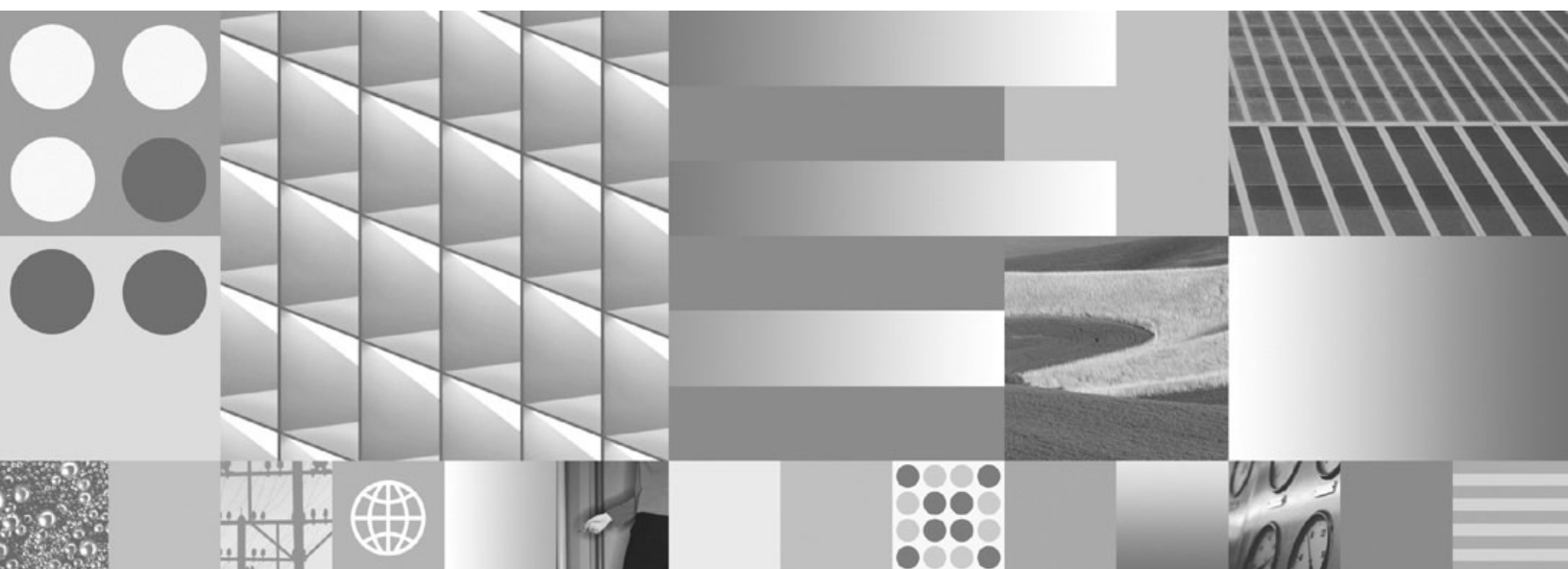


IBM DB2 9.7
for Linux, UNIX, and Windows



バージョン 9 リリース 7

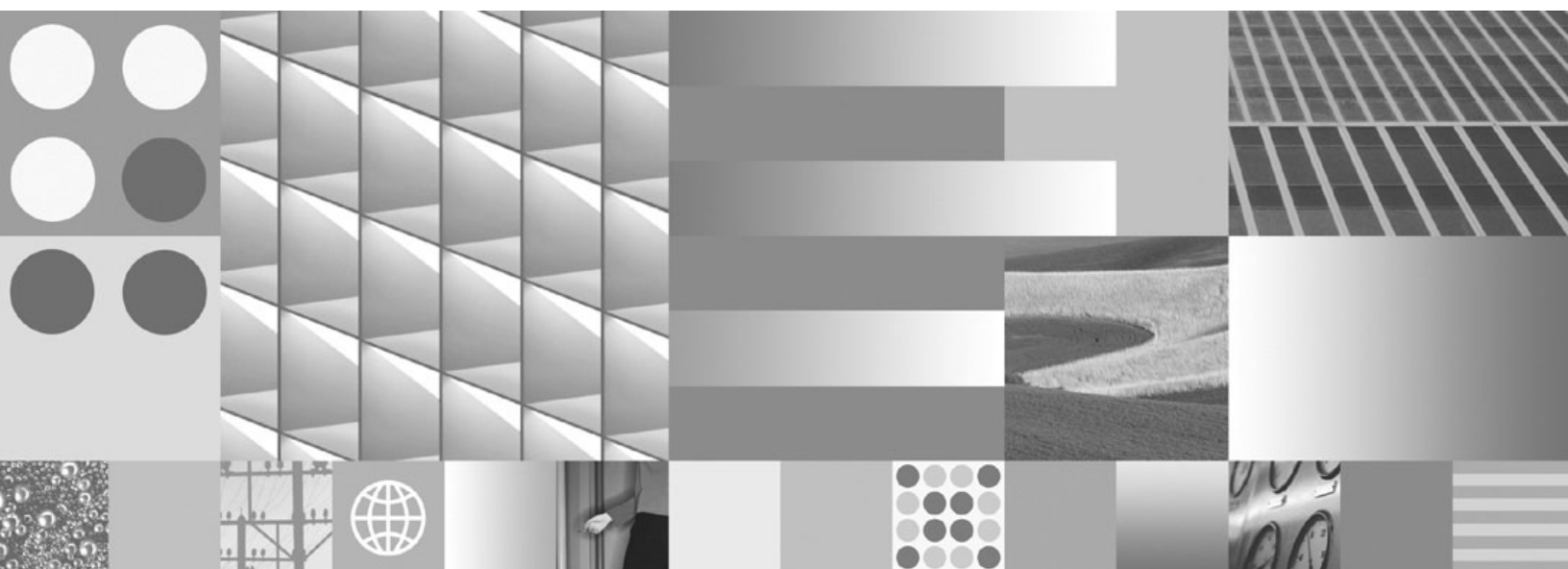


SQL および外部ルーチンの開発
2009 年 11 月更新版

IBM DB2 9.7
for Linux, UNIX, and Windows



バージョン 9 リリース 7



SQL および外部ルーチンの開発
2009 年 11 月更新版

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、381 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。

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 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 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。

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： SC27-2448-01
IBM DB2 9.7 for Linux, UNIX, and Windows
Version 9 Release 7
Developing User-defined Routines (SQL and External)
Updated November, 2009

発行： 日本アイ・ビー・エム株式会社

担当： トランスレーション・サービス・センター

第1刷 2009.11

© Copyright International Business Machines Corporation 1993, 2009.

目次

本書について	vii
本書の対象読者	vii
第 1 章 ルーチンの開発	1
ルーチン	2
第 2 章 ルーチンの概説	3
ルーチン使用の利点	3
ルーチンのタイプ	4
システム定義ルーチンおよびユーザー定義ルーチン	7
ルーチンの機能タイプ	11
ルーチンのインプリメンテーション	22
ルーチンの使用法	34
システム定義ルーチンを使用したデータベースの	
管理	34
ユーザー定義関数による SQL 関数サポートの拡	
張	35
SQL 表関数を使用した監査	35
ルーチン開発のためのツール	38
IBM Data Studio ルーチン開発サポート	38
ルーチン内で実行できる SQL ステートメント	39
ルーチンの SQL アクセス・レベル	45
ルーチンで実行できる SQL ステートメントの決	
定	46
ルーチンの移植性	47
ルーチンのインターオペラビリティ	47
ルーチンのパフォーマンス	48
ルーチンのセキュリティ	56
ルーチンのセキュリティ確保	58
SQL の入ったルーチンの許可およびバインド	59
プロシージャが表に対する読み取り/書き込みを実	
行する時に起きるデータの競合	63
第 3 章 外部ルーチン	67
外部ルーチンのフィーチャー	67
外部関数とメソッドのフィーチャー	68
外部ルーチンの SQL	79
外部ルーチンのパラメーター・スタイル	82
サポートされているルーチン・プログラミング言語	
外部ルーチンの開発でサポートされている API	
とプログラミング言語の比較	88
ルーチン開発に関するパフォーマンスの考慮	93
ルーチンのセキュリティに関する考慮事項	96
ルーチンのコード・ページに関する考慮事項	99
32 ビットおよび 64 ビット・アプリケーションお	
よびルーチンのサポート	100
外部ルーチンの 32 ビットと 64 ビットのサポ	
ート	102
64 ビット・データベース・サーバー上での 32	
ビット・ライブラリーを持つルーチンのパフォー	
マンス	103

外部ルーチンでの XML データ・タイプのサポート	103
外部ルーチンに関する制約事項	104
外部ルーチンの作成	107
ルーチンの作成	109
ルーチンのデバッグ	110
SQL プロシージャからのエラー・メッセージ	
の戻り	112
外部ルーチンのライブラリーおよびクラス管理	113
外部ルーチン・ライブラリーおよびクラスのデブ	
ロイメント	113
外部ルーチン・ライブラリーまたはクラス・ファ	
イルのセキュリティ	114
外部ルーチンのライブラリーおよびクラスの解決	
外部ルーチンのライブラリーおよびクラス・ファ	
イルに対する変更	115
外部ルーチンのライブラリーおよびクラス・ファ	
イルのバックアップおよびリストア	116
外部ルーチン・ライブラリー管理およびパフォー	
マンス	116

第 4 章 .NET 共通言語ランタイム	
(CLR) ルーチン	119
.NET CLR 言語での外部ルーチン開発のサポート	120
.NET CLR ルーチン開発のためのツール	120
.NET CLR ルーチンの設計	120
.NET CLR ルーチンでの SQL データ・タイプ	
表記	121
.NET CLR ルーチンのパラメーター	123
.NET CLR プロシージャからの結果セットの	
戻り	126
CLR ルーチンのセキュリティおよび実行モ	
ード	127
.NET CLR ルーチンに関する制約事項	128
.NET CLR ルーチンの作成	129
DB2 コマンド・ウィンドウから .NET CLR ル	
ーチンを作成する	130
.NET CLR ルーチン・コードのビルド	132
サンプル・ビルド・スクリプトを使った .NET	
共通言語ランタイム (CLR) ルーチン・コードの	
ビルド	133
DB2 コマンド・ウィンドウからの .NET 共通言	
語ランタイム (CLR) ルーチン・コードのビルド	
CLR .NET ルーチンのコンパイルとリンクのオ	
プション	137
.NET CLR ルーチンのデバッグ	138
.NET CLR ルーチンに関連したエラー	139
.NET CLR ルーチンの例	142
C# .NET CLR プロシージャの例	143
Visual Basic .NET CLR 関数の例	154
Visual Basic .NET CLR プロシージャの例	159

例: C# .NET CLR プロシージャでの XML および XQuery サポート	170
例: C プロシージャでの XML および XQuery サポート	174
C# .NET CLR 関数の例	178

第 5 章 C および C++ ルーチン 185

C での外部ルーチン開発のサポート	186
C++ での外部ルーチン開発のサポート	186
C および C++ ルーチン開発のためのツール	186
C および C++ ルーチンの設計	187
C および C++ ルーチン開発に必要な組み込みファイル (sqludf.h)	187
C および C++ ルーチンでのパラメーター	189
C および C++ ルーチンにおいてサポートされる SQL データ・タイプ	202
C および C++ ルーチンでの SQL データ・タイプ処理	205
C、C++、OLE、または COBOL ルーチンへの引数の引き渡し	215
C および C++ ルーチンでのグラフィック・ホスト変数	229
C++ のタイプ修飾	230
C および C++ プロシージャからの結果セットの戻り	232
C および C++ ルーチンの作成	233
C および C++ ルーチン・コードのビルド	236
サンプル bldrtn スクリプトを使用した C および C++ ルーチン・コードのビルド	236
DB2 コマンド・ウィンドウからの C および C++ ルーチン・コードのビルド	243
C および C++ ルーチンのコンパイルとリンクのオプション	245
構成ファイルを使用した C または C++ の組み込み SQL ストアード・プロシージャの構築	259
構成ファイルを使用した C または C++ のユーザー定義関数の構築 (AIX)	261
DB2 ルーチンの共用ライブラリーの再構築	262
データベース・マネージャー構成ファイルの更新	262

第 6 章 COBOL プロシージャ 265

COBOL での外部プロシージャ開発のサポート	267
COBOL 組み込み SQL アプリケーションでサポートされている SQL データ・タイプ	267
COBOL ルーチンのビルド	271
COBOL ルーチンのコンパイルとリンクのオプション	271
AIX での IBM COBOL ルーチンの構築	276
UNIX Micro Focus COBOL ルーチンの構築	277
Windows での IBM COBOL ルーチンの構築	278
Windows での Micro Focus COBOL ルーチンの構築	280

第 7 章 Java ルーチン 283

サポートされる Java ルーチン開発ソフトウェア	284
-------------------------------------	-----

Java ルーチンでの JDBC および SQLJ アプリケーション・プログラミング・インターフェースのサポート	284
SDK for Java ルーチン開発の仕様 (UNIX)	285
Java ルーチン用のドライバーの指定	286
Java (JDBC および SQLJ) ルーチン開発のためのツール	286
Java ルーチンの設計	287
Java ルーチンでサポートされている SQL データ・タイプ	288
SQLJ ルーチン内の接続コンテキスト	289
Java ルーチンのパラメーター	290
Java ルーチンに対するデータ・タイプ ARRAY のパラメーターの引き渡し	302
JDBC プロシージャからの結果セットの戻り	303
SQLJ プロシージャからの結果セットの戻り	303
JDBC アプリケーションおよびルーチンでのプロシージャの結果セットの受け取り	304
SQLJ アプリケーションおよびルーチンでのプロシージャの結果セットの受け取り	306
Java ルーチンに関する制約事項	307
Java の表関数実行モデル	307
Java ルーチンの作成	309
コマンド行からの Java ルーチンの作成	310
Java ルーチン・コードのビルド	312
JDBC ルーチンの構築	313
SQL ルーチンの構築	314
Java (SQLJ) ルーチンのコンパイルとリンクのオプション	315
Java ルーチン・クラス・ファイルの DB2 データベース・サーバーへのデプロイ	316
データベース・サーバーでの JAR ファイル管理	318
Java ルーチン・クラスの更新	319
Java (JDBC) ルーチンの例	319
例: Java (JDBC) プロシージャでの配列データ・タイプ	319
例: Java (JDBC) プロシージャでの XML および XQuery サポート	320

第 8 章 OLE オートメーション・ルーチンの設計 325

OLE オートメーション・ルーチンの作成	325
OLE ルーチン・オブジェクト・インスタンスおよびスクリッチパッドに関する考慮事項	326
OLE オートメーションでサポートされている SQL データ・タイプ	327
BASIC および C++ での OLE オートメーション・ルーチン	328

第 9 章 OLE DB ユーザー定義表関数 331

OLE DB 表 UDF の作成	332
完全修飾行セット名	333
OLE DB でサポートされている SQL データ・タイプ	334

第 10 章 ルーチンの呼び出し 337

SQL の入ったルーチンの許可およびバインド	338	コマンド行プロセッサから SQL 状態ヘルプを表示する	372
ルーチン名およびパス	342	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	372
ネストされたルーチンの呼び出し	344	DB2 インフォメーション・センターでの希望する言語でのトピックの表示	372
64 ビット・データベース・サーバーでの 32 ビット・ルーチンの呼び出し	344	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	373
プロシージャの参照	345	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・センターの手動更新	375
プロシージャの呼び出し	345	DB2 チュートリアル	377
関数の参照	360	DB2 トラブルシューティング情報	378
関数選択	361	ご利用条件	378
UDF またはメソッドのパラメーターとしての特 殊タイプ	362	付録 B. 特記事項	381
UDF パラメーターとしての LOB 値	363	索引	385
スカラー関数またはメソッドの呼び出し	364		
ユーザー定義の表関数の呼び出し	364		
付録 A. DB2 技術情報の概説	367		
DB2 テクニカル・ライブラリー (ハードコピーまたは PDF 形式)	368		
DB2 の印刷資料の注文方法	371		

本書について

本書では、必要な機能性を備えたシステム定義ルーチンがない場合に行う、ユーザー定義ルーチンの開発に関する情報を提供します。

本書の対象読者

本書は、すべてのレベルのデータベース設計者、データベース管理者、およびアプリケーション開発者を対象としています。

- データベースのアーキテクチャーの中の複数のコンテキストで再利用可能な SQL および関連ロジックをモジュール化するため、どのようにルーチン・オブジェクトを作成および使用できるかについて学習することに興味を持つデータベース設計者。
- データベース管理システム内でのユーザー定義ルーチンの作成、管理、デプロイ、保護、トラブルシューティングの方法、およびルーチンを使用してシステム・パフォーマンスを向上させる方法について学習することに興味を持つデータベース管理者。
- アプリケーションのモジュール性とパフォーマンスを向上させるため、いつ、どのように SQL ステートメントとアプリケーション・ロジックをルーチンにカプセル化するか、およびユーザー定義ルーチンの設計、作成、ビルドの段階的な方法について学習することに興味を持つアプリケーション開発者。アプリケーション開発者は、SQL ステートメントの作成の経験、およびルーチン開発用にサポートされているプログラミング言語 (C、C++、Java™、COBOL、C#、Visual Basic、または別の .NET CLR サポート対象プログラミング言語) のいずれかを使ったプログラミングの経験がなければなりません。

第 1 章 ルーチンの開発

多くの場合、ルーチンを開発するのは、必要とされる機能を提供するシステム定義ルーチンが使用できない場合です。

ルーチンの機能タイプとルーチンのインプリメンテーションが異なっても、ルーチン開発の基本ステップは、一般にすべてのルーチンに共通しています。作成するルーチンのタイプ、使用するインプリメンテーションを決定し、ルーチンのインターフェースを定義し、ルーチンのロジックを開発し、SQL を実行してルーチンを作成し、ルーチンをテストした後に、一般使用に向けてルーチンをデプロイします。

どのようなタイプのルーチンを開発するかによって、従うべき特定の手順があります。このトピックから、ルーチン開発を始める際の適切なトピックを見つけることができます。

前提条件

- ルーチンの基本概念を読んで理解する。
 - ルーチンのタイプ、ルーチンの便利な用途、ルーチン開発用ツール、ルーチンのベスト・プラクティスなどについて学習するには、以下のトピックを参照してください。
 - 3 ページの『第 2 章 ルーチンの概説』
- 用意されているルーチン開発ツールについて知る。開発ツールを使用すれば、ルーチンの開発を短期間で簡単に行うことができます。
 - ルーチン開発に使用できるツールについて学習するには、以下のトピックを参照してください。
 - 38 ページの『ルーチン開発のためのツール』

手順

1. 既存のシステム定義ルーチンが自分のルーチンのニーズに合っているかどうかを判別する。
 - システム定義ルーチンがニーズに合っている場合は、337 ページの『第 10 章 ルーチンの呼び出し』を参照できます。
2. どのような機能タイプのルーチンを開発するかを決定する。
3. どのようなルーチン・インプリメンテーションを使用するかを決定する。
 - SQL ルーチンを必要とする場合は、SQL ルーチン (SQL PL) に関する情報を参照してください。
 - 外部ルーチンを必要とする場合は、外部ルーチンに関する情報を参照してください。

SQL ルーチンの開発と外部ルーチンの開発は似ていますが、異なる点もあります。両方のルーチン・タイプとも、まずロジックを設計してから、データベース内にルーチンを作成するために、ルーチンの機能タイプ固有の CREATE ステートメントを実行する必要があります。この種のルーチン作成ステートメントには、CREATE PROCEDURE、CREATE FUNCTION、CREATE METHOD があります。各 CREATE ステートメントに固有の節でルーチンの特性を定義します。定義する特性

としては、ルーチン名、ルーチンのパラメーターの数とタイプ、ルーチン・ロジックに関する詳細情報があります。DB2® は、それらの節に指定されている情報に基づいて、ルーチンを識別し、呼び出しの時点でそのルーチンを実行します。ルーチンの CREATE ステートメントが正常に実行されると、データベース内にそのルーチンが作成されます。そのルーチンの特性は、ユーザーが照会できる DB2 のシステム・カタログ表に保管されます。CREATE ステートメントの実行によってルーチンを作成する作業のことをルーチンの定義またはルーチンの登録ともいいます。

外部ルーチンのロジックは、データベース・ファイル・システムに置かれるユーザー作成のライブラリーまたはクラスにインプリメントされるので、ロジックをプログラミングしてビルドし、結果のライブラリー・ファイルまたはクラス・ファイルを適切に配置するための追加ステップが必要になります。

ルーチンを開発したなら、以下の作業が必要です。

- ルーチンのデバッグ
- 実稼働環境へのルーチンのデプロイ
- ルーチンを実行するためのユーザーへの特権の付与
- ルーチンの呼び出し
- ルーチンのパフォーマンスのチューニング

ルーチン

ルーチンは、プログラミング・サブルーチンのように、さまざまな SQL インターフェースから呼び出せるプログラミングおよびデータベース・ロジックをカプセル化できるデータベース・オブジェクトです。ルーチンは、製品に付属のシステム定義のものもあれば、ユーザーが作成できるユーザー定義のものもあります。ルーチンは、SQL ステートメント、プログラミング言語、またはその混用によってインプリメントすることができます。さまざまなタイプのルーチンがさまざまなインターフェースを提供しますが、それらは SQL ステートメント、クライアント・アプリケーション、および一部のデータベース・オブジェクトの機能の拡張のために使用できます。

DB2 によってサポートされるルーチンおよびインプリメンテーションのタイプの詳細については、4 ページの『ルーチンのタイプ』というトピックを参照してください。

ルーチンが多機能であることが、その有用な応用範囲が広いことの理由の 1 つとなっています。

第 2 章 ルーチンの概説

ルーチンは、プログラミング・サブルーチンのように呼び出せるロジックをカプセル化するために使用できる、一種のデータベース・オブジェクトのタイプです。データベースまたはデータベース・アプリケーション・アーキテクチャーのルーチンには、便利な用途が多数あります。ルーチンを使用すると、データベースの全体的な設計、データベースのパフォーマンス、およびデータ・セキュリティーを向上させるだけでなく、基本的な監査機構のインプリメントなども行うことができます。

ルーチン使用の利点

以下の利点を活用するには、ルーチンを使用します。

SQL インターフェースからの呼び出しが可能なアプリケーション・ロジックのカプセル化 多数の異なるクライアント・アプリケーションが共通の要件を持つ環境では、ルーチンを効果的に使用することによって、コードの再利用、標準化、保守の作業を単純化できます。ルーチンを使用している環境内であれば、共通アプリケーションの動作の一面を変更する必要が生じた場合に、その動作をカプセル化した該当のルーチンを変更するだけで済みます。ルーチンがない場合は、各アプリケーションごとにそのロジックを変更する必要があります。

他のデータベース・オブジェクトへのアクセスの制御

ルーチンを使用して、データベース・オブジェクトへのアクセスを制御することができます。例えば、CREATE TABLE などの特定の SQL ステートメントを発行する許可が基本的に与えられていないユーザーにも、そのステートメントの 1 つ以上の特定インプリメンテーションを含んだルーチンを呼び出す許可を与えることができます。こうして特権をカプセル化することで特権の管理が単純化されます。

ネットワーク・トラフィックの削減によるアプリケーション・パフォーマンスの向上 クライアント・コンピューターでアプリケーションを実行する場合、各 SQL ステートメントは別々にクライアント・コンピューターからデータベース・サーバー・コンピューターに送信されて実行され、結果セットも別々に戻されます。その結果、ネットワーク・トラフィックが肥大化してしまいます。ユーザーとの対話をほとんど必要とせず、データベースとの多くの対話を必要とする処理があれば、ネットワーク・トラフィックの量を最小化するためにその処理をサーバー上にインストールしたり、より強力なデータベース・サーバー上で作業を実行できるようにするという処置には意味があります。

SQL の実行の高速化と効率化

ルーチンはデータベース・オブジェクトなので、クライアント・アプリケーションよりも SQL の要求とデータを効率的に送信できます。このため、SQL ステートメントはクライアント・アプリケーションで実行するよりも、ルーチン内で実行する方が高いパフォーマンスを発揮します。NOT FENCED 節を使って作成されるルーチンは、データベース・マネージャー

と同じプロセスで実行されるので、通信に共用メモリーを使用でき、結果としてアプリケーション・パフォーマンスが向上します。

別のプログラミング言語でインプリメントされたロジックとのインターオペラビリティの実現

コード・モジュールは、それぞれのプログラマーが別々のプログラミング言語でインプリメントできます。また、一般的にコードはできる限り再利用することが望ましいので、DB2 のルーチンは、高度なインターオペラビリティをサポートしています。

- 1 つのプログラム言語のクライアント・アプリケーションから、別々のプログラム言語でインプリメントしたルーチン呼び出すことができます。例えば、C クライアント・アプリケーションから、.NET 共通言語ランタイム・ルーチン呼び出すことができます。
- ルーチンはそのタイプやインプリメンテーションに関係なく、別のルーチン呼び出すことができます。例えば Java プロシージャは、組み込み SQL スカラー関数を呼び出すことができます。
- 1 つのオペレーティング・システム上のデータベース・サーバーに作成したルーチンを、別のオペレーティング・システム上で実行する DB2 クライアントから呼び出せます。

ここで説明した利点は、ルーチン使用の数ある利点の中のほんの一部にすぎません。データベース管理者、データベース設計者、データベース・アプリケーション開発者など、さまざまなユーザーがルーチンの使用によって恩恵を受けます。このような理由から、ルーチンには検討に値する多くの用途があります。

各種の機能要件に合わせたさまざまなタイプのルーチンがあり、それぞれのインプリメンテーションの方法もさまざまです。ルーチンのタイプとインプリメンテーションの選択によって、上記の利点をどの程度具体化できるかが決まる場合もあります。ルーチンは基本的に、ロジックをカプセル化するための強力な手法です。この手法を活用すれば、SQL を拡張し、アプリケーションの構造と保守作業を改善し、場合によってはアプリケーションのパフォーマンスを向上させることができます。

ルーチンのタイプ

ルーチンのタイプは多種多様です。ルーチンはさまざまな方法でグループ化できますが、主に、そのシステム定義またはユーザー定義、機能、およびインプリメンテーションごとにグループ化されます。

サポートされているルーチンの定義は次のとおりです。

- 7 ページの『システム定義ルーチン』
- 8 ページの『ユーザー定義ルーチン』

サポートされているルーチンの機能タイプは次のとおりです。

- 12 ページの『ルーチン: プロシージャ』 (ストアド・プロシージャとも呼ばれる)
- 13 ページの『ルーチン: 関数』
- 17 ページの『ルーチン: メソッド』

サポートされているルーチンのインプリメンテーションは次のとおりです。

- 23 ページの『組み込みルーチンのインプリメンテーション』
- 23 ページの『ソース派生ルーチンのインプリメンテーション』
- 23 ページの『SQL ルーチンのインプリメンテーション』
- 24 ページの『外部ルーチンのインプリメンテーション』

以下の図に、ルーチンの種別階層を示します。ルーチンはすべて、システム定義かユーザー定義のどちらかです。暗いグレー/青色のボックスのものがルーチンの機能タイプで、明るいグレー/オレンジ色のボックスのものがサポートされるルーチンのインプリメンテーションです。組み込みルーチンのインプリメンテーションのタイプは固有であるため、強調されています。

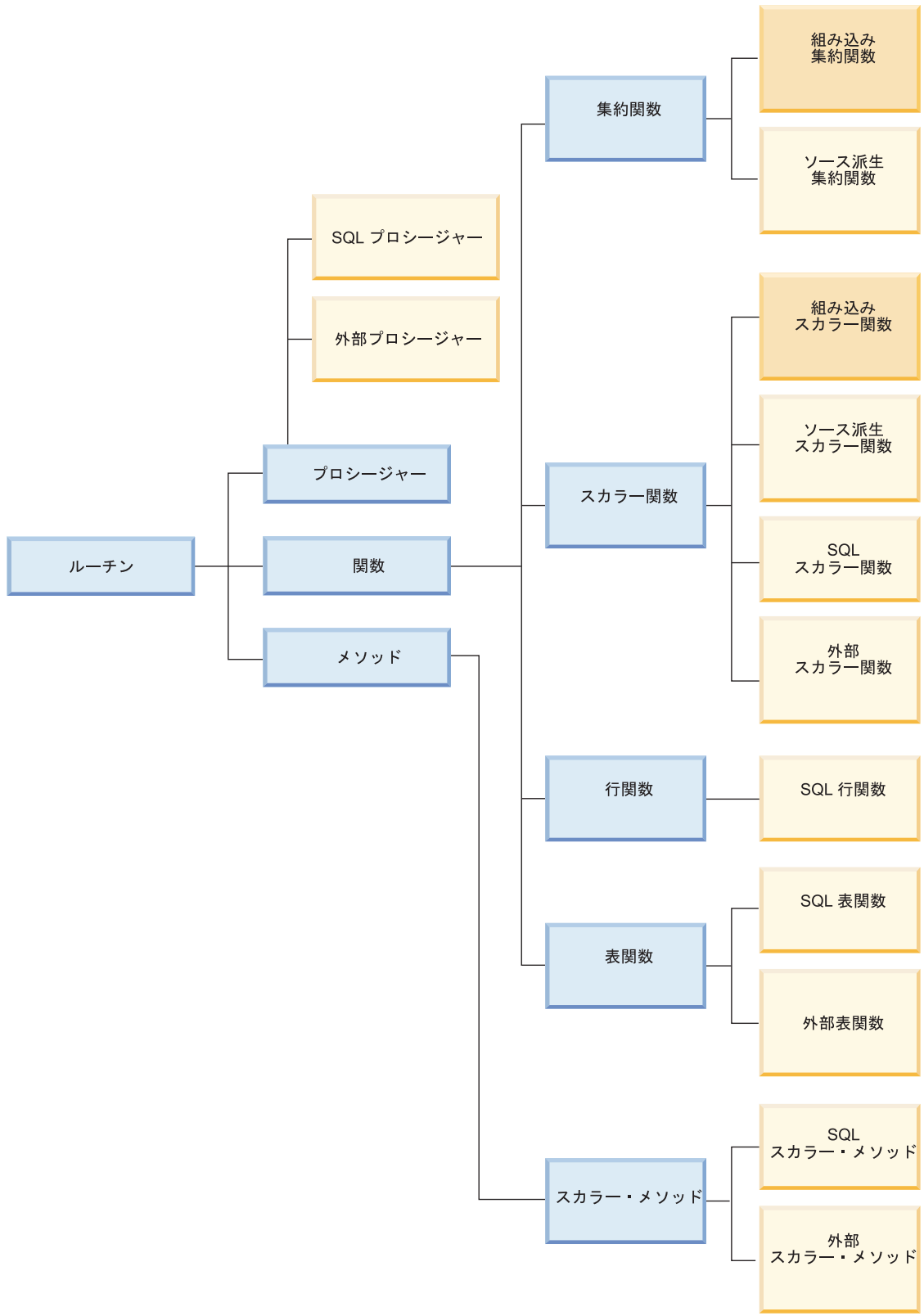


図 1. ルーチンの種別

各種ルーチンは、SQL 言語の機能を拡張し、より多くのモジュラー・データベース・アプリケーションを開発するための拡張サポートを提供しています。

システム定義ルーチンおよびユーザー定義ルーチン

ルーチンをカテゴリー化する最も簡単な方法の 1 つは、それらを『システム定義ルーチン』 および 8 ページの『ユーザー定義ルーチン』 に分割することです。

システム定義ルーチンは、製品に付属するルーチンです。このルーチンは、管理機能からデータベース・システムおよびカタログのレポート機能に至るタスクのための、幅広いサポートを提供しています。これには、即使用することができ、前提条件のセットアップまたは登録ステップが不要です。ただし、ユーザーにはルーチンを呼び出すための必須の特権が必要です。

ユーザー定義ルーチンとは、ユーザーが独自に作成するルーチンのことです。ユーザー定義ルーチンは、ユーザーが現在使用可能なサポートを超えて SQL 言語を拡張するための手段を提供します。ユーザー定義ルーチンは、組み込みルーチンのソース化 (そのロジックを再使用する)、SQL ステートメントのみの使用、または SQL および別のプログラミング言語の使用を含むさまざまな方法でインプリメントすることができます。

システム定義ルーチン

システム定義ルーチンは、製品に付属するルーチンです。このルーチンは、管理機能からデータベース・システムおよびカタログのレポート機能に至るタスクのための、幅広いルーチン・サポートを提供しています。これには、即使用可能で、前提条件のセットアップまたはルーチン登録ステップが不要であるという特徴があります。ただし、ユーザーにはルーチンを呼び出すための特権が必要です。これには、以前 SQL 管理ルーチンと呼ばれていた組み込みルーチンおよびシステム定義ルーチンを含めることができます。

組み込みシステム定義ルーチンは、標準的な演算子のサポートと、基本的なスカラー関数および集約関数のサポートを提供します。組み込みルーチンは、最初に使用するべきルーチンです。これは厳密に型付けされており、最高のパフォーマンスを提供するからです。組み込みルーチンの振る舞いと重複する外部ルーチンは作成しないでください。外部ルーチンには組み込みルーチンほどのパフォーマンスや安全性がありません。

DB2 の SYSPROC、SYSFUN、および SYSTOOLS の各スキーマには、使用できるその他のシステム定義ルーチンが用意されています。このルーチンは基本的に、システムによって定義され、製品に付属する SQL および外部ルーチンです。それらの追加ルーチンは DB2 に付属しているものですが、組み込みルーチンではありません。むしろ、事前にインストールされるユーザー定義ルーチンという形でインプリメントされています。この種のルーチンは、基本的にユーティリティー関数をカプセル化したものです。例えば、*SNAP_GET_TAB_V91*、*HEALTH_DB_HI*、*SNAP_WRITE_FILE*、*REBIND_ROUTINE_PACKAGE* などがあります。CURRENT PATH 特殊レジスターに SYSPROC スキーマと SYSFUN スキーマがあれば、これらの関数やプロシーチャーをすぐに使用できます。管理機能を実行する外部ルーチンのインプリメントを検討している場合には、DB2 に付属するシステム定義ルーチンのセットをよく調べるのが得策です。

特に `ADMIN_CMD` プロシージャは、SQL インターフェースを介して多くの一般的な DB2 コマンドを実行するための標準インターフェースを提供するため、便利かもしれません。

システム定義ルーチンは即使用可能なルーチンであるため、複雑な SQL 照会や強力なデータベース・アプリケーションをより短時間で簡単にインプリメントすることができます。

ユーザー定義ルーチン

DB2 には、頻繁に使用される算術関数、ストリング関数、`cast` 関数の機能を取り込んだルーチンが用意されています。しかし、DB2 では独自のロジックをカプセル化するためのルーチンを作成することも可能です。この種のルーチンのことをユーザー定義ルーチンといいます。つまり、各ルーチン・タイプでサポートされているいずれかのインプリメンテーション・スタイルで、ユーザーが独自の 12 ページの『ルーチン: プロシージャ』、13 ページの『ルーチン: 関数』、17 ページの『ルーチン: メソッド』を作成できるということです。通常、プロシージャやメソッドの場合に「ユーザー定義」という表現を使用することはあまりありません。ユーザー定義関数は一般に UDF と呼ばれます。

