効の場合は LOB ロケーターを使用します。

### **loginTimeout**

データ・ソースへの接続に対する秒単位の最大待ち時間。 `loginTimeout` で指定された秒数が経過すると、ドライバーはデータ・ソースへの接続をクローズします。このプロパティのデータ・タイプは `int` です。デフォルトは `0` です。値 `0` は、タイムアウト値がデフォルトのシステム・タイムアウト値であることを意味します。このプロパティは、DB2 for z/OS上の IBM Data Server Driver for JDBC and SQLJ Type 2 接続ではサポートされません。

### **logWriter**

`DataSource` オブジェクトのすべてのロギングおよびトレース・メッセージが出力される文字出力ストリーム。このプロパティのデータ・タイプは `java.io.PrintWriter` です。デフォルト値は `null` です。この場合、`DataSource` のロギングもトレースも出力されません。

### **password**

接続の確立に使用するパスワード。このプロパティのデータ・タイプは `String` です。 `DataSource` インターフェースを使用して接続を確立するときは、`DataSource.getConnection` メソッドを次の形式で呼び出すと、このプロパティ値をオーバーライドすることができます。

```
getConnection(user, password);
```

### **portNumber**

DRDA サーバーが要求を `listen` するポート番号。このプロパティのデータ・タイプは `int` です。

### **resultSetHoldability**

コミット操作後もカーソルをオープンしたままにするかどうかを指定します。このプロパティのデータ・タイプは `int` です。有効な値は以下のとおりです。

#### **DB2BaseDataSource.HOLD\_CURSORS\_OVER\_COMMIT (1)**

コミット操作後にカーソルをクローズします。

#### **DB2BaseDataSource.CLOSE\_CURSORS\_AT\_COMMIT (2)**

コミット操作後にカーソルをオープンしたままにします。

#### **DB2BaseDataSource.NOT\_SET (0)**

コミット操作後にカーソルをクローズします。これはデフォルト値です。



## **securityMechanism**

DRDA セキュリティー・メカニズムを指定します。このプロパティーのデータ・タイプは `int` です。可能な値は以下のとおりです。

### **CLEAR\_TEXT\_PASSWORD\_SECURITY (3)**

ユーザー ID およびパスワード

### **USER\_ONLY\_SECURITY (4)**

ユーザー ID のみ

### **ENCRYPTED\_PASSWORD\_SECURITY (7)**

ユーザー ID および暗号化されたパスワード

### **ENCRYPTED\_USER\_AND\_PASSWORD\_SECURITY (9)**

暗号化されたユーザー ID およびパスワード

### **KERBEROS\_SECURITY (11)**

Kerberos。この値は IDS への接続には適用されません。

### **ENCRYPTED\_USER\_AND\_DATA\_SECURITY (12)**

暗号化されたユーザー ID および暗号化されたセキュリティー・センシティブ・データ。この値は DB2 for z/OS への接続だけに適用されます。

### **ENCRYPTED\_USER\_PASSWORD\_AND\_DATA\_SECURITY (13)**

暗号化されたユーザー ID とパスワード、および暗号化されたセキュリティー・センシティブ・データ。この値は IDS への接続には適用されません。

### **PLUGIN\_SECURITY (15)**

プラグイン・セキュリティー。この値は DB2 Database for Linux, UNIX, and Windows への接続だけに適用されます。

### **ENCRYPTED\_USER\_ONLY\_SECURITY (16)**

暗号化されたユーザー ID。この値は IDS への接続には適用されません。

このプロパティーを指定すると、指定されたセキュリティー・メカニズムのみが使用されます。セキュリティー・メカニズムが接続でサポートされていない場合、例外が出されます。

`securityMechanism` のデフォルト値は、`CLEAR_TEXT_PASSWORD_SECURITY` です。サーバーが、`CLEAR_TEXT_PASSWORD_SECURITY` をサポートしない一方で、`ENCRYPTED_USER_AND_PASSWORD_SECURITY` をサポートするときは、IBM Data Server Driver for JDBC and SQLJ ドライバーはセキュリティー・メカニズムを `ENCRYPTED_USER_AND_PASSWORD_SECURITY` に更新し、サーバーへの接続を試みます。リクエスターとサーバー間でこれ以外のセキュリティー・メカニズム・サポートの不一致がある場合には、エラーになります。

## **sendDataAsIs**

IBM Data Server Driver for JDBC and SQLJ が入力パラメーター値をターゲット列のデータ・タイプに変換しないことを指定します。このプロパティーのデータ・タイプは `boolean` です。デフォルトは `false` です。



このプロパティを使用する必要があるのは、アプリケーション内のデータ・タイプが、対応するデータベース表内のデータ・タイプに常に一致するようにしているアプリケーションの場合のみです。

#### **serverName**

データ・ソースのホスト名または TCP/IP アドレス。このプロパティのデータ・タイプは String です。

#### **traceDirectory**

トレース情報の書き込み先のディレクトリーを指定します。このプロパティのデータ・タイプは String です。 traceDirectory を指定すると、同一の DataSource 上にある複数の接続のトレース情報が、複数のファイルに書き込まれます。

traceDirectory を指定すると、接続のトレースは、traceFile\_origin\_n という名前のファイルへ書き込まれます。

n は、DataSource の n 番目の接続です。

origin は、使用中のログ書き込みプログラムの起点を示します。 origin に指定できる値は次のとおりです。

**cpds** DB2ConnectionPoolDataSource オブジェクトのログ書き込みプログラム。

**driver** DB2Driver オブジェクトのログ書き込みプログラム。

**global** DB2TraceManager オブジェクトのログ書き込みプログラム。

**sds** DB2SimpleDataSource オブジェクトのログ書き込みプログラム。

**xads** DB2XADataSource オブジェクトのログ書き込みプログラム。

traceFile プロパティも指定した場合、traceDirectory 値は使用されません。

#### **traceFile**

IBM Data Server Driver for JDBC and SQLJ がトレース情報を書き込むファイルの名前を指定します。このプロパティのデータ・タイプは String です。 traceFile プロパティは、ファイルに出力トレース・ストリームを送信する logWriter プロパティの代わりに使用できます。

#### **traceFileAppend**

traceFile プロパティによって指定されるファイルへ付加するか、あるいは上書きするかを指定します。このプロパティのデータ・タイプは boolean です。デフォルトは false です。これは、traceFile プロパティによって指定されるファイルが上書きされることを意味します。

#### **traceLevel**

何をトレースするかを指定します。このプロパティのデータ・タイプは int です。

traceLevel プロパティで、以下のトレースを 1 つ以上指定することができます。

- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_NONE (X'00')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_CONNECTION\_CALLS (X'01')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_STATEMENT\_CALLS (X'02')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_RESULT\_SET\_CALLS (X'04')

- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_DRIVER\_CONFIGURATION (X'10')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_CONNECTS (X'20')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_DRDA\_FLOWS (X'40')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_RESULT\_SET\_META\_DATA (X'80')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_PARAMETER\_META\_DATA (X'100')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_DIAGNOSTICS (X'200')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_SQLJ (X'400')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_XA\_CALLS (DB2 Database for Linux, UNIX, and Windows 用の IBM Data Server Driver for JDBC and SQLJ Type 2 接続のみ) (X'800')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_META\_CALLS (X'2000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_DATASOURCE\_CALLS (X'4000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_LARGE\_OBJECT\_CALLS (X'8000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_SYSTEM\_MONITOR (X'20000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_TRACEPOINTS (X'40000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_ALL (X'FFFFFFFF')

複数のトレースを指定するには、以下の技法のいずれかを使用します。

- 2 つ以上のトレース値にビット単位 OR (|) 演算子を使用する。たとえば、DRDA フローおよび接続呼び出しをトレースするには、traceLevel に以下の値を指定します。

```
TRACE_DRDA_FLOWS|TRACE_CONNECTION_CALLS
```

- ビット単位の補数 (~) 演算子とトレース値を使用して、特定のトレース以外のすべてを指定する。たとえば、DRDA フロー以外のすべてをトレースする場合、traceLevel に以下の値を指定します。

```
~TRACE_DRDA_FLOWS
```

#### user

接続の確立に使用するユーザー ID。このプロパティのデータ・タイプは String です。 DataSource インターフェースを使用して接続を確立するときは、DataSource.getConnection メソッドを次の形式で呼び出すと、このプロパティ値をオーバーライドすることができます。

```
getConnection(user, password);
```

#### xaNetworkOptimization

IBM Data Server Driver for JDBC and SQLJ Type 4 接続に関して XA ネットワーク最適化を使用可能にするかどうかを指定します。1 つの Java プロセスから XA Start と XA End が発行され、別の Java プロセスから XA Prepare と XA Commit が発行されるような環境では、XA ネットワーク最適化を使用不可にする必要があるかもしれません。XA ネットワーク最適化では、XA Prepare が XA End より先にデータ・ソースに到達する可能性があります。この場合、XAER\_PROTO エラーが発生します。XAER\_PROTO エラーを防ぐには、XA ネットワーク最適化を使用不可にしてください。

デフォルトは true で、XA ネットワーク最適化が使用可能になります。

xaNetworkOptimization が false の場合 (つまり XA ネットワーク最適化が使用不可の場合)、ドライバは XA End 時にすべてのオープン・カーソルをクローズします。

xaNetworkOptimization は、DataSource オブジェクトで、あるいは getConnection 呼び出しの url パラメーターで設定することができます。接続の取得後に xaNetworkOptimization の値を変更することはできません。

#### **com.ibm.db2.jcc.DB2ConnectionPoolDataSource.maxStatements**

PooledConnection と関連付けられた内部ステートメント・キャッシュを制御します。このプロパティーのデータ・タイプは int です。可能な値は以下のとおりです。

##### **正の整数**

PooledConnection について内部ステートメント・キャッシュを使用可能にし、IBM Data Server Driver for JDBC and SQLJ がキャッシュ内でオープンしたままにするステートメントの数を指定します。

##### **0 または負の整数**

PooledConnection について内部ステートメント・キャッシングを使用不可にします。0 がデフォルトです。

maxStatements は、PooledConnection オブジェクトが作成されるときにのみ、PooledConnection と関連付けられた内部ステートメント・キャッシュを制御します。maxStatements は、既に存在している PooledConnection オブジェクトでのキャッシングには無効です。

maxStatements が適用されるのは、IBM Data Server Driver for JDBC and SQLJ Type 4 接続だけです。

## **DB2 for z/OS および DB2 Database for Linux, UNIX, and Windows に共通の IBM Data Server Driver for JDBC and SQLJ のプロパティー**

IBM Data Server Driver for JDBC and SQLJ の一部のプロパティーは、DB2 for z/OS および DB2 Database for Linux, UNIX, and Windows だけに適用されます。

特に注記されていない限り、すべてのプロパティーは com.ibm.db2.jcc.DB2BaseDataSource にあります。

それらのプロパティーは以下のとおりです。

#### **clientRerouteAlternateServerName**

クライアント・リルート用の 1 つ以上の代替サーバー名を指定します。

clientRerouteAlternateServerName は、クライアント・リルートが JNDI ストアなしでセットアップされるときに使用されます。このプロパティーのデータ・タイプは String です。

複数のサーバー名を指定する場合は、サーバー名をコンマ (,) で区切ります。

clientRerouteAlternateServerName に指定される値の数は、

clientRerouteAlternatePortNumber に指定される値の数と一致していなければなりません。

clientRerouteAlternateServerName は、DB2 Database for Linux, UNIX, and Windows に対する IBM Data Server Driver for JDBC and SQLJ Type 2 接続、および IBM Data Server Driver for JDBC and SQLJ Type 4 接続に適用されます。

### **clientRerouteAlternatePortNumber**

クライアント・リルート用の 1 つ以上の代替ポート番号を指定します。  
`clientRerouteAlternatePortNumber` は、クライアント・リルートが JNDI ストアなしでセットアップされるときに使用されます。このプロパティのデータ・タイプは `String` です。

複数のポート番号を指定する場合は、ポート番号をコンマ (,) で区切ります。  
`clientRerouteAlternatePortNumber` に指定される値の数は、  
`clientRerouteAlternateServerName` に指定される値の数と一致していなければなりません。

`clientRerouteAlternatePortNumber` は、DB2 Database for Linux, UNIX, and Windows に対する IBM Data Server Driver for JDBC and SQLJ Type 2 接続、および IBM Data Server Driver for JDBC and SQLJ Type 4 接続に適用されます。

### **clientRerouteServerListJNDIName**

転送サーバー情報の JNDI リポジトリ内の `DB2ClientRerouteServerList` インスタンスへの JNDI 参照を識別します。`clientRerouteServerListJNDIName` は、IBM Data Server Driver for JDBC and SQLJ Type 4 接続と、`DataSource` インターフェースにより設定された接続にのみ適用されます。

`clientRerouteServerListJNDIName` の値がヌルでない場合、  
`clientRerouteServerListJNDIName` は以下の機能を提供します。

- 転送サーバーに関する情報を、複数の JVM にわたって保持できます。
- データ・ソースへの最初の接続が失敗した場合に代替サーバー・ロケーションを提供します。

### **clientRerouteServerListJNDIContext**

`DB2ClientRerouteServerList` インスタンスのバインディングおよび参照に使用される JNDI コンテキストを指定します。`clientRerouteServerListJNDIContext` は、IBM Data Server Driver for JDBC and SQLJ Type 4 接続と、`DataSource` インターフェースにより設定された接続にのみ適用されます。

`clientRerouteServerListJNDIContext` を設定しない場合、IBM Data Server Driver for JDBC and SQLJ はシステム・プロパティまたは `jndi.properties` ファイルを使って初期コンテキストを作成します。

`clientRerouteServerListJNDIContext` は、以下のメソッドを使用してのみ設定できます。

```
public void setClientRerouteServerListJNDIContext(javax.naming.Context registry)
```

### **currentDegree**

動的に準備される照会の実行の並列処理の度合いを指定します。このプロパティのタイプは `String` です。`currentDegree` 値は、データ・ソース上に `CURRENT DEGREE` 特殊レジスターを設定するために使用されます。`currentDegree` が設定されていない場合、データ・ソースに渡される値はありません。

### **currentFunctionPath**

JDBC プログラム内にある SQL ステートメント内の非修飾の表データ・タイプ名および関数名を解決するために使用される、SQL パスを指定します。このプロパティのデータ・タイプは `String` です。DB2 Database for Linux, UNIX, and Windows サーバーの場合の最大長は 254 バイトです。DB2 for z/OS サー

パーの場合の最大長は 2048 バイトです。値は、スキーマ名のコンマで区切られたリストになります。それらの名前は、通常 ID または区切り ID にできません。

#### **currentMaintainedTableTypesForOptimization**

動的 SQL 照会の処理をデータ・ソースが最適化するときに検討の対象にできるオブジェクトのタイプを特定する値を指定します。このレジスターには、表タイプを表すキーワードが含まれます。このプロパティのデータ・タイプは String です。

currentMaintainedTableTypesForOptimization に指定できる値は次のとおりです。

##### **ALL**

すべてのマテリアライズ照会表が検討の対象になることを示します。

##### **NONE**

どのマテリアライズ照会表も検討の対象にならないことを示します。

##### **SYSTEM**

REFRESH DEFERRED されるシステム保守のマテリアライズ照会表のみが検討の対象となることを示します。

##### **USER**

REFRESH DEFERRED されるユーザー保守のマテリアライズ照会表のみが検討の対象となることを示します。

#### **currentPackagePath**

サーバー上のコレクションのコンマで区切られたリストを指定します。データベース・サーバーはこれらのコレクションから JDBC および SQLJ パッケージを検索します。

currentPackagePath および currentPackageSet プロパティの優先順位規則は、CURRENT PACKAGESET および CURRENT PACKAGE PATH 特殊レジスターの優先順位規則に従います。

#### **currentPackageSet**

JDBC および SQLJ パッケージの検索に使用するコレクション ID を指定します。このプロパティのデータ・タイプは String です。デフォルトは NULLID です。currentPackageSet が設定されている場合、その値は jdbcCollection の値をオーバーライドします。

DB2Binder ユーティリティを複数回実行することにより、データベース・サーバーで IBM Data Server Driver for JDBC and SQLJ の複数のインスタンスをインストールできます。DB2binder ユーティリティに含まれる -collection オプションを使用すると、各 IBM Data Server Driver for JDBC and SQLJ インスタンスのコレクション ID をインストーラーで指定できます。接続に対する IBM Data Server Driver for JDBC and SQLJ のインスタンスを選択するには、いずれかの IBM Data Server Driver for JDBC and SQLJ インスタンスのコレクション ID に一致する currentPackageSet 値を指定します。

currentPackagePath および currentPackageSet プロパティの優先順位規則は、CURRENT PACKAGESET および CURRENT PACKAGE PATH 特殊レジスターの優先順位規則に従います。

#### **currentRefreshAge**

タイム・スタンプ期間の値を指定します。これは、マテリアライズ照会表を使用

して照会の処理を最適化できるようにするために、REFRESH TABLE ステートメントをシステム保守の REFRESH DEFERRED マテリアライズ照会表で処理してからの最大期間です。このプロパティは、動的ステートメントのキャッシュの突き合わせに影響を与えます。このプロパティのデータ・タイプは LONG です。

### currentSchema

動的に準備される SQL ステートメントで非修飾データベース・オブジェクトを修飾するために使用されるデフォルトのスキーマ名を指定します。このプロパティの値は、データベース・サーバーでの CURRENT SCHEMA 特殊レジスタ一内の値を設定します。

### cursorSensitivity

JDBC ResultSet の java.sql.ResultSet.TYPE\_SCROLL\_SENSITIVE 値を、基礎となるデータベース・カーソルの SENSITIVE DYNAMIC 属性、SENSITIVE STATIC 属性、または ASENSITIVE 属性にマップするかどうかを指定します。このプロパティのデータ・タイプは int です。指定できる値は TYPE\_SCROLL\_SENSITIVE\_STATIC (0)、TYPE\_SCROLL\_SENSITIVE\_DYNAMIC (1)、または TYPE\_SCROLL\_ASENSITIVE (2) です。デフォルトは TYPE\_SCROLL\_SENSITIVE\_STATIC です。

データ・ソースがセンシティブ動的両方向スクロール・カーソルをサポートしていない場合に、TYPE\_SCROLL\_SENSITIVE\_DYNAMIC が要求されると、JDBC ドライバーは警告を累積し、感度を SENSITIVE STATIC にマップします。センシティブ静的カーソルをサポートしない DB2 for i5/OS データベース・サーバーの場合、java.sql.ResultSet.TYPE\_SCROLL\_SENSITIVE は常に SENSITIVE DYNAMIC にマップされます。

### dateFormat

下記のことを指定します。

- DATE 列に対する PreparedStatement.setString メソッドの String 引数を指定する際の形式。
- DATE 列に対する ResultSet.getString または CallableStatement.getString メソッドの結果が返される形式。

dateFormat のデータ・タイプは int です。

dateFormat に指定できる値は次のとおりです。

定数	整数値	形式
com.ibm.db2.jcc.DB2BaseDataSource.ISO	1	yyyy-mm-dd
com.ibm.db2.jcc.DB2BaseDataSource.USA	2	mm/dd/yyyy
com.ibm.db2.jcc.DB2BaseDataSource.EUR	3	dd.mm.yyyy
com.ibm.db2.jcc.DB2BaseDataSource.JIS	4	yyyy-mm-dd

デフォルトは com.ibm.db2.jcc.DB2BaseDataSource.ISO です。

### decimalRoundingMode

DB2 for z/OS または DB2 Database for Linux, UNIX, and Windows データベース・サーバー上での、10 進浮動小数点値の丸めモードを指定します。

可能な値は以下のとおりです。



#### **com.ibm.db2.jcc.DB2BaseDataSource.ROUND\_DOWN (1)**

0 の方向に丸めます (切り捨て)。破棄された数字は無視されます。

#### **com.ibm.db2.jcc.DB2BaseDataSource.ROUND\_CEILING (2)**

正の無限大方向に丸めます。破棄される数字がすべてゼロであるか、または符号が負の場合、破棄される数字の除去を除いて結果は未変更のままになります。それ以外の場合、結果の係数は 1 だけ増やされます。

#### **com.ibm.db2.jcc.DB2BaseDataSource.ROUND\_HALF\_EVEN (3)**

値を最も近い値に丸めます。複数の値が等距離にある場合は、最後の数字が偶数になるように値を丸めます。破棄される数字が、左側の次の桁内の 1 の半分 (0.5) より大きい場合、結果の係数は 1 だけ増やされます。その数字が半分より小さい場合、結果の係数は調整されません (つまり、破棄された数字は無視されます)。それ以外の場合、結果の係数の右端の数字が偶数であれば、その係数は変更されませんが、右端の数字が奇数であれば、その桁が偶数になるように 1 だけ増大 (切り上げ) します。

#### **com.ibm.db2.jcc.DB2BaseDataSource.ROUND\_HALF\_UP (4)**

値を最も近い値に丸めます。複数の値が等距離にある場合は、0 の反対方向に値を丸めます。破棄される数字が、左側の次の桁内の 1 の半分 (0.5) 以上の場合、結果の係数は 1 だけ増やされます。そうでない場合、破棄された数字は無視されます。

#### **com.ibm.db2.jcc.DB2BaseDataSource.ROUND\_FLOOR (6)**

負の無限大方向に丸めます。破棄される数字がすべてゼロであるか、または符号が正の場合、破棄される数字の除去を除いて、結果は未変更のままになります。それ以外の場合、符号は負で、結果の係数は 1 だけ増やされます。

#### **com.ibm.db2.jcc.DB2BaseDataSource.ROUND\_UNSET (-2147483647)**

丸めモードは明示的に設定されませんでした。IBM Data Server Driver for JDBC and SQLJ は、decimalRoundingMode を使用してデータ・ソース上で丸めモードを設定することはしません。

IBM Data Server Driver for JDBC and SQLJ はその丸めモードに以下の値を使用します。

- DB2 for z/OS または DB2 Database for Linux, UNIX, and Windows データベース・サーバーの場合、10 進浮動小数点値の丸めモードは ROUND\_HALF\_EVEN です。

decimalRoundingMode が設定されている場合、decimalRoundingMode 値は、DB2 for z/OS データベース・サーバー上で CURRENT DECFLOAT ROUNDING MODE 特殊レジスターを設定するために使用されます。

#### **fullyMaterializeInputStreams**

ストリームが、クライアントからデータ・ソースに送信される前に、完全にマテリアライズ化されるかどうかを示します。このプロパティのデータ・タイプは boolean です。デフォルトは false です。

fullyMaterializeInputStreams の値が true の場合、JDBC ドライバーはストリームをサーバーに送信する前に完全にマテリアライズ化したことになります。



### **gssCredential**

Kerberos セキュリティーを使用するデータ・ソースの場合に、他のプリンシパルから渡される委任証明書を指定します。このプロパティーのデータ・タイプは `org.ietf.jgss.GSSCredential` です。委任証明書は、例えばクライアントが WebSphere Application Server に接続し、次いでそれがデータ・ソースに接続するような、複数層の環境において使用されます。 `GSSContext.getDelegCred` メソッドを呼び出すことにより、このプロパティーの値をクライアントから取得します。 `GSSContext` は IBM Java Generic Security Service (GSS) API に含まれています。このプロパティーを設定する場合、 `Mechanism` および `KerberosServerPrincipal` プロパティーも設定しなければなりません。

このプロパティーを適用できるのは IBM Data Server Driver for JDBC and SQLJ Type 4 接続だけです。

IBM Data Server Driver for JDBC and SQLJ での Kerberos セキュリティーの使用については、『IBM Data Server Driver for JDBC and SQLJ 使用時の Kerberos セキュリティーの使用』を参照してください。

### **kerberosServerPrincipal**

Kerberos セキュリティーを使用するデータ・ソースの場合、 Kerberos 鍵配布センター (KDC) への登録時にデータ・ソースに使用される名前を指定します。このプロパティーのデータ・タイプは `String` です。

このプロパティーを適用できるのは IBM Data Server Driver for JDBC and SQLJ Type 4 接続だけです。

### **maxRetriesForClientReroute**

自動クライアント・リルート中に、データ・ソースへの 1 次接続に失敗した場合の再試行回数を制限します。

このプロパティーのデータ・タイプは `int` です。

IBM Data Server Driver for JDBC and SQLJ が `maxRetriesForClientReroute` プロパティーを使用するのは、 `retryIntervalForClientReroute` プロパティーも設定されている場合のみです。 `maxRetriesForClientReroute` のデフォルト値は、 `MAX_RETRIES_NOT_SET (-1)` です。

`maxRetriesForClientReroute` または `retryIntervalForClientReroute` が設定されていない場合、IBM Data Server Driver for JDBC and SQLJ は再試行を 10 分間実行します。

### **progressiveStreaming**

データ・ソース上で連続ストリーミングがサポートされている場合に、JDBC ドライバーが連続ストリーミングを使用するかどうかを指定します。連続ストリーミングを使用すると、LOB または XML オブジェクトのサイズに基づいて、LOB または XML のデータを戻すのに最も効率のよいモードがデータ・ソースで動的に判別されます。 `streamBufferSize` パラメーターの値は、データが戻されるときにマテリアライズ化されるかどうかを決定します。

`progressiveStreaming` のデータ・タイプは `int` です。有効な値は `DB2BaseDataSource.YES (1)` および `DB2BaseDataSource.NO (2)` です。

`progressiveStreaming` プロパティーが指定されない場合、 `progressiveStreaming` 値は `DB2BaseDataSource.NOT_SET (0)` になります。

連続ストリーミングをサポートするデータ・ソースへの接続の場合に、`progressiveStreaming` の値が `DB2BaseDataSource.YES` または `DB2BaseDataSource.NOT_SET` であると、JDBC ドライバーは連続ストリーミングを使用して LOB および XML データを戻します。

`progressiveStreaming` の値が `DB2BaseDataSource.NO` であるか、またはデータ・ソースが連続ストリーミングをサポートしない場合、JDBC ドライバーが LOB または XML データを戻す方法は、`fullyMaterializeLobData` プロパティの値によって異なります。

#### **readOnly**

接続を読み取り専用にするかどうかを指定します。このプロパティのデータ・タイプは `boolean` です。デフォルトは `false` です。

#### **resultSetHoldabilityForCatalogQueries**

`DatabaseMetaData` メソッドの代わりに実行される照会用のカーソルをコミット操作後もオープンしたままにするかどうかを指定します。このプロパティのデータ・タイプは `int` です。

アプリケーションが `DatabaseMetaData` メソッドを実行すると、IBM Data Server Driver for JDBC and SQLJ はターゲット・データ・ソースのカタログに対して照会を実行します。デフォルトでは、そのようなカーソルの保持可能性はアプリケーション・カーソルの保持可能性と同じです。カタログ照会に別の保持可能性を使用するには、`resultSetHoldabilityForCatalogQueries` プロパティを使用します。可能な値は以下のとおりです。

##### **DB2BaseDataSource.HOLD\_CURSORS\_OVER\_COMMIT (1)**

`resultSetHoldability` の設定とは無関係に、コミット操作後にカタログ照会用のカーソルをクローズします。

##### **DB2BaseDataSource.CLOSE\_CURSORS\_AT\_COMMIT (2)**

`resultSetHoldability` の設定とは無関係に、コミット操作後にカタログ照会用のカーソルをオープンしたままにします。

##### **DB2BaseDataSource.NOT\_SET (0)**

カタログ照会に `resultSetHoldability` 設定を使用します。これはデフォルト値です。

有効な値は `HOLD_CURSORS_OVER_COMMIT (1)` または `CLOSE_CURSORS_AT_COMMIT (2)` です。これらの値は、JDBC 3.0 で定義される `ResultSet.HOLD_CURSORS_OVER_COMMIT` および `ResultSet.CLOSE_CURSORS_AT_COMMIT` 定数と同じです。

#### **retryIntervalForClientReroute**

自動クライアント・リルートの場合、接続の再試行を行う間隔を秒単位の時間で指定します。

このプロパティのデータ・タイプは `int` です。

IBM Data Server Driver for JDBC and SQLJ が `retryIntervalForClientReroute` プロパティを使用するのは、`maxRetriesForClientReroute` プロパティも設定されている場合のみです。 `retryIntervalForClientReroute` のデフォルト値は、`RETRY_INTERVAL_NOT_SET (-1)` です。

maxRetriesForClientReroute または retryIntervalForClientReroute が設定されていない場合、IBM Data Server Driver for JDBC and SQLJ は再試行を 10 分間実行します。

#### **returnAlias**

getTables などの表情報を戻す DatabaseMetaData メソッドについて、JDBC ドライバーが表の別名およびシノニムの行を戻すかどうかを指定します。returnAlias のデータ・タイプは int です。可能な値は以下のとおりです。

- 0 表情報を戻す DatabaseMetaData メソッドからの出力内に、表の別名またはシノニムの行を戻しません。
- 1 別名またはシノニムを持つ表の場合、表情報を戻す DatabaseMetaData メソッドからの出力内に、表の行とともに、それらの表の別名およびシノニムの行を戻します。これはデフォルトです。

#### **sslConnection**

IBM Data Server Driver for JDBC and SQLJ が SSL ソケットを使用してデータ・ソースに接続するかどうかを指定します。sslConnection を true に設定すると、SSL ソケットが接続で使用されます。sslConnection を false に設定すると、プレーン・ソケットが接続で使用されます。

#### **streamBufferSize**

LOB または XML データをチャンク化するための、JDBC ドライバー・バッファのサイズをバイト数で指定します。JDBC ドライバーは、連続ストリーミングを使用するかどうかに関係なく、streamBufferSize 値を使用します。streamBufferSize のデータ・タイプは int です。デフォルトは 1048576 です。

JDBC ドライバーが連続ストリーミングを使用する場合、LOB または XML データがマテリアライズ化されるのは、そのデータがバッファ内に収まり、このドライバーが fullyMaterializeLobData プロパティを使用しない場合です。

#### **supportsAsynchronousXARollback**

IBM Data Server Driver for JDBC and SQLJ が非同期の XA ロールバック操作をサポートするかどうかを指定します。このプロパティのデータ・タイプは int です。デフォルトは DB2BaseDataSource.NO (2) です。BEA WebLogic Server アプリケーション・サーバーに対してアプリケーションを実行する場合、supportsAsynchronousXARollback を DB2BaseDataSource.YES (1) に設定します。

#### **sysSchema**

アプリケーションが DatabaseMetaData メソッドを呼び出す際に検索される、シャドウ・カタログ表またはビューのスキーマを指定します。sysSchema プロパティは、以前は cliSchema という名称でした。

#### **timeFormat**

下記のことを指定します。

- TIME 列に対する PreparedStatement.setString メソッドの String 引数を指定する際の形式。
- TIME 列に対する ResultSet.getString または CallableStatement.getString メソッドの結果が戻される形式。

timeFormat のデータ・タイプは int です。

timeFormat に指定できる値は次のとおりです。

定数	整数値	形式
com.ibm.db2.jcc.DB2BaseDataSource.ISO	1	hh:mm:ss
com.ibm.db2.jcc.DB2BaseDataSource.USA	2	h:mm am または h:mm pm
com.ibm.db2.jcc.DB2BaseDataSource.EUR	3	hh.mm.ss
com.ibm.db2.jcc.DB2BaseDataSource.JIS	4	hh:mm:ss

デフォルトは com.ibm.db2.jcc.DB2BaseDataSource.ISO です。

#### useCachedCursor

PreparedStatement オブジェクトの基礎カーソルをキャッシュに入れて、PreparedStatement の後続の実行で再利用するかどうかを指定します。useCachedCursor のデータ・タイプは boolean です。

useCachedCursor が true に設定されている場合、PreparedStatement オブジェクトのカーソルはキャッシュに入れられ、これによりパフォーマンスは向上します。true がデフォルトです。

PreparedStatement オブジェクトが、それらの PreparedStatement オブジェクトの実行間に列のタイプまたは長さが変わる表にアクセスする場合は、useCachedCursor を false に設定します。

#### useJDBC4ColumnNameAndLabelSemantics

IBM Data Server Driver for JDBC and SQLJ が、ResultSetMetaData.getColumnNames、ResultSetMetaData.getColumnLabels、および ResultSet.findColumn メソッド呼び出しで列ラベルを処理する方法を指定します。可能な値は以下のとおりです。

##### com.ibm.db2.jcc.DB2BaseDataSource.YES (1)

IBM Data Server Driver for JDBC and SQLJ は、JDBC 4.0 仕様に準拠する以下の規則を使用して、ResultSetMetaData.getColumnNames、ResultSetMetaData.getColumnLabels、および ResultSet.findColumn が戻す値を判別します。

- ResultSetMetaData.getColumnNames によって戻される列名はデータベースからの名前です。
- ResultSetMetaData.getColumnLabels によって戻される列ラベルは SQL AS 節で指定されたラベルです。SQL AS 節が指定されていない場合、ラベルは列の名前です。
- ResultSet.findColumn は、SQL AS 節で指定された列のラベルを入力としてとります。SQL AS 節が指定されていない場合、ラベルは列名です。
- IBM Data Server Driver for JDBC and SQLJ は、SQL LABEL ON ステートメントによって割り当てられた列ラベルを使用しません。

これらの規則は、次のデータベース・システムへの接続の場合、IBM Data Server Driver for JDBC and SQLJ バージョン 4.0 以降に適用されます。

- DB2 for z/OS バージョン 8 以降
- DB2 Database for Linux, UNIX, and Windows バージョン 8.1 以降

- DB2 for i5/OS V5R3 以降

これらのデータベース・システムより前のバージョンの場合、`com.ibm.db2.jcc.DB2BaseDataSource.NO` の `useJDBC4ColumnNameAndLabelSemantics` 値の規則が適用されます。

#### **com.ibm.db2.jcc.DB2BaseDataSource.NO (2)**

IBM Data Server Driver for JDBC and SQLJ は以下の規則を使用して、`ResultSetMetaData.getColumnName`、`ResultSetMetaData.getColumnLabel`、および `ResultSet.findColumn` が戻す値を判別します。

データ・ソースが LABEL ON ステートメントをサポートしないか、またはソース列が LABEL ON ステートメントで定義されていない場合、以下の規則が適用されます。

- `ResultSetMetaData.getColumnName` によって戻される値は、SQL AS 節が指定されていない場合はデータベースからの名前です。SQL AS 節が指定されている場合、戻される値は列ラベルです。
- `ResultSetMetaData.getColumnLabel` によって戻される値は SQL AS 節で指定されたラベルです。SQL AS 節が指定されていない場合、戻される値は列の名前です。
- `ResultSet.findColumn` は列名を入力としてとります。

ソース列が LABEL ON ステートメントで定義されている場合、以下の規則が適用されます。

- `ResultSetMetaData.getColumnName` によって戻される値は、SQL AS 節が指定されていない場合はデータベースからの列名です。SQL AS 節が指定されている場合、戻される値は AS 節で指定された列ラベルです。
- `ResultSetMetaData.getColumnLabel` によって戻される値は LABEL ON ステートメントで指定されたラベルです。
- `ResultSet.findColumn` は列名を入力としてとります。

これらの規則は、IBM Data Server Driver for JDBC and SQLJ のバージョン 3.50 より前の動作に準拠します。

#### **com.ibm.db2.jcc.DB2BaseDataSource.NOT\_SET (0)**

これがデフォルトの動作です。

IBM Data Server Driver for JDBC and SQLJ バージョン 3.50 以前の場合、`useJDBC4ColumnNameAndLabelSemantics` のデフォルト動作は `com.ibm.db2.jcc.DB2BaseDataSource.NO` の動作と同じです。

IBM Data Server Driver for JDBC and SQLJ バージョン 4.0 以降の場合:

- `useJDBC4ColumnNameAndLabelSemantics` のデフォルト動作は、次のデータベース・システムへの接続の場合、`com.ibm.db2.jcc.DB2BaseDataSource.YES` の動作と同じです。
  - DB2 for z/OS バージョン 8 以降
  - DB2 Database for Linux, UNIX, and Windows バージョン 8.1 以降
  - DB2 for i5/OS V5R3 以降

- これらのデータベース・システムより前のバージョンへの接続の場合、`useJDBC4ColumnNameAndLabelSemantics` のデフォルト動作は `com.ibm.db2.jcc.DB2BaseDataSource.NO` です。

#### **com.ibm.db2.jcc.DB2ConnectionPoolDataSource.maxStatements**

`PooledConnection` と関連付けられた内部ステートメント・キャッシュを制御します。このプロパティのデータ・タイプは `int` です。可能な値は以下のとおりです。

##### **正の整数**

`PooledConnection` について内部ステートメント・キャッシュを使用可能にし、IBM Data Server Driver for JDBC and SQLJ がキャッシュ内でオープンしたままにするステートメントの数を指定します。

##### **0 または負の整数**

`PooledConnection` について内部ステートメント・キャッシングを使用不可にします。0 がデフォルトです。

`maxStatements` は、`PooledConnection` オブジェクトが作成されるときにのみ、`PooledConnection` と関連付けられた内部ステートメント・キャッシュを制御します。`maxStatements` は、既に存在している `PooledConnection` オブジェクトでのキャッシングには無効です。

`maxStatements` が適用されるのは、IBM Data Server Driver for JDBC and SQLJ Type 4 接続だけです。

## **DB2 for z/OS および IBM Informix Dynamic Server に共通の IBM Data Server Driver for JDBC and SQLJ のプロパティ**

IBM Data Server Driver for JDBC and SQLJ の一部のプロパティは、IDS および DB2 for z/OS データベース・サーバーに適用されます。

それらのプロパティは以下のとおりです。

### **keepDynamic**

既に準備済みの動的 SQL ステートメントを再利用できるように、コミット・ポイント後もその SQL ステートメントをデータ・ソースが動的ステートメント・キャッシュ内に維持し続けるかどうかを指定します。このプロパティのデータ・タイプは `int` です。有効な値は `DB2BaseDataSource.YES` (1) および `DB2BaseDataSource.NO` (2) です。

`keepDynamic` プロパティが指定されない場合、`keepDynamic` 値は `DB2BaseDataSource.NOT_SET` (0) になります。接続先が DB2 for z/OS サーバーの場合、プロパティが設定されていなければ接続に対する動的ステートメントのキャッシングは行われません。接続先が IDS データ・ソースの場合、プロパティが設定されていなければ接続に対する動的ステートメントのキャッシングが行われます。

`keepDynamic` は、`DB2Binder -keepdynamic` オプションと一緒に使用します。指定された `keepDynamic` プロパティの値は、`DB2Binder` の実行時に指定された `-keepdynamic` 値と一致していなければなりません。

DB2 for z/OS データベース・サーバーの場合、動的ステートメントのキャッシングを行えるのは、データ・ソース上で EDM 動的ステートメント・キャッシ



ユが使用可能になっている場合のみです。動的ステートメント・キャッシュを使用可能にするには、CACHEDYN サブシステム・パラメーターを DB2BaseDataSource.YES に設定する必要があります。

#### **retrieveMessagesFromServerOnGetMessage**

JDBC SQLException.getMessage または SQLWarning.getMessage 呼び出しで、エラー・メッセージ・テキストを取得する DB2 for z/OS ストアード・プロシージャを IBM Data Server Driver for JDBC and SQLJ に呼び出させるかどうかを指定します。このプロパティのデータ・タイプは boolean です。デフォルトは false です。これは、メッセージ・テキスト全体がクライアントに戻されるわけではないことを意味します。

例えば、retrieveMessagesFromServerOnGetMessage を true に設定すると、存在しない表 ADMF001.NO\_TABLE に対する SQL 操作が試みられた後、次のようなメッセージが SQLException.getMessage によって戻されます。

```
ADMF001.NO_TABLE IS AN UNDEFINED NAME. SQLCODE=-204,  
SQLSTATE=42704, DRIVER=3.50.54
```

retrieveMessagesFromServerOnGetMessage を false に設定すると、次のようなメッセージが戻されます。

```
DB2 SQL Error: SQLCODE=-204, SQLSTATE=42704, DRIVER=3.50.54
```

このプロパティを true に設定する代わりに、アプリケーションで IBM Data Server Driver for JDBC and SQLJ 限定の DB2Sqlca.getMessage メソッドを使用することもできます。いずれの技法でも、ストアード・プロシージャが呼び出されて作業単位が開始されます。

## **IBM Informix Dynamic Server および DB2 Database for Linux, UNIX, and Windows データベース・サーバーに共通の IBM Data Server Driver for JDBC and SQLJ のプロパティ**

IBM Data Server Driver for JDBC and SQLJ の一部のプロパティは、IBM Informix Dynamic Server (IDS) および DB2 Database for Linux, UNIX, and Windows データベース・サーバーに適用されます。

それらのプロパティは以下のとおりです。

#### **currentLockTimeout**

ロックを即時に取得できないときに、DB2 Database for Linux, UNIX, and Windows サーバーがロックを待機するかどうかを指定します。このプロパティのデータ・タイプは int です。可能な値は以下のとおりです。

*integer* 整数の秒数 待機します。 *integer* は、-1 から 32767 までです。

#### **LOCK\_TIMEOUT\_NO\_WAIT**

ロックを待機しません。これはデフォルトです。

#### **LOCK\_TIMEOUT\_WAIT\_INDEFINITELY**

無期限にロックを待機します。

#### **LOCK\_TIMEOUT\_NOT\_SET**

データ・ソースにデフォルトを使用します。



### queryDataSize

各フェッチ操作でデータ・ソースから戻される、照会データの量を制御するために使用されるヒントをバイト単位で指定します。この値は、データの検索に必要なデータ・ソースへの往復の数を制御することで、アプリケーションを最適化するために使用できます。

queryDataSize にさらに大きな値を使用すると、ネットワーク・トラフィックは少なくなり、結果としてパフォーマンスを向上させることができます。例えば、結果セットのサイズが 50 KB であり、queryDataSize の値が 32768 (32 KB) の場合、結果セットを検索するにはデータベース・サーバーへの 2 回の往復が必要です。しかし、queryDataSize が 61440 (60 KB) に設定されていれば、結果セットを検索するために必要なデータ・ソースへの往復は 1 回だけです。

以下の表では、各データ・ソースの queryDataSize の最小値、最大値、およびデフォルト値をリストしています。

表 33. queryDataSize の最小値、最大値、およびデフォルト値

データ・ソース	最小 queryDataSize 値	最大 queryDataSize 値	デフォルトの queryDataSize 値
DB2 Database for Linux, UNIX, and Windows	4096	65535	32767
IDS	4096	10485760	32767
DB2 for i5/OS	4096	65535	32767
DB2 for z/OS	適用外	適用外	照会データ・サイズは常に 32767 です。

## DB2 Database for Linux, UNIX, and Windows 用の IBM Data Server Driver for JDBC and SQLJ のプロパティ

IBM Data Server Driver for JDBC and SQLJ の一部のプロパティは、DB2 Database for Linux, UNIX, and Windows サーバーだけに適用されます。

それらのプロパティは以下のとおりです。

### connectNode

アプリケーションの接続先のターゲット・データベース・パーティション・サーバーを指定します。このプロパティのデータ・タイプは int です。値は 0 から 999 の範囲から選択できます。デフォルトは、ポート 0 で定義されたデータベース・パーティション・サーバーです。connectNode は DB2 Database for Linux, UNIX, and Windows サーバーへの IBM Data Server Driver for JDBC and SQLJ Type 4 接続にのみ適用されます。

### currentExplainMode

CURRENT EXPLAIN MODE 特殊レジスタの値を指定します。CURRENT EXPLAIN MODE 特殊レジスタを使用して、EXPLAIN 機能を有効/無効にすることができます。このプロパティのデータ・タイプは String です。最大長は 254 バイトです。このプロパティは、DB2 Database for Linux, UNIX, and Windows などの、CURRENT EXPLAIN MODE 特殊レジスタをサポートするデータ・ソースへの接続にのみ適用されます。

### **currentExplainSnapshot**

CURRENT EXPLAIN SNAPSHOT 特殊レジスタの値を指定します。  
CURRENT EXPLAIN SNAPSHOT 特殊レジスタを使用して、Explain スナップショット機能を有効/無効にすることができます。このプロパティのデータ・タイプは String です。最大長は 8 バイトです。このプロパティは、DB2 Database for Linux, UNIX, and Windows などの、CURRENT EXPLAIN SNAPSHOT 特殊レジスタをサポートするデータ・ソースへの接続にのみ適用されます。

### **currentQueryOptimization**

データベース・マネージャーが動的 SQL ステートメントのバインド時に実行する照会最適化のクラスを制御する値を指定します。このプロパティのデータ・タイプは int です。currentQueryOptimization に指定できる値は次のとおりです。

- 0 最小限の最適化を実行してアクセス・プランを生成することを指定します。このクラスは、的確に索引付けされた表への単純な動的 SQL アクセスの場合に最適です。
- 1 DB2 Database for Linux, UNIX, and Windows バージョン 1 とほぼ同等の最適化を実行してアクセス・プランを生成することを指定します。
- 2 最適化のレベルを DB2 Database for Linux, UNIX, and Windows バージョン 1 のレベルより高く指定しますが、特に非常に複雑な照会の場合、最適化コストはレベル 3 以上よりかなり低く指定されます。
- 3 中程度の最適化を実行してアクセス・プランを生成することを指定します。
- 5 かなりの量の最適化を実行してアクセス・プランを生成することを指定します。複雑な動的 SQL 照会の場合、アクセス・プランの選択に費やす時間を制限するためにヒューリスティック・ルールが使用されます。可能なかぎり、照会は、基礎となる基本表ではなくマテリアライズ照会表を使用します。
- 7 かなりの量の最適化を実行してアクセス・プランを生成することを指定します。この値は 5 に似ていますが、ヒューリスティック・ルールは使用しません。
- 9 最大限の最適化を実行してアクセス・プランを生成することを指定します。これによって、評価対象とすることができるアクセス・プランの数が大幅に増加します。このクラスを使用する必要があるのは、大きな表を使用する非常に複雑でしかも非常に長期間にわたって実行される照会用として、より優れたアクセス・プランを生成できるかどうかを判断する場合です。Explain およびパフォーマンス測定を使用して、より優れたプランが生成されたことを検証することができます。

### **optimizationProfile**

SQL の最適化中に使用される最適化プロファイルを指定します。このプロパティのデータ・タイプは String です。optimizationProfile 値は OPTIMIZATION PROFILE 特殊レジスタの設定に使用されます。デフォルトは NULL です。

optimizationProfile は DB2 Database for Linux, UNIX, and Windows サーバーだけに適用されます。

### **optimizationProfileToFlush**

最適化プロファイル・キャッシュから除去される最適化プロファイルの名前を指定します。このプロパティのデータ・タイプは String です。デフォルトは NULL です。

### **plugin**

クライアント・サイドの JDBC セキュリティー・プラグインの名前。このプロパティは Object タイプを持ち、JDBC セキュリティー・プラグイン・メソッドの新規インスタンスを収容します。

### **pluginName**

サーバー・サイドのセキュリティー・プラグイン・モジュールの名前。

### **useTransactionRedirect**

パフォーマンス向上のために DB2 システムが SQL ステートメントを異なるデータベース・パーティションに送信するかどうかを指定します。このプロパティのデータ・タイプは boolean です。デフォルトは false です。

このプロパティは、以下の条件下でのみ適用されます。

- 接続先がデータベース・パーティション・フィーチャー (DPF) を使用する DB2 Database for Linux, UNIX, and Windows サーバーである。
- パーティション・キーがトランザクション全体で一定のままである。

useTransactionRedirect が true の場合、IBM Data Server Driver for JDBC and SQLJ は接続要求を、トランザクション内の最初の送信可能ステートメントのターゲット・データを含む DPF ノードに送信します。次に DB2 Database for Linux, UNIX, and Windows は、必要に応じて SQL ステートメントを別のパーティションに送信します。

## **DB2 for z/OS 用の IBM Data Server Driver for JDBC and SQLJ のプロパティ**

IBM Data Server Driver for JDBC and SQLJ の一部のプロパティは、DB2 for z/OS サーバーだけに適用されます。

それらのプロパティは以下のとおりです。

### **accountingInterval**

DB2 for z/OS アカウント・レコードをコミット・ポイントで作成するか、データ・ソースへの物理接続の終了時に作成するかを指定します。このプロパティのデータ・タイプは String です。accountingInterval の値が "COMMIT" の場合、アカウント・レコードはコミット・ポイントで作成されます。それ以外の場合、アカウント・レコードはデータ・ソースへの物理接続の終了時に作成されます。

accountingInterval は、DB2 for z/OS 上の IBM Data Server Driver for JDBC and SQLJ Type 2 接続に対してのみ適用されます。accountingInterval は、CICS または IMS のもとでの接続や、Java ストアード・プロシージャーでの接続に対しては適用されません。

accountingInterval プロパティは、db2.jcc.accountingInterval 構成プロパティをオーバーライドします。

### **charOutputSize**

Types.CHAR として登録される INOUT または OUT ストアード・プロシージャ・パラメーターに使用するバイトの最大数を指定します。charOutputSize は、DB2 for z/OS データベース・サーバーへの IBM Data Server Driver for JDBC and SQLJ Type 2 接続にのみ適用されます。

実行時にはストアード・プロシージャ INOUT および OUT パラメーターの DESCRIBE 情報を使用できないため、デフォルトでは、IBM Data Server Driver for JDBC and SQLJ は INOUT または OUT 文字パラメーターの最大長をそれぞれ 32767 に設定します。ストアード・プロシージャに多数の Types.CHAR パラメーターが含まれている場合、この最大値の設定によって、必要量を大きく超えるストレージが割り振られる可能性があります。

ストレージをより効率的に使用するには、それぞれの Types.CHAR INOUT または OUT パラメーターに対して、予想される最大長を charOutputSize に設定してください。

charOutputSize は、Types.VARCHAR または Types.LONGVARCHAR として登録された INOUT または OUT パラメーターには影響を与えません。Types.VARCHAR および Types.LONGVARCHAR パラメーターの場合、ドライバーはデフォルトの長さ 32767 を使用します。

charOutputSize の値を選択するとき、文字変換中の拡張の可能性を考慮に入れる必要があります。IBM Data Server Driver for JDBC and SQLJ は、出力パラメーター値に使用されるサーバー・サイドの CCSID に関する情報を持たないため、ドライバーはストアード・プロシージャの出力データを UTF-8 Unicode 形式で要求します。charOutputSize 値は、パラメーター値が UTF-8 Unicode に変換された後に必要とされる最大バイト数でなければなりません。UTF-8 Unicode 文字は、最大で 3 バイトを必要とします。(3 バイトの UTF-8 文字の例としてユーロ記号があります。) 出力データに関する情報がない場合に charOutputSize を十分に大きな値にするには、最も長い CHAR パラメーターに定義されている長さの 3 倍に charOutputSize を設定してください。

### **clientAccountingInformation**

接続の現在のクライアントに関するアカウントリング情報を指定します。この情報は、クライアントのアカウントリングを目的としています。この値は、接続中に変更することもできます。このプロパティのデータ・タイプは String です。DB2 for z/OS サーバーの場合、最大長は 255 バイトです。Java 空ストリング ("" ) はこの値として有効ですが、Java NULL 値は無効です。

このプロパティは、DB2 for z/OS 上の IBM Data Server Driver for JDBC and SQLJ Type 2 接続に対してのみ適用されます。

### **clientApplicationInformation**

エンド・ユーザーのアプリケーションの、アプリケーションまたはトランザクションの名前を指定します。このプロパティを使用して、クライアント・エンド・ユーザーの ID をアカウントリングおよびモニターのために提供できます。この値は、接続中に変更することもできます。このプロパティのデータ・タイプは String です。DB2 for z/OS サーバーの場合、最大長は 32 バイトです。Java 空ストリング ("" ) はこの値として有効ですが、Java NULL 値は無効です。

このプロパティーは、DB2 for z/OS 上の IBM Data Server Driver for JDBC and SQLJ Type 2 接続に対してのみ適用されます。

#### **clientDebugInfo**

CLIENT DEBUGINFO 接続属性の値を指定します。これは DB2 for z/OS サーバーに、接続を使用しているストアード・プロシージャおよびユーザー定義関数がデバッグ・モードで実行していることを通知します。CLIENT DEBUGINFO は、DB2 統合デバッガーにより使用されます。このプロパティーのデータ・タイプは String です。最大長は 254 バイトです。

このプロパティーは、DB2 for z/OS 上の IBM Data Server Driver for JDBC and SQLJ Type 2 接続に対してのみ適用されます。

#### **clientProgramId**

エンド・ユーザーを識別するために使用できるクライアント・プログラム ID の値を指定します。このプロパティーのデータ・タイプは String で、長さは 80 バイトです。プログラム ID 値が 80 バイトより小さい場合、値にはブランクが埋め込まれます。

このプロパティーは、DB2 for z/OS 上の IBM Data Server Driver for JDBC and SQLJ Type 2 接続に対してのみ適用されます。

#### **clientProgramName**

クライアントの物理接続の継続期間中固定されるアプリケーション ID を指定します。このプロパティーの値は、DB2 for z/OS サーバー上では相関 ID となります。データベース管理者はこのプロパティーを使用して、DB2 for z/OS サーバー上の作業をクライアント・アプリケーションに関連付けることができます。このプロパティーのデータ・タイプは String です。最大長は 12 バイトです。この値をnullにすると、IBM Data Server Driver for JDBC and SQLJ によって *db2jccthread-name* の値が提供されます。

#### **clientUser**

接続のための現行クライアントのユーザー名を指定します。この情報は、クライアントのアカウントिंगを目的としています。JDBC 接続ユーザー名とは異なり、この値は接続中に変更することもできます。DB2 for z/OS サーバーの場合、最大長は 16 バイトです。

このプロパティーは、DB2 for z/OS 上の IBM Data Server Driver for JDBC and SQLJ Type 2 接続に対してのみ適用されます。

#### **clientWorkstation**

接続対象の現行クライアントのワークステーション名を指定します。この情報は、クライアントのアカウントिंगを目的としています。この値は、接続中に変更することもできます。このプロパティーのデータ・タイプは String です。DB2 for z/OS サーバーの場合、最大長は 18 バイトです。Java 空ストリング ("") はこの値として有効ですが、Java NULL 値は無効です。

このプロパティーは、DB2 for z/OS 上の IBM Data Server Driver for JDBC and SQLJ Type 2 接続に対してのみ適用されます。

#### **currentSQLID**

下記のことを指定します。

- 動的に作成された CREATE、GRANT、および REVOKE SQL ステートメントに対する許可検査で使用される許可 ID。



- 動的に発行された CREATE ステートメントによって作成される、表スペース、データベース、ストレージ・グループ、またはシノニムの所有者。
- 動的 SQL ステートメントで指定される、すべての表、ビュー、別名、および索引名の暗黙修飾子。

currentSQLID は、DB2 for z/OS サーバーの CURRENT SQLID 特殊レジスター内に値を設定します。currentSQLID プロパティが設定されていない場合、デフォルトのスキーマ名は、CURRENT SQLID 特殊レジスター内の値です。

#### **enableConnectionConcentrator**

IBM Data Server Driver for JDBC and SQLJ の接続コンセントレーター機能を使用可能にするかどうかを示します。接続コンセントレーター機能を使用できるのは、DB2 for z/OS サーバーへの接続の場合のみです。

enableConnectionConcentrator のデータ・タイプは boolean です。デフォルトは false です。ただし、enableSysplexWLB を true に設定した場合のデフォルトは true になります。

#### **enableSysplexWLB**

IBM Data Server Driver for JDBC and SQLJ の Sysplex ワークロード・バランシング機能を使用可能にするかどうかを示します。Sysplex ワークロード・バランシング機能を使用できるのは、DB2 for z/OS サーバーへの接続の場合のみです。

enableSysplexWLB のデータ・タイプは boolean です。デフォルトは false です。enableSysplexWLB を true に設定した場合、enableConnectionConcentrator はデフォルトで true に設定されます。

#### **jdbcCollection**

実行時に IBM Data Server Driver for JDBC and SQLJ のインスタンスによって使用される、パッケージのコレクション ID を指定します。jdbcCollection のデータ・タイプは String です。デフォルトは NULLID です。

このプロパティは、DB2Binder -collection オプションとともに使用されます。DB2Binder ユーティリティは、jdbcCollection 値と一致する -collection 値を使用して、サーバーで前もって IBM Data Server Driver for JDBC and SQLJ パッケージをバインドしていなければなりません。

jdbcCollection の設定によって、SQLJ アプリケーションで使用されるコレクションが判別されることはありません。SQLJ に関しては、SQLJ カスタマイザーの -collection オプションでコレクションが判別されます。

jdbcCollection は、DB2 for z/OS上の IBM Data Server Driver for JDBC and SQLJ Type 2 接続に対しては適用されません。

#### **maxTransportObjects**

関連 DataSource オブジェクトとのすべての接続で使用できるトランスポート・オブジェクトの最大数を指定します。IBM Data Server Driver for JDBC and SQLJ はトランスポート・オブジェクトおよびグローバル・トランスポート・オブジェクト・プールを使用して、接続コンセントレーターおよび Sysplex ワークロード・バランシングをサポートします。データ・ソースへの各物理接続ごとに、トランスポート・オブジェクトが 1 つあります。

このプロパティのデータ・タイプは int です。

enableConnectionConcentrator または enableSysplexWLB プロパティを設定せず、接続コンセントレーターまたは Sysplex ワークロード・บาลancingの使用が有効になっていない場合、maxTransportObjects 値は無視されます。

maxTransportObjects 値に達していない場合に、グローバル・トランスポート・オブジェクト・プール内のトランスポート・オブジェクトが使用可能でないと、新規のトランスポート・オブジェクトがプールで作成されます。

maxTransportObjects 値に達した場合、アプリケーションは、db2.jcc.maxTransportObjectWaitTime 構成プロパティに指定されている期間待機します。その期間が経過した後も使用可能なトランスポート・オブジェクトがプール内にない場合、プールから SQLException がスローされます。

maxTransportObjects は db2.jcc.maxTransportObjects 構成プロパティをオーバーライドしません。maxTransportObjects は、他の DataSource オブジェクトからの接続に対しては何の効果もありません。maxTransportObjects 値のほうが db2.jcc.maxTransportObjects 値より大きくても、maxTransportObjects によって db2.jcc.maxTransportObjects 値が増加するわけではありません。

maxTransportObjects のデフォルト値は -1 です。これは、ドライバーの db2.jcc.maxTransportObjects 値によってのみ、DataSource のトランスポート・オブジェクト数が制限されることを意味します。

#### **queryCloseImplicit**

すべての行を取り出した後、すぐにカーソルをクローズするかどうかを指定します。queryCloseImplicit は、DB2 for z/OS データベース・サーバーに対する IBM Data Server Driver for JDBC and SQLJ Type 4 接続にのみ適用されます。有効な値は DB2BaseDataSource.QUERY\_CLOSE\_IMPLICIT\_YES (1) および DB2BaseDataSource.QUERY\_CLOSE\_IMPLICIT\_NO (2) です。デフォルトは DB2BaseDataSource.QUERY\_CLOSE\_IMPLICIT\_YES です。

DB2BaseDataSource.QUERY\_CLOSE\_IMPLICIT\_YES の値のほうが優れたパフォーマンスを提供できます。なぜなら、この設定の結果としてネットワーク・トラフィックが減るからです。

#### **sendCharInputsUTF8**

IBM Data Server Driver for JDBC and SQLJ が文字入力データを DB2 for z/OS データベース・サーバーの CCSID に変換するか、またはデータベース・サーバーによる変換のために UTF-8 エンコードでデータを送信するかを指定します。sendCharInputsUTF8 は、DB2 for z/OS データベース・サーバーに対する IBM Data Server Driver for JDBC and SQLJ Type 2 接続にのみ適用されます。このプロパティのデータ・タイプは int です。このプロパティがドライバー・レベル (db2.jcc.sendCharInputsUTF8) でも設定されている場合、この値はドライバー・レベルの値をオーバーライドします。

可能な値は以下のとおりです。

##### **com.ibm.db2.jcc.DB2BaseDataSource.NO (2)**

データが DB2 for z/OS データベース・サーバーに送信される前に、IBM Data Server Driver for JDBC and SQLJ が文字入力データをターゲット・エンコードに変換することを指定します。

com.ibm.db2.jcc.DB2BaseDataSource.NO がデフォルトです。

##### **com.ibm.db2.jcc.DB2BaseDataSource.YES (1)**

IBM Data Server Driver for JDBC and SQLJ が文字入力データを



UTF-8 エンコードで DB2 for z/OS データベース・サーバーに送信することを指定します。データベース・サーバーはデータを UTF-8 エンコードからターゲット CCSID に変換します。

com.ibm.db2.jcc.DB2BaseDataSource.YES は、SDK for Java によるターゲット CCSID への変換によって文字変換問題が発生する場合にのみ指定します。IBM Data Server Driver for JDBC and SQLJ Type 2 接続を使用して Unicode 改行文字 (U+000A) を CCSID 37 である表列に挿入し、そのデータを z/OS 以外のクライアントから取り出すときに、最も一般的な問題が発生します。文字を列に挿入するときに SDK for Java が変換を行う場合、改行文字は EBCDIC 改行文字 X'15' に変換されません。しかし、取り出し時には、z/OS 以外のオペレーティング・システム上の一部の SDK for Java では X'15' 文字を、改行文字 (U+000A) ではなく Unicode 改行文字 (U+0085) に変換します。改行文字により、一部の XML パーサーが予期しない動作を起こします。

sendCharInputsUTF8 を com.ibm.db2.jcc.DB2BaseDataSource.YES に設定する場合、DB2 for z/OS データベース・サーバーは列への挿入時に U+000A 文字を EBCDIC 改行文字 X'25' に変換します。そのため、文字は常に改行文字として取り出されます。

データベース・サーバー上でデータをターゲット CCSID に変換すると、IBM Data Server Driver for JDBC and SQLJ が、ドライバーによる変換の場合よりも多くのメモリーを使用することがあります。ドライバーは文字データの変換用のメモリーを、ソース・エンコードからデータベース・サーバーに送信されるデータのエンコードに割り振ります。ドライバーが表列に送信される文字データに割り振るスペースの量は、データの最大長に基づいています。UTF-8 データは各文字に最大で 3 バイトを必要とします。そのため、ドライバーが UTF-8 データをデータベース・サーバーに送信する場合、ドライバーは入力データ内の最大文字数の 3 倍を割り振る必要があります。ドライバーが変換を行う場合、ターゲット CCSID が 1 バイト CCSID であれば、ドライバーは入力データ内の最大文字数のみ割り振る必要があります。

### **sqljEnableClassLoaderSpecificProfiles**

IBM Data Server Driver for JDBC and SQLJ が、複数の J2EE アプリケーション (.ear) ファイルで同じ Java 名を持つ SQLJ プロファイルの使用およびロードを許可するかどうかを指定します。このプロパティーのデータ・タイプは boolean です。デフォルトは false です。sqljEnableClassLoaderSpecificProfiles は DataSource プロパティーです。このプロパティーは主として WebSphere Application Server での使用を意図しています。

### **useRowsetCursor**

データ・ソースが複数行フェッチをサポートする場合、IBM Data Server Driver for JDBC and SQLJ が常に両方向スクロール・カーソルに複数行 FETCH を使用するかどうかを指定します。このプロパティーのデータ・タイプは boolean です。デフォルトは true です。useRowsetCursor は、DB2 for z/OS データベース・サーバーへの接続にのみ適用されます。

位置指定の更新または削除操作を実行するために JDBC 1 技法を使用するアプリケーションは、useRowSetCursor を false に設定することにより、引き続き希望する結果を実現することができます。JDBC 1 技法には、

ResultSet.setCursorName メソッドを使用して、ResultSet のカーソルの名前を取得して、以下の形式の位置指定 UPDATE または位置指定 DELETE ステートメントを定義することが含まれます。

```
UPDATE table SET col1=value1,...coln=valueN WHERE CURRENT OF cursorname  
DELETE FROM table WHERE CURRENT OF cursorname
```

これらのアプリケーションは、IBM Data Server Driver for JDBC and SQLJ が複数行 FETCH を使用している場合には正しく作動しません。useRowSetCursor を false に設定すると、各フェッチ操作は必ず単一行を戻します。