ユーザー定義ルーチンの作成

ユーザー定義のプロシージャ、関数、メソッドをデータベース内に作成するには、それぞれのルーチン・タイプに該当する `CREATE` ステートメントを実行します。この種のルーチン作成ステートメントには、次のものがあります。

- 「SQL リファレンス」内の『`CREATE PROCEDURE` ステートメント』
- 「SQL リファレンス」内の『`CREATE FUNCTION` ステートメント』
- 「SQL リファレンス」内の『`CREATE METHOD` ステートメント』

各 `CREATE` ステートメントに固有の節では、ルーチンの特性を定義します。具体的には、ルーチン名、ルーチンの引数の数とタイプ、ルーチン・ロジックの詳細などの特性があります。DB2 は、それらの節に指定されている情報に基づいて、ルーチンを識別し、呼び出しの時点でそのルーチンを実行します。ルーチンの `CREATE` ステートメントが正常に実行されると、データベース内にそのルーチンが作成されます。そのルーチンの特性は、ユーザーが照会できる DB2 カタログ・ビューに保管されます。`CREATE` ステートメントの実行によってルーチンを作成する作業のことをルーチンの定義またはルーチンの登録ともいいます。

ユーザー定義ルーチンの定義は、`SYSTOOLS` システム・カタログの表スキーマに保管されます。

ユーザー定義ルーチン・ロジックのインプリメンテーション

ルーチンのロジックを指定するために、以下の 3 つのインプリメンテーション・スタイルのいずれかを使用できます。

- 23 ページの『ソース派生ルーチンのインプリメンテーション』: ユーザー定義ルーチンは、既存の組み込みルーチンのロジックからのソース派生が可能です。
- 23 ページの『SQL ルーチンのインプリメンテーション』: ユーザー定義ルーチンは、SQL ステートメントだけを使用してインプリメントできます。

- 24 ページの『外部ルーチンのインプリメンテーション』: ユーザー定義ルーチンは、サポートされるプログラミング言語のセットの 1 つを使ってインプリメントできます。

非 SQL プログラミング言語でルーチンを作成すると、コードからビルドされるライブラリーまたはクラスは、EXTERNAL NAME 節で指定した値によって、ルーチン定義と関連付けられます。ルーチンが呼び出されると、そのルーチンと関連付けられているライブラリーまたはクラスが実行されます。

ユーザー定義ルーチンには、すべてではないものの、さまざまな SQL ステートメントが含まれます。

ユーザー定義ルーチンは厳密に型付けされたルーチンですが、型処理やエラー処理のメカニズムはルーチン開発者が開発または拡張する必要があります。

データベースをアップグレードするときに、ルーチンのインプリメンテーションの検査または更新が必要になる可能性があります。

通常、ユーザー定義ルーチンは高いパフォーマンスを発揮しますが、システム定義ルーチンほどではありません。

ユーザー定義ルーチンは、システム定義ルーチン、およびサポートされているいずれかの形式でインプリメントされている他のユーザー定義ルーチンを呼び出すことができます。この柔軟性により、ユーザーは、再利用可能なルーチン・モジュールの完全なライブラリーを基本的には自由にビルドすることができます。

一般に、ユーザー定義ルーチンは、システム定義ルーチンが存在しない場合に、SQL 言語を拡張し、複数の照会またはデータベース・アプリケーションで再利用できるロジックをモジュラー方式にするための手段を提供します。

システム定義ルーチンとユーザー定義ルーチンの比較

システム定義ルーチンとユーザー定義ルーチンの違いを理解しておくことは、実際に独自のルーチンをビルドする必要があるかどうか、または既存のルーチンを再利用できるかどうか判断する上で役に立ちます。既存のルーチンを再利用するべき時と独自のルーチンを作成するべき時を判断できると、時間と労力を節約することができます。ルーチンのパフォーマンスを最大限に引き出すことができます。

システム定義ルーチンとユーザー定義ルーチンには、さまざまな違いがあります。この違いについては、以下の表で要約されています。

表 1. システム定義ルーチンとユーザー定義ルーチンの比較

特性	システム定義ルーチン	ユーザー定義ルーチン
フィーチャー・サポート	<p>拡張数値オペレーター、ストリング処理、および管理機能を即時に使用できます。</p> <p>これらのルーチンは、サポートされるインターフェースからルーチンをただ呼び出すだけで使用できます。</p>	<p>ユーザー定義ルーチン内ですべての SQL ステートメントがサポートされているわけではありませんが、大多数はサポートされています。システム定義ルーチンの機能を拡張する場合は、システム定義ルーチンの呼び出しをユーザー定義ルーチンの中でラップさせることができます。ユーザー定義ルーチンは、ルーチンのロジックのインプリメンテーションに無限の機会を提供します。</p> <p>これらのルーチンを使用するにはまずそれらを作成しなければならず、その後、サポートされているインターフェースから呼び出すことができます。</p>
保守	保守は不要です。	外部ルーチンでは、関連付けられている外部ルーチン・ライブラリーの管理が必要です。
アップグレード	アップグレードの影響はほとんどあるいはまったくありません。	リリース間のアップグレードの場合、ルーチンの検査が必要になる場合があります。
パフォーマンス	同等のユーザー定義ルーチンよりも高いパフォーマンスを実現します。	通常、同等のシステム定義ルーチンほどのパフォーマンスは実現しません。
固定	エラー処理	ルーチンの開発者がエラー処理をプログラミングしなければなりません。

可能な場合にはいつも、システム定義ルーチンの使用を選択すべきです。システム定義ルーチンは、SQL ステートメントの編成とアプリケーション開発を簡単にするために提供されており、最高のパフォーマンスを発揮するよう最適化されています。ユーザー定義ルーチンは、インプリメントする特定のビジネス・ロジックを実行するシステム定義ルーチンがない場合に、柔軟に独自のルーチンをビルドすることができます。

システム定義ルーチンまたはユーザー定義ルーチンを使用する場合の判断

システム定義ルーチンは、時間の節約となるそのまますぐに使用できるカプセル化された機能を提供するのに対し、ユーザー定義ルーチンは、必要とされる機能が適切に含まれているシステム定義ルーチンがない場合に、独自のルーチンを定義する柔軟性を提供します。

システム定義ルーチンまたはユーザー定義ルーチンのいずれを使用するかを決定するには、以下のようにします。

1. ルーチンでカプセル化する機能を決定する。

2. 要求の一部またはすべてを満たすシステム定義ルーチンがあるかどうかを、選択可能なシステム定義ルーチンのリストからチェックする。
 - 要求の一部を満たすが、すべてを満たすわけではないシステム定義ルーチンがある場合は、以下のようにする。
 - 欠落している機能が、アプリケーションに簡単に追加できる機能であるかどうかを判別する。そうであれば、システム定義ルーチンを使用して、欠落した機能をカバーするようにアプリケーションを変更することができます。欠落している機能がアプリケーションに簡単には追加できない場合、または欠落している機能を多くの場所で繰り返す必要がある場合は、欠落した機能を含み、システム定義ルーチンを呼び出すユーザー定義ルーチンの作成を考慮してください。
 - ルーチンの要件が次第に変化していき、ルーチン定義の変更が頻繁に必要なことが予想される場合は、システム定義ルーチンではなくユーザー定義ルーチンを使用することを考慮してください。
 - ルーチンとの間で受け渡しする追加のパラメーターがあるかどうかを判別します。それがあある場合、システム定義ルーチンへの呼び出しをカプセル化するユーザー定義ルーチンの作成を考慮してください。
 - カプセル化する機能を適切に取り込めるシステム定義ルーチンがない場合は、ユーザー定義ルーチンを作成してください。

時間と労力を節約するために、可能な場合はいつでもシステム定義ルーチンの使用を考慮してください。必要とする機能が、システム定義ルーチンでは選択できない場合もあります。そのような場合は、ユーザー定義ルーチンを作成する必要があります。場合によっては、必要とする追加機能をカバーするユーザー定義ルーチンからの、システム定義ルーチンへの呼び出しを含めることもできます。

ルーチンの機能タイプ

ルーチンにはさまざまな機能タイプがあります。それぞれの機能タイプは、さまざまな目的に応じて、異なるインターフェースからルーチンを呼び出すためのサポートを提供します。それぞれのルーチンの機能タイプは、フィーチャーおよび SQL サポートの異なるセットを提供します。

- プロシージャ (ストアド・プロシージャともいう) は、クライアント・アプリケーション、ルーチン、トリガー、動的コンパウンド・ステートメントに対するサブルーチン拡張として機能します。プロシージャを呼び出すには、そのプロシージャへの参照を指定した CALL ステートメントを実行します。プロシージャは、入力、出力、および入出力パラメーターをとり、さまざまな SQL ステートメントを実行して、複数の結果セットを呼び出し側に戻すことができます。
- 関数とは、入力データ値の集合と結果値の集合とを関係付けるものです。関数を使用することで、SQL を拡張、カスタマイズすることができます。関数の呼び出しは、選択リスト、式、または FROM 節など、SQL ステートメントのエレメント内から行います。関数のタイプには、集約関数、スカラー関数、行関数、表関数の 4 つがあります。
- メソッドを使用することにより、ユーザー定義タイプの追加の動作を定義すると同時に、ユーザー定義タイプ属性にアクセスすることができます。構造化タイプは、1 つ以上の名前付き属性を含み、それぞれがデータ・タイプを持っている、

ユーザー定義データ・タイプです。属性は、タイプのインスタンスを記述するプロパティです。例えば、図形は、デカルト座標のリストといった属性を持っています。メソッドは、基本的に構造化タイプの属性に対する操作としてその構造化タイプに対してインプリメントします。図形であれば、その図形の体積を計算する目的などでメソッドを使用できます。

それぞれのルーチンの機能タイプについては、各ルーチン・タイプのトピックを参照してください。

ルーチン: プロシージャ

プロシージャ (ストアド・プロシージャともいう) は、CREATE PROCEDURE ステートメントを実行することによって作成されるデータベース・オブジェクトです。プロシージャは、ロジックおよび SQL ステートメントをカプセル化し、クライアント・アプリケーション、ルーチン、トリガー、動的コンパウンド・ステートメントに対するサブルーチン拡張として機能します。プロシージャを呼び出すには、そのプロシージャへの参照を指定した CALL ステートメントを実行します。プロシージャは、入力、出力、および入出力パラメータをとり、さまざまな SQL ステートメントを実行して、複数の結果セットを呼び出し側に戻すことができます。

フィーチャー

- 特定のサブルーチン・モジュールを定式化したロジック・エレメントと SQL ステートメントのカプセル化を使用可能にします。
- クライアント・アプリケーション、他のルーチン、トリガー、および動的コンパウンド・ステートメントなど、CALL ステートメントを実行可能な場所であればどこからでも呼び出せます。
- 複数の結果セットを戻します。
- 大きな SQL ステートメントのセットの実行をサポートします。単一パーティション・データベースまたは複数のパーティション・データベースにおいて、表データの読み取りや変更を行う SQL ステートメントの実行がサポートされます。
- 入力、出力、および入出力パラメータ用のパラメータ・サポート
- ネストされたプロシージャの呼び出しおよび関数呼び出しがサポートされます。
- プロシージャに対する再帰呼び出しがサポートされます。
- プロシージャ内でのセーブポイントおよびトランザクション制御がサポートされます。

制限

- CALL ステートメント以外の SQL ステートメント内からプロシージャを呼び出すことはできません。その代わりに、列の値を変換するロジックを記述するために、関数を使用できます。
- プロシージャ呼び出しの出力パラメータ値および結果セットを別の SQL ステートメントで直接使用することはできません。アプリケーション・ロジックを使用して、これらを後続の SQL ステートメントで使用可能な変数に割り当てる必要があります。
- プロシージャは、呼び出しと呼び出しの間で状態を保存できません。

- 『Restrictions on procedures』のトピックを参照してください。

一般的な使用法

- アプリケーション・ロジックの標準化
 - 複数のアプリケーションで、データベースに対して同じような参照、変更操作を実行する場合、1つのプロシージャによって、そのロジックのための統一的なインターフェースを提供できます。こうして、プロシージャを再使用することができます。ビジネス・ロジックの変更に伴ってインターフェースを変更する必要がある場合でも、その1つのプロシージャを変更するだけで済みます。
- アプリケーション内のデータベース操作と非データベース・ロジックとの分離
 - プロシージャは、複数のインスタンスで再使用できる特定のタスクに関連したロジックおよびデータベース・アクセスをカプセル化する、サブルーチンのインプリメンテーションを容易にします。例えば、従業員管理アプリケーションは、従業員を雇用するタスクに固有のデータベース操作をカプセル化することができます。この種のプロシージャでは、複数の表に従業員情報を挿入し、入力パラメーターに基づいて週給を計算して、出力パラメーターとして週給値を戻すことができます。また別のプロシージャでは、表内のデータの統計分析を行い、分析結果を含んだ結果セットを戻すこともできます。
- SQL ステートメントのグループに対する特権の管理の単純化
 - 複数の SQL ステートメントをグループ化して、1つの名前付きデータベース・オブジェクトにカプセル化することにより、プロシージャはデータベース管理者がより少ない特権を管理できるようにします。データベース管理者は、ルーチン内の各 SQL ステートメントの実行に必要な特権を付与する必要がなくなり、ルーチンを呼び出すための特権を管理するだけですみます。

サポートされるインプリメンテーション

- すぐに使用可能なシステム定義プロシージャがあります。あるいは、ユーザーがユーザー定義プロシージャを作成することもできます。プロシージャでは、以下のユーザー定義インプリメンテーションがサポートされています。
 - SQL インプリメンテーション
 - 外部インプリメンテーション
 - 『Supported external routine implementation programming languages』のトピックを参照してください。

ルーチン: 関数

関数とは、入力データ値の集合と結果値の集合とを関係付けるものです。それらを使用することで、SQL を拡張、カスタマイズすることができます。関数の呼び出しは、選択リストや FROM 節など、SQL ステートメントの要素内から行います。関数には次の 4 つのタイプがあります。

- 「SQL リファレンス 第 1 巻」内の『集約関数』
- スカラー関数
- 行関数

- 表関数

集約関数

集約関数 (列関数ともいう) は、類似した入力値のセットの評価結果としてスカラー値を戻します。類似した入力値の指定は、表内の列や VALUES 節内の組などで行えます。この値のセットのことを引数セットといいます。例えば、以下の照会は、SUM 集約関数を使用して、在庫と注文を合わせたボルトの合計数量を計算します。

```
SELECT SUM (qinstock + qonorder)
FROM inventory
WHERE description LIKE '%Bolt%'
```

スカラー関数

スカラー関数は、1 つ以上のスカラー・パラメーターのセットごとに 1 つのスカラー値を戻す関数です。例えば、LENGTH 関数や SUBSTR 関数などはスカラー関数です。また、関数入力パラメーターに対する複雑な数値計算を行うスカラー関数を作成することもできます。選択リストや FROM 節など、SQL ステートメント内で式が有効な場所であれば、どこからでもスカラー関数を参照できます。以下の例は、組み込み LENGTH スカラー関数を参照する照会を示しています。

```
SELECT lastname, LENGTH(lastname)
FROM employee
```

行関数 行関数は、1 つ以上のスカラー・パラメーターのセットごとに 1 つの行を戻す関数です。行関数は、構造化タイプの属性を行内の組み込みデータ・タイプ値にマップする Transform 関数としてのみ使用することができます。

表関数 表関数は、1 つ以上のパラメーターのセットのグループごとに、表を参照する SQL ステートメントにその表を戻す関数です。表関数は SELECT ステートメントの FROM 節内でしか参照できません。表関数から戻される表は、結合、グループ化演算、UNION のようなセット演算、読み取り専用ビューを対象とするあらゆる演算に組み込まれます。以下の例は、在庫表を更新して、その更新された在庫表に対する照会の結果セットを戻す SQL 表関数を示します。

```
CREATE FUNCTION updateInv(itemNo VARCHAR(20), amount INTEGER)
RETURNS TABLE (productName VARCHAR(20),
                quantity INTEGER)
LANGUAGE SQL
MODIFIES SQL DATA
BEGIN ATOMIC

    UPDATE Inventory as I
    SET quantity = quantity + amount
    WHERE I.itemID = itemNo;

    RETURN
    SELECT I.itemName, I.quantity
    FROM Inventory as I
    WHERE I.itemID = itemNo;
END
```

関数は、以下のフィーチャーのサポートを提供します。

- 関数は、DB2、DB2 for z/OS®、および DB2 Database for System i®、その他を含む、DB2 ブランドのデータベース製品全体に渡ってサポートされています。
- SQL ステートメントの実行を適度にサポートします。

- 入力パラメーターおよびスカラー関数または集約関数の戻り値に対するパラメーター・サポート
- 関数を参照する照会に、関数のロジックを効果的にまとめることができます。
- 外部関数は、各行または値ごとに個別の関数副次呼び出しの間の中間値を保管するサポートを提供します。

すぐに使用可能なシステム定義関数があります。あるいは、ユーザーがユーザー定義関数を作成することもできます。関数は、SQL 関数または外部関数としてインプリメントすることができます。SQL 関数はコンパイルまたはインライン化することができます。インライン関数は、コンパイルされた関数よりも速く実行できますが、実行できるのは SQL PL 言語のサブセットだけです。詳しくは CREATE FUNCTION ステートメントを参照してください。

ルーチン: スカラー関数: スカラー関数は、1 つ以上のスカラー・パラメーターのセットごとに 1 つのスカラー値を戻す関数です。例えば、LENGTH 関数や SUBSTR 関数などはスカラー関数です。また、関数入力パラメーターに対する複雑な数値計算を行うスカラー関数を作成することもできます。選択リストや FROM 節など、SQL ステートメント内で式が有効な場所であれば、どこからでもスカラー関数を参照できます。

フィーチャー

- 組み込みスカラー関数では優れたパフォーマンスが得られます。
- 組み込みスカラー関数は厳密に型付けされます。
- 式がサポートされている場所であればどこでも SQL ステートメントを使用して参照できます。
- ロジックが、それを参照する SQL ステートメントの一部としてサーバー上で実行されます。
- スカラー UDF の出力は、関数を参照するステートメントによって直接使用できます。
- スカラー UDF が述部で使用される場合、照会のパフォーマンス全体を向上させることができます。スカラー関数は、サーバーの候補行のセットに適用される場合にフィルターとして機能し、クライアントに戻さなければならない行の数を制限します。
- 外部スカラー・ユーザー定義関数の場合は、関数の反復呼び出しの際に、スクラッチパッドを使用して、呼び出しと呼び出しの間で状態を維持できます。

制限

- 設計上、単一のスカラー値のみが戻されます。
- トランザクション管理はスカラー関数内ではサポートされません。コミットおよびロールバックはスカラー関数本体内で実行できません。
- スカラー関数から結果セットを戻すことはできません。
- 単一パーティション・データベースでは、ユーザー定義の外部スカラー UDF に SQL ステートメントを含めることができます。これらのステートメントは、表のデータの読み取りは行えますが、その変更はできません。

- マルチパーティション・データベース環境では、ユーザー定義のスカラー UDF に SQL ステートメントを含めることはできません。
- 『Restrictions on scalar functions』を参照してください。

一般的な使用法

- SQL ステートメント内でストリングを操作します。
- SQL ステートメント内で基本の算術操作を実行します。
- ユーザー定義のスカラー関数を作成して、既存の組み込みスカラー関数のセットを拡張することができます。例えば、他のロジックとの組み合わせの中で既存の組み込みスカラー関数を再利用することにより、複雑な数学関数を作成できます。

サポートされるインプリメンテーション

- ソース派生インプリメンテーション
- 外部インプリメンテーション
 - 25 ページの『外部ルーチンの開発でサポートされている API とプログラミング言語の比較』というトピックを参照してください。

ルーチン: 行関数: 行関数は、ユーザー定義の構造化タイプでのみ使用可能な関数であり、1 つ以上のスカラー・パラメーターのセットごとに 1 つの行を戻します。行関数は、構造化タイプの属性を行内の組み込みデータ・タイプ値にマップする Transform 関数としてのみ使用することができます。行関数は、スタンドアロン方式や抽象データ・タイプのコンテキストの外部にある SQL ステートメント内では使用できません。

フィーチャー

- 構造化タイプ属性を組み込みデータ・タイプ値の行にマップすることができます。

制限

- スタンドアロン方式やユーザー定義の構造化タイプのコンテキストの外部にある SQL ステートメント内では使用できません。
- 『Restrictions on row functions』のトピックを参照してください。

一般的な使用法

構造化タイプ属性を照会または操作でアクセス可能にするためのものです。例えば、別の構造化タイプの人物を拡張し、人物属性とマネージャー固有の属性の組み合わせを持つ「manager」という名前のユーザー定義の構造化データ・タイプについて考えてみましょう。照会の中でこれらの値を参照する場合、属性値を参照可能な行の値に変換するための行関数を作成することで可能になります。

サポートされるインプリメンテーション

- SQL インプリメンテーション

ルーチン: 表関数: 表関数は、1 つ以上のパラメーターのセットのグループごとに、表を参照する SQL ステートメントにその表を戻す関数です。表関数は SELECT ステートメントの FROM 節内では参照できません。表関数から戻される表は、結合、グループ化演算、UNION のようなセット演算、読み取り専用ビューを対象とするあらゆる演算に組み込めます。

フィーチャー

- 処理用のデータ値のセットを戻します。
- SQL 照会の一部として参照できます。
- オペレーティング・システム呼び出しや、ファイルからのデータを読み取りはもちろん、単一パーティション・データベース内のデータにネットワーク経由でアクセスすることさえできます。
- 表関数呼び出しの結果は、表関数を参照する SQL ステートメントで直接アクセスできます。
- SQL 表関数は、SQL 表データを変更する SQL ステートメントをカプセル化できます。外部表関数は、SQL ステートメントをカプセル化できません。
- 1 つの表関数参照によって表関数を繰り返し何度も呼び出すことができ、スクラッチパッドを使用して呼び出しと呼び出しの間で状態を維持することができます。

制限

- トランザクション管理はユーザー定義表関数内ではサポートされません。コミットおよびロールバックは表 UDF 内で実行できません。
- 表関数から結果セットを戻すことはできません。
- 単一呼び出し用に設計されていません。
- 照会の FROM 節内でしか参照できません。
- ユーザー定義の外部表関数は、SQL データの読み取りはできますが、その変更は行えません。代わりに、SQL 表関数を使用して、SQL データを変更する SQL ステートメントを含めることができます。
- 『Restrictions on table functions』のトピックを参照してください。

一般的な使用法

- よく使用される複雑な副照会をカプセル化します。
- 非リレーショナル・データへの表インターフェースとして機能します。例えば、ユーザー定義の外部表関数は、スプレッドシートを読み取り、表に直接挿入できる値または照会内で直接かつ即時にアクセスできる値の表を作成できます。

サポートされるインプリメンテーション

- SQL インプリメンテーション
- 外部インプリメンテーション

ルーチン: メソッド

メソッドを使用することにより、構造化タイプの追加の動作を定義すると同時に、構造化タイプ属性にアクセスすることができます。構造化タイプは、1 つ以上の名前付き属性を含み、それぞれがデータ・タイプを持っている、ユーザー定義データ・タイプです。属性は、タイプのインスタンスを記述するプロパティです。例えば、図形は、デカルト座標のリストといった属性を持っています。

メソッドは、基本的に構造化タイプの属性に対する操作を表すために、その構造化タイプに対してインプリメントします。図形であれば、その図形の体積を計算する目的などでメソッドを使用できます。メソッドは、スカラー関数のすべてのフィーチャーを共有します。

フィーチャー

- 構造化タイプの属性にアクセスする機能。
- 構造化タイプの属性を設定する機能。
- 構造化タイプの属性に対する操作を作成し、関数値を戻す機能
- サブジェクト・タイプの動的タイプに重点を置く

制限

- スカラー値しか戻すことはできません。
- 構造化タイプに対してしか使用できません。
- 型付き表に対して呼び出すことはできません。

一般的な使用法

- 構造化タイプに対する操作の作成
- 構造化タイプのカプセル化

サポートされるインプリメンテーション

システム定義のメソッドはありません。ユーザーは、既存のユーザー定義の構造化タイプのためのユーザー定義のメソッドを作成できます。メソッドは、以下のいずれかのインプリメンテーションを使用してインプリメントできます。

- 23 ページの『SQL ルーチンのインプリメンテーション』
- 24 ページの『外部ルーチンのインプリメンテーション』：
C、C++、Java、C# (OLE API を使用)、Visual Basic (OLE API を使用)

SQL メソッドは簡単にインプリメントできますが、一般に構造化タイプの設計と組み合わせで設計されています。外部メソッドは、ロジックの柔軟なインプリメンテーションを大幅にサポートし、ユーザーが自分の希望するプログラミング言語でメソッド・ロジックを開発できるようにします。

ルーチンの機能タイプの比較

プロシージャ、関数、およびメソッドの違いについて理解することは、独自のルーチンをビルドする際にどの機能タイプをインプリメントするかを決定するのに役立ちます。また、どこでどのように既存のルーチンを参照するかを決定するのにも役立ちます。これにより、時間と労力を節約して、ルーチンの機能およびパフォーマンスを最大限に引き出すことができます。

12 ページの『ルーチン: プロシージャ』、13 ページの『ルーチン: 関数』、および 17 ページの『ルーチン: メソッド』には、さまざまな違いがあります。これらの相違点は、以下の表で説明されています。

表 2. ルーチンの機能タイプの比較

特性	プロシージャ	関数	メソッド
固有の機能特性および役立つアプリケーション	<ul style="list-style-type: none"> ロジックおよび SQL ステートメントのカプセル化を使用可能にします。 クライアント・アプリケーション、ルーチン、トリガー、および動的コンパウンド・ステートメントに対するサブルーチン拡張として機能します。 プロシージャを呼び出すには、そのプロシージャへの参照を指定した CALL ステートメントを実行します。 ネストされたプロシージャの呼び出しがサポートされます。 再帰プロシージャの呼び出しがサポートされません。 入力、出力、および入出力パラメーター用のパラメーター・サポート SQL ステートメント実行に対する拡張サポート 1 つ以上の結果セットを戻すことができます。 セーブポイントおよびトランザクション制御 	<ul style="list-style-type: none"> ロジックおよび SQL ステートメントのカプセル化を使用可能にします。 関数とは、入力データ値の集合と結果値の集合とを関係付けるものです。 関数を使用することで、SQL を拡張、カスタマイズすることができます。 関数の呼び出しは、選択リストや FROM 節など、SQL ステートメントのエレメント内から行います。 SQL ステートメントの実行を適度にサポートします。 入力パラメーターおよびスカラー関数または集約関数の戻り値に対するパラメーター・サポート。 外部関数は、スクラッチパッドを使用して、各行または値ごとに個別の関数副次呼び出しの間の間値を保管するサポートを提供します。 関数を参照する照会に、関数のロジックを効果的にまとめることができます。 	<ul style="list-style-type: none"> ロジックおよび SQL ステートメントのカプセル化を使用可能にします。 メソッドを使用することにより、構造化タイプの追加の動作を定義すると同時に、構造化タイプ属性にアクセスすることができます。 構造化タイプの属性にアクセスする機能。 構造化タイプの属性を設定する機能。 構造化タイプの属性に対する操作を作成し、関数値を戻す機能。
ルーチンの機能サブタイプ	<ul style="list-style-type: none"> 適用外 	<ul style="list-style-type: none"> スカラー関数 集約関数 行関数 表関数 	<ul style="list-style-type: none"> 適用外
呼び出しインターフェース	<ul style="list-style-type: none"> 呼び出しを行うには、そのプロシージャへの参照を指定した CALL ステートメントを実行します。 CALL ステートメントがサポートされる場合にはいつでもプロシージャの呼び出しがサポートされます。 	<ul style="list-style-type: none"> 呼び出しは、列の選択リスト、式、あるいは他の場所にある SELECT ステートメントの FROM 節内の SQL ステートメント内で行われます。 	<ul style="list-style-type: none"> 呼び出しは、メソッドに関連した構造化タイプを参照する SQL ステートメント内で行われます。

表 2. ルーチンの機能タイプの比較 (続き)

特性	プロシージャ	関数	メソッド
このタイプのシステム定義ルーチンが存在するか?	<ul style="list-style-type: none"> 多く存在します。 システム定義のプロシージャのリストについては、「SQL リファレンス」を参照してください。 	<ul style="list-style-type: none"> 多く存在します。 システム定義の関数のリストについては、「SQL リファレンス」を参照してください。 	<ul style="list-style-type: none"> いいえ
サポートされるユーザー定義ルーチンのインプリメンテーション	<ul style="list-style-type: none"> SQL 外部 <ul style="list-style-type: none"> - C/C++ (組み込み SQL または CLI API 呼び出しを使用) - COBOL - Java (JDBC) - Java (SQLJ) - .NET CLR - OLE: Visual Basic, Visual C++ 	<ul style="list-style-type: none"> SQL 外部 <ul style="list-style-type: none"> - C/C++ - Java (JDBC) - Java (SQLJ) - .NET CLR - OLE DB: Visual Basic, Visual C++ (表関数のみ) 	<ul style="list-style-type: none"> SQL 外部 <ul style="list-style-type: none"> - C - C++
ネストされた呼び出しのサポート	<ul style="list-style-type: none"> はい 	<ul style="list-style-type: none"> なし。ただし、関数は入力セットの値ごとに繰り返し呼び出され、スクラッチパッドを使用して中間値を保管することができます。 	<ul style="list-style-type: none"> いいえ
パフォーマンス	<ul style="list-style-type: none"> ルーチン・ロジックが効率的であり、ベスト・プラクティスが採用される場合、パフォーマンスは良好です。 	<ul style="list-style-type: none"> ルーチン・ロジックが効率的であり、ベスト・プラクティスが採用される場合、パフォーマンスは良好です。 ロジックがデータのみを照会し、データを変更しない場合、論理的に同等のプロシージャよりも優れたパフォーマンスを得ることができます。 	<ul style="list-style-type: none"> 良好なパフォーマンス
移植性	<ul style="list-style-type: none"> 移植性が高い SQL インプリメンテーションが使用される場合は特に移植可能。 さまざまなプログラミング言語でサポートされる 32 ビットおよび 64 ビットの外部ルーチン 	<ul style="list-style-type: none"> 移植性が高い SQL インプリメンテーションが使用される場合は特に移植可能。 さまざまなプログラミング言語でサポートされる 32 ビットおよび 64 ビットの外部ルーチン 	<ul style="list-style-type: none"> 移植性が高い

表 2. ルーチンの機能タイプの比較 (続き)

特性	プロシージャ	関数	メソッド
インターオペラビリティ	<ul style="list-style-type: none"> また、SQL アクセス・レベルがそのプロシージャより小さいか等しい関数を呼び出す SQL ステートメントを含めることができます。SQL アクセス・レベルは、CREATE PROCEDURE ステートメントで指定する次のものことで、アクセス・レベルが小さい順に、NO SQL、CONTAINS SQL、READS SQL DATA、MODIFIES SQL DATA となります。 	<ul style="list-style-type: none"> 関数は、その他の関数を呼び出す SQL ステートメントを含めることができ、関数の SQL アクセス・レベルより低い等しい SQL アクセス・レベルで、プロシージャを呼び出すことができます。 	<ul style="list-style-type: none"> メソッドは、メソッドの SQL アクセス・レベルより低い等しい SQL アクセス・レベルで、関数を呼び出すことができます。 メソッドは、プロシージャまたは他のメソッドを呼び出すことはできません。
制約事項		<ul style="list-style-type: none"> 表関数は、SELECT ステートメントの FROM 節内で参照される単一の表参照のみを戻すことができます。 	

一般に、ルーチンの機能特性および用途によって、使用するルーチンのタイプが決定します。しかし、パフォーマンスおよびサポートされるルーチンのインプリメンテーションも、使用するルーチンのタイプを決定する上で重要な役割を果たします。

使用するルーチン機能タイプの決定

プロシージャ、関数、およびメソッドは、さまざまな関数ルーチンおよびフィーチャー・サポートを提供します。どのルーチン・タイプを使用するか、インプリメントするかを決めることにより、ルーチンの機能をどこでどのように参照して呼び出せるかが決まることになります。また、どのルーチン・インプリメンテーションを使用できるかにも影響し、どのタイプの機能をルーチンに含めることができるかにも影響することがあります。ルーチンのインプリメントを開始する前にどのルーチン・タイプがニーズに最適であるかを判別しておけば、後々時間の節約になり、いらいらする可能性もなくなります。

ルーチン機能タイプの説明を読んで、それぞれの特性を確認してください。

プロシージャ、関数、メソッドのいずれを使用するかを決定するには、以下のようになります。

1. ルーチンにどの機能をカプセル化するか、ルーチンをどのインターフェースから呼び出すか、どのルーチン・インプリメンテーションを使用するかを決定する。

- 以下のトピックを参照する。

- 18 ページの『ルーチンの機能タイプの比較』

どのルーチン機能タイプがこうした要件をサポートするかを判別できます。

2. ルーチンにどの SQL ステートメントを組み込むかを決定する。
 - 以下のトピックを参照する。
 - 39 ページの『ルーチン内で実行できる SQL ステートメント』
 - どの機能ルーチンが必要な SQL ステートメントの実行をサポートするかを判別します。
3. ルーチンに組み込むものが 1 つ以上の照会だけの場合は、SQL 関数を使用することを考慮してください。こうした状況では SQL 関数のほうがパフォーマンスが優れています。なぜなら、別々にコンパイルされて呼び出されるプロシージャとは違って、SQL 関数は、それを参照する SQL ステートメントと一緒にインライン・コンパイルが行われるからです。
4. ルーチンの機能を将来拡張して別のルーチン・タイプの機能を組み込むことが必要になるかどうかを判断する (例えば、プロシージャは関数よりも多くの SQL ステートメントをサポートし、また一般的には関数よりも多くの SQL フィーチャーをサポートします)。後で関数をプロシージャに書き直さなくてもすむように、この時点でプロシージャのインプリメントを検討してください。

どのルーチン機能タイプをインプリメントするかを選択では、一般的には機能と SQL の要件が選択の動機になりますが、別の機能タイプと論理的に同等のルーチンを作成できる場合もあります。例えば、単一の結果セットを表関数として戻す基本プロシージャのほとんどは書き直すことができます。スカラー関数の単一出力パラメーターだけを持つ基本プロシージャも簡単に書き直すことができます。

どのルーチン機能タイプを使用するかを決定したなら、次はルーチン・インプリメンテーションの詳細を確認したり、どのルーチン・インプリメンテーションを使用するかを決定したりすることが課題になります。