## IBM Informix Dynamic Server 用の IBM Data Server Driver for JDBC and SQLJ のプロパティー

IBM Data Server Driver for JDBC and SQLJ の一部のプロパティーは、IBM Informix Dynamic Server (IDS) データベースにのみ適用されます。これらのプロパティーは IDS 環境変数に相当します。

IDS 固有のプロパティーは大文字で指定する必要があります。IDS 固有のプロパティー用の getXXX および setXXX メソッドは、get または set が先頭に付く大文字のプロパティー名によって形成されます。以下に例を示します。

```
boolean dbDate = DB2BaseDataSource.getDBDATE();
```

IDS 固有のプロパティーは以下のとおりです。

### DBANSIWARN

SQL ステートメントで ANSI 規格の構文が使用されていない場合、IBM Data Server Driver for JDBC and SQLJ が IDS データベースに、SQLWarning をアプリケーションに戻すよう指示するかどうかを指定します。このプロパティーのデータ・タイプは boolean です。可能な値は以下のとおりです。

#### false または 0

SQL ステートメントで ANSI 規格の構文が使用されていない場合、SQLWarning をアプリケーションに戻すように IDS データベースに指示する値をこのデータベースに送信しません。これはデフォルトです。

#### true または 1

SQL ステートメントで ANSI 規格の構文が使用されていない場合、SQLWarning をアプリケーションに戻すように IDS データベースに指示する値をこのデータベースに送信します。

IBM Data Server Driver for JDBC and SQLJ プロパティー DBANSIWARN を使用して DBANSIWARN IDS プロパティーを設定することはできますが、IBM Data Server Driver for JDBC and SQLJ プロパティー DBANSIWARN を使用して DBANSIWARN IDS プロパティーをリセットすることはできません。

### DBDATE

DATE 値のエンド・ユーザー形式を指定します。このプロパティーのデータ・タイプは String です。指定できる値については、「*IBM Informix Guide to SQL: Reference*」の DBDATE 環境変数の説明を参照してください。

デフォルト値は "Y4MD-" です。

## DBPATH

データベースを含んでいるデータベース・サーバーを識別する値のコロン区切りリストを指定します。このプロパティのデータ・タイプは `String` です。各値は以下のようにすることができます。

- 絶対パス名
- 相対パス名
- `IDS` データベース・サーバーのサーバー名
- サーバー名および絶対パス名

デフォルトは `."` です。

## DBSPACETEMP

一時表が置かれている既存の `dbspace` のコンマ区切りリストまたはコロン区切りリストを指定します。このプロパティのデータ・タイプは `String` です。

このプロパティが設定されていない場合、サーバーに送信される値はありません。 `DBSPACETEMP` 環境変数の値が使用されます。

## DBTEMP

一時ファイルおよび一時表が置かれている既存のディレクトリーの絶対パス名を指定します。このプロパティのデータ・タイプは `String` です。デフォルトは `"/tmp"` です。

## DBUPSPACE

`UPDATE STATISTICS` ステートメントが複数の列分布を同時に構成するときを使用できるシステム・ディスク・スペースの最大量およびメモリーの最大量を `K` バイト (KB) 単位で指定します。このプロパティのデータ・タイプは `String` です。

`DBUPSPACE` の形式は `"maximum-disk-space:maximum-memory"` です。

このプロパティが設定されていない場合、サーバーに送信される値はありません。 `DBUPSPACE` 環境変数の値が使用されます。

## DELIMIDENT

区切り付き `SQL ID` をアプリケーションで使用できるかどうかを指定します。このプロパティのデータ・タイプは `boolean` です。可能な値は以下のとおりです。

**false** アプリケーションに区切り付き `SQL ID` を含めることはできません。二重引用符 (`"`) または単一引用符 (`'`) でリテラル・ストリングを区切ります。これはデフォルトです。

**true** アプリケーションに区切り付き `SQL ID` を含めることができます。区切り付き `SQL ID` は二重引用符 (`"`) で囲む必要があります。単一引用符 (`'`) でリテラル・ストリングを区切ります。

## IFX\_DIRECTIVES

オブティマイザーが照会内からの照会最適化ディレクティブを許可するかどうかを指定します。このプロパティのデータ・タイプは `String` です。可能な値は以下のとおりです。

**"1" または "ON"**

最適化ディレクティブは受け入れられます。

#### **"0" または "OFF"**

最適化ディレクティブは受け入れられません。

このプロパティが設定されていない場合、サーバーに送信される値はありません。IFX\_DIRECTIVES 環境変数の値が使用されます。

#### **IFX\_EXTDIRECTIVES**

オブティマイザーが、sysdirectives システム・カタログ表からの外部照会最適化ディレクティブを、既存のアプリケーションでの照会に適用できるかどうかを指定します。可能な値は以下のとおりです。

#### **"1" または "ON"**

外部照会最適化ディレクティブは受け入れられます。

#### **"0" または "OFF"**

外部照会最適化は受け入れられません。

このプロパティが設定されていない場合、サーバーに送信される値はありません。IFX\_EXTDIRECTIVES 環境変数の値が使用されます。

#### **IFX\_UPDDESC**

UPDATE ステートメントの DESCRIBE が許可されるかどうかを指定します。このプロパティのデータ・タイプは String です。

NULL 以外の値は、UPDATE ステートメントの DESCRIBE が許可されることを示します。デフォルトは "1" です。

#### **IFX\_XASTDCOMPLIANCE\_XAEND**

グローバル・トランザクションが明示的ロールバックの後にのみ解放されるか、またはどのロールバックの後にでも解放されるかを指定します。このプロパティのデータ・タイプは String です。可能な値は以下のとおりです。

**"0"** グローバル・トランザクションは明示的ロールバックの後にのみ解放されます。この動作は X/Open XA 標準に準拠します。

**"1"** グローバル・トランザクションはいずれかのロールバックの後に解放されます。

このプロパティが設定されていない場合、サーバーに送信される値はありません。IFX\_XASTDCOMPLIANCE\_XAEND 環境変数の値が使用されます。

#### **INFORMIXOPCACHE**

クライアント・アプリケーションのステージング・エリア BLOB 領域に対してメモリー・キャッシュのサイズを K バイト (KB) 単位で指定します。このプロパティのデータ・タイプは String です。値 "0" は、キャッシュが使用されないことを示します。

このプロパティが設定されていない場合、サーバーに送信される値はありません。INFORMIXOPCACHE 環境変数の値が使用されます。

#### **INFORMIXSTACKSIZE**

データベース・サーバーがクライアント・セッションの 1 次スレッドに使用するスタック・サイズを K バイト (KB) 単位で指定します。このプロパティのデータ・タイプは String です。

このプロパティが設定されていない場合、サーバーに送信される値はありません。INFORMIXSTACKSIZE 環境変数の値が使用されます。

## NODEFDAC

ANSI 準拠でないデータベースで、新規表が現行セッション中に作成されるときに、データベース・サーバーがデフォルトの表特権 (SELECT、INSERT、UPDATE、および DELETE) が PUBLIC に付与されないようにするかどうかを指定します。このプロパティのデータ・タイプは String です。可能な値は以下のとおりです。

**"yes"** ANSI 準拠でないデータベースで、新規表が現行セッション中に作成されるときに、データベース・サーバーはデフォルトの表特権が PUBLIC に付与されないようにします。

**"no"** ANSI 準拠でないデータベースで、新規表が現行セッション中に作成されるときに、データベース・サーバーはデフォルトの表特権が PUBLIC に付与されないようにしません。これはデフォルトです。

## OPTCOMPIND

順序対の表に対して結合操作を実行するための優先メソッドを指定します。このプロパティのデータ・タイプは String です。可能な値は以下のとおりです。

**"0"** オプティマイザーは、可能であればソート・マージ結合またはハッシュ結合よりもネスト・ループ結合を選択します。

**"1"** 分離レベルが反復可能読み取りである場合、オプティマイザーは可能であればソート・マージ結合またはハッシュ結合よりもネスト・ループ結合を選択します。分離レベルが反復可能読み取りでない場合、オプティマイザーはコストに基づいて結合方法を選択します。

**"2"** トランザクション分離モードに関係なく、オプティマイザーはコストに基づいて結合方法を選択します。

このプロパティが設定されていない場合、サーバーに送信される値はありません。OPTCOMPIND 環境変数の値が使用されます。

## OPTOFC

optimize-OPEN-FETCH-CLOSE 機能を使用可能にするかどうかを指定します。このプロパティのデータ・タイプは String です。可能な値は以下のとおりです。

**"0"** アプリケーションのすべてのスレッドについて optimize-OPEN-FETCH-CLOSE 機能を使用不可にします。

**"1"** アプリケーションのすべてのスレッド内のすべてのカーソルについて optimize-OPEN-FETCH-CLOSE 機能を使用可能にします。

このプロパティが設定されていない場合、サーバーに送信される値はありません。OPTOFC 環境変数の値が使用されます。

## PDQPRIORITY

データベース・サーバーが使用する並列処理の度合いを指定します。

PDQPRIORITY 値は、データベース・サーバーがリソース (メモリー、プロセッサ、およびディスク読み取りを含む) を割り振る方法に影響を与えます。このプロパティのデータ・タイプは String です。可能な値は以下のとおりです。

**"HIGH"**

データベース・サーバーがリソースをすべてのユーザー間で割り振る場合、できる限り多くのリソースが照会に割り当てられます。



**"LOW" または "1"**

データベース・サーバーはフラグメント化表から値を並列でフェッチします。

**"OFF" または "0"**

並列処理は使用不可にされています。

このプロパティーが設定されていない場合、サーバーに送信される値はありません。 PDQPRIORITY 環境変数の値が使用されます。

**PSORT\_DBTEMP**

データベース・サーバーがソート操作に使用される一時ファイルを作成するディレクトリーの絶対パス名を指定します。このプロパティーのデータ・タイプは String です。

このプロパティーが設定されていない場合、サーバーに送信される値はありません。 PSORT\_DBTEMP 環境変数の値が使用されます。

**PSORT\_NPROCS**

データベース・サーバーが照会をソートするために使用できるスレッドの最大数を指定します。このプロパティーのデータ・タイプは String です。

PSORT\_NPROCS の最大値は "10" です。

このプロパティーが設定されていない場合、サーバーに送信される値はありません。 PSORT\_NPROCS 環境変数の値が使用されます。

**STMT\_CACHE**

shared-statement キャッシュを使用可能にするかどうかを指定します。このプロパティーのデータ・タイプは String です。可能な値は以下のとおりです。

**"0"** shared-statement キャッシュは使用不可にされています。

**"1"** 512 KB の shared-statement キャッシュが使用可能にされます。

このプロパティーが設定されていない場合、サーバーに送信される値はありません。 STMT\_CACHE 環境変数の値が使用されます。

---

## IBM Data Server Driver for JDBC and SQLJ 構成プロパティー

IBM Data Server Driver for JDBC and SQLJ 構成プロパティーにはドライバー全体にわたる有効範囲があります。

以下の IBM Data Server Driver for JDBC and SQLJ 構成プロパティーのいずれかを設定することができます。すべてのプロパティーはオプションです。

**db2.jcc.currentSchema または db2.jcc.override.currentSchema**

動的に準備される SQL ステートメントで非修飾データベース・オブジェクトを修飾するために使用されるデフォルトのスキーマ名を指定します。このプロパティーのこの値は、データベース・サーバーでの CURRENT SCHEMA 特殊レジスター内の値を設定します。

**decimalRoundingMode**

DB2 for z/OS または DB2 Database for Linux, UNIX, and Windows データベース・サーバー上での 10 進値または 10 進浮動小数点値の丸めモード、および 10 進データ・タイプをサポートするその他のすべてのデータ・ソース上での 10 進値の丸めモードを指定します。

可能な値は以下のとおりです。

**com.ibm.db2.jcc.DB2BaseDataSource.ROUND\_UP (0)**

0 から遠ざかる数値に値を丸めます。破棄される数字がすべてゼロである場合、破棄される数字の除去を除いて結果は未変更のままになります。それ以外の場合、結果の係数は 1 だけ増やされます。この丸めモードは、浮動小数点算術計算の IEEE ドラフト標準によってサポートされていないため、移植可能アプリケーションの作成時には推奨しません。

**com.ibm.db2.jcc.DB2BaseDataSource.ROUND\_DOWN (1)**

0 の方向に丸めます (切り捨て)。破棄された数字は無視されます。

**com.ibm.db2.jcc.DB2BaseDataSource.ROUND\_CEILING (2)**

正の無限大方向に丸めます。破棄される数字がすべてゼロであるか、または符号が負の場合、破棄される数字の除去を除いて結果は未変更のままになります。それ以外の場合、結果の係数は 1 だけ増やされます。

**com.ibm.db2.jcc.DB2BaseDataSource.ROUND\_HALF\_EVEN (3)**

値を最も近い値に丸めます。複数の値が等距離にある場合は、最後の数字が偶数になるように値を丸めます。破棄される数字が、左側の次の桁内の 1 の半分 (0.5) より大きい場合、結果の係数は 1 だけ増やされます。その数字が半分より小さい場合、結果の係数は調整されません (つまり、破棄された数字は無視されます)。それ以外の場合、結果の係数の右端の数字が偶数であれば、その係数は変更されませんが、右端の数字が奇数であれば、その桁が偶数になるように 1 だけ増大 (切り上げ) します。

**com.ibm.db2.jcc.DB2BaseDataSource.ROUND\_HALF\_UP (4)**

値を最も近い値に丸めます。複数の値が等距離にある場合は、0 の反対方向に値を丸めます。破棄される数字が、左側の次の桁内の 1 の半分 (0.5) 以上の場合、結果の係数は 1 だけ増やされます。そうでない場合、破棄された数字は無視されます。

**com.ibm.db2.jcc.DB2BaseDataSource.ROUND\_HALF\_DOWN (5)**

値を最も近い値に丸めます。複数の値が等距離にある場合は、0 の方向に値を丸めます。破棄される数字が、左側の次の桁内の 1 の半分 (0.5) より大きい場合、結果の係数は 1 だけ増やされます。そうでない場合、破棄された数字は無視されます。この丸めモードは、浮動小数点算術計算の IEEE ドラフト標準によってサポートされていないため、移植可能アプリケーションの作成時には推奨しません。

**com.ibm.db2.jcc.DB2BaseDataSource.ROUND\_FLOOR (6)**

負の無限大方向に丸めます。破棄される数字がすべてゼロであるか、または符号が正の場合、破棄される数字の除去を除いて、結果は未変更のままになります。それ以外の場合、符号は負で、結果の係数は 1 だけ増やされます。

**com.ibm.db2.jcc.DB2BaseDataSource.ROUND\_UNSET (-2147483647)**

丸めモードは明示的に設定されませんでした。IBM Data Server Driver for JDBC and SQLJ は、decimalRoundingMode を使用してデータ・ソース上で丸めモードを設定することはしません。



IBM Data Server Driver for JDBC and SQLJ はその丸めモードに以下の値を使用します。

- データ・ソースが DB2 for z/OS または DB2 Database for Linux, UNIX, and Windows である場合、丸めモードは 10 進値または 10 進浮動小数点値については ROUND\_HALF\_EVEN です。
- その他のデータ・ソースの場合、丸めモードは 10 進値については ROUND\_DOWN です。

#### **db2.jcc.jmxEnabled**

Java Management Extensions (JMX) が IBM Data Server Driver for JDBC and SQLJ インスタンスについて使用可能にされるかどうかを指定します。アプリケーションがリモート・トレース・コントローラーを使用できるようにするには、その前に JMX を使用可能にする必要があります。

可能な値は以下のとおりです。

#### **true または yes**

JMX が使用可能にされることを示します。

#### **その他の値**

JMX が使用不可にされることを示します。これはデフォルトです。

値

#### **db2.jcc.traceDirectory または db2.jcc.override.traceDirectory**

Java ドライバー・コードの IBM Data Server Driver for JDBC and SQLJ トレースを使用可能にし、トレース情報を書き込むディレクトリーを指定します。db2.jcc.override.traceDirectory を指定すると、同一の DataSource 上にある複数の接続のトレース情報が、複数のファイルに書き込まれます。

db2.jcc.override.traceDirectory を指定すると、接続は *file-name\_origin\_n* という名前のファイルにトレースされます。

- *n* は、DataSource の *n* 番目の接続です。
- db2.jcc.traceFileName と db2.jcc.override.traceFileName のどちらも指定されない場合、*file-name* は traceFile になります。db2.jcc.traceFileName または db2.jcc.override.traceFileName も指定されている場合、*file-name* は db2.jcc.traceFileName または db2.jcc.override.traceFileName の値になります。
- *origin* は、使用中のログ書き込みプログラムの起点を示します。origin に指定できる値は次のとおりです。

**cpds** DB2ConnectionPoolDataSource オブジェクトのログ書き込みプログラム。

**driver** DB2Driver オブジェクトのログ書き込みプログラム。

**global** DB2TraceManager オブジェクトのログ書き込みプログラム。

**sds** DB2SimpleDataSource オブジェクトのログ書き込みプログラム。

**xads** DB2XADDataSource オブジェクトのログ書き込みプログラム。

db2.jcc.override.traceDirectory プロパティーは、Connection または DataSource オブジェクトの traceDirectory プロパティーをオーバーライドします。

例えば、`db2.jcc.override.traceDirectory` の設定を以下のように指定することによって、IBM Data Server Driver for JDBC and SQLJ の Java コードを `/SYSTEM/tmp` という名前のディレクトリーにあるファイルにトレースすることができるようになります。

```
db2.jcc.override.traceDirectory=/SYSTEM/tmp
```

トレースのプロパティーは、IBM ソフトウェア・サポートの指示の下でのみ設定してください。

### **db2.jcc.traceLevel または db2.jcc.override.traceLevel**

何をトレースするかを指定します。

`db2.jcc.override.traceLevel` プロパティーは、`Connection` または `DataSource` オブジェクトの `traceLevel` プロパティーをオーバーライドします。

10 進値を指定して、1 つ以上のトレース・レベルを指定します。このトレース・レベルは、`Connection` または `DataSource` オブジェクトの `traceLevel` プロパティーに定義されているトレース・レベルと同じです。

複数のトレース・レベルを指定する場合には、値に対して OR (|) 演算を行い、`db2.jcc.traceLevel` または `db2.jcc.override.traceLevel` の指定に結果を 10 進数で指定します。

例えば、`TRACE_DRDA_FLOWS` および `TRACE_CONNECTIONS` を `db2.jcc.override.traceLevel` に指定するとします。`TRACE_DRDA_FLOWS` の値は 16 進値 `X'40'` です。`TRACE_CONNECTION_CALLS` の値は 16 進値 `X'01'` です。両方のトレースを指定するには、ビット単位 OR 演算をその 2 つの値に対して行います。結果は `X'41'` になります。これと等価の 10 進数は 65 です。それで、次のように指定します。

```
db2.jcc.override.traceLevel=65
```

### **db2.jcc.sqljUncustomizedWarningOrException**

カスタマイズされていない SQLJ アプリケーションが実行されるときに IBM Data Server Driver for JDBC and SQLJ が行うアクションを指定します。

`db2.jcc.sqljUncustomizedWarningOrException` は、次のような値をとることができます。

- 0 カスタマイズされていない SQLJ アプリケーションが実行されるとき、IBM Data Server Driver for JDBC and SQLJ は Warning または Exception をスローしません。これはデフォルトです。
- 1 カスタマイズされていない SQLJ アプリケーションが実行されるとき、IBM Data Server Driver for JDBC and SQLJ は Warning をスローします。
- 2 カスタマイズされていない SQLJ アプリケーションが実行されるとき、IBM Data Server Driver for JDBC and SQLJ は Exception をスローします。

### **db2.jcc.traceFile または db2.jcc.override.traceFile**

Java ドライバー・コードの IBM Data Server Driver for JDBC and SQLJ トレースを使用可能にし、トレース・ファイル名の基になる名前を指定します。

`db2.jcc.override.traceFile` プロパティーの値には、完全修飾 z/OS UNIX システム・サービス・ファイル名を指定します。

db2.jcc.override.traceFile プロパティは、Connection または DataSource オブジェクトの traceFile プロパティをオーバーライドします。

例えば、db2.jcc.override.traceFile の設定を以下のように指定することによって、IBM Data Server Driver for JDBC and SQLJ の Java コードを /SYSTEM/tmp/jdbctrace という名前のファイルにトレースすることができるようになります。

```
db2.jcc.override.traceFile=/SYSTEM/tmp/jdbctrace
```

トレースのプロパティは、IBM ソフトウェア・サポートの指示の下でのみ設定してください。

#### **db2.jcc.traceFileAppend または db2.jcc.override.traceFileAppend**

db2.jcc.override.traceFile プロパティによって指定されるファイルへ付加するか、あるいは上書きするかを指定します。このプロパティのデータ・タイプは boolean です。デフォルトは false です。これは、traceFile プロパティによって指定されるファイルが上書きされることを意味します。

db2.jcc.override.traceFileAppend プロパティは、Connection または DataSource オブジェクトの traceFileAppend プロパティをオーバーライドします。

例えば、db2.jcc.override.traceFileAppend の設定を以下のように指定することによって、トレース・データは既存のトレース・ファイルに追加されるようになります。

```
db2.jcc.override.traceFileAppend=true
```

トレースのプロパティは、IBM ソフトウェア・サポートの指示の下でのみ設定してください。

---

## JDBC API でのドライバーのサポート

DB2 および IBM Informix Dynamic Server (IDS) データベース・システムによってサポートされる JDBC ドライバーは、JDBC メソッドについて異なるレベルのサポートを持っています。

次の表は JDBC インターフェースをリストし、それらがどのドライバーによってサポートされるかを示しています。ドライバーおよびサポートされるプラットフォームは、以下のとおりです。

表 34. DB2 および IDS データベース・システム用の JDBC ドライバー

JDBC ドライバー名	関連するデータ・ソース
IBM Data Server Driver for JDBC and SQLJ	DB2 Database for Linux, UNIX, and Windows, DB2 for z/OS, または IBM Informix Dynamic Server (IDS)
Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバー (非推奨)	DB2 Database for Linux, UNIX, and Windows
IBM Informix JDBC ドライバー (IDS JDBC ドライバー)	IDS

メソッドの形式が JDBC 2.0 および JDBC 3.0 の場合、IBM Data Server Driver for JDBC and SQLJ はすべての形式をサポートします。Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバーは JDBC 2.0 形式のみサポートします。

表 35. Array メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ1 のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバーのサポート	IDS JDBC ドライバーのサポート
free <sup>2</sup>	はい	いいえ	いいえ
getArray	はい	いいえ	はい
getBaseType	はい	いいえ	はい
getBaseTypeName	はい	いいえ	はい
getResultSet	はい	いいえ	はい

注:

1. IBM Data Server Driver for JDBC and SQLJ では、DB2 Database for Linux, UNIX, and Windows Array メソッドがサポートされます。
2. これは JDBC 4.0 のメソッドです。

表 36. BatchUpdateException メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバーのサポート	IDS JDBC ドライバーのサポート
java.lang.Exception から継承されたメソッド	はい	はい	はい
getUpdateCounts	はい	はい	はい

表 37. Blob メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバーのサポート	IDS JDBC ドライバーのサポート
free <sup>1</sup>	はい	いいえ	いいえ
getBinaryStream	はい <sup>2</sup>	はい	はい
getBytes	はい	はい	はい
length	はい	はい	はい
position	はい	はい	はい
setBinaryStream <sup>3</sup>	はい	いいえ	いいえ
setBytes <sup>3</sup>	はい	いいえ	いいえ
truncate <sup>3</sup>	はい	いいえ	いいえ

表 37. Blob メソッドのサポート (続き)

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
-----------	--	---	--------------------

注:

- これは JDBC 4.0 のメソッドです。
- このメソッドのサポートされる形式には、以下の JDBC 4.0 形式が含まれます。

getBinaryStream(long pos, long length)

- このメソッドは、以下の場合には使用できません。
  - fullyMaterializeLobData プロパティが false に設定されている場合。
  - Java ストアード・プロシージャに渡される Blob の場合。

表 38. CallableStatement メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
java.sql.Statement から継承されたメソッド	はい	はい	はい
java.sql.PreparedStatement から継承されたメソッド	はい <sup>1</sup>	はい	はい
getArray	いいえ	いいえ	いいえ
getBigDecimal	はい <sup>3</sup>	はい	はい
getBlob	はい <sup>3</sup>	はい	はい
getBoolean	はい <sup>3</sup>	はい	はい
getByte	はい <sup>3</sup>	はい	はい
getBytes	はい <sup>3</sup>	はい	はい
getClob	はい <sup>3</sup>	はい	はい
getDate	はい <sup>3, 4</sup>	はい <sup>4</sup>	はい
getDouble	はい <sup>3</sup>	はい	はい
getFloat	はい <sup>3</sup>	はい	はい
getInt	はい <sup>3</sup>	はい	はい
getLong	はい <sup>3</sup>	はい	はい
getObject	はい <sup>3, 5</sup>	はい <sup>5</sup>	はい
getRef	いいえ	いいえ	いいえ
getRowId <sup>2</sup>	はい	いいえ	いいえ
getShort	はい <sup>3</sup>	はい	はい
getString	はい <sup>3</sup>	はい	はい
getTime	はい <sup>3, 4</sup>	はい <sup>4</sup>	はい
getTimestamp	はい <sup>3, 4</sup>	はい <sup>4</sup>	はい
getURL	はい	いいえ	いいえ
registerOutParameter	はい <sup>6</sup>	はい <sup>6</sup>	はい <sup>6</sup>

表 38. *CallableStatement* メソッドのサポート (続き)

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバ のサポート	IDS JDBC ドライバ のサポート
setAsciiStream	はい <sup>7</sup>	いいえ	はい
setBigDecimal	はい <sup>7</sup>	いいえ	はい
setBinaryStream	はい <sup>7</sup>	いいえ	はい
setBoolean	はい <sup>7</sup>	いいえ	はい
setByte	はい <sup>7</sup>	いいえ	はい
setBytes	はい <sup>7</sup>	いいえ	はい
setCharacterStream	はい <sup>7</sup>	いいえ	はい
setDate	はい <sup>7</sup>	いいえ	はい
setDouble	はい <sup>7</sup>	いいえ	はい
setFloat	はい <sup>7</sup>	いいえ	はい
setInt	はい <sup>7</sup>	いいえ	はい
setLong	はい <sup>7</sup>	いいえ	はい
setNull	はい <sup>7, 8</sup>	いいえ	はい
setObject	はい <sup>7</sup>	いいえ	はい
setShort	はい <sup>7</sup>	いいえ	はい
setString	はい <sup>7</sup>	いいえ	はい
setTime	はい <sup>7</sup>	いいえ	はい
setTimestamp	はい <sup>7</sup>	いいえ	はい
setURL	はい	いいえ	いいえ
wasNull	はい	はい	はい

**注:**

1. 継承された `getParameterMetaData` メソッドは、データ・ソースが DB2 for z/OS の場合にはサポートされません。
2. これは JDBC 4.0 のメソッドです。
3. 以下の形式の `CallableStatement.getXXX` メソッドは、データ・ソースが DB2 for z/OS の場合にはサポートされません。  
`getXXX(String parameterName)`
4. データベース・サーバーは日時値の時間帯調整を行いません。 JDBC ドライバーは、`java.util.Calendar` パラメーターを含む `getDate`、`getTime`、または `getTimestamp` メソッドの形式が指定された場合、サーバーから値を取り出した後でローカル時間帯の値を調整します。
5. 次の形式の `getObject` メソッドはサポートされません。  
`getObject(int parameterIndex, java.util.Map map)`
6. 次の形式の `registerOutParameter` メソッドはサポートされません。  
`registerOutParameter(int parameterIndex, int jdbcType, String typeName)`
7. データ・ソースが DB2 for z/OS の場合にはサポートされません。
8. 次の形式の `setNull` はサポートされません。  
`setNull(int parameterIndex, int jdbcType, String typeName)`

表 39. Clob メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバ のサポート	IDS JDBC ドライバ のサポート
free <sup>1</sup>	はい	いいえ	いいえ
getAsciiStream	はい	はい	はい
getCharacterStream	はい <sup>2</sup>	はい	はい
getSubString	はい	はい	はい
length	はい	はい	はい
position	はい	はい	はい
setAsciiStream <sup>3</sup>	はい	いいえ	はい
setCharacterStream <sup>3</sup>	はい	いいえ	はい
setString <sup>3</sup>	はい	いいえ	はい
truncate <sup>3</sup>	はい	いいえ	はい

## 注:

- これは JDBC 4.0 のメソッドです。
- このメソッドのサポートされる形式には、以下の JDBC 4.0 形式が含まれます。  
getCharacterStream(long pos, long length)
- このメソッドは、以下の場合には使用できません。
  - fullyMaterializeLobData プロパティが false に設定されている場合。
  - Java ストアード・プロシージャに渡される Clob の場合。

表 40. Connection メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバ のサポート	IDS JDBC ドライバ のサポート
clearWarnings	はい	はい	はい
close	はい	はい	はい
commit	はい	はい	はい
createStatement	はい	はい <sup>2</sup>	はい
createBlob <sup>1</sup>	はい	いいえ	いいえ
createClob <sup>1</sup>	はい	いいえ	いいえ
getAutoCommit	はい	はい	はい
getCatalog	はい	はい	はい
getClientInfo <sup>3</sup>	はい	いいえ	いいえ
getHoldability	はい	いいえ	いいえ
getMetaData	はい	はい	はい
getTransactionIsolation	はい	はい	はい
getTypeMap	いいえ	いいえ	はい
getWarnings	はい	はい	はい
isClosed	はい	はい	はい



表 40. Connection メソッドのサポート (続き)

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバ のサポート	IDS JDBC ドライバ のサポート
isReadOnly	はい	はい	はい
isValid <sup>3</sup>	はい	いいえ	いいえ
nativeSQL	はい	はい	はい
prepareCall	はい <sup>4</sup>	はい	はい
prepareStatement	はい	はい <sup>2</sup>	はい
releaseSavepoint	はい	いいえ	いいえ
rollback	はい	はい <sup>2</sup>	はい
setAutoCommit	はい	はい	はい
setCatalog	はい	はい	いいえ
setClientInfo <sup>3</sup>	はい	いいえ	いいえ
setReadOnly	はい <sup>5</sup>	はい	いいえ
setSavepoint	はい	いいえ	はい
setTransactionIsolation	はい	はい	はい
setTypeMap	いいえ	いいえ	はい

## 注:

- これは JDBC 4.0 のメソッドです。
- Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバーは、このメソッドの JDBC 3.0 形式をサポートしません。
- これは JDBC 4.0 のメソッドです。
- CALL ステートメント内のストアード・プロシージャが DB2 for z/OS 上にある場合、CALL ステートメントのパラメーターを式にすることはできません。
- ドライバーは設定を使用しません。IBM Data Server Driver for JDBC and SQLJ では、Connection または DataSource オブジェクトの readOnly プロパティにより、接続を読み取り専用として設定することができます。

表 41. ConnectionEvent メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバ のサポート	IDS JDBC ドライバ のサポート
java.util.EventObject から継承されたメソッド	はい	はい	はい
getSQLException	はい	はい	はい

表 42. ConnectionEventListener メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバ のサポート	IDS JDBC ドライバ のサポート
connectionClosed	はい	はい	はい

表 42. *ConnectionEventListener* メソッドのサポート (続き)

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
connectionErrorOccurred	はい	はい	はい

表 43. *ConnectionPoolDataSource* メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
getLoginTimeout	はい	はい	はい
getLogWriter	はい	はい	はい
getPooledConnection	はい	はい	はい
setLoginTimeout	はい <sup>1</sup>	はい	はい
setLogWriter	はい	はい	はい

注:

1. このメソッドは、DB2 for z/OS 上の IBM Data Server Driver for JDBC and SQLJ Type 2 接続ではサポートされません。

表 44. *DatabaseMetaData* メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2 JDBC	
		Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
allProceduresAreCallable	はい	はい	はい
allTablesAreSelectable	はい <sup>1</sup>	はい	はい <sup>1</sup>
dataDefinitionCausesTransactionCommit	はい	はい	はい
dataDefinitionIgnoredInTransactions	はい	はい	はい
deletesAreDetected	はい	はい	はい
doesMaxRowSizeIncludeBlobs	はい	はい	はい
getAttributes	はい <sup>2</sup>	いいえ	いいえ
getBestRowIdentifier	はい	はい	はい
getCatalogs	はい	はい	はい
getCatalogSeparator	はい	はい	はい
getCatalogTerm	はい	はい	はい
getClientInfoProperties <sup>6</sup>	はい	いいえ	いいえ
getColumnPrivileges	はい	はい	はい
getColumns	はい <sup>7</sup>	はい <sup>10</sup>	はい <sup>10</sup>
getConnection	はい	はい	はい
getCrossReference	はい	はい	はい

表 44. DatabaseMetaData メソッドのサポート (続き)

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポー ト	Linux、 UNIX、および Windows 用の DB2 JDBC Type 2 ドライ バーのサポート	IDS JDBC ド ライバーのサポ ート
getDatabaseMajorVersion	はい	いいえ	いいえ
getDatabaseMinorVersion	はい	いいえ	いいえ
getDatabaseProductName	はい	はい	はい
getDatabaseProductVersion	はい	はい	はい
getDefaultTransactionIsolation	はい	はい	はい
getDriverMajorVersion	はい	はい	はい
getDriverMinorVersion	はい	はい	はい
getDriverName	はい <sup>8</sup>	はい	はい
getDriverVersion	はい	はい	はい
getExportedKeys	はい	はい	はい
getFunctionColumns <sup>6</sup>	はい	いいえ	いいえ
getFunctions <sup>6</sup>	はい	いいえ	いいえ
getExtraNameCharacters	はい	はい	はい
getIdentifierQuoteString	はい	はい	はい
getImportedKeys	はい	はい	はい
getIndexInfo	はい	はい	はい
getJDBCMinorVersion	はい	いいえ	いいえ
getJDBCMajorVersion	はい	いいえ	いいえ
getMaxBinaryLiteralLength	はい	はい	はい
getMaxCatalogNameLength	はい	はい	はい
getMaxCharLiteralLength	はい	はい	はい
getMaxColumnNameLength	はい	はい	はい
getMaxColumnsInGroupBy	はい	はい	はい
getMaxColumnsInIndex	はい	はい	はい
getMaxColumnsInOrderBy	はい	はい	はい
getMaxColumnsInSelect	はい	はい	はい
getMaxColumnsInTable	はい	はい	はい
getMaxConnections	はい	はい	はい
getMaxCursorNameLength	はい	はい	はい
getMaxIndexLength	はい	はい	はい
getMaxProcedureNameLength	はい	はい	はい
getMaxRowSize	はい	はい	はい
getMaxSchemaNameLength	はい	はい	はい
getMaxStatementLength	はい	はい	はい
getMaxStatements	はい	はい	はい
getMaxTableNameLength	はい	はい	はい

表 44. DatabaseMetaData メソッドのサポート (続き)

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバーのサポート	IDS JDBC ドライバーのサポート
getMaxTablesInSelect	はい	はい	はい
getMaxUserNameLength	はい	はい	はい
getNumericFunctions	はい	はい	はい
getPrimaryKeys	はい	はい	はい
getProcedureColumns	はい <sup>7</sup> (292 ページ)	はい	はい
getProcedures	はい <sup>7</sup> (292 ページ)	はい	はい
getProcedureTerm	はい	はい	はい
getResultSetHoldability	はい	いいえ	いいえ
getRowIdLifetime <sup>6</sup>	はい	いいえ	いいえ
getSchemas	はい <sup>9</sup> (292 ページ)	はい <sup>10</sup>	はい <sup>10</sup>
getSchemaTerm	はい	はい	はい
getSearchStringEscape	はい	はい	はい
getSQLKeywords	はい	はい	はい
getSQLStateType	はい	いいえ	いいえ
getStringFunctions	はい	はい	はい
getSuperTables	はい <sup>2</sup>	いいえ	いいえ
getSuperTypes	はい <sup>2</sup>	いいえ	いいえ
getSystemFunctions	はい	はい	はい
getTablePrivileges	はい	はい	はい
getTables	はい	はい <sup>10</sup>	はい <sup>10</sup>
getTableTypes	はい	はい	はい
getTimeDateFunctions	はい	はい	はい
getTypeInfo	はい	はい	はい
getUDTs	いいえ	はい <sup>11</sup>	はい <sup>11</sup>
getURL	はい	はい	はい
getUserName	はい	はい	はい
getVersionColumns	はい	はい	はい
insertsAreDetected	はい	はい	はい
isCatalogAtStart	はい	はい	はい
isReadOnly	はい	はい	はい
locatorsUpdateCopy	はい <sup>3</sup>	はい	はい <sup>3</sup>
nullPlusNonNullIsNull	はい	はい	はい
nullsAreSortedAtEnd	はい <sup>4</sup>	はい	はい <sup>4</sup>

表 44. DatabaseMetaData メソッドのサポート (続き)

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバーのサポート	IDS JDBC ドライバーのサポート
nullableSortedAtStart	はい	はい	はい
nullableSortedHigh	はい <sup>5</sup>	はい	はい <sup>5</sup>
nullableSortedLow	はい <sup>1</sup>	はい	はい <sup>1</sup>
othersDeletesAreVisible	はい	はい	はい
othersInsertsAreVisible	はい	はい	はい
othersUpdatesAreVisible	はい	はい	はい
ownDeletesAreVisible	はい	はい	はい
ownInsertsAreVisible	はい	はい	はい
ownUpdatesAreVisible	はい	はい	はい
storesLowerCaseIdentifiers	はい <sup>1</sup>	はい	はい <sup>1</sup>
storesLowerCaseQuotedIdentifiers	はい <sup>4</sup>	はい	はい <sup>4</sup>
storesMixedCaseIdentifiers	はい	はい	はい
storesMixedCaseQuotedIdentifiers	はい	はい	はい
storesUpperCaseIdentifiers	はい <sup>5</sup>	はい	はい <sup>5</sup>
storesUpperCaseQuotedIdentifiers	はい	はい	はい
supportsAlterTableWithAddColumn	はい	はい	はい
supportsAlterTableWithDropColumn	はい <sup>1</sup>	はい	はい <sup>1</sup>
supportsANSI92EntryLevelSQL	はい	はい	はい
supportsANSI92FullSQL	はい	はい	はい
supportsANSI92IntermediateSQL	はい	はい	はい
supportsBatchUpdates	はい	はい	はい
supportsCatalogsInDataManipulation	はい <sup>1</sup>	はい	はい <sup>1</sup>
supportsCatalogsInIndexDefinitions	はい	はい	はい
supportsCatalogsInPrivilegeDefinitions	はい	はい	はい
supportsCatalogsInProcedureCalls	はい <sup>1</sup>	はい	はい <sup>1</sup>
supportsCatalogsInTableDefinitions	はい	はい	はい
SupportsColumnAliasing	はい	はい	はい
supportsConvert	はい	はい	はい
supportsCoreSQLGrammar	はい	はい	はい
supportsCorrelatedSubqueries	はい	はい	はい
supportsDataDefinitionAndDataManipulationTransactions	はい	はい	はい
supportsDataManipulationTransactionsOnly	はい	はい	はい
supportsDifferentTableCorrelationNames	はい <sup>4</sup>	はい	はい <sup>4</sup>
supportsExpressionsInOrderBy	はい	はい	はい
supportsExtendedSQLGrammar	はい	はい	はい
supportsFullOuterJoins	はい <sup>3</sup>	はい	はい <sup>3</sup>

表 44. DatabaseMetaData メソッドのサポート (続き)

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux , UNIX、および Windows 用の DB2 JDBC Type 2 ドライバーのサポート	IDS JDBC ドライバーのサポート
supportsGetGeneratedKeys	はい	いいえ	いいえ
supportsGroupBy	はい	はい	はい
supportsGroupByBeyondSelect	はい	はい	はい
supportsGroupByUnrelated	はい	はい	はい
supportsIntegrityEnhancementFacility	はい	はい	はい
supportsLikeEscapeClause	はい	はい	はい
supportsLimitedOuterJoins	はい	はい	はい
supportsMinimumSQLGrammar	はい	はい	はい
supportsMixedCaseIdentifiers	はい	はい	はい
supportsMixedCaseQuotedIdentifiers	はい <sup>3</sup>	はい	はい <sup>3</sup>
supportsMultipleOpenResults	はい <sup>5</sup>	いいえ	はい <sup>5</sup>
supportsMultipleResultSets	はい <sup>5</sup>	はい	はい <sup>5</sup>
supportsMultipleTransactions	はい	はい	はい
supportsNamedParameters	はい	いいえ	いいえ
supportsNonNullableColumns	はい	はい	はい
supportsOpenCursorsAcrossCommit	はい <sup>3</sup>	はい	はい <sup>3</sup>
supportsOpenCursorsAcrossRollback	はい	はい	はい
supportsOpenStatementsAcrossCommit	はい <sup>3</sup>	はい	はい <sup>3</sup>
supportsOpenStatementsAcrossRollback	はい <sup>3</sup>	はい	はい <sup>3</sup>
supportsOrderByUnrelated	はい	はい	はい
supportsOuterJoins	はい	はい	はい
supportsPositionedDelete	はい	はい	はい
supportsPositionedUpdate	はい	はい	はい
supportsResultSetConcurrency	はい	はい	はい
supportsResultSetHoldability	はい	いいえ	いいえ
supportsResultSetType	はい	はい	はい
supportsSavepoints	はい <sup>5</sup>	いいえ	はい <sup>5</sup>
supportsSchemasInDataManipulation	はい	はい	はい
supportsSchemasInIndexDefinitions	はい	はい	はい
supportsSchemasInPrivilegeDefinitions	はい	はい	はい
supportsSchemasInProcedureCalls	はい	はい	はい
supportsSchemasInTableDefinitions	はい	はい	はい
supportsSelectForUpdate	はい	はい	はい
supportsStoredProcedures	はい	はい	はい
supportsSubqueriesInComparisons	はい	はい	はい
supportsSubqueriesInExists	はい	はい	はい

表 44. DatabaseMetaData メソッドのサポート (続き)

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバーのサポート	IDS JDBC ドライバーのサポート
supportsSubqueriesInIns	はい	はい	はい
supportsSubqueriesInQuantifieds	はい	はい	はい
supportsSuperTables	はい	いいえ	いいえ
supportsSuperTypes	はい	いいえ	いいえ
supportsTableCorrelationNames	はい	はい	はい
supportsTransactionIsolationLevel	はい	はい	はい
supportsTransactions	はい	はい	はい
supportsUnion	はい	はい	はい
supportsUnionAll	はい	はい	はい
updatesAreDetected	はい	はい	はい
usesLocalFilePerTable	はい	はい	はい
usesLocalFiles	はい	はい	はい

**注:**

1. DB2 データ・ソースはこのメソッドに false を戻します。IDS データ・ソースは true を戻します。
2. このメソッドは、DB2 Database for Linux, UNIX, and Windows および IDS への接続の場合のみサポートされません。
3. IBM Data Server Driver for JDBC and SQLJ では、DB2 データ・ソースおよび IDS データ・ソースはこのメソッドに true を戻します。IDS JDBC ドライバーでは、IDS データ・ソースは false を戻します。
4. IBM Data Server Driver for JDBC and SQLJ では、DB2 データ・ソースおよび IDS データ・ソースはこのメソッドに false を戻します。IDS JDBC ドライバーでは、IDS データ・ソースは true を戻します。
5. DB2 データ・ソースはこのメソッドに true を戻します。IDS データ・ソースは false を戻します。
6. これは JDBC 4.0 のメソッドです。
7. このメソッドは、JDBC 4.0 仕様で記述された追加列を戻します。
8. IBM Data Server Driver for JDBC and SQLJ の JDBC 3.0 以前のインプリメンテーションでは、「IBM DB2 JDBC Universal Driver Architecture」を戻します。  
  
IBM Data Server Driver for JDBC and SQLJ の JDBC 4.0 インプリメンテーションでは、「IBM Data Server Driver for JDBC and SQLJ」を戻します。
9. このメソッドの JDBC 4.0 以前の形式がサポートされます。
10. Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバーは、このメソッドの JDBC 3.0 形式をサポートしません。
11. メソッドは実行されますが、空の ResultSet を戻します。



表 45. *DataSource* メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
<code>getConnection</code>	はい	はい	はい
<code>getLoginTimeout</code>	はい	はい <sup>1</sup>	はい
<code>getLogWriter</code>	はい	はい	はい
<code>setLoginTimeout</code>	はい <sup>2</sup>	はい <sup>1</sup>	はい
<code>setLogWriter</code>	はい	はい	はい

## 注:

1. DB2 JDBC Type 2 ドライバはこの設定を使用しません。
2. このメソッドは、DB2 for z/OS 上の IBM Data Server Driver for JDBC and SQLJ Type 2 接続ではサポートされません。

表 46. *DataTruncation* メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
<code>java.lang.Throwable</code> から継承されたメソッド	はい	はい	はい
<code>java.sql.SQLException</code> から継承されたメソッド	はい	はい	はい
<code>java.sql.SQLWarning</code> から継承されたメソッド	はい	はい	はい
<code>getDataSize</code>	はい	はい	はい
<code>getIndex</code>	はい	はい	はい
<code>getParameter</code>	はい	はい	はい
<code>getRead</code>	はい	はい	はい
<code>getTransferSize</code>	はい	はい	はい

表 47. *Driver* メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
<code>acceptsURL</code>	はい	はい	はい
<code>connect</code>	はい	はい	はい
<code>getMajorVersion</code>	はい	はい	はい
<code>getMinorVersion</code>	はい	はい	はい
<code>getPropertyInfo</code>	はい	はい	はい
<code>jdbcCompliant</code>	はい	はい	はい

表 48. *DriverManager* メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバ のサポート	IDS JDBC ドライバ のサポート
deregisterDriver	はい	はい	はい
getConnection	はい	はい	はい
getDriver	はい	はい	はい
getDrivers	はい	はい	はい
getLoginTimeout	はい	はい <sup>1</sup>	はい <sup>1</sup>
getLogStream	はい	はい	はい
getLogWriter	はい	はい	はい
println	はい	はい	はい
registerDriver	はい	はい	はい
setLoginTimeout	はい <sup>2</sup>	はい <sup>1</sup>	はい <sup>1</sup>
setLogStream	はい	はい	はい
setLogWriter	はい	はい	はい

注:

1. DB2 JDBC Type 2 ドライバーはこの設定を使用しません。
2. このメソッドは、DB2 for z/OS 上の IBM Data Server Driver for JDBC and SQLJ Type 2 接続ではサポートされません。

表 49. *ParameterMetaData* メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバ のサポート	IDS JDBC ドライバ のサポート
getParameterClassName	いいえ	いいえ	いいえ
getParameterCount	はい	いいえ	いいえ
getParameterMode	はい	いいえ	いいえ
getParameterType	はい	いいえ	いいえ
getParameterTypeName	はい	いいえ	いいえ
getPrecision	はい	いいえ	いいえ
getScale	はい	いいえ	いいえ
isNullable	はい	いいえ	いいえ
isSigned	はい	いいえ	いいえ

表 50. *PooledConnection* メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバ のサポート	IDS JDBC ドライバ のサポート
addConnectionEventListener	はい	はい	はい
addStatementEventListener <sup>1</sup>	はい	いいえ	いいえ

表 50. PooledConnection メソッドのサポート (続き)

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバ のサポート	IDS JDBC ドライバ のサポート
close	はい	はい	はい
getConnection	はい	はい	はい
removeConnectionEventListener	はい	はい	はい
removeStatementEventListener <sup>1</sup>	はい	いいえ	いいえ

注:

1. これは JDBC 4.0 のメソッドです。

表 51. PreparedStatement メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバ のサポート	IDS JDBC ドライバ のサポート
java.sql.Statement から継承されたメソ ッド	はい	はい	はい
addBatch	はい	はい	はい
clearParameters	はい	はい	はい
execute	はい	はい	はい
executeQuery	はい	はい	はい
executeUpdate	はい	はい	はい
getMetaData	はい	はい	はい
getParameterMetaData	はい	はい	はい
setArray	いいえ	いいえ	いいえ
setAsciiStream	はい <sup>1, 2</sup>	はい	はい
setBigDecimal	はい	はい	はい
setBinaryStream	はい <sup>1, 3</sup>	はい	はい
setBlob	はい <sup>4</sup>	はい	はい
setBoolean	はい	はい	はい
setByte	はい	はい	はい
setBytes	はい	はい	はい
setCharacterStream	はい <sup>1, 5</sup>	はい	はい
setClob	はい <sup>6</sup>	はい	はい
setDate	はい <sup>8</sup>	はい <sup>8</sup>	はい <sup>8</sup>
setDouble	はい	はい	はい
setFloat	はい	はい	はい
setInt	はい	はい	はい
setLong	はい	はい	はい
setNull	はい <sup>9</sup>	はい <sup>9</sup>	はい <sup>9</sup>
setObject	はい	はい	はい

表 51. *PreparedStatement* メソッドのサポート (続き)

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバ のサポート	IDS JDBC ドライバ のサポート
<code>setRef</code>	いいえ	いいえ	いいえ
<code>setRowId<sup>7</sup></code>	はい	いいえ	いいえ
<code>setShort</code>	はい	はい	はい
<code>setString</code>	はい <sup>10</sup>	はい <sup>10</sup>	はい <sup>10</sup>
<code>setTime</code>	はい <sup>8</sup>	はい <sup>8</sup>	はい <sup>8</sup>
<code>setTimestamp</code>	はい <sup>8</sup>	はい <sup>8</sup>	はい <sup>8</sup>
<code>setUnicodeStream</code>	はい	はい	はい
<code>setURL</code>	はい	はい	はい

注:

- `length` パラメーターの値が `-1` の場合、`InputStream` または `Reader` からのすべてのデータが読み取られてデータ・ソースに送信されます。
- このメソッドのサポートされる形式には、以下の JDBC 4.0 形式が含まれます。  
`setAsciiStream(int parameterIndex, InputStream x, long length)`  
`setAsciiStream(int parameterIndex, InputStream x)`
- このメソッドのサポートされる形式には、以下の JDBC 4.0 形式が含まれます。  
`setBinaryStream(int parameterIndex, InputStream x, long length)`  
`setBinaryStream(int parameterIndex, InputStream x)`
- このメソッドのサポートされる形式には、以下の JDBC 4.0 形式が含まれます。  
`setBlob(int parameterIndex, InputStream inputStream, long length)`
- このメソッドのサポートされる形式には、以下の JDBC 4.0 形式が含まれます。  
`setCharacterStream(int parameterIndex, Reader reader, long length)`  
`setCharacterStream(int parameterIndex, Reader reader)`
- このメソッドのサポートされる形式には、以下の JDBC 4.0 形式が含まれます。  
`setClob(int parameterIndex, Reader reader, long length)`
- これは JDBC 4.0 のメソッドです。
- データベース・サーバーは日時値の時間帯調整を行いません。JDBC ドライバーは、`java.util.Calendar` パラメーターを含む `setDate`、`setTime`、または `setTimestamp` メソッドの形式が指定された場合、値をサーバーに送信する前にローカル時間帯の値を調整します。
- 次の形式の `setNull` はサポートされません。  
`setNull(int parameterIndex, int jdbcType, String typeName)`
- 列に `FOR BIT DATA` 属性がある場合またはデータ・タイプが `BLOB` である場合、`setString` はサポートされません。

表 52. *Ref* メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバ のサポート	IDS JDBC ドライバ のサポート
<code>getBaseTypeName</code>	いいえ	いいえ	いいえ

表 53. ResultSet メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバ のサポート	IDS JDBC ドライバ のサポート
absolute	はい	はい	はい
afterLast	はい	はい	はい
beforeFirst	はい	はい	はい
cancelRowUpdates	はい	いいえ	いいえ
clearWarnings	はい	はい	はい
close	はい	はい	はい
deleteRow	はい	いいえ	いいえ
findColumn	はい	はい	はい
first	はい	はい	はい
getArray	いいえ	いいえ	いいえ
getAsciiStream	はい	はい	はい
getBigDecimal	はい	はい	はい
getBinaryStream	はい <sup>1</sup>	はい	はい
getBlob	はい	はい	はい
getBoolean	はい	はい	はい
getByte	はい	はい	はい
getBytes	はい	はい	はい
getCharacterStream	はい	はい	はい
getClob	はい	はい	はい
getConcurrency	はい	はい	はい
getCursorName	はい	はい	はい
getDate	はい <sup>2</sup>	はい <sup>2</sup>	はい <sup>2</sup>
getDouble	はい	はい	はい
getFetchDirection	はい	はい	はい
getFetchSize	はい	はい	はい
getFloat	はい	はい	はい
getInt	はい	はい	はい
getLong	はい	はい	はい
getMetaData	はい	はい	はい
getObject	はい <sup>3</sup>	はい <sup>3</sup>	はい <sup>3</sup>
getRef	いいえ	いいえ	いいえ
getRow	はい	はい	はい
getRowId <sup>9</sup>	はい	いいえ	いいえ
getShort	はい	はい	はい
getStatement	はい	はい	はい
getString	はい	はい	はい
getTime	はい <sup>2</sup>	はい <sup>2</sup>	はい <sup>2</sup>

表 53. ResultSet メソッドのサポート (続き)

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバ のサポート	IDS JDBC ドライバ のサポート
getTimestamp	はい <sup>2</sup>	はい <sup>2</sup>	はい <sup>2</sup>
getType	はい	はい	はい
getUnicodeStream	はい	はい	はい
getURL	はい	はい	はい
getWarnings	はい	はい	はい
insertRow	はい	いいえ	いいえ
isAfterLast	はい	はい	はい
isBeforeFirst	はい	はい	はい
isFirst	はい	はい	はい
isLast	はい	はい	はい
last	はい	はい	はい
moveToCurrentRow	はい	いいえ	いいえ
moveToInsertRow	はい	いいえ	いいえ
next	はい	はい	はい
previous	はい	はい	はい
refreshRow	はい	いいえ	いいえ
relative	はい	はい	はい
rowDeleted	はい	いいえ	いいえ
rowInserted	はい	いいえ	いいえ
rowUpdated	はい	いいえ	いいえ
setFetchDirection	はい	はい	はい
setFetchSize	はい	はい	はい
updateArray	いいえ	いいえ	いいえ
updateAsciiStream	はい <sup>4</sup>	いいえ	いいえ
updateBigDecimal	はい	いいえ	いいえ
updateBinaryStream	はい <sup>5</sup>	いいえ	いいえ
updateBlob	はい <sup>6</sup>	いいえ	いいえ
updateBoolean	はい	いいえ	いいえ
updateByte	はい	いいえ	いいえ
updateBytes	はい	いいえ	いいえ
updateCharacterStream	はい <sup>7</sup>	いいえ	いいえ
updateClob	はい <sup>8</sup>	いいえ	いいえ
updateDate	はい	いいえ	いいえ
updateDouble	はい	いいえ	いいえ
updateFloat	はい	いいえ	いいえ
updateInt	はい	いいえ	いいえ
updateLong	はい	いいえ	いいえ

表 53. *ResultSet* メソッドのサポート (続き)

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバ のサポート	IDS JDBC ドライバ のサポート
<code>updateNull</code>	はい	いいえ	いいえ
<code>updateObject</code>	はい	いいえ	いいえ
<code>updateRef</code>	いいえ	いいえ	いいえ
<code>updateRow</code>	はい	いいえ	いいえ
<code>updateRowId<sup>9</sup></code>	はい	いいえ	いいえ
<code>updateShort</code>	はい	いいえ	いいえ
<code>updateString</code>	はい	いいえ	いいえ
<code>updateTime</code>	はい	いいえ	いいえ
<code>updateTimestamp</code>	はい	いいえ	いいえ
<code>wasNull</code>	はい	はい	はい

注:

- `getBinaryStream` は CLOB 列にサポートされません。
- データベース・サーバーは日時値の時間帯調整を行いません。 JDBC ドライバーは、`java.util.Calendar` パラメーターを含む `getDate`、`getTime`、または `getTimestamp` メソッドの形式が指定された場合、サーバーから値を取り出した後でローカル時間帯の値を調整します。
- 次の形式の `getObject` メソッドはサポートされません。  
`getObject(int parameterIndex, java.util.Map map)`
- このメソッドのサポートされる形式には、以下の JDBC 4.0 形式が含まれます。  
`updateAsciiStream(int columnIndex, InputStream x)`  
`updateAsciiStream(String columnLabel, InputStream x)`  
`updateAsciiStream(int columnIndex, InputStream x, long length)`  
`updateAsciiStream(String columnLabel, InputStream x, long length)`
- このメソッドのサポートされる形式には、以下の JDBC 4.0 形式が含まれます。  
`updateBinaryStream(int columnIndex, InputStream x)`  
`updateBinaryStream(String columnLabel, InputStream x)`  
`updateBinaryStream(int columnIndex, InputStream x, long length)`  
`updateBinaryStream(String columnLabel, InputStream x, long length)`
- このメソッドのサポートされる形式には、以下の JDBC 4.0 形式が含まれます。  
`updateBlob(int columnIndex, InputStream x)`  
`updateBlob(String columnLabel, InputStream x)`  
`updateBlob(int columnIndex, InputStream x, long length)`  
`updateBlob(String columnLabel, InputStream x, long length)`
- このメソッドのサポートされる形式には、以下の JDBC 4.0 形式が含まれます。  
`updateCharacterStream(int columnIndex, Reader reader)`  
`updateCharacterStream(String columnLabel, Reader reader)`  
`updateCharacterStream(int columnIndex, Reader reader, long length)`  
`updateCharacterStream(String columnLabel, Reader reader, long length)`
- このメソッドのサポートされる形式には、以下の JDBC 4.0 形式が含まれます。  
`updateClob(int columnIndex, Reader reader)`  
`updateClob(String columnLabel, Reader reader)`  
`updateClob(int columnIndex, Reader reader, long length)`  
`updateClob(String columnLabel, Reader reader, long length)`
- これは JDBC 4.0 のメソッドです。



表 54. *ResultSetMetaData* メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバ のサポート	IDS JDBC ドライバ のサポート
getCatalogName	はい	はい	はい
getColumnClassName	いいえ	はい	はい
getColumnCount	はい	はい	はい
getColumnDisplaySize	はい	はい	はい
getColumnLabel	はい	はい	はい
getColumnName	はい	はい	はい
getColumnType	はい	はい	はい
getColumnTypeName	はい	はい	はい
getPrecision	はい	はい	はい
getScale	はい	はい	はい
getSchemaName	はい	はい	はい
getTableName	はい <sup>1</sup>	はい	はい
isAutoIncrement	はい	はい	はい
isCaseSensitive	はい	はい	はい
isCurrency	はい	はい	はい
isDefinitelyWritable	はい	はい	はい
isNullable	はい	はい	はい
isReadOnly	はい	はい	はい
isSearchable	はい	はい	はい
isSigned	はい	はい	はい
isWritable	はい	はい	はい

注:

1. IDS データ・ソースの場合、getTableName は値を戻しません。

表 55. *RowId* メソッドのサポート<sup>1</sup>

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート <sup>2</sup>	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバ のサポート	IDS JDBC ドライバ のサポート
equals	はい	いいえ	いいえ
getBytes	はい	いいえ	いいえ
hashCode	いいえ	いいえ	いいえ
toString	はい	いいえ	いいえ

注:

1. これらのメソッドは JDBC 4.0 メソッドです。
2. これらのメソッドは、DB2 for z/OS、DB2 for i5/OS、および IDS データ・ソースへの接続にサポートされます。

表 56. *SQLException* メソッドのサポート<sup>1</sup>

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
java.lang.Exception から継承されたメソッド	はい	いいえ	いいえ
java.lang.Throwable から継承されたメソッド	はい	いいえ	いいえ
java.lang.Object から継承されたメソッド	はい	いいえ	いいえ
getFailedProperties	はい	いいえ	いいえ

**注:**

1. これは JDBC 4.0 クラスです。

表 57. *SQLData* メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
getSQLTypeName	いいえ	いいえ	いいえ
readSQL	いいえ	いいえ	いいえ
writeSQL	いいえ	いいえ	いいえ

表 58. *SQLDataException* メソッドのサポート<sup>1</sup>

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
java.lang.Exception から継承されたメソッド	はい	いいえ	いいえ
java.lang.Throwable から継承されたメソッド	はい	いいえ	いいえ
java.lang.Object から継承されたメソッド	はい	いいえ	いいえ

**注:**

1. これは JDBC 4.0 クラスです。

表 59. *SQLException* メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
java.lang.Exception から継承されたメソッド	はい	はい	はい
getSQLState	はい	はい	はい

表 59. *SQLException* メソッドのサポート (続き)

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
<code>getErrorCode</code>	はい	はい	はい
<code>getNextException</code>	はい	はい	はい
<code>setNextException</code>	はい	はい	はい

表 60. *SQLFeatureNotSupported* メソッドのサポート<sup>1</sup>

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
<code>java.lang.Exception</code> から継承されたメソッド	はい	いいえ	いいえ
<code>java.lang.Throwable</code> から継承されたメソッド	はい	いいえ	いいえ
<code>java.lang.Object</code> から継承されたメソッド	はい	いいえ	いいえ

注:

1. これは JDBC 4.0 クラスです。

表 61. *SQLInput* メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
<code>readArray</code>	いいえ	いいえ	いいえ
<code>readAsciiStream</code>	いいえ	いいえ	いいえ
<code>readBigDecimal</code>	いいえ	いいえ	いいえ
<code>readBinaryStream</code>	いいえ	いいえ	いいえ
<code>readBlob</code>	いいえ	いいえ	いいえ
<code>readBoolean</code>	いいえ	いいえ	いいえ
<code>readByte</code>	いいえ	いいえ	いいえ
<code>readBytes</code>	いいえ	いいえ	いいえ
<code>readCharacterStream</code>	いいえ	いいえ	いいえ
<code>readClob</code>	いいえ	いいえ	いいえ
<code>readDate</code>	いいえ	いいえ	いいえ
<code>readDouble</code>	いいえ	いいえ	いいえ
<code>readFloat</code>	いいえ	いいえ	いいえ
<code>readInt</code>	いいえ	いいえ	いいえ
<code>readLong</code>	いいえ	いいえ	いいえ
<code>readObject</code>	いいえ	いいえ	いいえ

表 61. *SQLInput* メソッドのサポート (続き)

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバ のサポート	IDS JDBC ドライバ のサポート
readRef	いいえ	いいえ	いいえ
readShort	いいえ	いいえ	いいえ
readString	いいえ	いいえ	いいえ
readTime	いいえ	いいえ	いいえ
readTimestamp	いいえ	いいえ	いいえ
wasNull	いいえ	いいえ	いいえ

表 62. *SQLIntegrityConstraintViolationException* メソッドのサポート<sup>1</sup>

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバ のサポート	IDS JDBC ドライバ のサポート
java.lang.Exception から継承されたメソッド	はい	いいえ	いいえ
java.lang.Throwable から継承されたメソッド	はい	いいえ	いいえ
java.lang.Object から継承されたメソッド	はい	いいえ	いいえ

注:

1. これは JDBC 4.0 クラスです。

表 63. *SQLInvalidAuthorizationSpecException* メソッドのサポート<sup>1</sup>

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバ のサポート	IDS JDBC ドライバ のサポート
java.lang.Exception から継承されたメソッド	はい	いいえ	いいえ
java.lang.Throwable から継承されたメソッド	はい	いいえ	いいえ
java.lang.Object から継承されたメソッド	はい	いいえ	いいえ

注:

1. これは JDBC 4.0 クラスです。

表 64. *SQLNonTransientConnectionException* メソッドのサポート<sup>1</sup>

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
java.lang.Exception から継承されたメソッド	はい	いいえ	いいえ
java.lang.Throwable から継承されたメソッド	はい	いいえ	いいえ
java.lang.Object から継承されたメソッド	はい	いいえ	いいえ

注:

1. これは JDBC 4.0 クラスです。

表 65. *SQLNonTransientException* メソッドのサポート<sup>1</sup>

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
java.lang.Exception から継承されたメソッド	はい	いいえ	いいえ
java.lang.Throwable から継承されたメソッド	はい	いいえ	いいえ
java.lang.Object から継承されたメソッド	はい	いいえ	いいえ

注:

1. これは JDBC 4.0 クラスです。

表 66. *SQLOutput* メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
writeArray	いいえ	いいえ	いいえ
writeAsciiStream	いいえ	いいえ	いいえ
writeBigDecimal	いいえ	いいえ	いいえ
writeBinaryStream	いいえ	いいえ	いいえ
writeBlob	いいえ	いいえ	いいえ
writeBoolean	いいえ	いいえ	いいえ
writeByte	いいえ	いいえ	いいえ
writeBytes	いいえ	いいえ	いいえ
writeCharacterStream	いいえ	いいえ	いいえ
writeClob	いいえ	いいえ	いいえ
writeDate	いいえ	いいえ	いいえ
writeDouble	いいえ	いいえ	いいえ

表 66. *SQLOutput* メソッドのサポート (続き)

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバ のサポート	IDS JDBC ドライバ のサポート
writeFloat	はい	はい	はい
writeInt	はい	はい	はい
writeLong	はい	はい	はい
writeObject	はい	はい	はい
writeRef	はい	はい	はい
writeShort	はい	はい	はい
writeString	はい	はい	はい
writeStruct	はい	はい	はい
writeTime	はい	はい	はい
writeTimestamp	はい	はい	はい

表 67. *SQLRecoverableException* メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバ のサポート	IDS JDBC ドライバ のサポート
java.lang.Exception から継承されたメソッド	はい	はい	はい
java.lang.Throwable から継承されたメソッド	はい	はい	はい
java.lang.Object から継承されたメソッド	はい	はい	はい

**注:**

1. これは JDBC 4.0 クラスです。

表 68. *SQLException* メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバ のサポート	IDS JDBC ドライバ のサポート
java.lang.Exception から継承されたメソッド	はい	はい	はい
java.lang.Throwable から継承されたメソッド	はい	はい	はい
java.lang.Object から継承されたメソッド	はい	はい	はい

**注:**

1. これは JDBC 4.0 クラスです。

表 69. *SQLException* メソッドのサポート<sup>1</sup>

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
java.lang.Exception から継承されたメソッド	はい	いいえ	いいえ
java.lang.Throwable から継承されたメソッド	はい	いいえ	いいえ
java.lang.Object から継承されたメソッド	はい	いいえ	いいえ

**注:**

1. これは JDBC 4.0 クラスです。

表 70. *SQLTransientConnectionException* メソッドのサポート<sup>1</sup>

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
java.lang.Exception から継承されたメソッド	はい	いいえ	いいえ
java.lang.Throwable から継承されたメソッド	はい	いいえ	いいえ
java.lang.Object から継承されたメソッド	はい	いいえ	いいえ

**注:**

1. これは JDBC 4.0 クラスです。

表 71. *SQLTransientException* メソッドのサポート<sup>1</sup>

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
java.lang.Exception から継承されたメソッド	はい	いいえ	いいえ
java.lang.Throwable から継承されたメソッド	はい	いいえ	いいえ
java.lang.Object から継承されたメソッド	はい	いいえ	いいえ

**注:**

1. これは JDBC 4.0 クラスです。



表 72. *SQLTransientRollbackException* メソッドのサポート<sup>1</sup>

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
java.lang.Exception から継承されたメソッド	はい	いいえ	いいえ
java.lang.Throwable から継承されたメソッド	はい	いいえ	いいえ
java.lang.Object から継承されたメソッド	はい	いいえ	いいえ

注:

1. これは JDBC 4.0 クラスです。

表 73. *SQLXML* メソッドのサポート<sup>1</sup>

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
free	はい	いいえ	いいえ
getBinaryStream	はい	いいえ	いいえ
getCharacterStream	はい	いいえ	いいえ
getSource	はい	いいえ	いいえ
getString	はい	いいえ	いいえ
setBinaryStream	はい	いいえ	いいえ
setCharacterStream	はい	いいえ	いいえ
setResult	はい	いいえ	いいえ
setString	はい	いいえ	いいえ

注:

1. これらは JDBC 4.0 メソッドです。

表 74. *Statement* メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
addBatch	はい	はい	はい
cancel	はい <sup>1</sup>	はい <sup>2</sup>	はい
clearBatch	はい	はい	はい
clearWarnings	はい	はい	はい
close	はい	はい	はい
execute	はい	はい <sup>3</sup>	はい <sup>3</sup>
executeBatch	はい	はい	はい
executeQuery	はい	はい	はい

表 74. Statement メソッドのサポート (続き)

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2	
		JDBC Type 2 ドライバ のサポート	IDS JDBC ドライバ のサポート
executeUpdate	はい	はい <sup>3</sup>	はい <sup>3</sup>
getConnection	はい	はい	はい
getFetchDirection	はい	はい	はい
getFetchSize	はい	はい	はい
getGeneratedKeys	はい	いいえ	いいえ
getMaxFieldSize	はい	はい	はい
getMaxRows	はい	はい	はい
getMoreResults	はい	はい <sup>3</sup>	はい <sup>3</sup>
getQueryTimeout	はい <sup>2</sup>	はい	はい
getResultSet	はい	はい	はい
getResultSetConcurrency	はい	はい	はい
getResultSetHoldability	はい	いいえ	いいえ
getResultSetType	はい	はい	はい
getUpdateCount <sup>4</sup>	はい	はい	はい
getWarnings	はい	はい	はい
isClosed <sup>6</sup>	はい	いいえ	いいえ
isPoolable <sup>6</sup>	はい	いいえ	いいえ
setCursorName	はい	はい	はい
setEscapeProcessing	はい	はい	はい
setFetchDirection	はい	はい	はい
setFetchSize	はい	はい	はい
setMaxFieldSize	はい	はい	はい
setMaxRows	はい	はい	はい
setPoolable <sup>6</sup>	はい	いいえ	いいえ
setQueryTimeout	はい <sup>5</sup>	はい	はい

表 74. *Statement* メソッドのサポート (続き)

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
-----------	--	---	--------------------

注:

- IBM Data Server Driver for JDBC and SQLJ の場合、`Statement.cancel()` は以下の環境でのみサポートされます。
  - Linux、UNIX、または Windows クライアントから DB2 Database for Linux, UNIX, and Windows サーバー、バージョン 8 以降へのタイプ 2 またはタイプ 4 接続
  - Linux、UNIX、または Windows クライアントから DB2 for z/OS サーバー、バージョン 9 以降へのタイプ 2 またはタイプ 4 接続
  - z/OS クライアントから DB2 Database for Linux, UNIX, and Windows サーバー、バージョン 8 以降へのタイプ 4 接続
  - z/OS クライアントから DB2 for z/OS サーバー、バージョン 9 以降へのタイプ 4 接続
- Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバの場合、`Statement.cancel()` は以下の環境でのみサポートされます。
  - DB2 Database for Linux, UNIX, and Windows サーバー、バージョン 8 以降への接続
  - DB2 for z/OS サーバー、バージョン 9 以降への接続
- Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバは、このメソッドの JDBC 3.0 形式をサポートしません。
- ストアード・プロシージャ ResultSet ではサポートされていません。
- DB2 for z/OS および DB2 for i5/OS 上の IBM Data Server Driver for JDBC and SQLJ Type 2 接続では、このメソッドは `seconds` の値が 0 の場合にのみサポートされます。
- これは JDBC 4.0 のメソッドです。

表 75. *Struct* メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
<code>getSQLTypeName</code>	いいえ	いいえ	いいえ
<code>getAttributes</code>	いいえ	いいえ	いいえ

表 76. *Wrapper* メソッドのサポート

JDBC メソッド <sup>1</sup>	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
<code>isWrapperFor</code>	はい	いいえ	いいえ
<code>unwrap</code>	はい	いいえ	いいえ

注:

- これらは JDBC 4.0 メソッドです。

表 77. *javax.sql.XAConnection* メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート <sup>1</sup>	Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
<i>javax.sql.PooledConnection</i> から継承されたメソッド	はい	はい	はい
<i>getXAResource</i>	はい	はい	はい

**注:**

- これらのメソッドは、DB2 Database for Linux, UNIX, and Windows サーバーへの IBM Data Server Driver for JDBC and SQLJ Type 2 接続、または IBM Data Server Driver for JDBC and SQLJ Type 4 接続でサポートされません。

表 78. *XADataSource* メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
<i>getLoginTimeout</i>	はい	はい	はい
<i>getLogWriter</i>	はい	はい	はい
<i>getXAConnection</i>	はい	はい	はい
<i>setLoginTimeout</i>	はい	はい	はい
<i>setLogWriter</i>	はい	はい	はい

表 79. *javax.transaction.xa.XAResource* メソッドのサポート

JDBC メソッド	IBM Data Server Driver for JDBC and SQLJ のサポート	Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバのサポート	IDS JDBC ドライバのサポート
<i>commit</i>	はい <sup>1</sup>	はい	はい
<i>end</i>	はい <sup>1</sup>	はい	はい
<i>forget</i>	はい <sup>1</sup>	はい	はい
<i>getTransactionTimeout</i>	はい <sup>2</sup>	はい	はい
<i>isSameRM</i>	はい <sup>1</sup>	はい	はい
<i>prepare</i>	はい <sup>1</sup>	はい	はい
<i>recover</i>	はい <sup>1</sup>	はい	はい
<i>rollback</i>	はい <sup>1</sup>	はい	はい
<i>setTransactionTimeout</i>	はい <sup>2</sup>	はい	はい
<i>start</i>	はい <sup>1</sup>	はい	はい

**注:**

- これらのメソッドは、DB2 Database for Linux, UNIX, and Windows サーバーへの IBM Data Server Driver for JDBC and SQLJ Type 2 接続、または IBM Data Server Driver for JDBC and SQLJ Type 4 接続でサポートされません。
- このメソッドは、DB2 Database for Linux, UNIX, and Windows バージョン 9.1 以降への IBM Data Server Driver for JDBC and SQLJ Type 4 接続でサポートされます。

## SQLJ ステートメント参照情報

SQLJ ステートメントは、トランザクション制御および SQL ステートメント実行に使用されます。

### SQLJ 節

SQLJ プログラム内の SQL ステートメントは SQLJ 節です。

#### 構文



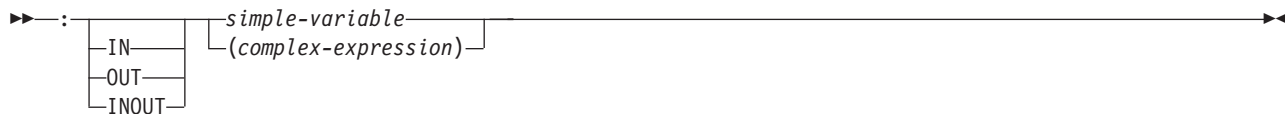
#### 使用上の注意

SQLJ 節内のキーワードは、それらのキーワードが実行可能節内の SQL ステートメントの一部でない限り、大文字と小文字を区別します。

### SQLJ ホスト式

ホスト式とは、SQLJ アプリケーション・プログラムの SQLJ 節によって参照される Java 変数または式です。

#### 構文



#### 説明

: 後に続く変数または式がホスト式であることを示します。コロンは変数または式の直前に付ける必要があります。

#### INIOUTINOUT

ストアド・プロシージャ呼び出しでパラメーターとして使用されるホスト式の場合、パラメーターがストアド・プロシージャにデータを提供するか (IN)、ストアド・プロシージャからデータを取り出すか (OUT)、またはその両方を行うか (INOUT) を識別します。デフォルトは、IN です。

#### simple-variable

Java 非修飾 ID を指定します。

#### complex-expression

単一値を生成する Java 式を指定します。

#### 使用上の注意

- 複合式は括弧で囲む必要があります。

- ANSI/ISO 規則は、ホスト式を静的 SQL ステートメントに配置できる場所で適用されます。

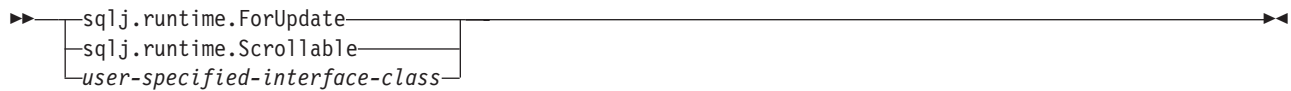
## SQLJ インプリメント節

インプリメント節は、Java インターフェースの 1 つ以上のクラスから派生したものです。

### 構文



#### interface-element:



### 説明

#### interface-element

**ユーザー定義** Java インターフェース、SQLJ インターフェース `sqlj.runtime.ForUpdate`、または SQLJ インターフェース `sqlj.runtime.Scrollable` を指定します。

**位置指定 UPDATE または位置指定 DELETE** 操作用にイテレーターを宣言する際に、`sqlj.runtime.ForUpdate` をインプリメントする必要があります。SQLJ の位置指定 UPDATE または位置指定 DELETE 操作の実行については、『SQLJ アプリケーションでの位置指定 UPDATE および DELETE 操作の実行』を参照してください。

**スクロール可能イテレーター**を宣言する際に、`sqlj.runtime.Scrollable` をインプリメントする必要があります。スクロール可能イテレーターについては、『SQLJ アプリケーションでのスクロール可能イテレーターの使用』を参照してください。

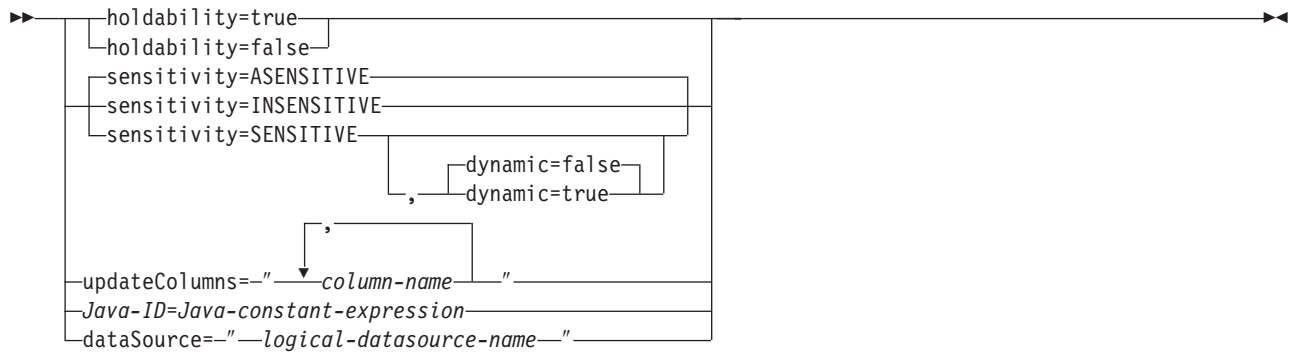
## SQLJ with 節

with 節は、イテレーターまたは接続コンテキストの 1 つ以上の属性のセットを指定します。

### 構文



#### with-element:



## 説明

### holdability

イテレータの場合、COMMIT が実行された後、イテレータが表内でその位置を保持するかどうかを指定します。holdability の値は true または false でなければなりません。

### sensitivity

イテレータの場合、基礎表に対して行われる変更が、イテレータが開かれた後にそのイテレータから可視になるかどうかを指定します。値は、INSENSITIVE、SENSITIVE、または ASENSITIVE でなければなりません。デフォルトは ASENSITIVE です。

### dynamic

sensitivity=SENSITIVE で定義されたイテレータの場合、以下のケースが該当するかどうかを指定します。

- アプリケーションが位置指定 UPDATE および DELETE ステートメントをイテレータで実行する場合、それらの変更がイテレータから可視になる。
- アプリケーションが、そのアプリケーションの内部ではあるがイテレータの外部で INSERT、UPDATE、および DELETE ステートメントを実行する場合、それらの変更がイテレータから可視になる。

dynamic の値は true または false でなければなりません。デフォルトは false です。

DB2 Database for Linux, UNIX, and Windows サーバーは動的両方向スクロール・カーソルをサポートしません。アプリケーションが DB2 for z/OS サーバーのバージョン 9 以降でデータにアクセスする場合にのみ、true を指定します。

### updateColumns

イテレータの場合、イテレータが位置指定 UPDATE ステートメントに使用される際に変更される列を指定します。updateColumns の値は、コンマで区切られた列名を含むリテラル・ストリングでなければなりません。

### column-name

イテレータの場合、イテレータを使用して更新される結果表の列を指定します。

### Java-ID

イテレータまたは接続コンテキストの場合、イテレータまたは接続コンテキ



ストのユーザー定義属性を識別する Java 変数を指定します。  
*Java-constant-expression* の値もユーザー定義です。

#### dataSource

接続コンテキストの場合、アプリケーションの接続先のデータ・ソースを表す、個別に作成された DataSource オブジェクトの論理名を指定します。このオプションは、IBM Data Server Driver for JDBC and SQLJ にのみ使用可能です。

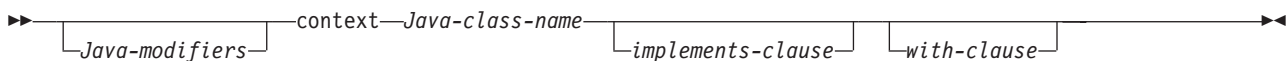
#### 使用上の注意

- with エレメントの左側の値はその with 節内で固有でなければなりません。
- イテレーター宣言節の with エレメントで updateColumns を指定する場合、イテレーター宣言節には、sqlj.runtime.ForUpdate インターフェースを指定する implements 節も含まれていなければなりません。
- SQLJ プログラムをカスタマイズしない場合、JDBC ドライバーは with 節内の holdability の値を無視します。その代わりに、ドライバーは holdability に JDBC ドライバー設定を使用します。

## SQLJ 接続宣言節

接続宣言節は、SQLJ アプリケーション・プログラム内のデータ・ソースへの接続を宣言します。

#### 構文



#### 説明

##### Java-modifiers

Java クラス宣言に有効な修飾子 (static、public、private、または protected など) を指定します。

##### Java-class-name

有効な Java ID を指定します。プログラム準備処理において、SQLJ は名前がこの ID である接続コンテキスト・クラスを生成します。

##### implements-clause

このクラスの説明については、『SQLJ インプリメント節』を参照してください。接続宣言節では、インプリメント節が参照するインターフェース・クラスはユーザー定義インターフェース・クラスでなければなりません。

##### with-clause

このクラスの説明については、『SQLJ with 節』を参照してください。

#### 使用上の注意

- SQLJ は、指定される接続宣言節ごとに接続クラス宣言を生成します。SQLJ データ・ソース接続は、生成された接続クラスのオブジェクトです。
- Java クラス定義を Java プログラム内で配置できる場所であればどこでも接続宣言節を指定できます。

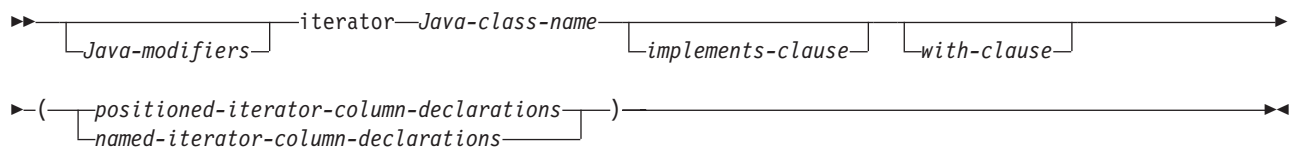
## SQLJ イテレーター宣言節

イテレーター宣言節は、SQLJ アプリケーション・プログラムで位置指定イテレーター・クラスまたは名前指定イテレーター・クラスを宣言します。

イテレーターには照会からの結果表が含まれます。SQLJ は、指定されたイテレーター宣言節ごとにイテレーター・クラスを生成します。イテレーターは、イテレーター・クラスのオブジェクトです。

イテレーター宣言節には、位置指定イテレーターの形式と名前指定イテレーターの形式があります。2 種類のイテレーターは固有であり、異なるインターフェースでインプリメントされている Java タイプとは互換性はありません。

### 構文



#### positioned-iterator-column declarations:



#### named-iterator-column-declarations:



### 説明

#### Java-modifiers

Java クラス宣言に有効な修飾子 (static、public、private、または protected など)。

#### Java-class-name

有効な Java ID。プログラム準備処理中に、SQLJ は名前がこの ID であるイテレーター・クラスを生成します。

#### implements-clause

このクラスの説明については、『SQLJ インプリメント節』を参照してください。位置指定 UPDATE または位置指定 DELETE 操作用のイテレーターを宣言するイテレーター宣言節の場合、インプリメント節はインターフェース `sqlj.runtime.ForUpdate` を指定する必要があります。スクロール可能イテレーターを宣言するイテレーター宣言節の場合、インプリメント節はインターフェース `sqlj.runtime.Scrollable` を指定する必要があります。

#### with-clause

このクラスの説明については、『SQLJ with 節』を参照してください。

### positioned-iterator-column-declarations

Java データ・タイプのリストを指定します。これは、位置指定イテレーター内の列のデータ・タイプです。リスト内のデータ・タイプはコンマで区切る必要があります。位置指定イテレーター宣言内のデータ・タイプの順序は、結果表内の列の順序と同じでなければなりません。シリアライズド・プロファイル・カスタマイズ中のオンライン検査が成功するには、イテレーター内の列のデータ・タイプは結果表内の列のデータ・タイプと互換性がなければなりません。互換性のあるデータ・タイプのリストについては、『Java、JDBC、および SQL のデータ・タイプ』を参照してください。

### named-iterator-column-declarations

Java データ・タイプおよび Java ID のリストを指定します。これらは、名前指定イテレーター内の列のデータ・タイプおよび名前です。データ・タイプと名前の対はコンマで区切る必要があります。イテレーター内の列の名前は、大/小文字の区別は別として、結果表の列の名前と一致しなければなりません。シリアライズド・プロファイル・カスタマイズ中のオンライン検査が成功するには、イテレーター内の列のデータ・タイプは結果表内の列のデータ・タイプと互換性がなければなりません。互換性のあるデータ・タイプのリストについては、『Java、JDBC、および SQL のデータ・タイプ』を参照してください。

## 使用上の注意

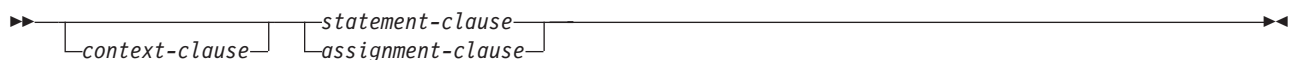
- イテレーター宣言節は、Java クラス宣言を配置できる場所であれば、Java プログラム内のどこにでも配置できます。
- 名前指定イテレーター宣言に Java データ・タイプと Java ID の複数の対が含まれている場合、リスト内のすべての Java ID は固有でなければなりません。2 つの Java ID の大/小文字の区別だけが異なる場合は、それらは固有ではありません。

## SQLJ 実行可能節

実行可能節には SQL ステートメントまたは代入ステートメントが含まれます。代入ステートメントは、SQL 操作の結果を Java 変数に割り当てます。

このトピックでは、実行可能節の一般形式について説明します。

### 構文



## 使用上の注意

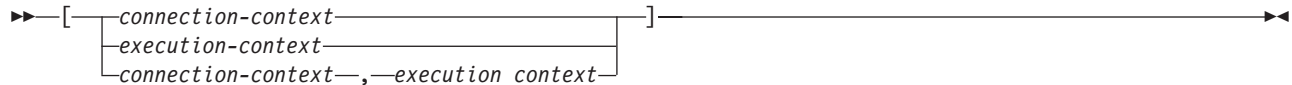
- 実行可能節は、Java ステートメントを配置できる場所であれば、Java プログラム内のどこにでも配置できます。
- SQLJ は、クラス `java.sql.SQLException` を使用して実行可能節からの負の SQL コードを報告します。

SQLJ で実行可能節の実行中にランタイム例外が発生する場合、タイプ OUT または INOUT のホスト式の値が未定義です。

## SQLJ コンテキスト節

コンテキスト節は、接続コンテキスト、実行コンテキスト、またはその両方を指定します。接続コンテキストを使用して、データ・ソースに接続します。実行コンテキストを使用して、SQL ステートメントの実行をモニターおよび変更します。

### 構文



### 説明

#### connection-context

SQLJ プログラムで以前に宣言されている有効な Java ID を指定します。その ID は、SQLJ が接続宣言節のために生成する接続コンテキスト・クラスのインスタンスとして宣言する必要があります。

#### execution-context

SQLJ プログラムで以前に宣言されている有効な Java ID を指定します。その ID は、クラス `sqlj.runtime.ExecutionContext` のインスタンスとして宣言する必要があります。

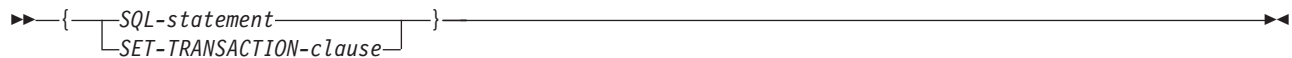
### 使用上の注意

- 実行可能節で接続コンテキストを指定しない場合、SQLJ はデフォルト接続コンテキストを使用します。
- 実行コンテキストを指定しない場合、SQLJ はステートメントの接続コンテキストから実行コンテキストを取得します。

## SQLJ ステートメント節

ステートメント節には SQL ステートメントまたは SET TRANSACTION 節が含まれます。

### 構文



### 説明

#### SQL-statement

318 ページの表 80 の DB2 Database for Linux, UNIX, and Windows SQL ステートメントをステートメント節に組み込むことができます。

#### SET-TRANSACTION-clause

プログラム内の SQL ステートメントの分離レベルおよび接続用のアクセス・モードを設定します。SET TRANSACTION 節は SET TRANSACTION ステートメントに相当します。これは、1992 年の ANSI/ISO SQL 標準で記述されており、SQL の一部のインプリメンテーションでサポートされます。詳しくは、『SQLJ SET-TRANSACTION 節』を参照してください。

表 80. SQLJ ステートメント節内の有効な SQL ステートメント

ALTER DATABASE  
ALTER FUNCTION  
ALTER INDEX  
ALTER PROCEDURE  
ALTER STOGROUP  
ALTER TABLE  
ALTER TABLESPACE  
CALL  
COMMENT ON  
COMMIT  
CREATE ALIAS  
CREATE DATABASE  
CREATE DISTINCT TYPE  
CREATE FUNCTION  
CREATE GLOBAL TEMPORARY TABLE  
CREATE INDEX  
CREATE PROCEDURE  
CREATE STOGROUP  
CREATE SYNONYM  
CREATE TABLE  
CREATE TABLESPACE  
CREATE TRIGGER  
CREATE VIEW  
DECLARE GLOBAL TEMPORARY TABLE  
DELETE  
DROP ALIAS  
DROP DATABASE  
DROP DISTINCT TYPE  
DROP FUNCTION  
DROP INDEX  
DROP PACKAGE  
DROP PROCEDURE  
DROP STOGROUP  
DROP SYNONYM  
DROP TABLE  
DROP TABLESPACE  
DROP TRIGGER  
DROP VIEW  
FETCH  
GRANT  
INSERT  
LOCK TABLE  
MERGE  
REVOKE  
ROLLBACK  
SAVEPOINT  
SELECT INTO  
SET CURRENT DEFAULT TRANSFORM GROUP  
SET CURRENT DEGREE  
SET CURRENT EXPLAIN MODE  
SET CURRENT EXPLAIN SNAPSHOT  
SET CURRENT ISOLATION  
SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION  
SET CURRENT OPTIMIZATION HINT  
SET CURRENT PACKAGESET (USER はサポートされない)

表 80. SQLJ ステートメント節内の有効な SQL ステートメント (続き)

SET CURRENT PRECISION  
SET CURRENT QUERY OPTIMIZATION  
SET CURRENT REFRESH AGE  
SET CURRENT SCHEMA  
SET PATH  
UPDATE

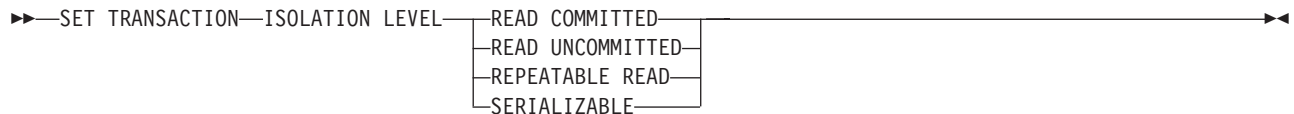
### 使用上の注意

- SQLJ は位置指定 DELETE および UPDATE 操作と検索の DELETE および UPDATE 操作の両方をサポートします。
- FETCH ステートメント、位置指定 DELETE ステートメント、または位置指定 UPDATE ステートメントの場合、イテレーターを使用して結果表内の行を参照する必要があります。

## SQLJ SET TRANSACTION 節

SET TRANSACTION 節は、現在の作業単位の分離レベルを設定するものです。

### 構文



### 説明

#### ISOLATION LEVEL

以下の分離レベルの 1 つを指定します。

#### READ COMMITTED

現在の DB2 分離レベルをカーソル固定にすることを指定します。

#### READ UNCOMMITTED

現在の DB2 分離レベルを非コミット読み取りにすることを指定します。

#### REPEATABLE READ

現在の DB2 分離レベルを読み取り固定にすることを指定します。

#### SERIALIZABLE

現在の DB2 分離レベルを反復可能読み取りにすることを指定します。

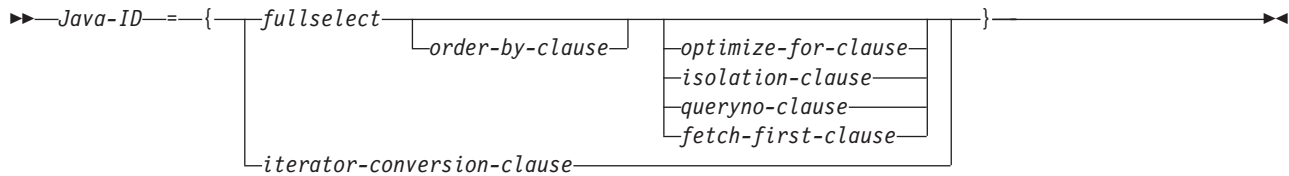
### 使用上の注意

SET TRANSACTION はトランザクションの開始時にのみ実行できます。

## SQLJ 代入節

代入節は、SQL 操作の結果を Java 変数に割り当てます。

## 構文



## 説明

### Java-ID

イテレーター・クラスのインスタンスとして以前に宣言されたイテレーターを識別します。

### fullselect

結果表を生成します。

### iterator-conversion-clause

このクラスの説明については、『SQLJ iterator-conversion-clause』を参照してください。

## 使用上の注意

- *Java-ID* によって識別されるオブジェクトが位置指定イテレーターの場合、結果セット内の列の数がイテレーター内の列の数に一致しなければなりません。さらに、結果セット内の列ごとのデータ・タイプがイテレーター内の対応する列のデータ・タイプと互換性を持っている必要があります。互換性のある Java および SQL データ・タイプについては、『Java、JDBC、および SQL のデータ・タイプ』を参照してください。
- *Java-ID* によって識別されるオブジェクトが名前指定イテレーターの場合、各 accessor メソッドの名前が、大/小文字の区別は別として、結果セット内の列の名前に一致しなければなりません。さらに、accessor メソッドが戻すオブジェクトのデータ・タイプが結果セット内の対応する列のデータ・タイプと互換性を持っている必要があります。
- Java プログラム内の、Java 代入ステートメントを配置できる場所であれば、どこでも代入節を入れることができます。ただし、Java 代入式を配置できる場所には、代入節を入れることはできません。例えば、FOR ステートメントの制御リストでは代入節を指定できません。

## SQLJ イテレーター変換節

イテレーター変換節は、JDBC ResultSet をイテレーターに変換します。

## 構文



## 説明

### host-expression

SQLJ イテレーターに変換される JDBC ResultSet を指定します。



## 使用上の注意

- JDBC `ResultSet` が変換されるイテレーターが位置指定イテレーターの場合、`ResultSet` 内の列の数はイテレーター内の列の数に一致しなければなりません。さらに、`ResultSet` 内の各列のデータ・タイプがイテレーター内の対応する列のデータ・タイプと互換性を持っている必要があります。
- イテレーターが名前指定イテレーターの場合、各 `accessor` メソッドの名前が、大小文字の区別は別として、`ResultSet` 内の列の名前に一致しなければなりません。さらに、`accessor` メソッドが戻すオブジェクトのデータ・タイプが `ResultSet` 内の対応する列のデータ・タイプと互換性を持っている必要があります。
- イテレーター変換節によって生成されたイテレーターが閉じられると、そのイテレーターの生成元の `ResultSet` も閉じられます。

---

## sqlj.runtime パッケージ内のインターフェースおよびクラス

`sqlj.runtime` パッケージは、SQLJ プログラマーにより直接または間接に使用されるランタイム・クラスおよびインターフェースを定義します。

`AsciiStream` などのクラスは、SQLJ プログラマーにより直接使用されます。`ResultSetIterator` などのインターフェースは、生成されたクラス宣言の一部としてインプリメントされます。

### sqlj.runtime インターフェース

以下の表では、`sqlj.runtime` のインターフェースを要約しています。

表 81. `sqlj.runtime` インターフェースのサマリー

インターフェースの名前	目的
<code>ConnectionContext</code>	データ・ソースへの接続中に実行される SQL 操作を管理します。
<code>ForUpdate</code>	位置指定 UPDATE または DELETE ステートメントで使用されるイテレーターによりインプリメントされます。
<code>NamedIterator</code>	名前指定イテレーターとして宣言されるイテレーターによりインプリメントされません。
<code>PositionedIterator</code>	位置指定イテレーターとして宣言されるイテレーターによりインプリメントされません。
<code>ResultSetIterator</code>	照会結果を JDBC <code>ResultSet</code> を使用して処理することを許可するすべてのイテレーターによりインプリメントされます。
<code>Scrollable</code>	スクロール可能イテレーターの操作の一連のメソッドを提供します。

### sqlj.runtime クラス

以下の表では、`sqlj.runtime` のクラスを要約しています。

表 82. `sqlj.runtime` クラスのサマリー

クラス名	目的
<code>AsciiStream</code>	バイトが ASCII として解釈される入力ストリームの処理のためのクラス。
<code>BinaryStream</code>	バイトがバイナリーとして解釈される入力ストリームの処理のためのクラス。
<code>CharacterStream</code>	バイトが文字として解釈される入力ストリームの処理のためのクラス。

表 82. `sqlj.runtime` クラスのサマリー (続き)

クラス名	目的
<code>DefaultRuntime</code>	ほとんどの JVM 環境で SQLJ に期待される実行時動作を満たすために、SQLJ によりインプリメントされます。このクラスは内部使用専用であり、この資料では説明されません。
<code>ExecutionContext</code>	SQL 操作の実行を制御するために、SQLJ 実行コンテキストの宣言時にインプリメントされます。
<code>RuntimeContext</code>	実行時環境で提供されるシステム固有サービスを定義します。このクラスは内部使用専用であり、この資料では説明されません。
<code>SQLException</code>	<code>java.sql.SQLException</code> クラスから派生します。SQL NULL 値がフェッチされ、ホスト ID に Java プリミティブ・タイプで入れられると、 <code>sqlj.runtime.SQLException</code> がスローされます。
<code>StreamWrapper</code>	<code>java.io.InputStream</code> インスタンスをラップします。
<code>UnicodeStream</code>	バイトが Unicode として解釈される入力ストリームの処理のためのクラス。

## sqlj.runtime.ConnectionContext インターフェース

`sqlj.runtime.ConnectionContext` インターフェースは、特定のデータ・ソースとのセッション中に実行される SQL 操作を管理するメソッドのセットを提供します。

SQLJ 接続宣言節を変換すると、SQLJ によって接続コンテキスト・クラスが作成されます。接続コンテキスト・オブジェクトは、動的 SQL 操作を実行できる JDBC Connection オブジェクトを保守します。接続コンテキスト・オブジェクトはデフォルトの `ExecutionContext` オブジェクトも保守します。

### 変数

#### CLOSE\_CONNECTION

形式:

```
public static final boolean CLOSE_CONNECTION=true;
```

`close` メソッドに渡すことができる定数。これは、基礎となる JDBC Connection オブジェクトをクローズする必要があることを示します。

#### KEEP\_CONNECTION

形式:

```
public static final boolean KEEP_CONNECTION=false;
```

`close` メソッドに渡すことができる定数。これは、基礎となる JDBC Connection オブジェクトをクローズしてはならないことを示します。

### メソッド

#### close()

形式:

```
public abstract void close() throws SQLException
```

以下の機能を実行します。

- 指定された接続コンテキスト・オブジェクトによって使用されるすべてのリソースを解放する。

- オープンしている `ConnectedProfile` オブジェクトをクローズする。
- 基礎となる `JDBC Connection` オブジェクトをクローズする。

`close()` は `close(CLOSE_CONNECTION)` と同等です。

### **close(boolean)**

形式:

```
public abstract void close (boolean close-connection)
    throws SQLException
```

以下の機能を実行します。

- 指定された接続コンテキスト・オブジェクトによって使用されるすべてのリソースを解放する。
- オープンしている `ConnectedProfile` オブジェクトをクローズする。
- `close-connection` パラメーターの値に従って、基礎となる `JDBC Connection` オブジェクトをクローズする。

パラメーターは以下のとおりです。

#### *close-connection*

接続コンテキスト・オブジェクトがクローズされるときに基礎となる `JDBC Connection` オブジェクトがクローズされるかどうかを指定します。

#### **CLOSE\_CONNECTION**

基礎となる `JDBC Connection` オブジェクトをクローズします。

#### **KEEP\_CONNECTION**

基礎となる `JDBC Connection` オブジェクトをクローズしません。

### **getConnectedProfile**

形式:

```
public abstract ConnectedProfile getConnectedProfile(Object profileKey)
    throws SQLException
```

このメソッドは、`SQLJ` 変換プログラムによって生成されるコードで使用されません。これはアプリケーション・プログラムによる直接の使用向けのものではありません。

### **getConnection**

形式:

```
public abstract Connection getConnection()
```

指定された接続コンテキスト・オブジェクトの基礎となる `JDBC Connection` オブジェクトを戻します。

### **getExecutionContext**

形式:

```
public abstract ExecutionContext getExecutionContext()
```

指定された接続コンテキスト・オブジェクトに関連したデフォルトの `ExecutionContext` オブジェクトを戻します。

### **isClosed**

形式:

```
public abstract boolean isClosed()
```

指定された接続コンテキスト・オブジェクトがクローズされている場合、`true` を返します。接続コンテキスト・オブジェクトがクローズされていない場合、`false` を返します。

## コンストラクター

以下のコンストラクターは、ステートメント `#sql context Ctx;` の変換の結果である `ConnectionContext` インターフェースの具体的なインプリメンテーションで定義されます。

### **Ctx(String, boolean)**

形式:

```
public Ctx(String url, boolean autocommit)
    throws SQLException
```

パラメーターは以下のとおりです。

*url* JDBC `getConnection` メソッドで指定されたデータ・ソースの表記。

*autocommit*

接続の自動コミットが有効であるかどうかを示します。値 `true` は自動コミットが有効であることを意味します。値 `false` は自動コミットが無効であることを意味します。

### **Ctx(String, String, String, boolean)**

形式:

```
public Ctx(String url, String user, String password,
    boolean autocommit)
    throws SQLException
```

パラメーターは以下のとおりです。

*url* JDBC `getConnection` メソッドで指定されたデータ・ソースの表記。

*user*

データ・ソースへの接続を行うためのユーザー ID。

*password*

データ・ソースへの接続を行うためのユーザー ID のパスワード。

*autocommit*

接続の自動コミットが有効であるかどうかを示します。値 `true` は自動コミットが有効であることを意味します。値 `false` は自動コミットが無効であることを意味します。

### **Ctx(String, Properties, boolean)**

形式:

```
public Ctx(String url, Properties info, boolean autocommit)
    throws SQLException
```

パラメーターは以下のとおりです。

*url* JDBC `getConnection` メソッドで指定されたデータ・ソースの表記。

### *info*

接続のドライバー・プロパティのセットを含むオブジェクト。任意の IBM Data Server Driver for JDBC and SQLJ プロパティを指定できません。

### *autocommit*

接続の自動コミットが有効であるかどうかを示します。値 `true` は自動コミットが有効であることを意味します。値 `false` は自動コミットが無効であることを意味します。

## **Ctx(Connection)**

形式:

```
public Ctx(java.sql.Connection JDBC-connection-object)
    throws SQLException
```

パラメーターは以下のとおりです。

### *JDBC-connection-object*

以前に作成された JDBC Connection オブジェクト。

コンストラクター呼び出しが `SQLException` をスローする場合、JDBC Connection オブジェクトはオープンしたままです。

## **Ctx(ConnectionContext)**

形式:

```
public Ctx(sqlj.runtime.ConnectionContext SQLJ-connection-context-object)
    throws SQLException
```

パラメーターは以下のとおりです。

### *SQLJ-connection-context-object*

以前に作成された SQLJ ConnectionContext オブジェクト。

以下のコンストラクターは、ステートメント `#sql context Ctx with (dataSource = "jdbc/TestDS");` の変換の結果である `ConnectionContext` インターフェースの具体的なインプリメンテーションで定義されます。

## **Ctx()**

形式:

```
public Ctx()
    throws SQLException
```

## **Ctx(String, String)**

形式:

```
public Ctx(String user, String password,
)
    throws SQLException
```

パラメーターは以下のとおりです。

### *user*

データ・ソースへの接続を行うためのユーザー ID。

### *password*

データ・ソースへの接続を行うためのユーザー ID のパスワード。

### **Ctx(Connection)**

形式:

```
public Ctx(java.sql.Connection JDBC-connection-object)
    throws SQLException
```

パラメーターは以下のとおりです。

*JDBC-connection-object*

以前に作成された JDBC Connection オブジェクト。

コンストラクター呼び出しが SQLException をスローする場合、JDBC Connection オブジェクトはオープンしたままです。

### **Ctx(ConnectionContext)**

形式:

```
public Ctx(sqlj.runtime.ConnectionContext SQLJ-connection-context-object)
    throws SQLException
```

パラメーターは以下のとおりです。

*SQLJ-connection-context-object*

以前に作成された SQLJ ConnectionContext オブジェクト。

## **メソッド**

以下の追加メソッドは、ステートメント #sql context Ctx; の変換の結果である ConnectionContext インターフェースの具体的なインプリメンテーションで生成されます。

### **getDefaultContext**

形式:

```
public static Ctx getDefaultContext()
```

Ctx クラスのデフォルトの接続コンテキスト・オブジェクトを戻します。

### **getProfileKey**

形式:

```
public static Object getProfileKey(sqlj.runtime.profile.Loader loader,
    String profileName) throws SQLException
```

このメソッドは、SQLJ 変換プログラムによって生成されるコードで使用されます。これはアプリケーション・プログラムによる直接の使用向けのものではありません。

### **getProfile**

形式:

```
public static sqlj.runtime.profile.Profile getProfile(Object key)
```

このメソッドは、SQLJ 変換プログラムによって生成されるコードで使用されます。これはアプリケーション・プログラムによる直接の使用向けのものではありません。

### **getTypeMap**

形式:

```
public static java.util.Map getTypeMap()
```

ConnectionContext に関連したユーザー定義タイプのマップである java.util.Map をインプリメントするクラスのインスタンスを戻します。関連したタイプのマップがない場合、Java NULL が戻されます。

このメソッドは、実行可能節およびイテレーター宣言節用に SQLJ 変換プログラムによって生成されるコードで使用されますが、JDBC ステートメントで直接使用するために SQLJ アプリケーションで呼び出すこともできます。

#### SetDefaultContext

形式:

```
public static void Ctx setDefaultContext(Ctx default-context)
```

Ctx クラスのデフォルトの接続コンテキスト・オブジェクトを設定します。

**推奨:** このメソッドをマルチスレッド・アプリケーションに使用しないでください。代わりに、明示的コンテキストを使用してください。

## sqlj.runtime.ForUpdate インターフェース

SQLJ は sqlj.runtime.ForUpdate インターフェースを implements sqlj.runtime.ForUpdate を指定したイテレーター宣言節を含む SQLJ プログラムでインプリメントします。

位置指定 UPDATE または DELETE 操作 (UPDATE...WHERE CURRENT OF または DELETE...WHERE CURRENT OF) を実行する SQLJ プログラムは、implements sqlj.runtime.ForUpdate を指定したイテレーター宣言節を含んでいる必要があります。

### メソッド

#### getCursorName

形式:

```
public abstract String getCursorName() throws SQLException
```

このメソッドは、SQLJ 変換プログラムによって生成されるコードで使用されます。これはアプリケーション・プログラムによる直接の使用向けのものではありません。

## sqlj.runtime.NamedIterator インターフェース

sqlj.runtime.NamedIterator インターフェースは、SQLJ アプリケーションが名前指定イテレーターのイテレーター宣言節を実行するときにインプリメントされます。

名前指定イテレーターには結果表の列名が組み込まれ、イテレーター内の列の順序は重要ではありません。

sqlj.runtime.NamedIterator インターフェースのインプリメンテーションには、結果表の各列の accessor メソッドが組み込まれます。accessor メソッドは、結果表の列からのデータを戻します。accessor メソッドの名前は、名前指定イテレーターの対応する列の名前と一致します。



## メソッド (ResultSetIterator インターフェースから継承)

### close

形式:

```
public abstract void close() throws SQLException
```

イテレーターが使用するデータベース・リソースを解放します。

### isClosed

形式:

```
public abstract boolean isClosed() throws SQLException
```

close メソッドが呼び出されている場合に、true の値を返します。close メソッドが呼び出されていない場合には、false を返します。

### next

形式:

```
public abstract boolean next() throws SQLException
```

イテレーターを次の行に進めます。next メソッドのインスタンスを最初に呼び出す前に、イテレーターは結果表の最初の行の前に置かれます。next は、次の行が使用可能な場合は true の値を、すべての行が検索済みの場合は false の値を返します。

## sqlj.runtime.PositionedIterator インターフェース

sqlj.runtime.PositionedIterator インターフェースは、SQLJ アプリケーションが位置指定イテレーターのイテレーター宣言節を実行するときにインプリメントされます。

位置指定イテレーター内の列の順序は、結果表内の列の順序と同じでなければならず、位置指定イテレーターには結果表の列名は組み込まれません。

## メソッド

sqlj.runtime.PositionedIterator はすべての **ResultSetIterator** メソッドを継承し、以下の追加のメソッドを組み込みます。

### endFetch

形式:

```
public abstract boolean endFetch() throws SQLException
```

イテレーターが行上に置かれていない場合は、true の値を返します。イテレーターが行上に置かれている場合は、false の値を返します。

## sqlj.runtime.ResultSetIterator インターフェース

sqlj.runtime.ResultSetIterator インターフェースは、すべてのイテレーター宣言節に対して SQLJ によってインプリメントされます。

非型付きイテレーターは、sqlj.runtime.ResultSetIterator インターフェースのインスタンスの宣言により直接生成できます。一般には、非型付きイテレーターの使用は推奨されていません。

## 変数

### ASENSITIVE

形式:

```
public static final int ASENSITIVE
```

getSensitivity メソッドにより戻すことができる定数。これはイテレーターが ASENSITIVE として定義されていることを示します。

### FETCH\_FORWARD

形式:

```
public static final int FETCH_FORWARD
```

以下のメソッドにより使用できる定数。

- `sqlj.runtime.Scrollable.setFetchDirection` および  
`sqlj.runtime.ExecutionContext.setFetchDirection` により設定される
- `sqlj.runtime.ExecutionContext.getFetchDirection` により戻される

これはイテレーターが結果表内の行を、先頭から最後まで順方向でフェッチすることを示します。

### FETCH\_REVERSE

形式:

```
public static final int FETCH_REVERSE
```

以下のメソッドにより使用できる定数。

- `sqlj.runtime.Scrollable.setFetchDirection` および  
`sqlj.runtime.ExecutionContext.setFetchDirection` により設定される
- `sqlj.runtime.ExecutionContext.getFetchDirection` により戻される

これはイテレーターが結果表内の行を、最後から先頭までの逆方向でフェッチすることを示します。

### FETCH\_UNKNOWN

形式:

```
public static final int FETCH_UNKNOWN
```

以下のメソッドにより使用できる定数。

- `sqlj.runtime.Scrollable.setFetchDirection` および  
`sqlj.runtime.ExecutionContext.setFetchDirection` により設定される
- `sqlj.runtime.ExecutionContext.getFetchDirection` により戻される

これはイテレーターが結果表内の行を、不明の順序でフェッチすることを示します。

### INSENSITIVE

形式:

```
public static final int INSENSITIVE
```

getSensitivity メソッドにより戻すことができる定数。これはイテレーターが INSENSITIVE として定義されていることを示します。

### SENSITIVE

形式:

```
public static final int SENSITIVE
```

`getSensitivity` メソッドにより戻すことができる定数。これはイテレーターが `SENSITIVE` として定義されていることを示します。

## メソッド

### **clearWarnings**

形式:

```
public abstract void clearWarnings() throws SQLException
```

`clearWarnings` の呼び出し後に、`getWarnings` は新しい警告がイテレーターについて報告されるまで `NULL` を戻します。

### **close**

形式:

```
public abstract void close() throws SQLException
```

イテレーターをクローズし、基礎となるデータベース・リソースを解放します。

### **getFetchSize**

形式:

```
synchronized public int getFetchSize() throws SQLException
```

複数の行が必要な場合に `SQLJ` によりフェッチされる行数を戻します。戻り値は、`setFetchSize` メソッドにより設定された値か、または `setFetchSize` により値が設定されていない場合は `0` です。

### **getResultSet**

形式:

```
public abstract ResultSet getResultSet() throws SQLException
```

イテレーターに関連付けられている `JDBC ResultSet` オブジェクトを戻します。

### **getRow**

形式:

```
synchronized public int getRow() throws SQLException
```

現在行の番号を戻します。最初の行は番号 `1`、`2` 番目の行は番号 `2`、というようになります。イテレーターが行上に置かれていない場合は、`0` が戻されます。

### **getSensitivity**

形式:

```
synchronized public int getSensitivity() throws SQLException
```

イテレーターの感度を戻します。感度は、指定された感度値、またはイテレーター宣言節の `with` 節でデフォルト指定された値により決定されます。

### **getWarnings**

形式:

```
public abstract SQLWarning getWarnings() throws SQLException
```

イテレーター上の呼び出しにより報告された最初の警告を戻します。後続のイテレーター警告は、この `SQLWarning` にチェーンニングされます。警告チェーンは、イテレーターが新規行に移動するごとに自動的にクリアされます。

#### **isClosed**

形式:

```
public abstract boolean isClosed() throws SQLException
```

イテレーターがクローズされている場合は、`true` の値を戻します。そうでない場合は `false` を戻します。

#### **next**

形式:

```
public abstract boolean next() throws SQLException
```

イテレーターを次の行に進めます。 `next` を最初に呼び出す前に、イテレーターは結果表の最初の行の前に置かれます。 `next` は、次の行が使用可能な場合は `true` の値を、すべての行が検索済みの場合は `false` の値を戻します。

#### **setFetchSize**

形式:

```
synchronized public void setFetchSize(int number-of-rows) throws SQLException
```

複数の行が必要な場合にフェッチすべき行数に関して、SQLJ にヒントを与えます。

パラメーターは以下のとおりです。

##### *number-of-rows*

特定の実行コンテキストと関連付けられたイテレーターについて SQLJ がフェッチする予想行数。

*number-of-rows* が 0 より小さいか、フェッチ可能な行の最大数より大きい場合、`SQLException` がスローされます。

## **sqlj.runtime.Scrollable インターフェース**

`sqlj.runtime.Scrollable` は、結果表内を移動し、結果表内での位置を確認するためのメソッドを提供します。

`sqlj.runtime.Scrollable` は、スクロール可能イテレーターが宣言されるときにインプリメントされます。

### **メソッド**

#### **absolute(int)**

形式:

```
public abstract boolean absolute (int n) throws SQLException
```

イテレーターを指定された行に移動させます。

$n > 0$  の場合、結果表の  $n$  行にイテレーターが置かれます。  $n < 0$  で、 $m$  が結果表に含まれる行数の場合は、結果表の  $m+n+1$  行にイテレーターが置かれます。

$n$  の絶対値が結果表に含まれる行数よりも大きい場合、 $n$  が正であれば最後の行の後ろ、 $n$  が負であれば最初の行の前にカーソルが置かれます。

`absolute(0)` は `beforeFirst()` と同じです。 `absolute(1)` は `first()` と同じです。 `absolute(-1)` は `last()` と同じです。

イテレーターが行の上にある場合、`true` が戻されます。そうでない場合、`false` が戻されます。

#### **afterLast()**

形式:

```
public abstract void afterLast() throws SQLException
```

イテレーターを結果表の最後の行の後ろに移動します。

#### **beforeFirst()**

形式:

```
public abstract void beforeFirst() throws SQLException
```

イテレーターを結果表の最初の行の前に移動します。

#### **first()**

形式:

```
public abstract boolean first() throws SQLException
```

イテレーターを結果表の最初の行に移動します。

イテレーターが行の上にある場合、`true` が戻されます。そうでない場合、`false` が戻されます。

#### **getFetchDirection()**

形式:

```
public abstract int getFetchDirection() throws SQLException
```

イテレーターのフェッチ方向を戻します。可能な値は以下のとおりです。

##### **sqlj.runtime.ResultSetIterator.FETCH\_FORWARD**

行は最初から最後に向けて、順方向に処理されます。

##### **sqlj.runtime.ResultSetIterator.FETCH\_REVERSE**

行は最後から最初に向けて、逆方向に処理されます。

##### **sqlj.runtime.ResultSetIterator.FETCH\_UNKNOWN**

処理の順序は不明です。

#### **isAfterLast()**

形式:

```
public abstract boolean isAfterLast() throws SQLException
```

イテレーターが結果表の最後の行の後ろにある場合、`true` が戻されます。そうでない場合、`false` が戻されます。

#### **isBeforeFirst()**

形式:

```
public abstract boolean isBeforeFirst() throws SQLException
```

イテレーターが結果表の最初の行の前にある場合、true が戻されます。そうでない場合、false が戻されます。

#### **isFirst()**

形式:

```
public abstract boolean isFirst() throws SQLException
```

イテレーターが結果表の最初の行にある場合、true が戻されます。そうでない場合、false が戻されます。

#### **isLast()**

形式:

```
public abstract boolean isLast() throws SQLException
```

イテレーターが結果表の最後の行にある場合、true が戻されます。そうでない場合、false が戻されます。

#### **last()**

形式:

```
public abstract boolean last() throws SQLException
```

イテレーターを結果表の最後の行に移動します。

イテレーターが行の上にある場合、true が戻されます。そうでない場合、false が戻されます。

#### **previous()**

形式:

```
public abstract boolean previous() throws SQLException
```

イテレーターを結果表の直前の行に移動します。

イテレーターが行の上にある場合、true が戻されます。そうでない場合、false が戻されます。

#### **relative(int)**

形式:

```
public abstract boolean relative(int n) throws SQLException
```

$n > 0$  の場合、現在行から  $n$  行後ろの行にイテレーターが置かれます。  $n < 0$  の場合、現在行から  $n$  行前の行にイテレーターが置かれます。  $n = 0$  の場合、現在行にイテレーターが置かれます。

このメソッドを使用するには、カーソルが結果表の有効な行に置かれている必要があります。カーソルが最初の行の前あるいは最後の行の後ろにある場合、メソッドは `SQLException` をスローします。

$m$  が結果表に含まれる行数、 $x$  が結果表内の現在行の番号とします。  $n > 0$  であり、 $x + n > m$  である場合、イテレーターは最後の行の後ろに置かれます。  $n < 0$  であり、 $x + n < 1$  である場合、イテレーターは最初の行の前に置かれます。

イテレーターが行の上にある場合、true が戻されます。そうでない場合、false が戻されます。

### **setFetchDirection(int)**

形式:

```
public abstract void setFetchDirection (int) throws SQLException
```

このイテレーター・オブジェクトの行が処理される方向を、SQLJ ランタイム環境に示します。可能な値は以下のとおりです。

#### **sqlj.runtime.ResultSetIterator.FETCH\_FORWARD**

行は最初から最後に向けて、順方向に処理されます。

#### **sqlj.runtime.ResultSetIterator.FETCH\_REVERSE**

行は最後から最初に向けて、逆方向に処理されます。

#### **sqlj.runtime.ResultSetIterator.FETCH\_UNKNOWN**

処理の順序は不明です。

## **sqlj.runtime AsciiStream クラス**

sqlj.runtime AsciiStream クラスは、指定された長さを持つ ASCII データの入カストリーム用です。

sqlj.runtime AsciiStream クラスは java.io.InputStream クラスから派生しており、sqlj.runtime.StreamWrapper クラスを拡張するものです。SQLJ は、sqlj.runtime AsciiStream オブジェクト内のバイトを ASCII 文字と解釈します。ASCII 文字を持つ InputStream オブジェクトは、sqlj.runtime AsciiStream オブジェクトとして渡される必要があります。

### **コンストラクター**

#### **AsciiStream(InputStream)**

形式:

```
public AsciiStream(java.io.InputStream input-stream)
```

未指定の長さの ASCII java.io.InputStream オブジェクトを作成します。

パラメーターは以下のとおりです。

*input-stream*

SQLJ が AsciiStream オブジェクトとして解釈する InputStream オブジェクト。

#### **AsciiStream(InputStream, int)**

形式:

```
public AsciiStream(java.io.InputStream input-stream, int length)
```

指定された長さの ASCII java.io.InputStream オブジェクトを作成します。

パラメーターは以下のとおりです。

*input-stream*

SQLJ が AsciiStream オブジェクトとして解釈する InputStream オブジェクト。



*length*

SQLJ が AsciiStream オブジェクトとして解釈する InputStream オブジェクトの長さ。

## sqlj.runtime.BinaryStream クラス

sqlj.runtime.BinaryStream クラスは、指定された長さを持つバイナリー・データの入カストリーム用です。

sqlj.runtime.BinaryStream クラスは java.io.InputStream クラスから派生しており、sqlj.runtime.StreamWrapper クラスを拡張するものです。SQLJ は、sqlj.runtime.BinaryStream オブジェクト内のバイトをバイナリー文字として解釈します。バイナリー文字を持つ InputStream オブジェクトは、sqlj.runtime.BinaryStream オブジェクトとして渡される必要があります。

### コンストラクター

#### BinaryStream(InputStream)

形式:

```
public BinaryStream(java.io.InputStream input-stream)
```

未指定の長さのバイナリー java.io.InputStream オブジェクトを作成します。

パラメーターは以下のとおりです。

*input-stream*

SQLJ が BinaryStream オブジェクトとして解釈する InputStream オブジェクト。

#### BinaryStream(InputStream, int)

形式:

```
public BinaryStream(java.io.InputStream input-stream, int length)
```

指定された長さのバイナリー java.io.InputStream オブジェクトを作成します。

パラメーターは以下のとおりです。

*input-stream*

SQLJ が BinaryStream オブジェクトとして解釈する InputStream オブジェクト。

*length*

SQLJ が BinaryStream オブジェクトとして解釈する InputStream オブジェクトの長さ。

## sqlj.runtime.CharacterStream クラス

sqlj.runtime.CharacterStream クラスは、指定された長さを持つ文字データの入カストリーム用です。

sqlj.runtime.CharacterStream クラスは java.io.Reader クラスから派生しており、java.io.FilterReader クラスを拡張するものです。SQLJ は、sqlj.runtime.CharacterStream オブジェクト内のバイトを Unicode データとして解釈し

ます。Unicode データを持つ Reader オブジェクトは、`sqlj.runtime.CharacterStream` オブジェクトとして渡される必要があります。

## コンストラクター

### **CharacterStream(InputStream)**

形式:

```
public CharacterStream(java.io.Reader input-stream)
```

未指定の長さの文字 `java.io.Reader` オブジェクトを作成します。

パラメーターは以下のとおりです。

*input-stream*

SQLJ が `CharacterStream` オブジェクトとして解釈する `Reader` オブジェクト。

### **CharacterStream(InputStream, int)**

形式:

```
public CharacterStream(java.io.Reader input-stream, int length)
```

指定された長さの文字 `java.io.Reader` オブジェクトを作成します。

パラメーターは以下のとおりです。

*input-stream*

SQLJ が `CharacterStream` オブジェクトとして解釈する `Reader` オブジェクト。

*length*

SQLJ が `CharacterStream` オブジェクトとして解釈する `Reader` オブジェクトの長さ。

## メソッド

### **getReader**

形式:

```
public Reader getReader()
```

`CharacterStream` オブジェクトでラップされる、基礎となる `Reader` オブジェクトを戻します。

### **getLength**

形式:

```
public void getLength()
```

コンストラクターまたは `setLength` への最後の呼び出しで指定された、ラップされた `Reader` オブジェクトの文字単位での長さを戻します。

### **setLength**

形式:

```
public void setLength (int length)
```

オブジェクトが SQL 操作への入力引数として渡されるときに、`Reader` オブジェクトから読み取られる文字数を設定します。

パラメーターは以下のとおりです。

*length*

Reader オブジェクトから読み取られる文字数。

## sqlj.runtime.ExecutionContext クラス

sqlj.runtime.ExecutionContext クラスは、実行コンテキスト用に定義されています。実行コンテキストは、SQL ステートメントの実行を制御するために使用されます。

### 変数

#### ADD\_BATCH\_COUNT

形式:

```
public static final int ADD_BATCH_COUNT
```

getUpdateCount メソッドにより戻すことができる定数。これは前のステートメントが実行されておらず、既存のステートメント・バッチに追加されたことを示します。

#### AUTO\_BATCH

形式:

```
public static final int AUTO_BATCH
```

setBatchLimit メソッドに渡すことができる定数。これは暗黙のバッチ実行を実行する必要があり、SQLJ がバッチ・サイズを決定する必要があることを示します。

#### EXEC\_BATCH\_COUNT

形式:

```
public static final int EXEC_BATCH_COUNT
```

getUpdateCount メソッドから戻すことができる定数。これはステートメント・バッチが実行されたばかりであることを示します。

#### EXCEPTION\_COUNT

形式:

```
public static final int EXCEPTION_COUNT
```

getUpdateCount メソッドから戻すことができる定数。これは前の実行が完了する前に例外がスローされたか、または実行コンテキスト・オブジェクトに対して操作が実行されていないことを示します。

#### NEW\_BATCH\_COUNT

形式:

```
public static final int NEW_BATCH_COUNT
```

getUpdateCount メソッドから戻すことができる定数。これは前のステートメントが実行されておらず、新規のステートメント・バッチに追加されたことを示します。

#### QUERY\_COUNT

形式:

```
public static final int QUERY_COUNT
```

setBatchLimit メソッドに渡すことができる定数。これは前の実行で結果セットが作成されたことを示します。

### UNLIMITED\_BATCH

形式:

```
public static final int UNLIMITED_BATCH
```

getUpdateCount メソッドから戻すことができる定数。これは、バッチ・サイズに関係なく、ステートメントを引き続きステートメント・バッチに追加する必要があることを示します。

コンストラクター:

### ExecutionContext

形式:

```
public ExecutionContext()
```

ExecutionContext インスタンスを作成します。

## メソッド

### cancel

形式:

```
public void cancel() throws SQLException
```

実行コンテキスト・オブジェクトを使用するスレッドにより現在実行中の SQL 操作をキャンセルします。実行コンテキスト・オブジェクト上に保留中のステートメント・バッチがある場合、そのステートメント・バッチはキャンセルされてクリアされます。

cancel メソッドは、ステートメントがキャンセルできない場合は SQLException をスローします。

### execute

形式:

```
public boolean execute ( ) throws SQLException
```

このメソッドは、SQLJ 変換プログラムによって生成されるコードで使用されません。これはアプリケーション・プログラムによる直接の使用向けのものではありません。

### executeBatch

形式:

```
public synchronized int[] executeBatch() throws SQLException
```

保留中のステートメント・バッチを実行し、更新カウンタの配列を戻します。保留中のステートメント・バッチがない場合は、NULL が戻されます。このメソッドが呼び出されると、呼び出しが例外の結果になった場合でも、ステートメント・バッチはクリアされます。

戻される配列内の各要素は、以下のいずれかの値になります。

-2 この値は、SQL ステートメントは正常に実行されたが、更新された行数は判別できなかったことを示します。

-3 この値は SQL ステートメントが失敗したことを示します。

その他の整数

この値は、ステートメントにより更新された行の数です。

`executeBatch` メソッドは、データベース・エラーがステートメント・バッチの実行中に発生した場合、`SQLException` をスローします。

#### **executeQuery**

形式:

```
public ResultSet executeQuery ( ) throws SQLException
```

このメソッドは、SQLJ 変換プログラムによって生成されるコードで使用されません。これはアプリケーション・プログラムによる直接の使用向けのものではありません。

#### **executeUpdate**

形式:

```
public int executeUpdate() throws SQLException
```

このメソッドは、SQLJ 変換プログラムによって生成されるコードで使用されません。これはアプリケーション・プログラムによる直接の使用向けのものではありません。

#### **getBatchLimit**

形式:

```
synchronized public int getBatchLimit()
```

バッチが暗黙的に実行される前にバッチに追加されたステートメントの数を返します。

戻り値は以下のいずれかになります。

#### **UNLIMITED\_BATCH**

この値はバッチ・サイズが無制限であることを示します。

#### **AUTO\_BATCH**

この値はバッチ・サイズが有限であるが不明であることを示します。

その他の整数

現在のバッチ制限。

#### **getBatchUpdateCounts**

形式:

```
public synchronized int[] getBatchUpdateCounts()
```

バッチ内で正常に実行された各ステートメントにより更新された行数を含む配列を返します。配列内の要素の順序は、ステートメントがバッチに挿入された順序に対応します。バッチ内のどのステートメントも正常に完了しなかった場合には `NULL` を返します。

戻される配列内の各要素は、以下のいずれかの値になります。

-2 この値は、SQL ステートメントは正常に実行されたが、更新された行数は判別できなかったことを示します。

-3 この値は SQL ステートメントが失敗したことを示します。

その他の整数

この値は、ステートメントにより更新された行の数です。

### **getFetchDirection**

形式:

```
synchronized public int getFetchDirection() throws SQLException
```

特定の実行コンテキストから生成されたスクロール可能反復子オブジェクトの、現在のフェッチ方向を戻します。フェッチ方向が実行コンテキストに設定されていない場合、`sqlj.runtime.ResultSetIterator.FETCH_FORWARD` が戻されます。

### **getFetchSize**

形式:

```
synchronized public int getFetchSize() throws SQLException
```

複数の行が必要な場合に SQLJ によりフェッチされる行数を戻します。この値は、特定の実行コンテキストから生成された反復子オブジェクトだけに適用されます。戻り値は、`setFetchSize` メソッドにより設定された値か、または `setFetchSize` により値が設定されていない場合は 0 です。

### **getMaxFieldSize**

形式:

```
public synchronized int getMaxFieldSize()
```

特定の実行コンテキストを使用する照会内のストリング (文字、グラフィック、または可変長バイナリー) 列に対して戻される、バイトの最大数を戻します。この制限を超えた場合、SQLJ は残りのバイトを破棄します。値 0 は、バイトの最大数が無制限であることを意味します。

### **getMaxRows**

形式:

```
public synchronized int getMaxRows()
```

特定の実行コンテキストを使用する照会に対して戻される、行の最大数を戻します。この制限を超えた場合、SQLJ は残りの行を破棄します。値 0 は、行の最大数が無制限であることを意味します。

### **getNextResultSet()**

形式:

```
public ResultSet getNextResultSet() throws SQLException
```

ストアード・プロシージャ呼び出しの後に、ストアード・プロシージャから結果セットを戻します。

以下のいずれかの条件が当てはまるときは、NULL 値が戻されます。

- 戻される結果セットがそれ以上存在しない。
- ストアード・プロシージャ呼び出しが結果セットを作成しなかった。
- ストアード・プロシージャ呼び出しが実行コンテキストの下で実行されていない。

getNextResultSet() を呼び出す場合、SQLJ は現在オープンしている結果セットをクローズし、次の結果セットに進みます。

getNextResultSet への呼び出し中にエラーが発生した場合、現在の JDBC ResultSet オブジェクトのリソースは解放され、SQLException がスローされます。 getNextResultSet の後続の呼び出しは NULL を戻します。

### getNextResultSet(int)

形式:

```
public ResultSet getNextResultSet(int current)
```

ストアド・プロシージャ呼び出しの後に、ストアド・プロシージャから結果セットを戻します。

以下のいずれかの条件が当てはまるときは、NULL 値が戻されます。

- 戻される結果セットがそれ以上存在しない。
- ストアド・プロシージャ呼び出しが結果セットを作成しなかった。
- ストアド・プロシージャ呼び出しが実行コンテキストの下で実行されていない。

getNextResultSet への呼び出し中にエラーが発生した場合、現在の JDBC ResultSet オブジェクトのリソースは解放され、SQLException がスローされます。 getNextResultSet の後続の呼び出しは NULL を戻します。

パラメーターは以下のとおりです。

#### *current*

次の結果セットに進む前に、現在オープンしている結果セットを使用して SQLJ が実行する内容を示します。

#### **java.sql.Statement.CLOSE\_CURRENT\_RESULT**

次の ResultSet オブジェクトが戻されるときに、現在の ResultSet オブジェクトをクローズすることを指定します。

#### **java.sql.Statement.KEEP\_CURRENT\_RESULT**

次の ResultSet オブジェクトが戻されるときに、現在の ResultSet オブジェクトをオープンしたままにすることを指定します。

#### **java.sql.Statement.CLOSE\_ALL\_RESULTS**

次の ResultSet オブジェクトが戻されるときに、オープンしているすべての ResultSet オブジェクトをクローズすることを指定します。

### getQueryTimeout

形式:

```
public synchronized int getQueryTimeout()
```

特定の実行コンテキスト・オブジェクトを使用する SQL 操作が実行できる、最大秒数を戻します。SQL 操作がこの制限を超える場合、SQLException がスローされます。戻り値は、setQueryTimeout メソッドにより設定された値か、または setQueryTimeout により値が設定されていない場合は 0 です。0 は、実行時間が無制限であることを示します。

### getUpdateCount

形式:



```
public abstract int getUpdateCount() throws SQLException
```

以下を戻します。

#### **ExecutionContext.ADD\_BATCH\_COUNT**

ステートメントが既存のバッチに追加された場合。

#### **ExecutionContext.NEW\_BATCH\_COUNT**

ステートメントが新規バッチ内の最初のステートメントであった場合。

#### **ExecutionContext.EXCEPTION\_COUNT**

前のステートメントが `SQLException` を生成したか、または前に実行されたステートメントがない場合。

#### **ExecutionContext.EXEC\_BATCH\_COUNT**

ステートメントがバッチの一部であり、バッチが実行された場合。

#### **ExecutionContext.QUERY\_COUNT**

前のステートメントが反復子オブジェクトまたは `JDBC ResultSet` を作成した場合。

#### **その他の整数**

ステートメントが、バッチに追加されたのではなく実行された場合。この値は、ステートメントにより更新された行の数です。

### **getWarnings**

形式:

```
public synchronized SQLWarning getWarnings()
```

特定の実行コンテキストを使用して実行された最後の `SQL` 操作によりレポートされた、最初の警告を戻します。後続の警告は、最初の警告にチェーンングされます。警告が出されていない場合は、`NULL` が戻されます。

`getWarnings` は、正の `SQLCODE` を検索するために使用します。

### **isBatching**

形式:

```
public synchronized boolean isBatching()
```

実行コンテキストに対してバッチが使用可能である場合には `true` を戻します。バッチが使用不可の場合は `false` を戻します。

### **registerStatement**

形式:

```
public RTStatement registerStatement(ConnectionContext connCtx,  
    Object profileKey, int stmtNdx)  
    throws SQLException
```

このメソッドは、`SQLJ` 変換プログラムによって生成されるコードで使用されません。これはアプリケーション・プログラムによる直接の使用向けのものではありません。

### **releaseStatement**

形式:

```
public void releaseStatement() throws SQLException
```

このメソッドは、SQLJ 変換プログラムによって生成されるコードで使用されません。これはアプリケーション・プログラムによる直接の使用向けのものではありません。

### **setBatching**

形式:

```
public synchronized void setBatching(boolean batching)
```

パラメーターは以下のとおりです。

#### *batching*

特定の実行コンテキストで登録されたバッチ可能ステートメントが、ステートメント・バッチに追加できるかどうかを示します。

#### **true**

ステートメントはステートメント・バッチに追加できます。

#### **false**

ステートメントは個別に実行されます。

`setBatching` は、`setBatching` の呼び出し後のプログラム内で実行されるステートメントだけに影響を与えます。これは前のステートメントまたは既存のステートメント・バッチには影響を与えません。

### **setBatchLimit**

形式:

```
public synchronized void setBatchLimit(int batch-size)
```

バッチが暗黙的に実行される前にバッチに追加されるステートメントの最大数を設定します。

パラメーターは以下のとおりです。

#### *batch-size*

以下のいずれかの値になります。

#### **ExecutionContext.UNLIMITED\_BATCH**

SQLJ がバッチ可能であるが非互換であるステートメント、またはバッチ可能ではないステートメントを検出した場合にのみ、暗黙の実行が行われることを示します。この値を設定することは、`setBatchLimit` を呼び出さないことと同じです。

#### **ExecutionContext.AUTO\_BATCH**

バッチ内のステートメントの数が SQLJ により設定された数に達したときに、暗黙の実行が行われることを示します。

#### 正の整数

SQLJ がバッチを暗黙に実行する前にバッチに追加されたステートメントの数。SQLJ がバッチ可能であるが非互換であるステートメント、またはバッチ可能ではないステートメントを検出した場合、この数のステートメントが追加される前にバッチは実行されている可能性があります。

`setBatchLimit` は、`setBatchLimit` の呼び出し後のプログラム内で実行されるステートメントだけに影響を与えます。既存のステートメント・バッチには影響を与えません。

### **setFetchDirection**

形式:

```
public synchronized void setFetchDirection(int direction) throws SQLException
```

特定の実行コンテキストから生成されたスクロール可能反復子オブジェクトの現在のフェッチ方向に関して、SQLJ にヒントを与えます。

パラメーターは以下のとおりです。

*direction*

以下のいずれかの値になります。

**sqlj.runtime.ResultSetIterator.FETCH\_FORWARD**

行は順方向でフェッチされます。これはデフォルトです。

**sqlj.runtime.ResultSetIterator.FETCH\_REVERSE**

行は逆方向でフェッチされます。

**sqlj.runtime.ResultSetIterator.FETCH\_UNKNOWN**

フェッチの順序は不明です。

その他の入力値の結果は `SQLException` になります。

### **setFetchSize**

形式:

```
synchronized public void setFetchSize(int number-of-rows) throws SQLException
```

複数の行が必要な場合にフェッチすべき行数に関して、SQLJ にヒントを与えます。

パラメーターは以下のとおりです。

*number-of-rows*

特定の実行コンテキストと関連付けられたイテレーターについて SQLJ がフェッチする予想行数。

*number-of-rows* が 0 より小さいか、フェッチ可能な行の最大数より大きい場合、`SQLException` がスローされます。

### **setMaxFieldSize**

形式:

```
public void setMaxFieldSize(int max-bytes)
```

特定の実行コンテキストを使用する照会内のストリング (文字、グラフィック、または可変長バイナリー) 列に対して戻される、バイトの最大数を指定します。この制限を超えた場合、SQLJ は残りのバイトを破棄します。

パラメーターは以下のとおりです。

*max-bytes*

SQLJ が BINARY、VARBINARY、CHAR、VARCHAR、GRAPHIC、または

VARGRAPHIC 列から戻す最大バイト数。値 0 は、バイト数が無制限であることを意味します。0 がデフォルトです。

#### **setMaxRows**

形式:

```
public synchronized void setMaxRows(int max-rows)
```

特定の実行コンテキストを使用する照会に対して戻される、行の最大数を指定します。この制限を超えた場合、SQLJ は残りの行を破棄します。

パラメーターは以下のとおりです。

*max-rows*

特定の実行コンテキストを使用する照会に対して SQLJ が戻す、行の最大数。値 0 は、行数が無制限であることを意味します。0 がデフォルトです。

#### **setQueryTimeout**

形式:

```
public synchronized void setQueryTimeout(int timeout-value)
```

特定の実行コンテキスト・オブジェクトを使用する SQL 操作が実行できる、最大秒数を指定します。SQL 操作がこの制限を超える場合、SQLException がスローされます。

パラメーターは以下のとおりです。

*timeout-value*

特定の実行コンテキスト・オブジェクトを使用する SQL 操作が実行できる、最大秒数。0 は、実行時間が無制限であることを示します。0 がデフォルトです。

## **sqlj.runtime.SQLNullException クラス**

sqlj.runtime.SQLNullException クラスは java.sql.SQLException クラスから派生します。

SQL NULL 値がフェッチされ、ホスト ID に Java プリミティブ・タイプで入れられると、sqlj.runtime.SQLNullException がスローされます。SQLNullException のインスタンスの SQLSTATE 値は '22002' です。

## **sqlj.runtime.StreamWrapper クラス**

sqlj.runtime.StreamWrapper クラスは java.io.InputStream インスタンスをラップし、java.io.InputStream クラスを拡張するものです。

sqlj.runtime.AsciiStream、sqlj.runtime.BinaryStream、および sqlj.runtime.UnicodeStream クラスは sqlj.runtime.StreamWrapper を拡張するものです。

sqlj.runtime.StreamWrapper は、

sqlj.runtime.AsciiStream、sqlj.runtime.BinaryStream、および sqlj.runtime.UnicodeStream オブジェクトの長さを指定するメソッドをサポートします。

## コンストラクター

### **StreamWrapper(InputStream)**

形式:

```
protected StreamWrapper(InputStream input-stream)
```

未指定の長さの `sqlj.runtime.StreamWrapper` オブジェクトを作成します。

パラメーターは以下のとおりです。

*input-stream*

`sqlj.runtime.StreamWrapper` オブジェクトがラップする `InputStream` オブジェクト。

### **StreamWrapper(InputStream, int)**

形式:

```
protected StreamWrapper(java.io.InputStream input-stream, int length)
```

指定された長さの `sqlj.runtime.StreamWrapper` オブジェクトを作成します。

パラメーターは以下のとおりです。

*input-stream*

`sqlj.runtime.StreamWrapper` オブジェクトがラップする `InputStream` オブジェクト。

*length*

`InputStream` オブジェクトのバイト単位の長さ。

## メソッド

### **getInputStream**

形式:

```
public InputStream getInputStream()
```

`StreamWrapper` オブジェクトでラップされる、基礎となる `InputStream` オブジェクトを戻します。

### **getLength**

形式:

```
public void getLength()
```

コンストラクターまたは `setLength` への最後の呼び出しで指定された、ラップされた `InputStream` オブジェクトのバイト単位での長さを戻します。

### **setLength**

形式:

```
public void setLength (int length)
```

オブジェクトが `SQL` 操作への入力引数として渡されるときに、ラップされた `InputStream` オブジェクトから読み取られるバイト数を設定します。

パラメーターは以下のとおりです。

*length*

ラップされた `InputStream` オブジェクトから読み取られるバイト数。

## sqlj.runtime.UnicodeStream クラス

sqlj.runtime.UnicodeStream クラスは、指定された長さを持つ Unicode データの入力ストリーム用です。

sqlj.runtime.UnicodeStream クラスは java.io.InputStream クラスから派生しており、sqlj.runtime.StreamWrapper クラスを拡張するものです。SQLJ は、sqlj.runtime.UnicodeStream オブジェクト内のバイトを Unicode 文字として解釈します。Unicode 文字を持つ InputStream オブジェクトは、sqlj.runtime.UnicodeStream オブジェクトとして渡される必要があります。

### コンストラクター

#### UnicodeStream(InputStream)

形式:

```
public UnicodeStream(java.io.InputStream input-stream)
```

未指定の長さの Unicode java.io.InputStream オブジェクトを作成します。

パラメーターは以下のとおりです。

*input-stream*

SQLJ が UnicodeStream オブジェクトとして解釈する InputStream オブジェクト。

#### UnicodeStream(InputStream, int)

形式:

```
public UnicodeStream(java.io.InputStream input-stream, int length)
```

指定された長さの Unicode java.io.InputStream オブジェクトを作成します。

パラメーターは以下のとおりです。

*input-stream*

SQLJ が UnicodeStream オブジェクトとして解釈する InputStream オブジェクト。

*length*

SQLJ が UnicodeStream オブジェクトとして解釈する InputStream オブジェクトの長さ。

---

## JDBC に対する IBM Data Server Driver for JDBC and SQLJ 拡張機能

IBM Data Server Driver for JDBC and SQLJ は、JDBC 仕様によって提供されるサポートに対して拡張機能のセットを提供します。

対応する標準クラスが含まれるクラスで IBM Data Server Driver for JDBC and SQLJ 限定のメソッドを使用するには、関連した標準 JDBC クラスのインスタンスを IBM Data Server Driver for JDBC and SQLJ 限定のクラスのインスタンスにキャストします。以下に例を示します。

```
javax.sql.DataSource ds =  
    new com.ibm.db2.jcc.DB2SimpleDataSource();  
((com.ibm.db2.jcc.DB2BaseDataSource) ds).setServerName("sysmvs1.st1.ibm.com");
```

表 83 は IBM Data Server Driver for JDBC and SQLJ 限定のインターフェースを要約したものです。

表 83. IBM Data Server Driver for JDBC and SQLJ によって提供される IBM Data Server Driver for JDBC and SQLJ 限定のインターフェースのサマリー

インターフェースの名前	適用できるデータ・ソース	目的
DB2Connection	1, 2, 3	java.sql.Connection インターフェースを拡張するものです。
DB2DatabaseMetaData	1, 2, 3	java.sql.DatabaseMetaData インターフェースを拡張するものです。
DB2Diagnosable	1, 2, 3	DB2 SQLException から DB2 診断を取得するメカニズムを提供します。
DB2PreparedStatement	1, 2, 3	com.ibm.db2.jcc.DB2Statement および java.sql.PreparedStatement インターフェースを拡張するものです。
DB2RowID	1, 2	ROWID データ・タイプで使用するための Java オブジェクトを宣言するのに使用されます。
DB2Statement	1, 2, 3	java.sql.Statement インターフェースを拡張するものです。
DB2TraceManagerMXBean	1, 2, 3	リモート・トレース・コントローラー用に MBean インターフェースを提供します。
DB2SystemMonitor	1, 2, 3	接続用にシステム・モニター・データを収集するのに使用されます。
DB2Xml	1, 2	XML 列のデータの更新、および XML 列からのデータの取り出しを実行するために使用されます。

注: インターフェースは以下のデータ・ソースの接続に適用されます。

1. DB2 for z/OS
2. DB2 Database for Linux, UNIX, and Windows
3. IBM Informix Dynamic Server

表 84 は、IBM Data Server Driver for JDBC and SQLJ 限定のクラスを要約したものです。

表 84. IBM Data Server Driver for JDBC and SQLJ によって提供される IBM Data Server Driver for JDBC and SQLJ 限定のクラスのサマリー

クラス名	適用できるデータ・ソース	目的
DB2Administrator (DB2 Database for Linux, UNIX, and Windows のみ)	2 (349 ページ)	DB2Administrator クラスのインスタンスは、DB2CataloguedDatabase オブジェクトの検索に使用されます。
DB2BaseDataSource	1 (349 ページ), 2 (349 ページ), 3 (349 ページ)	javax.sql.DataSource、javax.sql.ConnectionPoolDataSource、および javax.sql.XADataSource のすべての IBM Data Server Driver for JDBC and SQLJ 固有のインプリメンテーションのための抽象データ・ソース親クラスです。



表 84. IBM Data Server Driver for JDBC and SQLJ によって提供される IBM Data Server Driver for JDBC and SQLJ 限定のクラスのサマリー (続き)

クラス名	適用できるデータ・ソース	目的
DB2CataloguedDatabase	2	ローカル DB2 Database for Linux, UNIX, and Windows データベースについての情報を検索するメソッドが含まれています。
DB2ClientRerouteServerList	1, 2	java.io.Serializable および javax.naming.Referenceable インターフェースをインプリメントします。
DB2ConnectionPoolDataSource	1, 2, 3	PooledConnection オブジェクトのファクトリー。
DB2ExceptionFormatter	1, 2, 3	診断情報をストリームに印刷するためのメソッドが含まれます。
DB2JCCPlugin	2	JDBC セキュリティー・プラグインのインプリメンテーションのための抽象クラスです。
DB2PooledConnection	1, 2, 3	以前から存在するトラステッド接続でユーザーを切り替えるためにアプリケーション・サーバーが使用できるメソッドを提供します。
DB2PoolMonitor	1, 2	接続コンセントレーターと Sysplex ワークロード・บาลancingのグローバル・トランスポート・オブジェクト・プールをモニターするためのメソッドを提供します。
DB2SimpleDataSource	1, 2, 3	DataBaseDataSource クラスを拡張します。接続プールまたは分散トランザクションはサポートしていません。
DB2Sqlca	1, 2, 3	DB2 SQLCA をカプセル化したものです。
DB2TraceManager	1, 2, 3	グローバル・ログ書き込みプログラムを制御します。
DB2XADataSource	1, 2, 3	XADataSource オブジェクトのファクトリー。このインターフェースをインプリメントするオブジェクトは、Java Naming and Directory Interface (JNDI) に基づくネーム・サービスで登録されます。

注: このクラスは以下のデータ・ソースの接続に適用されます。

1. DB2 for z/OS
2. DB2 Database for Linux, UNIX, and Windows
3. IBM Informix Dynamic Server

## DB2Administrator クラス

com.ibm.db2.jcc.DB2Administrator クラスのインスタンスは、DB2CataloguedDatabase オブジェクトの検索に使用されます。DB2Administrator は、DB2 Database for Linux, UNIX, and Windows データベースだけに適用されます。

### DB2Administrator のメソッド

#### getInstance

形式:

```
public static DB2Administrator getInstance()
```

DB2Administrator クラスのインスタンスを戻します。

### getCataloguedDatabases

形式:

```
public DB2CataloguedDatabase[] getCataloguedDatabases()  
    throws java.sql.SQLException
```

ローカル・データベース・ディレクトリー内の各ローカル・データベースの DB2CataloguedDatabase オブジェクトを含む配列を検索します。

ローカル DB2 システムが使用可能であり、カタログにデータベースが含まれていない場合、長さゼロの配列が戻されます。使用可能なローカル DB2 システムがない場合は、ヌルが戻されます。ローカル・システムが DB2 Database for Linux, UNIX, and Windows システムではない場合、SQLException がスローされます。

## DB2BaseDataSource クラス

com.ibm.db2.jcc.DB2BaseDataSource クラスは、javax.sql.DataSource、javax.sql.ConnectionPoolDataSource、および javax.sql.XADataSource のすべての IBM Data Server Driver for JDBC and SQLJ 固有のインプリメンテーションのための抽象データ・ソース親クラスです。

DB2BaseDataSource は、java.sql.Wrapper インターフェースをインプリメントします。

### DB2BaseDataSource のプロパティー

以下のプロパティーは、IBM Data Server Driver for JDBC and SQLJ に対してのみ定義されます。

これらの各プロパティーには、プロパティーの値を設定する setXXX メソッド、および値を検索する getXXX メソッドがあります。 setXXX メソッドの形式は以下のとおりです。

```
void setProperty-name(data-type property-value)
```

getXXX メソッドの形式は以下のとおりです。

```
data-type getProperty-name()
```

Property-name は、非修飾プロパティー名です。IBM Informix Dynamic Server (IDS) に固有ではないプロパティーの場合、プロパティー名の先頭文字を大文字にしています。IDS のみによって使用されるプロパティーの場合は、プロパティー名の文字すべてが大文字になっています。

以下の表に、IBM Data Server Driver for JDBC and SQLJ のプロパティーとそのデータ・タイプをリストします。

表 85. DB2BaseDataSource プロパティーとそれらのデータ・タイプ

プロパティー名	適用できるデータ・ソース	
	データ・タイプ	データ・タイプ
com.ibm.db2.jcc.DB2BaseDataSource.accountingInterval	1 (355 ページ)	String

表 85. DB2BaseDataSource プロパティとそれらのデータ・タイプ (続き)

プロパティ名	適用できるデータ・ソース	データ・タイプ
com.ibm.db2.jcc.DB2BaseDataSource.blockingReadConnectionTimeout	1 (355 ページ), 2 (355 ページ), 3 (355 ページ)	int
com.ibm.db2.jcc.DB2BaseDataSource.charOutputSize	1 (355 ページ)	short
com.ibm.db2.jcc.DB2BaseDataSource.clientAccountingInformation (DB2 for z/OS 上の IBM Data Server Driver for JDBC and SQLJ Type 2 接続のみ)	1 (355 ページ), 2 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.clientApplicationInformation (DB2 for z/OS 上の IBM Data Server Driver for JDBC and SQLJ Type 2 接続のみ)	1 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.clientDebugInfo (DB2 for z/OS 上の IBM Data Server Driver for JDBC and SQLJ Type 2 接続のみ)	1 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.clientProgramId (DB2 for z/OS 上の IBM Data Server Driver for JDBC and SQLJ Type 2 接続のみ)	1 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.clientProgramName	1 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.clientRerouteAlternateServerName	1 (355 ページ), 2 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.clientRerouteAlternatePortNumber	1 (355 ページ), 2 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.clientRerouteServerListJNDIContext	1 (355 ページ), 2 (355 ページ)	javax.naming.Context
com.ibm.db2.jcc.DB2BaseDataSource.clientRerouteServerListJNDIName	1 (355 ページ), 2 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.clientUser (DB2 for z/OS 上の IBM Data Server Driver for JDBC and SQLJ Type 2 接続のみ)	1 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.clientWorkstation (DB2 for z/OS 上の IBM Data Server Driver for JDBC and SQLJ Type 2 接続のみ)	1 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.connectNode	2 (355 ページ)	int
com.ibm.db2.jcc.DB2BaseDataSource.currentDegree	1 (355 ページ), 2 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.currentExplainMode	2 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.currentExplainSnapshot	2 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.currentFunctionPath	1 (355 ページ), 2 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.currentLockTimeout	2 (355 ページ), 3 (355 ページ)	int
com.ibm.db2.jcc.DB2BaseDataSource.currentMaintainedTableTypesForOptimization	1 (355 ページ), 2 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.currentPackagePath	1 (355 ページ), 2 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.currentPackageSet	1 (355 ページ), 2 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.currentQueryOptimization	2 (355 ページ)	int
com.ibm.db2.jcc.DB2BaseDataSource.currentRefreshAge	1 (355 ページ), 2 (355 ページ)	long
com.ibm.db2.jcc.DB2BaseDataSource.currentSchema	1 (355 ページ), 2 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.cursorSensitivity	1 (355 ページ), 2 (355 ページ)	int
com.ibm.db2.jcc.DB2BaseDataSource.currentSQLID	1 (355 ページ)	String

表 85. DB2BaseDataSource プロパティとそれらのデータ・タイプ (続き)

プロパティ名	適用できるデータ・ソース	データ・タイプ
com.ibm.db2.jcc.DB2BaseDataSource.databaseName	1 (355 ページ), 2 (355 ページ), 3 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.dateFormat	1 (355 ページ), 2 (355 ページ)	int
com.ibm.db2.jcc.DB2BaseDataSource.decimalRoundingMode	1 (355 ページ), 2 (355 ページ)	int
com.ibm.db2.jcc.DB2BaseDataSource.defaultIsolationLevel	1 (355 ページ), 2 (355 ページ), 3 (355 ページ)	int
com.ibm.db2.jcc.DB2BaseDataSource.deferPrepares	1 (355 ページ), 2 (355 ページ), 3 (355 ページ)	boolean
com.ibm.db2.jcc.DB2BaseDataSource.description	1 (355 ページ), 2 (355 ページ), 3 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.downgradeHoldCursorsUnderXa	1 (355 ページ), 2 (355 ページ), .3 (355 ページ)	boolean
com.ibm.db2.jcc.DB2BaseDataSource.driverType	1 (355 ページ), 2 (355 ページ), 3 (355 ページ)	int
com.ibm.db2.jcc.DB2BaseDataSource.enableConnectionConcentrator	1 (355 ページ)	boolean
com.ibm.db2.jcc.DB2BaseDataSource.enableSysplexWLB	1 (355 ページ)	boolean
com.ibm.db2.jcc.DB2BaseDataSource.fullyMaterializeInputStreams	1 (355 ページ), 2 (355 ページ)	boolean
com.ibm.db2.jcc.DB2BaseDataSource.fullyMaterializeLobData	1 (355 ページ), 2 (355 ページ), 3 (355 ページ)	boolean
com.ibm.db2.jcc.DB2BaseDataSource.gssCredential	1 (355 ページ), 2 (355 ページ)	オブジェクト
com.ibm.db2.jcc.DB2BaseDataSource.jdbcCollection	1 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.keepDynamic	1 (355 ページ), 3 (355 ページ)	int
com.ibm.db2.jcc.DB2BaseDataSource.kerberosServerPrincipal	1 (355 ページ), 2 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.loginTimeout (DB2 for z/OS 上の IBM Data Server Driver for JDBC and SQLJ Type 2 接続ではサポートされない)	1 (355 ページ), 2 (355 ページ), 3 (355 ページ)	int
com.ibm.db2.jcc.DB2BaseDataSource.logWriter	1 (355 ページ), 2 (355 ページ), 3 (355 ページ)	PrintWriter
com.ibm.db2.jcc.DB2BaseDataSource.maxRetriesForClientReroute	1 (355 ページ), 2 (355 ページ)	int
com.ibm.db2.jcc.DB2BaseDataSource.maxTransportObjects	1 (355 ページ)	int
com.ibm.db2.jcc.DB2BaseDataSource.optimizationProfile	2 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.optimizationProfileToFlush	2 (355 ページ)	String

表 85. DB2BaseDataSource プロパティとそれらのデータ・タイプ (続き)

プロパティ名	適用できるデータ・ソース	データ・タイプ
com.ibm.db2.jcc.DB2BaseDataSource.password	1 (355 ページ), 2 (355 ページ), 3 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.pkList ( IBM Data Server Driver for JDBC and SQLJ Type 2 接続)	1 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.planName ( IBM Data Server Driver for JDBC and SQLJ Type 2 接続のみ)	1 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.plugin	2 (355 ページ)	オブジェクト
com.ibm.db2.jcc.DB2BaseDataSource.pluginName	2 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.portNumber	1 (355 ページ), 2 (355 ページ), 3 (355 ページ)	int
com.ibm.db2.jcc.DB2BaseDataSource.progressiveStreaming	1 (355 ページ), 2 (355 ページ)	int
com.ibm.db2.jcc.DB2BaseDataSource.queryDataSize	1 (355 ページ), 2 (355 ページ), 3 (355 ページ)	int
com.ibm.db2.jcc.DB2BaseDataSource.queryCloseImplicit	1 (355 ページ), 3 (355 ページ)	int
com.ibm.db2.jcc.DB2BaseDataSource.readOnly	1 (355 ページ), 2 (355 ページ)	boolean
com.ibm.db2.jcc.DB2BaseDataSource.resultSetHoldability	1 (355 ページ), 2 (355 ページ), 3 (355 ページ)	int
com.ibm.db2.jcc.DB2BaseDataSource.resultSetHoldabilityForCatalogQueries	1 (355 ページ), 2 (355 ページ)	int
com.ibm.db2.jcc.DB2BaseDataSource.retrieveMessagesFromServerOnGetMessage	1 (355 ページ), 3 (355 ページ)	boolean
com.ibm.db2.jcc.DB2BaseDataSource.retryIntervalForClientReroute	1 (355 ページ), 2 (355 ページ)	int
com.ibm.db2.jcc.DB2BaseDataSource.returnAlias	1 (355 ページ), 2 (355 ページ)	short
com.ibm.db2.jcc.DB2BaseDataSource.securityMechanism	1 (355 ページ), 2 (355 ページ), 3 (355 ページ)	int
com.ibm.db2.jcc.DB2BaseDataSource.sendCharInputsUTF8	1 (355 ページ)	int
com.ibm.db2.jcc.DB2BaseDataSource.sendDataAsIs	1 (355 ページ), 2 (355 ページ), 3 (355 ページ)	boolean
com.ibm.db2.jcc.DB2BaseDataSource.serverName	1 (355 ページ), 2 (355 ページ), 3 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.sqlJEnableClassLoaderSpecificProfiles	1 (355 ページ)	boolean
com.ibm.db2.jcc.DB2BaseDataSource.sslConnection	1 (355 ページ), 2 (355 ページ)	boolean
com.ibm.db2.jcc.DB2BaseDataSource.streamBufferSize	1 (355 ページ), 2 (355 ページ)	int
com.ibm.db2.jcc.DB2BaseDataSource.supportsAsynchronousXARollback	1 (355 ページ), 2 (355 ページ)	int

表 85. DB2BaseDataSource プロパティとそれらのデータ・タイプ (続き)

プロパティ名	適用できるデータ・ソース	データ・タイプ
com.ibm.db2.jcc.DB2BaseDataSource.sysSchema	1 (355 ページ), 2 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.timeFormat	1 (355 ページ), 2 (355 ページ)	int
com.ibm.db2.jcc.DB2BaseDataSource.traceDirectory	1 (355 ページ), 2 (355 ページ), 3 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.traceFile	1 (355 ページ), 2 (355 ページ), 3 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.traceFileAppend	1 (355 ページ), 2 (355 ページ), 3 (355 ページ)	boolean
com.ibm.db2.jcc.DB2BaseDataSource.traceLevel	1 (355 ページ), 2 (355 ページ), 3 (355 ページ)	int
com.ibm.db2.jcc.DB2BaseDataSource.useCachedCursor	1 (355 ページ), 2 (355 ページ)	boolean
com.ibm.db2.jcc.DB2BaseDataSource.useJDBC4ColumnNameAndLabelSemantics	1 (355 ページ), 2 (355 ページ)	int
com.ibm.db2.jcc.DB2BaseDataSource.user	1 (355 ページ), 2 (355 ページ), 3 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.useRowsetCursor	1 (355 ページ)	boolean
com.ibm.db2.jcc.DB2BaseDataSource.useTransactionRedirect	2 (355 ページ)	boolean
com.ibm.db2.jcc.DB2BaseDataSource.xaNetworkOptimization	1 (355 ページ), 2 (355 ページ), 3 (355 ページ)	boolean
com.ibm.db2.jcc.DB2BaseDataSource.DBANSIWARN	3 (355 ページ)	boolean
com.ibm.db2.jcc.DB2BaseDataSource.DBDATE	3 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.DBPATH	3 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.DBSPACETIME	3 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.DBTEMP	3 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.DBUPSPACE	3 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.DELIMIDENT	3 (355 ページ)	boolean
com.ibm.db2.jcc.DB2BaseDataSource.IFX_DIRECTIVES	3 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.IFX_EXTDIRECTIVES	3 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.IFX_UPDDESC	3 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.IFX_XASTDCOMPLIANCE_XAEND	3 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.INFORMIXOPCACHE	3 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.INFORMIXSTACKSIZE	3 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.NODEFDAC	3 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.OPTCOMPIND	3 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.OPTOFC	3 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.PDQPRIORITY	3 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.PSORT_DBTEMP	3 (355 ページ)	String
com.ibm.db2.jcc.DB2BaseDataSource.PSORT_NPROCS	3 (355 ページ)	String

表 85. DB2BaseDataSource プロパティとそれらのデータ・タイプ (続き)

プロパティ名	適用できるデータ・ソース	
	データ・ソース	データ・タイプ
com.ibm.db2.jcc.DB2BaseDataSource.STMT_CACHE	3	String

注: プロパティは以下のデータ・ソースの接続に適用されます。

1. DB2 for z/OS
2. DB2 Database for Linux, UNIX, and Windows
3. IBM Informix Dynamic Server

## DB2BaseDataSource のメソッド

DB2BaseDataSource プロパティの getXXX および setXXX メソッドに加えて、以下のメソッドが IBM Data Server Driver for JDBC and SQLJ のためだけに定義されています。

### getReference

形式:

```
public javax.naming.Reference getReference()
    throws javax.naming.NamingException
```

DataSource オブジェクトの Reference を検索します。Reference については、以下の URL にある JNDI 文書から javax.naming.Referenceable についての説明を参照してください。

<http://java.sun.com/products/jndi/docs.html>

## DB2CataloguedDatabase クラス

com.ibm.db2.jcc.DB2CataloguedDatabase クラスには、ローカル DB2 Database for Linux, UNIX, and Windows データベースについての情報を検索するメソッドが含まれています。

DB2CataloguedDatabase メソッドの呼び出しにはデータベース接続は不要です。

### DB2CataloguedDatabase のメソッド

#### getServerName

形式:

```
public String getServerName()
```

データベースが置かれているサーバーの名前を検索します。

#### getPortNumber

形式:

```
public int getPortNumber()
```

DB2 インスタンスと関連付けられているポート番号を検索します。

#### getDatabaseName

形式:

```
public String getDatabaseName()
```

データベース名を検索します。



### **getDatabaseAlias**

形式:

```
public String getDatabaseAlias()
```

データベース別名を検索します。

## **DB2ClientRerouteServerList クラス**

com.ibm.db2.jcc.DB2ClientRerouteServerList クラスは、java.io.Serializable および javax.naming.Referenceable インターフェースをインプリメントします。

### **DB2ClientRerouteServerList のメソッド**

#### **getAlternatePortNumber**

形式:

```
public int[] getAlternatePortNumber()
```

代替サーバーと関連付けられているポート番号を検索します。

#### **getAlternateServerName**

形式:

```
public String[] getAlternateServerName()
```

代替サーバーの名前を含む配列を検索します。これらの値は IP アドレスまたは DNS サーバー名です。

#### **getPrimaryPortNumber**

形式:

```
public int getPrimaryPortNumber()
```

1 次サーバーと関連付けられているポート番号を検索します。

#### **getPrimaryServerName**

形式:

```
public String[] getPrimaryServerName()
```

1 次サーバーの名前を検索します。この値は IP アドレスまたは DNS サーバー名です。

#### **setAlternatePortNumber**

形式:

```
public void setAlternatePortNumber(int[] alternatePortNumberList)
```

代替サーバーと関連付けられているポート番号を設定します。

#### **setAlternateServerName**

形式:

```
public void setAlternateServerName(String[] alternateServer)
```

サーバーの代替サーバー名を設定します。これらの値は IP アドレスまたは DNS サーバー名です。

#### **setPrimaryPortNumber**

形式:

```
public void setPrimaryPortNumber(int primaryPortNumber)
```

1 次サーバーと関連付けられているポート番号を設定します。

### **setPrimaryServerName**

形式:

```
public void setPrimaryServerName(String primaryServer)
```

サーバーの 1 次サーバー名を設定します。この値は IP アドレスまたは DNS サーバー名です。

## **DB2Connection インターフェース**

com.ibm.db2.jcc.DB2Connection インターフェースは java.sql.Connection を拡張するものです。

DB2Connection は java.sql.Wrapper インターフェースをインプリメントします。

### **DB2Connection のメソッド**

以下のメソッドは、IBM Data Server Driver for JDBC and SQLJ に対してのみ定義されます。

#### **alternateWasUsedOnConnect**

形式:

```
public boolean alternateWasUsedOnConnect()  
    throws java.sql.SQLException
```

接続を取得するためにドライバーが代替サーバー情報を使用した場合は、true を戻します。代替サーバー情報は、データベース・サーバーが 1 次サーバーと代替サーバーの変更時に更新する、DB2BaseDataSource 上の一時 clientRerouteServerList 情報から入手可能です。

#### **changeDB2Password**

形式:

```
public abstract void changeDB2Password(String oldPassword,  
    String newPassword)  
    throws java.sql.SQLException
```

Connection オブジェクトのユーザー用の、データ・ソースへのアクセス用のパスワードを変更します。

パラメーターの説明:

##### **oldPassword**

Connection のオリジナル・パスワード。

##### **newPassword**

Connection の新規パスワード。

#### **createArrayOf**

形式:

```
Array createArrayOf(String typeName,  
    Object[] elements)  
    throws SQLException;
```

java.sql.Array オブジェクトを作成します。

パラメーターの説明:

#### **typeName**

配列の要素がマップする SQL データ・タイプ。typeName は組み込みデータ・タイプまたは特殊タイプにすることができます。

#### **elements**

Array オブジェクトを取り込む要素。

### **deregisterDB2XmlObject**

形式:

```
public void deregisterDB2XmlObject(String sqlIdSchema,  
    String sqlIdName)  
    throws SQLException
```

データ・ソースから、事前に登録された XML スキーマを削除します。

deregisterDB2XmlObject は、SYSPROC.XSR\_REMOVE ストアド・プロシージャを呼び出して、XML スキーマを削除します。

パラメーターの説明:

#### **sqlIdSchema**

XML スキーマの SQL スキーマ名。sqlIdSchema は最大長が 128 バイトの文字列値です。sqlIdSchema の値は、SQL スキーマ名の命名規則に従う必要があります。名前の先頭を文字列 'SYS' にすることはできません。sqlIdSchema の値が NULL の場合、データベース・システムは CURRENT SCHEMA 特殊レジスター内の値を使用します。

#### **sqlIdName**

XML スキーマの SQL 名。sqlIdName は最大長が 128 バイトの文字列値です。sqlIdName の値は SQL ID の規則に従わなければなりません。sqlIdSchema の値が NULL の場合、sqlIdName の値は NULL にすることができます。この場合、データベース・システムは sqlIdName の値を生成します。

この方法は、IBM Informix Dynamic Server データ・ソースへの接続には適用されません。

### **getDB2ClientProgramId**

形式:

```
public String getDB2ClientProgramId()  
    throws java.sql.SQLException
```

クライアントのユーザー定義プログラム ID を返します。プログラム ID は、データ・ソースでアプリケーションを識別するために使用できます。

この方法は、IBM Informix Dynamic Server データ・ソースへの接続には適用されません。

### **getDB2ClientAccountingInformation**

形式:

```
public String getDB2ClientAccountingInformation()  
    throws SQLException
```

現在のクライアントのアカウント情報に戻します。

**重要:** `getDB2ClientAccountingInformation` は、IBM Data Server Driver for JDBC and SQLJ の JDBC 4.0 インプリメンテーションでは推奨されていません。代わりに、`java.sql.Connection.getClientInfo` を使用してください。

この方法は、IBM Informix Dynamic Server データ・ソースへの接続には適用されません。

#### **getDB2ClientApplicationInformation**

形式:

```
public String getDB2ClientApplicationInformation()  
    throws java.sql.SQLException
```

現在のクライアントのアプリケーション情報に戻します。

**重要:** `getDB2ClientApplicationInformation` は、IBM Data Server Driver for JDBC and SQLJ の JDBC 4.0 インプリメンテーションでは推奨されていません。代わりに、`java.sql.Connection.getClientInfo` を使用してください。

この方法は、IBM Informix Dynamic Server データ・ソースへの接続には適用されません。

#### **getDB2ClientUser**

形式:

```
public String getDB2ClientUser()  
    throws java.sql.SQLException
```

接続のための現在のクライアントのユーザー名に戻します。この名前は、JDBC 接続のユーザー値ではありません。

**重要:** `getDB2ClientUser` は、IBM Data Server Driver for JDBC and SQLJ の JDBC 4.0 インプリメンテーションでは推奨されていません。代わりに、`java.sql.Connection.getClientInfo` を使用してください。

この方法は、IBM Informix Dynamic Server データ・ソースへの接続には適用されません。

#### **getDB2ClientWorkstation**

形式:

```
public String getDB2ClientWorkstation()  
    throws java.sql.SQLException
```

現在のクライアントの現在のクライアント・ワークステーション名に戻します。

**重要:** `getDB2ClientWorkstation` は、IBM Data Server Driver for JDBC and SQLJ の JDBC 4.0 インプリメンテーションでは推奨されていません。代わりに、`java.sql.Connection.getClientInfo` を使用してください。

この方法は、IBM Informix Dynamic Server データ・ソースへの接続には適用されません。

#### **getDB2Correlator**

形式:

```
String getDB2Correlator()  
    throws java.sql.SQLException
```

DRDA が ACCRDB コマンドで送る、`crrtkn` (相関トークン) インスタンス変数の値を返します。相関トークンは、サーバーへの論理接続を一意的に識別します。

#### **getDB2CurrentPackagePath**

形式:

```
public String getDB2CurrentPackagePath()  
    throws java.sql.SQLException
```

JDBC および SQLJ パッケージが検索される DB2 パッケージ・コレクションのリストを返します。

`getDB2CurrentPackagePath` メソッドは、DB2 データベース・システムへの接続にのみ適用されます。

#### **getDB2CurrentPackageSet**

形式:

```
public String getDB2CurrentPackageSet()  
    throws java.sql.SQLException
```

接続のためのコレクション ID を返します。

`getDB2CurrentPackageSet` メソッドは、DB2 データベース・システムへの接続にのみ適用されます。

#### **getDB2SecurityMechanism**

形式:

```
public int getDB2SecurityMechanism()  
    throws java.sql.SQLException
```

接続に有効なセキュリティー・メカニズムを返します。

- 3 平文パスワード・セキュリティー
- 4 ユーザー ID のみのセキュリティー
- 7 パスワード暗号化セキュリティー
- 9 ユーザー ID とパスワードの暗号化セキュリティー
- 11 Kerberos セキュリティー
- 12 ユーザー ID とデータの暗号化セキュリティー
- 13 ユーザー ID、パスワード、およびデータの暗号化セキュリティー
- 15 プラグイン・セキュリティー
- 16 ユーザー ID のみの暗号化セキュリティー

#### **getDB2SystemMonitor**

形式:

```
public abstract DB2SystemMonitor getDB2SystemMonitor()  
    throws java.sql.SQLException
```

接続のシステム・モニター・オブジェクトを戻します。 IBM Data Server Driver for JDBC and SQLJ 接続ごとに 1 つのシステム・モニターを使用できます。

#### **getJccLogWriter**

形式:

```
public PrintWriter getJccLogWriter()  
    throws java.sql.SQLException
```

IBM Data Server Driver for JDBC and SQLJ トレースの現在のトレースの宛先を戻します。

#### **installDB2JavaStoredProcedure**

形式:

```
public void DB2Connection.installDB2JavaStoredProcedure(  
    java.io.InputStream jarFile,  
    int jarFileLength,  
    String jarId)  
    throws java.sql.SQLException
```

sqlj.install\_jar ストアード・プロシージャを DB2 Database for Linux, UNIX, and Windows サーバー上で呼び出し、そのサーバー用のカタログ内の JAR ファイルの新規定義を作成します。

パラメーターの説明:

##### **jarFile**

サーバーに定義される JAR ファイルの内容。

##### **jarFileLength**

サーバーに定義される JAR ファイルの長さ。

##### **jarId**

データベース内の JAR の名前。 *schema.JAR-id* または *JAR-id* の形式になります。これは、SQL ステートメントで JAR を参照する場合に使用する名前です。 *schema* を省略した場合、データベース・システムは CURRENT SCHEMA 特殊レジスター内にある SQL 許可 ID を使用します。JAR の所有者は、CURRENT SQLID 特殊レジスター内の許可 ID です。

この方法は、IBM Informix Dynamic Server データ・ソースへの接続には適用されません。

#### **isDB2Alive**

形式:

```
public boolean DB2Connection.isDB2Alive()  
    throws java.sql.SQLException
```

データ・ソースへの接続のソケットが依然としてアクティブである場合は、true を戻します。

**重要:** isDB2Alive は、IBM Data Server Driver for JDBC and SQLJ の JDBC 4.0 インプリメンテーションでは推奨されていません。代わりに、Connection.isValid を使用してください。

#### **isDB2GatewayConnection**

形式:

```
public boolean DB2Connection.isDB2GatewayConnection()
    throws java.sql.SQLException
```

データ・ソースへの接続が中間 DB2 Connect ゲートウェイを経由する場合は、true を返します。そうでない場合は false を返します。

### prepareDB2OptimisticLockingQuery

形式:

```
public java.sql.PreparedStatement
    DB2Connection.prepareDB2OptimisticLockingQuery(String sql,
        int returnOptimisticLockingColumns)
    throws SQLException
```

オプティミスティック・ロック情報を要求できる PreparedStatement オブジェクトを作成します。

パラメーターの説明:

**sql** 準備される SQL ステートメント。

### returnOptimisticLockingColumns

オプティミスティック・ロック列を戻すかどうかを指定します。可能な値は以下のとおりです。

表 86.

値	説明
DB2Statement.RETURN_OPTLOCK_COLUMN_NONE (0)	オプティミスティック・ロック列を戻しません。
DB2Statement.RETURN_OPTLOCK_COLUMN_ALWAYS (1)	行の変更列が単一の行を一意的に表さない場合でも、行の変更列を結果セットに追加します。この設定は、データベース準備属性 WITH ROW CHANGE COLUMNS POSSIBLY DISTINCT に相当します。
DB2Statement.RETURN_OPTLOCK_COLUMN_NO_FALSE_NEGATIVES (2)	行の変更列が単一の行を一意的に表す場合のみ、その行の変更列を結果セットに追加します。この設定は、データベース準備属性 WITH ROW CHANGE COLUMNS ALWAYS DISTINCT に相当します。

この方法は、IBM Informix Dynamic Server データ・ソースへの接続には適用されません。

### reconfigureDB2Connection

形式:

```
public void reconfigureDB2Connection(java.util.Properties properties)
    throws SQLException
```

接続を新規設定で再構成します。接続は、再構成前に接続プールに戻す必要はありません。このメソッドは、トランザクションの進行中に呼び出して、トラステッドまたは非トラステッド接続に使用することができます。

パラメーターの説明:

### properties

接続の新規プロパティ。これらのプロパティは、DB2Connection インスタンスですでに定義されているプロパティをオーバーライドします。

### registerDB2XmlSchema

形式:



```

public void registerDB2XmlSchema(String[] sqlIdSchema,
    String[] sqlIdName,
    String[] xmlSchemaLocations,
    InputStream[] xmlSchemaDocuments,
    int[] xmlSchemaDocumentsLengths,
    InputStream[] xmlSchemaDocumentsProperties,
    int[] xmlSchemaDocumentsPropertiesLengths,
    InputStream xmlSchemaProperties,
    int xmlSchemaPropertiesLength,
    boolean isUsedForShredding)
    throws SQLException
public void registerDB2XmlSchema(String[] sqlIdSchema,
    String[] sqlIdName,
    String[] xmlSchemaLocations,
    String[] xmlSchemaDocuments,
    String[] xmlSchemaDocumentsProperties,
    String xmlSchemaProperties,
    boolean isUsedForShredding)
    throws SQLException

```

1 つ以上の XML スキーマ文書を、データベースでの XML スキーマの登録用に提供します。registerDB2XmlSchema は、SYSPROC.XSR\_REGISTER、SYSPROC.XSR\_ADDSCHEMADOC、および SYSPROC.XSR\_COMPLETE ストアード・プロシージャを呼び出し、XML スキーマを 1 つ以上の XML スキーマ文書で登録します。複数の XML スキーマ文書が registerDB2XmlSchema への 1 呼び出しで処理される場合、これらの文書は単一トランザクションの一部として処理されます。

registerDB2XmlSchema の最初のフォームは、入力ストリームから読み取られる XML スキーマ文書です。registerDB2XmlSchema の 2 番目のフォームは、ストリングから読み取られる XML スキーマ文書です。

パラメーターの説明:

#### sqlIdSchema

XML スキーマの SQL スキーマ名。sqlIdSchema 配列の最初のエレメントだけが使用されます。sqlIdSchema は最大長が 128 バイトのストリング値です。sqlIdSchema の値は、SQL スキーマ名の命名規則に従う必要があります。名前の先頭をストリング 'SYS' にすることはできません。sqlIdSchema の値が NULL の場合、データベース・システムは CURRENT SCHEMA 特殊レジスター内の値を使用します。

#### sqlIdName

XML スキーマの SQL 名。sqlIdName 配列の最初のエレメントだけが使用されます。sqlIdName は最大長が 128 バイトのストリング値です。sqlIdName の値は SQL ID の規則に従わなければなりません。sqlIdSchema の値が NULL の場合、sqlIdName の値は NULL にすることができます。この場合、データベース・システムは sqlIdName の値を生成します。

#### xmlSchemaLocations

登録されるスキーマの 1 次 XML スキーマ文書の、XML スキーマ・ロケーション。XML スキーマ・ロケーションの値は、通常は URI 形式です。各 xmlSchemaLocations 値は、最大長が 1000 バイトのストリング値です。この値は、この文書を参照する XML スキーマ文書内で指定された情報と突き合わせるためだけに使用されます。データベース・システムは、フォーマットの検査は実行せず、URI の解決を試行することはありません。

### **xmlSchemaDocuments**

1 次 XML スキーマ文書の内容。各 `xmlSchemaDocuments` 値は、最大長が 30MB の String または `InputStream` 値です。値は NULL であってはなりません。

### **xmlSchemaDocumentsLengths**

`registerDB2XmlSchema` の最初のフォームが使用される場合、`xmlSchemaDocuments` パラメーター内の XML スキーマ文書の長さ。各 `xmlSchemaDocumentsLengths` 値は `int` 値です。

### **xmlSchemaDocumentsProperties**

1 次 XML スキーマ文書のプロパティーが含まれます。これは、外部 XML スキーマ・バージョン管理システムで使用されるプロパティーなどです。データベース・システムは、これらの値の内容の検証は行いません。これらの値は検索のために XSR 表に保管され、他のツールまたは XML スキーマ・リポジトリのインプリメンテーションで使用されます。各 `xmlSchemaDocumentsProperties` 値は、最大長が 5MB の String または `InputStream` 値です。渡されるプロパティーがない場合、値は NULL です。

### **xmlSchemaDocumentsPropertiesLengths**

`registerDB2XmlSchema` の最初のフォームが使用される場合、`xmlSchemaDocumentsProperties` パラメーター内の XML スキーマ・プロパティーの長さ。各 `xmlSchemaDocumentsPropertiesLengths` 値は `int` 値です。

### **xmlSchemaProperties**

XML スキーマ全体のプロパティーが含まれます。これは、外部 XML スキーマ・バージョン管理システムで使用されるプロパティーなどです。データベース・システムは、この値の内容の検証は行いません。これらの値は検索のために XSR 表に保管され、他のツールまたは XML スキーマ・リポジトリのインプリメンテーションで使用されます。 `xmlSchemaProperties` 値は、最大長が 5MB の String または `InputStream` 値です。渡されるプロパティーがない場合、値は NULL です。

### **xmlSchemaPropertiesLengths**

`registerDB2XmlSchema` の最初のフォームが使用される場合、`xmlSchemaProperties` パラメーター内の XML スキーマ・プロパティーの長さ。 `xmlSchemaPropertiesLengths` 値は `int` 値です。

### **isUsedForShredding**

XML 分解に使用されるアノテーションがスキーマ内にあるかどうかを示します。 `isUsedForShredding` はブール値です。

この方法は、IBM Informix Dynamic Server データ・ソースへの接続には適用されません。

### **updateDB2XmlSchema**

形式:

```
public void updateDB2XmlSchema(String[] targetSqlIdSchema,
    String[] targetSqlIdName,
    String[] sourceSqlIdSchema,
    String[] sourceSqlIdName,
    String[] xmlSchemaLocations,
    boolean dropSourceSchema)
    throws SQLException
```

XML スキーマの内容を、XML スキーマ・リポジトリ内の別の XML スキーマの内容で更新し、オプションでソース・スキーマをドロップします。UpdateDB2XmlSchema は、SYSPROC.XSR\_XSR\_UPDATE ストアド・プロシージャを呼び出して、XML スキーマを更新します。ターゲット XML スキーマ内のスキーマ文書は、ソース XML スキーマからのスキーマ文書で置き換えられます。updateDB2XmlSchema を呼び出す前に、ソースおよびターゲットの XML スキーマの登録を完了させておく必要があります。

ターゲット XML スキーマの更新には、SQL ALTERIN 特権が必要です。ソース XML スキーマのドロップには、SQL DROPIN 特権が必要です。

パラメーターの説明:

#### **targetSqlIdSchema**

更新される登録済み XML スキーマの SQL スキーマ名。  
targetSqlIdSchema は最大長が 128 バイトのストリング値です。

#### **targetSqlIdName**

更新される登録済み XML スキーマの名前。targetSqlIdName は最大長が 128 バイトのストリング値です。

#### **sourceSqlIdSchema**

ターゲット XML スキーマの更新に使用される、登録済み XML スキーマの SQL スキーマ名。sourceSqlIdSchema は最大長が 128 バイトのストリング値です。

#### **sourceSqlIdName**

ターゲット XML スキーマの更新に使用される、登録済み XML スキーマの名前。sourceSqlIdName は最大長が 128 バイトのストリング値です。

#### **dropSourceSchema**

ターゲット XML スキーマの更新後にソース XML スキーマがドロップされるかどうかを示します。dropSourceSchema はブール値です。false がデフォルトです。

この方法は、IBM Informix Dynamic Server データ・ソースへの接続には適用されません。

### **removeDB2JavaStoredProcedure**

形式:

```
public void DB2Connection.removeDB2JavaStoredProcedure(  
    String jarId)  
    throws java.sql.SQLException
```

sqlj.remove\_jar ストアド・プロシージャを DB2 Database for Linux, UNIX, and Windows サーバー上で呼び出し、そのサーバー用のカタログ内の JAR ファイルの定義を削除します。

パラメーターの説明:

#### **jarId**

データベース内の JAR の名前。schema.JAR-id または JAR-id の形式になります。これは、SQL ステートメントで JAR を参照する場合に使用する名前

です。 *schema* を省略した場合、データベース・システムは CURRENT SCHEMA 特殊レジスター内にある SQL 許可 ID を使用します。

この方法は、IBM Informix Dynamic Server データ・ソースへの接続には適用されません。

### replaceDB2JavaStoredProcedure

形式:

```
public void DB2Connection.replaceDB2JavaStoredProcedure(
    java.io.InputStream jarFile,
    int jarFileLength,
    String jarId)
    throws java.sql.SQLException
```

sqlj.replace\_jar ストアード・プロシージャを DB2 Database for Linux, UNIX, and Windows サーバー上で呼び出し、そのサーバー用のカタログ内の JAR ファイルの定義を置き換えます。

パラメーターの説明:

#### jarFile

サーバー上で置き換えられる JAR ファイルの内容。

#### jarFileLength

サーバー上で置き換えられる JAR ファイルの長さ。

#### jarId

データベース内の JAR の名前。 *schema.JAR-id* または *JAR-id* の形式になります。これは、SQL ステートメントで JAR を参照する場合に使用する名前です。 *schema* を省略した場合、データベース・システムは CURRENT SCHEMA 特殊レジスター内にある SQL 許可 ID を使用します。JAR の所有者は、CURRENT SQLID 特殊レジスター内の許可 ID です。

この方法は、IBM Informix Dynamic Server データ・ソースへの接続には適用されません。

### reuseDB2Connection (トラステッド接続再利用)

形式:

```
public Connection reuseDB2Connection(byte[] cookie,
    String user,
    String password,
    String usernameRegistry,
    byte[] userSecToken,
    String originalUser,
    java.util.Properties properties)
    throws java.sql.SQLException
public Connection reuseDB2Connection(byte[] cookie,
    org.ietf.GSSCredential gssCredential,
    String usernameRegistry,
    byte[] userSecToken,
    String originalUser,
    java.util.Properties properties)
    throws java.sql.SQLException
```

トラステッド・アプリケーション・サーバーで使用され、新規ユーザーの代わりに、前から存在するトラステッド接続を再利用します。リセットできるプロパティ (新規ユーザー ID を含む) が渡されます。データベース・サーバーは、関

連した物理接続をリセットします。 reuseDB2Connection が正常に実行する場合、異なるプロパティを持つ接続が使用可能になり、新規ユーザーが即時使用することができます。

reuseDB2Connection の 2 番目の形式は、IBM Data Server Driver for JDBC and SQLJ Type 4 接続でのみサポートされます。

パラメーターの説明:

#### **cookie**

JDBC ドライバーが Connection インスタンスについて生成する固有の cookie。 cookie は、初期トラステッド接続を確立したアプリケーション・サーバーおよび基礎となる JDBC ドライバーにのみ認識されます。アプリケーション・サーバーは、プール接続インスタンスの作成時にドライバーによって作成された cookie を渡します。 JDBC ドライバーは、提供された cookie が基礎となるトラステッド物理接続の cookie と一致することを調べて、そのトラステッド物理接続を確立したアプリケーション・サーバーから発信された要求を確認します。 cookies が一致する場合、異なるプロパティを持つ接続が使用可能になり、新規ユーザーが即時使用することができます。

#### **user**

データベース・システムがデータベース許可 ID を確立するために使用するクライアント ID。ユーザーがアプリケーション・サーバーによって認証されなかった場合、アプリケーション・サーバーは非認証 ユーザーを表すクライアント ID を渡す必要があります。

#### **password**

*user* のパスワード。

#### **gssCredential**

データ・ソースが Kerberos セキュリティーを使用する場合に、別のプリンシパルから渡される委任証明書を指定します。

#### **userNameRegistry**

ワークステーション・ユーザー ID を z/OS RACF ID にマップするマッピング・サービスを識別する名前。マッピング・サービスの例として、Integrated Security Services Enterprise Identity Mapping (EIM) があります。マッピング・サービスはプラグインによって定義されます。  
*userNameRegistry* の有効な値はプラグイン・プロバイダーによって定義されます。*userNameRegistry* が NULL の場合、*user* のマッピングは行われません。

#### **userSecToken**

クライアントのセキュリティー・トークン。この値は、DB2 for z/OS アカウンティング・データの一部としてトレースされます。*userSecToken* の内容はアプリケーション・サーバーによって記述され、データベース・システムによってアプリケーション・サーバー・セキュリティー・トークンとして参照されます。

#### **originalUser**

アプリケーション・サーバーによって使用されたオリジナル・ユーザー ID。

### properties

再利用される接続のプロパティ。

### reuseDB2Connection (再認証付きの非トラステッド再利用)

形式:

```
public DB2Connection reuseDB2Connection(String user,
    String password,
    java.util.Properties properties)
    throws java.sql.SQLException
public DB2Connection reuseDB2Connection(
    org.ietf.jgss.GSSCredential gssCredential,
    java.util.Properties properties)
    throws java.sql.SQLException
```

異種プール環境で、再認証後に既存の `Connection` インスタンスを再利用します。

パラメーターの説明:

### user

接続の確立に使用される許可 ID。

### password

接続の確立に使用される許可 ID のパスワード。

### gssCredential

データ・ソースが Kerberos セキュリティーを使用する場合に、別のプリンシパルから渡される委任証明書を指定します。

### properties

再利用される接続のプロパティ。これらのプロパティは、`DB2Connection` インスタンスですでに定義されているプロパティをオーバーライドします。

### reuseDB2Connection (再認証なしの非トラステッドまたはトラステッド再利用)

形式:

```
public DB2Connection reuseDB2Connection(java.util.Properties properties)
    throws java.sql.SQLException
```

再認証なしで、既存の接続インスタンスを再利用します。このメソッドは、プロパティが変更されない場合に `Connection` インスタンスを再利用することを意図しています。

このメソッドは、接続のダーティー再利用のためのものです。これは、オブジェクトがプールから再利用されると接続状態がリセットされないことを意味します。特殊レジスター設定およびプロパティ設定は、渡されるプロパティによってオーバーライドされない限り、依然として有効です。グローバル一時表は削除されません。指定されなかったプロパティは再初期化されません。分離レベル、自動コミット・モード、および読み取り専用モードなどの、JDBC 標準の一時プロパティはすべて JDBC デフォルトにリセットされます。 `user`、`password`、`databaseName`、`serverName`、`portNumber`、`planName`、および `pkList` などの特定のプロパティは未変更のままです。