ルーチンのインプリメンテーション

ルーチンのインプリメントはさまざまな方法で行えます。ルーチンのインプリメンテーションは基本的に、ルーチンが呼び出されるときに実行されるロジックを含むルーチンの基底形式です。サポートされているさまざまなルーチンのインプリメンテーションを理解することは、ルーチンの働きを理解し、ユーザー定義ルーチンをインプリメントする際に選択するルーチンのインプリメンテーションを決定するのに役立ちます。

使用可能なルーチンのインプリメンテーションとして次のものがあります。

- 組み込みルーチン
- ソース派生ルーチン
- SQL ルーチン
- 外部ルーチン

システム定義ルーチンは、組み込みルーチン、SQL ルーチン、または外部ルーチンとしてインプリメントできます。ただし、そのインプリメンテーションは基本的にはユーザーには見ることができず、通常はユーザーにはほとんど関係がありません。

ユーザー定義ルーチンは、ソース派生ルーチン、SQL ルーチン、または外部ルーチンとしてインプリメントできます。

インプリメンテーションの特性はそれぞれ異なり、多少なりとも機能サポートにつながります。特定のインプリメンテーションに決定する前に、各インプリメンテーションに関連したサポートされる機能および制限を検討することが得策です。各インプリメンテーションについて読んでから、次のトピックをお読みください。

- 30 ページの『ルーチンのインプリメンテーションの比較』

ルーチンのインプリメンテーションを十分理解しておくことは、インプリメンテーションについて適切な決定を下し、既存のルーチンのデバッグおよびトラブルシューティングを行う上で役に立ちます。

組み込みルーチンのインプリメンテーション

組み込みルーチンは DB2 データベース・マネージャーのコードにあらかじめ組み込まれています。この種のルーチンは厳密に型付けされており、データベース・コードに対してネイティブなロジックを持っているので、高い機能性を発揮します。これらのルーチンは SYSIBM スキーマの中にあります。組み込みスカラー関数と組み込み集約関数の例を以下に示します。

- 組み込みスカラー関数: +、-、*、/、substr、concat、length、char、decimal、days
- 組み込み集約関数: avg、count、min、max、stdev、sum、variance

組み込み関数は、頻繁に必要なほとんどのタイプのキャスト、ストリング処理、算術計算機能に対応しています。これらの関数は、SQL ステートメントの中ですぐに使用できます。利用可能な組み込み関数の完全なリストについては、「SQL リファレンス」を参照してください。

ソース派生ルーチンのインプリメンテーション

ソース派生ルーチンのインプリメンテーションでインプリメントされたルーチンは、別の関数（ソース関数）のセマンティクスを複製したルーチンです。現時点で、ソース派生関数になれるのは、スカラー関数と集約関数だけです。ソース派生関数が特に便利なのは、ソース・タイプのセマンティクスを選択的に継承した特殊タイプを使用する場合です。ソース派生関数は基本的に、関数の SQL インプリメンテーションの特別な形式と言えます。

SQL ルーチンのインプリメンテーション

SQL ルーチンのインプリメンテーションは、SQL ステートメントだけで構成されています。SQL ルーチンのインプリメンテーションの特徴は、SQL ステートメントが、データベースでのルーチンの作成に使用される CREATE ステートメント内に組み込まれるルーチンのロジックを定義しているという点です。SQL ルーチンは構文が単純なので、短時間で簡単にインプリメントできます。また、DB2 との関係が近いのでパフォーマンスも優れています。

SQL プロシージャ型言語 (SQL PL) は基本 SQL の言語拡張であり、SQL でプログラミング・ロジックをインプリメントするためのステートメントと言語エレメントから成っています。SQL PL には、変数や条件ハンドラーを宣言するためのステートメント (DECLARE ステートメント)、変数に値を割り当てるためのステートメント (代入文)、プロシージャ・ロジックをインプリメントするためのステートメント (IF、WHILE、FOR、GOTO、LOOP、SIGNAL などの制御ステートメント) のセットが含まれています。SQL プロシージャ、関数、トリガー、およびコンパウンド SQL ステートメントの作成には、SQL と SQL PL を使用できます。SQL

プロシージャと関数は、SQL グローバル変数、ユーザー定義条件、データ・タイプと組み合わせてモジュール化することができます。

外部ルーチンのインプリメンテーション

外部ルーチンのインプリメンテーションは、データベースの外部に存在するプログラミング言語のコードによってルーチンのロジックが定義されるインプリメンテーションです。他のルーチンのインプリメンテーションと同様に、外部インプリメンテーションのルーチンも CREATE ステートメントを実行することによりデータベースに作成されます。コンパイルされたライブラリーに保管されるルーチンのロジックは、データベース・サーバーの特別なディレクトリー・パスにあります。ルーチン名と外部コード・アプリケーションとの関連付けは、CREATE ステートメントの EXTERNAL 節で宣言します。

外部ルーチンは、サポートされるどの外部ルーチンのプログラミング言語でも作成できます。

外部ルーチンのインプリメンテーションは、SQL ルーチンのインプリメンテーションよりもいくらか複雑かもしれませんが、しかし、選択したインプリメンテーション・プログラミング言語の全機能とパフォーマンスを活用できるという点で非常に強力です。また、外部関数には、データベースの外部 (つまり、ネットワークやファイル・システムなど) に存在するエンティティーにアクセスして操作を実行できるというメリットもあります。DB2 データベースとの対話はそれほど必要としないものの、大量のロジックや非常に複雑なロジックを組み込む必要があるルーチンの場合は、外部ルーチンのインプリメンテーションが望ましいと言えます。

例えば、VARCHAR データ・タイプを操作する新しいストリング関数や、DOUBLE データ・タイプを操作する複雑な数学関数など、組み込みデータ・タイプの利便性を活用する新しい関数をインプリメントするときには、外部ルーチンを使用するのが理想的です。さらに、外部ルーチンのインプリメンテーションは、E メール送信などの外部アクションを伴うロジックにも最適です。

データ・アクセスよりもプログラミング・ロジックを重視してロジックをカプセル化する必要がある場合、サポートされているいずれかの外部ルーチンのプログラミング言語によるプログラミングが苦にならないのであれば、外部インプリメンテーションのルーチンの作成手順をマスターした時点で、外部ルーチンがいかに強力かをすぐに実感できるはずです。

外部ルーチンの開発でサポートされている API およびプログラミング言語: 以下の API および関連したプログラミング言語を使用して、DB2 外部ルーチン (プロシージャおよび関数) を開発することができます。

- ADO.NET
 - .NET 共通言語ランタイム・プログラミング言語
- CLI
- 組み込み SQL
 - C
 - C++
 - COBOL (プロシージャでのみサポート)
- JDBC

- Java
- OLE
 - Visual Basic
 - Visual C++
 - この API をサポートするその他のプログラミング言語。
- OLE DB (表関数でのみサポートされる)
 - この API をサポートするプログラミング言語。
- SQLJ
 - Java

外部ルーチンの開発でサポートされている API とプログラミング言語の比較: 外部ルーチンのインプリメントを開始する前に、サポートされているさまざまな外部ルーチンのアプリケーション・プログラミング・インターフェース (API) およびプログラミング言語の特性および制限について考慮することは重要です。これによって最初から正しいインプリメンテーションを選ぶことができ、必要なルーチンのフィーチャーを使用することができます。

表 3. 外部ルーチン API とプログラミング言語の比較

API とプログラミング言語	フィーチャー・サポート	パフォーマンス	セキュリティ	スケーラビリティ	制限
SQL (SQL PL を含む)	<ul style="list-style-type: none"> • SQL は、容易に学習および使用できるハイレベル言語であり、インプリメンテーションを素早く実行できるようにします。 • SQL プロシージャ型言語 (SQL PL) エレメントは、SQL 操作および照会において制御フロー・ロジックを許可します。 	<ul style="list-style-type: none"> • 非常に良い。 • SQL ルーチンは、Java ルーチンよりも高いパフォーマンスを実現します。 • SQL ルーチンのパフォーマンスは、NOT FENCED 節で作成される C および C++ 外部ルーチンと同程度です。 	<ul style="list-style-type: none"> • 非常に安全。 • SQL プロシージャは、常にデータベース・マネージャーと同じメモリーで稼働します。これは、キーワード NOT FENCED を使ってデフォルトで作成されるルーチンに対応します。 	<ul style="list-style-type: none"> • 高いスケーラビリティ。 	<ul style="list-style-type: none"> • データベース・サーバーのファイル・システムにアクセスできません。 • データベースの外部に常駐するアプリケーションを呼び出すことはできません。

表 3. 外部ルーチン API とプログラミング言語の比較 (続き)

API とプログラミング言語	フィーチャー・サポート	パフォーマンス	セキュリティ	スケーラビリティ	制限
<p>組み込み SQL (C および C++ を含む)</p>	<ul style="list-style-type: none"> 低いレベルではあるが、強力なプログラミング言語。 	<ul style="list-style-type: none"> 非常に良い。 C および C++ ルーチンは、Java ルーチンよりも高いパフォーマンスを実現します。 NOT FENCED 節で作成される C および C++ 外部ルーチンのパフォーマンスは、SQL ルーチンと同程度です。 	<ul style="list-style-type: none"> C および C++ ルーチンではプログラミング・エラーが生じやすくなります。 プログラマーは、ルーチンのインプリメンテーションを単調にして時間を浪費させる共通メモリーおよびポインター操作のエラーを避けるため、C に熟達している必要があります。 C および C++ ルーチンは、データベース・マネージャーの混乱によって実行時にルーチン内で例外が発生するのを避けるため、FENCED 節および NOT THREADSAFE 節を使用して作成しなければなりません。これらはデフォルトの節です。これらの節を使用することで、パフォーマンスにいくらか悪い影響が出る場合がありますが、安全な実行が保証されます。ルーチンのセキュリティを参照してください。 	<ul style="list-style-type: none"> C および C++ ルーチンが FENCED 節および NOT THREADSAFE 節で作成されている場合、スケーラビリティは削減されます。これらのルーチンは、データベース・マネージャー・プロセスとは別に、分離した <i>db2fmp</i> プロセス内で実行されます。<i>db2fmp</i> プロセスは、並行して実行されるルーチンごとに必要になります。 	<ul style="list-style-type: none"> サポートされるパラメーター引き渡しスタイルは複数存在するので、混乱してしまう可能性があります。ユーザーは、可能な限りパラメーター・スタイル SQL を使用する必要があります。

表 3. 外部ルーチン API とプログラミング言語の比較 (続き)

API とプログラミング言語	フィーチャー・サポート	パフォーマンス	セキュリティ	スケーラビリティ	制限
組み込み SQL (COBOL)	<ul style="list-style-type: none"> • ビジネス (通常はファイルを取り扱う) アプリケーションの開発に適した高水準プログラミング言語。 • これまでは実動ビジネス・アプリケーション用に広く使用されてきました。しかし、その普及度は減少しています。 • COBOL は、ポインター、および、再帰的な呼び出しをサポートしていないプログラミング言語です。 	<ul style="list-style-type: none"> • COBOL ルーチンは、他の外部ルーチンのインプリメンテーション・オプションを指定して作成されたルーチンと比較してパフォーマンスが劣ります。 	<ul style="list-style-type: none"> • 現時点で情報はありません。 	<ul style="list-style-type: none"> • 現時点で情報はありません。 	<ul style="list-style-type: none"> • 32 ビットの COBOL プロシージャを 64 ビットの DB2 インスタンス内に作成して呼び出すことができますが、これらのルーチンは 64 ビットの DB2 インスタンス内にある 64 ビットの COBOL プロシージャと比較してパフォーマンスが劣ります。
JDBC (Java) および SQLJ (Java)	<ul style="list-style-type: none"> • スタンドアロン・アプリケーション、アプレット、およびサーブレットの開発に適した、高水準のオブジェクト指向プログラミング言語。 • Java オブジェクトおよびデータ・タイプは、データベース接続の確立、SQL ステートメントの実行、およびデータの操作を容易にします。 	<ul style="list-style-type: none"> • Java ルーチンは、C および C++ ルーチンまたは SQL ルーチンと比較してパフォーマンスが劣ります。 	<ul style="list-style-type: none"> • Java ルーチンでは、危険操作の制御は Java 仮想マシン (JVM) によって担われるので、Java ルーチンのほうが C および C++ ルーチンよりも安全です。これにより、信頼性は向上し、1 つの Java ルーチンのコードが、同じプロセス内で実行中の別のルーチンに悪影響を与えることはほとんどありません。 	<ul style="list-style-type: none"> • 優れたスケーラビリティ • FENCED THREADSAFE 節で作成された Java ルーチン (デフォルト) は、スケーラビリティに優れています。fenced の Java ルーチンはすべて、いくつかの JVM を共用します。特定の db2fmp プロセスの Java ヒープが使い果たされると、システムでは複数の JVM が起用されるからです。 	<ul style="list-style-type: none"> • 危険性を含んだ操作を避けるために、Java ルーチンから Java Native Interface (JNI) 呼び出しを行うことはできないことになっています。

表 3. 外部ルーチン API とプログラミング言語の比較 (続き)

API とプログラミング言語	フィーチャー・サポート	パフォーマンス	セキュリティ	スケーラビリティ	制限
<p>.NET 共通言語ランタイムのサポートされる言語 (C#, Visual Basic、およびその他を含む)</p>	<ul style="list-style-type: none"> 管理対象コードの Microsoft® .NET モデルの一部です。 ソース・コードは Microsoft .NET Framework 共通言語ランタイムで解釈できる中間言語 (IL) バイト・コードにコンパイルされます。 CLR アセンブリは、別の .NET プログラム言語のソース・コードからコンパイルしたサブアセンブリからでもビルドできます。つまり、ユーザーとしては、さまざまな言語で作成したコード・モジュールの再利用と統合が可能になります。 	<ul style="list-style-type: none"> CLR ルーチンは、実行時にデータベース・マネージャーの割り込みの可能性を最小限に抑えるために、FENCED NOT THREADSAFE 節でのみ作成できます。これにより、パフォーマンスにいくらか悪影響が出る可能性があります。 デフォルト節値を使用するならば、実行時にデータベース・マネージャーによる割り込みが起きる可能性は小さくなりますが、CLR ルーチンは FENCED として実行する必要があるため、NOT FENCED として指定できる他の外部ルーチンと比べるとわずかに実行速度が遅くなる可能性があります。 	<ul style="list-style-type: none"> CLR ルーチンは、FENCED NOT THREADSAFE 節でのみ作成できます。このルーチンはデータベース・マネージャーの外の、別個の db2fmp プロセスで実行されるので、安全性が確保されます。 	<ul style="list-style-type: none"> 入手可能な情報ははありません。 	<ul style="list-style-type: none"> 『.NET CLR ルーチンに関する制約事項』のトピックを参照してください。

表 3. 外部ルーチン API とプログラミング言語の比較 (続き)

API とプログラミング言語	フィーチャー・サポート	パフォーマンス	セキュリティ	スケーラビリティ	制限
<ul style="list-style-type: none"> OLE 	<ul style="list-style-type: none"> OLE ルーチンは、Visual C++、Visual Basic、および OLE でサポートされているその他の言語でインプリメントすることができます。 	<ul style="list-style-type: none"> OLE 自動化ルーチンの速度は、インプリメントに使用する言語によって異なります。一般的にこのルーチンは、OLE C/C++ 以外のルーチンよりも遅いです。 OLE ルーチンは、FENCED NOT THREADSAFE モードでのみ実行できるため、OLE 自動化ルーチンはスケーラビリティにはあまり優れていません。 	<ul style="list-style-type: none"> 入手可能な情報はあります。 	<ul style="list-style-type: none"> 入手可能な情報はあります。 	<ul style="list-style-type: none"> 入手可能な情報はあります。

表 3. 外部ルーチン API とプログラミング言語の比較 (続き)

API とプログラミング言語	フィーチャー・サポート	パフォーマンス	セキュリティ	スケーラビリティ	制限
<ul style="list-style-type: none"> OLE DB 	<ul style="list-style-type: none"> OLE DB は、ユーザー定義の表関数の作成に使用できます。 OLE DB 関数は、外部の OLE DB データ・ソースに接続します。 	<ul style="list-style-type: none"> OLE DB 関数のパフォーマンスは、OLE DB Provider によって異なります。ただし、一般に OLE DB 関数は、論理的に同等な Java 関数よりパフォーマンスが優れていますが、論理的に同等な C、C++、または SQL 関数より速度が遅くなります。ただし関数が呼び出される場所である照会内の特定の述部を OLE DB Provider で評価することができるので、DB2 が処理しなければならない行数は減ります。多くの場合、それによってパフォーマンスが向上することになります。 	<ul style="list-style-type: none"> 入手可能な情報はありません。 	<ul style="list-style-type: none"> 入手可能な情報はありません。 	<ul style="list-style-type: none"> OLE DB は、ユーザー定義の表関数の作成にのみ使用できません。

ルーチンのインプリメンテーションの比較

サポートされるルーチンのインプリメンテーションの違いについて理解することは、独自のルーチンをビルドする際に使用するルーチンのインプリメンテーションを決定するのに役立ちます。これにより、時間と労力を節約して、ルーチンの機能およびパフォーマンスを最大限に引き出すことができます。

組み込み、ソース派生、SQL、および外部ルーチンのインプリメンテーションは多くの点で異なります。これらの相違点は、以下の表で説明されています。

表4. ルーチンのインプリメンテーションの比較

特性	組み込み	ソース派生	SQL	外部
フィーチャーおよび使用法	<ul style="list-style-type: none"> データベース・マネージャー・コードに対してネイティブなロジックを持っているので、高い機能性を発揮します。 多くの一般的なキャスト関数、ストリング処理関数、および算術計算組み込み関数はSYSIBMスキーマ内にあります。 	<ul style="list-style-type: none"> 組み込み関数の機能に基本的な拡張機能を提供するために使用されません。 SQL および SQL PL は、ルーチン・ロジックのインプリメントを迅速かつ簡易に行う、ハイレベルのプログラミング言語サポートを提供しません。 	<ul style="list-style-type: none"> SQL ステートメントを実行できる、より複雑な関数によって、組み込み関数のセットを拡張するために使用されます。 	<ul style="list-style-type: none"> 開発者は、自分の選択したサポートされるプログラミング言語でロジックをプログラムできます。 複雑なロジックをインプリメントすることができます。 外部アクション (データベースの外部で影響力を持つアクション) が直接サポートされます。これには、サーバー・ファイル・システムに対する読み取りまたは書き込み、サーバーでのアプリケーションまたはスクリプトの呼び出し、SQL、ソース派生、または組み込みインプリメンテーションでサポートされない SQL ステートメントの発行が含まれます。
インプリメンテーションがデータベース・マネージャー・コードに組み込まれているか?	<ul style="list-style-type: none"> はい 	<ul style="list-style-type: none"> いいえ 	<ul style="list-style-type: none"> いいえ 	<ul style="list-style-type: none"> いいえ
このインプリメンテーションを含めることができる、サポートされるルーチンの機能タイプ	<ul style="list-style-type: none"> 適用外 	<ul style="list-style-type: none"> 関数 <ul style="list-style-type: none"> スカラー関数 集約関数 	<ul style="list-style-type: none"> プロシージャー 関数 メソッド 	<ul style="list-style-type: none"> プロシージャー 関数 メソッド

表4. ルーチンのインプリメンテーションの比較 (続き)

特性	組み込み	ソース派生	SQL	外部
サポートされる SQL ステートメント	<ul style="list-style-type: none"> 適用外 	<ul style="list-style-type: none"> 適用外 	<ul style="list-style-type: none"> SQL PL ステートメントのすべてを含む、ほとんどの SQL ステートメントをルーチン内で実行することができます。 『ルーチン内で実行できる SQL ステートメント』のトピックを参照してください。 	<ul style="list-style-type: none"> SQL PL ステートメントのサブセットを含む、多くの SQL ステートメントをルーチン内で実行することができます。 『ルーチン内で実行できる SQL ステートメント』のトピックを参照してください。
パフォーマンス	<ul style="list-style-type: none"> 高速 	<ul style="list-style-type: none"> 一般に、組み込み関数と同程度の速さです。 	<ul style="list-style-type: none"> SQL が効率的に書き込まれた場合、データベース操作がプログラミング・ロジックよりも強調された場合、および SQL ルーチンのベスト・プラクティスが採用された場合に、非常によいパフォーマンスが得られます。 『SQL routine best practices』のトピックを参照してください。 	<ul style="list-style-type: none"> プログラミング・ロジックが効率的に書き込まれた場合、および外部ルーチンのベスト・プラクティスが採用された場合に、非常によいパフォーマンスが得られます。 『External routine best practices』のトピックを参照してください。
移植性	<ul style="list-style-type: none"> 適用外 	<ul style="list-style-type: none"> ソース派生関数は、ドロップして他の DB2 データベースに再作成するのが容易です。 	<ul style="list-style-type: none"> SQL 関数は、他のデータベースで簡単にドロップおよび再作成することができます。 	<ul style="list-style-type: none"> 外部関数は、他のデータベースでドロップおよび再作成することができません。ただし、環境に互換性があり、必要とするサポートされるソフトウェアが使用可能であるかどうかを確かめるために注意を払う必要があります。 『Deploying external routines』のトピックを参照してください。

表 4. ルーチンのインプリメンテーションの比較 (続き)

特性	組み込み	ソース派生	SQL	外部
インターオペラビリティ	<ul style="list-style-type: none"> 適用外 	<ul style="list-style-type: none"> これは、組み込み関数を参照できる場所であればどこでも参照できます。ソース派生関数は他の関数を呼び出すことはできません。 	<ul style="list-style-type: none"> SQL ルーチンは、SQL ステートメントの多くの部分で参照することができます。SQL ルーチンは、SQL ルーチンの SQL アクセス・レベルより低いか等しい SQL アクセス・レベルで、他の SQL および外部ルーチンを呼び出すことができます。 	<ul style="list-style-type: none"> 外部ルーチンは、外部ルーチンの SQL アクセス・レベルより低いか等しい SQL アクセス・レベルで、外部ルーチンおよび他の SQL ルーチンを呼び出すことができます。

一般に、ルーチンの機能特性および用途によって、使用するルーチンのタイプが決定します。しかし、パフォーマンスおよびサポートされるルーチンのインプリメンテーションも、使用するルーチンのタイプを決定する上で重要な役割を果たします。

使用するルーチン・インプリメンテーションの判別

組み込み、ソース派生、SQL、または外部の各ルーチン・インプリメンテーションを備えたルーチンの使用または作成の選択は、ルーチンが提供できる機能、ルーチンのパフォーマンス、およびデバッグを必要とする実行時間問題の可能性に影響を与える場合があります。

可能な場合はいつでも、必要とするサポートを提供する既存のシステム定義ルーチンがあれば、それを使用します。可能な場合はいつでも、既存の組み込みルーチンを使用します。必要とする機能が、既存の組み込み関数の機能と非常によく似ている場合、既存のものを拡張するソース派生関数の作成を考慮します。

ルーチンを作成する必要がある場合は、以下の手順に従います。ルーチンの設計を先に進める前に、使用するルーチン・インプリメンテーションを決定することは重要です。

ルーチンを作成する際に、ソース派生、SQL、外部ルーチンのインプリメンテーションのいずれを使用するかを決定するには、以下のようにします。

1. プロシージャ、関数、またはメソッドを作成するかどうかを決定する。これはいつでも、ルーチンを開発する場合の最初のステップになります。さらに、そのルーチン・タイプのサポート・インプリメンテーションも決定します。参照先:
 - 18 ページの『ルーチンの機能タイプの比較』
2. ルーチンにどの SQL ステートメントを組み込むかを決定する。ルーチン内で実行する SQL ステートメントのセットにより、ルーチン・インプリメンテーションの選択が制限される場合があります。参照先:
 - 46 ページの『ルーチンで実行できる SQL ステートメントの決定』

3. 現在または将来、ルーチン・ロジックがデータベース外にあるデータ、ファイル、またはアプリケーションにアクセスする必要があるかどうかを判別します。データ、ファイル、またはアプリケーションは、データベース・サーバーのファイル・システム内または使用できるネットワーク内にある場合があります。
 - ルーチン・ロジックがデータベース外のエンティティにアクセスする必要がある場合、外部ルーチン・インプリメンテーションを使用する必要があります。
4. プロシージャ・フロー・ロジックの量と比較して、ルーチン内に組み込む照会の数を決定する。
 - ルーチン・ロジックに、主要なプロシージャ・フロー・ロジックと、ごく少数の照会が含まれている場合は、外部ルーチンを作成する。
 - ルーチン・ロジックに、数多くの照会と、最小量のプロシージャ・フロー・ロジックが含まれている場合は、SQL ルーチンを作成する。

ルーチンの使用法

ルーチンを使用して、データベース設計者、データベース管理者、およびアプリケーション開発者が一様によく直面する多くの問題を解決することができます。これは、アプリケーションの構造、保守、パフォーマンスの向上に寄与します。ルーチンを使用する可能性のあるシナリオの例のいくつかを以下にリストします。

- ルーチンを使用したデータベースの管理
- ユーザー定義関数による SQL 関数サポートの拡張
- ルーチンおよび他の SQL フィーチャーを使用したデータ変更の監査

システム定義ルーチンを使用したデータベースの管理

アプリケーションを使用したデータベースの管理が可能です。管理機能を実行することを明示的な目的としたシステム定義ルーチンの導入により、この管理がいつも簡単になりました。DB2 のバージョン 8.1 では、管理用タスクの実行にすぐに使用できるシステム定義の一連のプロシージャと関数が、

SYSPROC、SYSFUN、SYSTOOLS スキーマで提供されています。この管理用タスクには、SQL インターフェースを介した DB2 コマンドの実行、構成パラメーターの変更、パッケージ管理、スナップショット関連タスクなどが含まれます。管理用タスクを実行するためにアプリケーションが必要な場合や、SQL インターフェースを介して管理用タスクの結果にアクセスし、結果を他の照会でフィルタリング、ソート、変更、または再利用するが、そのための独自のルーチンを作成することは望まない場合には、システム定義の管理ルーチンの使用を選ぶことができます。

DB2 for Linux®, UNIX®, and Windows® バージョン 9.1 で開始して、ADMIN_CMD という名前の新しいシステム定義の管理ルーチンがあります。これに他の多数のシステム定義ルーチンが加わって、包括的管理サポートが提供されます。

SQL インターフェースを介して DB2 コマンドを起動する ADMIN_CMD

バージョン 9.1 では、SQL インターフェースを介して DB2 コマンドを実行するための、ADMIN_CMD と呼ばれる新しいシステム定義の管理ルーチンがあります。基本的には、このルーチンを使用すれば、適切なフラグと値をストリング・パラメーターとして指定した DB2 コマンドを、引数として

渡すことができます。このルーチンは DB2 コマンドを含むストリングを実行し、より大きい照会または操作の一部として使用できる表形式またはスカラー形式でその結果を戻します。この機能により、データベース管理アプリケーションの作成がこれまでより簡単に行えるようになりました。

システム定義の管理ルーチン

システム定義のルーチンの例として、
SNAPSHOT_TABLE、HEALTH_DB_HI、SNAPSHOT_FILEW、
REBIND_ROUTINE_PACKAGE などがあります。これらおよび他の多数のシステム定義ルーチンは、CLP から、またはデータベース・アプリケーション内の指定したルーチンの呼び出しがサポートされている場所であればどこでも使用できます。

ADMIN_CMD ルーチンおよび他のシステム定義ルーチンは、CURRENT PATH 値に SYSPROC スキーマ名と SYSFUN スキーマ名が組み込まれている場合 (デフォルト) に使用できます。

システム定義ルーチンの使用法の例については、システム定義ルーチン固有の参照資料を参照してください。

ユーザー定義関数による SQL 関数サポートの拡張

必要なロジックがシステム定義関数にカプセル化されていない場合は、独自のユーザー定義関数を作成できます。ユーザー定義関数は、SQL 関数の基本セットの拡張に最適です。ユーザーまたはユーザーのグループが複雑な数式や特定のストリング処理をインプリメントしたり、値の意味変換を行ったりするための関数を必要とする場合は、それを行うための、既存の組み込み SQL 関数のように参照できる高性能 SQL 関数を簡単に作成できます。

例えば、ある通貨の値を別の通貨に変換する関数が必要になったユーザーを考えてみましょう。このような関数はシステム定義ルーチンのセットには用意されていません。しかし、この関数をユーザー定義の SQL スカラー関数として作成することができます。この関数が作成されると、SQL ステートメント内のスカラー関数がサポートされる場所であればどこでもそれを参照できます。

これとは別に、表の特定の列に変更が加えられるたびに E メールを送信する、もっと複雑な関数を必要としているユーザーがいます。このような関数はシステム定義ルーチンのセットには用意されていません。しかし、この関数を、C プログラミング言語によるインプリメンテーションで、ユーザー定義の外部プロシージャとして作成することができます。このプロシージャが作成されると、トリガー内などの、プロシージャがサポートされる場所であればどこでもそれを参照できます。

これらの例は、ユーザー定義ルーチンを作成することによって SQL 言語をどれだけ簡単に拡張できるかを示します。

SQL 表関数を使用した監査

データベース・ユーザーによる表データへのアクセスや表データの変更をモニターするデータベース管理者は、SQL データを変更する SQL 表関数を作成および使用して、表に対するトランザクションの監査を実行できます。

ビジネス・タスク (従業員の個人情報の更新など) を実行する SQL ステートメントをカプセル化した表関数には、その関数を呼び出したユーザーによる表のアクセスや変更の詳細を別の表に記録する SQL ステートメントを別に組み込むことができます。表関数の本体でアクセスまたは変更された行の結果セットを戻す SQL 表関数を作成することさえできます。表に対する変更の履歴として、戻された行の結果セットを別の表に挿入したり、保管したりできます。

SQL 表関数の作成および登録に必要な特権のリストについては、以下のステートメントを参照してください。

- CREATE FUNCTION (SQL スカラー、表、または行) ステートメント

SQL 表関数の定義者にも、SQL 表関数本体にカプセル化した SQL ステートメントの実行権限が必要です。カプセル化したそれぞれの SQL ステートメントごとに、必要な特権のリストを参照してください。表に関する INSERT、UPDATE、DELETE 特権をユーザーに付与するには、以下のステートメントを参照してください。

- GRANT (表、ビュー、またはニックネーム特権) ステートメント

SQL 表関数のアクセス対象の表は、SQL 表関数を呼び出す前に存在していなければなりません。

例 1: SQL 表関数を使用した表データのアクセスの監査

この関数は、入力引数 deptno によって指定された部門の全従業員の給料データにアクセスします。audit_table という名前の監査表、関数を呼び出すユーザー ID、読み取られた表の名前、どの情報にアクセスされたかの説明、および現在時刻も記録されます。この表関数には、SQL データを変更する INSERT ステートメントが含まれているので、キーワード MODIFIES SQL DATA を指定していることに注意してください。

```
CREATE FUNCTION sal_by_dept (deptno CHAR(3))
  RETURNS TABLE (lastname VARCHAR(10),
                 firstname VARCHAR(10),
                 salary INTEGER)
LANGUAGE SQL
MODIFIES SQL DATA
NO EXTERNAL ACTION
NOT DETERMINISTIC
BEGIN ATOMIC
  INSERT INTO audit_table(user, table, action, time)
  VALUES (USER,
          'EMPLOYEE',
          'Read employee salaries in department: ' || deptno,
          CURRENT_TIMESTAMP);
RETURN
  SELECT lastname, firstname, salary
  FROM employee as E
  WHERE E.dept = deptno;
END
```

例 2: SQL 表関数を使用した表データの更新の監査

この関数は updEmpNum によって指定された従業員の給料を amount で指定された金額で更新し、audit_table という名前の監査表、ルーチンを呼び出したユーザー、変更された表の名前、およびそのユーザーによって作成された変更のタイプも記録します。FROM 節でデータ変更ステートメント (ここでは UPDATE ステートメント) を参照する SELECT ステートメントを使用して、更新された行の値を戻します。この表関数には、INSERT ステートメントと、データ変更ステートメント UPDATE を参照する SELECT

ステートメントが含まれているので、キーワード `MODIFIES SQL DATA` を指定していることに注意してください。

```
CREATE FUNCTION update_salary(updEmpNum CHAR(4), amount INTEGER)
  RETURNS TABLE (emp_lastname VARCHAR(10),
                 emp_firstname VARCHAR(10),
                 newSalary INTEGER)
LANGUAGE SQL
MODIFIES SQL DATA
NO EXTERNAL ACTION
NOT DETERMINISTIC
BEGIN ATOMIC
  INSERT INTO audit_table(user, table, action, time)
  VALUES (USER,
          'EMPLOYEE',
          'Update emp salary. Values: '
          || updEmpNum || ' ' || char(amount),
          CURRENT_TIMESTAMP);
RETURN
  SELECT lastname, firstname, salary
  FROM FINAL TABLE(UPDATE employee
                   SET salary = salary + amount
                   WHERE employee.empnum = updEmpNum);
END
```

例 3: トランザクションを監査するために使用される SQL 表関数の呼び出し

以下は、ユーザーが従業員の給料を 500 円更新するルーチンを呼び出す方法を示しています。

```
SELECT emp_lastname, emp_firstname, newsalary
FROM TABLE(update_salary(CHAR('1136'), 500)) AS T
```

結果セットは、従業員のラストネーム、ファーストネーム、および新規給料とともに戻されます。関数の呼び出し側は、監査レコードが作成されていることを知りません。

```
EMP_LASTNAME EMP_FIRSTNAME NEWSALARY
-----
JONES        GWYNETH          90500
```

監査表には以下のような新規レコードが含まれます。

```
USER      TABLE      ACTION
-----
MBROOKS  EMPLOYEE    Update emp salary. Values: 1136 500
2003-07-24-21.01.38.459255
```

例 4: SQL 表関数の本体内で変更された行の検索

この関数は、従業員番号 `EMPNUM` によって指定されている従業員の給料を `amount` で指定されている金額で更新し、変更された行の元の値を呼び出し側に戻します。この例は、`FROM` 節内のデータ変更ステートメントを参照する `SELECT` ステートメントを利用します。このステートメントの `FROM` 節内にある `OLD TABLE` を指定すると、`UPDATE` ステートメントのターゲットである `employee` 表からの元の行データの戻りにフラグが立てられます。 `OLD TABLE` の代わりに `FINAL TABLE` を使用すると、`employee` 表の更新に続く行の値の戻りにフラグが立てられます。

```
CREATE FUNCTION update_salary (updEmpNum CHAR(4), amount DOUBLE)
  RETURNS TABLE (empnum CHAR(4),
                 emp_lastname VARCHAR(10),
                 emp_firstname VARCHAR(10),
                 dept CHAR(4),
                 newsalary integer)
LANGUAGE SQL
```

```

MODIFIES SQL DATA
NO EXTERNAL ACTION
DETERMINISTIC
BEGIN ATOMIC
RETURN
    SELECT empnum, lastname, firstname, dept, salary
    FROM OLD TABLE(UPDATE employee
                    SET salary = salary + amount
                    WHERE employee.empnum = updEmpNum);
END