パラメーターの説明:



## properties

再利用される接続のプロパティ。これらのプロパティは、DB2Connection インスタンスですでに定義されているプロパティをオーバーライドします。

## setDB2ClientAccountingInformation

形式:

```
public void setDB2ClientAccountingInformation(String info)
    throws java.sql.SQLException
```

接続のアカウントリング情報を指定します。この情報は、クライアントのアカウントリングを目的としています。この値は、接続中に変更することもできます。

パラメーターの説明:

### info

ユーザー指定のアカウントリング情報。最大長は、サーバーによって異なります。DB2 Database for Linux, UNIX, and Windows サーバーの場合、最大長は 255 バイトです。DB2 for z/OS サーバーの場合、最大長は 22 バイトです。Java 空ストリング ("") はこのパラメーター値として有効ですが、Java NULL 値は無効です。

**重要:** setDB2ClientAccountingInformation は、IBM Data Server Driver for JDBC and SQLJ の JDBC 4.0 インプリメンテーションでは推奨されていません。代わりに、java.sql.Connection.setClientInfo を使用してください。

この方法は、IBM Informix Dynamic Server データ・ソースへの接続には適用されません。

## setDB2ClientDebugInfo

形式:

```
public void setDB2ClientDebugInformation(String debugInfo)
    throws java.sql.SQLException
public void setDB2ClientDebugInformation(String mgrInfo,
    String traceInfo)
    throws java.sql.SQLException
```

CLIENT DEBUGINFO 接続属性の値を設定します。これはデータベース・システムに、接続を使用しているストアード・プロシージャおよびユーザー定義関数がデバッグ・モードで実行していることを通知します。CLIENT DEBUGINFO は、DB2 統合デバッガーにより使用されます。CLIENT DEBUGINFO ストリング全体を設定するには、最初の形式を使用してください。CLIENT DEBUGINFO ストリング内のセッション・マネージャーおよびトレース情報のみを変更するには、2 番目の形式を使用してください。

setDB2ClientDebugInfo メソッドは、DB2 for z/OS データベース・システムへの接続にのみ適用されます。

CLIENT DEBUGINFO 属性を、0 より長いストリングに設定する場合は、以下のいずれかの特権が必要です。

- DEBUGSESSION 特権
- SYSADM 権限

パラメーターの説明:



### **debugInfo**

以下の形式の、最大で 254 バイトのストリング。

*Mip:port,Iip,Ppid,Ttid,Cid,Llvl*

ストリングの各部の意味は、以下のとおりです。

**M** *ip:port*

セッション・マネージャーの IP アドレスおよびポート番号。

**Iip** クライアント IP アドレス

**Ppid** クライアント・プロセス ID

**Ttid** クライアント・スレッド ID (オプション)

**Cid** データ接続生成済み ID

**Llvl** デバッグ・ライブラリー診断トレース・レベル。

以下に例を示します。

*M9.72.133.89:8355,I9.72.133.89,P4552,T123,C1,L0*

このストリングの詳細記述については、SET CLIENT DEBUGINFO の説明を参照してください。

### **mgrInfo**

以下の形式のストリング。これは統合デバッガー・セッション・マネージャーの IP アドレスおよびポート番号を指定します。

*M ip:port*

以下に例を示します。

*M9.72.133.89:8355*

このストリングの詳細記述については、SET CLIENT DEBUGINFO の説明を参照してください。

### **trcInfo**

以下の形式のストリング。これはデバッグ・ライブラリー診断トレース・レベルを指定します。

*Llvl*

以下に例を示します。

*L0*

このストリングの詳細記述については、SET CLIENT DEBUGINFO の説明を参照してください。

この方法は、IBM Informix Dynamic Server データ・ソースへの接続には適用されません。

### **setDB2ClientProgramId**

形式:

```
public abstract void setDB2ClientProgramId(String program-ID)
    throws java.sql.SQLException
```

DB2 for z/OS サーバー上で、接続用のユーザー定義プログラム ID を設定します。このプログラム ID は、呼び出し元を識別するために使用される 80 バイトのストリングです。DB2 for z/OS サーバーは、ストリングを IFCID 316 トレース・レコード内に他の統計と共に配置するので、どのプログラムが特定の SQL ステートメントと関連しているかを識別できます。

この方法は、IBM Informix Dynamic Server データ・ソースへの接続には適用されません。

### setDB2ClientUser

形式:

```
public void setDB2ClientUser(String user)
    throws java.sql.SQLException
```

接続のための現行クライアントのユーザー名を指定します。この名前はクライアントのアカウントングを目的としており、JDBC 接続のユーザー値ではありません。JDBC 接続のユーザーの場合とは異なり、現在のクライアントのユーザー名は接続時に変更することができます。

パラメーターの説明:

#### user

現在のクライアントのユーザー ID。最大長は、サーバーによって異なります。DB2 Database for Linux, UNIX, and Windows サーバーの場合、最大長は 255 バイトです。DB2 for z/OS サーバーの場合、最大長は 16 バイトです。Java 空ストリング ("") はこのパラメーター値として有効ですが、Java NULL 値は無効です。

**重要:** getDB2ClientUser は、IBM Data Server Driver for JDBC and SQLJ の JDBC 4.0 インプリメンテーションでは推奨されていません。代わりに、java.sql.Connection.getClientInfo を使用してください。

この方法は、IBM Informix Dynamic Server データ・ソースへの接続には適用されません。

### setDB2ClientWorkstation

形式:

```
public void setDB2ClientWorkstation(String name)
    throws java.sql.SQLException
```

接続のための現在のクライアントのワークステーション名を指定します。この名前は、クライアントのアカウントングを目的としています。現在のクライアントのワークステーション名は、接続時に変更することができます。

パラメーターの説明:

#### 名前

現在のクライアントのワークステーション名。最大長は、サーバーによって異なります。DB2 Database for Linux, UNIX, and Windows サーバーの場合、最大長は 255 バイトです。DB2 for z/OS サーバーの場合、最大長は 18 バイトです。Java 空ストリング ("") はこのパラメーター値として有効ですが、Java NULL 値は無効です。

**重要:** `getDB2ClientWorkstation` は、IBM Data Server Driver for JDBC and SQLJ の JDBC 4.0 インプリメンテーションでは推奨されていません。代わりに、`java.sql.Connection.getClientInfo` を使用してください。

この方法は、IBM Informix Dynamic Server データ・ソースへの接続には適用されません。

### **setDB2CurrentPackagePath**

形式:

```
public void setDB2CurrentPackagePath(String packagePath)
    throws java.sql.SQLException
```

データベース・システムが JDBC および SQLJ パッケージを検索するコレクション ID のリストを指定します。

`setDB2CurrentPackagePath` メソッドは、DB2 データベース・システムへの接続にのみ適用されます。

パラメーターの説明:

#### **packagePath**

コレクション ID の、コンマで区切られたリスト。

### **setDB2CurrentPackageSet**

形式:

```
public void setDB2CurrentPackageSet(String packageSet)
    throws java.sql.SQLException
```

接続のためのコレクション ID を指定します。この値を設定すると、接続に使用される IBM Data Server Driver for JDBC and SQLJ インスタンスのコレクション ID も設定されます。

`setDB2CurrentPackageSet` メソッドは、DB2 データベース・システムへの接続にのみ適用されます。

パラメーターの説明:

#### **packageSet**

接続のためのコレクション ID。 `packageSet` 値の最大長は 18 バイトです。このメソッドは、SQL `SET CURRENT PACKAGESET` ステートメントの実行の代替方法としてプログラム内で呼び出すことができます。

### **setJccLogWriter**

形式:

```
public void setJccLogWriter(PrintWriter logWriter)
    throws java.sql.SQLException

public void setJccLogWriter(PrintWriter logWriter, int traceLevel)
    throws java.sql.SQLException
```

IBM Data Server Driver for JDBC and SQLJ トレースを有効または無効にします。または、アクティブな接続中にトレースの宛先を変更します。

パラメーターの説明:

### logWriter

IBM Data Server Driver for JDBC and SQLJ がトレース出力を書き込む、タイプ `java.io.PrintWriter` のオブジェクト。トレースをオフにするには、`logWriter` の値を `null` に設定します。

### traceLevel

収集するトレースのタイプを指定します。有効な値については、『IBM Data Server Driver for JDBC and SQLJ のプロパティ』の `traceLevel` プロパティに関する説明を参照してください。

## DB2ConnectionPoolDataSource クラス

`DB2ConnectionPoolDataSource` は `PooledConnection` オブジェクトのファクトリーです。このインターフェースをインプリメントするオブジェクトは、Java Naming and Directory Interface (JNDI) に基づくネーム・サービスで登録されます。

`com.ibm.db2.jcc.DB2ConnectionPoolDataSource` クラスは `com.ibm.db2.jcc.DB2BaseDataSource` クラスを拡張し、`javax.sql.ConnectionPoolDataSource`、`java.io.Serializable`、および `javax.naming.Referenceable` インターフェースをインプリメントします。

### DB2ConnectionPoolDataSource のプロパティ

これらのプロパティは、IBM Data Server Driver for JDBC and SQLJ に対してのみ定義されます。これらのプロパティの説明については、『IBM Data Server Driver for JDBC and SQLJ のプロパティ』を参照してください。

これらのプロパティには、プロパティの値を設定する `setXXX` メソッド、および値を検索する `getXXX` メソッドがあります。 `setXXX` メソッドの形式は以下のとおりです。

```
void setProperty-name(data-type property-value)
```

`getXXX` メソッドの形式は以下のとおりです。

```
data-type getProperty-name()
```

`Property-name` は、先頭文字が大文字になっている非修飾プロパティ名です。

以下の表に、IBM Data Server Driver for JDBC and SQLJ のプロパティとそのデータ・タイプをリストします。

表 87. `DB2ConnectionPoolDataSource` のプロパティとそれらのデータ・タイプ

プロパティ名	データ・タイプ
<code>com.ibm.db2.jcc.DB2ConnectionPoolDataSource.maxStatements</code>	<code>int</code>

## DB2ConnectionPoolDataSource のメソッド

### getDB2PooledConnection

形式:

```
public DB2PooledConnection getDB2PooledConnection(String user,  
String password,  
java.util.Properties properties)  
throws java.sql.SQLException
```

```
public DB2PooledConnection getDB2PooledConnection(  
    org.ietf.jgss.GSSCredential gssCredential,  
    java.util.Properties properties)  
    throws java.sql.SQLException
```

異種プール環境での初期非トラステッド接続を確立します。

最初のフォーム `getDB2PooledConnection` はユーザー ID とパスワードを提供します。2番目のフォーム `getDB2PooledConnection` は、Kerberos セキュリティーを使用する接続のためのものです。

パラメーターの説明:

**user**

接続の確立に使用される許可 ID。

**password**

接続の確立に使用される許可 ID のパスワード。

**gssCredential**

データ・ソースが Kerberos セキュリティーを使用する場合に、別のプリンシパルから渡される委任証明書を指定します。

**properties**

接続のプロパティー。

**getDB2TrustedPooledConnection**

形式:

```
public Object[] getDB2TrustedPooledConnection(String user,  
    String password,  
    java.util.Properties properties)  
    throws java.sql.SQLException  
public Object[] getDB2TrustedPooledConnection(  
    java.util.Properties properties)  
    throws java.sql.SQLException  
public Object[] getDB2TrustedPooledConnection(  
    org.ietf.jgss.GSSCredential gssCredential,  
    java.util.Properties properties)  
    throws java.sql.SQLException
```

システム許可 ID を使用するアプリケーション・サーバーは、トラステッド接続を確立するためにこのメソッドを使用します。以下のエレメントは `Object[]` で戻されます。

- 最初のエレメントは、トラステッド `DB2PooledConnection` インスタンスです。
- 2番目のエレメントは、生成されたプール接続インスタンスの固有の cookie です。

最初のフォーム `getDB2TrustedPooledConnection` はユーザー ID とパスワードを提供し、2番目のフォーム `getDB2TrustedPooledConnection` は、`DB2ConnectionPoolDataSource` オブジェクトのユーザー ID とパスワードを使用します。3番目のフォーム `getDB2TrustedPooledConnection` は、Kerberos セキュリティーを使用する接続のためのものです。

パラメーターの説明:

**user**

データベース・サーバーへのトラステッド接続の確立に使用される DB2 許可 ID。

**password**

トラステッド接続の確立に使用される許可 ID のパスワード。

**gssCredential**

データ・ソースが Kerberos セキュリティーを使用する場合に、別のプリンシパルから渡される委任証明書を指定します。

**properties**

接続のプロパティ。

## DB2DatabaseMetaData インターフェース

com.ibm.db2.jcc.DB2DatabaseMetaData は java.sql.DatabaseMetaData インターフェースを拡張するものです。

DB2DatabaseMetaData は java.sql.Wrapper インターフェースをインプリメントします。

### DB2DatabaseMetaData のメソッド:

以下のメソッドは、IBM Data Server Driver for JDBC and SQLJ に対してのみ定義されます。

**isIDSDatabaseAnsiCompliant**

形式:

```
public boolean isIDSDatabaseAnsiCompliant();
```

アクティブな IBM Informix Dynamic Server (IDS) データベースが ANSI 準拠である場合、true を戻します。そうでない場合は false を戻します。

ANSI 準拠データベースとは、WITH LOG MODE ANSI オプションを指定して作成されたデータベースのことです。

このメソッドは、IDS データ・ソースへの接続にのみ適用されます。データ・ソースが IDS データ・ソースでない場合、SQLException がスローされます。

**isIDSDatabaseLogging**

形式:

```
public boolean isIDSDatabaseLogging();
```

現在のアクティブな IDS データベースがロギングをサポートする場合、true を戻します。そうでない場合は false を戻します。

ロギングをサポートする IDS データベースとは、WITH LOG MODE ANSI、WITH BUFFERED LOG、または WITH LOG オプションを指定して作成されたデータベースです。

このメソッドは、IDS データ・ソースへの接続にのみ適用されます。データ・ソースが IDS データ・ソースでない場合、SQLException がスローされます。

### **isResetRequiredForDB2eWLM**

形式:

```
public boolean isResetRequiredForDB2eWLM();
```

ターゲット・データベース・サーバーが eWLM をサポートするためにクリーンな再利用を必要とする場合は、true を戻します。そうでない場合は false を戻します。

### **supportsDB2ProgressiveStreaming**

形式:

```
public boolean supportsDB2ProgressiveStreaming();
```

ターゲット・データ・ソースが連続ストリーミングをサポートする場合は、true を戻します。そうでない場合は false を戻します。

## **DB2Diagnosable インターフェース**

com.ibm.db2.jcc.DB2Diagnosable インターフェースは、SQLException からの DB2 診断を取得するメカニズムを提供します。

### **DB2Diagnosable のメソッド**

以下のメソッドは、IBM Data Server Driver for JDBC and SQLJ に対してのみ定義されます。

#### **getSqlca**

形式:

```
public DB2Sqlca getSqlca()
```

IBM Data Server Driver for JDBC and SQLJ の下で作成された java.sql.Exception からの DB2Sqlca オブジェクトを戻します。

#### **getThrowable**

形式:

```
public Throwable getThrowable()
```

IBM Data Server Driver for JDBC and SQLJ の下で作成された java.sql.Exception からの java.lang.Throwable オブジェクトを戻します。

#### **printTrace**

形式:

```
static public void printTrace(java.io.PrintWriter printWriter,  
                             String header)
```

IBM Data Server Driver for JDBC and SQLJ の下で java.sql.Exception がスローされた後に診断情報を印刷します。

パラメーターの説明:

#### **printWriter**

診断情報の宛先。

#### **header**

出力の先頭に印刷されるユーザー定義情報。



## DB2ExceptionFormatter クラス

com.ibm.db2.jcc.DB2ExceptionFormatter クラスには、診断情報をストリームに印刷するためのメソッドが含まれます。

### DB2ExceptionFormatter のメソッド

以下のメソッドは、IBM Data Server Driver for JDBC and SQLJ に対してのみ定義されます。

#### printTrace

形式:

```
static public void printTrace(java.sql.SQLException sqlException,  
    java.io.PrintWriter printWriter, String header)
```

```
static public void printTrace(DB2Sqlca sqlca,  
    java.io.PrintWriter printWriter, String header)
```

```
static public void printTrace(java.lang.Throwable throwable,  
    java.io.PrintWriter printWriter, String header)
```

例外がスローされた後に診断情報を印刷します。

パラメーターの説明:

#### SQLExceptionsqlcalthrowable

前の JDBC または Java 操作中にスローされた例外。

#### printWriter

診断情報の宛先。

#### header

出力の先頭に印刷されるユーザー定義情報。

## DB2JCCPlugin クラス

com.ibm.db2.jcc.DB2JCCPlugin クラスは、DB2 Database for Linux, UNIX, and Windows プラグイン・サポートを提供するためにインプリメント可能なメソッドを定義する、抽象クラスです。このクラスは DB2 Database for Linux, UNIX, and Windows だけに適用されます。

### DB2JCCPlugin のメソッド

以下のメソッドは、IBM Data Server Driver for JDBC and SQLJ に対してのみ定義されます。

#### getTicket

形式:

```
public abstract byte[] getTicket(String user,  
    String password,  
    byte[] returnedToken)  
    throws org.ietf.jgss.GSSException
```

ユーザー用の Kerberos チケットを検索します。

パラメーターの説明:

**user**

Kerberos チケットが検索されるユーザー ID。

**password**

*user* のパスワード。

**returnedToken**

## DB2PooledConnection クラス

`com.ibm.db2.jcc.DB2PooledConnection` クラスは、以前から存在するトラステッド接続でユーザーを切り替えるためにアプリケーション・サーバーが使用できるメソッドを提供します。

### DB2PooledConnection のメソッド

以下のメソッドは、IBM Data Server Driver for JDBC and SQLJ に対してのみ定義されます。

**getConnection (再認証なしの非トラステッドまたはトラステッド再利用)**

形式:

```
public DB2Connection getConnection()  
    throws java.sql.SQLException
```

このメソッドは、接続のダーティー再利用のためのものです。これは、オブジェクトがプールから再利用されると接続状態がリセットされないことを意味します。特殊レジスター設定およびプロパティー設定は、渡されるプロパティーによってオーバーライドされない限り、依然として有効です。グローバル一時表は削除されません。指定されなかったプロパティーは再初期化されません。分離レベル、自動コミット・モード、および読み取り専用モードなどの、JDBC 標準の一時プロパティーはすべて JDBC デフォルトにリセットされます。 *user*、*password*、*databaseName*、*serverName*、*portNumber*、*planName*、および *pkList* などの特定のプロパティーは未変更のままです。

**getDB2Connection (トラステッド再利用)**

形式:

```
public DB2Connection getDB2Connection(byte[] cookie,  
    String user,  
    String password,  
    String userRegistry,  
    byte[] userSecToken,  
    String originalUser,  
    java.util.Properties properties)  
    throws java.sql.SQLException  
public Connection getDB2Connection(byte[] cookie,  
    org.ietf.GSSCredential gssCredential,  
    String usernameRegistry,  
    byte[] userSecToken,  
    String originalUser,  
    java.util.Properties properties)  
    throws java.sql.SQLException
```

トラステッド接続に関連したユーザーを認証なしで切り替えます。

`reuseDB2Connection` の 2 番目の形式は、IBM Data Server Driver for JDBC and SQLJ Type 4 接続でのみサポートされます。

パラメーターの説明:

#### **cookie**

JDBC ドライバーが `Connection` インスタンスについて生成する固有の `cookie`。 `cookie` は、初期トラステッド接続を確立したアプリケーション・サーバーおよび基礎となる JDBC ドライバーにのみ認識されます。アプリケーション・サーバーは、プール接続インスタンスの作成時にドライバーによって作成された `cookie` を渡します。 JDBC ドライバーは、提供された `cookie` が基礎となるトラステッド物理接続の `cookie` と一致することを調べて、そのトラステッド物理接続を確立したアプリケーション・サーバーから発信された要求を確認します。 `cookie` が一致する場合、異なるプロパティを持つ接続が使用可能になり、新規ユーザーは即時使用することができます。

#### **user**

データ・ソースがデータベース・サーバーの許可 ID を確立するために使用するクライアント ID。ユーザーがアプリケーション・サーバーによって認証されなかった場合、アプリケーション・サーバーは非認証ユーザーを表すユーザー ID を渡す必要があります。

#### **password**

`user` のパスワード。

#### **gssCredential**

データ・ソースが Kerberos セキュリティーを使用する場合に、別のプリンシパルから渡される委任証明書を指定します。

#### **userNameRegistry**

ワークステーション・ユーザー ID を z/OS RACF ID にマップするマッピング・サービスを識別する名前。マッピング・サービスの例として、Integrated Security Services Enterprise Identity Mapping (EIM) があります。マッピング・サービスはプラグインによって定義されます。  
`userNameRegistry` の有効な値はプラグイン・プロバイダーによって定義されます。`userNameRegistry` が NULL の場合、接続はマッピング・サービスを使用しません。

#### **userSecToken**

クライアントのセキュリティ・トークン。この値は、DB2 for z/OS アカウンティング・データの一部としてトレースされます。`userSecToken` の内容はアプリケーション・サーバーによって記述され、データ・ソースによってアプリケーション・サーバー・セキュリティ・トークンとして参照されます。

#### **originalUser**

オリジナル要求をアプリケーション・サーバーに送信するクライアント ID。`originalUser` は、アプリケーション・サーバーによって使用されたオリジナル・ユーザー ID として、DB2 for z/OS アカウンティング・データに組み込まれます。

#### **properties**

再利用される接続のプロパティ。これらのプロパティは、`DB2PooledConnection` インスタンスですでに定義されているプロパティをオーバーライドします。

## getDB2Connection (再認証付きの非トラステッド再利用)

形式:

```
public DB2Connection getDB2Connection(  
    String user,  
    String password,  
    java.util.Properties properties)  
    throws java.sql.SQLException  
public DB2Connection getDB2Connection(org.ietf.jgss.GSSCredential gssCredential,  
    java.util.Properties properties)  
    throws java.sql.SQLException
```

非トラステッド接続に関連したユーザーを認証付きで切り替えます。

最初のフォーム `getDB2Connection` はユーザー ID とパスワードを提供します。2 番目のフォーム `getDB2Connection` は、Kerberos セキュリティーを使用する接続のためのものです。

パラメーターの説明:

### user

データ・ソースがデータベース・サーバーの許可 ID を確立するために使用するユーザー ID。

### password

`user` のパスワード。

### properties

再利用される接続のプロパティー。これらのプロパティーは、`DB2PooledConnection` インスタンスですでに定義されているプロパティーをオーバーライドします。

## getDB2Connection (再認証なしの非トラステッドまたはトラステッド再利用)

形式:

```
public java.sql.Connection getDB2Connection(  
    java.util.Properties properties)  
    throws java.sql.SQLException
```

再認証なしで、非トラステッド接続を再利用します。

このメソッドは、接続のダーティー再利用のためのものです。これは、オブジェクトがプールから再利用されると接続状態がリセットされないことを意味します。特殊レジスター設定およびプロパティー設定は、渡されるプロパティーによってオーバーライドされない限り、依然として有効です。グローバル一時表は削除されません。指定されなかったプロパティーは再初期化されません。分離レベル、自動コミット・モード、および読み取り専用モードなどの、JDBC 標準の一時プロパティーはすべて JDBC デフォルトにリセットされます。 `user`、`password`、`databaseName`、`serverName`、`portNumber`、`planName`、および `pkList` などの特定のプロパティーは未変更のままです。

パラメーターの説明:

### properties

再利用される接続のプロパティー。これらのプロパティーは、`DB2PooledConnection` インスタンスですでに定義されているプロパティーをオーバーライドします。

## DB2PoolMonitor クラス

com.ibm.db2.jcc.DB2PoolMonitor クラスでは、接続コンセントレーターと Sysplex ワークロード・บาลancingで使されるグローバル・トランスポート・オブジェクト・プールをモニターするメソッドが提供されます。

### DB2PoolMonitor フィールド

以下のフィールドは、IBM Data Server Driver for JDBC and SQLJ に対してのみ定義されます。

**public static final int TRANSPORT\_OBJECT = 1**

この値は DB2PoolMonitor.getPoolMonitor メソッドのパラメーターです。

### DB2PoolMonitor メソッド

以下のメソッドは、IBM Data Server Driver for JDBC and SQLJ に対してのみ定義されます。

#### agedOutObjectCount

形式:

```
public abstract int agedOutObjectCount()
```

db2.jcc.maxTransportObjectIdleTime で指定されていたアイドル時間を超過したためにプールから削除されたオブジェクトの数を取得します。

#### createdObjectCount

形式:

```
public abstract int createdObjectCount()
```

プールの作成時以降に IBM Data Server Driver for JDBC and SQLJ により作成されたオブジェクトの数を取得します。

#### getMonitorVersion

形式:

```
public int getMonitorVersion()
```

IBM Data Server Driver for JDBC and SQLJ に付属している DB2PoolMonitor クラスのバージョンを取得します。

#### getPoolMonitor

形式:

```
public static DB2PoolMonitor getPoolMonitor(int monitorType)
```

DB2PoolMonitor クラスのインスタンスを取得します。

パラメーターの説明:

#### monitorType

モニター・タイプ。この値は DB2PoolMonitor.TRANSPORT\_OBJECT でなければなりません。

#### heavyWeightReusedObjectCount

形式:

```
public abstract int heavyWeightReusedObjectCount()
```

プールから再利用されたオブジェクトの数を取得します。

### **lightWeightReusedObjectCount**

形式:

```
public abstract int lightWeightReusedObjectCount()
```

再利用されたがプール内に存在しなかったオブジェクトの数を取得します。これは、接続オブジェクトによってトランスポート・オブジェクトがトランザクション境界で解放された場合に発生することがあります。この接続オブジェクトで、そのトランスポート・オブジェクトが後で必要になった際に、元のトランスポート・オブジェクトが他のどの接続オブジェクトによってもまだ使用されていない場合は、この接続オブジェクトでそのトランスポート・オブジェクトを使用することができます。

### **longestBlockedRequestTime**

形式:

```
public abstract long longestBlockedRequestTime()
```

要求がブロックされた最長の時間をミリ秒単位で取得します。

### **numberOfConnectionReleaseRefused**

形式:

```
public abstract int numberOfConnectionReleaseRefused()
```

接続の解除が拒否された回数を取得します。

### **numberOfRequestsBlocked**

形式:

```
public abstract int numberOfRequestsBlocked()
```

プールに対して IBM Data Server Driver for JDBC and SQLJ から出されたが、プールがその最大容量に達したためプールによってブロックされた要求の数を取得します。ブロックされた要求が正常に完了する可能性があるのは、`db2.jcc.maxTransportObjectWaitTime` を超過して例外がスローされる前にオブジェクトがプールに返された場合です。

### **numberOfRequestsBlockedDataSourceMax**

形式:

```
public abstract int numberOfRequestsBlockedDataSourceMax()
```

プールに対して IBM Data Server Driver for JDBC and SQLJ から出されたが、プールが `DataSource` オブジェクトの最大容量に達したためプールによってブロックされた要求の数を取得します。

### **numberOfRequestsBlockedPoolMax**

形式:

```
public abstract int numberOfRequestsBlockedPoolMax()
```

プールに対して IBM Data Server Driver for JDBC and SQLJ から出されたが、プールの最大数に達したためプールによってブロックされた要求の数を取得します。

### **removedObjectCount**

形式:

```
public abstract int removedObjectCount()
```

プールの作成時以降にプールから削除されたオブジェクトの数を取得します。

### **shortestBlockedRequestTime**

形式:

```
public abstract long shortestBlockedRequestTime()
```

要求がブロックされた最短の時間をミリ秒単位で取得します。

### **successfulRequestsFromPool**

形式:

```
public abstract int successfulRequestsFromPool()
```

プールの作成時以降にプールに対して IBM Data Server Driver for JDBC and SQLJ から出されて正常に完了した要求の数を取得します。正常に完了した要求とは、プールからオブジェクトが返されたことを意味します。

### **totalPoolObjects**

形式:

```
public abstract int totalPoolObjects()
```

プール内に現在存在するオブジェクトの数を取得します。

### **totalRequestsToPool**

形式:

```
public abstract int totalRequestsToPool()
```

プールの作成時以降にプールに対して IBM Data Server Driver for JDBC and SQLJ から出された要求の総数を取得します。

### **totalTimeBlocked**

形式:

```
public abstract long totalTimeBlocked()
```

プールによってブロックされた要求のミリ秒単位の合計時間を取得します。アプリケーションによって複数のスレッドが使用される場合は、この時間は、アプリケーションの経過実行時間よりはるかに長くなる可能性があります。

## **DB2PreparedStatement インターフェース**

com.ibm.db2.jcc.DB2PreparedStatement インターフェースは、com.ibm.db2.jcc.DB2Statement および java.sql.PreparedStatement インターフェースを拡張するものです。

### **DB2PreparedStatement のメソッド**

以下のメソッドは、IBM Data Server Driver for JDBC and SQLJ に対してのみ定義されます。

#### **executeDB2QueryBatch**

形式:



```
public void executeDB2QueryBatch()  
    throws java.sql.SQLException
```

パラメーターが指定された照会を含むステートメント・バッチを実行します。

この方法は、IBM Informix Dynamic Server データ・ソースへの接続の場合はサポートされません。

## DB2ResultSet インターフェース

com.ibm.db2.jcc.DB2ResultSet インターフェースは、IBM Data Server Driver for JDBC and SQLJ 限定の照会情報が入手できるオブジェクトの作成に使用されます。

DB2ResultSet は、java.sql.Wrapper インターフェースをインプリメントします。

### DB2ResultSet のメソッド:

以下のメソッドは、IBM Data Server Driver for JDBC and SQLJ に対してのみ定義されます。

#### getDB2RowChangeToken

形式:

```
public long DB2ResultSet.getDB2RowChangeToken()  
    throws java.sql.SQLException
```

現在行の行変更トークンを (使用可能であれば) 戻します。オプティミスティック・ロック列が要求されていないか、または使用不可である場合は、0 を戻します。

#### getDB2RID

形式:

```
public Object DB2ResultSet.getDB2RID()  
    throws java.sql.SQLException
```

現在行の RID を (使用可能であれば) 戻します。オプティミスティック・ロック列が要求されており、使用可能であれば、RID は使用可能です。オプティミスティック・ロック列が要求されていないか、または使用不可である場合は、NULL を戻します。

#### getDB2RIDType

形式:

```
public int DB2ResultSet.getDB2RIDType()  
    throws java.sql.SQLException
```

DB2ResultSet 内の RID 列のデータ・タイプを戻します。戻り値は java.sql.Types 定数にマップされます。DB2ResultSet に RID 列が含まれていない場合、java.sql.Types.NULL が戻されます。

## DB2ResultSetMetaData インターフェース

com.ibm.db2.jcc.DB2ResultSetMetaData インターフェースには、ResultSet オブジェクトに関する情報を提供するメソッドが用意されています。

com.ibm.db2.jcc.DB2ResultSetMetaData メソッドを使用する前に、  
java.sql.ResultSet.getMetaData 呼び出しから戻される java.sql.ResultSetMetaData オブジェクトを com.ibm.db2.jcc.DB2ResultSetMetaData にキャストする必要があります。

## DB2ResultSetMetaData のメソッド:

以下のメソッドは、IBM Data Server Driver for JDBC and SQLJ に対してのみ定義されます。

### getDB2OptimisticLockingColumns

形式:

```
public int getDB2OptimisticLockingColumns()  
    throws java.sql.SQLException
```

オプティミスティック・ロック列が使用可能であるかどうかを示す値を返します。可能な値は以下のとおりです。

- 0 オプティミスティック・ロック列は使用不可です。
- 1 オプティミスティック・ロック列は使用可能ですが、変更トークンが false negative を避ける細分性を持っていない可能性があります。
- 2 オプティミスティック・ロック列は使用可能であり、変更トークンは false negative を避ける細分性を持っています。

### isDB2ColumnNameDerived

形式:

```
public boolean isDB2ColumnNameDerived (int column)  
    throws java.sql.SQLException
```

ResultSet 列の名前が、ResultSet を生成した SQL SELECT リストから派生したものである場合に、true を返します。

たとえば、ResultSet が SQL ステートメント SELECT EMPNAME, SUM(SALARY) FROM EMP から生成されるとします。列名 EMPNAME ResultSet は SQL SELECT リストから派生していますが、SUM(SALARY) に対応する ResultSet 内の列名は SELECT リストから派生しません。

パラメーターの説明:

#### column

ResultSet 内の列の名前。

## DB2RowID インターフェース

com.ibm.db2.jcc.DB2RowID インターフェースは、SQL ROWID データ・タイプで使用するための Java オブジェクトを宣言するために使用されます。

### DB2RowID のメソッド

以下のメソッドは、IBM Data Server Driver for JDBC and SQLJ に対してのみ定義されます。

#### getBytes

形式:

```
public byte[] getBytes()
```

com.ibm.jcc.DB2RowID オブジェクトをバイトに変換します。

## DB2SimpleDataSource クラス

com.ibm.db2.jcc.DB2SimpleDataSource クラスは DB2BaseDataSource クラスを拡張するものです。

DB2BaseDataSource オブジェクトは、接続プールまたは分散トランザクションはサポートしていません。これには DB2BaseDataSource クラスに含まれるすべてのプロパティとメソッドが含まれています。加えて、DB2SimpleDataSource には以下の IBM Data Server Driver for JDBC and SQLJ 専用プロパティが含まれています。

DB2SimpleDataSource は java.sql.Wrapper インターフェースをインプリメントします。

### DB2SimpleDataSource のプロパティ

以下のプロパティは、IBM Data Server Driver for JDBC and SQLJ に対してのみ定義されます。このプロパティの説明については、『IBM Data Server Driver for JDBC and SQLJ のプロパティ』を参照してください。

```
String com.ibm.db2.jcc.DB2SimpleDataSource.password
```

### DB2SimpleDataSource のメソッド

以下のメソッドは、IBM Data Server Driver for JDBC and SQLJ に対してのみ定義されます。

#### setPassword

形式:

```
public void setPassword(String password)
```

DB2SimpleDataSource オブジェクトのパスワードを設定します。対応する getPassword メソッドはありません。したがって、パスワードを検索して暗号化解除する方法がないため、パスワードは暗号化できません。

## DB2Sqlca クラス

com.ibm.db2.jcc.DB2Sqlca クラスは SQLCA をカプセル化したものです。

### DB2Sqlca のメソッド

以下のメソッドは、IBM Data Server Driver for JDBC and SQLJ に対してのみ定義されます。

#### getMessage

形式:

```
public abstract String getMessage()
```

エラー・メッセージ・テキストを戻します。

#### getSqlCode

形式:

```
public abstract int getSqlCode()
```

SQL エラー・コード値を戻します。

#### **getSqlErrd**

形式:

```
public abstract int[] getSqlErrd()
```

各エレメントに SQLCA SQLERRD が含まれている配列を戻します。

#### **getSqlErrmc**

形式:

```
public abstract String getSqlErrmc()
```

スペースで区切られた SQLCA SQLERRMC 値を含むSTRINGを戻します。

#### **getSqlErrmcTokens**

形式:

```
public abstract String[] getSqlErrmcTokens()
```

各エレメントに SQLCA SQLERRMC トークンが含まれている配列を戻します。

#### **getSqlErrp**

形式:

```
public abstract String getSqlErrp()
```

SQLCA SQLERRP 値を戻します。

#### **getSqlState**

形式:

```
public abstract String getSqlState()
```

SQLCA SQLSTATE 値を戻します。

#### **getSqlWarn**

形式:

```
public abstract char[] getSqlWarn()
```

各エレメントに SQLCA SQLWARN 値が含まれている配列を戻します。

## **DB2Statement インターフェース**

com.ibm.db2.jcc.DB2Statement インターフェースは java.sql.Statement インターフェースを拡張するものです。

DB2Statement は java.sql.Wrapper インターフェースをインプリメントします。

### **DB2Statement のフィールド**

以下のフィールドは、IBM Data Server Driver for JDBC and SQLJ に対してのみ定義されます。

```
public static final int RETURN_OPTLOCK_COLUMN_NONE = 0  
public static final int RETURN_OPTLOCK_COLUMN_ALWAYS = 1
```

```
public static final int RETURN_OPTLOCK_COLUMN_NO_FALSE_NEGATIVES
= 2
```

これらの値は `DB2Statement.executeDB2OptimisticLockingQuery` メソッドの引数です。

## DB2Statement のメソッド

以下のメソッドは、IBM Data Server Driver for JDBC and SQLJ に対してのみ定義されます。

### executeDB2OptimisticLockingQuery

形式:

```
public java.sql.ResultSet DB2Statement.executeDB2OptimisticLockingQuery(
    String sql,
    int returnOptLockingColumn)
    throws java.sql.SQLException
```

SQL 照会ステートメントを実行し、要求された場合はオプティミスティック・ロック情報を含む `ResultSet` を戻します。

パラメーターの説明:

**sql** 単一の `ResultSet` を戻す SQL `SELECT` ステートメント。

### returnOptimisticLockingColumns

オプティミスティック・ロック列を戻すかどうかを指定します。可能な値は以下のとおりです。

表 88.

値	説明
<code>DB2Statement.RETURN_OPTLOCK_COLUMN_NONE (0)</code>	オプティミスティック・ロック列を戻しません。
<code>DB2Statement.RETURN_OPTLOCK_COLUMN_ALWAYS (1)</code>	行の変更列が単一の行を一意的に表さない場合でも、行の変更列を結果セットに追加します。この設定は、データベース準備属性 <code>WITH ROW CHANGE COLUMNS POSSIBLY DISTINCT</code> に相当します。
<code>DB2Statement.RETURN_OPTLOCK_COLUMN_NO_FALSE_NEGATIVES (2)</code>	行の変更列が単一の行を一意的に表す場合にのみ、その行の変更列を結果セットに追加します。この設定は、データベース準備属性 <code>WITH ROW CHANGE COLUMNS ALWAYS DISTINCT</code> に相当します。

### getDB2ClientProgramId

形式:

```
public String getDB2ClientProgramId()
    throws java.sql.SQLException
```

データ・ソース上に保管されている、接続のユーザー定義クライアント・プログラム ID を戻します。

### setDB2ClientProgramId

形式:

```
public abstract void setDB2ClientProgramId(String program-ID)
    throws java.sql.SQLException
```

DB2 for z/OS サーバー上で、接続用のユーザー定義プログラム ID を設定します。このプログラム ID は、呼び出し元を識別するために使用される 80 バイ

トのストリングです。DB2 for z/OS サーバーは、ストリングを IFCID 316 トレース・レコード内に他の統計と共に配置するので、どのプログラムが特定の SQL ステートメントと関連しているかを識別できます。

### **getIDSSerial**

形式:

```
public int getIDSSerial()  
    throws java.sql.SQLException
```

前に実行された INSERT ステートメントの SERIAL 列から自動生成キーを検索します。

getIDSSerial を正常に実行するには、以下の条件が当てはまらなければなりません。

- INSERT ステートメントが、このメソッドが呼び出される前に実行された最後の SQL ステートメントである。
- この行が挿入される表に SERIAL 列が含まれている。
- INSERT ステートメントを準備または実行する JDBC Connection.prepareStatement メソッドまたは Statement.executeUpdate メソッドのフォームには、自動生成キーを要求するパラメーターがない。

このメソッドは、IBM Informix Dynamic Server (IDS) データベースへの接続にのみ適用されます。

### **getIDSSerial8**

形式:

```
public long getIDSSerial8()  
    throws java.sql.SQLException
```

前に実行された INSERT ステートメントの SERIAL8 列から自動生成キーを検索します。

getIDSSerial8 を正常に実行するには、以下の条件が当てはまらなければなりません。

- INSERT ステートメントが、このメソッドが呼び出される前に実行された最後の SQL ステートメントである。
- この行が挿入される表に SERIAL8 列が含まれている。
- INSERT ステートメントを準備または実行する JDBC Connection.prepareStatement メソッドまたは Statement.executeUpdate メソッドのフォームには、自動生成キーを要求するパラメーターがない。

このメソッドは、IDS データ・ソースへの接続にのみ適用されます。

### **getIDSSQLStatementOffSet**

形式:

```
public int getIDSSQLStatementOffSet()  
    throws java.sql.SQLException
```

IDS データ・ソース上で SQL ステートメントを実行した後に、ステートメントに構文エラーがあると、getIDSSQLStatementOffSet は構文エラーのステートメント・テキストにオフセットを戻します。

getIDSSQLStatementOffSet は以下を戻します。

- 0、ステートメントに構文エラーがない場合。
- -1、データ・ソースが IDS ではない場合。

このメソッドは、IDS データ・ソースへの接続にのみ適用されます。

## DB2SystemMonitor インターフェース

com.ibm.db2.jcc.DB2SystemMonitor インターフェースは、接続用のシステム・モニター・データを収集するために使用されます。接続ごとに 1 つの DB2SystemMonitor インスタンスを持つことができます。

### DB2SystemMonitor のフィールド

以下のフィールドは、IBM Data Server Driver for JDBC and SQLJ に対してのみ定義されます。

**public final static int RESET\_TIMES**

**public final static int ACCUMULATE\_TIMES**

これらの値は DB2SystemMonitor.start メソッドの引数です。RESET\_TIMES は、モニターの開始前に時間カウンターをゼロに設定します。

ACCUMULATE\_TIMES は時間カウンターをゼロに設定しません。

### DB2SystemMonitor のメソッド

以下のメソッドは、IBM Data Server Driver for JDBC and SQLJ に対してのみ定義されます。

**enable**

形式:

```
public void enable(boolean on)
    throws java.sql.SQLException
```

接続と関連付けられたシステム・モニターを使用可能にします。このメソッドをモニター中に呼び出すことはできません。enable を呼び出すとすべての時間がリセットされます。

**getApplicationTimeMillis**

形式:

```
public long getApplicationTimeMillis()
    throws java.sql.SQLException
```

アプリケーション、JDBC ドライバー、ネットワーク入出力、およびデータベース・サーバーの経過時間の合計を戻します。ミリ秒単位の時間。

モニターされる経過時間インターバルは、JDBC ドライバーの処理における以下のポイント間の、ミリ秒単位の差異です。

**インターバルの開始**

start の呼び出し時。

**インターバルの終了**

stop の呼び出し時。



システム・モニターが無効の場合、`getApplicationTimeMillis` は 0 を返します。最初に `stop` メソッドを呼び出さずにこのメソッドを呼び出すと、結果は `SQLException` になります。

### **getCoreDriverTimeMicros**

形式:

```
public long getCoreDriverTimeMicros()  
    throws java.sql.SQLException
```

システム・モニターが有効であったときに収集された、経過モニター API 時間の合計を返します。この時間はマイクロ秒単位です。

モニターされる API は、処理時間が収集される JDBC ドライバー・メソッドです。一般に、経過時間はネットワーク入出力またはデータベース・サーバー対話の結果となる API についてのみモニターされます。例えば、`PreparedStatement.setXXX` メソッドと `ResultSet.getXXX` メソッドはモニターされません。

モニターされる API 経過時間には、メソッド呼び出しのドライバーに費やされる合計時間が含まれます。この時間には、ネットワーク入出力時間およびデータベース・サーバー経過時間が含まれます。

モニターされる API 経過時間インターバルは、JDBC ドライバーの処理における以下のポイント間の、マイクロ秒単位の差異です。

#### **インターバルの開始**

モニターされる API がアプリケーションにより呼び出される時。

#### **インターバルの終了**

モニターされる API が制御をアプリケーションに戻す直前。

システム・モニターが無効の場合、`getCoreDriverTimeMicros` は 0 を返します。最初に `stop` メソッドを呼び出さずにこのメソッドを呼び出すか、または基礎となる JVM がマイクロ秒単位のレポート時間をサポートしないと、結果は `SQLException` になります。

### **getNetworkIOTimeMicros**

形式:

```
public long getNetworkIOTimeMicros()  
    throws java.sql.SQLException
```

システム・モニターが有効であったときに収集された、経過ネットワーク入出力時間の合計を返します。この時間はマイクロ秒単位です。

経過ネットワーク入出力時間には、ネットワーク入出力ストリームとの間での DRDA データの読み取り/書き込み時間が含まれます。ネットワーク入出力経過時間インターバルは、JDBC ドライバーで以下の操作を実行するための時間インターバルです。

- DRDA メッセージをデータベース・サーバーに送信するために TCP/IP コマンドを発行する。この時間インターバルは、ネットワーク入出力ストリームとの間での書き込み/フラッシュが実行される直前と直後のポイント間の、マイクロ秒単位の差異です。

- DRDA 応答メッセージをデータベース・サーバーから受け取るために TCP/IP コマンドを発行する。この時間インターバルは、ネットワーク入出力ストリーム上で読み取りが実行される直前と直後のポイント間の、マイクロ秒単位の差異です。

ネットワーク入出力時間インターバルは、すべての送受信操作についてキャプチャーされます。これにはコミットとロールバックのためのメッセージの送信も含まれます。

ネットワーク入出力を待機するために費やす時間は、優先順位が低い SQL 要求のためのデータベース・サーバーでの CPU ディスパッチングの遅延により影響を受ける場合があります。

システム・モニターが無効の場合、`getNetworkIOTimeMicros` は 0 を戻します。最初に `stop` メソッドを呼び出さずにこのメソッドを呼び出すか、または基礎となる JVM がマイクロ秒単位のレポート時間をサポートしないと、結果は `SQLException` になります。

### **getServerTimeMicros**

形式:

```
public long getServerTimeMicros()  
    throws java.sql.SQLException
```

システム・モニターが有効であったときに収集された、レポートされたすべてのデータベース・サーバー経過時間の合計を戻します。この時間はマイクロ秒単位です。

データベース・サーバーは、経過時間を以下の条件下でレポートします。

- データベース・サーバーが経過時間データをクライアントに戻すことをサポートしている。

DB2 Database for Linux, UNIX, and Windows バージョン 9.5 以降、および DB2 for z/OS はこの機能をサポートします。

- データベース・サーバーがモニター可能な操作を実行している。例えば、データベース・サーバー経過時間はコミットまたはロールバックのためには戻されません。

データベース・サーバー経過時間は、要求データ・ストリームの解析、コマンドの処理、データベース・サーバーでの応答データ・ストリームの生成のための経過時間と定義されます。データ・ストリームの送受信のためのネットワーク時間は含まれません。

データベース・サーバー経過時間インターバルは、データベース・サーバー処理における以下のポイント間の、マイクロ秒単位の差異です。

#### **インターバルの開始**

オペレーティング・システムが、JDBC ドライバーから受け取った TCP/IP メッセージの処理をデータベース・サーバーにディスパッチした時。

#### **インターバルの終了**

データベース・サーバーが応答メッセージをクライアントに戻すために TCP/IP コマンドを発行する準備ができていた時。

システム・モニターが無効の場合、`getServerTimeMicros` は 0 を返します。最初に `stop` メソッドを呼び出さずにこのメソッドを呼び出すと、結果は `SQLException` になります。

#### **start**

形式:

```
public void start (int lapMode)
    throws java.sql.SQLException
```

システム・モニターが有効な場合、`start` は接続のシステム・モニター・データの収集を開始します。`lapMode` の有効値は、`RESET_TIMES` または `ACCUMULATE_TIMES` です。

このメソッドを無効なシステム・モニターを使用して呼び出しても、何も実行されません。このメソッドを複数回、`stop` 呼び出しを間に挟まずに呼び出すと、結果は `SQLException` になります。

#### **stop**

形式:

```
public void stop()
    throws java.sql.SQLException
```

システム・モニターが有効な場合、`stop` は接続のシステム・モニター・データの収集を停止します。モニターが停止した後に、モニター時間は `DB2SystemMonitor` の `getXXX` メソッドで入手できます。

このメソッドを無効なシステム・モニターを使用して呼び出しても、何も実行されません。このメソッドを最初に `start` を呼び出さずに呼び出すか、または `start` 呼び出しを間に挟まずに複数回呼び出すと、結果は `SQLException` になります。

## **DB2TraceManager クラス**

`com.ibm.db2.jcc.DB2TraceManager` クラスはグローバル・ログ書き込みプログラムを制御します。

グローバル・ログ書き込みプログラムはドライバー全体にわたるものであり、すべての接続に適用されます。グローバル・ログ書き込みプログラムは、その他の JDBC ログ書き込みプログラムをオーバーライドします。`DB2TraceManager` クラスは、グローバル・ログ書き込みプログラムを始動するだけでなく、すべてのタイプのログ書き込みプログラムによるトレースを中断および再開する機能を提供します。つまり、`DB2TraceManager` クラスの中断および再開メソッドは、現在および将来の `DriverManager` ログ書き込みプログラム、`DataSource` ログ書き込みプログラム、または `IBM Data Server Driver for JDBC and SQLJ` 限定の接続レベルのログ書き込みプログラムすべてに適用されることになります。

### **DB2TraceManager のメソッド**

#### **getTraceManager**

形式:

```
static public DB2TraceManager getTraceManager()
    throws java.sql.SQLException
```

グローバル・ログ書き込みプログラムのインスタンスを取得します。

## setLogWriter

形式:

```
public abstract void setLogWriter(String traceDirectory,
    String baseTraceFileName, int traceLevel)
    throws java.sql.SQLException
public abstract void setLogWriter(String traceFile,
    boolean fileAppend, int traceLevel)
    throws java.sql.SQLException
public abstract void setLogWriter(java.io.PrintWriter logWriter,
    int traceLevel)
    throws java.sql.SQLException
```

グローバル・トレースを有効にします。 `setLogWriter` が呼び出されると、`DB2TraceManager.unsetLogWriter` が呼び出されるまで、`DataSource` または `Connection` トレースに対するすべての呼び出しは廃棄されます。

`setLogWriter` が呼び出されると、使用する `setLogWriter` の形式に応じて、将来の `Connection` または `DataSource` トレースはすべてトレース・ファイルまたは `PrintWriter` にリダイレクトされます。 `setLogWriter` が呼び出されるとグローバル・トレースは中断され、トレースが再開されたときに指定した設定が有効になります。

パラメーターの説明:

### traceDirectory

グローバル・トレース情報の書き込み先のディレクトリーを指定します。この設定は、`DataSource` または `DriverManager` 接続の `traceDirectory` および `logWriter` プロパティーの設定をオーバーライドします。

`traceDirectory` パラメーターを指定して `setLogWriter` の形式を使用すると、`JDBC` ドライバーは、`setLogWriter` が呼び出されたときに `traceFileAppend` プロパティーを `false` に設定します。つまり、既存のログ・ファイルは上書きされます。`JDBC` ドライバーの各接続は、指定したディレクトリー内の異なるファイルにトレースされます。そのディレクトリー内のファイルの命名規則は、`baseTraceFileName` に `NULL` 以外の値を指定するかどうかに応じて以下のように異なります。

- `baseTraceFileName` に `NULL` 値を指定すると、接続のトレースは `traceFile_global_n` という名前のファイルへ書き込まれます。

`n` は `n` 番目の `JDBC` ドライバー接続です。

- `baseTraceFileName` に `NULL` 以外の値を指定すると、接続のトレースは `baseTraceFileName_global_n` という名前のファイルへ書き込まれます。

`baseTraceFileName` は `baseTraceFileName` パラメーターの値です。

`n` は `n` 番目の `JDBC` ドライバー接続です。

### baseTraceFileName

グローバル・トレース情報の書き込み先のファイル名の語幹を指定します。`baseTraceFileName` と `traceDirectory` の組み合わせによって、グローバル・トレース・ログ・ファイルの絶対パス名が決定されます。

### traceFileName

グローバル・トレース情報の書き込み先のファイルを指定します。この設定

は、DataSource または DriverManager 接続の traceFile および logWriter プロパティの設定をオーバーライドします。

traceFileName パラメーターを指定して setLogWriter の形式を使用すると、1 つのログ・ファイルだけが書き込まれます。

traceFileName にはディレクトリー・パスを含めることができます。

### logWriter

すべてのグローバル・ログ・レコードの書き込み先の文字出力ストリームを指定します。

この値は、DataSource または DriverManager 接続で logWriter プロパティをオーバーライドします。

### traceLevel

何をトレースするかを指定します。

traceLevel パラメーターで、以下のトレースを 1 つ以上指定することができます。

- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_NONE (X'00')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_CONNECTION\_CALLS (X'01')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_STATEMENT\_CALLS (X'02')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_RESULT\_SET\_CALLS (X'04')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_DRIVER\_CONFIGURATION (X'10')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_CONNECTS (X'20')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_DRDA\_FLOWS (X'40')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_RESULT\_SET\_META\_DATA (X'80')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_PARAMETER\_META\_DATA (X'100')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_DIAGNOSTICS (X'200')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_SQLJ (X'400')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_XA\_CALLS (DB2 Database for Linux, UNIX, and Windows 用の IBM Data Server Driver for JDBC and SQLJ Type 2 接続のみ) (X'800')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_META\_CALLS (X'2000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_DATASOURCE\_CALLS (X'4000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_LARGE\_OBJECT\_CALLS (X'8000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_SYSTEM\_MONITOR (X'20000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_TRACEPOINTS (X'40000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_ALL (X'FFFFFFFF')

複数のトレースを指定するには、以下の技法のいずれかを使用します。

- 2 つ以上のトレース値にビット単位 OR (|) 演算子を使用する。たとえば、DRDA フローおよび接続呼び出しをトレースするには、traceLevel に以下の値を指定します。

```
TRACE_DRDA_FLOWS|TRACE_CONNECTION_CALLS
```

- ビット単位の補数 ( 波形記号 (~)) 演算子とトレース値を使用して、特定のトレース以外のすべてを指定する。たとえば、DRDA フロー以外のすべてをトレースする場合、traceLevel に以下の値を指定します。

```
~TRACE_DRDA_FLOWS
```

### fileAppend

traceFile パラメーターによって指定されるファイルへ付加するか、あるいは上書きするかを指定します。 true は既存のファイルを上書きしないことを意味します。

### unsetLogWriter

形式:

```
public abstract void unsetLogWriter()  
    throws java.sql.SQLException
```

将来の接続に対してグローバル・ログ書き込みプログラムを無効にします。

### suspendTrace

形式:

```
public void suspendTrace()  
    throws java.sql.SQLException
```

現在および将来の接続に対して、Connection レベルまたは DataSource レベルのグローバル・トレースをすべて中断します。グローバル・ログ書き込みプログラムを有効または無効にするときに、suspendTrace を呼び出すことができます。

### resumeTrace

形式:

```
public void resumeTrace()  
    throws java.sql.SQLException
```

現在および将来の接続に対して、Connection レベルまたは DataSource レベルのグローバル・トレースをすべて再開します。グローバル・ログ書き込みプログラムを有効または無効にするときに、resumeTrace を呼び出すことができます。グローバル・ログ書き込みプログラムを無効にすると、resumeTrace は Connection レベルまたは DataSource レベルのトレースを再開します。グローバル・ログ書き込みプログラムを有効にすると、resumeTrace はグローバル・トレースを再開します。

### getLogWriter

形式:

```
public abstract java.io.PrintWriter getLogWriter()  
    throws java.sql.SQLException
```

グローバル・ログ書き込みプログラムが設定されている場合、その PrintWriter を戻します。それ以外の場合、getLogWriter は NULL を戻します。

### getTraceFile

形式:

```
public abstract String getTraceFile()  
    throws java.sql.SQLException
```

グローバル・ログ書き込みプログラムが設定されている場合、その宛先ファイルの名前を戻します。それ以外の場合、getTraceFile は NULL を戻します。



### **getTraceDirectory**

形式:

```
public abstract String getTraceDirectory()  
    throws java.sql.SQLException
```

グローバル・ログ書き込みプログラム・ファイルが設定されている場合、その宛先ディレクトリーの名前を戻します。それ以外の場合、`getTraceDirectory` は `NULL` を戻します。

### **getTraceLevel**

形式:

```
public abstract int getTraceLevel()  
    throws java.sql.SQLException
```

グローバル・トレースが設定されている場合、そのトレース・レベルを戻します。それ以外の場合、`getTraceLevel` は `-1 (TRACE_ALL)` を戻します。

### **getTraceFileAppend**

形式:

```
public abstract boolean getTraceFileAppend()  
    throws java.sql.SQLException
```

グローバル・トレース・レコードがトレース・ファイルに追加されている場合、`true` を戻します。それ以外の場合、`getTraceFileAppend` は `false` を戻します。

## **DB2TraceManagerMXBean インターフェース**

`com.ibm.db2.jcc.mx.DB2TraceManagerMXBean` インターフェースは、アプリケーションがリモート・トレース・コントローラー用の `MXBean` として `DB2TraceManager` を使用可能にするための手段です。

### **DB2TraceManagerMXBean のメソッド**

#### **setTraceFile**

形式:

```
public void setTraceFile(String traceFile,  
    boolean fileAppend, int traceLevel)  
    throws java.sql.SQLException
```

リモート・トレース・マネージャーがトレース情報を書き込むファイルの名前、およびトレースされる情報のタイプを指定します。

パラメーターの説明:

#### **traceFileName**

グローバル・トレース情報の書き込み先のファイルを指定します。この設定は、`DataSource` または `DriverManager` 接続の `traceFile` および `logWriter` プロパティーの設定をオーバーライドします。

`traceFileName` パラメーターを指定して `setLogWriter` の形式を使用すると、1 つのログ・ファイルだけが書き込まれます。

`traceFileName` にはディレクトリー・パスを含めることができます。



### fileAppend

traceFile パラメーターによって指定されるファイルへ付加するか、あるいは上書きするかを指定します。 true は既存のファイルを上書きしないことを意味します。

### traceLevel

何をトレースするかを指定します。

traceLevel パラメーターで、以下のトレースを 1 つ以上指定することができます。

- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_NONE (X'00')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_CONNECTION\_CALLS (X'01')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_STATEMENT\_CALLS (X'02')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_RESULT\_SET\_CALLS (X'04')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_DRIVER\_CONFIGURATION (X'10')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_CONNECTS (X'20')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_DRDA\_FLOWS (X'40')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_RESULT\_SET\_META\_DATA (X'80')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_PARAMETER\_META\_DATA (X'100')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_DIAGNOSTICS (X'200')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_SQLJ (X'400')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_XA\_CALLS (DB2 Database for Linux, UNIX, and Windows 用の IBM Data Server Driver for JDBC and SQLJ Type 2 接続のみ) (X'800')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_META\_CALLS (X'2000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_DATASOURCE\_CALLS (X'4000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_LARGE\_OBJECT\_CALLS (X'8000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_SYSTEM\_MONITOR (X'20000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_TRACEPOINTS (X'40000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_ALL (X'FFFFFFFF')

複数のトレースを指定するには、以下の技法のいずれかを使用します。

- 2 つ以上のトレース値にビット単位 OR (|) 演算子を使用する。たとえば、DRDA フローおよび接続呼び出しをトレースするには、traceLevel に以下の値を指定します。

```
TRACE_DRDA_FLOWS|TRACE_CONNECTION_CALLS
```

- ビット単位の補数 (波形記号 (~)) 演算子とトレース値を使用して、特定のトレース以外のすべてを指定する。たとえば、DRDA フロー以外のすべてをトレースする場合、traceLevel に以下の値を指定します。

```
~TRACE_DRDA_FLOWS
```

### getTraceFile

形式:

```
public void getTraceFile()  
    throws java.sql.SQLException
```

リモート・トレース・コントローラーの宛先ファイルの名前が設定されている場合、その宛先ファイルの名前を戻します。それ以外の場合、`getTraceFile` は `NULL` を戻します。

### **setTraceDirectory**

形式:

```
public void setTraceDirectory(String traceDirectory,  
    String baseTraceFileName,  
    int traceLevel) throws java.sql.SQLException
```

リモート・トレース・コントローラーがトレース情報を書き込むディレクトリーの名前、およびトレースされる情報のタイプを指定します。

パラメーターの説明:

#### **traceDirectory**

トレース情報の書き込み先のディレクトリーを指定します。この設定は、`DataSource` または `DriverManager` 接続の `traceDirectory` および `logWriter` プロパティーの設定をオーバーライドします。

JDBC ドライバーの各接続は、指定したディレクトリーの異なるファイルにトレースされます。そのディレクトリー内のファイルの命名規則は、`baseTraceFileName` に `NULL` 以外の値を指定するかどうかに応じて以下のように異なります。

- `baseTraceFileName` に `NULL` 値を指定すると、接続のトレースは `traceFile_global_n` という名前のファイルへ書き込まれます。

`n` は `n` 番目の JDBC ドライバー接続です。

- `baseTraceFileName` に `NULL` 以外の値を指定すると、接続のトレースは `baseTraceFileName_global_n` という名前のファイルへ書き込まれます。

`baseTraceFileName` は `baseTraceFileName` パラメーターの値です。

`n` は `n` 番目の JDBC ドライバー接続です。

#### **baseTraceFileName**

グローバル・トレース情報の書き込み先のファイル名の語幹を指定します。`baseTraceFileName` と `traceDirectory` の組み合わせによって、グローバル・トレース・ログ・ファイルの絶対パス名が決定されます。

#### **traceLevel**

何をトレースするかを指定します。

`traceLevel` パラメーターで、以下のトレースを 1 つ以上指定することができます。

- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_NONE (X'00')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTION_CALLS (X'01')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_STATEMENT_CALLS (X'02')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_RESULT_SET_CALLS (X'04')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRIVER_CONFIGURATION (X'10')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTS (X'20')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRDA_FLOWS (X'40')`

- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_RESULT_SET_META_DATA (X'80')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_PARAMETER_META_DATA (X'100')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DIAGNOSTICS (X'200')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_SQLJ (X'400')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_XA_CALLS (DB2 Database for Linux, UNIX, and Windows 用の IBM Data Server Driver for JDBC and SQLJ Type 2 接続のみ) (X'800')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_META_CALLS (X'2000')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DATASOURCE_CALLS (X'4000')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_LARGE_OBJECT_CALLS (X'8000')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_SYSTEM_MONITOR (X'20000')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_TRACEPOINTS (X'40000')`
- `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL (X'FFFFFFFF')`

複数のトレースを指定するには、以下の技法のいずれかを使用します。

- 2 つ以上のトレース値にビット単位 **OR** (`|`) 演算子を使用する。たとえば、**DRDA** フローおよび接続呼び出しをトレースするには、`traceLevel` に以下の値を指定します。

```
TRACE_DRDA_FLOWS|TRACE_CONNECTION_CALLS
```

- ビット単位の補数 (波形記号 `~`) 演算子とトレース値を使用して、特定のトレース以外のすべてを指定する。たとえば、**DRDA** フロー以外のすべてをトレースする場合、`traceLevel` に以下の値を指定します。

```
~TRACE_DRDA_FLOWS
```

### **getTraceFileAppend**

形式:

```
public abstract boolean getTraceFileAppend()
    throws java.sql.SQLException
```

トレース・コントローラーによって生成されるトレース・レコードがトレース・ファイルに追加される場合、`true` を戻します。それ以外の場合、`getTraceFileAppend` は `false` を戻します。

### **getTraceDirectory**

形式:

```
public void getTraceDirectory()
    throws java.sql.SQLException
```

トレース・コントローラーによって生成されるトレース・レコードの宛先ディレクトリーの名前が設定されている場合、その宛先ディレクトリーの名前を戻します。それ以外の場合、`getTraceDirectory` は `NULL` を戻します。

### **getTraceLevel**

形式:

```
public void getTraceLevel()
    throws java.sql.SQLException
```

トレース・コントローラーによって生成されるトレース・レコードのトレース・レベルが設定されている場合、そのトレース・レベルを戻します。それ以外の場合、`getTraceLevel` は `-1 (TRACE_ALL)` を戻します。

#### **unsetLogWriter**

形式:

```
public abstract void unsetLogWriter()
    throws java.sql.SQLException
```

将来の接続に対してグローバル・ログ書き込みプログラムを無効にします。

#### **suspendTrace**

形式:

```
public void suspendTrace()
    throws java.sql.SQLException
```

現在および将来の接続に対して、`Connection` レベルまたは `DataSource` レベルのグローバル・トレースをすべて中断します。グローバル・ログ書き込みプログラムを有効または無効にするときに、`suspendTrace` を呼び出すことができます。

#### **resumeTrace**

形式:

```
public void resumeTrace()
    throws java.sql.SQLException
```

現在および将来の接続に対して、`Connection` レベルまたは `DataSource` レベルのグローバル・トレースをすべて再開します。グローバル・ログ書き込みプログラムを有効または無効にするときに、`resumeTrace` を呼び出すことができます。グローバル・ログ書き込みプログラムを無効にすると、`resumeTrace` は `Connection` レベルまたは `DataSource` レベルのトレースを再開します。グローバル・ログ書き込みプログラムを有効にすると、`resumeTrace` はグローバル・トレースを再開します。

## **DB2XADataSource クラス**

`DB2XADataSource` は `XADataSource` オブジェクトのファクトリーです。このインターフェースをインプリメントするオブジェクトは、`Java Naming and Directory Interface (JNDI)` に基づくネーム・サービスで登録されます。

`com.ibm.db2.jcc.DB2XADataSource` クラスは `com.ibm.db2.jcc.DB2BaseDataSource` クラスを拡張し、`javax.sql.XADataSource`、`java.io.Serializable`、および `javax.naming.Referenceable` インターフェースをインプリメントします。

## **DB2XADataSource のメソッド**

#### **getDB2TrustedXAConnection**

形式:

```
public Object[] getDB2TrustedXAConnection(String user,
    String password,
    java.util.Properties properties)
    throws java.sql.SQLException
public Object[] getDB2TrustedXAConnection(
    java.util.Properties properties)
    throws java.sql.SQLException
```

```
public Object[] getDB2TrustedXAConnection(  
    org.ietf.jgss.GSSCredential gssCredential,  
    java.util.Properties properties)  
    throws java.sql.SQLException
```

システム許可 ID を使用するアプリケーション・サーバーは、トラステッド接続を確立するためにこのメソッドを使用します。以下のエレメントは Object[] で戻されます。

- 最初のエレメントは、DB2TrustedXAConnection インスタンスです。
- 2 番目のエレメントは、生成された XA 接続インスタンスの固有の cookie です。

最初のフォーム getDB2TrustedXAConnection はユーザー ID とパスワードを提供します。2 番目のフォーム getDB2TrustedXAConnection は、DB2XADataSource オブジェクトのユーザー ID とパスワードを使用します。3 番目のフォーム getDB2TrustedXAConnection は、Kerberos セキュリティーを使用する接続のためのものです。

パラメーターの説明:

**user**

トラステッド接続の確立に使用される許可 ID。

**password**

トラステッド接続の確立に使用される許可 ID のパスワード。

**gssCredential**

データ・ソースが Kerberos セキュリティーを使用する場合に、別のプリンシパルから渡される委任証明書を指定します。

**properties**

接続のプロパティー。

**getDB2TrustedPooledConnection**

形式:

```
public Object[] getDB2TrustedPooledConnection(java.util.Properties properties)  
    throws java.sql.SQLException
```

システム許可 ID を使用するアプリケーション・サーバーは、DB2XADataSource オブジェクトのユーザー ID とパスワードを使用して、トラステッド接続を確立するためにこのメソッドを使用します。以下のエレメントは Object[] で戻されます。

- 最初のエレメントは、トラステッド DB2TrustedPooledConnection インスタンスです。
- 2 番目のエレメントは、生成されたプール接続インスタンスの固有の cookie です。

パラメーターの説明:

**properties**

接続のプロパティー。

**getDB2XAConnection**

形式:

```

public DB2XAConnection getDB2XAConnection(String user,
String password,
java.util.Properties properties)
throws java.sql.SQLException
public DB2XAConnection getDB2XAConnection(
org.ietf.jgss.GSSCredential gssCredential,
java.util.Properties properties)
throws java.sql.SQLException

```

異種プール環境での初期非トラステッド接続を確立します。

最初のフォーム `getDB2PooledConnection` はユーザー ID とパスワードを提供します。2番目のフォーム `getDB2XAConnection` は、Kerberos セキュリティーを使用する接続のためのものです。

パラメーターの説明:

**user**

接続の確立に使用される許可 ID。

**password**

接続の確立に使用される許可 ID のパスワード。

**gssCredential**

データ・ソースが Kerberos セキュリティーを使用する場合に、別のプリンシパルから渡される委任証明書を指定します。

**properties**

接続のプロパティー。

## DB2Xml インターフェース

`com.ibm.db2.jcc.DB2Xml` インターフェースは、DB2 XML データ・タイプで使用するための Java オブジェクトを宣言するために使用されます。

### DB2Xml のメソッド

以下のメソッドは、IBM Data Server Driver for JDBC and SQLJ に対してのみ定義されます。

**closeDB2Xml**

形式:

```

public void closeDB2Xml()
throws SQLException

```

`com.ibm.jcc.DB2Xml` オブジェクトと関連付けられているリソースを解放します。

**getDB2AsciiStream**

形式:

```

public java.io.InputStream getDB2AsciiStream()
throws SQLException

```

データを `DB2Xml` オブジェクトから検索し、そのデータを US-ASCII エンコードに変換します。

**getDB2BinaryStream**

形式:

```
public java.io.InputStream getDB2BinaryStream()  
    throws SQLException
```

データを DB2Xml オブジェクトからバイナリー・ストリームとして検索します。バイナリー・ストリーム内のバイトの文字エンコード方式は、XML 1.0 仕様で定義されています。

#### **getDB2Bytes**

形式:

```
public byte[] getDB2Bytes()  
    throws SQLException
```

データを DB2Xml オブジェクトからバイト配列として検索します。バイトの文字エンコード方式は、XML 1.0 仕様で定義されています。

#### **getDB2CharacterStream**

形式:

```
public java.io.Reader getDB2CharacterStream()  
    throws SQLException
```

データを DB2Xml オブジェクトから java.io.Reader オブジェクトとして検索します。

#### **getDB2String**

形式:

```
public String getDB2String()  
    throws SQLException
```

データを DB2Xml オブジェクトから String 値として検索します。

#### **getDB2XmlAsciiStream**

形式:

```
public InputStream getDB2XmlAsciiStream()  
    throws SQLException
```

データを DB2Xml オブジェクトから検索し、そのデータを US-ASCII エンコードに変換し、戻されるデータに US-ASCII のエンコード仕様で XML 宣言を組み込みます。

#### **getDB2XmlBinaryStream**

形式:

```
public java.io.InputStream getDB2XmlBinaryStream(String targetEncoding)  
    throws SQLException
```

データを DB2Xml オブジェクトからバイナリー・ストリームとして検索し、そのデータを *targetEncoding* に変換し、戻されるデータに *targetEncoding* のエンコード仕様で XML 宣言を組み込みます。

パラメーター:

*targetEncoding*

IANA 文字セット・レジストリーにリストされる有効なエンコード名。

DB2 サーバーでサポートされているエンコード名は、『CCSID とシリアライズされた XML 出力データのエンコーディング名とのマップ』でリストされています。



### getDB2XmlBytes

形式:

```
public byte[] getDB2XmlBytes(String targetEncoding)
    throws SQLException
```

データを DB2Xml オブジェクトからバイト配列として検索し、そのデータを *targetEncoding* に変換し、戻されるデータに *targetEncoding* のエンコード仕様で XML 宣言を組み込みます。

パラメーター:

*targetEncoding*

IANA 文字セット・レジストリーにリストされる有効なエンコード名。

DB2 サーバーでサポートされているエンコード名は、『CCSID とシリアライズされた XML 出力データのエンコーディング名とのマップ』でリストされています。

### getDB2XmlCharacterStream

形式:

```
public java.io.Reader getDB2XmlCharacterStream()
    throws SQLException
```

データを DB2Xml オブジェクトから `java.io.Reader` オブジェクトとして検索し、そのデータを ISO-10646-UCS-2 エンコードに変換し、戻されるデータに ISO-10646-UCS-2 のエンコード仕様で XML 宣言を組み込みます。

### getDB2XmlString

形式:

```
public String getDB2XmlString()
    throws SQLException
```

データを DB2Xml オブジェクトから `String` オブジェクトとして検索し、そのデータを ISO-10646-UCS-2 エンコードに変換し、戻されるデータに ISO-10646-UCS-2 のエンコード仕様で XML 宣言を組み込みます。

### isDB2XmlClosed

形式:

```
public boolean isDB2XmlClosed()
    throws SQLException
```

`com.ibm.jcc.DB2Xml` オブジェクトがクローズされているかどうかを示します。

---

## 現行の IBM Data Server Driver for JDBC and SQLJ と以前の DB2 JDBC ドライバーとの JDBC の相違点

JDBC アプリケーションを古いドライバーから IBM Data Server Driver for JDBC and SQLJ にマイグレーションする前に、それらのドライバー間の相違点を理解する必要があります。

**重要:** Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバー (DB2 JDBC Type 2 ドライバー) は、推奨されていません。この情報は、アプリケーションの IBM Data Server Driver for JDBC and SQLJ への移行を支援するために提供されています。

## サポートされるメソッド

JDBC ドライバーによりサポートされるメソッドの比較については、『JDBC API でのドライバーのサポート』を参照してください。

## JDBC ドライバーによる連続ストリーミングの使用

IBM Data Server Driver for JDBC and SQLJ バージョン 3.50 以降の場合、連続ストリーミングの動作は、DB2 Database for Linux, UNIX, and Windows バージョン 9.5 以降への接続の場合の LOB 検索のデフォルトです。

連続ストリーミングは IBM Data Server Driver for JDBC and SQLJ バージョン 3.1 以降でサポートされています。ただし、IBM Data Server Driver for JDBC and SQLJ バージョン 3.2 以降では、連続ストリーミングの動作は、DB2 for z/OS バージョン 9.1 以降への接続の場合の LOB および XML 検索のデフォルトです。

IBM Data Server Driver for JDBC and SQLJ の以前のバージョンおよび DB2 JDBC Type 2 ドライバーは、連続ストリーミングをサポートしませんでした。

**重要:** 連続ストリーミングでは、LOB または XML の値を `ResultSet` からアプリケーション変数に取り出す場合、カーソルを移動するまで、または `ResultSet` 上のカーソルをクローズするまでは、そのアプリケーション変数の内容を操作できます。その後は、アプリケーション変数の内容は操作できなくなります。アプリケーション変数内の LOB に対して何らかのアクションを実行すると、`SQLException` を受け取ります。例えば、連続ストリーミングが有効で、以下のようなステートメントを実行するとします。

```
...
ResultSet rs = stmt.executeQuery("SELECT CLOBCOL FROM MY_TABLE");
rs.next(); // Retrieve the first row of the ResultSet
Clob clobFromRow1 = rs.getClob(1); // Put the CLOB from the first column of
// the first row in an application variable
String substr1Clob = clobFromRow1.getSubString(1,50);
// Retrieve the first 50 bytes of the CLOB
rs.next(); // Move the cursor to the next row.
// clobFromRow1 is no longer available.
// String substr2Clob = clobFromRow1.getSubString(51,100);
// This statement would yield an SQLException
Clob clobFromRow2 = rs.getClob(1); // Put the CLOB from the first column of
// the second row in an application variable
rs.close(); // Close the ResultSet.
// clobFromRow2 is also no longer available.
```

`rs.next()` を実行してカーソルを `ResultSet` の 2 番目の行に置いた後は、`clobFromRow1` 内の CLOB 値は使用できなくなります。同様に、`rs.close()` を実行して `ResultSet` をクローズした後、`clobFromRow1` および `clobFromRow2` 内の値は使用できなくなります。

この動作の変更によるエラーを避けるには、以下のアクションのいずれかをとる必要があります。

- アプリケーションを変更します。

LOB データをアプリケーション変数に取り出すアプリケーションは、データの取り出しに使用されたカーソルが移動されるか閉じられるまでの間に限り、それらのアプリケーション変数のデータを操作できます。

- 連続ストリーミングを使用不可にします。それには、`progressiveStreaming` プロパティを `DB2BaseDataSource.NO (2)` に設定します。

## IBM Data Server Driver for JDBC and SQLJ バージョン 4.0 以降の `ResultSetMetaData` 値

IBM Data Server Driver for JDBC and SQLJ バージョン 4.0 以降の `ResultSetMetaData.getColumnNames` および `ResultSetMetaData.getColumnLabels` の値は、IBM Data Server Driver for JDBC and SQLJ のそれ以前のバージョンまたはその他の IBM JDBC ドライバーに戻される値とは異なります。

IBM Data Server Driver for JDBC and SQLJ バージョン 4.0 以降を使用する必要があるが、アプリケーションが旧バージョンの JDBC ドライバーに戻された `ResultSetMetaData.getColumnNames` および `ResultSetMetaData.getColumnLabels` の値を戻す必要がある場合、`useJDBC4ColumnNameAndLabelSemantics` `Connection` および `DataSource` プロパティを `DB2BaseDataSource.NO (2)` に設定できます。

## 自動生成キーを使用したバッチ更新により `SQLException` が出される

IBM Data Server Driver for JDBC and SQLJ バージョン 3.50 以降を使用する場合、SQL ステートメントを自動生成キーの取り出しのために準備し、バッチ更新に `PreparedStatement` オブジェクトを使用すると、`SQLException` が出されます。

IBM Data Server Driver for JDBC and SQLJ のバージョン 3.50 より前のバージョンは、自動生成キーを戻すように準備された `PreparedStatement` オブジェクトに対してアプリケーションが `addBatch` または `executeBatch` メソッドを呼び出すときに、`SQLException` をスローしません。ただし、`PreparedStatement` オブジェクトは自動生成キーを戻しません。

## スクロール可能 `ResultSet` および更新可能 `ResultSet` のサポート

IBM Data Server Driver for JDBC and SQLJ は、スクロール可能 `ResultSet` および更新可能 `ResultSet` をサポートします。

DB2 JDBC Type 2 ドライバーはスクロール可能 `ResultSet` をサポートしますが、更新可能 `ResultSet` はサポートしません。

## URL 構文の相違点

`DriverManager.getConnection` メソッドの `url` パラメーターの構文はドライバーごとに異なります。詳細については、以下のトピックを参照してください。

- 『`DriverManager` インターフェースと IBM Data Server Driver for JDBC and SQLJ を使用したデータ・ソースへの接続』
- 『`DriverManager` インターフェースと DB2 JDBC Type 2 ドライバーを使用したデータ・ソースへの接続』

## ドライバー・エラーについて戻されるエラー・コードと `SQLSTATE` の相違点

IBM Data Server Driver for JDBC and SQLJ は、他のドライバーとは異なり、内部エラーには既存の `SQLCODE` または `SQLSTATE` を使用しません。『IBM Data

Server Driver for JDBC and SQLJ から発行されるエラー・コード』と『IBM Data Server Driver for JDBC and SQLJ から発行される SQLSTATE』を参照してください。

内部エラーが発生すると、z/OS 用の JDBC/SQLJ ドライバーは ODBC SQLSTATE を戻します。

## エラー・メッセージ・テキストが戻される程度

IBM Data Server Driver for JDBC and SQLJ では、SQLException.getMessage() を実行する際に、retrieveMessagesFromServerOnGetMessage プロパティーが true に設定されていないならば、フォーマット済みメッセージ・テキストは戻されません。

DB2 JDBC Type 2 ドライバーでは、SQLException.getMessage() を実行すると、フォーマット済みメッセージ・テキストが戻されます。

## セキュリティー・メカニズム

JDBC ドライバーには様々なセキュリティー・メカニズムがあります。

IBM Data Server Driver for JDBC and SQLJ セキュリティー・メカニズムについては、『IBM Data Server Driver for JDBC and SQLJ 使用時のセキュリティー』を参照してください。

DB2 JDBC Type 2 ドライバーのセキュリティー・メカニズムについては、『DB2 JDBC Type 2 ドライバー使用時のセキュリティー』を参照してください。

## 読み取り専用接続のサポート

IBM Data Server Driver for JDBC and SQLJ では、Connection または DataSource オブジェクトの readOnly プロパティーにより、接続を読み取り専用にすることができます。

DB2 JDBC Type 2 ドライバーは、接続を読み取り専用にするかどうかを判断する際に Connection.setReadOnly 値を使用します。ただし、Connection.setReadOnly(true) を設定しても、接続が読み取り専用になるとは限りません。

## BIT DATA 列について ResultSet.getString から戻される結果

IBM Data Server Driver for JDBC and SQLJ は、CHAR FOR BIT DATA または VARCHAR FOR BIT DATA 列について ResultSet.getString 呼び出しからのデータを小文字の 16 進数ストリングとして戻します。

DB2 JDBC Type 2 ドライバーはデータを大文字の 16 進数ストリングとして戻します。

## どの行にも影響しない executeUpdate 呼び出しの結果

executeUpdate 呼び出しがどの行にも影響を与えない場合、IBM Data Server Driver for JDBC and SQLJ は SQLWarning を生成します。

DB2 JDBC Type 2 ドライバーは SQLWarning を生成しません。

## TIMESTAMP 列についての getDate または getTime 呼び出しの結果

getDate または getTime 呼び出しが TIMESTAMP 列に対して行われる場合、IBM Data Server Driver for JDBC and SQLJ は SQLWarning を生成しません。

getDate または getTime 呼び出しが TIMESTAMP 列に対して行われる場合、DB2 JDBC Type 2 ドライバーは SQLWarning を生成します。

## 長さが一致しない PreparedStatement.setXXXStream について例外がスローされる場合

PreparedStatement.setBinaryStream、PreparedStatement.setCharacterStream、または PreparedStatement.setUnicodeStream メソッドを使用する場合、length パラメーター値は入力ストリーム内のバイト数に一致しなければなりません。

バイト数が一致しない場合、後続の PreparedStatement.executeUpdate メソッドが実行するまで、IBM Data Server Driver for JDBC and SQLJ は例外をスローしません。そのため、IBM Data Server Driver for JDBC and SQLJ では、長さが一致しないと、一部のデータがサーバーに送信されることがあります。そのデータはサーバーによって切り捨てられるかまたは埋め込まれます。呼び出し側アプリケーションはロールバック要求を出して、切り捨てられたデータまたは埋め込まれたデータを含むデータベース更新を取り消す必要があります。

DB2 JDBC Type 2 ドライバーは、PreparedStatement.setBinaryStream、PreparedStatement.setCharacterStream、または PreparedStatement.setUnicodeStream メソッドが実行した後、例外をスローします。

## PreparedStatement.setXXXStream のデフォルト・マッピング

IBM Data Server Driver for JDBC and SQLJ では、PreparedStatement.setBinaryStream、PreparedStatement.setCharacterStream、または PreparedStatement.setUnicodeStream メソッドを使用しており、ターゲット列のデータ・タイプに関する情報を使用できない場合、入力データは BLOB または CLOB データ・タイプにマップされます。

DB2 JDBC Type 2 ドライバーの場合、入力データは VARCHAR FOR BIT DATA または VARCHAR データ・タイプにマップされます。

## 文字変換の実行方法

文字データがクライアントとサーバーとの間で転送される場合、データは受信側が処理できる形式に変換される必要があります。

IBM Data Server Driver for JDBC and SQLJ の場合、データ・ソースからクライアントに送信される文字データは Java の組み込み文字コンバーターを使用して変換されます。IBM Data Server Driver for JDBC and SQLJ がサポートする変換は、基礎となる JRE インプリメンテーションによってサポートされるものに制限されません。

タイプ 4 接続を使用する IBM Data Server Driver for JDBC and SQLJ クライアントは、データを Unicode UTF-8 としてデータ・ソースに送信します。

DB2 JDBC Type 2 ドライバーでは、文字変換が DB2 サーバーによってサポートされる場合に変換を実行できます。

これらのドライバーはデータ・ソースからの CCSID 情報を使用します (使用可能な場合)。ドライバーは入力パラメーター・データを送信する前に、そのデータをデータ・ソースの CCSID に変換します。ターゲット CCSID 情報が使用不可の場合、ドライバーはデータを Unicode UTF-8 として送信します。

## 入力パラメーターの暗黙的または明示的データ・タイプ変換

`PreparedStatement.setXXX` メソッドを実行し、`setXXX` メソッドから生成されるデータ・タイプがパラメーター値が割り当てられている表列のデータ・タイプに一致しない場合、データ・タイプの変換が行われなければ、ドライバーはエラーを戻します。

IBM Data Server Driver for JDBC and SQLJ では、ターゲット・データ・タイプが認識されている場合、および `deferPrepares` と `sendDataAsIs` 接続プロパティーが `false` に設定されている場合に、正しい SQL データ・タイプへの変換が暗黙的に行われます。この場合、`setXXX` 呼び出しで暗黙値が明示値をオーバーライドします。`deferPrepares` 接続プロパティーまたは `sendDataAsIs` 接続プロパティーが `true` に設定されている場合、`PreparedStatement.setObject` メソッドを使用してパラメーターを正しい SQL データ・タイプに変換する必要があります。

DB2 JDBC Type 2 ドライバーの場合、パラメーターのデータ・タイプがそのデフォルト SQL データ・タイプに一致しなければ、`PreparedStatement.setObject` メソッドを使用してパラメーターを正しい SQL データ・タイプに変換する必要があります。

## 入力パラメーターの String から BINARY への変換のサポート

$x$  がタイプ `String` のオブジェクトの場合、IBM Data Server Driver for JDBC and SQLJ は以下の形式の `PreparedStatement.setObject` 呼び出しをサポートしません。

```
setObject(parameterIndex, x, java.sql.Types.BINARY)
```

DB2 JDBC Type 2 ドライバーはこのタイプの呼び出しをサポートします。ドライバーは  $x$  の値を 16 進数ストリングと解釈します。

## 10 進スケールが一致しない PreparedStatement.setObject の結果

IBM Data Server Driver for JDBC and SQLJ の場合、10 進入力パラメーターを指定して `PreparedStatement.setObject` を呼び出すときに、入力パラメーターのスケールがターゲット列のスケールより大きければ、ドライバーは入力値の末尾桁を切り捨ててからその値を列に割り当てます。

DB2 JDBC Type 2 ドライバーは入力値の末尾桁を丸めてから、その値を列に割り当てます。

## ResultSet.getBigDecimal スケール・パラメーターの有効範囲

推奨されない形式の `ResultSet.getBigDecimal` には、第 2 のパラメーターとして `scale` パラメーターがあります。IBM Data Server Driver for JDBC and SQLJ では、0 から 32 の範囲のスケール・パラメーターを許可します。



DB2 JDBC Type 2 ドライバーは -1 から 32 の範囲を許可します。

## 入力パラメーター用の `java.lang.Character` データ・タイプからの変換のサポート

次の形式の `PreparedStatement.setObject` の場合、IBM Data Server Driver for JDBC and SQLJ は  $x$  を JDBC データ・タイプに変換する際に、JDBC データ・タイプへの Java オブジェクトの標準データ・タイプ・マッピングをサポートします。

```
setObject(parameterIndex, x)
```

DB2 JDBC Type 2 ドライバーは、`java.lang.Character` から `CHAR` への  $x$  の非標準マッピングをサポートします。

## 文字列に対する `ResultSet.getBinaryStream` のサポート

IBM Data Server Driver for JDBC and SQLJ は、文字列に `FOR BIT DATA` 属性が含まれる場合にのみ、その文字列を表す引数を指定した `ResultSet.getBinaryStream` をサポートします。

DB2 JDBC Type 2 ドライバーの場合、`ResultSet.getBinaryStream` 引数が文字列であれば、列には `FOR BIT DATA` 属性がなくてもかまいません。

## バイナリー列に対する `ResultSet.getBinaryStream` から戻されるデータ

IBM Data Server Driver for JDBC and SQLJ では、`ResultSet.getBinaryStream` をバイナリー列に対して実行する場合、戻されるデータの形式は小文字の 16 進数字の対です。

DB2 JDBC Type 2 ドライバーでは、`ResultSet.getBinaryStream` をバイナリー列に対して実行する場合、戻されるデータの形式は大文字の 16 進数字の対です。

## Boolean 入カタイプおよび `CHAR` ターゲット・タイプの `setObject` を使用した結果

IBM Data Server Driver for JDBC and SQLJ では、`PreparedStatement.setObject` (`parameterIndex,x,CHAR`) を実行しており、 $x$  がブール値の場合、値 "0" または "1" が表列に挿入されます。

DB2 JDBC Type 2 ドライバーでは、ストリング "false" または "true" が表列に挿入されます。表列の長さは少なくとも 5 でなければなりません。

## `getBoolean` を使用して `CHAR` 列から値を取り出した結果

IBM Data Server Driver for JDBC and SQLJ では、`ResultSet.getBoolean` または `CallableStatement.getBoolean` を実行して `CHAR` 列からブール値を取り出すときに、列に値 "false" または "0" が含まれる場合、値 `false` が戻されます。列にそれ以外の値が含まれる場合、`true` が戻されます。

DB2 JDBC Type 2 ドライバーでは、`ResultSet.getBoolean` または `CallableStatement.getBoolean` を実行して `CHAR` 列からブール値を取り出すときに、列に値 "true" または "1" が含まれる場合、値 `true` が戻されます。列にそれ以外の



値が含まれる場合、false が戻されます。

## 閉じられたカーソルに対して ResultSet.next() を実行した結果

IBM Data Server Driver for JDBC and SQLJ では、閉じられたカーソルに対して ResultSet.next() を実行すると、SQLException がスローされます。これは JDBC 標準に準拠します。

DB2 JDBC Type 2 ドライバーでは、閉じられたカーソルに対して ResultSet.next() を実行すると、false の値が戻され、例外はスローされません。

## DatabaseMetaData 呼び出しで NULL 引数を指定した結果

IBM Data Server Driver for JDBC and SQLJ では、JDBC 仕様で null が許可されている場合のみ、DatabaseMetaData メソッド呼び出しで引数に null を指定できます。そうでない場合、例外がスローされます。

DB2 JDBC Type 2 ドライバーでは、null は、検索を絞り込むために引数を使用しないことを意味します。

## メソッド引数の大文字への変換

IBM Data Server Driver for JDBC and SQLJ は、メソッド呼び出し内のどの引数も大文字に変換しません。

DB2 JDBC Type 2 ドライバーは、Statement.setCursorName 呼び出しの引数を大文字に変換します。カーソル名が大文字に変換されないようにするには、カーソル名の前と後に文字 "¥" を付けます。以下に例を示します。

```
Statement.setCursorName("¥"mycursor¥");
```

## タイム・スタンプのエスケープ節のサポート

IBM Data Server Driver for JDBC and SQLJ は、TIME のエスケープ節として、以下の標準形式をサポートします。

```
{t 'hh:mm:ss'}
```

標準形式に加えて、DB2 JDBC Type 2 ドライバーは次の形式の TIME エスケープ節をサポートします。

```
{ts 'hh:mm:ss'}
```

## ステートメント・バッチへの CALL ステートメントの組み込み

IBM Data Server Driver for JDBC and SQLJ は、ステートメント・バッチ内の CALL ステートメントをサポートします。

DB2 JDBC Type 2 ドライバーは、ステートメント・バッチ内の CALL ステートメントをサポートしません。

## SQL ステートメント・テキストからの余分な文字の除去

IBM Data Server Driver for JDBC and SQLJ は SQL ステートメント・テキストをデータ・ソースに渡す前に、空白文字 (スペース、タブ、改行文字など) をそのテキストから除去しません。

DB2 JDBC Type 2 ドライバーは、SQL ステートメント・テキストをデータ・ソースに渡す前に、空白文字をそのテキストから除去します。

## PreparedStatement.executeBatch を実行した結果

PreparedStatement.executeBatch ステートメントが IBM Data Server Driver for JDBC and SQLJ を使用して実行されると、ドライバーは更新カウン트의 int 配列を戻します。配列の各エレメントには、バッチ内のステートメントによって更新された行数が含まれます。

PreparedStatement.executeBatch ステートメントが DB2 JDBC Type 2 ドライバーを使用して実行されると、ドライバーは更新カウンートを判別できないため、更新カウンごとに -3 を戻します。

## コンパウンド SQL のサポート

IBM Data Server Driver for JDBC and SQLJ ドライバーはコンパウンド SQL ブロックをサポートしません。

コンパウンド SQL は複数の SQL ステートメントを単一の実行可能ブロックにグループ化できるようにします。以下に例を示します。

```
EXEC SQL BEGIN COMPOUND ATOMIC STATIC
  UPDATE ACCOUNTS SET ABALANCE = ABALANCE + :delta
  WHERE AID = :aid;
  UPDATE TELLERS SET TBALANCE = TBALANCE + :delta
  WHERE TID = :tid;
  INSERT INTO TELLERS (TID, BID, TBALANCE) VALUES (:i, :branch_id, 0);
  COMMIT;
END COMPOUND;
```

DB2 JDBC Type 2 ドライバーは、PreparedStatement.executeUpdate または Statement.executeUpdate を使用してコンパウンド SQL ブロックの実行をサポートします。

## バッチ更新でパラメーターを設定しない結果

パラメーターが設定されていない場合、IBM Data Server Driver for JDBC and SQLJ ドライバーは PreparedStatement.addBatch 呼び出しの後で例外をスローします。

パラメーターがバッチ内のいずれのステートメントにも設定されていない場合、DB2 JDBC Type 2 ドライバーは PreparedStatement.executeBatch 呼び出しの後で例外をスローします。

## アンカタログ・ストアード・プロシージャを呼び出す機能

IBM Data Server Driver for JDBC and SQLJ ドライバーでは、DB2 カタログで定義されていないストアード・プロシージャを呼び出すことはできません。

DB2 JDBC Type 2 ドライバーでは、DB2 カタログで定義されていないストアド・プロシージャを呼び出すことができます。

## ストアド・プロシージャ・パラメーターのデータ・タイプの指定

IBM Data Server Driver for JDBC and SQLJ ドライバーでは、データ・ソースが CALL ステートメントの動的実行をサポートしていない場合、ストアド・プロシージャ定義で指定されたとおりに正確に CALL ステートメント・パラメーターを指定する必要があります。

例えば、DB2 for z/OS データ・ソースは CALL ステートメントの動的実行をサポートしません。DB2 for z/OS サーバー上のストアド・プロシージャの最初のパラメーターが、CREATE PROCEDURE ステートメントで次のように定義されているとします。

```
OUT PARM1 DECIMAL(3,0)
```

呼び出し側アプリケーションでは、`cs.registerOutParameter(1, Types.DECIMAL)` のようなステートメントは正しくありません。データ・タイプ `cs.registerOutParameter(1, Types.DECIMAL, 0)` だけでなくスケールを指定する形式の `registerOutParameter` メソッドを使用する必要があります。

DB2 JDBC Type 2 ドライバーでは、呼び出しアプリケーションのパラメーター・データ・タイプが CREATE PROCEDURE ステートメントのデータ・タイプと一致している必要はありません。

## ResultSetMetaData.getColumnNames および ResultSetMetaData.getColumnLabels の値の例

IBM Data Server Driver for JDBC and SQLJ バージョン 4.0 以降の場合、`ResultSetMetaData.getColumnNames` および `ResultSetMetaData.getColumnLabels` に戻される値は、それ以前の JDBC ドライバーに戻される値とは異なります。

次の例は、IBM Data Server Driver for JDBC and SQLJ バージョン 4.0、および `useJDBC4ColumnNameAndLabelSemantics` プロパティーが設定されていない場合に以前の JDBC ドライバーに戻される値を示しています。

すべての照会は、次のように定義されている表を使用します。

```
CREATE TABLE MYTABLE(INTCOL INT)
```

**例:** 次の照会には AS 節が含まれます。これは、結果セット内の列のラベルを定義するものです。

```
SELECT MYCOL AS MYLABEL FROM MYTABLE
```

次の表は、照会について戻される `ResultSetMetaData.getColumnNames` および `ResultSetMetaData.getColumnLabels` の値をリストしています。

表 89. AS 節を含む照会の、IBM Data Server Driver for JDBC and SQLJ バージョン 4.0 より前および後の `ResultSetMetaData.getColumnNames` および `ResultSetMetaData.getColumnLabels`

ターゲットのデータ・ソース	IBM Data Server Driver for JDBC and SQLJ バージョン 4.0 より前の動作		IBM Data Server Driver for JDBC and SQLJ バージョン 4.0 以降の動作	
	<code>getColumnNames</code> 値	<code>getColumnLabels</code> 値	<code>getColumnNames</code> 値	<code>getColumnLabels</code> 値
DB2 Database for Linux, UNIX, and Windows	MYLABEL	MYLABEL	MYCOL	MYLABEL
IBM Informix Dynamic Server	MYLABEL	MYLABEL	MYCOL	MYLABEL
DB2 for z/OS バージョン 8 以降、および DB2 for i5/OS バージョン V5R3 以降	MYLABEL	MYLABEL	MYCOL	MYLABEL
DB2 for z/OS バージョン 7、および DB2 for i5/OS バージョン V5R2	MYLABEL	MYLABEL	MYLABEL	MYLABEL

**例:** 次の照会には AS 節が含まれていません。

```
SELECT MYCOL FROM MYTABLE
```

照会に対する `ResultSetMetaData.getColumnNames` および `ResultSetMetaData.getColumnLabels` メソッドは、ターゲット・データ・ソースに関係なく MYCOL を戻します。

**例:** DB2 for z/OS または DB2 for i5/OS データ・ソースでは、列のラベルを定義するために次のように LABEL ON ステートメントが使用されます。

```
LABEL ON COLUMN MYTABLE.MYCOL IS 'LABELONCOL'
```

次の照会には AS 節が含まれます。これは、ResultSet 内の列のラベルを定義するものです。

```
SELECT MYCOL AS MYLABEL FROM MYTABLE
```

次の表は、照会について戻される `ResultSetMetaData.getColumnNames` および `ResultSetMetaData.getColumnLabels` の値をリストしています。

表 90. AS 節を含む照会で LABEL ON ステートメントを指定した表列の、IBM Data Server Driver for JDBC and SQLJ バージョン 4.0 より前および後の `ResultSetMetaData.getColumnNames` および `ResultSetMetaData.getColumnLabels`

ターゲットのデータ・ソース	IBM Data Server Driver for JDBC and SQLJ バージョン 4.0 より前の動作		IBM Data Server Driver for JDBC and SQLJ バージョン 4.0 以降の動作	
	<code>getColumnNames</code> 値	<code>getColumnLabels</code> 値	<code>getColumnNames</code> 値	<code>getColumnLabels</code> 値
DB2 for z/OS バージョン 8 以降、および DB2 for i5/OS V5R3 以降	MYLABEL	LABELONCOL	MYCOL	MYLABEL

表 90. AS 節を含む照会で LABEL ON ステートメントを指定した表列の、 IBM Data Server Driver for JDBC and SQLJ バージョン 4.0 より前および後の ResultSetMetaData.getColumnNames および ResultSetMetaData.getColumnLabels (続き)

ターゲットのデータ・ソース	IBM Data Server Driver for JDBC and SQLJ バージョン 4.0 より前の動作		IBM Data Server Driver for JDBC and SQLJ バージョン 4.0 以降の動作	
	getColumnNames 値	getColumnLabels 値	getColumnNames 値	getColumnLabels 値
DB2 for z/OS バージョン 7、および DB2 for i5/OS V5R2	MYLABEL	LABELONCOL	MYCOL	LABELONCOL

例: DB2 for z/OS または DB2 for i5/OS データ・ソースでは、列のラベルを定義するために次のように LABEL ON ステートメントが使用されます。

```
LABEL ON COLUMN MYTABLE.MYCOL IS 'LABELONCOL'
```

次の照会には AS 節が含まれていません。

```
SELECT MYCOL FROM MYTABLE
```

次の表は、照会について戻される ResultSetMetaData.getColumnNames および ResultSetMetaData.getColumnLabels の値をリストしています。

表 91. AS 節を含まない照会で LABEL ON ステートメントを指定した表列の、 IBM Data Server Driver for JDBC and SQLJ バージョン 4.0 より前および後の ResultSetMetaData.getColumnNames および ResultSetMetaData.getColumnLabels

ターゲットのデータ・ソース	IBM Data Server Driver for JDBC and SQLJ バージョン 4.0 より前の動作		IBM Data Server Driver for JDBC and SQLJ バージョン 4.0 の動作	
	getColumnNames 値	getColumnLabels 値	getColumnNames 値	getColumnLabels 値
DB2 for z/OS バージョン 8 以降、および DB2 for i5/OS V5R3 以降	MYCOL	LABELONCOL	MYCOL	MYCOL
DB2 for z/OS バージョン 7、および DB2 for i5/OS V5R2	MYCOL	LABELONCOL	MYLABEL	LABELONCOL

## IBM Data Server Driver for JDBC and SQLJ と他の DB2 JDBC ドライバーとの SQLJ の相違点

IBM Data Server Driver for JDBC and SQLJ とそれより前の JDBC ドライバーにはいくつかの相違点があります。 IBM Data Server Driver for JDBC and SQLJ に移行する場合、これらの相違点を考慮するように SQLJ プログラムを変更する必要があります。

**重要:** Linux、UNIX、および Windows 用の DB2 JDBC Type 2 ドライバー (DB2 JDBC Type 2 ドライバー) は、推奨されていません。この情報は、アプリケーションの IBM Data Server Driver for JDBC and SQLJ への移行を支援するために提供されています。

IBM Data Server Driver for JDBC and SQLJ での SQLJ サポートは、以下の分野でその他の DB2 JDBC ドライバーでの SQLJ サポートと異なります。

## db2sqljcustomize エラーおよび -collection パラメーター

IBM Data Server Driver for JDBC and SQLJ の一部である db2sqljcustomize ユーティリティには -collection パラメーターがあります。DB2 JDBC Type 2 ドライバーの一部である db2profrc ユーティリティには -collection パラメーターがありません。db2sqljcustomize ユーティリティが DB2 for z/OS サーバー上でバインド操作を実行する場合、-collection パラメーターに小文字が含まれていれば、db2sqljcustomize は -4499 エラーを戻します。DB2 for z/OS ではコレクション ID に小文字を含めることはできないからです。この状況は、db2profrc では発生しません。

## シリアライズド・プロファイルでの相違点

DB2 JDBC Type 2 ドライバーと IBM Data Server Driver for JDBC and SQLJ は、SQLJ 変換プログラムおよび SQLJ カスタマイザー・ユーティリティを実行する際に、異なるバイナリー・コードを生成します。そのため、DB2 JDBC Type 2 ドライバー sqlj および db2profrc ユーティリティを使用して変換およびカスタマイズされた SQLJ アプリケーションは、IBM Data Server Driver for JDBC and SQLJ の使用時には実行しません。**IBM Data Server Driver for JDBC and SQLJ でそれらの SQLJ アプリケーションを実行するには、その前に IBM Data Server Driver for JDBC and SQLJ sqlj および db2sqljcustomize ユーティリティを使用してアプリケーションを再変換および再カスタマイズする必要があります。**これは、アプリケーションを変更していない場合でも行わなければなりません。

## SQL VALUES サポート

DB2 JDBC Type 2 ドライバーは SQLJ ステートメント節で SQL VALUES ステートメントをサポートしますが、IBM Data Server Driver for JDBC and SQLJ はサポートしません。そのため、VALUES ステートメントを含む SQLJ アプリケーションを変更する必要があります。

**例:** SQLJ プログラムに以下のステートメントが含まれるとします。

```
#sql [ctxt] hv = {VALUES (MY_ROUTINE(1))};
```

IBM Data Server Driver for JDBC and SQLJ の場合、そのステートメントを次のように変更する必要があります。

```
#sql [ctxt] {SELECT MY_ROUTINE(1) INTO :hv FROM SYSIBM.SYSDUMMY1};
```

## コンパウンド SQL ステートメント・サポート

DB2 JDBC Type 2 ドライバーは SQLJ ステートメント節でコンパウンド SQL ステートメントをサポートしますが、IBM Data Server Driver for JDBC and SQLJ はサポートしません。そのため、BEGIN COMPOUND および END COMPOUND を指定した SQLJ ステートメントを含む SQLJ アプリケーションを変更する必要があります。コンパウンド・ステートメントを使用してバッチ更新を行う場合、代わりに SQLJ バッチ更新プログラミング・インターフェースを使用できます。



## 接続手法の相違点

使用可能な接続手法およびそれらの接続手法に使用されるドライバー名と URL は、ドライバーによって異なります。詳しくは、『SQLJ を使用したデータ・ソースへの接続』を参照してください。

## スクロール可能イテレーターおよび更新可能イテレーターのサポート

IBM Data Server Driver for JDBC and SQLJ での SQLJ はスクロール可能イテレーターおよび更新可能イテレーターをサポートします。

Linux、UNIX、Windows 用の DB2 JDBC Type 2 ドライバー (DB2 JDBC Type 2 ドライバー) は、両方向スクロール・カーソルをサポートしますが、更新可能イテレーターはサポートしません。

## WebSphere Application Server での SQL ステートメントの動的実行

WebSphere Application Server バージョン 5.0.1 以上では、SQLJ プログラムをカスタマイズすれば、SQL ステートメントは静的に実行されます。

## db2sqljcustomize および db2sqljprint の代替名はサポートされない

DB2 JDBC Type 2 ドライバーは、元々は、SQLJ プロファイル・カスタマイザー・コマンドでは db2profrc という名前を、SQLJ プロファイル・プリンター・コマンドでは db2profp という名前を使用していました。IBM Data Server Driver for JDBC and SQLJ では、SQLJ プロファイル・カスタマイザー・コマンドの名前は db2sqljcustomize で、SQLJ プロファイル・プリンター・コマンドの名前は db2sqljprint です。DB2 Database for Linux, UNIX, and Windows の前のリリースでは、db2profrc は db2sqljcustomize の代替名として受け入れられ、db2profp は db2sqljprint の代替名として受け入れられていました。これらの代替名は受け入れられなくなりました。

---

## IBM Data Server Driver for JDBC and SQLJ に影響を与える SDK for Java の相違点

SDK for Java のバージョン間での動作の相違点は、IBM Data Server Driver for JDBC and SQLJ の下でプログラムを実行するときに受け取る結果のばらつきの原因となる場合があります。

### DBCS 置換文字の取り出された値

DBCS 置換文字 (コード・ページ Cp943 では X'FCFC' など) をデータベース表から取り出す場合、取り出される値は、IBM SDK for Java を使用するかまたは Sun SDK for Java を使用するかによって異なります。

Sun SDK for Java の場合、置換文字は U+0000 として取り出されます。IBM SDK for Java の場合、置換文字は X'FFFD' として取り出されます。



## サポートされるコード・ページ

IBM SDK for Java は Sun SDK for Java よりも多くの DBCS コード・ページをサポートします。そのため、Sun SDK for Java でのサポートされないコード・ページのためにエラーが発生する場合は、IBM SDK for Java を使用してください。

## 暗号化のための IBM SDK for Java 要件

IBM SDK for Java は 256 ビット暗号化をサポートしますが、Sun SDK for Java にはこのサポートがありません。そのため、暗号化を含む IBM Data Server Driver for JDBC and SQLJ セキュリティー・メカニズムのいずれかを使用する場合は、IBM SDK for Java を使用する必要があります。

## システム・モニターのサポート

IBM Data Server Driver for JDBC and SQLJ でのシステム・モニターのサポートには、コア・ドライバー時間とネットワーク I/O 時間のコレクションが含まれます。この情報を取り出すには、SDK for Java バージョン 5 またはそれ以降に含まれている機能が必要です。ただし、IBM SDK for Java バージョン 1.4.2 にも、コア・ドライバー時間とネットワーク I/O 時間のコレクションを使用可能にするサポートがあります。IBM SDK for Java バージョン 1.4.2 を使用する場合、コア・ドライバー時間とネットワーク I/O 時間はその値に最も近いマイクロ秒に丸められます。SDK for Java バージョン 5 またはそれ以降を使用する場合、コア・ドライバー時間とネットワーク I/O 時間はその値に最も近いナノ秒に丸められます。

---

## IBM Data Server Driver for JDBC and SQLJ から発行されるエラー・コード

+4200 から +4299、+4450 から +4499、-4200 から -4299、および -4450 から -4499 の範囲のエラー・コードは、IBM Data Server Driver for JDBC and SQLJ 用に予約されています。

IBM Data Server Driver for JDBC and SQLJ エラーが発生した後で `SQLException.getMessage` メソッドを呼び出すと、以下のものを含むストリングが戻されます。

- 接続がタイプ 2 であるかまたはタイプ 4 接続であるか
- IBM ソフトウェア・サポートの診断情報
- ドライバーのレベル
- 説明のためのメッセージ
- エラー・コード
- SQLSTATE

以下に例を示します。

```
[jcc][t4][20128][12071][3.50.54] Invalid queryBlockSize specified: 1,048,576,012.  
Using default query block size of 32,767.  ERRORCODE=0, SQLSTATE=
```

現在、IBM Data Server Driver for JDBC and SQLJ は以下のエラー・コードを発行します。

表 92. IBM Data Server Driver for JDBC and SQLJ により発行されるエラー・コード

エラー・コード	メッセージ・テキストおよび説明	SQLSTATE
+4204	RETURN DATA UNTIL 節に指定されているとおり、エラーが検出されて許容されました。  <b>説明:</b> 許容されたエラーには、フェデレーテッド接続、認証、および許可エラーが含まれます。この警告は、DB2 Database for Linux, UNIX, and Windows サーバーへの接続にのみ適用されます。これは、カーソル操作 (ResultSet.next または ResultSet.previous 呼び出しなど) で false が戻される場合にのみ発行されます。	02506
+4222	<i>text-from-getMessage</i>  <b>説明:</b> データ・ソースへの接続中に警告状態が生じました。  <b>ユーザーの対応:</b> SQLException.getMessage を呼び出して、この問題に関する具体的な情報を取得します。	
+4223	<i>text-from-getMessage</i>  <b>説明:</b> 初期化中に警告状態が生じました。  <b>ユーザーの対応:</b> SQLException.getMessage を呼び出して、この問題に関する具体的な情報を取得します。	
+4225	<i>text-from-getMessage</i>  <b>説明:</b> データがサーバーへ送信されたか、サーバーから受信されたときに警告状態が生じました。  <b>ユーザーの対応:</b> SQLException.getMessage を呼び出して、この問題に関する具体的な情報を取得します。	
+4226	<i>text-from-getMessage</i>  <b>説明:</b> SQLJ アプリケーションのカスタマイズまたはバインド中に警告状態が生じました。  <b>ユーザーの対応:</b> SQLException.getMessage を呼び出して、この問題に関する具体的な情報を取得します。	
+4228	<i>text-from-getMessage</i>  <b>説明:</b> 他のカテゴリーに該当しない警告状態が生じました。  <b>ユーザーの対応:</b> SQLException.getMessage を呼び出して、この問題に関する具体的な情報を取得します。	
+4450	機能がサポートされていません: <i>feature-name</i>	
+4460	<i>text-from-getMessage</i>  <b>説明:</b> 指定値は有効なオプションではありません。  <b>ユーザーの対応:</b> SQLException.getMessage を呼び出して、この問題に関する具体的な情報を取得します。	

表 92. IBM Data Server Driver for JDBC and SQLJ により発行されるエラー・コード (続き)

エラー・コード	メッセージ・テキストおよび説明	SQLSTATE
+4461	<i>text-from-getMessage</i>  <b>説明:</b> 指定値は範囲外で無効です。  <b>ユーザーの対応:</b> SQLException.getMessage を呼び出して、この問題に関する具体的な情報を取得します。	
+4462	<i>text-from-getMessage</i>  <b>説明:</b> 必須値が欠落しています。  <b>ユーザーの対応:</b> SQLException.getMessage を呼び出して、この問題に関する具体的な情報を取得します。	
+4470	<i>text-from-getMessage</i>  <b>説明:</b> ターゲット・リソースが閉じられているため、要求された操作を実行できません。  <b>ユーザーの対応:</b> SQLException.getMessage を呼び出して、この問題に関する具体的な情報を取得します。	
+4471	<i>text-from-getMessage</i>  <b>説明:</b> ターゲット・リソースが使用中であるため、要求された操作を実行できません。  <b>ユーザーの対応:</b> SQLException.getMessage を呼び出して、この問題に関する具体的な情報を取得します。	
+4472	<i>text-from-getMessage</i>  <b>説明:</b> ターゲット・リソースが使用できないため、要求された操作を実行できません。  <b>ユーザーの対応:</b> SQLException.getMessage を呼び出して、この問題に関する具体的な情報を取得します。	
+4474	<i>text-from-getMessage</i>  <b>説明:</b> ターゲット・リソースが変更不可であるため、要求された操作を実行できません。  <b>ユーザーの対応:</b> SQLException.getMessage を呼び出して、この問題に関する具体的な情報を取得します。	
-4200	操作が無効です: Global Transaction 中に XA 環境で無効な COMMIT または ROLLBACK が呼び出されました。  <b>説明:</b> XA 環境のグローバル・トランザクション中のアプリケーションがコミットまたはロールバックを発行しました。グローバル・トランザクション中のコミットまたはロールバック操作は無効です。	2D521

表 92. IBM Data Server Driver for JDBC and SQLJ により発行されるエラー・コード (続き)

エラー・コード	メッセージ・テキストおよび説明	SQLSTATE
-4201	操作が無効です: Global Transaction 中は setAutoCommit(true) が許可されていません。  <b>説明:</b> XA 環境のグローバル・トランザクション中のアプリケーションが setAutoCommit(true) ステートメントを実行しました。グローバル・トランザクションで setAutoCommit(true) を発行することは無効です。	2D521
-4203	<i>function</i> の実行エラー。サーバーは <i>rc</i> を戻しました。  <b>説明:</b> SQL ステートメントの実行中に XA 接続でエラーが発生しました。  ネットワークの最適化のために、IBM Data Server Driver for JDBC and SQLJ は次の SQL ステートメントが実行されるまで一部の XA フローを遅らせます。遅延した XA フローでエラーが発生する場合、そのエラーは現行の SQL ステートメントによってスローされる SQLException の一部として報告されます。	
-4210	プールからトランスポート・オブジェクトを取得中にタイムアウトになりました。	
-4211	プールからオブジェクトを取得中にタイムアウトになりました。	
-4212	Sysplex member unavailable.	
-4214	<i>text-from-getMessage</i>  <b>説明:</b> 許可に失敗しました。  <b>ユーザーの対応:</b> SQLException.getMessage を呼び出して、この問題に関する具体的な情報を取得します。	28000
-4220	<i>text-from-getMessage</i>  <b>説明:</b> 文字変換中にエラーが発生しました。  <b>ユーザーの対応:</b> SQLException.getMessage を呼び出して、この問題に関する具体的な情報を取得します。	
-4221	<i>text-from-getMessage</i>  <b>説明:</b> 暗号化または暗号化解除中にエラーが発生しました。  <b>ユーザーの対応:</b> SQLException.getMessage を呼び出して、この問題に関する具体的な情報を取得します。	
-4222	<i>text-from-getMessage</i>  <b>説明:</b> データ・ソースへの接続中にエラーが発生しました。  <b>ユーザーの対応:</b> SQLException.getMessage を呼び出して、この問題に関する具体的な情報を取得します。	

表 92. IBM Data Server Driver for JDBC and SQLJ により発行されるエラー・コード (続き)

エラー・コード	メッセージ・テキストおよび説明	SQLSTATE
-4223	<i>text-from-getMessage</i>  <b>説明:</b> 初期化中にエラーが発生しました。  <b>ユーザーの対応:</b> SQLException.getMessage を呼び出して、この問題に関する具体的な情報を取得します。	
-4224	<i>text-from-getMessage</i>  <b>説明:</b> リソースのクリーンアップ中にエラーが発生しました。  <b>ユーザーの対応:</b> SQLException.getMessage を呼び出して、この問題に関する具体的な情報を取得します。	
-4225	<i>text-from-getMessage</i>  <b>説明:</b> データがサーバーへ送信されたか、サーバーから受信されたときにエラーが発生しました。  <b>ユーザーの対応:</b> SQLException.getMessage を呼び出して、この問題に関する具体的な情報を取得します。	
-4226	<i>text-from-getMessage</i>  <b>説明:</b> SQLJ アプリケーションのカスタマイズまたはバインド中にエラーが発生しました。  <b>ユーザーの対応:</b> SQLException.getMessage を呼び出して、この問題に関する具体的な情報を取得します。	
-4227	<i>text-from-getMessage</i>  <b>説明:</b> リセット中にエラーが発生しました。  <b>ユーザーの対応:</b> SQLException.getMessage を呼び出して、この問題に関する具体的な情報を取得します。	
-4228	<i>text-from-getMessage</i>  <b>説明:</b> 他のカテゴリに該当しないエラーが発生しました。  <b>ユーザーの対応:</b> SQLException.getMessage を呼び出して、この問題に関する具体的な情報を取得します。	
-4296	例外 <i>exception-class</i> : サーバー <i>server</i> ポート <i>port</i> メッセージ <i>message</i> でのソケットのオープン・エラー。  <b>説明:</b> ドライバーが接続を確立できませんでした。  <b>ユーザーの対応:</b> サーバーのホスト名またはアドレスおよびポート番号が正しいこと、またサーバーが開始されていることを確認します。DB2 Database for Linux, UNIX, and Windows データ・ソースへの接続の場合、svcname 構成パラメーターが正しく構成されていることを確認します。	08001

表 92. IBM Data Server Driver for JDBC and SQLJ により発行されるエラー・コード (続き)

エラー・コード	メッセージ・テキストおよび説明	SQLSTATE
-4297	接続の基礎となるソケット、ソケット入力ストリーム、またはソケット出力ストリーム上での操作中に、通信エラーが発生しました。エラーの場所: <i>function</i> 。メッセージ: <i>message</i> 。	08001
-4298	データベース・マネージャーは新規要求を受け入れることができません。予期しないエラー状態がターゲット・システムで検出されたため、進行中のすべての要求を終了したか、または、この特定の要求を終了しました。	58009
-4450	機能がサポートされていません: <i>feature-name</i>	
-4460	<i>text-from-getMessage</i>  <b>説明:</b> 指定値は有効なオプションではありません。  <b>ユーザーの対応:</b> <code>SQLException.getMessage</code> を呼び出して、この問題に関する具体的な情報を取得します。	
-4461	<i>text-from-getMessage</i>  <b>説明:</b> 指定値は範囲外で無効です。  <b>ユーザーの対応:</b> <code>SQLException.getMessage</code> を呼び出して、この問題に関する具体的な情報を取得します。	42815
-4462	<i>text-from-getMessage</i>  <b>説明:</b> 必須値が欠落しています。  <b>ユーザーの対応:</b> <code>SQLException.getMessage</code> を呼び出して、この問題に関する具体的な情報を取得します。	
-4463	<i>text-from-getMessage</i>  <b>説明:</b> 指定された値に構文エラーがあります。  <b>ユーザーの対応:</b> <code>SQLException.getMessage</code> を呼び出して、この問題に関する具体的な情報を取得します。	42601
-4470	<i>text-from-getMessage</i>  <b>説明:</b> ターゲット・リソースが閉じられているため、要求された操作を実行できません。  <b>ユーザーの対応:</b> <code>SQLException.getMessage</code> を呼び出して、この問題に関する具体的な情報を取得します。	
-4471	<i>text-from-getMessage</i>  <b>説明:</b> ターゲット・リソースが使用中であるため、要求された操作を実行できません。  <b>ユーザーの対応:</b> <code>SQLException.getMessage</code> を呼び出して、この問題に関する具体的な情報を取得します。	

表 92. IBM Data Server Driver for JDBC and SQLJ により発行されるエラー・コード (続き)

エラー・コード	メッセージ・テキストおよび説明	SQLSTATE
-4472	<i>text-from-getMessage</i>  <b>説明:</b> ターゲット・リソースが使用不可であるため、要求された操作を実行できません。  <b>ユーザーの対応:</b> SQLException.getMessage を呼び出して、この問題に関する具体的な情報を取得します。	
-4473	<i>text-from-getMessage</i>  <b>説明:</b> ターゲット・リソースが使用できなくなったため、要求された操作を実行できません。  <b>ユーザーの対応:</b> SQLException.getMessage を呼び出して、この問題に関する具体的な情報を取得します。	
-4474	<i>text-from-getMessage</i>  <b>説明:</b> ターゲット・リソースが変更不可であるため、要求された操作を実行できません。  <b>ユーザーの対応:</b> SQLException.getMessage を呼び出して、この問題に関する具体的な情報を取得します。	
-4475	<i>text-from-getMessage</i>  <b>説明:</b> ターゲット・リソースへのアクセスが制限されているため、要求された操作を実行できません。  <b>ユーザーの対応:</b> SQLException.getMessage を呼び出して、この問題に関する具体的な情報を取得します。	
-4476	<i>text-from-getMessage</i>  <b>説明:</b> 要求された操作はターゲット・リソースに対しては許可されていないため、この操作を実行できません。  <b>ユーザーの対応:</b> SQLException.getMessage を呼び出して、この問題に関する具体的な情報を取得します。	
-4496	保留カーソルの SQL OPEN が XA 接続に対して発行されました。JDBC ドライバーでは、XA 接続の場合にデータベース・サーバー上で保留カーソルをオープンすることはできません。	
-4497	アプリケーションはロールバックを発行する必要があります。作業単位は既に DB2 サーバーでロールバックされましたが、その作業単位に關与する他のリソース・マネージャーが、変更内容をまだロールバックしていない可能性があります。このアプリケーションの保全性を確保するために、アプリケーションがロールバックを発行するまで、すべての SQL 要求が拒否されます。	



表 92. IBM Data Server Driver for JDBC and SQLJ により発行されるエラー・コード (続き)

エラー・コード	メッセージ・テキストおよび説明	SQLSTATE
-4498	<p>接続は失敗しましたが、再確立されました。ホスト名または IP アドレス: <i>host-name</i>、サービス名またはポート番号: <i>port</i>、特殊レジスター変更インディケータ: <i>rc</i>。</p> <p><b>説明:</b> <i>host-name</i> および <i>port</i> は、接続が再確立されたデータ・ソースを示します。 <i>rc</i> は、特殊レジスター値を設定した SQL ステートメントが再実行されたかどうかを以下のように示します。</p> <ol style="list-style-type: none"> <li>1 特殊レジスター値を設定した SQL ステートメントは再実行された。</li> <li>2 特殊レジスター値を設定した SQL ステートメントは再実行されなかった可能性がある。</li> </ol> <p>DB2 for z/OS サーバーに対するクライアント・リルトの場合、最後のコミット・ポイント後に設定された特殊レジスター値は再確立されません。</p> <p>アプリケーションは以前のコミット・ポイントにロールバックされます。</p>	
-4499	<p>致命的エラーが発生したため、切断されました。既存の接続が使用不能になりました。</p> <p><b>説明:</b> 考えられる原因の 1 つは、ネットワーク・エラーが原因でソケットが切断したことです。</p>	
-30108	Sysplex においてクライアント・リルト例外が発生しました。	08506
-99999	IBM Data Server Driver for JDBC and SQLJ がまだエラー・コードのないエラーを出しました。	

## IBM Data Server Driver for JDBC and SQLJ から発行される SQLSTATE

46600 から 466ZZ の範囲の SQLSTATE は IBM Data Server Driver for JDBC and SQLJ 用に予約されています。

次の表は、IBM Data Server Driver for JDBC and SQLJ によって生成または使用される SQLSTATE をリストしています。

表 93. IBM Data Server Driver for JDBC and SQLJ により戻される SQLSTATE

クラス	SQLSTATE	説明
01xxx		警告
02xxx		データなし
	02501	カーソル位置が現在行の FETCH に対して無効です。

表 93. IBM Data Server Driver for JDBC and SQLJ により戻される SQLSTATE (続き)

SQLSTATE		
クラス	SQLSTATE	説明
	02506	容認できるエラー
08xxx		接続例外
	08003	接続が存在しません。
	08004	アプリケーション・サーバーが、接続の確立を拒否しました。
	08506	クライアント・リルトの例外です。
0Axxx		機能がサポートされていない
	0A502	このデータベース・インスタンスではアクションまたは操作ができません。
22xxx		データ例外
	22007	日時値のストリング表記が無効です。
	22021	文字がコード化文字セットにありません。
23xxx		制約違反
	23502	列に挿入されるかまたは列を更新する値が NULL ですが、列に NULL 値を入れることはできません。
24xxx		カーソル状態が無効
	24501	識別されたカーソルがオープンしていません。
28xxx		許可例外
	28000	許可名が無効です。
2Dxxx		トランザクション終了が無効
	2D521	SQL COMMIT または ROLLBACK が、現在の操作環境では無効です。
34xxx		カーソル名が無効
	34000	カーソル名が無効です。
3Bxxx		SAVEPOINT が無効
	3B503	SAVEPOINT、RELEASE SAVEPOINT、または ROLLBACK TO SAVEPOINT は、トリガーまたはグローバル・トランザクションでは許可されていません。
40xxx		トランザクションのロールバック
42xxx		構文エラーまたはアクセス規則違反
	42601	文字、トークン、または節が、無効もしくは欠けています。
	42734	重複するパラメーター名、SQL 変数名、カーソル名、条件名、またはラベルが検出されました。
	42807	INSERT、UPDATE、または DELETE は、このオブジェクトでは許可されません。
	42808	INSERT または UPDATE 操作で識別された列は更新できません。
	42815	データ・タイプ、長さ、位取り、値、または CCSID が無効です。
	42820	数値定数が長すぎるか、またはそのデータ・タイプの範囲内にはない値を持っています。

表 93. IBM Data Server Driver for JDBC and SQLJ により戻される SQLSTATE (続き)

SQLSTATE クラス	SQLSTATE	説明
	42968	現行ソフトウェア・ライセンスがないため、接続が失敗しました。
58xxx		システム・エラー
	58008	分散プロトコル・エラーのため、実行が失敗しました。このエラーは、後続の DDM コマンドまたは SQL ステートメントの正常な実行には影響しません。
	58009	会話の割り振り解除の原因となる分散プロトコル・エラーのため、実行が失敗しました。
	58012	指定したパッケージ名と整合性トークンを持つバインド処理がアクティブではありません。
	58014	DDM コマンドはサポートされていません。
	58015	DDM オブジェクトはサポートされていません。
	58016	DDM パラメーターはサポートされていません。
	58017	DDM パラメーターの値がサポートされていません。

## IBM Data Server Driver for JDBC and SQLJ のバージョンおよび環境の情報を確認する方法

IBM Data Server Driver for JDBC and SQLJ のバージョンや、ドライバーが実行される環境に関する情報を判別するには、コマンド行で DB2Jcc ユーティリティーを実行します。

### DB2Jcc の構文

```

▶▶—java—com.ibm.db2.jcc.DB2Jcc—┬──-version┬──-configuration┬──-help▶▶

```

### DB2Jcc オプションの説明

#### -version

IBM Data Server Driver for JDBC and SQLJ がその名前とバージョンを表示することを指定します。

#### -configuration

IBM Data Server Driver for JDBC and SQLJ がその名前とバージョン、その環境に関する情報 (Java ランタイム環境、オペレーティング・システム、パス情報、およびライセンス制限など) を表示することを指定します。