```

ルーチン開発のためのツール

プロシージャおよび関数の開発にはさまざまな開発環境やツールが使用可能です。ツールの中には、DB2 Database for Linux, UNIX, and Windows に付属するものもあれば、一般の統合開発環境内に統合されているコンポーネントであるものもあります。プロシージャおよび関数の開発に使用できるグラフィカル・インターフェース、非グラフィカル・インターフェース、およびツールがあります。

DB2 データベース・サーバー内のルーチンの開発に使用できる DB2 付属のグラフィカル・ユーザー・インターフェース (GUI) ツールには次のものがあります。

- IBM® Data Studio

DB2 データベース・サーバー内のルーチンの開発に使用できる DB2 付属のコマンド行インターフェースには次のものがあります。

- DB2 コマンド行プロセッサ (DB2 CLP)

いくつかの IBM ソフトウェア製品も DB2 データベース・サーバー内のルーチンの開発のためのグラフィカル・ツールを提供します。以下はそのいくつかの例に過ぎません。

- IBM 分散統一デバッガー
- IBM Optim™ Development Studio
- IBM Rational® Application Developer
- IBM Rational Web Developer
- IBM WebSphere® Studio

いくつかのオープン・ソース・ソフトウェア製品も DB2 データベース・サーバー内のルーチンの開発のためのグラフィカル・ツールを提供します。以下はその一例に過ぎません。

- DB2 Web Tools for the Eclipse Framework

いくつかの DB2 のフィーチャーは、他のベンダー提供のソフトウェアでルーチンを開発できるようにするためのグラフィック・ツール・サポートを追加するために使用できます。例えば、次のものがあります。

- IBM DB2 Development Add-In for Microsoft Visual Studio .NET 1.2

IBM Data Studio ルーチン開発サポート

IBM Data Studio は、ストアード・プロシージャを作成、構築、デバッグ、テスト、およびデプロイするための使いやすい開発環境を提供します。IBM Data Studio は、ルーチンの作成プロセスを単純化するグラフィカル・ツールを提供します。こ

の単純化は、基本の CREATE ステートメントの生成、DB2 サーバーでのストアド・プロシージャの構築およびインストールに関する詳細にあまり労力をかけることなく、ストアド・プロシージャのロジックに注意を向けることにより実現されます。さらに、あるオペレーティング・システムで開発したストアド・プロシージャを、他のサーバー・オペレーティング・システムで構築できます。

IBM Data Studio は、迅速な開発をサポートするグラフィカル・アプリケーションです。これを使用して、以下のタスクを実行することができます。

- 新規のストアド・プロシージャを作成する。
- ローカルおよびリモート DB2 サーバーでストアド・プロシージャを作成する。
- 既存のストアド・プロシージャを変更して再作成する。
- インストールされたストアド・プロシージャのテストとデバッグを行う。

IBM Data Studio では、プロジェクトの作業を管理することができます。それぞれの IBM Data Studio プロジェクトは、DB2 for z/OS サーバーなどの特定のデータベースへの接続を保存します。さらに、各データベース上にストアド・プロシージャのサブセットを表示するためのフィルターを作成することができます。新規または既存の Data Studio プロジェクトを開く場合は、その名前、スキーマ、言語、またはコレクション ID (DB2 for z/OS サーバーの場合) に基づいてストアド・プロシージャを表示するために、ストアド・プロシージャにフィルターをかけることができます。

IBM Data Studioについて詳しくは、IDM インフォメーション・センター:
<http://publib.boulder.ibm.com/infocenter/idm/v2r2/index.jsp>を参照してください。

ルーチン内で実行できる SQL ステートメント

ルーチンで SQL ステートメントを正常に実行できるかどうかは、制約事項によって、特定の前提条件が満たされるかどうかの条件によって決まります。ただし、ルーチンで多数の SQL ステートメントを実行することは可能です。

ステートメントがルーチンを呼び出す場合、そのステートメントの有効な SQL データ・アクセス・レベルは次のうちの大きいほうになります。

- 以下の表にあるステートメントの SQL データ・アクセス・レベル。
- ルーチンの作成時に指定された、ルーチンの SQL データ・アクセス・レベル。

例えば、CALL ステートメントの SQL データ・アクセス・レベルは CONTAINS SQL です。しかし、READS SQL DATA として定義されたストアド・プロシージャが呼び出されると、CALL ステートメントの有効な SQL データ・アクセス・レベルは READS SQL DATA になります。

以下の表は、SQL PL 制御ステートメントなどの、サポートされているすべての SQL ステートメントを一覧で示し、各種のルーチン内で各 SQL ステートメントを実行できるかどうかを示しています。最初の列に一覧で示されているどの SQL ステートメントの場合も、そのステートメントをルーチンで実行できるかどうか、その後続く各列に X で示しています。最終列は、ステートメントを実行できる最

小の SQL アクセス・レベルを示しています。脚注に他に指示がなければ、どの SQL ステートメントも、静的および動的の両方の方式で実行できます。

表 5. ルーチン内で実行できる SQL ステートメント

SQL ステートメント	コンパウンド SQL (コンパイル) ステートメント (1) で実行可能	コンパウンド SQL (インライン) ステートメント (2) で実行可能	外部プロシージャ内の実行可能ファイル	外部関数内の実行可能ファイル	必要最低限の SQL データ・アクセス・レベル
ALLOCATE CURSOR	X		X	X	MODIFIES SQL DATA
ALTER {BUFFERPOOL, DATABASE PARTITION GROUP, FUNCTION, METHOD, NICKNAME, PROCEDURE, SEQUENCE, SERVER, TABLE, TABLESPACE, TYPE, USER MAPPING, VIEW}			X	X	MODIFIES SQL DATA
ASSOCIATE LOCATORS	X				
AUDIT			X	X	MODIFIES SQL DATA
BEGIN DECLARE SECTION			X	X	NO SQL(3)
CALL	X	X	X	X	READS SQL DATA
CASE	X	X			CONTAINS SQL
CLOSE	X		X	X	READS SQL DATA
COMMENT ON	X		X	X	MODIFIES SQL DATA
COMMIT	X(6)		X(6)		MODIFIES SQL DATA
コンパウンド SQL	X	X	X	X	CONTAINS SQL
CONNECT(2)					

表 5. ルーチン内で実行できる SQL ステートメント (続き)

SQL ステートメント	コンパウンド SQL (コンパイル) ステートメント (1) で実行可能	コンパウンド SQL (インライン) ステートメント (2) で実行可能	外部プロシージャ内の実行可能ファイル	外部関数内の実行可能ファイル	必要最低限の SQL データ・アクセス・レベル
CREATE {ALIAS, BUFFERPOOL, DATABASE PARTITION GROUP, DISTINCT TYPE, EVENT MONITOR, FUNCTION, FUNCTION MAPPING, GLOBAL TEMPORARY TABLE(11), INDEX(11), INDEX EXTENSION, METHOD, NICKNAME, PROCEDURE, SCHEMA, SEQUENCE, SERVER, TABLE(11), TABLESPACE, TRANSFORM, TRIGGER, TYPE, TYPE MAPPING, USER MAPPING, VIEW(11), WRAPPER }	X(8)		X		MODIFIES SQL DATA
DECLARE CURSOR	X	X	X		NO SQL(3)
DECLARE TEMPORARY TABLE	X		X	X	MODIFIES SQL DATA
DELETE	X	X	X	X	MODIFIES SQL DATA
DESCRIBE(9)			X	X	READS SQL DATA
DISCONNECT(4)					
DROP	X(8)		X	X	MODIFIES SQL DATA

表 5. ルーチン内で実行できる SQL ステートメント (続き)

SQL ステートメント	コンパウンド SQL (コンパイル) ステートメント (1) で実行可能	コンパウンド SQL (インライン) ステートメント (2) で実行可能	外部プロシージャ内の実行可能ファイル	外部関数内の実行可能ファイル	必要最低限の SQL データ・アクセス・レベル
END DECLARE SECTION			X	X	NO SQL(3)
EXECUTE	X		X	X	CONTAINS SQL(5)
EXECUTE IMMEDIATE	X		x	X	CONTAINS SQL(5)
EXPLAIN	X		X	X	MODIFIES SQL DATA
FETCH	X		X	X	READS SQL DATA
FREE LOCATOR			X	X	CONTAINS SQL
FLUSH EVENT MONITOR			X	X	MODIFIES SQL DATA
FLUSH PACKAGE CACHE			X	X	MODIFIES SQL DATA
FOR	X	X			READS SQL DATA
FREE LOCATOR	X		X	X	CONTAINS SQL
GET DIAGNOSTICS	X	X			READS SQL DATA
GOTO	X	X			CONTAINS SQL
GRANT	X		X	X	MODIFIES SQL DATA
IF	X	X			CONTAINS SQL
INCLUDE			X	X	NO SQL
INSERT	X	X	X	X	MODIFIES SQL DATA
ITERATE	X	X			CONTAINS SQL
LEAVE	X	X			CONTAINS SQL
LOCK TABLE	X		X	X	CONTAINS SQL
LOOP	X	X			CONTAINS SQL
MERGE	X	X	X	X	MODIFIES SQL DATA
OPEN	X		X	X	READS SQL DATA(7)
PREPARE	X		X	X	CONTAINS SQL
REFRESH TABLE			X	X	MODIFIES SQL DATA
RELEASE(4)					

表 5. ルーチン内で実行できる SQL ステートメント (続き)

SQL ステートメント	コンパウンド SQL (コンパイル) ステートメント (1) で実行可能	コンパウンド SQL (インライン) ステートメント (2) で実行可能	外部プロシージャ内の実行可能ファイル	外部関数内の実行可能ファイル	必要最低限の SQL データ・アクセス・レベル
RELEASE SAVEPOINT	X		X	X	MODIFIES SQL DATA
RENAME TABLE			X	X	MODIFIES SQL DATA
RENAME TABLESPACE			X	X	MODIFIES SQL DATA
REPEAT	X	X			CONTAINS SQL
RESIGNAL	X				MODIFIES SQL DATA
RETURN	X				CONTAINS SQL
REVOKE			X	X	MODIFIES SQL DATA
ROLLBACK(6)	X		X		
ROLLBACK TO SAVEPOINT	X		X	X	MODIFIES SQL DATA
SAVEPOINT	X				MODIFIES SQL DATA
SELECT ステートメント	X		X	X	READS SQL DATA
SELECT INTO	X		X(10)	X(10)	READS SQL DATA(7)
SET CONNECTION(4)					
SET INTEGRITY			X		MODIFIES SQL DATA
SET 特殊レジスタ	X	X	X	X	CONTAINS SQL
SET 変数	X	X			CONTAINS SQL
SIGNAL	X	X			MODIFIES SQL DATA
TRANSFER OWNERSHIP			X	X	MODIFIES SQL DATA
TRUNCATE			X	X	MODIFIES SQL DATA
UPDATE	X	X	X		MODIFIES SQL DATA
VALUES INTO	X		X	X	READS SQL DATA
WHENEVER	X		X		NO SQL(3)
WHILE	X	X			

注:

1. コンパウンド SQL (コンパイル) ステートメントは、SQL プロシージャ、SQL 関数、トリガーの本体として、またはスタンドアロンのステートメントとして使用することができます。
2. コンパウンド SQL (インライン) ステートメントは、SQL 関数、SQL メソッド、およびトリガーの本体として、またはスタンドアロンのステートメントとして使用することができます。
3. NO SQL オプションでは SQL ステートメントを指定できないことを暗黙に示していますが、実行不能ステートメントは制限されません。
4. 接続管理ステートメントは、どのルーチン実行コンテキストでも許可されません。
5. この状況は、実行されるステートメントに応じて決まります。EXECUTE ステートメントに指定されるステートメントは、有効な特定の SQL アクセス・レベルのコンテキストで許可されるステートメントでなければなりません。例えば、SQL アクセス・レベル READS SQL DATA が有効な場合、ステートメントは INSERT、UPDATE、または DELETE にすることができません。
6. TO SAVEPOINT 節を指定しない COMMIT ステートメントおよび ROLLBACK ステートメントをストアード・プロシージャで使用できます。ただし、ストアード・プロシージャがアプリケーションから直接呼び出される場合、またはアプリケーションからのネストされたストアード・プロシージャ呼び出しを介して間接的に呼び出される場合に限りです。トリガー、関数、メソッド、またはアトミック・コンパウンド・ステートメントがストアード・プロシージャへの呼び出しチェーンに含まれる場合、作業単位の COMMIT または ROLLBACK は許可されません。
7. SQL アクセス・レベル READS SQL DATA が有効な場合、SQL データ変更ステートメントを SELECT INTO ステートメントに組み込んだり、OPEN ステートメントによって参照されるカーソルに組み込むことはできません。
8. SQL プロシージャが発行できるのは、索引、表、およびビュー用の CREATE および DROP ステートメントだけです。
9. DESCRIBE SQL ステートメントの構文は、CLP DESCRIBE コマンドとは違っています。
10. これは組み込み SQL ルーチンでのみサポートされています。
11. SQL プロシージャ内で参照される場合、ステートメントの実行は静的にのみ行えます。

エラー

40 ページの表 5 は、最初の列で指定された SQL ステートメントが、指定された SQL データ・アクセス・レベルのルーチン内で実行可能かどうかを示します。ステートメントがこのデータ・アクセス・レベルを超えている場合にルーチンを実行すると、エラーが戻されます。

- NO SQL データ・アクセス・レベルで定義されたルーチン内で実行可能 SQL ステートメントが検出された場合、SQLSTATE 38001 が戻されます。
- 他の実行コンテキストの場合、特定のコンテキストでサポートされていない SQL ステートメントがあると SQLSTATE 38003 が戻されます。

- CONTAINS SQL コンテキストで許可されていない他の SQL ステートメントについては、SQLSTATE 38004 が戻されます。
- READS SQL DATA コンテキストでは、SQLSTATE 38002 が戻されます。
- SQL ルーチンの作成時に、SQL データ・アクセス・レベルに一致しないステートメントがあると、SQLSTATE 42985 が戻されます。

ルーチンの SQL アクセス・レベル

ルーチンが SQL ステートメントを実行できる度合いは、ルーチンの SQL アクセス・レベルによって決定されます。ルーチンの SQL アクセス・レベルは、特定のタイプのルーチンで何が許可されるか、またルーチンを定義する CREATE ステートメントの中でどのような制限が明示的に指定されるか、という 2 つの要素によって決定されます。

SQL アクセス・レベルは次のとおりです。

- NO SQL
- CONTAINS SQL
- READS SQL
- MODIFIES SQL

この SQL アクセス・レベルの節を使用して、データベース・マネージャーが安全にステートメントを実行し、最良のパフォーマンスを得るためのステートメント情報をデータベース・マネージャーに提供します。

各種ルーチンのデフォルト SQL アクセス・レベルと最大 SQL アクセス・レベルが以下に示されています。

表 6. ルーチンのデフォルト SQL アクセス・レベルと最大 SQL アクセス・レベル

ルーチン・タイプ	デフォルト SQL アクセス・レベル	許可される最大 SQL アクセス・レベル
SQL プロシージャ	MODIFIES SQL DATA	MODIFIES SQL DATA
SQL 関数 (スカラー関数)	READS SQL DATA	READS SQL DATA
SQL 関数 (表関数)	READS SQL DATA	MODIFIES SQL DATA
外部プロシージャ	MODIFIES SQL DATA	MODIFIES SQL DATA
外部関数 (スカラー関数)	READS SQL DATA	READS SQL DATA
外部関数 (表関数)	READS SQL DATA	READS SQL DATA

ルーチンが最高のパフォーマンスを発揮するのは、ルーチン CREATE ステートメントに最も限定的な (有効な) SQL アクセス節を指定した場合です。

ルーチンの CREATE ステートメントに関して、次の考慮事項があります。

- READS SQL DATA を明示的に指定する場合、ルーチン内のどの SQL ステートメントもデータを変更することができません。
- CONTAINS SQL DATA を明示的に指定する場合、ルーチン内のどの SQL ステートメントもデータを変更したり、データを読み取ったりすることができません。

- NO SQL を明示的に指定する場合、ルーチン内に実行可能 SQL ステートメントを含めてはなりません。

ルーチンで実行できる SQL ステートメントの決定

ルーチン内で多くの SQL ステートメントを実行できますが、すべての SQL ステートメントを実行できるわけではありません。ルーチン内で特定の SQL ステートメントを実行できるかどうかは、ルーチンのタイプ、ルーチンのインプリメンテーション、ルーチンに指定された最大 SQL アクセス・レベル、ルーチンの定義者および呼び出し側の特権によって決まります。

ルーチンをインプリメントする前に、ルーチン内でどのような SQL ステートメントを実行できるかを決定しておけば、最初からルーチンのタイプとインプリメンテーションを確実に正しく選択できます。

ルーチン内で正常に SQL ステートメントを実行するには、以下の前提条件を満たす必要があります。

- ルーチンの SQL アクセス・レベルが、その特定の SQL ステートメントの実行を許可していなければならない。
 - ルーチンの SQL アクセス・レベルは、そのルーチン用の CREATE ステートメントで指定されます。
 - ルーチンのタイプによっては、サポートされない SQL アクセス・レベルがあります。下記の制約事項を参照してください。
- ルーチン定義者は、その SQL ステートメントを実行するのに必要な特権を持っていないといけない。
 - サポートされている各 SQL ステートメントを実行するのに必要な特権は、「SQL リファレンス」に記載されています。
- これら以外に、そのステートメントの実行を制限している別の制約事項がない。
 - 特定の SQL ステートメント固有の制約事項のリストについては、「SQL リファレンス」を参照してください。

ルーチン内で実行できる一連の SQL ステートメントには、以下の制約事項による制限があります。これらの制約事項は特に、特定のルーチン・タイプにどの SQL アクセス・レベルを指定できるかを制限するものです。

- 外部関数は MODIFIES SQL DATA アクセス・レベルで指定できません。
- トリガーから呼び出される外部プロシージャは MODIFIES SQL DATA アクセス・レベルで指定できません。

特定のルーチンでどの SQL ステートメントを呼び出せるかを判別するには、以下のようになります。

1. ルーチンの SQL アクセス・レベルを判別する。既存のルーチンの場合は、そのルーチンの作成時に使用された CREATE ステートメントを調べてください。NO SQL、CONTAINS SQL、READS SQL DATA、MODIFIES SQL DATA のいずれかの SQL アクセス・レベル節が、DDL で明示的に定義されている可能性があります。こうした節が明示的に指定されていない場合は、そのルーチンのデフォルト値が暗黙的に使用されます。
 - SQL プロシージャの場合、デフォルトは MODIFIES SQL DATA です。

- SQL 関数の場合、デフォルトは MODIFIES SQL DATA です。
 - 外部プロシージャの場合、デフォルトは MODIFIES SQL DATA です。
 - 外部関数の場合、デフォルトは READS SQL DATA です。
2. 『ルーチン内で実行できる SQL ステートメント』のトピック内の表を参照する。該当する SQL ステートメントを名前で調べて見つけてください。
 3. 特定のタイプのルーチンとインプリメンテーションでその SQL ステートメントがサポートされているかどうかを確認する。
 4. ステートメントを実行するのに必要な SQL アクセス・レベルが、そのルーチンの SQL アクセス・レベルと一致するかどうか確認する。
 5. 使用上の注意や脚注をよく読み、その SQL ステートメントの実行に関して他に制約事項がないことを確認する。

その SQL ステートメントがルーチン内で実行可能であることが示され、ルーチンの SQL アクセス・レベルがルーチン内でそのステートメントを実行するための前提条件を満たし、その他のすべての前提条件が満たされていれば、その SQL ステートメントはそのルーチンから正常に実行できます。

ルーチンの移植性

ルーチンの移植性は、ルーチンをデプロイするときの容易さのことを指します。移植性は、オペレーティング・システムの互換性、ランタイム環境の互換性、ソフトウェアの互換性、呼び出しインターフェースの互換性などの要因と、ルーチン内で実行される SQL ステートメントのサポートの互換性などの他のルーチンのインプリメンテーションの要因で構成されます。

ルーチンをデプロイする環境とルーチンが開発された環境が異なる場合には、ルーチンの移植性は必要不可欠です。一般に、オペレーティング・システム間、さらには各種 DB2 データベース製品およびエディション間の DB2 ルーチンの移植性は高くなっています。後々の作業のやり直しの可能性を最小化するために、ルーチンを開発する前に移植性に関する潜在的な問題について考慮することが得策です。

以下のトピックには、ルーチンの移植性を制限できる要因に関連した情報が含まれています。

- サポートされる DB2 Database for Linux, UNIX, and Windows エディション
- サポートされる開発およびコンパイラ・ソフトウェア
- ルーチン内で実行できる SQL ステートメント
- ルーチンに関する制約事項
- ルーチンのデプロイ

ルーチンのインターオペラビリティ

さまざまなタイプを持ち、さまざまなプログラミング・インプリメンテーションを使用するルーチンのインターオペラビリティにより、ルーチンはデータベース・システムの存続期間を通して再利用性の高いモジュールになることができます。コード・モジュールは、多数のプログラマーによってインプリメントされることが多く、しかもそれぞれのプログラマーが別々のプログラム言語を専門にしているケースも少なくありません。また、開発時間を短縮し、開発コストを節約するには、で

きる限りコードを再利用することが望ましいと言えます。こうした状況を踏まえ、DB2 のルーチン・インフラストラクチャーは、高度なルーチン・インターオペラビリティをサポートするように設計されています。

ルーチンのインターオペラビリティの特徴は、さまざまなタイプおよびインプリメンテーションのルーチンを、その他のルーチンから追加要件なしでシームレスに参照して呼び出すという機能です。DB2 のルーチンは、以下の方法でインターオペラビリティを提供します。

- 1 つのプログラム言語のクライアント・アプリケーションから、別々のプログラム言語でインプリメントしたルーチンを呼び出すことができます。
 - 例えば、C クライアント・アプリケーションから、.NET 共通言語ランタイム・ルーチンを呼び出すことができます。
- ルーチンのタイプやルーチンのインプリメンテーション言語に関係なく、1 つのルーチンから別のルーチンを呼び出すことができます。
 - 例えば、Java プロシージャ (1 つのタイプのルーチン) は、SQL スカラー関数 (インプリメンテーション言語の異なる別のタイプのルーチン) を呼び出すことができます。
- 1 つのオペレーティング・システム上のデータベース・サーバーに作成したルーチンを、別のオペレーティング・システム上で実行する DB2 クライアントから呼び出せます。

各種の機能要件に合わせたさまざまなタイプのルーチンがあり、それぞれのインプリメンテーションの方法もさまざまです。ルーチンのタイプとインプリメンテーションの選択によって、上記の利点をどの程度具体化できるかが決まる場合もあります。ルーチンは基本的に、ロジックをカプセル化するための強力な手法です。この手法を活用すれば、SQL を拡張し、アプリケーションの構造と保守作業を改善し、場合によってはアプリケーションのパフォーマンスを向上させることができます。

ルーチンのパフォーマンス

ルーチンのパフォーマンスは様々な要因による影響を受けます。これには、ルーチンのタイプおよびインプリメンテーション、ルーチン内の SQL ステートメントの数、ルーチン内の SQL の複雑さの度合い、ルーチンに対するパラメーターの数、ルーチンのインプリメンテーションにおけるロジックの有効性、ルーチン内のエラー処理などが含まれます。ユーザーはしばしば、アプリケーションのパフォーマンスを向上させるためにルーチンをインプリメントすることを選ぶため、ルーチンのパフォーマンスを最大限に活用することは重要です。

以下の表は、ルーチンのパフォーマンスに影響を与える一般的な要因のいくつかの概略を示し、それぞれの要因を変更することによってルーチンのパフォーマンスを向上させる方法についての推奨事項を示しています。特定のルーチン・タイプに影響を与えるパフォーマンス要因の詳細については、その特定のルーチン・タイプのパフォーマンスおよびチューニングのトピックを参照してください。

表7. パフォーマンスの考慮事項およびルーチンのパフォーマンスの推奨事項

パフォーマンスの考慮事項	パフォーマンスの推奨事項
ルーチン・タイプ: プロシージャー、関数、メソッド	<ul style="list-style-type: none"> • プロシージャー、関数、およびメソッドはさまざまな目的で使用され、さまざまな場所で参照されます。それらには機能上の違いがあるため、パフォーマンスを直接比較することは困難です。 • 一般に、プロシージャーは関数として再作成できる場合があります (特に、プロシージャーがスカラー値を戻す場合と照会データのみを戻す場合)、それによってパフォーマンスがわずかに向上することがあります。しかし、それらの利点は一般に SQL ロジックをインプリメントするために必要な SQL を単純化することによって得られます。 • 複雑な初期化を伴うユーザー定義関数では、スクラッチパッドを利用して、最初の呼び出し時に必要とされる値を保管できます。そうすれば、それらの値は以後の呼び出しで使用することができます。
ルーチンのインプリメンテーション: システム定義またはユーザー定義	<ul style="list-style-type: none"> • 同等のロジックの場合、組み込みルーチンがパフォーマンスに最も優れており、その次に優れているのはシステム定義ルーチンです。それは、それらのルーチンはユーザー定義ルーチンよりもデータベース・エンジンと密接な関係を保つからです。 • ユーザー定義ルーチンは、適切にコード化されており、ベスト・プラクティスに従っているのであれば、良いパフォーマンスで実行できます。
ルーチンのインプリメンテーション: SQL または外部ルーチンのインプリメンテーション	<ul style="list-style-type: none"> • SQL ルーチンは外部ルーチンよりも効率的です。なぜなら、SQL ルーチンは DB2 データベース・サーバーによって直接実行されるからです。 • SQL プロシージャーは一般に、論理的に同等の外部プロシージャーよりもパフォーマンスに優れています。 • 単純なロジックの場合、SQL 関数のパフォーマンスは、同等の外部関数のパフォーマンスと同程度になります。 • 数学アルゴリズムおよびストリング処理関数などの SQL をほとんど必要としない複合ロジックの場合、C などの低レベルのプログラミング言語で記述した外部ルーチンを使用する方が適切です。なぜなら SQL サポートへの依存が少ないからです。 • サポートされる外部ルーチン・プログラミング言語オプションのフィーチャー (パフォーマンスを含む) の比較については、ルーチン・インプリメンテーションの比較を参照してください。

表 7. パフォーマンスの考慮事項およびルーチンのパフォーマンスの推奨事項 (続き)

パフォーマンスの考慮事項	パフォーマンスの推奨事項
外部ルーチンのインプリメンテーションのプログラミング言語	<ul style="list-style-type: none"> • 外部ルーチンのインプリメンテーションを選択する際に考慮すべきパフォーマンスのフィーチャーの比較については外部ルーチン API とプログラミング言語の比較を参照してください。 • Java (JDBC および SQLJ API) <ul style="list-style-type: none"> – メモリー要件が非常に大きい Java ルーチンを作成する場合は、FENCED NOT THREADSAFE 節を指定するのが最適です。消費するメモリー領域が平均的な Java ルーチンでは、FENCED THREADSAFE 節を指定できます。 – FENCED THREADSAFE Java ルーチンの呼び出しの場合、DB2 は、ルーチンを十分に実行できる大きさの Java ヒープを持つ、スレッド化された Java fenced モードのプロセスを選択しようとします。独自のプロセスで大量のヒープを消費するルーチンを分離できないと、マルチスレッド化された Java db2fmp プロセスで Java ヒープ不足エラーが発生する可能性があります。対照的に、FENCED THREADSAFE ルーチンは、少数の JVM を共有できるため、良いパフォーマンスが得られます。 • C および C++ <ul style="list-style-type: none"> – 一般に、C および C++ ルーチンのパフォーマンスは、その他の外部ルーチンのインプリメンテーションよりも優れており、SQL ルーチンと同程度です。 – 最適な C および C++ ルーチンを実行するには、それらが 32 ビットの DB2 インスタンスにデプロイされる場合には 32 ビット・フォーマットでコンパイルし、64 ビットの DB2 インスタンスにデプロイされる場合には 64 ビット・フォーマットでコンパイルする必要があります。 • COBOL <ul style="list-style-type: none"> – 一般に、COBOL のパフォーマンスも十分ですが、ルーチンのインプリメンテーションとして COBOL は推奨されていません。
ルーチン内の SQL ステートメントの数	<ul style="list-style-type: none"> • ルーチンには複数の SQL ステートメントが含まれている必要があります。そうしないと、ルーチン呼び出しのオーバーヘッドのパフォーマンス・コスト効率が低くなります。 • 複数のデータベース照会を行い、中間結果を処理し、処理したデータのサブセットを最終的に戻すようなロジックは、ルーチンのカプセル化に最適のロジックです。このタイプのロジックの例としては、複雑なデータ・マイニング、および関連データの検索を必要とする大規模な更新があります。負荷の高い SQL 処理はデータベース・サーバー上で行われ、呼び出し側には少量のデータ結果セットしか戻されません。

表7. パフォーマンスの考慮事項およびルーチンのパフォーマンスの推奨事項 (続き)

パフォーマンスの考慮事項	パフォーマンスの推奨事項
ルーチン内の SQL ステートメントの複雑さ	<ul style="list-style-type: none"> ルーチンに非常に複雑な照会を組み込んで、データベース・サーバーのメモリーおよびパフォーマンスの機能を十分に生かして活用できるようにするのは、良い方法です。 SQL ステートメントが複雑すぎることを心配しないでください。
ルーチン内の静的または動的 SQL 実行	<ul style="list-style-type: none"> 一般に、静的 SQL のパフォーマンスは動的 SQL よりも優れています。ルーチンでは、静的 SQL または動的 SQL を使用する場合に、それ以上の違いはありません。
ルーチンに対するパラメーターの数	<ul style="list-style-type: none"> ルーチンに対するパラメーターの数を最小限にすると、ルーチンとルーチン呼び出し側との間で受け渡されるバッファの数が最小限になるため、ルーチンのパフォーマンスを向上できます。
ルーチン・パラメーターのデータ・タイプ	<ul style="list-style-type: none"> ルーチン定義で CHAR パラメーターではなく VARCHAR パラメーターを使用することにより、ルーチンのパフォーマンスを向上させることができます。CHAR データ・タイプではなく VARCHAR データ・タイプを使用すると、パラメーターの引き渡しの前に DB2 によってパラメーターにスペースが埋め込まれなくなります。これで、ネットワークを経由したパラメーターの転送に要する時間が短縮されます。 <p>例えば、クライアント・アプリケーションが CHAR(200) パラメーターを予期するルーチンにストリング "A SHORT STRING" を渡す場合、DB2 はパラメーターに 186 個のスペースを埋め込み、ストリングを NULL で終了してから、200 文字のストリングと NULL 終止符全体をネットワーク経由でルーチンに送信する必要があります。</p> <p>それと比べて、VARCHAR(200) パラメーターを予期するルーチンに同じストリング "A SHORT STRING" を渡すと、DB2 は単に 14 文字ストリングと NULL 終止符をネットワーク経由で渡します。</p>
ルーチンに対するパラメーターの初期化	<ul style="list-style-type: none"> ルーチンに対する入力パラメーターを常に初期化することは、特に入力ルーチン・パラメーター値が NULL の場合には適切です。NULL 値のルーチン・パラメーターの場合、フルサイズのバッファではなく、小さいバッファまたは空のバッファをルーチンに渡すことができます。それにより、パフォーマンスを向上させることができます。

表7. パフォーマンスの考慮事項およびルーチンのパフォーマンスの推奨事項 (続き)

パフォーマンスの考慮事項	パフォーマンスの推奨事項
ルーチン内のローカル変数の数	<ul style="list-style-type: none"> ルーチン内で宣言されるローカル変数の数を最小限にすることにより、ルーチン内で実行される SQL ステートメントの数を最小限にして、パフォーマンスを向上させることができます。 一般に、使用する変数を可能な限り少なくすることを目標としてください。変数を再利用することで混乱が生じない場合は、変数を再利用してください。
ルーチン内のローカル変数の初期化	<ul style="list-style-type: none"> 可能であれば、単一の SQL ステートメント内で複数のローカル変数を初期化するのは、良い方法です。そうすれば、ルーチンの合計の SQL 実行時間を節約できます。
プロシージャが戻す結果セットの数	<ul style="list-style-type: none"> ルーチンによって戻される結果セットの数を減らせると、ルーチンのパフォーマンスを向上させることができます。
ルーチンによって戻される結果セットのサイズ	<ul style="list-style-type: none"> ルーチンによって戻される結果セットごとに、結果を定義する照会によって、戻された列および戻された行数をできるだけフィルタリングするようにしてください。不要なデータの列または行を戻すことは効率的ではなく、ルーチンのパフォーマンスが最適ではなくなる可能性があります。
ルーチン内のロジックの有効性	<ul style="list-style-type: none"> アプリケーションと同様に、ルーチンのパフォーマンスも、インプリメントが不十分なアルゴリズムによって制限されることがあります。ルーチンをプログラミングする際にはできる限り効率的であるようにし、一般に推奨されているコーディングのベスト・プラクティスをできる限り適用するようにしてください。 SQL を分析し、可能な限り照会を単純なフォームにしてください。これは多くの場合、CASE ステートメントの代わりに CASE 式を使用したり、複数の SQL ステートメントを CASE 式をスイッチとして使用する単一のステートメントに縮小したりすることによって行えます。

表7. パフォーマンスの考慮事項およびルーチンのパフォーマンスの推奨事項 (続き)

パフォーマンスの考慮事項	パフォーマンスの推奨事項
<p>ルーチンのランタイム・モード (FENCED または NOT FENCED 節の指定)</p>	<p>NOT FENCED 節の使用法:</p> <ul style="list-style-type: none"> • 一般に、ルーチンを NOT FENCED 節で作成すること (DB2 データベース・マネージャーと同じプロセスで実行することを指定) は、FENCED 節で作成すること (エンジンのアドレス・スペースの外部の特殊な DB2 プロセスで実行することを指定) より望ましいと言えます。 • ルーチンを not fenced として実行すると、ルーチンのパフォーマンスの向上を期待できませんが、unfenced ルーチンのユーザー・コードが意図せずにまたは故意にデータベースを破損したり、データベース制御構造に損傷を与えることがあります。NOT FENCED 節を使用するのは、パフォーマンスの利点を最大限にする必要があるとき、およびルーチンが安全であると判断される場合に限らなければなりません。(C/C++ ルーチンを NOT FENCED として登録するリスクの評価およびその軽減の詳細については、『ルーチンのセキュリティ』を参照してください。) ルーチンがデータベース・マネージャーのプロセスで実行できるほど安全でない場合は、ルーチンの作成時に FENCED 節を使用してください。安全でない可能性があるコードの作成および実行を制限するために、DB2 では、ユーザーが NOT FENCED ルーチンを作成するには、特殊権限 CREATE_NOT_FENCED_ROUTINE を持っていなければなりません。 • NOT FENCED ルーチンの実行中に異常終了が発生する場合、ルーチンが NO SQL として登録されていると、データベース・マネージャーは適切なりカバリーを試行します。しかし、NO SQL として定義されていないルーチンの場合、データベース・マネージャーは失敗します。 • ルーチンが GRAPHIC または DBCLOB データを使用する場合は、NOT FENCED ルーチンを WCHARTYPE NOCONVERT オプションでプリコンパイルする必要があります。

表7. パフォーマンスの考慮事項およびルーチンのパフォーマンスの推奨事項 (続き)

パフォーマンスの考慮事項	パフォーマンスの推奨事項
<p>ルーチンのランタイム・モード (FENCED または NOT FENCED 節の指定)</p>	<p>FENCED THREADSAFE 節の使用法</p> <ul style="list-style-type: none"> • FENCED THREADSAFE 節で作成されたルーチンは、その他のルーチンと同じプロセスで実行します。具体的には、Java 以外のルーチンはあるプロセスを共用し、Java(TM) ルーチンは他の言語で作成されたルーチンとは分離した、別のルーチンを共用します。この分離により、Java ルーチンは、他の言語で作成された、エラーを起こしやすいルーチンから保護されます。また、Java ルーチンのプロセスには JVM が含まれています。これは、メモリー・コストが高くなり、他のルーチン・タイプでは使用されません。 FENCED THREADSAFE ルーチンの複数の呼び出しではリソースを共用するため、それぞれが独自の専用プロセスで実行する FENCED NOT THREADSAFE ルーチンよりもシステムのオーバーヘッドが減ります。 • ご使用のルーチンが他のルーチンと同じプロセスで実行しても安全であると感じる場合、それを登録する際に THREADSAFE 節を使用してください。 NOT FENCED ルーチンと同様に、C/C++ ルーチンを FENCED THREADSAFE として登録するリスクの評価およびその軽減の詳細については、『ルーチンのセキュリティに関する考慮事項』のトピックを参照してください。 • FENCED THREADSAFE ルーチンが異常終了する場合、このルーチンを実行しているスレッドだけが終了されます。プロセス内のその他のルーチンは実行を続けます。しかし、このスレッドが異常終了する原因になった障害は、プロセス内の他のルーチンのスレッドに悪影響を及ぼし、トラップ、ハング、およびデータの破損の原因となることがあります。あるスレッドが異常終了した後は、そのプロセスは新規のルーチンの呼び出しに使用されません。すべてのアクティブ・ユーザーがこのプロセスでジョブを完了すると、それは終了されます。 • Java ルーチンを登録する際に、特に指定されない限り、THREADSAFE であると見なされます。その他の LANGUAGE タイプはすべて、デフォルトで NOT THREADSAFE です。 LANGUAGE OLE および OLE DB を使用するルーチンは THREADSAFE として指定できません。 • NOT FENCED ルーチンは THREADSAFE でなければなりません。ルーチンを NOT FENCED NOT THREADSAFE として登録することはできません (SQLCODE -104)。 • UNIX(R) のユーザーは、db2fmp (Java) または db2fmp (C) を探すことにより、Java および C の THREADSAFE プロセスを参照できます。

表7. パフォーマンスの考慮事項およびルーチンのパフォーマンスの推奨事項 (続き)

パフォーマンスの考慮事項	パフォーマンスの推奨事項
ルーチンのランタイム・モード (FENCED または NOT FENCED 節の指定)	<p>FENCED NOT THREADSAFE モード</p> <ul style="list-style-type: none"> • FENCED NOT THREADSAFE ルーチンはそれぞれ、独自の専用プロセスで実行します。多数のルーチンを実行している場合、このことはデータベース・システムのパフォーマンスに悪影響を及ぼす可能性があります。ルーチンが他のルーチンと同じプロセスで実行できるほど安全でない場合は、ルーチンを登録する際に NOT THREADSAFE 節を使用してください。 • UNIX では、NOT THREADSAFE プロセスは db2fmp (pid) (pid は fenced モード・プロセスを使用するエージェントのプロセス ID) またはプール NOT THREADSAFE db2fmp の場合は db2fmp (idle) として表示されます。
ルーチン内の SQL アクセスのレベル: NO SQL、CONTAINS SQL、READS SQL DATA、MODIFIES SQL DATA	<ul style="list-style-type: none"> • 低レベルの SQL アクセス節で作成されたルーチンは、高レベルの SQL アクセス節で作成されたルーチンよりもパフォーマンスに優れています。そのため、最も限定的なレベルの SQL アクセス節でルーチンを宣言してください。例えば、ルーチンが SQL データの読み取りだけを行う場合、ルーチンを MODIFIES SQL DATA 節で作成するのではなく、より限定的な READS SQL DATA 節で作成します。
ルーチンの決定論 (DETERMINISTIC または NOT DETERMINISTIC 節の指定)	<ul style="list-style-type: none"> • DETERMINISTIC または NOT DETERMINISTIC 節でルーチンを宣言しても、ルーチンのパフォーマンスに影響を与えません。
ルーチンによってとられる外部アクションの数および複雑さ (EXTERNAL ACTION 節の指定)	<ul style="list-style-type: none"> • 外部ルーチンによって実行される外部アクションの数および外部アクションの複雑さによっては、ルーチンのパフォーマンスの妨げとなることがあります。この原因となる要因としては、ネットワーク・トラフィック、書き込みまたは読み取り用のファイルへのアクセス、外部アクションの実行に要する時間、および外部アクションのコードまたは動作のハングに関連したリスクがあります。
入力パラメーターが NULL のときのルーチン呼び出し (CALLED ON NULL INPUT 節の指定)	<ul style="list-style-type: none"> • NULL の入力パラメーター値を受け取った場合はロジックが実行されず、ルーチンが即時に戻される結果になるのであれば、NULL の入力パラメーター値が検出されたときにルーチンが完全には呼び出されないように、ルーチンを変更することができます。ルーチン入力パラメーターを受け取った場合に呼び出しを早期に終了するルーチンを作成するには、ルーチンを作成して、CALLED ON NULL INPUT 節を指定します。

表7. パフォーマンスの考慮事項およびルーチンのパフォーマンスの推奨事項 (続き)

パフォーマンスの考慮事項	パフォーマンスの推奨事項
タイプ XML のプロシージャ・パラメーター	<ul style="list-style-type: none"> データ・タイプ XML のパラメーターを渡すことは、C または JAVA プログラミング言語でインプリメントされた外部プロシージャで行う場合、SQL プロシージャで行う場合と比べてかなり効率が低くなります。データ・タイプ XML の 1 つ以上のパラメーターを渡すときには、外部プロシージャではなく SQL プロシージャを使用することを考慮してください。 XML データは、ストアード・プロシージャに IN、OUT、または INOUT パラメーターとして渡されるときにマテリアライズされます。Java ストアード・プロシージャを使用している場合、XML 引数の数量とサイズ、および並行に実行されている外部ストアード・プロシージャの数に基づいて、ヒープ・サイズ (JAVA_HEAP_SZ 構成パラメーター) を増やさなければならない場合があります。

いったんルーチンを作成してデプロイすると、環境およびルーチンに固有のどのような要因がルーチンのパフォーマンスに影響を与えるのかを判別するのが難しくなる可能性があります。そのため、パフォーマンスを念頭においてルーチンを設計することが重要です。

ルーチンのセキュリティ

ルーチンのセキュリティは、ルーチンの継続的な機能を実際にし、改ざんのリスクを最小限に抑え、データベース・システム環境を保護するために最も重要なものです。ルーチンのセキュリティ考慮事項にはいくつかのカテゴリーがあり、それぞれのリスクのレベルは異なります。ユーザーは、ルーチンの開発または保守を行う場合、望ましくない結果を可能な限り軽減するために、これらのリスクについて知っている必要があります。

ルーチンの作成者に関するセキュリティ管理

ルーチンのセキュリティは、データベースでルーチンを作成するのに必要な CREATE ステートメントの実行に必要な特権がユーザーに付与されるときに始まります。これらの特権を付与することによるリスクを理解することは大切です。

- ルーチンの CREATE ステートメントを実行する特権を持つユーザーは、複数のルーチンを作成できます。
- ルーチンの CREATE ステートメントを実行する特権を持つユーザーは、ユーザーが持っている他の特権に従って、データベースのレイアウトまたはデータベースのデータを変更するルーチンを作成できます。
- ルーチンを正常に作成したユーザーには、ルーチンを呼び出すのに必要な EXECUTE 特権が自動的に付与されます。
- ルーチンを正常に作成したユーザーには、ルーチンを変更するのに必要な ALTER ROUTINE 特権が自動的に付与されます。

データベースおよびデータを変更するユーザーのリスクを最小限に抑えるには、以下のようにします。

- ルーチンを作成する特権を持つユーザーの数を最小限に抑えます。
- 過去の従業員のユーザー ID が除去されていることを確認するか、またはそのユーザー ID を再使用する場合、プロシージャ関連の特権を必ず審査してください。

1 つまたは多くの、あるいはすべてのデータベース・ユーザーに対して特権の付与および取り消しを行う方法について詳しくは、データベース・オブジェクトおよびデータへのアクセス制御に関するトピックを参照してください。

ルーチンを誰が呼び出すかに関するセキュリティー管理

ユーザーが権限を持っていないと、処理が実行できないため、ユーザーが特権を必要とするときを判断するのは容易です。ユーザーがこれらの特権を必要としなくなる時期を決定することは困難です。これは、ルーチンを呼び出す特権を持つユーザーについて特に当てはまります。それらのユーザーが特権を保持することを許可すると、リスクが生じる可能性があるためです。

- ルーチンを呼び出すための EXECUTE 特権が付与されたユーザーは、この特権が除去されるまでルーチンを引き続き呼び出すことができます。ルーチンに重要なロジックが含まれているか、ルーチンが重要なデータ上で機能する場合、これはビジネス上のリスクとなる可能性があります。

データベースおよびデータを変更するユーザーのリスクを最小限に抑えるには、以下のようにします。

- ルーチンを呼び出す特権を持つユーザーの数を最小限に抑えます。
- 過去の従業員のユーザー ID が除去されていることを確認するか、またはそのユーザー ID を再使用する場合、プロシージャ関連の特権を必ず審査してください。
- 悪意のあるユーザーがルーチンを呼び出していると思われる場合には、それらのルーチンごとに EXECUTE 特権を取り消す必要があります。

FENCED 節または NOT FENCED 節で定義されたルーチンのセキュリティー管理

ルーチン用の CREATE ステートメントを定式化する場合、FENCED 節または NOT FENCED 節のどちらを指定するか決定する必要があります。ルーチンを fenced または unfenced で作成することの利点について理解したなら、NOT FENCED として外部インプリメンテーションを行った場合の、実行中のルーチンに関連したリスクを見極めることが重要です。

- NOT FENCED 節で作成されたルーチンは、データベース・マネージャーの共用メモリーを無意識または意識的に破壊したり、データベースの制御構造を損壊したり、データベース・マネージャー・リソースにアクセスしてデータベース・マネージャーが失敗する可能性があります。データベースやその表を破壊するといったリスクもあります。

データベース・マネージャーとそのデータベースの整合性を確保するには、以下のようにします。

- NOT FENCED 節を指定する、作成予定のルーチンを徹底的にスクリーニングします。そのようなルーチンは、十分にテスト、デバッグを行い、予期せぬ副次効果を引き起こさないようにする必要があります。ルーチン・コードの検査では、メモリー管理と静的変数の使用に対して厳重な注意を払います。破損が生じる可能性が最も高いと言えるのは、コードがメモリーを適切に管理しない場合や静的変数を不正に使用する場合です。このような問題は、Java や .NET 以外のプログラミング言語でよく見られます。

NOT FENCED ルーチンを登録するには、CREATE_NOT_FENCED_ROUTINE 権限が必要です。CREATE_NOT_FENCED_ROUTINE 権限を付与する場合、付与された人はデータベース・マネージャーとそのすべてのリソースに無制限にアクセスできるようになることに注意してください。

注: NOT FENCED ルーチンは、共通の基準に準拠した構成ではサポートされていません。

ルーチンのセキュリティー確保

ルーチンを作成する際、ルーチン、ルーチン・ライブラリー (外部ルーチンの場合)、およびルーチンと対話することになるユーザーの特権を、ルーチンのセキュリティーを念頭に置いて管理するようにすることが重要です。

ルーチンのセキュリティー・ストラテジーほど手の込んだものは必要ありませんが、ルーチンのセキュリティーに寄与する要因に留意し、規律ある方法に従ってルーチンのセキュリティーを確保することをお勧めします。

前提条件

- 『ルーチンのセキュリティー』のトピックをお読みください。
- データベース・システム内のルーチンを完全にセキュリティー確保するには、以下の権限が必要です。
 - データベース・サーバーのオペレーティング・システムに対する root ユーザー・アクセス権。
 - SECADM または ACCESSCTRL のいずれかの権限。

ルーチンを作成する場合も、既存のルーチンを査定する場合も、ルーチンのセキュリティー確保の手順は同様です。

1. ルーチンを作成するのに必要な特権を持つユーザー ID の数を制限し、それらのユーザーがこの特権を持てるようにする。
 - ルーチン用の CREATE ステートメントの実行が正常に終了すると、そのユーザー ID に、EXECUTE 特権 (ユーザーがルーチン呼び出せる) と GRANT EXECUTE 特権 (ルーチン呼び出せる権限を他のユーザーに付与できる) を含む他の特権が自動的に付与されます。
 - この特権を持つユーザーを少なくし、ふさわしいユーザーにこの特権を付与します。
2. 不当または不十分にレビューまたはテストされた可能性のあるコードはないか、ルーチンを査定する。
 - ルーチンの出所を考慮に入れてください。そのルーチンを提供した関係者は信頼できますか?

- データベース・サーバーのファイル・システムに対して読み取りや書き込みを試みるコードや、そこにあるファイルを置き換えようとするコードのような、不正コードはないか探してください。
 - メモリー管理、ポインター操作、静的変数の使用に関連したコードのうち、ルーチンを失敗させる可能性のある、インプリメントが不十分なものはないか探してください。
 - コードが十分にテストされていることを確認してください。
3. 過度に危険と思われるルーチンやコーディングが不十分と思われるルーチンは受け入れない (リスクを考えれば、受け入れる価値があるとは限りません)。
 4. 含まれるリスクは、リスクの可能性が低いルーチンに関連したものだけにする。
 - SQL ユーザー定義 SQL ルーチンは、データベース・マネージャーのメモリー・スペース内で安全に実行できるため、NOT FENCED THREADSAFE ルーチンとしてデフォルトで作成されます。こうしたルーチンについては、何も行う必要はありません。
 - ルーチンを作成する CREATE ステートメントで、FENCED 節を指定してください。こうすることで、ルーチン操作がデータベース・マネージャーに影響を与えることがなくなります。これはデフォルトの節です。
 - ルーチンがマルチスレッド・ルーチンの場合は、ルーチン用の CREATE ステートメントで NOT THREADSAFE 節を指定します。こうすることで、ルーチン内に障害や不正コードがあっても、共有スレッド・プロセス内で実行される他のルーチンはその影響を受けなくなります。
 5. ルーチンが外部ルーチンの場合は、データベース・サーバーにルーチン・インプリメンテーションのライブラリー・ファイルまたはクラス・ファイルを置く。ルーチンのデプロイに関する一般推奨事項、および外部ルーチンのライブラリー・ファイルまたはクラス・ファイルのデプロイに関する具体的な推奨事項に従ってください。

SQL の入ったルーチンの許可およびバインド

ルーチン・レベルの許可については、まずルーチンに関連したいくつかのロール、それぞれのロールの判別、各ロールに関連した特権について説明することが大切です。

パッケージ所有者

ルーチンのインプリメンテーションにかかわっているパッケージの所有者です。パッケージの所有者とは、パッケージをデータベースにバインドするために BIND コマンドを実行したユーザーです (ただし、プリコンパイルバインドの OWNER オプションを使用してパッケージの所有権をオーバーライドし、別のユーザーにその所有権を設定した場合は別です)。BIND コマンドを実行すると、パッケージに関する EXECUTE WITH GRANT 特権がパッケージ所有者に付与されます。ルーチンのライブラリーまたは実行可能ファイルは、複数のパッケージで構成されることがあるので、そのような場合は、複数のパッケージ所有者が関連付けられます。

ルーチン定義者

ルーチンを登録するために CREATE ステートメントを発行した ID です。基本的には DBA がルーチン定義者になりますが、ルーチンのパッケージ所

有者がルーチン定義者になることもあります。ルーチンを呼び出すと、パッケージのロード時に、そのルーチンを実行する許可が、そのルーチンに関連した 1 つ以上のパッケージを実行する定義者の許可 (ルーチンの呼び出し側の許可ではない) に照らしてチェックされます。ルーチンを正常に呼び出すには、ルーチン定義者に以下の特権のいずれかが必要です。

- ルーチンの 1 つ以上のパッケージに関する EXECUTE 特権とルーチンに関する EXECUTE 特権
- DATAACCESS 権限

ルーチン定義者とルーチンのパッケージ所有者が同じユーザーの場合、ルーチン定義者はパッケージに関するその必要な EXECUTE 特権を持つこととなります。定義者がパッケージ所有者でない場合は、ACCESSCTRL または SECADM 権限を持つユーザーか、パッケージに関する CONTROL または EXECUTE WITH GRANT OPTION 特権が、パッケージに関する EXECUTE 特権を定義者に明示的に付与する必要があります。(パッケージの作成者には、パッケージに対する CONTROL および EXECUTE WITH GRANT OPTION 特権が自動的に付与されます。)

ルーチンを登録する CREATE ステートメントを実行すると、ルーチンに関する EXECUTE WITH GRANT OPTION 特権が定義者に暗黙的に GRANT されます。

ルーチン定義者のロールは、ルーチンに関連付けられているパッケージを実行する特権と、PUBLIC またはルーチンを読み出す必要のある特定のユーザーに対して、ルーチンに関する EXECUTE 特権を付与する特権を 1 つの許可 ID にカプセル化することです。

注: SQL ルーチンの場合、ルーチン定義者は暗黙的にパッケージ所有者にもなります。したがって定義者は、ルーチンの CREATE ステートメントの実行時に、ルーチンとルーチン・パッケージの両方に関する EXECUTE WITH GRANT OPTION を持つこととなります。

ルーチンの呼び出し側

ルーチンを読み出す ID です。どのユーザーをルーチンの呼び出し側とするかを決定するには、どのようにルーチンが呼び出されるかを考慮することが必要です。ルーチンは、コマンド・ウィンドウか、組み込み SQL アプリケーション内から呼び出せます。メソッドと UDF の場合は、ルーチン参照を別の SQL ステートメントに組み込みます。プロシージャは、CALL ステートメントによって呼び出します。アプリケーション内の動的 SQL の場合、呼び出し側は、すぐ上のレベルのルーチンのランタイム許可 ID か、そのルーチン呼び出しが入ったアプリケーションのランタイム許可 ID です (ただし、この ID は、そのルーチンまたはアプリケーションをバインドしたときに使用した DYNAMICRULES オプションに依存する場合があります)。静的 SQL の場合、呼び出し側は、ルーチンの参照を含むパッケージの OWNER プリコンパイル/BIND オプションの値です。正常にルーチンを読み出すには、これらのユーザーにルーチンに関する EXECUTE 特権が必要です。GRANT ステートメントを明示的に実行することによってこの特権をだれかに GRANT できるのは、ルーチンに関する EXECUTE WITH

GRANT OPTION 特権を持つユーザー (この特権が明示的に取り消されていないルーチン定義者も含まれる) と、ACCESSCTRL 権限または SECADM 権限を持つユーザーです。

例えば、動的 SQL を含むアプリケーションに関連したパッケージを DYNAMICRULES BIND でバインドした場合は、そのパッケージを呼び出した人物ではなく、そのアプリケーションの実行時許可 ID がパッケージ所有者になります。また、パッケージ所有者は、実際のバインド・プログラムになるか、プリコンパイルバインドの OWNER オプションの値になります。その場合、ルーチンの呼び出し側は、アプリケーションを実行しているユーザーの ID ではなく、そのオプション値をとります。

注:

1. ルーチン内の静的 SQL の場合、パッケージ所有者の特権は、ルーチン本体内の SQL ステートメントの実行に関して十分なものでなければなりません。ルーチンに対するネストされた参照がある場合、これらの SQL ステートメントで表へのアクセス権または EXECUTE 特権が必要になることもあります。
2. ルーチン内の動的 SQL の場合、特権を検査されるユーザー ID は、ルーチン本体の BIND の DYNAMICRULES オプションによって規制されます。
3. ルーチン・パッケージ所有者は、ルーチンの定義者に対してパッケージでの実行許可を与える必要があります。許可を与えるのは、ルーチンの登録の前でも後でもかまいませんが、いずれにしてもルーチンの呼び出し前でなければなりません。そうでない場合は、エラー (SQLSTATE 42051) が戻されます。

以下の図とテキストに、ルーチンに関する EXECUTE 特権の管理に関するステップを示します。

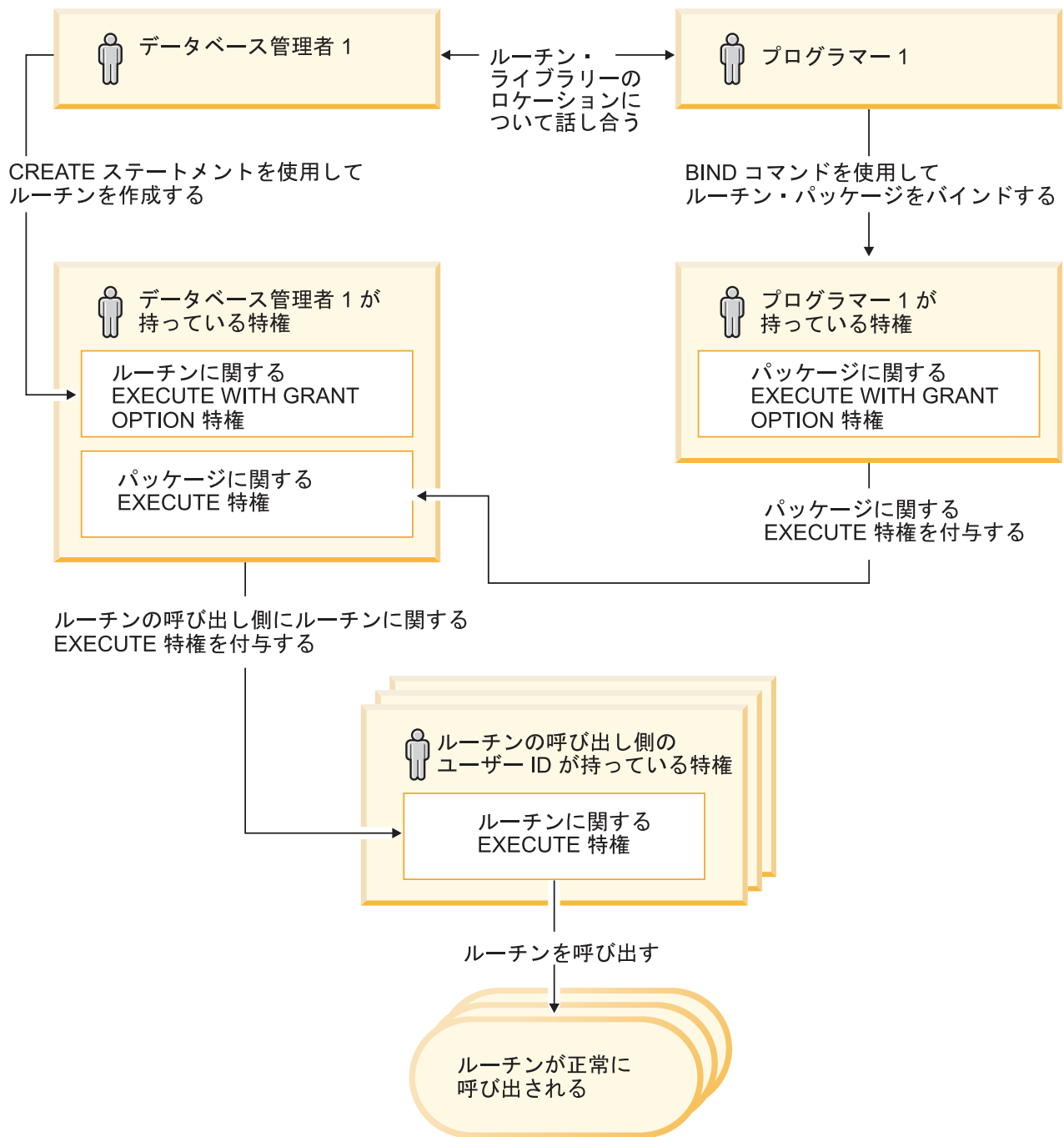


図 2. ルーチンに関する EXECUTE 特権の管理

1. 定義者は、該当する CREATE ステートメントを実行してルーチンを登録します。このステートメントでは、所定のレベルの SQL アクセス権の指定のもとにルーチンを DB2 に登録し、ルーチンのシグニチャーを確立し、ルーチンの実行可能プログラムの位置を指定します。定義者がパッケージ所有者も兼ねていない場合は、パッケージ所有者およびルーチン・プログラムの作成者と連絡を取り合って、ルーチン・ライブラリーが存在する場所を明確にする必要があります。そうしないと、CREATE ステートメントの EXTERNAL 節でその場所を正確に指定できないからです。CREATE ステートメントを正常に実行すると、定義者は

そのルーチンに関する EXECUTE WITH GRANT 特権を持つこととなりますが、ルーチンのパッケージに関する EXECUTE 特権を持つことにはなりません。

2. 定義者は、ルーチンの使用を許可される予定のすべてのユーザーに対してルーチンに関する EXECUTE 特権を付与する必要があります。(そのルーチンのパッケージがルーチンを再帰的に呼び出す場合、次のステップの前に必ずこのステップを実行する必要があります。)
3. パッケージ所有者は、ルーチン・プログラムをプリコンパイルおよびバインドするか、または他でそれを代行させます。プリコンパイルとバインドが正常に完了すると、パッケージの所有者にはそれぞれのパッケージに関する EXECUTE WITH GRANT OPTION 特権が暗黙的に GRANT されます。このリストのステップ 1 の後にこのステップが続いているのは、ルーチン内で SQL を繰り返す場合に備えるためではありません。どのような場合でもそのような繰り返しが存在しない場合は、ルーチンの CREATE ステートメントを実行する前にプリコンパイルバインドを実行してかまいません。
4. 各パッケージ所有者は、それぞれのルーチン・パッケージに関する EXECUTE 特権をルーチンの定義者に対して明示的に付与する必要があります。このステップは、前のステップの少し後で実行しなければなりません。パッケージ所有者がルーチンの定義者も兼ねている場合は、このステップを省略できます。
5. ルーチンの静的使用: ルーチンを参照するパッケージのバインド所有者は、ルーチンに対する EXECUTE 特権をすでに与えられていなければなりません。したがって、この時点で前のステップはすでに完了していなければなりません。ルーチンの実行時に DB2 は、すべてのパッケージに対する必要な EXECUTE 特権を定義者が持っているかどうかを検査します。したがってこの時点で、該当する各パッケージを対象としたステップ 3 は完了していなければなりません。
6. ルーチンの動的使用: DYNAMICRULES オプションによって制御される呼び出し元アプリケーションの許可 ID には、ルーチンに対する EXECUTE 特権がなければならず (ステップ 4)、ルーチンの定義者にはパッケージに対する EXECUTE 特権がなければなりません (ステップ 3)。

プロシージャが表に対する読み取り/書き込みを実行する時に起きるデータの競合

データベースの整合性を保つには、表に対する読み取りと書き込みの実行時に競合が起きないようにする必要があります。例えば、アプリケーションが EMPLOYEE 表を更新しようとしているときに、ステートメントがルーチンを呼び出すとします。しかもそのルーチンは EMPLOYEE 表の読み取りを試みて、アプリケーションによって行が更新されていることを検出したとします。その行はルーチンから見て不確定な状態にあります。おそらくその行には、更新されている列と、更新されていない列があるはずです。そのような部分的に更新済みの行に対してルーチンが処置を加えると、間違ったアクションがとられる可能性があります。そのような問題が起きないようにするために DB2 では、どの表の場合も競合を生じる操作は許可されていません。

ルーチンによる表の読み取りと書き込みの際に DB2 によってどのように競合が回避されるかを説明するには、以下の 2 つの用語が必要です。

最上位ステートメント

最上位ステートメントとは、アプリケーションから発行されるか、または最上位ステートメントとして呼び出されたストアード・プロシージャから発行される任意の SQL ステートメントのことです。プロシージャが動的コンパウンド・ステートメントまたはトリガー内で呼び出される場合は、そのコンパウンド・ステートメント、またはトリガーを起動させるステートメントが最上位ステートメントになります。SQL 関数または SQL メソッドにネストされた CALL ステートメントが含まれる場合は、その関数またはメソッドを呼び出すステートメントが最上位ステートメントになります。

表アクセス・コンテキスト

表アクセス・コンテキストとは、表操作で競合を生じてもかまわない許容範囲のことを言います。表アクセス・コンテキストは、以下の場合にそのつど作成されます。

- 最上位ステートメントが SQL ステートメントを発行する場合。
- UDF またはメソッドが呼び出される場合。
- プロシージャが、トリガー、動的コンパウンド・ステートメント、SQL 関数、SQL メソッドのいずれかから呼び出される場合。

例えば、アプリケーションがストアード・プロシージャを呼び出す場合、CALL が最上位ステートメントになるので、表アクセス・コンテキストが与えられます。ストアード・プロシージャが UPDATE を実行した場合も UPDATE が最上位ステートメントになる (このストアード・プロシージャは、最上位ステートメントとして呼び出したため) ので、表アクセス・コンテキストが与えられます。UPDATE が UDF を呼び出すと、その UDF には別個に表アクセス・コンテキストが与えられ、その UDF 内の SQL ステートメントは最上位ステートメントにはなりません。

読み取りまたは書き込みのためのアクセスがすでに実行された表は、そのアクセスを行った最上位ステートメント内では競合から保護されます。ただし、別の最上位ステートメントからか、または別の最上位ステートメントから呼び出されたルーチンからであれば、その表の読み取りまたは書き込みを行うことはできます。

次のような規則が定められています。

1. 表アクセス・コンテキスト内では、競合なしで 1 つの表の読み取りと書き込みを行えます。
2. 表が表アクセス・コンテキスト内で読み取られている場合、他のコンテキストでもその表を読み取ることができます。ただし、他のいずれかのコンテキストでその表への書き込みを試みた場合は、競合が起きます。
3. 表アクセス・コンテキスト内で表に書き込んでいる場合に、他のコンテキストで表の読み取りや書き込みを行うと必ず競合が生じます。

競合が起きた場合、その競合の原因となったステートメントにエラー (SQLCODE -746、SQLSTATE 57053) が戻されます。

以下に、表の読み取りおよび書き込みの競合の例を示します。

アプリケーションから次のようなステートメントが発行されたとします。

```
UPDATE t1 SET c1 = udf1(c2)
```

UDF1 には次のようなステートメントが入っています。

```
DECLARE cur1 CURSOR FOR SELECT c1, c2 FROM t1
OPEN cur1
```

この場合、規則 3 に違反したために競合が生じます。この種の競合は、アプリケーションまたは UDF を設計し直さないと解決できません。

以下の場合、競合は起きません。

アプリケーションから次のようなステートメントが発行されたとします。

```
DECLARE cur2 CURSOR FOR SELECT udf2(c1) FROM t2
OPEN cur2
FETCH cur2 INTO :hv
UPDATE t2 SET c2 = 5
```

UDF2 には次のようなステートメントが入っています。

```
DECLARE cur3 CURSOR FOR SELECT c1, c2 FROM t2
OPEN cur3
FETCH cur3 INTO :hv
```

カーソルを使用して UDF2 は表 T2 を読み取ることができます。2 つの表アクセス・コンテキストはどちらも同一の表を読み取れるからです。UDF2 が表を読み取っていてもアプリケーションは T2 を更新することができます。UDF2 は、更新とは別のアプリケーション・レベル・ステートメントで呼び出されたからです。

第 3 章 外部ルーチン

外部ルーチンは、データベース外部の、データベース・サーバーのファイル・システムに存在するプログラミング言語アプリケーションにロジックをインプリメントしたルーチンです。ルーチンと外部コード・アプリケーションとの関連付けは、ルーチンの CREATE ステートメントの EXTERNAL 節で宣言します。

作成できるのは、外部プロシージャ、外部関数、および外部メソッドです。これらすべては外部プログラミング言語でインプリメントされていますが、それぞれのルーチンの機能タイプには異なるフィーチャーがあります。外部ルーチンのインプリメントを決定する前に、『外部ルーチンの概説』のトピックを読むことによって、外部ルーチンの種類、またそれがどのようにインプリメントされ、どのように使用できるかを理解しておくことが大切です。その知識を基にして、データベース環境内で外部ルーチンを使うタイミングとその使い方に関して十分な情報を得た上での決定が行えるように、関連リンクの宛先のトピックから外部ルーチンの詳細を学習することができます。

外部ルーチンのフィーチャー

外部ルーチンは、一般的なルーチン・フィーチャーの大多数のサポート、および SQL ルーチンでサポートされない追加フィーチャーのサポートを提供します。以下のフィーチャーは、外部ルーチンには固有のものであります。

データベースの外部に常駐するファイル、データ、およびアプリケーションへのアクセス 外部ルーチンは、データベース自体の外部に常駐するデータまたはファイルにアクセスしてそれらを操作することができます。データベースの外部に常駐するアプリケーションを呼び出すこともできます。例えば、データ、ファイル、またはアプリケーションがデータベース・サーバーのファイル・システム内や使用できるネットワーク内に存在することがあります。

さまざまな外部ルーチンのパラメーター・スタイル・オプション

プログラミング言語での外部ルーチンのインプリメンテーションは、パラメーター・スタイルの選択を使用することによって行うことができます。選択したプログラミング言語の、希望するパラメーター・スタイルが存在する場合でも、時折、選択が存在します。一部のパラメーター・スタイルは、ルーチン・ロジック内で役立つ可能性のある、*dbinfo* 構造という構造内のルーチンとの間の追加データベースおよびルーチンのプロパティ情報の引き渡しをサポートします。

スクラッチパッドを使用した次の外部関数の呼び出しまでの状態の保存

外部ユーザー定義関数は、値のセットに対する次の関数呼び出しまでの状態の保存のサポートを提供します。これは、スクラッチパッドと呼ばれる構造を使用して行われます。これは、集約値を戻す関数と、バッファの初期化といったロジックの初期セットアップを必要とする関数の両方において便利です。

呼び出しタイプは個別の外部関数の呼び出しを識別する

外部ユーザー定義関数は、値のセットに対して複数回呼び出されます。それ

それぞれの呼び出しは、関数ロジック内で参照可能な呼び出しタイプの値で識別されます。例えば、ある関数の最初の呼び出し、データのフェッチ呼び出し、および最終呼び出しに関して特別な呼び出しタイプが存在します。呼び出しタイプは、特定のロジックを特定の呼び出しタイプに関連付けることができるため便利です。