#### -help

DB2Jcc ユーティリティーがサポートする各オプションを記述することを指定します。-help と共に他のオプションが指定されても、それらは無視されます。

## DB2Jcc の出力例

以下の出力は、-configuration パラメーターを指定して DB2Jcc を呼び出した結果です。

図 62. サンプル DB2Jcc 出力

```
(myid@mymachine) /home/myusrid $ java com.ibm.db2.jcc.DB2Jcc -version
[jcc] Driver: IBM DB2 JDBC Universal Driver Architecture 3.50.137

(myid@mymachine) /home/myusrid $ java com.ibm.db2.jcc.DB2Jcc -configuration
[jcc] BEGIN TRACE_DRIVER_CONFIGURATION
[jcc] Driver: IBM_DB2 JDBC Universal Driver Architecture 3.50.137
[jcc] Compatible JRE versions: { 1.4, 1.5 }
[jcc] Target server licensing restrictions: { z/OS: enabled; SQLDS: enabled; iSeries: enabled; DB2 for Unix/Windows: enabled; Cloudscape: enabled; Informix: enabled }
[jcc] Range checking enabled: true
[jcc] Bug check level: 0xff
[jcc] Default fetch size: 64
[jcc] Default isolation: 2
[jcc] Collect performance statistics: false
[jcc] No security manager detected.

[jcc] Detected local client host: lead.svl.ibm.com/9.30.10.102
[jcc] Access to package sun.io is permitted by security manager.

[jcc] JDBC 1 system property jdbc.drivers = null
[jcc] Java Runtime Environment version 1.4.2
[jcc] Java Runtime Environment vendor = IBM Corporation
[jcc] Java vendor URL = http://www.ibm.com/
[jcc] Java installation directory = /wsdb/v91/bldsupp/AIX5L64/jdk1.4.2_sr1/sh/./jre
[jcc] Java Virtual Machine specification version = 1.0
[jcc] Java Virtual Machine specification vendor = Sun Microsystems Inc.
[jcc] Java Virtual Machine specification name = Java Virtual Machine Specification
[jcc] Java Virtual Machine implementation version = 1.4.2
[jcc] Java Virtual Machine implementation vendor = IBM Corporation
[jcc] Java Virtual Machine implementation name = Classic VM
[jcc] Java Runtime Environment specification version = 1.4
[jcc] Java Runtime Environment specification vendor = Sun Microsystems Inc.
[jcc] Java Runtime Environment specification name = Java Platform API Specification
[jcc] Java class format version number = 48.0
[jcc] Java class path = ../home2/myusrid/sqllib/java/db2java.zip:/lib/classes.zip:/home2/myusrid/sqllib/java/sqlj.zip:/test:/home2/myusrid/sqllib/java/db2jcc.jar:/home2/myusrid/sqllib/java/db2jcc_license_cisuz.jar:...
[jcc] Java native library path = /wsdb/v91/bldsupp/AIX5L64/jdk1.4.2_sr1/sh/./jre/bin:/wsdb/v91/bldsupp/AIX5L64/jdk1.4.2_sr1/jre/bin/classic:/wsdb/v91/bldsupp/AIX5L64/jdk1.4.2_sr1/jre/bin:/home2/myusrid/sqllib/lib:/local/cobol:/home2/myusrid/sqllib/samples/c:/usr/lib
[jcc] Path of extension directory or directories = /wsdb/v91/bldsupp/AIX5L64/jdk1.4.2_sr1/sh/./jre/lib/ext
[jcc] Operating system name = AIX
[jcc] Operating system architecture = ppc64
[jcc] Operating system version = 5.3
[jcc] File separator ("/" on UNIX) = /
[jcc] Path separator (":" on UNIX) = :
[jcc] User's account name = myusrid
[jcc] User's home directory = /home2/myusrid
[jcc] User's current working directory = /home2/myusrid
[jcc] Dumping all system properties: { java.assistive=ON, java.runtime.name=Java(TM) 2 Runtime Environment, Standard Edition, sun.boot.library.path=/wsdb/v91/bldsupp/AIX5L64/jdk1.4.2_sr1/sh/./jre/bin, java.vm.version=1.4.2, java.vm.vendor=
```

```

IBM Corporation, java.vendor.url=http://www.ibm.com/, path.separator=;, java.vm.
name=Classic VM, file.encoding.pkg=sun.io, user.country=US, sun.os.patch.level=u
nknown, ... }
[jcc] Dumping all file properties: { }
[jcc] END TRACE_DRIVER_CONFIGURATION

```

## SQLJ プログラムの準備のためのコマンド

SQLJ プログラムを実行用に準備するには、コマンドを使用して、SQLJ ソース・コードを Java ソース・コードに変換し、その Java ソース・コードをコンパイルし、SQLJ シリアライズド・プロファイルを作成してカスタマイズし、DB2 パッケージをバインドします。

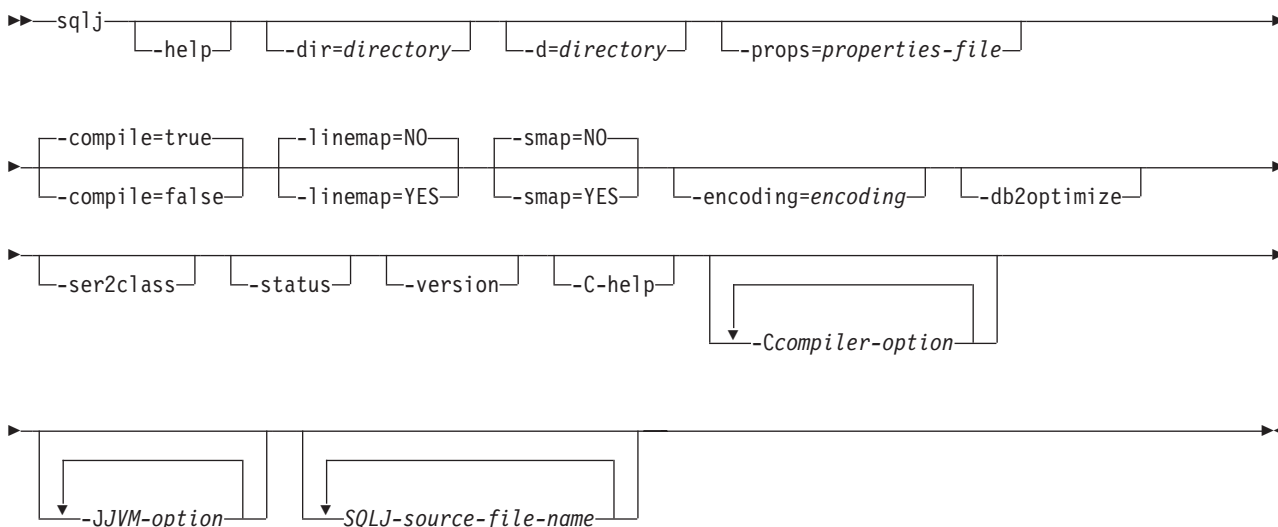
### sqlj - SQLJ 変換プログラム

sqlj コマンドは SQLJ ソース・ファイルを Java ソース・ファイルおよび 0 個以上の SQLJ シリアライズド・プロファイルに変換します。デフォルトでは、sqlj コマンドは Java ソース・ファイルのコンパイルも行います。

#### 許可

なし

#### コマンド構文



#### コマンド・パラメーター

##### -help

変換プログラムがサポートする各オプションが、SQLJ 変換プログラムによって説明されることを指定します。-help と共に他のオプションが指定されても、それらは無視されます。

##### -dir=directory

SQLJ が変換プログラムによって生成される .java ファイル、およびコンパイラ

ーによって生成される .class ファイルを入れるディレクトリーの名前を指定します。デフォルトは、SQLJ ソース・ファイルが入っているディレクトリーです。

変換プログラムは、生成ファイルディレクトリーに配置するときに、SQLJ ソース・ファイルのディレクトリー構造を使用します。例えば、変換プログラムが次の 2 つのファイル処理をします。

- Java パッケージにない file1.sqlj
- Java パッケージ sqlj.test にある file2.sqlj

また、変換プログラムを呼び出すときにパラメーター `-dir=/src` を指定するとします。変換プログラムは file1.sqlj 用の Java ソース・ファイルをディレクトリー `/src` に入れ、file2.sqlj 用の Java ソース・ファイルをディレクトリー `/src/sqlj/test` に入れます。

**-d=directory**

SQLJ が変換プログラムおよびコンパイラーによって生成されるバイナリー・ファイルを入れるディレクトリーの名前を指定します。これらのファイルには、.ser ファイル、name\_SJProfileKeys.class ファイル、およびコンパイラーによって生成される .class ファイルが含まれます。

デフォルトは、SQLJ ソース・ファイルが入っているディレクトリーです。

変換プログラムは、生成ファイルディレクトリーに配置するときに、SQLJ ソース・ファイルのディレクトリー構造を使用します。例えば、変換プログラムが次の 2 つのファイル処理をします。

- Java パッケージにない file1.sqlj
- Java パッケージ sqlj.test にある file2.sqlj

また、変換プログラムを呼び出すときにパラメーター `-d=/src` を指定するとします。変換プログラムは file1.sqlj 用のシリアライズド・プロファイルディレクトリー `/src` に入れ、file2.sqlj 用のシリアライズド・プロファイルディレクトリー `/src/sqlj/test` に入れます。

**-compile=truelfalse**

SQLJ 変換プログラムが生成済みの Java ソースをバイトコードにコンパイルするかどうかを指定します。

**true**

変換プログラムは生成済みの Java ソース・コードをコンパイルします。これはデフォルトです。

**false**

変換プログラムは生成済みの Java ソース・コードをコンパイルしません。

**-linemap=nolyes**

Java 例外の行番号が、SQLJ ソース・ファイル (.sqlj ファイル) の行番号に一致するか、SQLJ 変換プログラムによって生成される Java ソース・ファイル (.java ファイル) の行番号に一致するかを指定します。

**no** Java 例外の行番号は Java ソース・ファイルの行番号に一致します。これはデフォルトです。

**yes**

Java 例外の行番号は SQLJ ソース・ファイルの行番号に一致します。

**-smap=nolyes**

SQLJ 変換プログラムが SQLJ ソース・ファイルごとにソース・マップ (SMAP) ファイルを生成するかどうかを指定します。SMAP ファイルはいくつかの Java 言語デバッグ・ツールで使用されます。このファイルは SQLJ ソース・ファイルの行を、SQLJ 変換プログラムによって生成される Java ソース・ファイルの行にマップします。ファイルは Unicode UTF-8 コード化スキームです。その形式は、以下の Web サイトの Original Java Specification Request (JSR) 45 に説明されています。

<http://www.jcp.org>

**no** SMAP ファイルを生成しません。これはデフォルトです。

**yes**

SMAP ファイルを生成します。SMAP ファイルの名前は *SQLJ-source-file-name.java.smap* です。SQLJ 変換プログラムは SMAP ファイルを、生成された Java ソース・ファイルと同じディレクトリーに置きます。

**-encoding=encoding-name**

ソース・ファイルのエンコードを指定します。例として JIS または EUC があります。このオプションを指定しない場合、オペレーティング・システムのデフォルト・コンバーターが使用されます。

**-db2optimize**

SQLJ 変換プログラムが DB2 に合わせて最適化された接続コンテキスト・クラスのコードを生成することを指定します。**-db2optimize** は、デフォルト・コンテキストのコードではなく、ユーザー定義コンテキストのコードを最適化します。

SQLJ 変換プログラムを **-db2optimize** オプションを指定して実行する際に、アプリケーションが JDBC 3.0 またはそれ以前の関数を使用する場合、生成された Java アプリケーションをコンパイルするには IBM Data Server Driver for JDBC and SQLJ ファイル *db2jcc.jar* が CLASSPATH に入っていなければなりません。アプリケーションが JDBC 4.0 またはそれ以前の関数を使用する場合、生成された Java アプリケーションをコンパイルするには IBM Data Server Driver for JDBC and SQLJ ファイル *db2jcc4.jar* が CLASSPATH に入っていなければなりません。

**-ser2class**

SQLJ 変換プログラムが *.ser* ファイルを *.class* ファイルに変換することを指定します。

**-status**

SQLJ 変換プログラムが実行中に状況メッセージを表示することを指定します。

**-version**

SQLJ 変換プログラムが IBM Data Server Driver for JDBC and SQLJ のバージョンを表示することを指定します。情報の形式は次のとおりです。

IBM SQLJ *xxxx.xxxx.xx*

**-C-help**

SQLJ 変換プログラムが Java コンパイラーのヘルプ情報を表示することを指定します。



### -Ccompiler-option

ダッシュ (-) で始まる有効な Java コンパイラー・オプションを指定します。

-C とコンパイラー・オプションの間にはスペースを入れしないでください。複数のコンパイラー・オプションを指定する必要がある場合、各コンパイラー・オプションの前に -C を付けてください。以下に例を示します。

-C-g -C-verbose

すべてのオプションは Java コンパイラーに渡され、SQLJ 変換プログラムには使用されません。ただし、以下のオプションを除きます。

### -classpath

SQLJ 変換プログラムおよび Java コンパイラーによって使用されるユーザー・クラス・パスを指定します。この値は CLASSPATH 環境変数をオーバーライドします。

### -sourcepath

SQLJ 変換プログラムおよび Java コンパイラーがクラスまたはインターフェース定義を検索するソース・コード・パスを指定します。SQLJ 変換プログラムは、JAR または zip ファイル内ではなく、ディレクトリー内の .sqlj および .java ファイルのみ検索します。

### -JVM-option

sqlj コマンドの実行場所である Java 仮想マシン (JVM) に渡されるオプションを指定します。オプションは、ダッシュ (-) で始まる有効な JVM オプションでなければなりません。-J と JVM オプションの間にはスペースを入れしないでください。複数の JVM オプションを指定する必要がある場合、各コンパイラー・オプションの前に -J を付けてください。以下に例を示します。

-J-Xmx128m -J-Xmine2M

### SQLJ-source-file-name

変換される SQLJ ソース・ファイルのリストを指定します。これは必須パラメーターです。すべての SQLJ ソース・ファイル名には拡張子 .sqlj が付いていなければならない。

## 出力

各ソース・ファイル *program-name.sqlj* ごとに、SQLJ 変換プログラムは以下のファイルを作成します。

- 生成済みソース・プログラム

生成済みソース・ファイルには *program-name.java* という名前が付けられます。

- SQLJ 実行可能節で使用される接続コンテキスト・クラスごとのシリアルイズド・プロファイル・ファイル

直列化されたプロファイル名の形式は、以下のようになります。

*program-name\_SJProfileIDNumber.ser*

- SQLJ 変換プログラムが Java コンパイラーを呼び出す場合、コンパイラーが生成するクラス・ファイル

## 例

```
sqlj -encoding=UTF8 -C-0 MyApp.sqlj
```

## db2sqljcustomize - SQLJ プロファイル・カスタマイザー

db2sqljcustomize は、組み込み SQL ステートメントを含む SQLJ プロファイルを処理します。

デフォルトでは、それぞれの分離レベルごとに 1 つずつ、4 つの DB2 パッケージが db2sqljcustomize によって作成されます。db2sqljcustomize では、実行時に使用できるように、DB2 の固有情報を使ってプロファイルが拡大されます。

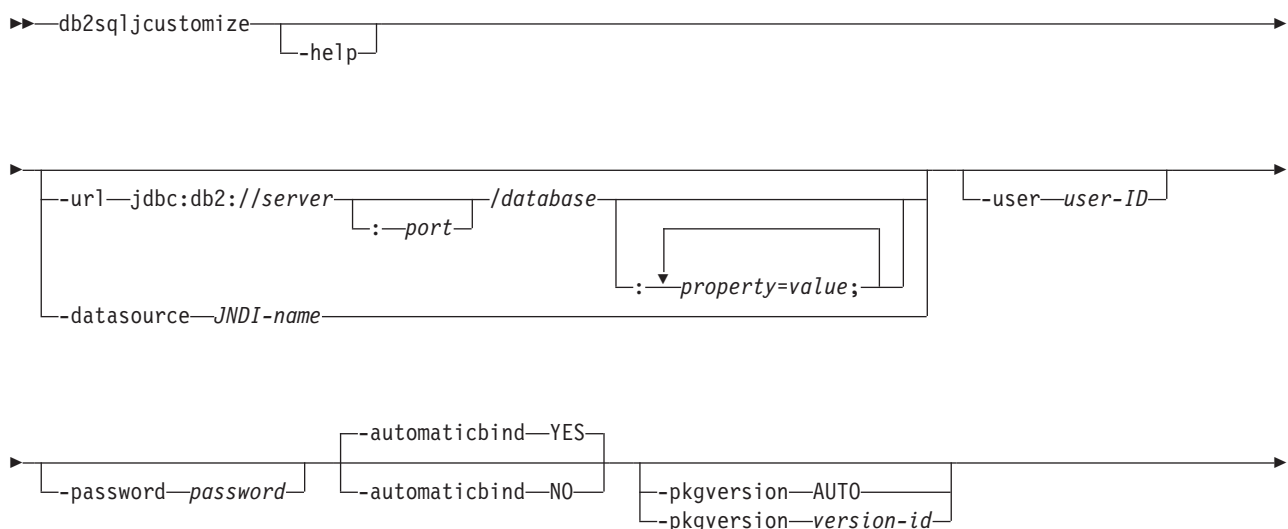
### 許可

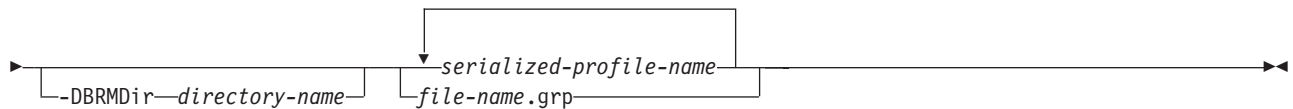
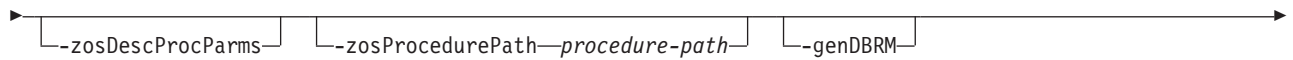
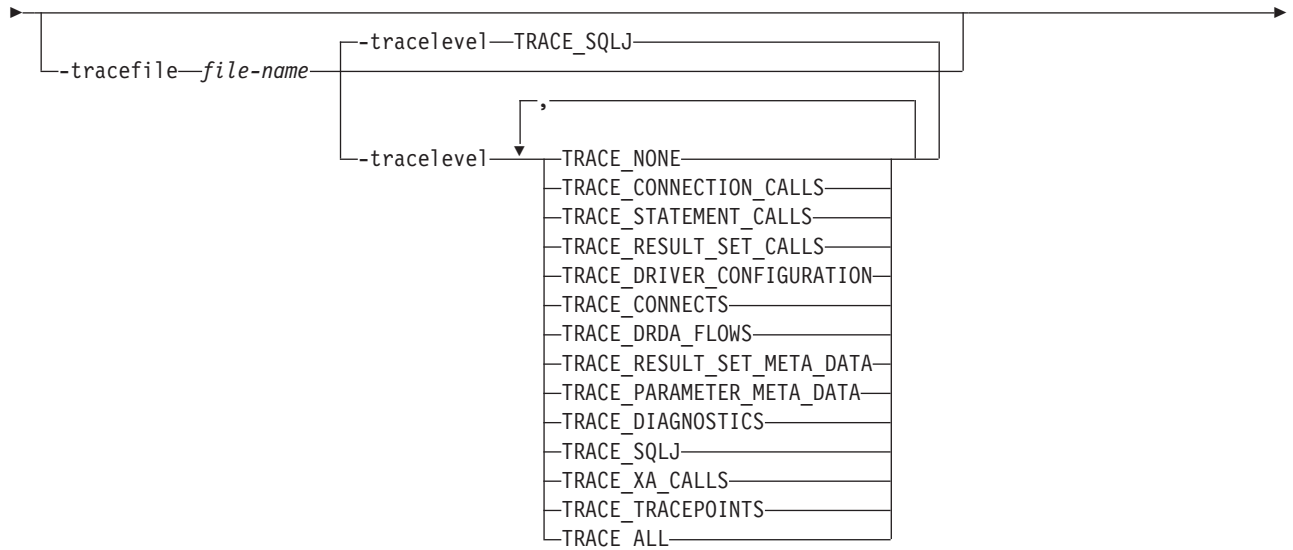
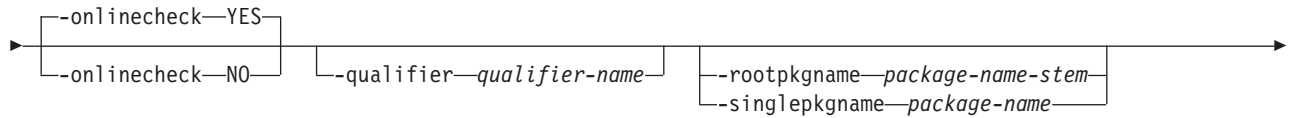
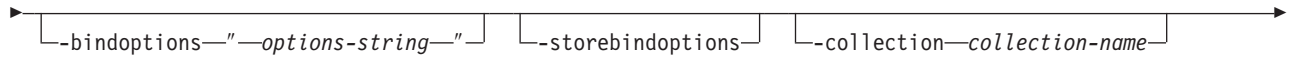
プロセスの特権セットには、以下の権限のうちの 1 つを組み込む必要があります。

- SYSADM 権限
- DBADM 権限
- パッケージが存在しない場合の BINDADD 特権と、以下のいずれかの特権
  - CREATEIN 特権
  - パッケージのスキーマ名が存在しない場合、データベースに対する IMPLICIT\_SCHEMA 権限
- パッケージが存在する場合に必要な特権:
  - スキーマに対する ALTERIN 特権
  - パッケージに対する BIND 特権

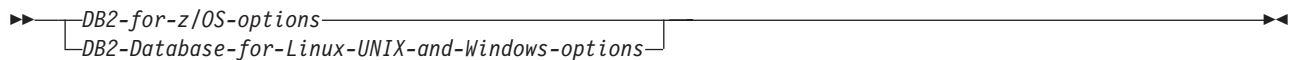
ユーザーには、アプリケーションで静的 SQL ステートメントをコンパイルするのに必要なすべての特権も必要です。グループに認可された特権が、静的ステートメントの許可の検査に使用されることはありません。ユーザーが、SYSADM 権限は持ってもバインドを完了するための明示的な特権を持っていない場合、DB2 データベース・マネージャーによって明示的な DBADM 権限が自動的に認可されます。

### コマンド構文

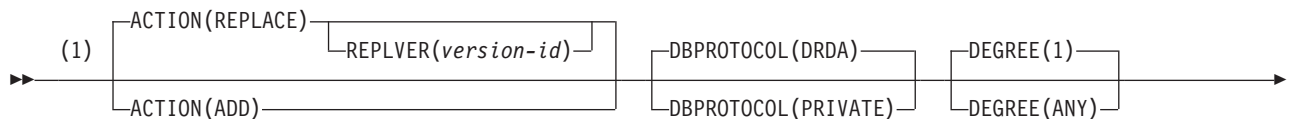


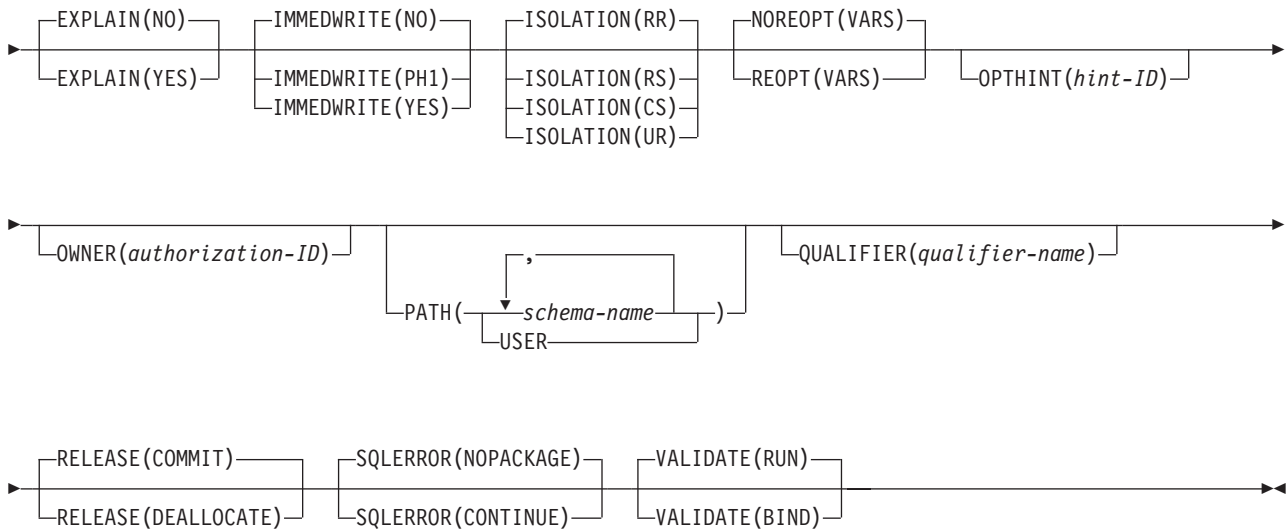


**options-string:**



**DB2 for z/OS options:**

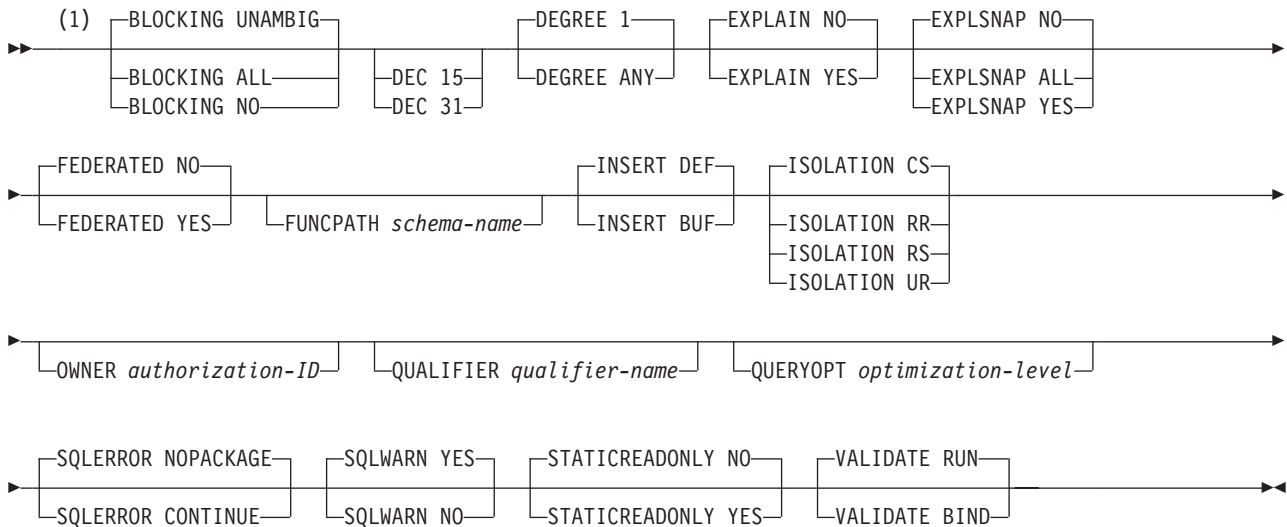




注:

- 1 これらのオプションは、どのような順序で指定してもかまいません。

### DB2 Database for Linux, UNIX, and Windows options



注:

- 1 これらのオプションは、どのような順序で指定してもかまいません。

### コマンド・パラメーター

#### -help

SQLJ カスタマイザーがサポートする各オプションが、このカスタマイザーによって説明されることを指定します。 -help と共に他のオプションが指定されても、それらは無視されます。

#### -url

プロファイルがカスタマイズされるデータ・ソースの URL を指定します。 -automaticbind オプションまたは -onlinecheck オプションが YES に指定さ

れているか、デフォルトで YES になった場合、この URL が表すデータ・ソースへの接続が確立されます。-url 値の可変部は、以下のとおりです。

**server**

DB2 サブシステムが存在する z/OS システムのドメイン名または IP アドレス。

**port**

DB2 サブシステムに割り当てられている TCP/IP サーバーのポート番号。デフォルトは 446 です。

**-url**

プロファイルがカスタマイズされるデータ・ソースの URL を指定します。-automaticbind オプションまたは -onlinecheck オプションが YES に指定されているか、デフォルトで YES になった場合、この URL が表すデータ・ソースへの接続が確立されます。-url 値の可変部は、以下のとおりです。

**server**

データベース・サーバーが存在するオペレーティング・システムのドメイン名または IP アドレス。

**port**

データベース・サーバーに割り当てられている TCP/IP サーバーのポート番号。デフォルトは 446 です。

**database**

プロファイルがカスタマイズされるデータベース・サーバーの名前。

接続を DB2 for z/OS サーバーに対して行う場合、*database* は、インストール中に定義される DB2 ロケーション名です。この値の中の文字はすべて大文字でなければなりません。サーバーで以下の SQL ステートメントを実行することにより、ロケーション名を判別できます。

```
SELECT CURRENT SERVER FROM SYSIBM.SYSDUMMY1;
```

接続を DB2 Database for Linux, UNIX, and Windows サーバーに対して行う場合、*database* は、インストール中に定義されるデータベース名です。

接続を IBM Cloudscape サーバーに対して行う場合、*database* は、データベースを含むファイルの完全修飾名です。この名前は、二重引用符 (") で囲まなければなりません。以下に例を示します。

```
"c:/databases/testdb"
```

```
property=value;
```

JDBC 接続のプロパティ。

```
property=value;
```

JDBC 接続のプロパティ。

**-datasource JNDI-name**

JNDI に登録された DataSource オブジェクトの論理名を指定します。DataSource オブジェクトは、プロファイルがカスタマイズされるデータ・ソースを表します。-automaticbind オプションまたは -onlinecheck オプションが YES に指定されているか、デフォルトで YES になった場合、このデータ・ソースへの接続が

確立されます。-datasource は、-url を指定する代わりに指定できます。  
DataSource オブジェクトは、IBM Data Server Driver for JDBC and SQLJ Type  
4 接続 を使用する接続を表す必要があります。

**-user *user-ID***

オンライン検査またはパッケージのバインドのためにデータ・ソースに接続する  
ときに使うユーザー ID を指定します。-url を指定する場合は、-user を指定し  
なければなりません。-datasource を指定し、*JNDI-name* が表す DataSource オブ  
ジェクトにユーザー ID が含まれない場合、-user を指定しなければなりません。

**-password *password***

オンライン検査またはパッケージのバインドのためにデータ・ソースに接続する  
ときに使うパスワードを指定します。-url を指定する場合は、-password を指  
定しなければなりません。-datasource を指定し、*JNDI-name* が表す DataSource  
オブジェクトにパスワードが含まれない場合、-password を指定しなければなり  
ません。

**-automaticbind YES|NO**

-url パラメーターで指定されているデータ・ソースにおいてカスタマイザーが  
DB2 パッケージをバインドするかどうかを指定します。

デフォルトは YES です。

パッケージ数と、それらのパッケージの分離レベルは、-rootpkgname および  
-singlepkgname オプションで制御します。

バインド操作が功を奏するには、以下の条件が満たされる必要があります。

- TCP/IP と DRDA がターゲットのデータ・ソースにインストールされてい  
なければなりません。
- -url、-username、および -password にそれぞれ有効な値を指定する必要があ  
ります。
- -username の値には、ターゲット・データ・ソースにおいてパッケージをバイ  
ンドする許可がなければなりません。

**-pkgversion AUTO|*version-id***

カスタマイズしようとしているシリアライズド・プロファイル用のパッケージが  
サーバー側でバインドされるときに使用するパッケージ・バージョンを指定しま  
す。db2sqljcustomize は、シリアライズド・プロファイル内と DB2 パッケージ  
内にバージョン ID を保管します。実行時のバージョン検査では、バージョン  
名ではなく、整合性トークンがベースになります。整合性トークンに基づいたバ  
ージョン名を自動的に生成するには、-pkgversion AUTO を指定します。

デフォルトでは、バージョンはありません。

**-bindoptions *options-string***

スペースで区切ったオプションのリストを指定します。それらのオプションに  
は、同名の DB2 プリコンパイル・オプションおよびバインド・オプションと同  
じ機能があります。DB2 for z/OS システム上で実行されるプログラムを準備  
する場合、DB2 for z/OS のオプションを指定します。DB2 Database for Linux,  
UNIX, and Windows システム上で実行されるプログラムを準備する場合、DB2  
Database for Linux, UNIX, and Windows のオプションを指定します。

**バインド・オプションに関する注意事項:**

- ISOLATION を指定するのは、-singlepkgname オプションも指定する場合だけです。
- STATICREADONLY をサポートするサーバーの場合は、STATICREADONLY の値は YES、その他のサーバーの場合は、NO となります。  
 STATICREADONLY YES を指定すると、DB2 は未確定カーソルを読み取り専用カーソルであるものとして処理します。イテレーター宣言のエラーのトラブルシューティングの場合、STATICREADONLY NO を明示的に指定する必要があります。あるいは、確定済みになるようにイテレーターを宣言します。たとえば、イテレーターを確定的に更新可能にする場合は、`sqlj.runtime.ForUpdate` をインプリメントするようにイテレーターを宣言します。イテレーターを読み取り専用にする場合、イテレーターを使用する SELECT ステートメントに FOR READ ONLY 節を付け加えます。

**重要:** パッケージのバインディングを行う場所であるデータ・ソースに適したプログラム準備オプションのみを指定してください。IBM Data Server Driver for JDBC and SQLJ の一部の値およびデフォルト値は、DB2 用の値およびデフォルト値とは異なります。

**-storebindoptions**

-bindoptions および -staticpositioned パラメーターの値をシリアライズド・プロファイル内に保管することを指定します。-bindoptions または -staticpositioned パラメーターを使用しないで `db2sqljbind` を呼び出すと、シリアライズド・プロファイルに保管されている値がバインド操作で使用されます。1 回の `db2sqljcustomize` の呼び出しで複数のシリアライズド・プロファイルを指定すると、パラメーター値は、それぞれのシリアライズド・プロファイル内に保管されます。保管値は、`db2sqljprint` ユーティリティからの出力に表示されます。

**-collection *collection-name***

`db2sqljcustomize` がバインドするパッケージの修飾子。 `db2sqljcustomize` は、カスタマイズされたシリアライズド・プロファイル内にその値を保管します。その値は、関連パッケージのバインディング時に使用されます。このパラメーターを指定しないと、`db2sqljcustomize` では NULLID のコレクション ID が使用されます。

**-onlinecheck YES|NO**

SQLJ プログラム内のデータ・タイプのオンライン検査を実行するかどうかを指定します。-url または -datasource オプションは、オンライン検査で使用されるデータ・ソースを決定します。-url または -datasource パラメーターを指定した場合のデフォルトは YES です。それ以外の場合のデフォルトは NO です。

**-qualifier *qualifier-name***

オンライン検査で使用するための、SQLJ プログラム内の非修飾オブジェクトの修飾子を指定します。パッケージのバインディング時には、この値は修飾子として使用されません。

**-rootpkgname|singlepkgname**

プログラムに関連したパッケージの名前を指定します。-automaticbind が NO である場合、これらのパッケージ名が `db2sqljbind` 実行時に使用されます。各パラメーターの意味は、以下のとおりです。



**-rootpkgname** *package-name-stem*

カスタマイザーが、4 つの DB2 分離レベルのそれぞれに対して 1 つずつ、4 つのパッケージを作成することを指定します。4 つのパッケージの名前は以下のとおりです。

*package-name-stem1*

分離レベル UR 用

*package-name-stem2*

分離レベル CS 用

*package-name-stem3*

分離レベル RS 用

*package-name-stem4*

分離レベル RR 用

-longpkgname が指定されていない場合、*package-name-stem* は 7 バイト以下の英数字ストリングでなければなりません。

-longpkgname が指定されている場合、*package-name-stem* は 127 バイト以下の英数字ストリングでなければなりません。

**-singlepkgname** *package-name*

カスタマイザーが *package-name* という名前の 1 つのパッケージを作成することを指定します。このオプションを指定した場合、プログラムはただ 1 つの分離レベルで実行されます。パッケージの分離レベルの指定は、-bindoptions オプション・ストリングの ISOLATION オプションで指定します。

-longpkgname が指定されていない場合、*package-name* は 8 バイト以下の英数字ストリングでなければなりません。

-longpkgname が指定されている場合、*package-name* は 128 バイト以下の英数字ストリングでなければなりません。

-singlepkgname オプションの使用は推奨されていません。

**推奨:** ターゲット・データ・ソースが DB2 for z/OS である場合は、*package-name-stem* または *package-name* の値には大文字を使用してください。特定の CCSID 値で定義された DB2 for z/OS システムでは、パッケージ名またはコレクション名内の小文字が許容されません。

-rootpkgname または -singlepkgname を指定しない場合、db2sqljcustomize は直列化されたプロファイル名に基づいた 4 つのパッケージ名を生成します。直列化されたプロファイル名の形式は、以下のようになります。

*program-name\_SJProfileIDNumber.ser*

生成される 4 つのパッケージ名は、以下の形式になります。

*Bytes-from-program-nameIDNumberPkgIsolation*

441 ページの表 94 生成されるパッケージ名の部分およびそれぞれの部分のバイト数を示します。

パッケージ名の最大長は、*maxlen* です。-longpkgname が指定されない場合、*maxlen* は 8 です。-longpkgname が指定される場合は、*maxlen* は 128 です。

表 94. db2sqljcustomize によって生成されるパッケージ名の部分

パッケージ名の部分	バイト数	値
<i>Bytes-from-program-name</i>	$m=\min(\text{Length}(\text{program-name}), \text{maxlen}-1-\text{Length}(\text{IDNumber}))$	<i>program-name</i> の最初の $m$ バイト。大文字。
<i>IDNumber</i>	$\text{Length}(\text{IDNumber})$	<i>IDNumber</i>
<i>PkgIsolation</i>	1	1、2、3、または 4。この値は、パッケージのトランザクション分離レベルを表します。表 95を参照してください。

表 95は、db2sqljcustomize によって生成されるパッケージ名の *PkgIsolation* 部分の値を示します。

表 95. *PkgIsolation* 値および関連した分離レベル

<i>PkgNumber</i> 値	パッケージの分離レベル
1	非コミット読み取り (UR)
2	カーソル固定 (CS)
3	読み取り固定 (RS)
4	反復可能読み取り (RR)

例: プロファイル名が *ThisIsMyProg\_SJProfile111.ser* であるとしてします。db2sqljcustomize オプション *-longpkgname* が指定されていません。したがって、*Bytes-from-program-name* は、*ThisIsMyProg* の大文字に変換された最初の 4 バイト、THIS です。*IDNumber* は 111 です。4 つのパッケージ名は以下のとおりです。

```
THIS1111
THIS1112
THIS1113
THIS1114
```

例: プロファイル名が *ThisIsMyProg\_SJProfile111.ser* であるとしてします。db2sqljcustomize オプション *-longpkgname* が指定されています。したがって、*Bytes-from-program-name* は、*ThisIsMyProg* が大文字に変換されたもの、つまり THISISMYPROG です。*IDNumber* は 111 です。4 つのパッケージ名は以下のとおりです。

```
THISISMYPROG1111
THISISMYPROG1112
THISISMYPROG1113
THISISMYPROG1114
```

例: プロファイル名が *A\_SJProfile0.ser* であるとしてします。*Bytes-from-program-name* は A で、*IDNumber* は 0 です。したがって、4 つのパッケージ名は以下のとおりです。

```
A01
A02
A03
A04
```

db2sqljcustomize にパッケージ名を生成させることは、推奨されていません。生成されたパッケージ名に既存のパッケージの名前と同じものがある場合、

db2sqljcustomize は既存のパッケージを上書きします。パッケージ名が必ず固有になるようにするためには、-rootpkgname を指定します。

#### **-longpkgname**

128 バイトまでの DB2 パッケージ名を db2sqljcustomize で生成できることを指定します。このオプションは、長いパッケージ名をサポートするサーバーでパッケージをバインドする場合にのみ使用してください。-singlepkgname または -rootpkgname を指定する場合、以下の条件では、-longpkgname も指定しなければなりません。

- -singlepkgname の引数が 8 バイトよりも長い。
- -rootpkgname の引数が 7 バイトよりも長い。

#### **-staticpositioned NOIYES**

イテレーターを使用する位置指定 UPDATE ステートメントと同じソース・ファイルに宣言されているイテレーターの場合、静的にバインドされたステートメントとしてその位置指定 UPDATE を実行するかどうかを指定します。デフォルトは NO です。NO は、動的に準備されるステートメントとして位置指定 UPDATE を実行することを意味します。

#### **-zosDescProcParms**

DB2 for z/OS は、ストアード・プロシージャ・パラメーターに対する DESCRIBE 操作の実行を指定します。

-zosDescProcParms は、DB2 for z/OS データベース・サーバーで実行されるプログラムにのみ適用されます。

DESCRIBE 情報が入手可能な場合、SQLJ には INOUT および OUT パラメーターの長さおよび精度に関する情報が含まれるため、これらのパラメーターに必要なメモリー量のみが割り振られます。DESCRIBE 情報が入手できることによって最も大きな影響を受けるのは、INOUT、LOB OUT、および decimal OUT 文字パラメーターのためのストレージの使用です。

-zosDescProcParms を指定すると、DB2 データベース・サーバーは、指定された値または -zosProcedurePath のデフォルト値を使用して、DESCRIBE 情報が要求されるストアード・プロシージャの非修飾名を解決します。

#### **-zosProcedurePath *procedure-path***

SQLJ プログラムのオンライン・チェック時に DB2 for z/OS が非修飾ストアード・プロシージャ名の解決に使用する、スキーマ名のリストを指定します。

-zosProcedurePath は、DB2 for z/OS データベース・サーバーで実行されるプログラムにのみ適用されます。

リストは、スキーマ名のコンマ区切りのリストを二重引用符で囲んだストリング値になります。DB2 データベース・サーバーは、そのリストを非修飾ストアード・プロシージャ名を解決するための SQL パスに挿入します。SQL パスは次のとおりです。

```
SYSIBM, SYSFUN, SYSPROC, procedure-path, qualifier-name, user-ID
```

*qualifier-name* は -qualifier パラメーターの値であり、*user-ID* は -user パラメーターの値です。

DB2 データベース・サーバーは、データベース・サーバーに存在するストアード・プロシージャの名前と一致するスキーマ名が見つかるまで、SQL パス内

のスキーマ名を左から右に確認します。一致するものがあると、DB2 データベース・サーバーはそのストアード・プロシージャのパラメーターに関する情報を DB2 カタログから取得します。DB2 データベース・サーバーが一致するものを検出できなかった場合、SQLJ は DB2 カタログ情報を使用せずにパラメーター・データを設定します。

-zosProcedurePath を指定しない場合、DB2 データベース・サーバーは次の SQL パスを使用します。

SYSIBM, SYSFUN, SYSPROC, *qualifier-name*, *user-ID*

-qualifier パラメーターを指定しない場合、SQL パスには *qualifier-name* は含まれません。

### **-genDBRM**

db2sqljcustomize がデータベース要求モジュール (DBRM) を生成することを指定します。これらの DBRM は、DB2 for z/OS プランおよびパッケージを作成するのに使用できます。

-genDBRM は、DB2 for z/OS データベース・サーバーで実行されるプログラムにのみ適用されます。

-genDBRM および -automaticbind NO が指定されていると、db2sqljcustomize は DBRM を作成しますが、それらを DB2 パッケージにバインドすることはありません。-genDBRM および -automaticbind YES が指定されていると、db2sqljcustomize は DBRM を作成し、それらを DB2 パッケージにバインドします。

各 DB2 分離レベルごとに 1 つの DBRM が作成されます。生成された DBRM ファイルの命名規則は、パッケージの命名規則と同じものです。例えば、-rootpkgname SQLJSA0 が指定されており、-genDBRM も指定されている場合、4 つの DBRM ファイルの名前は次のようになります。

- SQLJSA01
- SQLJSA02
- SQLJSA03
- SQLJSA04

### **-DBRMDir *directory-name***

-genDBRM を指定する場合、-DBRMDir は生成された DBRM ファイルを db2sqljcustomize が配置するローカル・ディレクトリーを指定します。デフォルトは現行ディレクトリーです。

-DBRMdir は、DB2 for z/OS データベース・サーバーで実行されるプログラムにのみ適用されます。

### **-tracefile *file-name***

トレースを使用可能にし、トレース情報のための出力ファイルを示します。このオプションを指定できるのは、IBM ソフトウェア・サポートから指示された場合のみです。

### **-tracelevel**

-tracefile を指定する場合、db2sqljcustomize の実行中に何をトレースするかを指示します。デフォルトは TRACE\_SQLJ です。このオプションを指定できるのは、IBM ソフトウェア・サポートから指示された場合のみです。

### *serialized-profile-name*|*file-name*.grp

カスタマイズしようとしている 1 つ以上のシリアライズド・プロファイルの名前を指定します。指定されたシリアライズド・プロファイルは、CLASSPATH 環境変数で指定されたディレクトリーになければなりません。

直列化されたプロファイル名の形式は、以下のようになります。

*program-name*\_SJProfileIDNumber.ser

シリアライズド・プロファイルの名前を指定するときは、.ser 拡張子を付けても付けなくてもかまいません。

*program-name* は、拡張子 .sqlj の付いていない SQLJ ソース・プログラムの名前です。*n* は、0 から *m-1* までの整数です。ただし、*m* は、SQLJ ソース・プログラムから SQLJ 変換プログラムが生成したシリアライズド・プロファイルの数です。

以下のいずれかの方法で、シリアライズド・プロファイルの名前を指定することができます。

- db2sqljcustomize コマンドで名前の一覧を表示します。複数のシリアライズド・プロファイル名は、スペースを使用して区切る必要があります。
- *file-name*.grp という名前のファイルに、シリアライズド・プロファイル名を 1 行に 1 つずつ指定し、db2sqljcustomize コマンドに *file-name*.grp を指定します。

複数のシリアライズド・プロファイル名を指定し、しかもデフォルト値の -automaticbind YES を指定または使用した場合、db2sqljcustomize によってそれらのプロファイルから 1 つの DB2 パッケージがバインドされます。

db2sqljcustomize を使用して、複数のシリアライズド・プロファイルから 1 つの DB2 パッケージを作成するには、-rootpkgname または -singlepkgname オプションも一緒に指定する必要があります。

複数のシリアライズド・プロファイル名を指定し、さらに -automaticbind NO も指定した場合に、db2sqljbind の実行時にそのシリアライズド・プロファイルを 1 つの DB2 パッケージにバインドするには、シリアライズド・プロファイル名が同じ順序で並んだ同じリストを db2sqljcustomize および db2sqljbind に指定する必要があります。

## 出力

db2sqljcustomize を実行すると、カスタマイズされたシリアライズド・プロファイルが作成されます。また、automaticbind 値が YES の場合は、DB2 パッケージも作成されます。

## 例

```
db2sqljcustomize -user richler -password mordecai
  -url jdbc:db2:/server:50000/sample -collection duddy
  -bindoptions "EXPLAIN YES" pgmname_SJProfile0.ser
```

## 使用上の注意

常にオンライン検査をお勧めします。シリアライズド・プロファイルをカスタマイズする際には、オンライン検査を使用するよう強くお勧めします。オンライン検査

によって、DB2 ホスト変数のデータ・タイプおよび長さに関する情報が確認されます。これは、以下の項目の場合特に重要です。

- `java.lang.String` ホスト変数および `CHAR` 列を備えた述部

他のホスト言語の文字変数とは違って、Java の `String` ホスト変数の宣言では、長さ属性は使用されません。文字ホスト変数が含まれる照会を正しく最適化するには、そのホスト変数の長さが `DB2` で必要です。たとえば、ある照会が述部をもっていて、そこで、`String` ホスト変数が `CHAR` 列と比較され、`CHAR` 列上で索引が定義されると仮定します。`DB2` がホスト変数の長さを判別できない場合、索引スキャンではなく表スペース・スキャンを行うことがあります。オンライン検査は、対応する文字カラムの長さを提示することによって、このような問題が起きないようにします。

- `java.lang.String` ホスト変数および `GRAPHIC` 列を備えた述部

オンライン検査を行わないと、`String` ホスト変数を `GRAPHIC` 列と比較する述部が検出された場合、バインド・エラー (`SQLCODE -134`) が `DB2` から発行されることがあります。

- リモート・サーバーでの `SQLJ SELECT` ステートメントの結果表内の列名:

オンライン検査を行わないと、リモートの `SELECT` の結果表の列名をドライバーが判別できません。

**複数のシリアルライズド・プロファイルの一括カスタマイズ:** 複数のシリアルライズド・プロファイルをまとめてカスタマイズし、1 つの `DB2` パッケージを作成することができます。そのように作成する場合に、`-staticpositioned YES` を指定すると、カーソルの宣言とは異なるソース・ファイル内に `UPDATE` または `DELETE` ステートメントがあっても、パッケージ内で事前に宣言されているカーソルを参照する位置指定の `UPDATE` または `DELETE` ステートメントは静的に実行されます。複数のソース・ファイルでプログラムが構成されている場合に、`-staticpositioned YES` の動作が望ましい場合、パッケージ内で位置指定の `UPDATE` または `DELETE` ステートメントより前にカーソル宣言が出現するように、`db2sqljcustomize` コマンド内でプロファイルを配列する必要があります。そのためには、イテレーターを参照する位置指定の `UPDATE` または `DELETE` ステートメントを収めたプロファイルより前に、結果表をイテレーターに割り当てる `SELECT` ステートメントを収めたプロファイルをリストします。

**データ・ソースでカスタマイズされたカスタマイズ済みシリアルライズド・プロファイルの、別のデータ・ソースでの使用:** `db2sqljcustomize` を実行して、ある 1 つのデータ・ソースで、`SQLJ` プログラム用にカスタマイズされたシリアルライズド・プロファイルを作成してから、別のデータ・ソースでそのシリアルライズド・プロファイルを使用することができます。それには、`db2sqljcustomize` を 1 回実行して作成したカスタマイズ済みシリアルライズド・プロファイルに対して、`db2sqljbind` を複数回実行します。それらのデータ・ソースでプログラムを実行するときは、そのプログラムからアクセスする `DB2` オブジェクトは、どちらのデータ・ソースでも同じでなければなりません。たとえば、どのデータ・ソースの表も、同じコード化スキームおよび同じデータ・タイプの同じ列をもっていなければなりません。

**-collection パラメーターの使用:** `db2sqljcustomize` は、作成した各カスタマイズ済みシリアルライズド・プロファイル内に `DB2` コレクション名を保管します。 `SQLJ`



プログラムが実行されると、ドライバーは、カスタマイズ済みシリアライズド・プロファイル内に保管されているコレクション名を使用して、実行するパッケージを検索します。カスタマイズ済みシリアライズド・プロファイルに保管されている名前は、`-collection` パラメーターの値によって判別されます。シリアライズド・プロファイルに保管できるコレクション ID は 1 つだけです。ただし、`-bindoptions` パラメーター内で `COLLECTION` オプションを指定すれば、1 つのシリアライズド・プロファイルを複数のパッケージ・コレクションにバインドすることができます。シリアライズド・プロファイルに指定されているもの以外のコレクション内にあるパッケージを実行するには、プログラム内で `SET CURRENT PACKAGESET` ステートメントを使用します。

**VERSION パラメーターの使用:** 1 つの SQLJ プログラムの複数バージョンのパッケージを 1 つのコレクションにバインドするときは、`VERSION` パラメーターを使用します。これを実行するのは、SQLJ ソース・プログラムを変更した後で、新旧バージョンのプログラムを実行することになった場合です。

2 つのバージョンのパッケージを保守するには、次のステップを行います。

1. ソース・プログラム内のコードを変更します。
2. 新規のシリアライズド・プロファイルを作成するように、ソース・プログラムを変換します。オリジナルのシリアライズド・プロファイルを上書きしないよう気を付けてください。
3. `db2sqljcustomize` を実行し、シリアライズド・プロファイルをカスタマイズしてから、オリジナル・パッケージと同じパッケージ名を付けて同じコレクション内に `DB2` パッケージを作成します。それには、オリジナル・パッケージの作成時に使用したのと同じ `-rootpkgname` および `-collection` の値を、新規のパッケージのバインド時に使用します。新規のカスタマイズ済みシリアライズド・プロファイル内と新規のパッケージ内にバージョン ID を書き込むには、`-bindoptions` パラメーターに `VERSION` オプションを指定します。

このステップを実行するときは、`VERSION` オプションを必ず指定してください。指定しないと、オリジナル・パッケージが上書きされてしまいます。

旧バージョンのプログラムを実行すると、旧バージョンのパッケージが `DB2` でロードされます。新バージョンのプログラムを実行すると、新バージョンのパッケージが `DB2` でロードされます。

**DB2 for z/OS でのパッケージおよびプランのバインディング:** `db2sqljcustomize` `-genDBRM` パラメーターを使用してローカル・システムに `DBRM` を作成することができます。続いて、それらの `DBRM` を `DB2 for z/OS` システムに転送して、それらをパッケージまたはプランにバインドすることができます。この手法を使用する場合、`DBRM` ファイルをバイナリー・ファイルとして `z/OS` システムに、つまりレコード形式 `FB` およびレコード長 `80` のパーティション・データ・セットに転送する必要があります。パッケージまたはプランをバインドするときには、以下のバインド・オプションの値を指定する必要があります。

#### ENCODING(EBCDIC)

`DB2 for z/OS` 上の `IBM Data Server Driver for JDBC and SQLJ` では、パッケージおよびプランには `EBCDIC` エンコードが必要になります。



## DYNAMICRULES(BIND)

このオプションにより、SQLJ が動的 SQL を使用するときの許可規則が一貫性のあるものになります。SQLJ は、複数の SQLJ プログラムが関係する位置指定 UPDATE または DELETE 操作に対して動的 SQL を使用します。

## DBPROTOCOL(DRDA)

専用プロトコルは推奨されていません。DBPROTOCOL(DRDA) を全アプリケーションで使用してください。特に、3 部から構成されるリモート表名を使用する SQLJ アプリケーションの場合は、必ず DBPROTOCOL(DRDA) を使用してください。そうしないと、それらのアプリケーションは失敗する可能性があります。

## db2sqljbind - SQLJ プロファイル・バインダー

db2sqljbind は、以前に db2sqljcustomize コマンドでカスタマイズされたシリアルライズド・プロファイルの DB2 パッケージをバインドします。

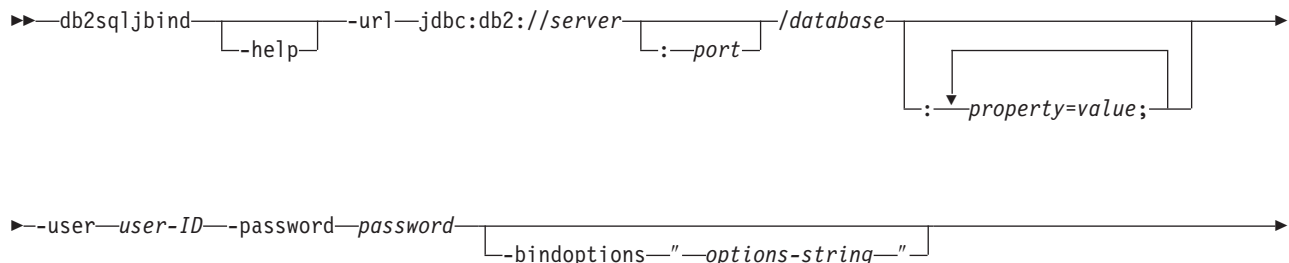
### 許可

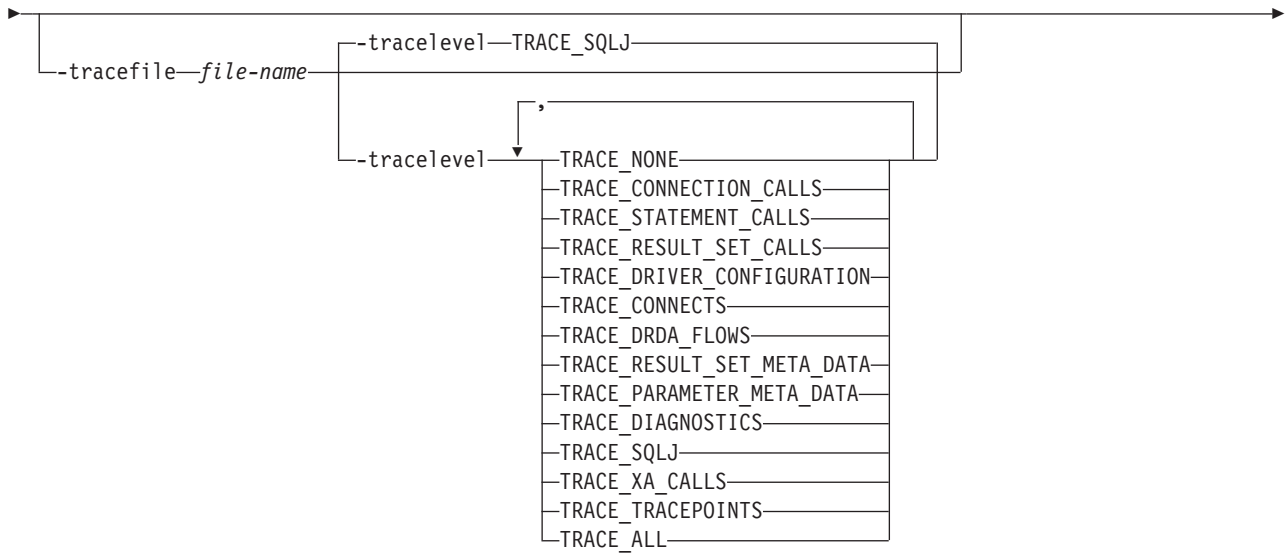
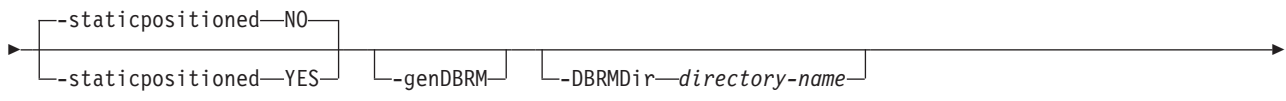
プロセスの特権セットには、以下の権限のうちの 1 つを組み込む必要があります。

- SYSADM 権限
- DBADM 権限
- パッケージが存在しない場合の BINDADD 特権と、以下のいずれかの特権
  - CREATEIN 特権
  - パッケージのスキーマ名が存在しない場合、データベースに対する IMPLICIT\_SCHEMA 権限
- パッケージが存在する場合に必要な特権:
  - スキーマに対する ALTERIN 特権
  - パッケージに対する BIND 特権

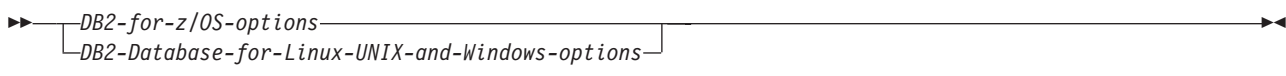
ユーザーには、アプリケーションで静的 SQL ステートメントをコンパイルするのに必要なすべての特権も必要です。グループに認可された特権が、静的ステートメントの許可の検査に使用されることはありません。ユーザーが、SYSADM 権限は持っていない場合、DB2 データベース・マネージャーによって明示的な DBADM 権限が自動的に認可されます。

### コマンド構文

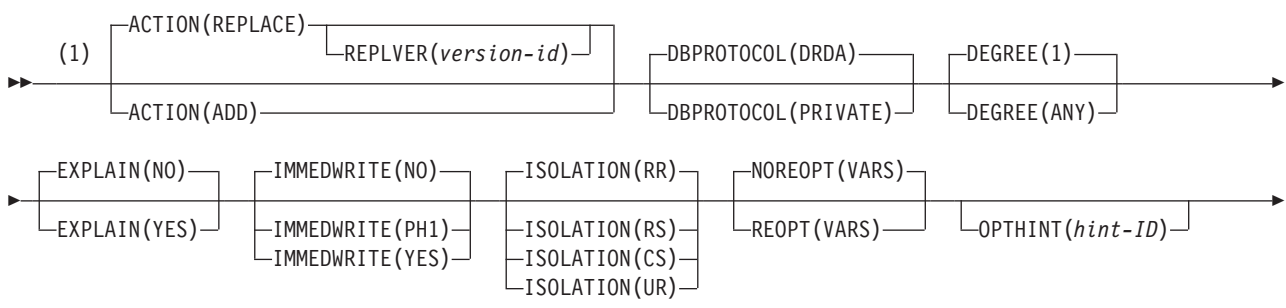


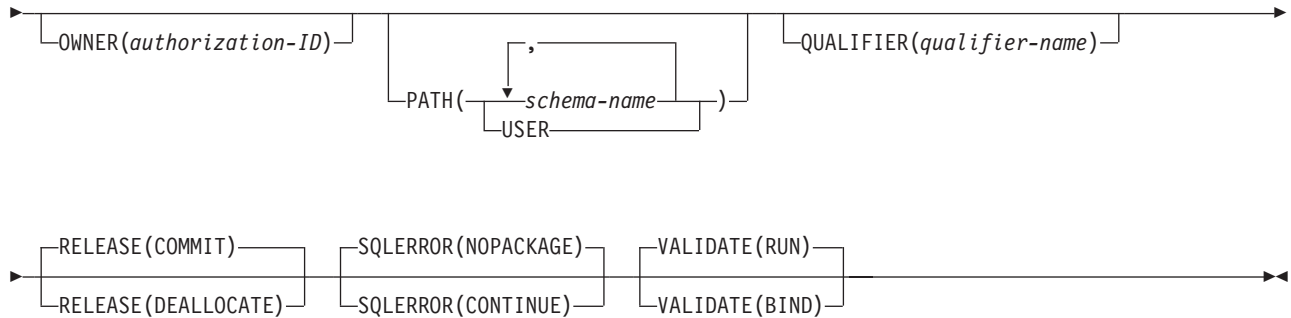


**options-string:**



**DB2 for z/OS options:**

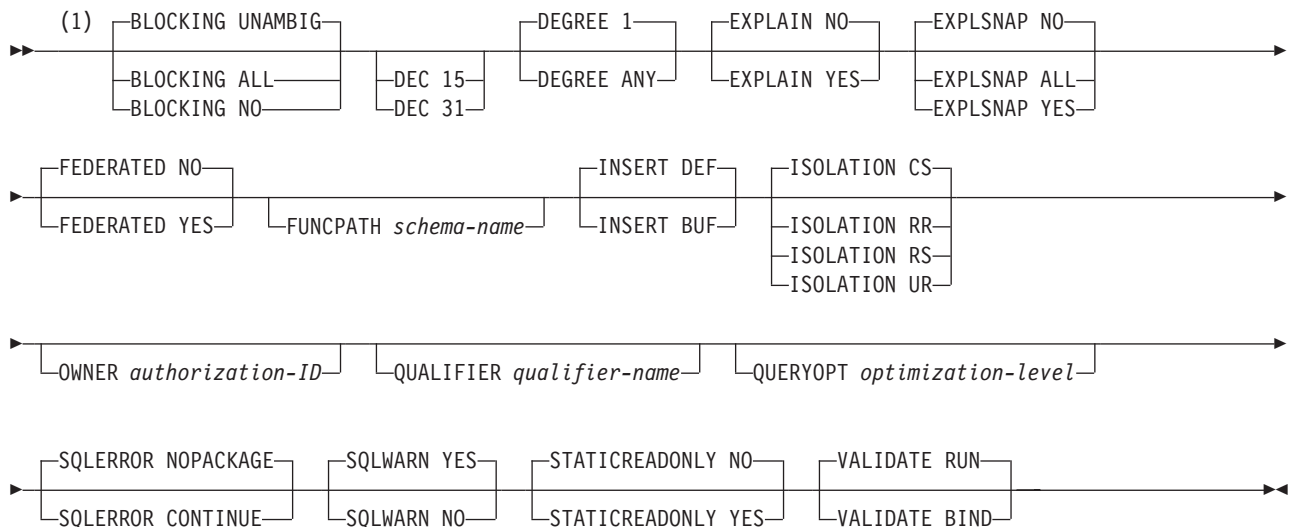




注:

- 1 これらのオプションは、どのような順序で指定してもかまいません。

### DB2 Database for Linux, UNIX, and Windows options



注:

- 1 これらのオプションは、どのような順序で指定してもかまいません。

## コマンド・パラメーター

### -help

db2sqljbind はそれがサポートする各オプションについて記述することを指定します。-help と共に他のオプションが指定されても、それらは無視されます。

### -url

プロファイルがカスタマイズされるデータ・ソースの URL を指定します。-automaticbind オプションまたは -onlinecheck オプションが YES に指定されているか、デフォルトで YES になった場合、この URL が表すデータ・ソースへの接続が確立されます。-url 値の可変部は、以下のとおりです。