外部関数とメソッドのフィーチャー

外部関数および外部メソッドは、入力データの特定のセットで何回か呼び出されて出力値のセットを生成する可能性のある関数に対してサポートを提供します。

外部関数とメソッドのフィーチャーについてさらに学習する場合は、以下のトピックを参照してください。

- 『外部スカラー関数』
- 70 ページの『外部スカラー関数およびメソッドの処理モデル』
- 71 ページの『外部表関数』
- 71 ページの『外部表関数の処理モデル』
- 73 ページの『Java の表関数実行モデル』
- 74 ページの『外部関数とメソッドのスクラッチパッド』
- 78 ページの『32 ビット・オペレーティング・システムおよび 64 ビット・オペレーティング・システムでのスクラッチパッド』

これらのフィーチャーは外部関数とメソッドに固有のものであり、SQL 関数および SQL メソッドには当てはまりません。

外部スカラー関数

外部スカラー関数は、外部のプログラム言語でロジックをインプリメントしたスカラー関数です。

こうした関数を作成して、既存の SQL 関数のセットを拡張するために使用できますし、DB2 組み込み関数 (例えば、LENGTH や COUNT) と同じやり方で呼び出すことができます。つまり、SQL ステートメント内で式が有効な場所であればどこからでも参照できるということです。

外部スカラー関数のロジックの実行は DB2 データベース・サーバーで行われます。また、組み込み関数やユーザー定義の SQL スカラー関数とは異なり、外部関数のロジックはデータベース・サーバー・ファイル・システムへのアクセス、システム呼び出しの実行、またはネットワークへのアクセスが可能です。

外部スカラー関数は、SQL データの読み取りはできますが、その変更は行えません。

外部スカラー関数は、1 つの関数参照で何度も呼び出すことができ、スクラッチパッド (メモリー・バッファー) の使用によって、その呼び出しと呼び出しの間で状態を維持することができます。このような機能は、最初のセットアップ・ロジックが複雑な場合に特に便利です。セットアップ・ロジックを最初の呼び出しで実行するときに、スクラッチパッドを使用して、スカラー関数のそれ以降の呼び出しでアクセスまたは更新するいくつかの値を保管できるからです。

外部スカラー関数のフィーチャー

- SQL ステートメント内で式がサポートされている場所であればどこからでも参照できます。
- スカラー関数の出力は、呼び出し元の SQL ステートメントによって直接処理できます。
- 外部スカラー・ユーザー定義関数の場合は、関数の反復呼び出しの際に、スクラッチパッドを使用して、呼び出しと呼び出しの間で状態を維持できます。
- サーバーで実行されるので、述部で使用する時のパフォーマンスが高くなります。サーバーで関数を候補行に対して適用できる場合は、クライアント・マシンに行を送信する前の時点でその行を考慮の対象から除外できる場合が多いので、サーバーからクライアントに渡す必要のあるデータ量を削減できます。

制限

- スカラー関数内ではトランザクション管理を行えません。つまり、スカラー関数内では COMMIT や ROLLBACK を発行できません。
- 結果セットを戻すことはできません。
- スカラー関数は、入力セットごとに 1 つのスカラー値を戻すようになっています。
- 外部スカラー関数は、一度の呼び出しによる使用を想定していません。むしろ、関数に対する 1 つの参照と 1 つの入力セットを用意して、各入力ごとに関数を一度ずつ呼び出し、そのたびに関数から 1 つのスカラー値が戻される、という設計になっています。スカラー関数を作成するときには、最初の呼び出しで一部のセットアップ作業を行い、その後で呼び出し時にアクセスできる一部の情報を保管するように設計できます。一度の呼び出しだけを必要とする機能には、SQL スカラー関数のほうが適しています。
- 単一パーティション・データベースでは、外部スカラー関数に SQL ステートメントを含めることができます。これらのステートメントは、表のデータの読み取りは行えますが、その変更はできません。データベースに複数のパーティションがある場合、外部スカラー関数に SQL ステートメントを含めることはできません。SQL スカラー関数は、データの読み取りや変更を行う SQL ステートメントを含めることができます。

一般的な使用法

- DB2 組み込み関数のセットを拡張します。
- 本来 SQL は実行できない SQL ステートメント内のロジックを実行します。
- 副照会としてよく再利用されるスカラー照会を SQL ステートメント内でカプセル化します。例えば、郵便番号を例にあげると、郵便番号が掲載されている都市の表を検索します。

サポートされている言語

- C
- C++
- Java
- OLE

- .NET 共通言語ランタイム言語

注:

1. 集約関数を作成するための機能は限定されています。列関数ともいう集約関数は、一連の類似値 (データ列) を受け取って、1 つの応答を戻します。ユーザー定義集約関数を作成できるのは、組み込み集約関数をソースとする場合だけです。例えば、基本タイプ INTEGER に基づいて特殊タイプ SHOESIZE を定義してある場合は、既存の組み込み集約関数 AVG(INTEGER) をソースとして、AVG(SHOESIZE) という関数を集約関数として定義できます。
2. また、行を戻す関数を作成することもできます。これは、行関数と呼ばれますが、構造化タイプ用の transform 関数としてのみ使用することができます。行関数の出力は単一行です。

外部スカラー関数およびメソッドの処理モデル

FINAL CALL 指定を使用して定義されたメソッドおよびスカラー UDF の処理モデルは以下のとおりです。

FIRST 呼び出し

これは特殊ケースの NORMAL 呼び出しですが、関数を使用して任意の初期処理を実行できるようにするための「最初」の呼び出しを表します。引数が評価されてから、関数に渡されます。通常、この呼び出しでは関数から値が戻されますが、エラーが戻される場合もあります。後者の場合は NORMAL または FINAL 呼び出しは行われません。FIRST 呼び出しでエラーが戻された場合は、FINAL 呼び出しは行われなため、メソッドまたは UDF は、戻る前に終結処理を行う必要があります。

NORMAL 呼び出し

これは、ステートメントのデータとロジックで示されているとおり、関数の 2 番目から最後から 2 番目までのすべての呼び出しを指します。どの NORMAL 呼び出しでも、引数が評価されてから渡された後で関数から値が戻されることになっています。NORMAL 呼び出しでエラーが戻された場合、それ以上 NORMAL 呼び出しは行われずに、FINAL 呼び出しが行われます。

FINAL 呼び出し

これは、ステートメントの終わりの処理 (またはカーソルのクローズ) の時点で行われる特殊な呼び出しです。ただし、FIRST 呼び出しが正常に完了していることを前提とします。FINAL 呼び出しでは引数値は渡されません。この呼び出しが行われるのは、関数がすべてのリソースを終結処理できるようにするためです。この呼び出しでは関数は値を戻しませんが、エラーを戻すことはあります。

FINAL CALL を指定して定義されていないメソッドまたはスカラー UDF の場合、関数への NORMAL 呼び出しのみが行われ、その場合は通常は、各呼び出しの値が戻されます。NORMAL 呼び出しでエラーが戻された場合や、ステートメントで別のエラーが生じた場合、その関数に対してはそれ以上呼び出しは行われません。

注: このモデルは、メソッドおよびスカラー UDF の通常のエラー処理を説明しています。システム障害や通信問題が発生した場合、エラー処理モデルによって指示

された呼び出しが行われないことがあります。例えば、FENCED UDF の場合、db2udf fenced 処理が何らかの原因で早く終了してしまうと、DB2 は指示された呼び出しを行うことができません。

外部表関数

ユーザー定義表関数は、表を参照している SQL にその表を引き渡します。表 UDF 参照は、SELECT ステートメントの FROM 節内でのみ有効です。表関数を使用する際は、次のことに注意してください。

- 表関数は表を送達しますが、DB2 と UDF の間の物理インターフェースは 1 行ずつ行われます。表関数への呼び出しには、OPEN、FETCH、CLOSE、FIRST、および FINAL の 5 タイプがあります。FIRST および FINAL 呼び出しがあるかどうかは、UDF の定義方法によって決まります。これらの呼び出しの判別には、スカラー関数で使われるのと同じ呼び出しタイプ 機構が使用されます。
- 表関数の CREATE FUNCTION ステートメントの RETURNS 節で定義されたすべての結果列を、戻さなければならないというわけではありません。CREATE FUNCTION の DBINFO キーワード、および対応する dbinfo 引数によって、特定の表関数参照に必要な列だけを戻すよう最適化できます。
- 戻される個々の列値は、スカラー関数が戻す値と同じ書式です。
- 表関数の CREATE FUNCTION ステートメントには、CARDINALITY 指定があります。これを指定することにより、定義者は DB2 オプティマイザーに結果の概算のサイズを通知することができ、それによってオプティマイザーは関数が参照されるときにより適切な決定を下すことができます。

表関数の CARDINALITY として指定された値と関係なく、カーディナリティーが無限の関数、つまり、FETCH 呼び出しの際に常に行を戻す関数を定義しないよう注意してください。DB2 では、照会処理内の触媒として end-of-table 条件を想定する状況が多くあります。GROUP BY や ORDER BY を使用している場合などがそうです。DB2 は、end-of-table に到達するまで、集合用のグループを作成せず、またすべてのデータがそろそろまで、ソートを行うことはできません。そのため、end-of-table 条件 (SQL 状態値 '02000') を決して戻さない表関数では、それを GROUP BY や ORDER BY 節で使用すると、無限処理ループが生じることがあります。

外部表関数の処理モデル

FINAL CALL 指定を使用して定義された表 UDF の処理モデルは以下のとおりです。

FIRST 呼び出し

この呼び出しは最初の OPEN 呼び出しの前に行いますが、その目的は、関数がすべての初期処理を実行できるようにすることにあります。この呼び出しの前に、スクラッチパッドがクリアされます。引数が評価されてから、関数に渡されます。この関数は行を戻しません。この関数がエラーを戻した場合、それ以降この関数への呼び出しは行われません。

OPEN 呼び出し

この呼び出しが行われるのは、スキャンに固有の特別な OPEN 処理を関数で実行できるようにするためです。この呼び出しの前にスクラッチパッド (ある場合) がクリアされることはありません。引数が評価されてから引き渡されます。この関数は、OPEN 呼び出しで行を戻すことはありません。この

関数が OPEN 呼び出しでエラーを戻した場合、FETCH または CLOSE 呼び出しは行われませんが、ステートメントの終わりで FINAL 呼び出しは行われます。

FETCH 呼び出し

FETCH 呼び出しは、表の終わりを意味する SQLSTATE 値が関数から戻されるまで継続して行われます。UDF は、この呼び出しに対応して、データ行を開発して戻すこととなります。引数値が関数に渡されることがありますが、その値は、OPEN のときに渡されたのと同じ値を指しています。したがって、この引数値は現行値でない可能性もあるので、信用することはできません。表関数の次の呼び出しまで現行値をそのまま維持している必要がある場合、スクラッチパッドを使用してください。この関数は、FETCH 呼び出しでエラーを戻すことはありますが、その場合でも CLOSE 呼び出しを行うことはできます。

CLOSE 呼び出し

この呼び出しが行われるのは、スキャンまたはステートメントの終了時点です。ただし、OPEN 呼び出しが正常に完了していることを前提とします。どの引数値も現行値ではありません。この関数はエラーを戻すことがあります。

FINAL 呼び出し

FINAL 呼び出しが行われるのは、ステートメントの終了時点です。ただし、FIRST 呼び出しが正常に完了していることを前提とします。この呼び出しが行われるのは、関数がすべてのリソースを終結処理できるようにするためです。この呼び出しでは関数は値を戻しませんが、エラーを戻すことはあります。

FINAL CALL を指定して定義されていない表 UDF の場合、関数への OPEN、FETCH、および CLOSE 呼び出しのみが行われます。どの OPEN 呼び出しでも、その前にスクラッチパッド (ある場合) がクリアされます。

FINAL CALL を指定して定義された表 UDF と、NO FINAL CALL を指定して定義されたものの違いは、表関数アクセスは「内部寄りの」アクセスとなる結合または副照会に関連したシナリオを見れば明らかになります。例えば、次のようなステートメントがあるとします。

```
SELECT x,y,z,... FROM table_1 as A,  
       TABLE(table_func_1(A.col1,...)) as B  
WHERE ...
```

この場合オプティマイザーは、table_1 の各行ごとに table_func_1 のスキャンをオープンします。それは、table_1 の col1 の値 (これが table_func_1 に渡されます) が使用されて表関数スキャンが定義されるからです。

上記の構文スタイルに従ったステートメントは、FOR EACH ROW OF 節を使用して再度記述することができます。照会結果は古いスタイルと同じになります。以下に例を示します。

```
SELECT B.x,B.y, B.z,...  
       TABLE(table_func_1(  
         FOR EACH ROW OF (SELECT * FROM table_1 WHERE ...))) as B
```

NO FINAL CALL の表 UDF の場合、table_1 の各行ごとに OPEN、FETCH、FETCH、...、CLOSE の呼び出しシーケンスが繰り返されます。なお、OPEN 呼び出しのたびに、新しいスクラッチパッドが支給されることに注意してください。スキヤンの終了時点ごとにさらに別のスキヤンがあるかどうかは表関数には分からないので、表関数は、CLOSE 処理中に完全な終結処理を実行する必要があります。そのため、繰り返しの必要な 1 回だけオープンされる重要な処理がある場合には効率が悪くなってしまいます。

FINAL CALL の表 UDF は、一回限りの FIRST 呼び出しおよび一回限りの FINAL 呼び出しの手段になります。これらの呼び出しを使用すれば、表関数のすべてのスキヤンを通して初期化と終了の手間を軽減することができます。これまでと同様に外側の表の各行ごとに OPEN、FETCH、FETCH、...、CLOSE の呼び出しは行われますが、FINAL 呼び出しが出されることが表関数には分かっているので、表関数は CLOSE 呼び出しの時点で一切終結処理（およびその後の OPEN での再割り振り）をする必要はなくなります。また、表関数リソースは複数のスキヤンを対象とすることが主な原因ですが、スキヤンのたびにスクラッチパッドがクリアされることはないことにも注意してください。

表 UDF を使用すれば、2 つのさらに別の呼び出しタイプを管理する必要はありますが、その代償として、上記の結合と副照会のシナリオに示されているような大幅な効率化を実現することができます。表関数を FINAL CALL と定義するべきかどうかは、予定している使用法によって決まります。

Java の表関数実行モデル

Java で作成されて PARAMETER STYLE DB2GENERAL を使用する表関数の場合、DB2 での特定のステートメントの処理のさまざまな時点で何が起きるか分かっていることが大切です。以下の表は、通常の場合のそれに関する詳細を示しています。NO FINAL CALL の場合と FINAL CALL の場合の両方が取り上げられており、どちらも SCRATCHPAD が指定されていると想定しています。

スキヤンの時点	NO FINAL CALL LANGUAGE JAVA SCRATCHPAD	FINAL CALL LANGUAGE JAVA SCRATCHPAD
表関数に対する最初の OPEN の前	<ul style="list-style-type: none"> 呼び出しなし。 	<ul style="list-style-type: none"> クラス・コンストラクターが呼び出される（すなわち、新しいスクラッチパッド）。FIRST 呼び出しで UDF メソッドが呼び出される。 コンストラクターが、クラスおよびスクラッチパッド変数を初期化する。メソッドが Web サーバーに接続する。

スキヤンの時点	NO FINAL CALL LANGUAGE JAVA SCRATCHPAD	FINAL CALL LANGUAGE JAVA SCRATCHPAD
表関数に対する各 OPEN 時	<ul style="list-style-type: none"> クラス・コンストラクターが呼び出される (すなわち、新しいスクラッチパッド)。 OPEN 呼び出しで UDF メソッドが呼び出される。 コンストラクターが、クラスおよびスクラッチパッド変数を初期化する。メソッドが Web サーバーに接続し、Web データのスキヤンをオープンする。 	<ul style="list-style-type: none"> OPEN 呼び出しで UDF メソッドがオープンされる。 メソッドが、必要な Web データのスキヤンをオープンする。(スクラッチパッドに保管されるものに応じて、CLOSE 位置変更後の再オープンを避けることができる。)
新しい行の表関数データに対する各 FETCH 時	<ul style="list-style-type: none"> FETCH 呼び出しで UDF メソッドが呼び出される。 メソッドが、次の行のデータまたは EOT を取り出して戻す。 	<ul style="list-style-type: none"> FETCH 呼び出しで UDF メソッドが呼び出される。 メソッドが、新しい行のデータまたは EOT を取り出して戻す。
表関数に対する CLOSE 時	<ul style="list-style-type: none"> CLOSE 呼び出しで UDF メソッドが呼び出される。 close() メソッドがクラスに対してあれば、呼び出される。 メソッドがその Web スキヤンをクローズし、Web サーバーから切断する。close() は必要とされない。 	<ul style="list-style-type: none"> CLOSE 呼び出しで UDF メソッドが呼び出される。 メソッドはスキヤンの最上部に位置変更するか、スキヤンをクローズする。持続されるどんな状態でも、スクラッチパッドに保管できる。
表関数に対する最後の CLOSE の後	<ul style="list-style-type: none"> 呼び出しなし。 	<ul style="list-style-type: none"> FINAL 呼び出しで UDF メソッドが呼び出される。 close() メソッドがクラスに対してあれば、呼び出される。 メソッドは Web サーバーから切断する。close() メソッドは必要とされない。

注:

- 「UDF メソッド」とは、UDF をインプリメントした Java クラス・メソッドのことです。これは、CREATE FUNCTION ステートメントの EXTERNAL NAME 節で識別されるメソッドです。
- NO SCRATCHPAD が指定された表関数では、UDF メソッドの呼び出しはこの表で示されているとおりですが、ユーザーはスクラッチパッドによる連続性を求めないために、クラス・コンストラクターが DB2 によって呼び出され、各呼び出しの前に新しいオブジェクトがインスタンス化されます。NO SCRATCHPAD が指定された (したがって連続性がない) 表関数が役立つかどうかは不明ですが、それらはサポートされています。

外部関数とメソッドのスクラッチパッド

スクラッチパッドを使用すれば、次の呼び出し時までユーザー定義関数またはメソッドの状態を保持しておくことができます。例えば、以下に次回の呼び出しまで状態を保持しておけば便利であることを示す例を 2 つ示します。

1. 正確に言えば、保管状態に依存する関数またはメソッド。

このような関数またはメソッドの例として、最初の呼び出し時に「1」を返し、2 回目以降の呼び出しごとに結果を 1 ずつ増分する単純な `counter` 関数があります。この関数を使用すると、特定の状況下では次のように `SELECT` 結果の行数を数えることができます。

```
SELECT counter(), a, b+c, ...
FROM tablex
WHERE ...
```

関数には、複数の呼び出しにまたがってカウンターの現行値を保管する場所が必要です。それによって、後続の呼び出しでも必ず同じ値が確保されます。その後の呼び出しごとにその値は増加されて、関数の結果として戻されます。

このタイプのルーチンは限定的なルーチンではありません。つまりその出力が、その `SQL` 引数の値にのみ依存することはありません。

2. 特定の初期化アクションを実行する機能によってパフォーマンスを改善できる関数またはメソッド。

このような関数またはメソッドの例として、文書アプリケーションの一部を成す `match` 関数があります。これは、特定の文書に特定のストリングが入っていれば「Y」を、入っていなければ「N」を戻します。

```
SELECT docid, doctitle, docauthor
FROM docs
WHERE match('myocardial infarction', docid) = 'Y'
```

このステートメントは、最初の引数で表される特定のテキスト・ストリング値を含む文書すべてを戻します。 `match` が行うことは以下のとおりです。

- 最初の処理に限り、以下を実行します。

ストリング `myocardial infarction` が入っていて、しかも `DB2` の外部で保存されているすべての文書 `ID` のリストを文書アプリケーションで検索します。この検索は処理に負荷がかかる処理であるため、関数はこの処理を 1 回だけ行い、検索したリストをその後の呼び出しでの使用に利用しやすい場所に保管します。

- 各呼び出し時には、以下を実行します。

この最初の呼び出しで保管された文書 `ID` のリストを用いて、2 番目の引数として渡された文書 `ID` がこのリストに載っているかどうかを確認します。

このタイプのルーチンは限定的ルーチンです。その応答は、入力される引数値にのみ依存します。上記の関数は、ある呼び出しから次の呼び出しへ情報を保管できるかどうかによってパフォーマンス（正確さではない）が左右されます。

以下のように、`CREATE` ステートメントに `SCRATCHPAD` を指定すれば、上述の 2 つの要件は両方とも満たされます。

```
CREATE FUNCTION counter()
  RETURNS int ... SCRATCHPAD;

CREATE FUNCTION match(varchar(200), char(15))
  RETURNS char(1) ... SCRATCHPAD 10000;
```

SCRATCHPAD キーワードは、ルーチン用のスクラッチパッドを割り振って保持するよう DB2 に指示します。スクラッチパッドのデフォルトのサイズは 100 バイトですが、スクラッチパッド・サイズ (バイト数) を指定することができます。 *match* の例は 10000 バイトの長さです。DB2 は、スクラッチパッドを最初の呼び出し前のバイナリー数のゼロに初期化します。表関数のスクラッチパッドが定義されている場合に、NO FINAL CALL (デフォルト) を使用してその表関数が定義されていると、DB2 は各 OPEN 呼び出しの前にスクラッチパッドをリフレッシュします。表関数オプション FINAL CALL を指定すると、DB2 は、初期化後のスクラッチパッドの内容を検査も変更もしません。スクラッチパッドを使用して定義されたスカラー関数の場合も、DB2 は初期化後のスクラッチパッドの内容を検査も変更もしません。各呼び出しごとにスクラッチパッドを指すポインターがルーチンに渡され、DB2 はそのルーチンの状態情報をスクラッチパッド内に保存します。

したがって *counter* の例の場合、最後に戻された値がスクラッチパッドに保管されます。また *match* の例では、スクラッチパッドが十分に大きい場合は文書のリストをスクラッチパッドに保管し、十分に大きくない場合はリスト用にメモリーを割り振って、取得したメモリーのアドレスをスクラッチパッドに保存します。スクラッチパッドは可変長にすることができます。その長さは、ルーチンの CREATE ステートメント内に定義します。

スクラッチパッドが適用されるのは、ステートメント内のルーチンへの個々の参照に対してのみです。ステートメント内のルーチンに対して複数の参照がある場合、どの参照にもそれぞれ独自のスクラッチパッドがあることになるので、参照同士が互いに通信しあうのにスクラッチパッドを使用することはできません。スクラッチパッドは、単一の DB2 エージェント (エージェントとは、ステートメントのあらゆる側面の処理を実行する DB2 エンティティーのことです) に対してのみ適用されます。エージェント同士がスクラッチパッド情報を共用するのを調整するための「グローバル・スクラッチパッド」はありません。このことは、ステートメントを処理するエージェントが DB2 によって複数確立される (単一パーティションまたはマルチパーティション・データベースのどちらかで) 場合は特に重要です。そのような場合、ステートメント内のルーチンへの参照は 1 つしかない場合でも、作業を行うエージェントは複数存在していて、そのおのおのが独自のスクラッチパッドをもつこととなります。マルチパーティション・データベースでは、UDF を参照するステートメントは、複数のパーティション上のデータを処理し、各パーティション上で UDF を呼び出しますが、スクラッチパッドは 1 つのパーティションにしか適用されません。結果として、UDF が実行されるパーティションごとにスクラッチパッドが 1 つずつ存在することとなります。

関数が正しく実行されるかどうか、その関数への参照ごとに 1 つのスクラッチパッドがあるかどうかで決まる場合、その関数を DISALLOW PARALLEL として登録します。これで、関数は 1 つのパーティションでしか実行されなくなるので、関数への 1 つの参照につき必ず 1 つのスクラッチパッドしか存在しないようにすることができます。

UDF またはメソッドは、システム・リソースを必要とする場合があるので、UDF またはメソッドを定義するときに FINAL CALL キーワードを使用すると便利です。このキーワードは、ステートメント処理の終了時点で UDF またはメソッドを呼び出すよう DB2 に指示するので、UDF またはメソッドはそのシステム・リソースを解放することができます。ルーチンは獲得したすべてのリソースを解放する

ことが不可欠です。ステートメントが繰り返し呼び出される環境では、小さい不手際は大きい不手際につながることもあり、大きい不手際は DB2 が破壊される原因になることがあります。

スクラッチパッドのサイズは固定されているので、UDF またはメソッド自体にメモリーの割り振りを組み込むことにより、最終呼び出しを利用してメモリーを解放するのも一案です。例えば上記の *match* 関数は、特定のテキスト・ストリングと一致する文書がどのくらいあるかを予測できません。したがって、*match* の定義は次のように行うとよいでしょう。

```
CREATE FUNCTION match(varchar(200), char(15))
  RETURNS char(1) ... SCRATCHPAD 10000 FINAL CALL;
```

UDF またはメソッドがスクラッチパッドを使用していてしかも副照会で参照される場合、もし、UDF またはメソッドに最終呼び出しが指定されているならば、副照会から次の副照会までの間に DB2 は最終呼び出しを行って、スクラッチパッドの内容をリフレッシュすることを決定する場合があります。UDF またはメソッドを副照会で使用している場合は、FINAL CALL や呼び出しタイプ引数を使用して UDF またはメソッドを定義するか、またはスクラッチパッドのバイナリー数のゼロ状態を必ず検査すれば、リフレッシュが起きないようにすることができます。

FINAL CALL を指定する場合は、UDF またはメソッドがタイプ FIRST の呼び出しを受け取ることに注意してください。これは、永続リソースを獲得して初期化するために使用することができます。

以下は、スクラッチパッドを使って列の項目の平方和を計算する UDF の単純な Java の例です。この例は、列を取り込み、列の先頭から現在行の項目までの累積平方和を含む列を戻します。

```
CREATE FUNCTION SumOfSquares(INTEGER)
  RETURNS INTEGER
  EXTERNAL NAME 'UDFsrv!SumOfSquares'
  DETERMINISTIC
  NO EXTERNAL ACTION
  FENCED
  NOT NULL CALL
  LANGUAGE JAVA
  PARAMETER STYLE DB2GENERAL
  NO SQL
  SCRATCHPAD 10
  FINAL CALL
  DISALLOW PARALLEL
  NO DBINFO@
```

```
// Sum Of Squares using Scratchpad UDF
public void SumOfSquares(int inColumn,
                        int outSum)
  throws Exception
{
  int sum = 0;
  byte[] scratchpad = getScratchpad();

  // variables to read from SCRATCHPAD area
  ByteArrayInputStream byteArrayIn = new ByteArrayInputStream(scratchpad);
  DataInputStream dataIn = new DataInputStream(byteArrayIn);

  // variables to write into SCRATCHPAD area
  byte[] byteArrayCounter;
  int i;
```

```

ByteArrayOutputStream byteArrayOut = new ByteArrayOutputStream(10);
DataOutputStream dataOut = new DataOutputStream(byteArrayOut);

switch(getCallType())
{
  case SQLUDF_FIRST_CALL:
    // initialize data
    sum = (inColumn * inColumn);
    // save data into SCRATCHPAD area
    dataOut.writeInt(sum);
    byteArrayCounter = byteArrayOut.toByteArray();
    for(i = 0; i < byteArrayCounter.length; i++)
    {
      scratchpad[i] = byteArrayCounter[i];
    }
    setScratchpad(scratchpad);
    break;
  case SQLUDF_NORMAL_CALL:
    // read data from SCRATCHPAD area
    sum = dataIn.readInt();
    // work with data
    sum = sum + (inColumn * inColumn);
    // save data into SCRATCHPAD area
    dataOut.writeInt(sum);
    byteArrayCounter = byteArrayOut.toByteArray();
    for(i = 0; i < byteArrayCounter.length; i++)
    {
      scratchpad[i] = byteArrayCounter[i];
    }
    setScratchpad(scratchpad);
    break;
}
//set the output value
set(2, sum);
} // SumOfSquares UDF

```

SumOfSquares UDF と同じタスクを実行する組み込み DB2 関数があることに注意してください。この例が選ばれたのは、スクラッチパッドの使用を示すためです。

32 ビット・オペレーティング・システムおよび 64 ビット・オペレーティング・システムでのスクラッチパッド

UDF またはメソッドのコードを 32 ビットと 64 ビットのオペレーティング・システムで相互に移植できるようにするには、64 ビット値の入ったスクラッチパッドを作成および使用する仕方に気を付ける必要があります。64 ビット・ポインターや `sqlint64` `BIGINT` 変数などの 1 つ以上の 64 ビット値の入ったスクラッチパッド構造では、明示的な長さ変数を宣言しないようお勧めします。

以下は、スクラッチパッドの構造宣言のサンプルです。

```

struct sql_scratchpad
{
  sqlint32 length;
  char data[100];
};

```

スクラッチパッドの独自の構造の定義では、ルーチンには次のような 2 つの選択肢があります。

1. スクラッチパッド `sql_scratchpad` 全体を再定義します。この場合、明示的な長さフィールドを組み込む必要があります。以下に例を示します。


```

struct sql_spad
{
    sqlint32 length;
    sqlint32 int_var;
    sqlint64 bigint_var;
};
void SQL_API_FN routine( ..., struct sql_spad* scratchpad, ... )
{
    /* Use scratchpad */
}

```

2. スクラッチパッド `sql_scratchpad` のデータ部分だけを再定義します。この場合、長さフィールドは必要ありません。

```

struct spaddata
{
    sqlint32 int_var;
    sqlint64 bigint_var;
};
void SQL_API_FN routine( ..., struct sql_scratchpad* spad, ... )
{
    struct spaddata* scratchpad = (struct spaddata*)spad->data;
    /* Use scratchpad */
}

```

アプリケーションは、スクラッチパッド内の長さフィールドの値を変更できないので、最初の例に示されているようにルーチンをコーディングしてもあまり有益ではありません。また 2 番目の例も、それぞれ異なるワード・サイズのコンピューター同士で相互に移植できるので、ルーチンを作成するにはこちらのほうがより望ましい方法です。

外部ルーチンの SQL

外部プログラム言語 (C、Visual Basic、C#、Java など) で書かれたどのルーチン内でも SQL を使用することができます。

ルーチン (ストアド・プロシージャ、UDF) の場合の CREATE ステートメントと、メソッドの場合の CREATE TYPE ステートメントには、ルーチンまたはメソッドの SQL アクセス・レベルを定義する節を組み込みます。ルーチンに組み込まれた SQL の特性に基づいて、以下のようなアプリケーション節を選択しなければなりません。

NO SQL

ルーチンには SQL はまったく入りません。

CONTAINS SQL

SQL は入りますが、データの読み取りや書き込みは行いません (例えば、SET SPECIAL REGISTER)。

READS SQL DATA

表からの読み取りを行う SQL は入ります (SELECT、VALUES ステートメント) が、表データは変更しません。

MODIFIES SQL DATA

表を更新する SQL が入ります。これは、ユーザー表を直接 (INSERT、UPDATE、DELETE ステートメント) または DB2 のカタログ表を暗黙で (DDL ステートメント) 更新します。この節を使用できるのは、ストアド・プロシージャと SQL 形式の表関数だけです。

実行時に DB2 は、定義されたレベルをルーチンが超えていないかどうかを検証します。例えば、CONTAINS SQL と定義されたルーチンが表からの選択を試みると、SQL データを読み取ろうとするのでエラー (SQLCODE -579、SQLSTATE 38004) になります。また、ネストされたルーチン参照も、参照を含む同じ SQL レベルか、より厳密な SQL レベルでなければなりません。例えば、SQL データの変更を行うルーチンは、SQL データの読み取りを行うルーチンを呼び出せますが、SQL データの読み取り専用のルーチン (READS SQL DATA 節を定義したルーチン) は、SQL データの変更を行うルーチンを呼び出せません。

ルーチンは、呼び出し元アプリケーションのデータベース接続の有効範囲内で SQL ステートメントを実行します。ルーチンは独自の接続を確立したり、呼び出し元アプリケーションの接続をリセットしたりすることはできません (SQLCODE -751、SQLSTATE 38003)。

MODIFIES SQL DATA と定義されたストアド・プロシージャだけが、COMMIT および ROLLBACK ステートメントを発行することができます。他のタイプのルーチン (UDF とメソッド) は、COMMIT も ROLLBACK も発行できません (SQLCODE -751、SQLSTATE 38003)。MODIFIES SQL DATA と定義されたストアド・プロシージャはトランザクションのコミットまたはロールバックを行うことはできますが、COMMIT または ROLLBACK は呼び出し元のアプリケーションから発行して、変更が不用意にコミットされないようにすることをお勧めします。データベースに対してタイプ 2 接続を確立しているアプリケーションからストアド・プロシージャが呼び出された場合、そのストアド・プロシージャは COMMIT または ROLLBACK ステートメントを発行することはできません。

また、MODIFIES SQL DATA と定義されたストアド・プロシージャだけが、独自のセーブポイントを確立して、そのセーブポイント内の独自の作業をロールバックすることができます。他のタイプのルーチン (UDF とメソッド) は、独自のセーブポイントを確立できません。ストアド・プロシージャ内に作成されたセーブポイントは、そのストアド・プロシージャが完了しても解放されません。アプリケーションはそのセーブポイントをロールバックすることができます。同様に、ストアド・プロシージャも、アプリケーションで定義されたセーブポイントをロールバックすることができます。DB2 は、ルーチンによって確立されたすべてのセーブポイントを戻す時に暗黙で解放します。

ルーチンは、DB2 から渡された sqlstate 引数に SQLSTATE 値を割り当てることで、正常に完了したかどうかを DB2 に知らせることができます。一部のパラメーター・スタイル (PARAMETER STYLEs JAVA、GENERAL、および GENERAL WITH NULLS) は、SQLSTATE 値の交換をサポートしていません。

ルーチンによって発行された SQL の取り扱いで DB2 にエラーが発生した場合、他のどのアプリケーションに対しても同様に、そのエラーはルーチンに戻されます。通常のユーザー・エラーの場合、ルーチンは選択に応じて代替りのアクションまたは訂正アクションをとることができます。例えば、ルーチンが表への挿入を試みたときに、重複キー・エラー (SQLCODE-813) が戻された場合、選択を行って代わりにその表の既存行を更新することができます。

ただし、DB2 が通常のやり方で先に進むのを妨げるようなもっと重大なエラーが生じることもあります。例えば、デッドロック、データベース・パーティションの障害、またはユーザー割り込みなどがその一例です。このようなエラーの一部は、呼

び出し元のアプリケーションまで伝搬されます。作業単位に関連したその他の重大エラーは、トランザクション制御ステートメント (COMMIT または ROLLBACK) の発行を許可された (a) アプリケーションまたは (b) ストアード・プロシージャのうちの、バックアウトでどちらか先に発生したほうまで到達します。