### server

データベース・サーバーが存在するオペレーティング・システムのドメイン名または IP アドレス。

**port**

データベース・サーバーに割り当てられている TCP/IP サーバーのポート番号。デフォルトは 446 です。

**database**

プロファイルがカスタマイズされるデータベース・サーバーの名前。

接続を DB2 for z/OS サーバーに対して行う場合、*database* は、インストール中に定義される DB2 ロケーション名です。この値の中の文字はすべて大文字でなければなりません。サーバーで以下の SQL ステートメントを実行することにより、ロケーション名を判別できます。

```
SELECT CURRENT SERVER FROM SYSIBM.SYSDUMMY1;
```

接続を DB2 Database for Linux, UNIX, and Windows サーバーに対して行う場合、*database* は、インストール中に定義されるデータベース名です。

接続を IBM Cloudscape サーバーに対して行う場合、*database* は、データベースを含むファイルの完全修飾名です。この名前は、二重引用符 (") で囲まなければなりません。以下に例を示します。

```
"c:/databases/testdb"
```

```
property=value;
```

JDBC 接続のプロパティ。

**-user user-ID**

パッケージのバインドのためにデータ・ソースに接続するときに使うユーザー ID を指定します。

**-password password**

パッケージのバインドのためにデータ・ソースに接続するときに使うパスワードを指定します。

**-bindoptions options-string**

スペースで区切ったオプションのリストを指定します。それらのオプションには、同名の DB2 プリコンパイル・オプションおよびバインド・オプションと同じ機能があります。DB2 for z/OS システム上で実行されるプログラムを準備する場合、DB2 for z/OS のオプションを指定します。DB2 Database for Linux, UNIX, and Windows システム上で実行されるプログラムを準備する場合、DB2 Database for Linux, UNIX, and Windows のオプションを指定します。

**バインド・オプションに関する注意事項:**

- 以下の条件が該当する場合にのみ **VERSION** を指定してください。
  - パッケージを DB2 Database for Linux, UNIX, and Windows システムでバインドしている場合、システムがバージョン 8 以降である。
  - 関連したパッケージを新規の **VERSION** 値にバインドする前にプログラム上で変換プログラムを再実行する。
- **STATICREADONLY** をサポートするサーバーの場合は、**STATICREADONLY** の値は **YES**、その他のサーバーの場合は、**NO** となります。  
**STATICREADONLY YES** を指定すると、DB2 は未確定カーソルを読み取り専用カーソルであるものとして処理します。イテレーター宣言のエラーのトラブルシューティングの場合、**STATICREADONLY NO** を明示的に指定する必要があります。あるいは、確定済みになるようにイテレーターを宣言しま

す。たとえば、イテレーターを確定的に更新可能にする場合は、`sqlj.runtime.ForUpdate` をインプリメントするようにイテレーターを宣言します。イテレーターを読み取り専用にする場合、イテレーターを使用する `SELECT` ステートメントに `FOR READ ONLY` 節を付け加えます。

**重要:** パッケージのバインディングを行う場所であるデータ・ソースに適したプログラム準備オプションのみを指定してください。IBM Data Server Driver for JDBC and SQLJ の一部の値およびデフォルト値は、DB2 用の値およびデフォルト値とは異なります。

#### **-staticpositioned NOIYES**

イテレーターを使用する位置指定 `UPDATE` ステートメントと同じソース・ファイルに宣言されているイテレーターの場合、静的にバインドされたステートメントとしてその位置指定 `UPDATE` を実行するかどうかを指定します。デフォルトは `NO` です。`NO` は、動的に準備されるステートメントとして位置指定 `UPDATE` を実行することを意味します。この値は、シリアライズド・プロファイルの以前の `db2sqljcustomize` 呼び出しのための `-staticpositioned` 値と同じでなければなりません。

#### **-genDBRM**

`db2sqljbind` がシリアライズド・プロファイルからデータベース要求モジュール (DBRM) を生成すること、および `db2sqljbind` がリモート・バインド操作を行わないことを指定します。

`-genDBRM` は、DB2 for z/OS データベース・サーバーで実行されるプログラムにのみ適用されます。

#### **-DBRMDir *directory-name***

`-genDBRM` を指定する場合、`-DBRMDir` は生成された DBRM ファイルを `db2sqljbind` が配置するローカル・ディレクトリーを指定します。デフォルトは現行ディレクトリーです。

`-DBRMdir` は、DB2 for z/OS データベース・サーバーで実行されるプログラムにのみ適用されます。

#### **-tracefile *file-name***

トレースを使用可能にし、トレース情報のための出力ファイルを示します。このオプションを指定できるのは、IBM ソフトウェア・サポートから指示された場合のみです。

#### **-tracelevel**

`-tracefile` を指定する場合、`db2sqljcustomize` の実行中に何をトレースするかを指示します。デフォルトは `TRACE_SQLJ` です。このオプションを指定できるのは、IBM ソフトウェア・サポートから指示された場合のみです。

#### ***serialized-profile-name***

パッケージのバインド元である 1 つ以上のシリアライズド・プロファイルの名前を指定します。直列化されたプロファイル名の形式は、以下のようになります。

`program-name_SJProfileIDNumber.ser`

*program-name* は、拡張子 *.sqlj* の付いていない SQLJ ソース・プログラムの名前です。*n* は、0 から *m-1* までの整数です。ただし、*m* は、SQLJ ソース・プログラムから SQLJ 変換プログラムが生成したシリアルライズド・プロファイルの数です。

単一の DB2 パッケージを複数のシリアルライズド・プロファイルからバインドするために複数のシリアルライズド・プロファイル名を指定する場合、`db2sqljcustomize` を実行したときに、同じシリアルライズド・プロファイル名を同じ順序で指定している必要があります。

## 例

```
db2sqljbind -user richler -password mordecai
            -url jdbc:db2://server:50000/sample -bindoptions "EXPLAIN YES"
            pgmname_SJProfile0.ser
```

## 使用上の注意

**db2sqljbind** によって作成されるパッケージ名: `db2sqljbind` によって作成されるパッケージの名前は、`db2sqljcustomize` を実行したときに `-rootpkgname` または `-singlepkgname` パラメーターを使用して指定された名前です。 `-rootpkgname` または `-singlepkgname` を指定しなかった場合、パッケージ名は分離レベル文字を付加したプロファイル名の最初の 7 バイトです。

**db2sqljbind** の **DYNAMICRULES** 値: **DYNAMICRULES** バインド・オプションによって、DB2 パッケージの実行時の属性数が決まります。この属性のうちの 2 つは、許可を検査するときに使用される許可 ID と、非修飾オブジェクト用に使用される修飾子です。SQLJ プログラムで動的に実行される位置指定 UPDATE および DELETE ステートメントの許可を正しいものにするために、`db2sqljbind` は常に DB2 パッケージを **DYNAMICRULES(BIND)** オプションにバインドします。このオプションは変更できません。 **DYNAMICRULES(BIND)** オプションを指定すると、**SET CURRENT SQLID** ステートメントおよび **SET CURRENT SCHEMA** ステートメントは SQLJ プログラムに対して無効になります。なぜなら、これらのステートメントは **BIND** 以外の **DYNAMICRULES** 値を使ってバインドされている動的ステートメントにのみ影響を与えるからです。

**DYNAMICRULES(BIND)** を指定する場合、動的 SQL ステートメント内の非修飾の表、ビュー、索引、および別名はバインド・オプション **QUALIFIER** の値を使って暗黙的に修飾されます。 **QUALIFIER** を指定しない場合、DB2 はパッケージ所有者の許可 ID を暗黙的な修飾子として使用します。この動作がご使用のプログラムに適していない場合、以下の手法のいずれかを使用して正しい修飾子を設定できます。

- 位置指定 UPDATE および DELETE ステートメントを静的に強制実行します。位置指定 UPDATE または DELETE ステートメントのカーソル (イテレーター) が位置指定 UPDATE または DELETE ステートメントと同じパッケージ内にある場合、`db2sqljcustomize` または `db2sqljbind` の `-staticpositioned YES` オプションを使用してこの強制実行を行えます。
- 位置指定 UPDATE および位置指定 DELETE ステートメント内の DB2 表名を完全修飾します。

## db2sqljprint - SQLJ プロファイル・プリンター

db2sqljprint は、プロファイルのカスタマイズ済みバージョンの内容をプレーン・テキストとして印刷します。

### 許可

なし

### コマンド構文

▶▶—db2sqljprint—*profilename*————▶▶

### コマンド・パラメーター

*profilename*

SQLJ プロファイル・ファイルの相対名または絶対名を指定します。SQLJ ファイルが Java ソース・ファイルに変換されると、そこに含まれる SQL 操作に関する情報は、プロファイルと呼ばれる SQLJ が生成したリソース・ファイルに保管されます。プロファイルは、オリジナルの入力ファイルの名前の後に続く接尾部 *\_SJProfileN* (N は整数) で識別されます。その拡張子は *.ser* です。プロファイル名を指定する際には、*.ser* 拡張子を付けても付けなくてもかまいません。

### 例

```
db2sqljprint pgmname_SJProfile0.ser
```





---

## 付録 A. DB2 技術情報の概説

DB2 技術情報は、以下のツールと方法を介して利用できます。

- DB2 インフォメーション・センター
  - トピック (タスク、概念、およびリファレンス・トピック)
  - DB2 ツールのヘルプ
  - サンプル・プログラム
  - チュートリアル
- DB2 資料
  - PDF ファイル (ダウンロード可能)
  - PDF ファイル (DB2 PDF DVD に含まれる)
  - 印刷資料
- コマンド行ヘルプ
  - コマンド・ヘルプ
  - メッセージ・ヘルプ

**注:** DB2 インフォメーション・センターのトピックは、PDF やハードコピー資料よりも頻繁に更新されます。最新の情報を入手するには、資料の更新が発行されたときにそれをインストールするか、[ibm.com](http://ibm.com)<sup>®</sup> にある DB2 インフォメーション・センターを参照してください。

技術資料、ホワイト・ペーパー、IBM Redbooks<sup>®</sup> 資料などのその他の DB2 技術情報には、オンライン ([ibm.com](http://ibm.com)) でアクセスできます。DB2 Information Management ソフトウェア・ライブラリー・サイト (<http://www.ibm.com/software/data/sw-library/>) にアクセスしてください。

### 資料についてのフィードバック

DB2 の資料についてのお客様からの貴重なご意見をお待ちしています。DB2 の資料を改善するための提案については、[db2docs@ca.ibm.com](mailto:db2docs@ca.ibm.com) まで E メールを送信してください。DB2 の資料チームは、お客様からのフィードバックすべてに目を通しますが、直接お客様に返答することはありません。お客様が関心をお持ちの内容について、可能な限り具体的な例を提供してください。特定のトピックまたはヘルプ・ファイルについてのフィードバックを提供する場合は、そのトピック・タイトルおよび URL を含めてください。

DB2 お客様サポートに連絡する場合には、この E メール・アドレスを使用しないでください。資料を参照しても、DB2 の技術的な問題が解決しない場合は、お近くの IBM サービス・センターにお問い合わせください。

## DB2 テクニカル・ライブラリー (ハードコピーまたは PDF 形式)

以下の表は、DB2 ライブラリーについて説明しています。DB2 ライブラリーに関する詳細な説明については、[www.ibm.com/shop/publications/order](http://www.ibm.com/shop/publications/order) にある IBM Publications Center にアクセスしてください。英語の DB2 バージョン 9.5 のマニュアル (PDF 形式) とその翻訳版は、[www.ibm.com/support/docview.wss?rs=71&uid=swg2700947](http://www.ibm.com/support/docview.wss?rs=71&uid=swg2700947) からダウンロードできます。

この表には印刷資料が入手可能かどうかを示されていますが、国または地域によっては入手できない場合があります。

表 96. DB2 の技術情報

Name	資料番号	印刷資料が入手可能かどうか
管理 API リファレンス	SC88-4431-00	はい
管理ルーチンおよびビュー	SC88-4435-00	いいえ
コール・レベル・インターフェース ガイドおよびリファレンス 第 1 巻	SC88-4433-00	はい
コール・レベル・インターフェース ガイドおよびリファレンス 第 2 巻	SC88-4434-00	はい
コマンド・リファレンス	SC88-4432-00	はい
データ移動ユーティリティガイドおよびリファレンス	SC88-4421-00	はい
データ・リカバリーと高可用性ガイドおよびリファレンス	SC88-4423-00	はい
データ・サーバー、データベース、およびデータベース・オブジェクトのガイド	SC88-4259-00	はい
データベース・セキュリティ・ガイド	SC88-4418-00	はい
ADO.NET および OLE DB アプリケーションの開発	SC88-4425-00	はい
組み込み SQL アプリケーションの開発	SC88-4426-00	はい
Java アプリケーションの開発	SC88-4427-00	はい
Perl および PHP アプリケーションの開発	SC88-4428-00	いいえ
SQL および 外部ルーチンの開発	SC88-4429-00	はい
データベース・アプリケーション開発の基礎	GC88-4430-00	はい
DB2 インストールおよび管理概説 (Linux および Windows 版)	GC88-4439-00	はい
国際化対応ガイド	SC88-4420-00	はい

表 96. DB2 の技術情報 (続き)

Name	資料番号	印刷資料が入手可能かどうか
メッセージ・リファレンス 第 1 巻	GI88-4109-00	いいえ
メッセージ・リファレンス 第 2 巻	GI88-4110-00	いいえ
マイグレーション・ガイド	GC88-4438-00	はい
Net Search Extender 管理および ユーザーズ・ガイド	SC88-4630-00	はい
注: この資料の内容は、DB2 イ ンフォメーション・センターに は含まれていません。		
パーティションおよびクラスタ リングのガイド	SC88-4419-00	はい
Query Patroller 管理およびユー ザーズ・ガイド	SC88-4611-00	はい
IBM データ・サーバー・クライ アント機能 概説およびインス トール	GC88-4441-00	いいえ
DB2 サーバー機能 概説および インストール	GC88-4440-00	はい
Spatial Extender and Geodetic Data Management Feature ユー ザーズ・ガイドおよびリファレ ンス	SC88-4629-00	はい
SQL リファレンス 第 1 巻	SC88-4436-00	はい
SQL リファレンス 第 2 巻	SC88-4437-00	はい
システム・モニター ガイドお よびリファレンス	SC88-4422-00	はい
テキスト検索ガイド	SC88-4424-00	はい
問題判別ガイド	GI88-4108-00	いいえ
データベース・パフォーマンス のチューニング	SC88-4417-00	はい
Visual Explain チュートリアル	SC88-4449-00	いいえ
新機能	SC88-4445-00	はい
ワークロード・マネージャー ガイドおよびリファレンス	SC88-4446-00	はい
pureXML ガイド	SC88-4447-00	はい
XQuery リファレンス	SC88-4448-00	いいえ

表 97. DB2 Connect 固有の技術情報

Name	資料番号	印刷資料が入手可能かどうか
DB2 Connect Personal Edition 概説およびインストール	GC88-4443-00	はい

表 97. DB2 Connect 固有の技術情報 (続き)

Name	資料番号	印刷資料が入手可能かどうか
DB2 Connect サーバー機能 概説およびインストール	GC88-4444-00	はい
DB2 Connect ユーザーズ・ガイド	SC88-4442-00	はい

表 98. WebSphere Information Integration の技術情報

Name	資料番号	印刷資料が入手可能かどうか
WebSphere Information Integration: Administration Guide for Federated Systems	SC19-1020-01	はい
WebSphere Information Integration: Replication および Event Publishing ASNCLP プログラム・リファレンス	SC88-4167-01	はい
WebSphere Information Integration: Configuration Guide for Federated Data Sources	SC19-1034-01	いいえ
WebSphere Information Integration: SQL Replication Guide and Reference	SC19-1030-01	はい

## DB2 の印刷資料の注文方法

DB2 の印刷資料が必要な場合、オンラインで購入することができますが、すべての国および地域で購入できるわけではありません。DB2 の印刷資料については、IBM 営業担当員にお問い合わせください。DB2 PDF ドキュメンテーション DVD の一部のソフトコピー・ブックは、印刷資料では入手できないことに留意してください。例えば、「DB2 メッセージ・リファレンス」はどちらの巻も印刷資料としては入手できません。

DB2 PDF ドキュメンテーション DVD で利用できる DB2 の印刷資料の大半は、IBM に有償で注文することができます。国または地域によっては、資料を IBM Publications Center からオンラインで注文することもできます。お客様の国または地域でオンライン注文が利用できない場合、DB2 の印刷資料については、IBM 営業担当員にお問い合わせください。DB2 PDF ドキュメンテーション DVD に収録されている資料の中には、印刷資料として提供されていないものもあります。

注: 最新で完全な DB2 資料は、DB2 インフォメーション・センター (<http://publib.boulder.ibm.com/infocenter/db2luw/v9r5>) で参照することができます。

DB2 の印刷資料は以下の方法で注文することができます。

- 日本 IBM 発行のマニュアルはインターネット経由でご購入いただけます。詳しくは <http://www.ibm.com/shop/publications/order> の「ご注文について」をご覧ください。

さい。資料の注文情報にアクセスするには、お客様の国、地域、または言語を選択してください。その後、各ロケーションにおける注文についての指示に従ってください。

- DB2 の印刷資料を IBM 営業担当員に注文するには、以下のようになります。
  1. 以下の Web サイトのいずれかから、営業担当員の連絡先情報を見つけてください。
    - IBM Directory of world wide contacts ([www.ibm.com/planetwide](http://www.ibm.com/planetwide))
    - IBM Publications Web サイト (<http://www.ibm.com/shop/publications/order>)  
国、地域、または言語を選択し、お客様の所在地に該当する Publications ホーム・ページにアクセスしてください。このページから、「このサイトについて」のリンクにアクセスしてください。
  2. 電話をご利用の場合は、DB2 資料の注文であることをご指定ください。
  3. 担当者に、注文する資料のタイトルと資料番号をお伝えください。タイトルと資料番号は、456 ページの『DB2 テクニカル・ライブラリー (ハードコピーまたは PDF 形式)』でご確認いただけます。

---

## コマンド行プロセッサから SQL 状態ヘルプを表示する

DB2 は、SQL ステートメントの結果の原因になったと考えられる条件の SQLSTATE 値を戻します。SQLSTATE ヘルプは、SQL 状態および SQL 状態クラス・コードの意味を説明します。

SQL 状態ヘルプを呼び出すには、コマンド行プロセッサを開いて以下のように入力します。

```
? sqlstate or ? class code
```

ここで、*sqlstate* は有効な 5 桁の SQL 状態を、*class code* は SQL 状態の最初の 2 桁を表します。

例えば、? 08003 を指定すると SQL 状態 08003 のヘルプが表示され、? 08 を指定するとクラス・コード 08 のヘルプが表示されます。

---

## 異なるバージョンの DB2 インフォメーション・センターへのアクセス

DB2 バージョン 9.5 のトピックを扱っている DB2 インフォメーション・センターの URL は、<http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/>です。

DB2 バージョン 9 のトピックを扱っている DB2 インフォメーション・センターの URL は <http://publib.boulder.ibm.com/infocenter/db2luw/v9/>です。

DB2 バージョン 8 のトピックについては、バージョン 8 のインフォメーション・センターの URL <http://publib.boulder.ibm.com/infocenter/db2luw/v8/>にアクセスしてください。

## DB2 インフォメーション・センターにおける特定の言語でのトピックの表示

DB2 インフォメーション・センターでは、ブラウザの設定で指定した言語でのトピックの表示が試みられます。トピックがその指定言語に翻訳されていない場合は、DB2 インフォメーション・センターでは英語でトピックが表示されます。

- Internet Explorer Web ブラウザーで、指定どおりの言語でトピックを表示するには、以下のようにします。
  1. Internet Explorer の「ツール」 -> 「インターネット オプション」 -> 「言語 ...」 ボタンをクリックします。「言語の優先順位」ウィンドウがオープンします。
  2. 該当する言語が、言語リストの先頭の項目に指定されていることを確認します。
    - リストに新しい言語を追加するには、「追加...」 ボタンをクリックします。

注: 言語を追加しても、特定の言語でトピックを表示するのに必要なフォントがコンピューターに備えられているとはかぎりません。

    - リストの先頭に新しい言語を移動するには、その言語を選択してから、その言語が言語リストに先頭に行くまで「上に移動」 ボタンをクリックします。
  3. ブラウザー・キャッシュを消去してから、ページをリフレッシュし、使用する言語で DB2 インフォメーション・センターを表示します。
- Firefox または Mozilla Web ブラウザーの場合に、使いたい言語でトピックを表示するには、以下のようにします。
  1. 「ツール」 -> 「オプション」 -> 「詳細」 ダイアログの「言語」セクションにあるボタンを選択します。「設定」ウィンドウに「言語」パネルが表示されます。
  2. 該当する言語が、言語リストの先頭の項目に指定されていることを確認します。
    - リストに新しい言語を追加するには、「追加...」 ボタンをクリックしてから、「言語を追加」ウィンドウで言語を選択します。
    - リストの先頭に新しい言語を移動するには、その言語を選択してから、その言語が言語リストに先頭に行くまで「上に移動」 ボタンをクリックします。
  3. ブラウザー・キャッシュを消去してから、ページをリフレッシュし、使用する言語で DB2 インフォメーション・センターを表示します。

ブラウザとオペレーティング・システムの組み合わせによっては、オペレーティング・システムの地域の設定も希望のロケールと言語に変更しなければならない場合があります。



---

## コンピューターまたはイントラネット・サーバーにインストールされた DB2 インフォメーション・センターの更新

DB2 インフォメーション・センターをローカルにインストールしている場合は、IBM から提供される更新をダウンロードおよびインストールすることができます。

ローカルにインストールされた DB2 インフォメーション・センターを更新するには、以下のことを行う必要があります。

1. コンピューター上の DB2 インフォメーション・センターを停止し、インフォメーション・センターをスタンドアロン・モードで再始動します。インフォメーション・センターをスタンドアロン・モードで実行すると、ネットワーク上の他のユーザーがそのインフォメーション・センターにアクセスできなくなります。これで、更新をダウンロードして適用できるようになります。
2. 「更新」機能を使用することにより、どんな更新が利用できるかを確認します。インストールする更新がある場合は、「更新」機能を使用してそれをダウンロードおよびインストールできます。

**注:** ご使用の環境において、インターネットに接続されていないマシンに DB2 インフォメーション・センターの更新をインストールする必要がある場合は、インターネットに接続されていて DB2 インフォメーション・センターがインストールされているマシンを使用して、更新サイトをローカル・ファイル・システムにミラーリングする必要があります。ネットワーク上の多数のユーザーが資料の更新をインストールする場合にも、更新サイトをローカルにミラーリングして、更新サイト用のプロキシーを作成することにより、個々のユーザーが更新を実行するのに要する時間を短縮できます。

更新パッケージが入手可能な場合、「更新」機能を使用してパッケージをダウンロードします。ただし、「更新」機能は、スタンドアロン・モードでのみ使用できます。

3. スタンドアロンのインフォメーション・センターを停止し、コンピューター上の DB2 インフォメーション・センターを再開します。

**注:** Windows Vista の場合、下記のコマンドは管理者として実行する必要があります。完全な管理者特権でコマンド・プロンプトまたはグラフィカル・ツールを起動するには、ショートカットを右クリックしてから、「管理者として実行」を選択します。

コンピューターまたはイントラネット・サーバーにインストール済みの DB2 インフォメーション・センターを更新するには、以下のようになります。

1. DB2 インフォメーション・センターを停止します。
  - Windows では、「スタート」→「コントロール パネル」→「管理ツール」→「サービス」をクリックします。次に、「DB2 インフォメーション・センター」サービスを右クリックして「停止」を選択します。
  - Linux では、以下のコマンドを入力します。

```
/etc/init.d/db2icdv95 stop
```
2. インフォメーション・センターをスタンドアロン・モードで開始します。
  - Windows の場合:
    - a. コマンド・ウィンドウを開きます。

- b. インフォメーション・センターがインストールされているパスにナビゲートします。デフォルトでは、DB2 インフォメーション・センターは <Program Files>¥IBM¥DB2 Information Center¥Version 9.5 ディレクトリーにインストールされています (<Program Files> は「Program Files」ディレクトリーのロケーション)。
- c. インストール・ディレクトリーの doc¥bin ディレクトリーにナビゲートします。
- d. 次のように help\_start.bat ファイルを実行します。

```
help_start.bat
```

• Linux の場合:

- a. インフォメーション・センターがインストールされているパスにナビゲートします。デフォルトでは、DB2 インフォメーション・センターは /opt/ibm/db2ic/V9.5 ディレクトリーにインストールされています。
- b. インストール・ディレクトリーの doc/bin ディレクトリーにナビゲートします。
- c. 次のように help\_start スクリプトを実行します。

```
help_start
```

システムのデフォルト Web ブラウザーが起動し、スタンドアロンのインフォメーション・センターが表示されます。

3. 「更新」ボタン (🔄) をクリックします。インフォメーション・センターの右側のパネルで、「更新の検索 (Find Updates)」をクリックします。既存の文書に対する更新のリストが表示されます。
4. ダウンロード・プロセスを開始するには、ダウンロードする更新をチェックして選択し、「更新のインストール (Install Updates)」をクリックします。
5. ダウンロードおよびインストール・プロセスが完了したら、「完了」をクリックします。
6. スタンドアロンのインフォメーション・センターを停止します。

- Windows の場合は、インストール・ディレクトリーの doc¥bin ディレクトリーにナビゲートしてから、次のように help\_end.bat ファイルを実行します。

```
help_end.bat
```

注: help\_end バッチ・ファイルには、help\_start バッチ・ファイルを使用して開始したプロセスを安全に終了するのに必要なコマンドが含まれています。Ctrl-C または他の方法を使用して、help\_start.bat を終了しないでください。

- Linux の場合は、インストール・ディレクトリーの doc/bin ディレクトリーにナビゲートしてから、次のように help\_end スクリプトを実行します。

```
help_end
```

注: help\_end スクリプトには、help\_start スクリプトを使用して開始したプロセスを安全に終了するのに必要なコマンドが含まれています。他の方法を使用して、help\_start スクリプトを終了しないでください。

7. DB2 インフォメーション・センターを再開します。

- Windows では、「スタート」→「コントロール パネル」→「管理ツール」→「サービス」をクリックします。次に、「DB2 インフォメーション・センター」サービスを右クリックして「開始」を選択します。
- Linux では、以下のコマンドを入力します。

```
/etc/init.d/db2icdv95 start
```

更新された DB2 インフォメーション・センターに、更新された新しいトピックが表示されます。

---

## DB2 チュートリアル

DB2 チュートリアルは、DB2 製品のさまざまな機能について学習するのを支援します。この演習をとおして段階的に学習することができます。

### はじめに

インフォメーション・センター (<http://publib.boulder.ibm.com/infocenter/db2help/>) から、このチュートリアルの XHTML 版を表示できます。

演習の中で、サンプル・データまたはサンプル・コードを使用する場合があります。個々のタスクの前提条件については、チュートリアルを参照してください。

### DB2 チュートリアル

チュートリアルを表示するには、タイトルをクリックします。

#### 「*pureXML* ガイド」の『**pureXML™**』

XML データを保管し、ネイティブ XML データ・ストアに対して基本的な操作を実行できるように、DB2 データベースをセットアップします。

#### 「*Visual Explain* チュートリアル」の『**Visual Explain**』

Visual Explain を使用して、パフォーマンスを向上させるために SQL ステートメントを分析し、最適化し、調整します。

---

## DB2 トラブルシューティング情報

DB2 製品を使用する際に役立つ、トラブルシューティングおよび問題判別に関する広範囲な情報を利用できます。

### DB2 ドキュメンテーション

トラブルシューティング情報は、DB2 問題判別ガイド、または DB2 インフォメーション・センターの「サポートおよびトラブルシューティング」セクションにあります。ここでは、DB2 診断ツールおよびユーティリティーを使用して、問題を切り分けて識別する方法、最も頻繁に起こる幾つかの問題に対するソリューションについての情報、および DB2 製品を使用する際に発生する可能性のある問題の解決方法についての他のアドバイスがあります。

### DB2 Technical Support の Web サイト

現在問題が発生していて、考えられる原因とソリューションを検索したい場合は、DB2 Technical Support の Web サイトを参照してください。

Technical Support サイトには、最新の DB2 資料、TechNotes、プログラム

診断依頼書 (APAR またはバグ修正)、フィックスパック、およびその他のリソースへのリンクが用意されています。この知識ベースを活用して、問題に対する有効なソリューションを探し出すことができます。

DB2 Technical Support の Web サイト (<http://www.ibm.com/software/data/db2/udb/support.html>) にアクセスしてください。

---

## ご利用条件

これらの資料は、以下の条件に同意していただける場合に限りご使用いただけます。

**個人使用:** これらの資料は、すべての著作権表示その他の所有権表示をしていただくことを条件に、非商業的な個人による使用目的に限り複製することができます。ただし、IBM の明示的な承諾をえずに、これらの資料またはその一部について、二次的著作物を作成したり、配布 (頒布、送信を含む) または表示 (上映を含む) することはできません。

**商業的使用:** これらの資料は、すべての著作権表示その他の所有権表示をしていただくことを条件に、お客様の企業内に限り、複製、配布、および表示することができます。ただし、IBM の明示的な承諾をえずにこれらの資料の二次的著作物を作成したり、お客様の企業外で資料またはその一部を複製、配布、または表示することはできません。

ここで明示的に許可されているもの以外に、資料や資料内に含まれる情報、データ、ソフトウェア、またはその他の知的所有権に対するいかなる許可、ライセンス、または権利を明示的にも黙示的にも付与するものではありません。

資料の使用が IBM の利益を損なうと判断された場合や、上記の条件が適切に守られていないと判断された場合、IBM はいつでも自らの判断により、ここで与えた許可を撤回できるものとさせていただきます。

お客様がこの情報をダウンロード、輸出、または再輸出する際には、米国のすべての輸出入関連法規を含む、すべての関連法規を遵守するものとします。

IBM は、これらの資料の内容についていかなる保証もしません。これらの資料は、特定物として現存するままの状態を提供され、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任なしで提供されます。

---

## 付録 B. 特記事項

本書は米国 IBM が提供する製品およびサービスについて作成したものです。

本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒106-8711  
東京都港区六本木 3-2-12  
日本アイ・ビー・エム株式会社  
法務・知的財産  
知的財産権ライセンス渉外

**以下の保証は、国または地域の法律に沿わない場合は、適用されません。** IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態を提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書は、IBM 以外の Web サイトおよびリソースへのリンクまたは参照を含む場合があります。IBM は、本書より参照もしくはアクセスできる、または本書からリンクされた IBM 以外の Web サイトもしくは第三者のリソースに対して一切の責任を負いません。IBM 以外の Web サイトにリンクが張られていることにより IBM が当該 Web サイトを推奨するものではなく、またその内容、使用もしくはサイトの所有者について IBM が責任を負うことを意味するものではありません。また、IBM は、お客様が IBM Web サイトから第三者の存在を知ることになった場合にも (もしくは、IBM Web サイトから第三者へのリンクを使用した場合にも)、お客様と第三者との間のいかなる取引に対しても一切責任を負いません。従って、お客様は、IBM が上記の外部サイトまたはリソースの利用について責任を負うものではなく、また、外部サイトまたはリソースからアクセス可能なコンテンツ、サービス、



製品、またはその他の資料一切に対して IBM が責任を負うものではないことを承諾し、同意するものとします。第三者により提供されるソフトウェアには、そのソフトウェアと共に提供される固有の使用条件が適用されます。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム (本プログラムを含む) との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

IBM Canada Limited  
Office of the Lab Director  
8200 Warden Avenue  
Markham, Ontario  
L6G 1C7  
CANADA

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性がありますが、その測定値が、一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確認できません。IBM 以外の製品の性能に関する質問は、それらの製品の供給者をお願いします。

IBM の将来の方向または意向に関する記述については、予告なしに変更または撤回される場合があり、単に目標を示しているものです。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

著作権使用許諾:

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、利便性もしくは機能性があることをほのめかしたり、保証することはできません。

それぞれの複製物、サンプル・プログラムのいかなる部分、またはすべての派生した創作物には、次のように、著作権表示を入れていただく必要があります。

© (お客様の会社名) (西暦年). このコードの一部は、IBM Corp. のサンプル・プログラムから取られています。 © Copyright IBM Corp. \_年を入れる\_. All rights reserved.

## 商標

DB2 バージョン 9.5 ドキュメンテーション・ライブラリーの資料に記載されている会社名、製品名、またはサービス名は、IBM Corporation の商標である可能性があります。IBM Corporation の商標については、<http://www.ibm.com/legal/copytrade.shtml> を参照してください。

以下は、それぞれ各社の商標または登録商標です。

Microsoft<sup>®</sup>、Windows、Windows NT<sup>®</sup>、および Windows ロゴは、Microsoft Corporation の米国およびその他の国における商標です。

Intel<sup>®</sup>、Intel ロゴ、Intel Inside<sup>®</sup> ロゴ、Intel Centrino<sup>®</sup>、Intel Centrino ロゴ、Celeron<sup>®</sup>、Intel Xeon<sup>®</sup>、Intel SpeedStep<sup>®</sup>、Itanium<sup>®</sup> および Pentium<sup>®</sup> は、Intel Corporation の米国およびその他の国における商標です。

Java およびすべての Java 関連の商標およびロゴは、Sun Microsystems, Inc. の米国およびその他の国における商標です。

UNIX は、The Open Group の米国およびその他の国における登録商標です。

Linux は、Linus Torvalds の米国およびその他の国における商標です。

Adobe<sup>®</sup>、Adobe ロゴ、PostScript<sup>®</sup>、PostScript ロゴは、Adobe Systems Incorporated の米国およびその他の国における登録商標または商標です。

他の会社名、製品名およびサービス名等はそれぞれ各社の商標です。





# 索引

日本語、数字、英字、特殊文字の順に配列されています。なお、濁音と半濁音は清音と同等に扱われています。

## [ア行]

### アプリケーション

Java 2 Platform, Enterprise Edition によるサポート 219

アプリケーション・プログラミング、JDBC 27

アプリケーション・プログラミング、SQLJ 115

### アプレット

使用するためのポイント 197

JDBC の構築 193

SQLJ の構築 195

### 暗号化されたセキュリティ・センシティブ・データ

IBM Data Server Driver for JDBC and SQLJ 178

### 暗号化されたユーザー ID または暗号化されたパスワードのセキュリティ

IBM Data Server Driver for JDBC and SQLJ 178

### 位置指定 DELETE

SQLJ 128

### 位置指定 UPDATE

SQLJ 128

### 位置指定イテレーター 140

結果セット・イテレーター 140

変数として渡される 132

### イテレーター

位置指定 DELETE 128

位置指定 UPDATE 128

JDBC 結果セットの取得 153

### イテレーター宣言節

SQLJ 315

### イテレーター変換節

SQLJ 320

### 印刷資料

注文 458

### インストール

IBM Data Server Driver for JDBC and SQLJ 5

### インフォメーション・センター

更新 461

バージョン 459

別の言語で表示する 460

### インプリメント節

SQLJ 312

### エラー処理

SQLJ 170

### オブジェクトの作成

JDBC 45

### オブジェクトの変更

JDBC 45

### オプティミスティック・ロック

JDBC アプリケーション 86

### オンライン検査

カスタマイズが必要 434

最適化の向上のため 434

制約事項 434

## [カ行]

### 拡張クライアント情報

IBM Data Server Driver for JDBC and SQLJ 82

### 環境変数

JDBC 18

SQLJ 18

### クライアント情報プロパティ

IBM Data Server Driver for JDBC and SQLJ 83, 84

### クライアント・リルト・サポート

IBM Data Server Driver for JDBC and SQLJ 109

### クライアント・リルト・サポート、クライアント操作

IBM Data Server Driver for JDBC and SQLJ 111

### グローバル・プロパティ、IBM Data Server Driver for JDBC and SQLJ 277

### 警告

IBM Data Server Driver for JDBC and SQLJ 99

### 結果セット情報の取得

JDBC 55

### 結果セット・イテレーター

位置指定イテレーター 140

同じファイルでの定義と使用 138

個別のファイルでの public 宣言 138, 154

説明 137

宣言に関する制約事項 140

名前指定イテレーター 138

SQLJ での行の取得 137, 138, 140

### 更新

インフォメーション・センター 461

DB2 インフォメーション・センター 461

### 更新可能な ResultSet

行が挿入されたかどうかのテスト 63

行の挿入 61

削除ホールになっているかどうかのテスト 61

JDBC 56, 57

### 構成

JDBC 18

SQLJ 18

### 構成パラメーターによるトレース

IBM Data Server Driver for JDBC and SQLJ、例 205

### 構成プロパティ

パラメーター 18

### 構成プロパティ、IBM Data Server Driver for JDBC and SQLJ 277

コマンド  
DB2 SQLJ プロファイル・バインド・プログラム 447  
DB2 SQLJ 変換 プログラム 430  
db2sqljbind 447  
db2sqljprint 453  
sqlj 430  
コマンド、SQLJ プログラムの準備 430  
コミット  
トランザクション、JDBC 98  
SQLJ トランザクション 169  
コミットまたはロールバック 169  
ご利用条件  
資料の使用 464  
コンテキスト節  
SQLJ 317  
コンテナー  
Java 2 Platform, Enterprise Edition 220

## [サ行]

作成  
DB2 表、SQLJ 128  
作成とデプロイ  
DataSource オブジェクト 40  
参照情報、Java 239  
システム・モニター  
IBM Data Server Driver for JDBC and SQLJ 211  
実行可能節  
SQLJ 316  
実行コンテキスト 157  
実行制御  
SQLJ 157  
自動コミット、デフォルトの  
JDBC 98  
自動生成キー  
JDBC アプリケーションでの取得 79  
取得  
DB2 表のデータ、JDBC 52  
使用可能化  
リモート・トレース・コントローラー 214  
資料  
使用に関するご利用条件 464  
PDF および印刷資料 456  
資料の概説 455  
スクロール可能イテレーター  
SQLJ 145  
スクロール可能な ResultSet  
JDBC 56, 57  
ステートメント参照、SQLJ 311  
ストアド・プロシージャ  
既知の数の結果セットの取得、JDBC 66  
結果セットの検索 149  
結果セットを開いたままにしておく、JDBC 68  
複数の結果セットの検索、JDBC 66  
不明な数の結果セットの取得、JDBC 67  
呼び出し、SQLJ 149

ストアド・プロシージャ (続き)  
JDBC プログラムからの呼び出し  
ARRAY パラメーター 77  
SQLJ プログラムからの呼び出し  
ARRAY パラメーター 160  
ストアド・プロシージャの呼び出し  
CallableStatement 64  
セーブポイント  
JDBC アプリケーションでの使用 78  
SQLJ アプリケーションでの使用 161  
制限  
SQLJ 変数名 126  
セキュリティ  
プラグイン  
JDBC サポート 183  
DB2 JDBC Type 2 ドライバー 191  
IBM Data Server Driver for JDBC and SQLJ 173  
SQLJ プログラムの準備 188  
セキュリティ、Kerberos  
IBM Data Server Driver for JDBC and SQLJ 179  
セキュリティ、暗号化されたセキュリティ・センシティブ・データ  
IBM Data Server Driver for JDBC and SQLJ 178  
セキュリティ、暗号化されたユーザー ID セキュリティーまたは暗号化されたパスワード  
IBM Data Server Driver for JDBC and SQLJ 178  
セキュリティ、ユーザー ID およびパスワード  
IBM Data Server Driver for JDBC and SQLJ 175  
セキュリティ、ユーザー ID のみ  
IBM Data Server Driver for JDBC and SQLJ 177  
接続、既存の使用  
SQLJ 124  
接続コンセントレーター  
IBM Data Server Driver for JDBC and SQLJ 233  
接続コンテキスト  
オブジェクト 117  
クラス 117  
クローズ 171  
デフォルト 117  
接続宣言節  
SQLJ 314  
接続のクローズ  
重要性 114, 171  
接続プール  
概要 231  
宣言  
JDBC アプリケーションでの変数 44

## [タ行]

代入節  
SQLJ 320  
チュートリアル  
トラブルシューティングと問題判別 463  
Visual Explain 463

データの検索  
イテレーターの複数インスタンスの使用、SQLJ 145  
データベース表での複数のイテレーターの使用、SQLJ 143  
DB2 表からの、SQLJ 137  
データの取り出し、JDBC 52  
データベースの互換性  
IBM Data Server Driver for JDBC and SQLJ 3  
データ・ソース  
DriverManager を使用した接続 32  
JDBC の DataSource を使用した接続 37  
JDBC を使用した接続 29  
データ・ソース情報の取得  
JDBC 43  
データ・ソースへの接続  
DataSource インターフェース 37  
SQLJ 117  
データ・タイプ・マッピング  
Java、JDBC、およびデータベース 239  
デフォルトの接続コンテキスト 117  
SQLJ 124  
特殊タイプ  
JDBC アプリケーションでの使用 76  
SQLJ アプリケーションでの使用 159  
特記事項 465  
ドライバーのバージョン  
IBM Data Server Driver for JDBC and SQLJ 428  
ドライバー・サポートの比較  
JDBC API 281  
トラステッド・コンテキスト  
JDBC サポート 185  
トラブルシューティング  
オンライン情報 463  
チュートリアル 463  
トランザクション制御、JDBC 97  
トランザクション制御、SQLJ 169  
トレース、IBM Data Server Driver for JDBC and SQLJ  
リモート・コントローラー 214  
トレース・データの収集  
SQLJ 203  
トレース・プログラム  
IBM Data Server Driver for JDBC and SQLJ、例 206

## [ナ行]

名前指定イテレーター  
結果セット・イテレーター 138  
変数として渡される 132

## [ハ行]

パッケージへのアクセス  
JDBC 42  
SQLJ 125  
バッチ更新  
JDBC 48

バッチ更新 (続き)  
SQLJ 133  
バッチ照会  
JDBC 54  
パラメーター情報の取得  
JDBC 51  
複数行の操作 63  
複数の結果セット  
数が不明な場合の取得、JDBC 67  
数がわかっている場合の取得、JDBC 66  
検索、JDBC 66  
ストアド・プロシージャの検索 149  
開いたままにしておく、JDBC 68  
プログラムの準備  
Java 193  
プロパティ  
構成  
パラメーター 18  
IBM Data Server Driver for JDBC and SQLJ 245  
分散トランザクションの例  
JDBC 223  
分離レベル  
JDBC 97  
SQLJ 169  
ヘルプ  
表示 460  
SQL ステートメントの 459  
変更  
DB2 表、SQLJ 128  
ホスト式  
SQLJ 125, 311

## [マ行]

戻りコード  
IBM Data Server Driver for JDBC and SQLJ のエラー 419  
問題診断  
JDBC 203  
SQLJ 203  
問題判別  
オンライン情報 463  
チュートリアル 463

## [ヤ行]

ユーザー ID およびパスワード・セキュリティ  
IBM Data Server Driver for JDBC and SQLJ 175  
ユーザー ID のみのセキュリティ  
IBM Data Server Driver for JDBC and SQLJ 177

## [ラ行]

リソースの解放  
接続のクローズ 114, 171

リテラル・パラメーター  
DB2 for z/OS のストアード・プロシージャー呼び出し、  
JDBC 64  
リモート・トレース・コントローラー  
使用可能化 214  
リモート・トレース・コントローラー、IBM Data Server Driver  
for JDBC and SQLJ 214  
ルーチン  
Java プログラムからの呼び出し  
XML パラメーター 94  
例  
deregisterDB2XMLObject 95  
registerDB2XMLSchema 95  
例外  
IBM Data Server Driver for JDBC and SQLJ 99  
連続ストリーミング  
JDBC 69, 71, 151  
ロールバック  
トランザクション、JDBC 98  
SQLJ トランザクション 169

## A

### ARRAY

パラメーター  
ストアード・プロシージャーの JDBC プログラムからの  
呼び出し 77  
SQLJ プログラムからの ストアード・プロシージャーの  
呼び出し 160

## B

BatchUpdateException、情報の取り出し 106

## C

### CallableStatement

ストアード・プロシージャーの呼び出し 64

### comment

SQLJ 127

## D

### DatabaseMetaData

データ・ソース情報の取得、JDBC 43

### DataSource インターフェース

SQLJ 121, 123

### DataSource オブジェクト

作成とデプロイ 40

### DB2 for z/OS

プランおよびパッケージのバインディング 434

### DB2 for z/OS サーバー

Java プログラムからアクセスするためのセットアップ 19

### DB2 for z/OS サーバー構成

IBM Data Server Driver for JDBC and SQLJ クライアント  
・リルート・サポート 110

### DB2 JDBC Type 2 ドライバー 107, 108

セキュリティ 191

DriverManager インターフェース 31

### DB2 SQLJ Profile Printer コマンド 453

### DB2 SQLJ 変換プログラム・コマンド 430

### DB2 インフォメーション・センター

更新 461

バージョン 459

別の言語で表示する 460

### DB2 資料の印刷方法 458

### DB2 表からのデータの取得

JDBC 53

### DB2 表のデータの更新

JDBC 46

### DB2 プロファイル・バインド・プログラム・コマンド 447

### DB2Administrator クラス

IBM Data Server Driver for JDBC and SQLJ のみのメソッド 349

IBM Data Server Driver for JDBC and SQLJ のみのプロパティ 349

### DB2BaseDataSource クラス

IBM Data Server Driver for JDBC and SQLJ のみのメソッド 350

IBM Data Server Driver for JDBC and SQLJ のみのプロパティ 350

### DB2Binder ユーティリティ 9

### DB2CataloguedDatabase クラス

IBM Data Server Driver for JDBC and SQLJ のみのメソッド 355

IBM Data Server Driver for JDBC and SQLJ のみのプロパティ 355

### DB2ClientRerouteServerList クラス

IBM Data Server Driver for JDBC and SQLJ のみのメソッド 356

IBM Data Server Driver for JDBC and SQLJ のみのプロパティ 356

### DB2Connection インターフェース

IBM Data Server Driver for JDBC and SQLJ のみのメソッド 357

### DB2ConnectionPoolDataSource クラス

IBM Data Server Driver for JDBC and SQLJ のみのメソッド 373

IBM Data Server Driver for JDBC and SQLJ のみのプロパティ 373

### DB2DatabaseMetaData インターフェース

IBM Data Server Driver for JDBC and SQLJ のみのメソッド 375

### DB2Diagnosable インターフェース

IBM Data Server Driver for JDBC and SQLJ のみのメソッド 376

### DB2Diagnosable クラス

SQLCA の取得 170

DB2ExceptionFormatter クラス  
IBM Data Server Driver for JDBC and SQLJ のみのメソッド 377

DB2JCCPlugin インターフェース  
IBM Data Server Driver for JDBC and SQLJ のみのメソッド 377

DB2LobTableCreator ユーティリティ 17

DB2PooledConnection インターフェース  
IBM Data Server Driver for JDBC and SQLJ のみのメソッド 378

DB2PoolMonitor クラス  
IBM Data Server Driver for JDBC and SQLJ のみのメソッド 381

DB2PreparedStatement インターフェース  
IBM Data Server Driver for JDBC and SQLJ のみのメソッド 383

DB2ResultSet インターフェース  
IBM Data Server Driver for JDBC and SQLJ のみのメソッド 384

DB2ResultSetMetaData インターフェース  
IBM Data Server Driver for JDBC and SQLJ のみのメソッド 385

DB2RowID インターフェース  
IBM Data Server Driver for JDBC and SQLJ のみのメソッド 385

DB2SimpleDataSource  
定義 40

DB2SimpleDataSource クラス  
IBM Data Server Driver for JDBC and SQLJ のみのメソッド 386  
IBM Data Server Driver for JDBC and SQLJ のみのプロパティ 386

DB2Sqlca クラス  
IBM Data Server Driver for JDBC and SQLJ のみのメソッド 386

db2sqljbind コマンド 447

db2sqljcustomize  
SQLJ プロファイル・カスタマイザー 434

db2sqljprint  
JCC カスタマイズ・プロファイルの形式設定 205

db2sqljprint コマンド 453

DB2Statement インターフェース  
IBM Data Server Driver for JDBC and SQLJ のみのメソッド 387

DB2SystemMonitor インターフェース  
IBM Data Server Driver for JDBC and SQLJ のみのメソッド 390

DB2T4XAIndoubtUtil  
DB2 UDB (OS/390 および z/OS 版) バージョン 7 による分散トランザクション 21

DB2TraceManager クラス  
IBM Data Server Driver for JDBC and SQLJ のみのメソッド 393

DB2TraceManagerMXBean インターフェース  
IBM Data Server Driver for JDBC and SQLJ のみのメソッド 397

DB2XADataSource クラス  
IBM Data Server Driver for JDBC and SQLJ のみのメソッド 401

DB2Xml インターフェース  
IBM Data Server Driver for JDBC and SQLJ のみのメソッド 403

deregisterDB2XMLObject 95

DriverManager インターフェース  
DB2 JDBC Type 2 ドライバー 31  
SQLJ 118, 119

## E

Enterprise Java Beans 228

executeUpdate  
DB2 for z/OS サーバーに対する 48

## G

getCause メソッド 99

## H

HP-UX  
Java 環境のセットアップ 24

## I

IBM Data Server Driver for JDBC and SQLJ 63  
暗号化されたユーザー ID または暗号化されたパスワードのセキュリティ 178  
インストール 5  
拡張クライアント情報 82  
クライアント情報プロパティ 83  
クライアント・リルート・サポート 109  
クライアント・リルート・サポート、クライアント操作 111  
警告 99  
セキュリティ 173  
セキュリティ・プラグインのサポート 183  
接続コンセントレーターの使用可能化 234  
接続コンセントレーターのモニター技法 236  
データベースとの互換性 3  
データ・ソースへの接続  
DriverManager インターフェース 32  
トラステッド・コンテキストのサポート 185  
バージョンの判別 428  
プロパティ 245  
戻りコード、内部エラー 419  
ユーザー ID およびパスワード・セキュリティ 175  
ユーザー ID のみのセキュリティ 177  
リモート・トレース・コントローラー 214  
例、構成パラメーターによるトレース 205  
例、トレース・プログラム 206  
例外 99

IBM Data Server Driver for JDBC and SQLJ (続き)  
 DB2 for z/OS サーバーにアクセスするためのセットアップ  
 19  
 DB2T4XAIndoubtUtil 21  
 JDBC 拡張機能 347  
 JNDI によるクライアント・リルート・サポート 113  
 Kerberos セキュリティー 179  
 LOB サポート、JDBC 69, 71  
 LOB サポート、SQLJ 151  
 SQLException の処理 102  
 SQLSTATE、内部エラー 426  
 XML サポート、SQLJ 162

IBM Data Server Driver for JDBC and SQLJ Type 2 接続  
 URL 形式 36

IBM Data Server Driver for JDBC and SQLJ Type 4 接続  
 URL 形式 34

IBM Data Server Driver for JDBC and SQLJ クライアント・リ  
 ルート・サポート  
 DB2 for z/OS サーバー構成 110

IBM Data Server Driver for JDBC and SQLJ グローバル・プロ  
 パティアー 277

IBM Data Server Driver for JDBC and SQLJ 構成プロパティアー  
 277

IBM Data Server Driver for JDBC and SQLJ タイプ 2 接続  
 いつ使用するか 39

IBM Data Server Driver for JDBC and SQLJ タイプ 4 接続  
 いつ使用するか 39

IBM Data Server Driver for JDBC and SQLJ の管理サービス  
 アクセス 216

IBM Data Server Driver for JDBC and SQLJ のプロパティアー  
 すべてのデータ・ソースの 246  
 DB2 Database for Linux, UNIX, and Windows 用の 265  
 DB2 for z/OS および DB2 Database for Linux, UNIX, and  
 Windows の 253  
 DB2 for z/OS 用の 267  
 IDS および DB2 Database for Linux, UNIX, and Windows  
 データベース・サーバーの 264  
 IDS および DB2 for z/OS データベース・サーバーの 263  
 IDS の 273

IBM Data Server Driver for JDBC and SQLJ のみのメソッド  
 DB2Administrator クラス 349  
 DB2BaseDataSource クラス 350  
 DB2CataloguedDatabase クラス 355  
 DB2ClientRerouteServerList クラス 356  
 DB2Connection インターフェース 357  
 DB2ConnectionPoolDataSource クラス 373  
 DB2DatabaseMetaData インターフェース 375  
 DB2Diagnosable インターフェース 376  
 DB2ExceptionFormatter クラス 377  
 DB2JCCPlugin インターフェース 377  
 DB2PooledConnection インターフェース 378  
 DB2PoolMonitor クラス 381  
 DB2PreparedStatement インターフェース 383  
 DB2ResultSet インターフェース 384  
 DB2ResultSetMetaData インターフェース 385  
 DB2RowID インターフェース 385

IBM Data Server Driver for JDBC and SQLJ のみのメソッド  
 (続き)  
 DB2SimpleDataSource クラス 386  
 DB2sqlca クラス 386  
 DB2Statement インターフェース 387  
 DB2SystemMonitor インターフェース 390  
 DB2TraceManager クラス 393  
 DB2TraceManagerMXBean インターフェース 397  
 DB2XADatasource クラス 401  
 DB2Xml インターフェース 403

IBM Data Server Driver for JDBC and SQLJ のみのプロパティ  
 ー  
 DB2Administrator クラス 349  
 DB2BaseDataSource クラス 350  
 DB2CataloguedDatabase クラス 355  
 DB2ClientRerouteServerList クラス 356  
 DB2ConnectionPoolDataSource クラス 373  
 DB2SimpleDataSource クラス 386

## J

### Java

アプリケーション  
 サポート 1  
 z/OS サーバーへのアクセス 19

アプレット  
 使用するためのポイント 197  
 環境のセットアップ (HP-UX) 24  
 構築  
 JDBC アプリケーション 194  
 JDBC アプレット 193  
 JDBC ルーチン 194  
 SQLJ アプリケーション 196  
 SQLJ アプレット 195  
 SQLJ ルーチン 199

Enterprise Java Beans 228

Java 2 Platform, Enterprise Edition  
 アプリケーション・サポート 219  
 概要 219  
 コンテナー 220  
 データベースの要件 221  
 トランザクション管理 222  
 要件 221  
 Enterprise Java Beans 228  
 server 221

Java Naming and Directory Interface (JNDI)  
 説明 221

Java Transaction API  
 説明 222

Java Transaction Service  
 説明 222

Java アプリケーション  
 環境のカスタマイズ 18

Java アプリケーション、構築 193

Java アプリケーションの構築 193

Java 環境のカスタマイズ 18



Java 参照情報 239  
Java プログラムの準備 193  
Java 分散トランザクション  
    タイムアウト値の設定 227  
JDBC  
    オブジェクトの作成 45  
    オブジェクトの変更 45  
    オプティミスティック・ロック 86  
    環境変数 18  
    更新可能な ResultSet 56, 57  
    構成 18  
    スクロール可能な ResultSet 56, 57  
    接続コンセントレーター 233  
    データ・タイプ・マッピング 239  
    パッケージへのアクセス 42  
    バッチ更新 48  
    バッチ照会 54  
    分離レベル 97  
    問題診断 203  
    DB2 for z/OS サーバーに対する executeUpdate メソッド  
        48  
    ResultSet の保持可能性 56, 57  
    ResultSet、行が挿入されたかどうかのテスト 63  
    ResultSet、行の挿入 61  
    ResultSet、削除ホールになっているかどうかのテスト 61  
    SQL の実行 45  
    SQLWarning の処理 105  
    Sysplex ワークロード・バランシング 233  
JDBC 4.0 での getColumnLabel の変更 414  
JDBC 4.0 での getColumnName の変更 414  
JDBC API  
    ドライバー・サポートの比較 281  
JDBC (Java Database Connectivity)  
    アプリケーションの構築 194  
    アプレット  
        構築 193  
        使用するためのポイント 197  
    ルーチン  
        構築 194  
    DB2 for z/OS サーバーへのアクセス 19  
    IBM Data Server Driver for JDBC and SQLJ  
        インストール 5  
JDBC アプリケーション  
    変数の宣言 44  
    例 27  
JDBC アプリケーション、データの取り出し 52  
JDBC アプリケーション、トランザクション制御 97  
JDBC アプリケーション・プログラミング 27  
JDBC および SQLJ  
    サポートされるドライバー 1  
JDBC 拡張機能  
    IBM Data Server Driver for JDBC and SQLJ 347  
JDBC 接続  
    使用 40  
JDBC ドライバー  
    JDBC の相違点 405

JDBC ドライバー (続き)  
    SQLJ の相違点 416  
JDBC トランザクション  
    コミット 98  
    自動コミット 98  
    ロールバック 98  
JDBC バッチ・エラー  
    BatchUpdateException 106  
JDBC 問題の診断 203  
JNDI (Java Naming and Directory Interface) 221  
JNDI によるクライアント・リルート・サポート  
    IBM Data Server Driver for JDBC and SQLJ 113  
JTA (Java Transaction API)  
    説明 222  
JTS (Java Transaction Service)  
    説明 222

## K

Kerberos セキュリティー  
    IBM Data Server Driver for JDBC and SQLJ 179

## L

LOB サポート  
    IBM Data Server Driver for JDBC and SQLJ 69  
    IBM Data Server Driver for JDBC and SQLJ、SQLJ 151  
    LOB ロケーター 71  
LOB 操作  
    IBM Data Server Driver for JDBC and SQLJ 71  
LOB 列  
    互換 Java データ・タイプの選択、JDBC 73  
    互換性のある Java データ・タイプの選択、SQLJ 152  
LOB ロケーター  
    IBM Data Server Driver for JDBC and SQLJ 70, 151

## P

ParameterMetaData  
    パラメーター情報の取得、JDBC 51  
PreparedStatement メソッド  
    パラメーター・マーカのない SQL ステートメント 47  
    パラメーター・マーカを指定した SQL ステートメント  
        46, 53

## R

registerDB2XMLSchema 95  
ResultSet  
    行が挿入されたかどうかのテスト、JDBC 63  
    行の挿入、JDBC 61  
    削除ホールになっているかどうかのテスト、JDBC 61  
ResultSet の保持可能性  
    JDBC 56, 57

ResultSetMetaData  
結果セット情報の取得、JDBC 55  
ResultSetMetaData.getColumnLabel の値の変更、JDBC 4.0 414  
ResultSetMetaData.getColumnName の値の変更、JDBC 4.0 414  
ROWID  
IBM Data Server Driver for JDBC and SQLJ 75, 157

## S

SDK for Java  
HP-UX Java 環境のセットアップ 24  
SDK for Java の相違点  
Java アプリケーションに影響を与える 418  
SDK for Java バージョン 1.5 機能 166  
Secure Socket Layer  
IBM Data Server Driver for JDBC and SQLJ 187  
SET TRANSACTION 節  
SQLJ 319  
setTransactionTimeout  
XAResource 227  
SQL ステートメント  
ヘルプを表示する 459  
SQLJ のエラー処理 170  
SQL の実行  
JDBC 45  
SQLJ 127  
SQLCA の取得  
DB2Diagnosable クラス 170  
SQLException  
IBM Data Server Driver for JDBC and SQLJ での処理 102  
SQLException の処理 107  
SQLJ 166  
イテレーター宣言節 315  
イテレーターの複数インスタンス 145  
イテレーター変換節 320  
インプリメント節 312  
エラー処理 170  
環境変数 18  
既存の接続の使用 124  
結果セット・イテレーター 137  
コンテキスト節 317  
実行可能節 316  
実行制御 157  
診断ユーティリティの実行 203  
スクロール可能イテレーター 145  
ストアード・プロシージャの呼び出し 149  
セキュリティ、プログラムの準備 188  
接続宣言節 314  
代入節 320  
データ・ソースへの接続 117  
デフォルトのコンテキストを使用した接続 124  
トレース・データの収集 203  
パッケージへのアクセス 125  
バッチ更新 133  
表上の複数のイテレーター 143  
分離レベル 169

SQLJ (続き)  
ホスト式 125, 311  
問題診断 203  
ランタイム環境のインストール 18  
comment 127  
DataSource インターフェースの使用 121, 123  
DB2 表の作成および変更 128  
DriverManager インターフェースの使用 118, 119  
SET TRANSACTION 節 319  
SQL の実行 127  
SQLCA の取得 170  
SQLWarning の処理 171  
with 節 312  
SQLJ アプリケーション  
例 115  
SQLJ アプリケーション、トランザクション制御 169  
SQLJ アプリケーション・プログラミング 115  
SQLJ (組み込み SQL (Java 用))  
アプリケーション  
構築 196  
UNIX でのコンパイル・オプション 198  
Windows でのコンパイル・オプション 199  
アプレット  
構築 195  
アプレット。使用するためのポイント 197  
ルーチン  
UNIX でのコンパイル・オプション 200  
Windows でのコンパイル・オプション 201  
ルーチンの構築 199  
sqlj コマンド 430  
SQLJ ステートメント参照 311  
SQLJ 節 311  
SQLJ トランザクション 169  
SQLJ の実行コンテキスト 157  
SQLJ プログラムの準備、コマンド 430  
SQLJ プロファイル・バインド・プログラム・コマンド 447  
SQLJ 変換プログラム・コマンド 430  
SQLJ 変数名  
制限 126  
SQLJ 問題の診断 203  
sqlj.runtime  
インターフェースおよびクラス 321  
sqlj.runtime.ASCIIStream 334, 345  
sqlj.runtime.BinaryStream 335  
sqlj.runtime.CharacterStream 335  
sqlj.runtime.ConnectionContext  
アプリケーションで呼び出されるメソッド 322  
sqlj.runtime.ExecutionContext  
アプリケーションで呼び出されるメソッド 337  
sqlj.runtime.ForUpdate  
位置指定 UPDATE および DELETE のための 327  
sqlj.runtime.NamedIterator  
アプリケーションで呼び出されるメソッド 327  
sqlj.runtime.PositionedIterator  
アプリケーションで呼び出されるメソッド 328

sqlj.runtime.ResultSetIterator  
アプリケーションで呼び出されるメソッド 328

sqlj.runtime.Scrollable  
アプリケーションで呼び出されるメソッド 331

sqlj.runtime.SQLNullException 345

sqlj.runtime.UnicodeStream 347

SQLSTATE  
IBM Data Server Driver for JDBC and SQLJ のエラー 426

SQLWarning  
JDBC での処理 105  
SQLJ での処理 171

SQLWarning の処理 108

SSL  
IBM Data Server Driver for JDBC and SQLJ 187

Statement.executeQuery  
DB2 表からのデータの取得 52

Sysplex ワークロード・バランシング  
IBM Data Server Driver for JDBC and SQLJ 233

## U

UNIX  
SQLJ アプリケーション  
コンパイル・オプション 198

SQLJ ルーチン  
コンパイル・オプション 200

URL 形式 IBM Data Server Driver for JDBC and SQLJ Type 2  
接続  
DB2BaseDataSource クラス 36

URL 形式 IBM Data Server Driver for JDBC and SQLJ Type 4  
接続  
DB2BaseDataSource クラス 34

## V

Visual Explain  
チュートリアル 463

## W

Windows オペレーティング・システム  
SQLJ アプリケーション  
コンパイル・オプション 199

SQLJ ルーチン  
コンパイル・オプション 201

with 節  
SQLJ 312

## X

XML  
パラメーター  
Java プログラムからのルーチンの呼び出し 94

XML サポート  
IBM Data Server Driver for JDBC and SQLJ 162

XML スキーマの除去  
Java API 95

XML スキーマの登録  
Java API 95

XML データ  
データの Java アプリケーションでの取り出し 164

Java アプリケーション 87

Java アプリケーションでの表からの取り出し 90

Java アプリケーションでの表の更新 88, 162







Printed in Japan

SC88-4427-00



日本アイ・ビー・エム株式会社  
〒106-8711 東京都港区六本木3-2-12

Spine information:

**DB2 Version 9.5 for Linux, UNIX, and Windows**

**Java アプリケーションの開発**