このようなエラーのいずれかが、ルーチンから発行された SQL の実行中に起きた場合、エラーはルーチンに戻されますが、重大エラーが起きたことが DB2 に記憶されます。その場合はさらに、そのルーチンおよびすべての呼び出しルーチンからそれ以後に発行されたすべての SQL は DB2 によって自動的に失敗させられます (SQLCODE -20139、SQLSTATE 51038)。これに対する唯一の例外は、トランザクション制御ステートメントの発行を許可されている最も外側のストアード・プロシージャにまでしかエラーがバックアウトされない場合です。その場合、そのストアード・プロシージャは SQL を引き続き発行することができます。

ルーチンは静的および動的の両方の SQL を発行することができますが、どちらの場合も、組み込み SQL を使用するのであればその SQL をプリコンパイルしてバインドする必要があります。静的 SQL の場合にプリコンパイル/バインドのプロセスで使用される情報は、組み込み SQL を使用する他のすべてのクライアント・アプリケーションの場合と同じです。動的 SQL の場合は、DYNAMICRULES プリコンパイル/ BIND オプションを使用して、組み込み動的 SQL の現在のスキーマと現在の認証 ID を制御することができます。この動作は、ルーチンとアプリケーションとは異なります。

ルーチンのパッケージまたはステートメントに対して定義されている分離レベルが順守されます。それに応じて、ルーチンが実行される分離レベルは、呼び出し元のアプリケーションよりも厳密にも寛容にもなります。このことには、呼び出し元のアプリケーションよりも厳密さの低い分離レベルをもつルーチンを呼び出すときには配慮することが大切です。例えば、反復可能読み取りアプリケーションからカーソル固定関数を呼び出した場合、UDF は非反復可能読み取り特性を示すことがあります。

呼び出し側のアプリケーションまたはルーチンは、特殊レジスター値に対してルーチンが加えた変更によって影響を受けることはありません。更新可能な特殊レジスターは、呼び出し側からルーチンへと継承されます。更新可能な特殊レジスターに加えられた変更は、呼び出し側には戻されません。更新不能の特殊レジスターには、独自のデフォルト値が与えられます。更新可能および更新不能の特殊レジスターの詳細は、「特殊レジスター」という関連項を参照してください。

ルーチンは、クライアント・アプリケーションと同じやり方でカーソルの OPEN、FETCH、および CLOSE を行うことができます。同じ関数を複数回呼び出す (再帰の場合など) と、そのつど独自のカーソル・インスタンスが与えられます。UDF とメソッドは、ステートメント呼び出しの完了前にカーソルをクローズする必要があります。そうしないと、エラーが起きます (SQLCODE -472、SQLSTATE 24517)。UDF またはメソッドを最後に呼び出したときに、オープンしたままになっているすべてのカーソルをクローズするのがよいと思われます。オープンしたままのカーソルは、ストアード・プロシージャの完了の前にクローズされないと、クライアント・アプリケーションまたは呼び出し側ルーチンに結果セットとして戻されます。

ルーチンに渡された引数が、自動的にホスト変数として扱われることはありません。つまり、ルーチンが SQL 内でホスト変数としてパラメーターを使用するには、独自のホスト変数を宣言して、パラメーター値をそのホスト変数にコピーする必要がありますということです。

注: 組み込み SQL ルーチンの場合、DATETIME オプションを ISO に設定したうえでプリコンパイルしてバインドする必要があります。

外部ルーチンのパラメーター・スタイル

外部ルーチンのインプリメンテーションは、ルーチンのパラメーター値の交換のための特定の規則に準拠していなければなりません。そのような規則をパラメーター・スタイル と呼びます。外部ルーチンのパラメーター・スタイルは、PARAMETER STYLE 節を指定してルーチンが作成されると指定されます。パラメーター・スタイルは、パラメーター値が外部ルーチンのインプリメンテーションに渡される仕様および順序を示します。さらに、パラメーター・スタイルは、追加の値が外部ルーチンのインプリメンテーションに渡されたときに取られる動作も指定します。例えば、パラメーター・スタイルの中には、各ルーチンのパラメーター値ごとに、追加された個々の NULL 標識値がルーチンのインプリメンテーションに渡されることを指定するものもあります。これにより、パラメーターの NULL 可能性に関する情報が提供されます (これがないと、ネイティブのプログラミング言語のデータ・タイプで判別することは容易ではありません)。

以下の表では、使用可能なパラメーター・スタイル、各パラメーター・スタイルをサポートするルーチンのインプリメンテーション、各パラメーター・スタイルをサポートするルーチンの機能タイプ、およびパラメーター・スタイルの説明のリストを記載しています。

表 8. パラメーター・スタイル

パラメーター・スタイル	サポートされる言語	サポートされるルーチン・タイプ	説明
SQL ¹	<ul style="list-style-type: none"> • C/C++ • OLE • .NET 共通言語ランタイム言語 • COBOL ² 	<ul style="list-style-type: none"> • UDF • ストアド・プロシージャ • メソッド 	<p>呼び出し時に渡されるパラメーターに加えて、以下の引数が以下に示されている順序でルーチンに渡されます。</p> <ul style="list-style-type: none"> • CREATE ステートメント内で宣言された各パラメーターまたは結果ごとの NULL 標識。 • DB2 に戻される SQLSTATE。 • ルーチンの修飾名。 • 個々のルーチン名。 • DB2 に戻される SQL 診断ストリング。 <p>CREATE ステートメントとルーチン・タイプに指定されているオプションに応じて、以下の引数を以下に示されている順序でルーチンに渡すことができます。</p> <ul style="list-style-type: none"> • スクラッチパッドのバッファー。 • ルーチンの呼び出しタイプ。 • dbinfo 構造 (データベースに関する情報が入っています)。

表 8. パラメーター・スタイル (続き)

パラメーター・スタイル	サポートされる言語	サポートされるルーチン・タイプ	説明
DB2SQL ¹	<ul style="list-style-type: none"> • C/C++ • OLE • .NET 共通言語ランタイム言語 • COBOL 	<ul style="list-style-type: none"> • ストアド・プロシージャ 	<p>呼び出し時に渡されるパラメーターに加えて、以下の引数が以下に示されている順序でストアド・プロシージャに渡されます。</p> <ul style="list-style-type: none"> • CALL ステートメント上の各パラメーターごとの NULL 標識の入ったベクトル。 • DB2 に戻される SQLSTATE。 • ストアド・プロシージャの修飾名。 • 個々のストアド・プロシージャ名。 • DB2 に戻される SQL 診断ストリング。 <p>CREATE PROCEDURE ステートメント内で DBINFO 節を指定すると、dbinfo 構造 (データベースに関する情報が入っています) がストアド・プロシージャに渡されます。</p>
JAVA	<ul style="list-style-type: none"> • Java 	<ul style="list-style-type: none"> • UDF • ストアド・プロシージャ 	<p>PARAMETER STYLE JAVA ルーチンは、Java 言語と SQLJ ルーチンの仕様に準拠したパラメーター引き渡し規則に従います。</p> <p>ストアド・プロシージャの場合には INOUT および OUT パラメーターは、値を戻しやすくするために単一の項目配列として渡されます。ストアド・プロシージャ用の Java メソッド・シグニチャーには、IN、OUT、および INOUT パラメーターのほかに、CREATE PROCEDURE ステートメントの DYNAMIC RESULT SETS 節に指定されている各結果セットごとにタイプ ResultSet[] のパラメーターが組み込まれています。</p> <p>PARAMETER STYLE JAVA の UDF とメソッドの場合、ルーチンの呼び出しに指定されたもの以外の追加引数は渡されません。</p> <p>PARAMETER STYLE JAVA ルーチンは、DBINFO または PROGRAM TYPE 節をサポートしていません。UDF の場合、PARAMETER STYLE JAVA を指定できるのは、パラメーターとして構造化データ・タイプを指定しておらず、かつ戻りタイプとして構造化タイプ、CLOB、DBCLOB、または BLOB データ・タイプを指定していない場合だけです (SQLSTATE 429B8)。また、PARAMETER STYLE JAVA UDF は表関数、呼び出しタイプ、またはスクラッチパッドをサポートしていません。</p>
DB2GENERAL	<ul style="list-style-type: none"> • Java 	<ul style="list-style-type: none"> • UDF • ストアド・プロシージャ • メソッド 	<p>このタイプのルーチンは、Java メソッドで使用するよう定義されたパラメーター引き渡し規則に従います。表 UDF やスクラッチパッド付きの UDF を開発したり、dbinfo 構造にアクセスする必要があったりしない限り、PARAMETER STYLE JAVA を使用することをお勧めします。</p> <p>PARAMETER STYLE DB2GENERAL ルーチンの場合、ルーチンの呼び出しに指定されたもの以外の追加引数は渡されません。</p>

表 8. パラメーター・スタイル (続き)

パラメーター・スタイル	サポートされる言語	サポートされるルーチン・タイプ	説明
GENERAL	<ul style="list-style-type: none"> • C/C++ • .NET 共通言語ランタイム言語 • COBOL 	<ul style="list-style-type: none"> • ストアド・プロシージャー 	<p>PARAMETER STYLE GENERAL ストアド・プロシージャーは、呼び出し元のアプリケーションまたはルーチン内の CALL ステートメントからパラメーターを受け取ります。CREATE PROCEDURE ステートメント内で DBINFO 節を指定すると、dbinfo 構造 (データベースに関する情報が入っています) がストアド・プロシージャーに渡されます。</p> <p>GENERAL は、DB2 for z/OS の SIMPLE ストアド・プロシージャーと同等です。</p>
GENERAL WITH NULLS	<ul style="list-style-type: none"> • C/C++ • .NET 共通言語ランタイム言語 • COBOL 	<ul style="list-style-type: none"> • ストアド・プロシージャー 	<p>PARAMETER STYLE GENERAL WITH NULLS ストアド・プロシージャーは、呼び出し元のアプリケーションまたはルーチン内の CALL ステートメントからパラメーターを受け取ります。CALL ステートメント上の各パラメーターごとの NULL 標識の入ったベクトルもその中に含まれます。CREATE PROCEDURE ステートメント内で DBINFO 節を指定すると、dbinfo 構造 (データベースに関する情報が入っています) がストアド・プロシージャーに渡されます。</p> <p>GENERAL WITH NULLS は、DB2 for z/OS の SIMPLE WITH NULLS ストアド・プロシージャーと同等です。</p>

注:

1. UDF およびメソッドの場合、PARAMETER STYLE SQL は PARAMETER STYLE DB2SQL と同等です。
2. COBOL を使用できるのは、ストアド・プロシージャーの開発でのみです。
3. .NET 共通言語ランタイム・メソッドはサポートされていません。

PROGRAM TYPE MAIN または PROGRAM TYPE SUB プロシージャーでのパラメーター処理

プロシージャーは、メインルーチンまたはサブルーチンのスタイルのパラメーターを受け入れることができます。それは、CREATE PROCEDURE ステートメントを使用してプロシージャーを登録するときには判別されます。

PROGRAM TYPE SUB の C または C++ プロシージャーは、C または C++ サブルーチンと同じやり方で引数を受け入れます。パラメーターをポインターとして受け渡します。例えば、次のような C プロシージャーのシグニチャーは、INTEGER、SMALLINT、および CHAR(3) タイプのパラメーターを受け入れます。

```
int storproc (sqlint32 *arg1, sqlint16 *arg2, char *arg3)
```

Java のプロシージャーは、サブルーチンとしてしか引数を受け入れません。IN パラメーターは単純な引数として渡します。OUT および INOUT パラメーターは、単一エレメントの配列として渡します。次のようなパラメーター・スタイルの Java のプロシージャーのシグニチャーは、INTEGER タイプの IN パラメーター、SMALLINT タイプの OUT パラメーター、および CHAR(3) タイプの INOUT パラメーターを受け入れます。

```
int storproc (int arg1, short arg2[], String arg[])
```

C プログラムの `main` 関数のような引数を受け入れるように C プロシージャを作成するには、`CREATE PROCEDURE` ステートメントでプログラム・タイプに `MAIN` を指定します。プログラム・タイプが `MAIN` のプロシージャを作成する場合は、以下の仕様に準拠していなければなりません。

- プロシージャは以下の 2 つの引数によってパラメーターを受け入れる。
 - パラメーター・カウンター変数。例えば、`argc`。
 - パラメーターを指すポインターの配列。例えば、`char **argv`。
- プロシージャは共用ライブラリーとして作成しなければならない。

`PROGRAM TYPE MAIN` プロシージャでは、DB2 は `argv` 配列の最初のエレメントの値 (`argv[0]`) をプロシージャの名前に設定します。`argv` 配列の残りのエレメントは、プロシージャの `PARAMETER STYLE` ステートメントで定義されるパラメーターに対応します。例えば、次のような組み込み型の C プロシージャは、`argv[1]` という 1 つの `IN` パラメーターを渡し、`argv[2]` および `argv[3]` という 2 つの `OUT` パラメーターを戻します。

`PROGRAM TYPE MAIN` の例の `CREATE PROCEDURE` ステートメントは、次のようになります。

```
CREATE PROCEDURE MAIN_EXAMPLE (IN job CHAR(8),
    OUT salary DOUBLE, OUT errorcode INTEGER)
    DYNAMIC RESULT SETS 0
    LANGUAGE C
    PARAMETER STYLE GENERAL
    NO DBINFO
    FENCED
    READS SQL DATA
    PROGRAM TYPE MAIN
    EXTERNAL NAME 'spserver!mainexample'
```

次に示すプロシージャのコード例では、`argv[1]` の値を `CHAR(8)` のホスト変数 `injob` にコピーしてから、`DOUBLE` ホスト変数の値 `outsalary` を `argv[2]` にコピーし、`SQLCODE` を `argv[3]` として戻します。

```
SQL_API_RC SQL_API_FN main_example (int argc, char **argv)
{
    EXEC SQL INCLUDE SQLCA;

    EXEC SQL BEGIN DECLARE SECTION;
        char injob[9];
        double outsalary;
    EXEC SQL END DECLARE SECTION;

    /* argv[0] contains the procedure name. */
    /* Parameters start at argv[1]          */
    strcpy (injob, (char *)argv[1]);

    EXEC SQL SELECT AVG(salary)
        INTO :outsalary
        FROM employee
        WHERE job = :injob;

    memcpy ((double *)argv[2], (double *)&outsalary, sizeof(double));

    memcpy ((sqlint32 *)argv[3], (sqlint32 *)&SQLCODE, sizeof(sqlint32));
}
```

```
return (0);  
} /* end main_example function */
```

サポートされているルーチン・プログラミング言語

一般的に、データベース・サーバー上でアプリケーションの機能性を発揮できるようにすることで、データベース管理システムの全体的なパフォーマンスを改善するために、ルーチンが使用されます。このような努力により報われる効果も、ルーチンの作成時に選択する言語によってはその成果はある程度限定されてしまいます。

特定の言語でルーチンをインプリメントする際に、事前に検討する必要のあるいくつかの注意点を以下に示します。

- 特定の言語と環境においてルーチンを開発するのに利用できるスキル。
- 言語のインプリメント・コードの信頼性と安全性。
- 特定の言語で作成されたルーチンのスケーラビリティ。

上記の基準を判断するための参考として、以下にサポートされている各種言語の特性を示します。

SQL

- SQL ルーチンは、Java ルーチンよりも高速であり、NOT FENCED C/C++ ルーチンとパフォーマンス的にはほぼ同等です。
- SQL ルーチンは、完全に SQL で記述するものであり、SQL プロシージャ型言語 (SQL PL) エレメントを組み込むこともできます。SQL PL には、ロジックのインプリメントに使用できる SQL 制御ステートメントが含まれます。
- SQL ルーチンは、一部始終 SQL ステートメントで構成されるので DB2 にとって「安全」とみなされます。SQL ルーチンは常にデータベース・エンジン内で直接実行されますが、それによって望ましいパフォーマンスとスケーラビリティが実現されます。

C/C++

- C/C++ 組み込み SQL と DB2 CLI ルーチンはどちらも、Java ルーチンよりも高速です。これらの NOT FENCED モードでの稼働時のパフォーマンスは SQL ルーチンとほぼ同等です。
- C/C++ ルーチンはエラーを生じやすいので、C/C++ のルーチンは、FENCED NOT THREADSAFE として登録することをお勧めします。これらの言語のルーチンは、メモリー破壊によって DB2 のデータベース・エンジンの機能に支障をきたす可能性が最も高いからです。ただし FENCED NOT THREADSAFE モードでの実行は安全ではあっても、パフォーマンスにオーバーヘッドがかかります。

C/C++ ルーチンを NOT FENCED または FENCED THREADSAFE と登録する場合のリスクの評定とその緩和に関する詳細は、『ルーチンのセキュリティに関する考慮事項』の項を参照してください。

- デフォルトでは C/C++ ルーチンは FENCED NOT THREADSAFE モードで実行されて、他のルーチンの実行を妨害しないように分離されます。

というわけで、データベース・サーバー上の並行実行中の C/C++ ルーチンあたりの db2fmp プロセスは 1 つになります。そのため、システムによってはスケーラビリティ上の問題が生じることがあります。

Java

- Java ルーチンは C/C++ または SQL ルーチンよりも遅いです。
- Java ルーチンでは危険操作の制御は JVM によって担われるので、Java ルーチンのほうが C/C++ ルーチンよりも安全です。それによって、信頼性が増大します。Java ルーチンが、同じプロセス内で稼働している別のルーチンに被害を与える可能性は低いからです。

注: 危険性を含んだ操作を避けるために、Java ルーチンから Java Native Interface (JNI) 呼び出しを行うことはできないことになっています。Java ルーチンから C/C++ コードを呼び出す必要がある場合は、別個にカタログされた C/C++ ルーチンを呼び出さなければなりません。

- FENCED THREADSAFE モード (デフォルト) での実行時には、Java ルーチンはスケーラビリティに優れています。FENCED の Java ルーチンはすべて、新規のいくつかの JVM を共用します (特定の db2fmp プロセスの Java ヒープが使い果たされると、システムでは複数の JVM が起用されるからです)。
- NOT FENCED Java ルーチンは現在サポートされていません。NOT FENCED として定義される Java ルーチンは、FENCED THREADSAFE として定義されているかのように呼び出されます。

.NET 共通言語ランタイム言語

- .NET 共通言語ランタイム (CLR) ルーチンは、.NET Framework の CLR で解釈できる中間言語 (IL) バイト・コードにコンパイルされるルーチンです。CLR ルーチンのソース・コードは、.NET Framework でサポートされているどの言語でも記述できます。
- .NET CLR ルーチンをコーディングするユーザーは、.NET CLR でサポートされているプログラム言語の中から好みの言語をどれでも選択できます。
- CLR アセンブリーは、別の .NET プログラム言語のソース・コードからコンパイルしたサブアセンブリーからでもビルドできます。つまり、ユーザーとしては、さまざまな言語で作成したコード・モジュールの再利用と統合が可能になります。
- CLR ルーチンは、FENCED NOT THREADSAFE ルーチンとしてのみ作成できます。したがって、エンジンの損壊の可能性は最小化されますが、NOT FENCED ルーチンで得られるパフォーマンス向上のメリットを生かすことはできません。

OLE

- OLE ルーチンは、Visual C++、Visual Basic、および OLE でサポートされているその他の言語でインプリメントすることができます。
- OLE 自動化ルーチンの速度は、インプリメントに使用する言語によって異なります。一般的にこのルーチンは、OLE C/C++ 以外のルーチンよりも遅いです。

- OLE ルーチンは、FENCED NOT THREADSAFE モードでのみ実行することができます。それによって、エンジンの損壊の可能性が最小化されます。またこれは、OLE オートメーション・ルーチンはスケーラビリティにあまり優れていないことも意味します。

OLE DB

- OLE DB は、表関数の定義にのみ使用できます。
- OLE DB 表関数は、外部の OLE DB データ・ソースに接続します。
- OLE DB Provider によっては OLE DB 表関数は概して Java 表関数よりも高速ですが、C/C++ 表関数や SQL を本体とする表関数よりも遅いです。ただし関数が呼び出される場所である照会内の特定の述部を OLE DB Provider で評価することができるので、DB2 が処理しなければならない行数は減ります。多くの場合、それによってパフォーマンスが向上することになります。
- OLE DB ルーチンは、FENCED NOT THREADSAFE モードでのみ実行することができます。それによって、エンジンの損壊の可能性が最小化されます。またこれは、OLE DB オートメーション表関数はスケーラビリティにあまり優れていないことも意味します。

外部ルーチンの開発でサポートされている API とプログラミング言語の比較

外部ルーチンのインプリメントを開始する前に、サポートされているさまざまな外部ルーチンのアプリケーション・プログラミング・インターフェース (API) およびプログラミング言語の特性および制限について考慮することは重要です。これによって最初から正しいインプリメンテーションを選ぶことができ、必要なルーチンのフィーチャーを使用することができます。

表9. 外部ルーチン API とプログラミング言語の比較

API とプログラミング言語	フィーチャー・サポート	パフォーマンス	セキュリティ	スケーラビリティ	制限
SQL (SQL PL を含む)	<ul style="list-style-type: none"> SQL は、容易に学習および使用できるハイレベル言語であり、インプリメンテーションを素早く実行できるようにします。 SQL プロシージャ型言語 (SQL PL) エレメントは、SQL 操作および照会において制御フロー・ロジックを許可します。 	<ul style="list-style-type: none"> 非常に良い。 SQL ルーチンは、Java ルーチンよりも高いパフォーマンスを実現します。 SQL ルーチンのパフォーマンスは、NOT FENCED 節で作成される C および C++ 外部ルーチンと同程度です。 	<ul style="list-style-type: none"> 非常に安全。 SQL プロシージャは、常にデータベース・マネージャーと同じメモリで稼働します。これは、キーワード NOT FENCED を使ってデフォルトで作成されるルーチンに対応します。 	<ul style="list-style-type: none"> 高いスケーラビリティ。 	<ul style="list-style-type: none"> データベース・サーバーのファイル・システムにアクセスできません。 データベースの外部に常駐するアプリケーションを呼び出すことはできません。

表9. 外部ルーチン API とプログラミング言語の比較 (続き)

API とプログラミング言語	フィーチャー・サポート	パフォーマンス	セキュリティ	スケーラビリティ	制限
<p>組み込み SQL (C および C++ を含む)</p>	<ul style="list-style-type: none"> 低いレベルではあるが、強力なプログラミング言語。 	<ul style="list-style-type: none"> 非常に良い。 C および C++ ルーチンは、Java ルーチンよりも高いパフォーマンスを実現します。 NOT FENCED 節で作成される C および C++ 外部ルーチンのパフォーマンスは、SQL ルーチンと同程度です。 	<ul style="list-style-type: none"> C および C++ ルーチンではプログラミング・エラーが生じやすくなります。 プログラマーは、ルーチンのインプリメンテーションを単調にして時間を浪費させる共通メモリーおよびポインター操作のエラーを避けるため、C に熟達している必要があります。 C および C++ ルーチンは、データベース・マネージャーの混乱によって実行時にルーチン内で例外が発生するのを避けるため、FENCED 節および NOT THREADSAFE 節を使用して作成しなければなりません。これらはデフォルトの節です。これらの節を使用することで、パフォーマンスにいくらか悪い影響が出る場合がありますが、安全な実行が保証されます。ルーチンのセキュリティを参照してください。 	<ul style="list-style-type: none"> C および C++ ルーチンが FENCED 節および NOT THREADSAFE 節で作成されている場合、スケーラビリティは削減されます。これらのルーチンは、データベース・マネージャー・プロセスとは別に、分離した <i>db2fmp</i> プロセス内で実行されます。<i>db2fmp</i> プロセスは、並行して実行されるルーチンごとに必要になります。 	<ul style="list-style-type: none"> サポートされるパラメーター引き渡しスタイルは複数存在するので、混乱してしまう可能性があります。ユーザーは、可能な限りパラメーター・スタイル SQL を使用する必要があります。

表9. 外部ルーチン API とプログラミング言語の比較 (続き)

API とプログラミング言語	フィーチャー・サポート	パフォーマンス	セキュリティ	スケーラビリティ	制限
組み込み SQL (COBOL)	<ul style="list-style-type: none"> • ビジネス (通常はファイルを取り扱う) アプリケーションの開発に適した高水準プログラミング言語。 • これまでは実動ビジネス・アプリケーション用に広く使用されてきました。しかし、その普及度は減少しています。 • COBOL は、ポインター、および、再帰的な呼び出しをサポートしていないプログラミング言語です。 	<ul style="list-style-type: none"> • COBOL ルーチンは、他の外部ルーチンのインプリメンテーション・オプションを指定して作成されたルーチンと比較してパフォーマンスが劣ります。 	<ul style="list-style-type: none"> • 現時点で情報はありません。 	<ul style="list-style-type: none"> • 現時点で情報はありません。 	<ul style="list-style-type: none"> • 32 ビットの COBOL プロシージャを 64 ビットの DB2 インスタンス内に作成して呼び出すことができますが、これらのルーチンは 64 ビットの DB2 インスタンス内にある 64 ビットの COBOL プロシージャと比較してパフォーマンスが劣ります。
JDBC (Java) および SQLJ (Java)	<ul style="list-style-type: none"> • スタンドアロン・アプリケーション、アプレット、およびサーブレットの開発に適した、高水準のオブジェクト指向プログラミング言語。 • Java オブジェクトおよびデータ・タイプは、データベース接続の確立、SQL ステートメントの実行、およびデータの操作を容易にします。 	<ul style="list-style-type: none"> • Java ルーチンは、C および C++ ルーチンまたは SQL ルーチンと比較してパフォーマンスが劣ります。 	<ul style="list-style-type: none"> • Java ルーチンでは、危険操作の制御は Java 仮想マシン (JVM) によって担われるので、Java ルーチンのほうが C および C++ ルーチンよりも安全です。これにより、信頼性は向上し、1 つの Java ルーチンのコードが、同じプロセス内で実行中の別のルーチンに悪影響を与えることはほとんどありません。 	<ul style="list-style-type: none"> • 優れたスケーラビリティ • FENCED THREADSAFE 節で作成された Java ルーチン (デフォルト) は、スケーラビリティに優れています。fenced の Java ルーチンはすべて、いくつかの JVM を共有します。特定の db2fmp プロセスの Java ヒープが使い果たされると、システムでは複数の JVM が起用されるからです。 	<ul style="list-style-type: none"> • 危険性を含んだ操作を避けるために、Java ルーチンから Java Native Interface (JNI) 呼び出しを行うことはできないことになっています。

表9. 外部ルーチン API とプログラミング言語の比較 (続き)

API とプログラミング言語	フィーチャー・サポート	パフォーマンス	セキュリティ	スケーラビリティ	制限
.NET 共通言語ランタイムのサポートされる言語 (C#, Visual Basic、およびその他を含む)	<ul style="list-style-type: none"> 管理対象コードの Microsoft .NET モデルの一部です。 ソース・コードは Microsoft .NET Framework 共通言語ランタイムで解釈できる中間言語 (IL) バイト・コードにコンパイルされます。 CLR アセンブリは、別の .NET プログラム言語のソース・コードからコンパイルしたサブアセンブリからでもビルドできます。つまり、ユーザーとしては、さまざまな言語で作成したコード・モジュールの再利用と統合が可能になります。 	<ul style="list-style-type: none"> CLR ルーチンは、実行時にデータベース・マネージャーの割り込みの可能性を最小限に抑えるために、FENCED NOT THREADSAFE 節でのみ作成できます。これにより、パフォーマンスにいくらか悪影響が出る可能性があります。 デフォルト節値を使用するならば、実行時にデータベース・マネージャーによる割り込みが起きる可能性は小さくなりますが、CLR ルーチンは FENCED として実行する必要があるため、NOT FENCED として指定できる他の外部ルーチンと比べるとわずかに実行速度が遅くなる可能性があります。 	<ul style="list-style-type: none"> CLR ルーチンは、FENCED NOT THREADSAFE 節でのみ作成できます。このルーチンはデータベース・マネージャーの外の、別個の db2fmp プロセスで実行されるので、安全性が確保されます。 	<ul style="list-style-type: none"> 入手可能な情報ははありません。 	<ul style="list-style-type: none"> 『.NET CLR ルーチンに関する制約事項』のトピックを参照してください。

表9. 外部ルーチン API とプログラミング言語の比較 (続き)

API とプログラミング言語	フィーチャー・サポート	パフォーマンス	セキュリティ	スケーラビリティ	制限
<ul style="list-style-type: none"> OLE 	<ul style="list-style-type: none"> OLE ルーチンは、Visual C++、Visual Basic、および OLE でサポートされているその他の言語でインプリメントすることができます。 	<ul style="list-style-type: none"> OLE 自動化ルーチンの速度は、インプリメントに使用する言語によって異なります。一般的にこのルーチンは、OLE C/C++ 以外のルーチンよりも遅いです。 OLE ルーチンは、FENCED NOT THREADSAFE モードでのみ実行できるため、OLE 自動化ルーチンはスケーラビリティにはあまり優れていません。 	<ul style="list-style-type: none"> 入手可能な情報はあります。 	<ul style="list-style-type: none"> 入手可能な情報はあります。 	<ul style="list-style-type: none"> 入手可能な情報はあります。

表9. 外部ルーチン API とプログラミング言語の比較 (続き)

API とプログラミング言語	フィーチャー・サポート	パフォーマンス	セキュリティ	スケーラビリティ	制限
<ul style="list-style-type: none"> OLE DB 	<ul style="list-style-type: none"> OLE DB は、ユーザー定義の表関数の作成に使用できます。 OLE DB 関数は、外部の OLE DB データ・ソースに接続します。 	<ul style="list-style-type: none"> OLE DB 関数のパフォーマンスは、OLE DB Provider によって異なります。ただし、一般に OLE DB 関数は、論理的に同等な Java 関数よりパフォーマンスが優れていますが、論理的に同等な C、C++、または SQL 関数より速度が遅くなります。ただし関数が呼び出される場所である照会内の特定の述部を OLE DB Provider で評価することができるので、DB2 が処理しなければならない行数は減ります。多くの場合、それによってパフォーマンスが向上することになります。 	<ul style="list-style-type: none"> 入手可能な情報はありません。 	<ul style="list-style-type: none"> 入手可能な情報はありません。 	<ul style="list-style-type: none"> OLE DB は、ユーザー定義の表関数の作成にのみ使用できます。

ルーチン開発に関するパフォーマンスの考慮

クライアント・アプリケーションを拡張する代わりにルーチンを開発することの大きな利点の 1 つは、パフォーマンスです。ルーチンのインプリメンテーション用のアプローチを選択するときには、次のようなパフォーマンス上の影響に配慮してください。

NOT FENCED モード

NOT FENCED ルーチンは、データベース・マネージャーと同じプロセス中で稼働します。一般的に、ルーチンを NOT FENCED で実行したほうが、FENCED モードで実行する場合よりもパフォーマンスが改良されます。FENCED ルーチンは、エンジンのアドレス・スペース外部の特別な DB2 プロセスで実行されるからです。

ルーチンを NOT FENCED モードで実行すればルーチンのパフォーマンスの向上は期待できますが、ユーザー・コードによってデータベースやデータベース制御構造が無意識または意識的に破壊される可能性があります。NOT FENCED ルーチンを使用してよいのは、パフォーマンス上の利点を最大化する必要がある場合と、ルーチンをセキュア化する必要がある場合だけです。(C/C++ ルーチンを NOT FENCED として登録する場合のリスクの評価とその緩和に関する詳細は、『ルーチンのセキュリティに関する考慮事項』の項を参照してください。) データベース・マネージャーのプロセスで実行するにはルーチンが十分安全でない場合、ルーチンの登録時に FENCED 節を使用します。安全でない可能性があるコードの作成および実行を制限するために、DB2 では、ユーザーが NOT FENCED ルーチンを作成するには、特殊権限 CREATE_NOT_FENCED_ROUTINE を持っていないければなりません。

NOT FENCED ルーチンの実行中に異常終了が発生する場合、ルーチンが NO SQL として登録されていると、データベース・マネージャーは適切なリカバリーを試行します。しかし、NO SQL として定義されていないルーチンの場合、データベース・マネージャーは失敗します。

ルーチンが GRAPHIC または DBCLOB データを使用する場合は、NOT FENCED ルーチンを WCHARTYPE NOCONVERT オプションでプリコンパイルする必要があります。

FENCED THREADSAFE モード

FENCED THREADSAFE ルーチンは、他のルーチンと同じプロセスで稼働します。具体的には、Java 以外のルーチンはあるプロセスを共用し、Java ルーチンは他の言語で作成されたルーチンとは分離した、別のプロセスを共用します。この分離により、Java ルーチンは、他の言語で作成された、エラーを起こしやすいルーチンから保護されます。また、Java ルーチンのプロセスには JVM が含まれています。これは、メモリー・コストが高くなり、他のルーチン・タイプでは使用されません。FENCED THREADSAFE ルーチンの複数の呼び出しではリソースを共用するため、それぞれが独自の専用プロセスで実行する FENCED NOT THREADSAFE ルーチンよりもシステムのオーバーヘッドが減ります。

ご使用のルーチンが他のルーチンと同じプロセスで実行しても安全であると感じる場合、それを登録する際に THREADSAFE 節を使用してください。NOT FENCED ルーチンと同様に、C/C++ ルーチンを FENCED THREADSAFE として登録するリスクの評価およびその軽減の詳細については、『ルーチンのセキュリティに関する考慮事項』のトピックを参照してください。

FENCED THREADSAFE ルーチンが異常終了した場合、このルーチンを実行していたスレッドだけが終了します。プロセス内のその他のルーチンは実行を続けます。ただし、このスレッドが異常終了する原因となった障害によって、プロセス中の他のルーチン・スレッドが悪影響を受けて、トラップ、ハング、またはデータ損傷を起こす可能性があります。あるスレッドが異常終了した後は、そのプロセスは新規のルーチンの呼び出しに使用されません。すべてのアクティブ・ユーザーがこのプロセスでジョブを完了すると、それは終了されます。

Java ルーチンを登録する際に、特に指定されない限り、THREADSAFE であると見なされます。その他の LANGUAGE タイプはすべて、デフォルトで NOT THREADSAFE です。LANGUAGE OLE および OLE DB を使用するルーチンは THREADSAFE として指定できません。

NOT FENCED ルーチンは THREADSAFE でなければなりません。ルーチンを NOT FENCED NOT THREADSAFE として登録することはできません (SQLCODE -104)。

UNIX のユーザーは、db2fmp (Java) または db2fmp (C) を探すことにより、Java および C の THREADSAFE プロセスを参照できます。

FENCED NOT THREADSAFE モード

FENCED NOT THREADSAFE ルーチンはそれぞれ、独自の専用プロセスで実行します。多数のルーチンを実行している場合、このことはデータベース・システムのパフォーマンスに悪影響を及ぼす可能性があります。ルーチンが他のルーチンと同じプロセスで実行できるほど安全でない場合は、ルーチンを登録する際に NOT THREADSAFE 節を使用してください。

UNIX では、NOT THREADSAFE プロセスは db2fmp (pid) (pid は fenced モード・プロセスを使用するエージェントのプロセス ID) またはプール NOT THREADSAFE db2fmp の場合は db2fmp (idle) として表示されます。

Java ルーチン

メモリーの所要量の大きい Java ルーチンを実行する予定の場合、そのルーチンを FENCED NOT THREADSAFE と登録することをお勧めします。

FENCED THREADSAFE Java ルーチンの呼び出しの場合、DB2 は、ルーチンを十分に実行できる大きさの Java ヒープを持つ、スレッド化された Java fenced モードのプロセスを選択しようとしています。独自のプロセス内で大量のヒープを消費するプロセスを分離しないと、マルチスレッドの Java db2fmp プロセスで Java ヒープ不足エラーが生じる可能性があります。このカテゴリーに当てはまらない Java ルーチンの場合、少数の JVM を共用できるスレッド・セーフ・モードにしたほうが FENCED ルーチンの実行は向上します。

NOT FENCED Java ルーチンは現在サポートされていません。NOT FENCED として定義される Java ルーチンは、FENCED THREADSAFE として定義されているかのように呼び出されます。

C/C++ ルーチン

一般的に、C または C++ ルーチンのほうが Java ルーチンよりも高速ですが、エラー、メモリーの破壊、および破損の可能性が高くなります。というわけで、メモリー操作を実行する機能のために、THREADSAFE または NOT FENCED モードの登録では C または C++ ルーチンはリスクの高い候補となります。そのようなリスクを軽減するには、セキュア・ルーチンのためのプログラミングの実践方法を順守 (『ルーチンのセキュリティーに関する考慮事項』の項を参照) して、ルーチンを徹底的にテストします。

SQL ルーチン

一般的に、SQL ルーチン、特に SQL プロシージャのほうが Java ルーチンよりもやはり高速であり、通常は C ルーチンに匹敵するパフォーマンスを備えています。SQL ルーチンは常に NOT FENCED モードで稼働するの

で、外部ルーチンよりもパフォーマンス上の利点はさらに大きくなります。複雑なロジックを組み込んだ UDF は、一般的に、SQL で作成するよりも C で作成するほうが実行速度が上がります。ロジックが単純な場合、SQL UDF は外部 UDF に匹敵します。

スクラッチパッド

スクラッチパッドとは、UDF およびメソッドへの割り当てが可能なメモリー・ブロックのことです。スクラッチパッドが適用されるのは、SQL ステートメント内のルーチンへの個々の参照に対してのみです。ステートメント内のルーチンに対して複数の参照がある場合、どの参照もそれ独自のスクラッチパッドをもつこととなります。スクラッチパッドを使用すれば、UDF またはメソッドは次の呼び出し時までその状態を保持しておくことができます。

複雑な初期化を伴う UDF およびメソッドの場合はスクラッチパッドを使用すれば、最初の呼び出しで必要であったすべての値を保管しておいて、以後のすべての呼び出しでそれを使用することができます。他の UDF やメソッドのロジックでも、呼び出しと呼び出しの間で中間値の保管が必要になる場合があります。

CHAR パラメーターにとって代わる VARCHAR パラメーターの使用

ルーチン定義で CHAR パラメーターの代わりに VARCHAR パラメーターを使用すれば、ルーチンのパフォーマンスを向上させることができます。CHAR データ・タイプではなく VARCHAR データ・タイプを使用すると、パラメーターの引き渡しの前に DB2 によってパラメーターにスペースが埋め込まれなくなります。これで、ネットワークを経由したパラメーターの転送に要する時間が短縮されます。

例えば、クライアント・アプリケーションが CHAR(200) パラメーターを予期するルーチンにストリング "A SHORT STRING" を渡す場合、DB2 はパラメーターに 186 個のスペースを埋め込み、ストリングを NULL で終了してから、200 文字のストリングと NULL 終止符全体をネットワーク経由でルーチンに送信する必要があります。

それと比べて、VARCHAR(200) パラメーターを予期するルーチンに同じストリング "A SHORT STRING" を渡すと、DB2 は単に 14 文字ストリングと NULL 終止符をネットワーク経由で渡します。

ルーチンのセキュリティーに関する考慮事項

ルーチンの開発および配置の作業は、データベース・アプリケーションのパフォーマンスと効率性を大幅に高める機会になります。ただし、データベース管理者がルーチンの配置を正しく管理しないと、セキュリティー上のリスクが生じる可能性もあります。ここでは、セキュリティー上のリスクについてとそのようなリスクを軽減するための手段について説明します。セキュリティー・リスクの後には、セキュリティーを確認されていないルーチンを安全に配置する方法に関する項が続いています。

セキュリティー・リスク

NOT FENCED ルーチンは、データベース・マネージャー・リソースにアクセスすることができます。

NOT FENCED ルーチンは、データベース・マネージャーと同じプロセスで稼働します。NOT FENCED ルーチンは、データベース・エンジンと密接なつながりをもっているため、データベース・マネージャーの共用メモリーを無意識または意識的に破壊したり、データベースの制御構造を損壊したりする可能性があります。どちらの支障の場合も、データベース・マネージャーが失敗することになります。また NOT FENCED ルーチンは、データベースとその表を破壊する可能性もあります。

データベース・マネージャーとそのデータベースの保全性を確保するには、NOT FENCED として登録する予定のルーチンを徹底的にスクリーニングする必要があります。そのようなルーチンは、全面的にテストおよびデバッグする必要がありますが、予測しきれない副次効果を示してはなりません。ルーチンの検査では、メモリー管理と静的変数の使用に対して厳重な注意を払います。破損が生じる可能性が最も高いと言えるのは、コードがメモリーを適切に管理しない場合や静的変数を不正に使用する場合です。このような問題は、Java や .NET 以外のプログラミング言語でよく見られます。

NOT FENCED ルーチンを登録するには、**CREATE_NOT_FENCED_ROUTINE** 権限が必要です。

CREATE_NOT_FENCED_ROUTINE 権限を付与する場合、付与された人はデータベース・マネージャーとそのすべてのリソースに無制限にアクセスできるようになることに注意してください。

注: NOT FENCED ルーチンは、共通の基準に準拠した構成ではサポートされていません。

FENCED THREADSAFE ルーチンは、他の **FENCED THREADSAFE** ルーチン内のメモリーにアクセスできます。

FENCED THREADSAFE ルーチンは、共用プロセス内のスレッドとして稼働します。このルーチンはいずれも、同一プロセス内の他のルーチン・スレッドによって使用されるメモリーを読み取ることができます。したがって、1 つのスレッド化ルーチンがスレッド化プロセス中の他のルーチンから機密データを収集することが可能になります。1 つのプロセスを共用している場合、メモリー管理が徹底していないルーチン・スレッドが他のルーチン・スレッドを破壊したり、スレッド化プロセス全体が破損する原因になったりするという別のリスクも付随します。

他の FENCED THREADSAFE ルーチンの保全性を確保するには、FENCED THREADSAFE として登録する予定のルーチンを徹底的にスクリーニングする必要があります。そのようなルーチンは、全面的にテストおよびデバッグする必要がありますが、予測しきれない副次効果を示してはなりません。ルーチンの検査では、メモリー管理と静的変数の使用に対して厳重な注意を払います。それは、破損が起きる可能性の最も高い箇所であるからです。Java 以外の言語の場合は特にそうです。

FENCED THREADSAFE ルーチンを登録するには、**CREATE_EXTERNAL_ROUTINE** 権限が必要です。

CREATE_EXTERNAL_ROUTINE 権限を付与する場合、付与された人は他の FENCED THREADSAFE ルーチンのメモリーをモニターまたは破壊できるようにすることに注意してください。

fenced プロセスの所有者からのデータベース・サーバーへの書き込みアクセス権限は、データベース・マネージャーの破壊を起こす可能性があります。

fenced プロセスの実行に使用されるユーザー ID は、db2icrt (インスタンスの作成) または db2iupdt (インスタンスの更新) システム・コマンドで定義されます。そのユーザー ID は、ルーチンのライブラリーとクラスを保管しているディレクトリー (UNIX 環境では sqllib/function、Windows 環境では sqllib¥function) への書き込みアクセス権限をもっているではありません。そのユーザー ID は、データベース・サーバー上のデータベースやオペレーティング・システムに対する読み取りまたは書き込みのアクセス権限ももっているではありません (それが不可能な場合は、少なくとも重要なファイルやディレクトリーに対する読み取りまたは書き込みのアクセス権限を与えないようにします)。

fenced プロセスの所有者がデータベース・サーバー上の各種のクリティカル・リソースへの書き込みアクセス権限をもっていると、システム破壊が起きる可能性があります。例えば、データベース管理者が、ルーチンを独自のプロセス内に囲い込み fenced することにより破壊の可能性を回避できると考えて、未知のソースから受け取った FENCED NOT THREADSAFE などのルーチンを登録したとします。ただし、fenced プロセスを所有しているユーザー ID は、sqllib/function ディレクトリーへの書き込みアクセス権限をもっているとします。ユーザーがこのルーチン呼び出すと、ユーザーが気付かないうちに sqllib/function 内のライブラリーは、NOT FENCED と登録されている別のバージョンのルーチン本体で上書きされてしまいます。後者のルーチンはデータベース・マネージャー全体への無制限のアクセス権限をもっているため、データベース表の機密情報の配布、データベースの破壊、認証情報の収集、またはデータベース・マネージャーの破壊を行う可能性があります。

fenced プロセスを所有するユーザー ID が、データベース・サーバー上の重要なファイルやディレクトリー (特に sqllib/function およびデータベースのデータ・ディレクトリー) への書き込みアクセス権限をもっていないことを確認してください。

ルーチンのライブラリーおよびクラスのぜい弱性

ルーチンのライブラリーおよびクラスを保管しているディレクトリーへのアクセスが制御されていないと、ルーチンのライブラリーとクラスの削除や上書きが可能になってしまいます。前の項で述べたとおり、NOT FENCED ルーチン本体が好ましくない (またはコーディングに不備のある) ルーチンに置き換えられると、データベース・サーバーとそのリソースの安定性、保水性、およびプライバシーが著しく損なわれる可能性があります。

ルーチンの保水性を保護するには、ルーチンのライブラリーとクラスが置かれたディレクトリーへのアクセスを管理しなければなりません。できる限り少数のユーザーにしか、そのディレクトリーとファイルにアクセスできないようにしてください。そのようなディレクトリーへの書き込みアクセス権限

を割り当てるときは、その権限によって該当ユーザー ID の所有者はデータベース・マネージャーとそのすべてのリソースに無制限にアクセスできるようになることに注意してください。

セキュリティに疑いがあるルーチンの展開

未知のソースからルーチンを取得してしまった場合、その作成、登録、呼び出しを行う前に、必ずその機能を綿密に調べてください。そのルーチンを徹底的にテストして不測の副次効果が発生しないことを確認したのでない限り、FENCED および NOT THREADSAFE として登録することをお勧めします。

安全なルーチンの基準を満たさないルーチンを配置する必要が生じた場合、そのルーチンを FENCED および NOT THREADSAFE で登録します。データベースの健全性が必ず維持されるようにするには、FENCED および NOT THREADSAFE ルーチンに関しては以下のとおりにしなければなりません。

- 他のルーチンと共用されない別個の DB2 プロセスで実行します。そうすれば、異常終了してもデータベース・マネージャーは影響を受けません。
- データベースによって使用されるメモリーとは別のメモリーを使用します。そうすれば、値の割り当てで間違いがあっても、データベース・マネージャーは影響を受けません。

ルーチンのコード・ページに関する考慮事項

文字データは、ルーチンの作成時に使用された PARAMETER CCSID オプションによって示されるコード・ページで外部ルーチンに渡されます。同様に、ルーチンから出力される文字ストリングも、PARAMETER CCSID オプションによって示されるコード・ページを使用しているものとデータベースでは見なされます。

例えば、コード・ページ C を使用しているクライアント・プログラムが、コード・ページ S のセクションにアクセスし、そのセクションがコード・ページ R のルーチンを読み出すとすると、以下ようになります。

1. SQL ステートメントを読み出すと、入力文字データは、クライアント・アプリケーションのコード・ページ (C) からセクションのコード・ページ (S) に変換されます。FOR BIT DATA として使用されるデータの BLOB の変換は行われません。
2. ルーチンのコード・ページがセクションのコード・ページと異なっている場合は、ルーチンを読み出される前に、入力文字データ (BLOB と FOR BIT DATA を除く) がルーチンのコード・ページ (R) に変換されます。

サーバー・ルーチンのプリコンパイル、コンパイル、バインドを実行するときには、ルーチンの呼び出し時に使用するコード・ページ (R) を使用することを強くお勧めします。ただしこれは、すべてのケースで可能であるとは限りません。例えば、Windows 環境では Unicode データベースを作成することができます。しかし Windows 環境に Unicode コード・ページがなければ、ルーチンを作成するアプリケーションを Windows のコード・ページでプリコンパイル、コンパイル、およびバインドする必要があります。プリコンパイラーが理解できない特殊な区切り文字がアプリケーションにない場合は、ルーチンは正常に作動します。

3. ルーチンが終了すると、データベース・マネージャーはすべての出力文字データを、必要に応じて、ルーチン・コード・ページ (R) からセクション・コード・ページ (S) へ変換します。実行中にルーチンでエラーが生じた場合のルーチンからの SQLSTATE と診断メッセージも、ルーチン・コード・ページからセクション・コード・ページに変換されます。BLOB または FOR BIT DATA の文字ストリングでは変換は行われません。
4. ステートメントが終了すると、出力文字データはセクション・コード・ページ (S) から元のクライアント・アプリケーションのコード・ページ (C) へ変換されます。FOR BIT DATA として使用された BLOB またはデータの変換は行われません。

CREATE FUNCTION、CREATE PROCEDURE、および CREATE TYPE ステートメントで DBINFO オプションを使用すれば、ルーチンのコード・ページがルーチンに渡されます。この解説を参考にして、コード・ページを重視するルーチンを多種多様なコード・ページで機能するように作成することができます。

32 ビットおよび 64 ビット・アプリケーションおよびルーチンのサポート

DB2 Database for Linux, UNIX, and Windows は、さまざまなプラットフォームでのアプリケーションおよびルーチン (プロシージャおよびユーザー定義関数 (UDF) を含む) の開発とデプロイメントのためのサポートを提供します。アプリケーションおよびルーチンを正しく機能させるためには、DB2 データベース・サポートの 32 ビットおよび 64 ビットに関する考慮事項を検討し、それを理解することが重要です。

まず、いくつかの点を明確にしておきましょう。

- 32 ビット・ハードウェア・プラットフォームは 32 ビット・オペレーティング・システムを稼働し、64 ビット・ハードウェア・プラットフォームは 64 ビット・オペレーティング・システムを稼働します。
- DB2 データベースの 32 ビット・インスタンスは 32 ビット・オペレーティング・システムか 64 ビット・オペレーティング・システムのいずれかにインストールできますが、DB2 データベースの 64 ビット・インスタンスは 64 ビット・オペレーティング・システムにしかインストールできません。
- 32 ビット・アプリケーションは、32 ビット・オペレーティング・システムで構築されたアプリケーションのことを指します。
- 64 ビット・アプリケーションは、64 ビット・オペレーティング・システムで構築されたアプリケーションのことを指します。

以下の表は、クライアント・アプリケーションおよびルーチンに関する DB2 データベースの 32 ビットおよび 64 ビットのサポートを、次の推定とともに概略しています。

表 10. 32 ビットまたは 64 ビット・ハードウェア・プラットフォーム上での 32 ビットおよび 64 ビット・アプリケーションの実行に関するサポート

	32 ビット・ハードウェアおよびオペレーティング・システム	64 ビット・ハードウェアおよびオペレーティング・システム
32 ビット・アプリケーション	YES	YES
64 ビット・アプリケーション	NO	YES

以下の表は、DB2 クライアント・アプリケーションから DB2 データベース・サーバーへの接続の作成に関するサポートを示しています。

表 11. 32 ビットおよび 64 ビット・クライアントから 32 ビットおよび 64 ビット・サーバーへの接続に関するサポート

	32 ビット・サーバー	64 ビット・サーバー
32 ビット・クライアント	YES	YES
64 ビット・アプリケーション	YES	YES

表 12. 32 ビットまたは 64 ビット・ハードウェア・プラットフォーム上での 32 ビットおよび 64 ビット・アプリケーションの実行に関するサポート

	32 ビット・ハードウェアおよびオペレーティング・システム	64 ビット・ハードウェアおよびオペレーティング・システム
32 ビット・アプリケーション	YES	YES
64 ビット・アプリケーション	NO	YES

以下の表は、DB2 クライアント・アプリケーションから DB2 データベース・サーバーへの接続の作成に関するサポートを示しています。

表 13. 32 ビットおよび 64 ビット・クライアントから 32 ビットおよび 64 ビット・サーバーへの接続に関するサポート

	32 ビット・サーバー	64 ビット・サーバー
32 ビット・クライアント	YES	YES
64 ビット・アプリケーション	YES	YES

表 14. 32 ビットおよび 64 ビット・サーバー上での *fenced* および *unfenced* プロシージャおよび UDF の実行に関するサポート

	32 ビット・サーバー	64 ビット・サーバー
32 ビットの <i>fenced</i> プロシージャまたは UDF	YES	YES ^{1, 2, 3}
64 ビットの <i>fenced</i> プロシージャまたは UDF	NO	YES

表 14. 32 ビットおよび 64 ビット・サーバー上での *fenced* および *unfenced* プロシージャ
 ーおよび UDF の実行に関するサポート (続き)

	32 ビット・サーバー	64 ビット・サーバー
32 ビットの <i>unfenced</i> プロシ ージャーまたは UDF	YES	NO ²
64 ビットの <i>unfenced</i> プロシ ージャーまたは UDF	NO	YES

注:

1. 64 ビット・サーバー上での 32 ビット・プロシージャの実行は低速になる場
 合があります。
2. 32 ビット・ルーチンを 64 ビット・サーバー上で機能させるには、これを
 FENCED および NOT THREADSAFE として作成する必要があります。
3. Linux/IA-64 データベース・サーバー上で 32 ビット・ルーチンを呼び出すこと
 はできません。

外部ルーチンの 32 ビットと 64 ビットのサポート

32 ビットと 64 ビットの外部ルーチンのサポートは、ルーチンの CREATE ステ
 ートメントの FENCED 節または NOT FENCED 節という 2 つ節のいずれかの指定
 によって決定されます。

外部ルーチンのルーチン本体はプログラミング言語で作成され、ルーチンの呼び出
 し時にロードされて実行されるライブラリーまたはクラス・ファイルにコンパイル
 されます。FENCED または NOT FENCED 節の指定は、データベース・マネー
 ージャーとは異なる *fenced* 環境で外部ルーチンを実行するか、データベース・マネー
 ージャーと同じアドレッシング・スペースで実行するかを決定します。後者は、通信に
 TCPIP の代わりに共有メモリーが使用されるため、より高いパフォーマンスを提供
 することができます。デフォルトで、ルーチンは、選択される他の節に関係なく常
 に *fenced* として作成されます。

以下の表は、同じオペレーティング・システムで実行されている 32 ビットおよび
 64 ビットのデータベース・サーバーで *fenced* および *unfenced* の 32 ビットおよ
 び 64 ビット・ルーチンを実行するための DB2 のサポートを説明しています。

表 15. 32 ビットおよび 64 ビットの外部ルーチンのサポート

ルーチンのビット幅	32 ビット・サーバー	64 ビット・サーバー
32 ビットの <i>fenced</i> プロシ ージャー または UDF	サポートされる	サポートされる ¹
64 ビットの <i>fenced</i> プロシ ージャー または UDF	サポートされていない ³	サポートされる
32 ビットの <i>unfenced</i> プロシ ージャー または UDF	サポートされる	サポートされる ^{1,2}
64 ビットの <i>unfenced</i> プロシ ージャー または UDF	サポートされていない ³	サポートされる

注:

1. 64 ビット・サーバー上での 32 ビット・ルーチンの実行には 64 ビット・サーバー上での 64 ビット・ルーチンの実行ほどの実行速度はありません。
2. 32 ビット・ルーチンを 64 ビット・サーバー上で機能させるには、これを FENCED および NOT THREADSAFE として作成する必要があります。
3. 64 ビット・アプリケーションおよびルーチンを 32 ビット・アドレッシング・スペースで実行することはできません。

表の中の注目すべき重要な点は、32 ビットの unfenced プロシージャを 64 ビットの DB2 サーバーで実行することができないという点です。32 ビットの unfenced ルーチンを 64 ビット・プラットフォームにデプロイしなければならない場合は、カタログする前に、そのルーチンの CREATE ステートメントから NOT FENCED 節を除去してください。

64 ビット・データベース・サーバー上での 32 ビット・ライブラリーを持つルーチンのパフォーマンス

64 ビット DB2 データベース・サーバー上で 32 ビット・ルーチン・ライブラリーを持つルーチンを呼び出すことは可能です。しかしこの場合、64 ビット・サーバー上で 64 ビット・ルーチンを呼び出すほどのパフォーマンスは発揮しません。パフォーマンスが低下する理由は、64 ビット・サーバー上で 32 ビット・ルーチンを実行しようとするたびにまずそれを 64 ビット・ライブラリーとして呼び出そうとするからです。これが失敗してから、そのライブラリーは 32 ビット・ライブラリーとして呼び出されます。32 ビット・ライブラリーを 64 ビット・ライブラリーとして呼び出す試行が失敗すると、db2diag ログ・ファイルにエラー・メッセージ (SQLCODE -444) が生成されます。

Java クラスのビット幅はそれぞれ異なります。Java 仮想マシン (JVM) だけが 32 ビットまたは 64 ビットとして分類されます。DB2 では JVM を使用するインスタンスと同じビット幅の JVM の使用のみがサポートされます。言い換えると、32 ビット DB2 インスタンスでは 32 ビット JVM だけを使用でき、64 ビット DB2 インスタンスでは 64 ビット JVM だけを使用できるということです。これにより、Java ルーチンは正しく機能し、可能な限り最高のパフォーマンスを発揮できます。

外部ルーチンでの XML データ・タイプのサポート

下記のプログラミング言語で書かれている外部プロシージャおよび関数は、データ・タイプ XML のパラメーターおよび変数をサポートします。

- C
- C++
- COBOL
- Java
- .NET CLR 言語

OLE および OLEDB 外部ルーチンは、データ・タイプ XML のパラメーターをサポートしません。

XML データ・タイプの値は、CLOB データ・タイプと同じ方法で外部ルーチンのコード中に示されます。

データ・タイプ XML の外部ルーチン・パラメーターを宣言するときは、データベース内でそのルーチンを作成するとき使用する CREATE PROCEDURE および CREATE FUNCTION ステートメントで、XML データ・タイプを CLOB データ・タイプとして保管することを指定する必要があります。CLOB 値のサイズは、XML パラメーターで表される XML 文書のサイズに近くなければなりません。

以下の CREATE PROCEDURE ステートメントは、parm1 という XML パラメーターを使用して C プログラミング言語でインプリメントされた外部プロシージャの CREATE PROCEDURE ステートメントを示しています。

```
CREATE PROCEDURE myproc(IN parm1 XML AS CLOB(2M), IN parm2 VARCHAR(32000))
LANGUAGE C
FENCED
PARAMETER STYLE SQL
EXTERNAL NAME 'mylib!myproc';
```

以下の例に示されているような外部 UDF の作成時にも、それに似た考慮事項が当てはまります。

```
CREATE FUNCTION myfunc (IN parm1 XML AS CLOB(2M))
RETURNS SMALLINT
LANGUAGE C
PARAMETER STYLE SQL
DETERMINISTIC
NOT FENCED
NULL CALL
NO SQL
NO EXTERNAL ACTION
EXTERNAL NAME 'mylib!myfunc'
```

XML データは、ストアード・プロシージャに IN、OUT、または INOUT パラメーターとして渡されるときにマテリアライズされます。Java ストアード・プロシージャを使用している場合、XML 引数の数量とサイズ、および並行に実行されている外部ストアード・プロシージャの数に基づいて、ヒープ・サイズ (JAVA_HEAP_SZ 構成パラメーター) を増やさなければならない場合があります。

外部ルーチン・コード内部では、XML パラメーターおよび変数値へのアクセス、その設定、および変更は、データベース・アプリケーションの場合と同じやり方で行われます。

外部ルーチンに関する制約事項

外部ルーチンには以下の制約事項が適用されます。これらの制約事項は、外部ルーチンを開発およびデバッグするときに考慮してください。

すべての外部ルーチンに適用される制約事項:

- 外部ルーチンに新しいスレッドを作成できない。
- 外部関数または外部メソッド内から接続レベル API を呼び出せない。
- キーボードからの入力の受信または標準出力への出力の表示を外部ルーチンから行えない。標準入出力ストリームは使用しないでください。以下に例を示します。

- 外部 Java ルーチン・コードでは、`System.out.println()` メソッドを発行しない。
- 外部 C または C++ ルーチン・コードでは、`printf()` を発行しない。
- 外部 COBOL ルーチン・コードでは、`display` を発行しない。

外部ルーチンはデータを標準出力に表示できませんが、データベース・サーバー・ファイル・システム上のファイルにデータを書き込むコードをこれに組み込むことは可能です。

UNIX 環境上で稼働する `fenced` ルーチンの場合、ファイルの作成先のターゲット・ディレクトリーやファイルそのものが適切な許可をもち、`sqllib/adm/.fenced` ファイルの所有者がその作成や書き込みを行えるようにしなければなりません。 `not fenced` ルーチンの場合、インスタンス所有者は、ファイルをオープンする場所であるディレクトリーを対象とした作成、読み取り、および書き込みの許可を持っていないければなりません。

注: DB2 は、ルーチンによって実行される外部の入出力と DB2 独自のトランザクションとの同期を試みることはありません。したがって、例えばトランザクションの処理中に UDF がファイルに書き込みを行った後で、そのトランザクションが何らかの理由でバックアウトされても、そのファイルへの書き込みの探索や取り消しは試みられません。

- 接続に関連したステートメントまたはコマンドは外部ルーチンで実行できない。この制限は、以下のステートメントにも適用されます。
 - BACKUP
 - CONNECT
 - CONNECT TO
 - CONNECT RESET
 - CREATE DATABASE
 - DROP DATABASE
 - FORWARD RECOVERY
 - RESTORE
- ルーチン内でオペレーティング・システム関数を使用することは推奨されていない。基本的にこの関数の使用には制限がありませんが、以下の場合例外です。
 - ユーザー定義のシグナル・ハンドラーを外部ルーチンにインストールしてはならない。この制約事項を守らないと、外部ルーチン実行時の不測の障害、データベースの異常終了、またはその他の問題を生じることがあります。シグナル・ハンドラーをインストールすると、JVM for Java ルーチンの操作が妨げられることもあります。
 - 処理を終了するシステム呼び出しを行うと DB2 の処理の 1 つが異常終了し、データベース・システムまたはデータベース・アプリケーションの障害が発生する。

他のシステム呼び出しによって DB2 データベース・マネージャーの通常の操作が妨害される場合にも、問題が発生することがあります。例えば、ユーザー定義関数が含まれるライブラリーを関数がメモリーからアンロードしようとする、重大な問題が発生することがあります。システム呼び出しが含まれる外部ルーチンのコーディングとテストには注意してください。

- 現行処理を終了させるコマンドを外部ルーチンに入れてはならない。外部ルーチンは、現行プロセスを終了させることなく常に DB2 データベース・マネージャーに制御を戻さなければなりません。
- 特別な場合を除き、データベースがアクティブになっている間に外部ルーチン・ライブラリー、クラス、またはアセンブリーを更新してはならない。DB2 データベース・マネージャーがアクティブになっている間に更新の必要が生じ、インスタンスの停止と開始ができない場合には、ルーチン用にライブラリー、クラス、またはアセンブリーを別に新規作成します。その後、新しいライブラリー、クラス、またはアセンブリー・ファイルの名前が参照されるように、ALTER ステートメントを使って外部ルーチンの EXTERNAL NAME 節の値を変更します。
- 外部ルーチンで環境変数 DB2CKPTR を使用できない。名前が「DB2」で始まるその他のすべての環境変数は、データベース・マネージャーの開始時にキャプチャーされ、外部ルーチンで使用することができます。
- 「DB2」で名前が始まらない環境変数の中には fenced の外部ルーチンで使用できないものもある。例えば、LIBPATH 環境変数は使用できません。ただし、この種の変数は not fenced の外部ルーチンでは使用することができます。
- DB2 データベース・マネージャーの開始後に設定された環境変数の値は外部ルーチンで使用できない。
- 外部ルーチン内での保護リソース (一度に 1 つのプロセスによってのみアクセスできるリソース) の使用は制限する必要がある。使用する場合は、2 つの外部ルーチンが保護リソースにアクセスしようとする際にデッドロックが発生する可能性があるだけ小さくなるようにしてください。保護リソースへのアクセス試行中に複数の外部ルーチンでデッドロックが発生すると、DB2 データベース・マネージャーはこの状況を検出または解決することができません。これにより、外部ルーチンのプロセスはハングしてしまいます。
- 外部ルーチン・パラメーターのメモリーを DB2 データベース・サーバーに対して明示的に割り振ってはならない。DB2 データベース・マネージャーはルーチンに対して、CREATE ステートメント内のパラメーター宣言に基づいてストレージを自動的に割り振ります。外部ルーチン内のパラメーターのストレージ・ポインターを変えないでください。ポインターを、ローカル作成したストレージ・ポインターに置き換えようとする、メモリー・リーク、データ破壊、または異常終了が発生する可能性があります。
- 外部ルーチン内で静的データまたはグローバル・データを使用してはならない。DB2 では、静的変数またはグローバル変数によって使用されたメモリーが、外部ルーチンの次の呼び出しまで不変のままでは限りません。UDF およびメソッドの場合はスクラッチパッドを使用すれば、次の呼び出しでも使用できるように値を保管しておくことができます。
- SQL パラメーター値はすべてバッファーに入れられる。これは、その値がコピーされて外部ルーチンに渡されることを意味します。外部ルーチンの入力パラメーターに変更が加えられても、SQL の値または処理に対してその変更は効力を持ちません。ただし、CREATE ステートメントで指定されている以上のデータを外部ルーチンが入力または出力パラメーターに書き込んだ場合、メモリー破壊が発生し、ルーチンは異常終了する可能性があります。

外部プロシージャにのみ適用される制約事項

- ネストされたストアード・プロシージャから結果セットが戻される場合、複数のネスト・レベルにまたがって同一名でカーソルをオープンすることができる。ただし、バージョン 8 より前のアプリケーションは、オープンした最初の結果セットにしかアクセスすることはできません。この制約事項は、別のパッケージ・レベルでオープンされたカーソルには適用されません。

外部関数にのみ適用される制約事項

- 外部関数は結果セットを戻すことはできない。外部関数内でオープンされたカーソルはすべて、関数の最終呼び出しが完了した時点でクローズされなければなりません。
- 外部ルーチンが戻す前に、外部ルーチンでのメモリーの動的割り振りを解放する必要がある。これが行われないと、メモリー・リークが発生したり、DB2 プロセス内でのメモリーの消費が増え続けたりすることになります。これは、データベース・システムでのメモリー不足を引き起こしかねません。

外部ユーザー定義関数および外部メソッドの場合、複数の関数呼び出しに必要な動的メモリーの割り振りをスクラッチパッドを使用して行える。この目的でスクラッチパッドを使用する場合は、CREATE FUNCTION または CREATE METHOD ステートメントで FINAL CALL 属性を指定します。こうすることで、割り振り済みのメモリーを、ルーチンが戻す前に確実に解放できます。

外部ルーチンの作成

プロシージャと関数を含む外部ルーチンは、他のインプリメンテーションを使用したルーチンと類似した方法で作成されます。しかし、ルーチンのインプリメンテーションでは、ソース・コードのコード化、コンパイル、およびデプロイが必要になるため、さらにいくつかのステップが必要です。

外部ルーチンをインプリメントするのは、以下のような場合です。

- データベースにアクセスするルーチンや、データベースの外部でアクションを実行するルーチンに複雑なロジックをカプセル化したい場合。
- 複数のアプリケーション、CLP、他のルーチン（プロシージャ、関数 (UDF)、メソッド）、トリガーのいずれかから、カプセル化されたロジックを呼び出す必要がある場合。
- そのロジックのコーディングに、SQL や SQL PL ステートメントを使用するよりも、プログラミング言語が最も使いやすいと感じる場合。
- ルーチン・ロジックにデータベース外での操作を実行させたい場合。これは例えば、データベース・サーバー上でのファイルへの書き込みまたはファイルからの読み取り、別のアプリケーションの実行、または SQL および SQL PL ステートメントで表すことができないロジックなどです。

前提条件

- IBM Data Server Clientがインストールされている必要があります。
- データベース・サーバーが、選択したインプリメンテーション・プログラミング言語コンパイラおよび開発ソフトウェアをサポートするオペレーティング・システムを実行していること。

- 選択したプログラミング言語の必須コンパイラおよびランタイム・サポートがデータベース・サーバー上にインストールされていること。
- CREATE PROCEDURE、CREATE FUNCTION、または CREATE METHOD ステートメントを実行する権限。

外部ルーチンに関連した制約事項のリストについては、以下を参照してください。

- 104 ページの『外部ルーチンに関する制約事項』

手順

1. 選択したプログラミング言語でルーチン・ロジックをコーディングします。
 - 外部ルーチン、ルーチン・フィーチャー、およびルーチン・フィーチャーのインプリメンテーションの一般情報については、『前提条件』のセクションで参照されているトピックを参照してください。
 - SQL ステートメントの実行をサポートするために必要な、必須ヘッダー・ファイルを使用またはインポートします。
 - DB2 SQL データ・タイプにマップするプログラミング言語データ・タイプを使用して、変数とパラメーターを正しく宣言します。
2. 選択したプログラミング言語のパラメーター・スタイルが要求する形式に従って、パラメーターを宣言する必要があります。パラメーターとプロトタイプ宣言の詳細については、以下を参照してください。
 - 82 ページの『外部ルーチンのパラメーター・スタイル』
3. コードからライブラリーまたはクラス・ファイルを作成します。
4. そのライブラリーまたはクラス・ファイルをデータベース・サーバー上の DB2 *function* ディレクトリーにコピーします。DB2 ルーチンに関連したアセンブリーまたはライブラリーは、*function* ディレクトリーに保管することをお勧めします。*function* ディレクトリーの詳細については、CREATE PROCEDURE ステートメントまたは CREATE FUNCTION ステートメントのいずれかの EXTERNAL 節を参照してください。

アセンブリーをサーバー上の別のディレクトリーにコピーすることもできますが、ルーチンを正常に呼び出すには、アセンブリーの完全修飾パス名をメモしておく必要があります。次のステップでこれが必要になるからです。

5. 該当するルーチン・タイプの SQL 言語 CREATE ステートメント (CREATE PROCEDURE または CREATE FUNCTION) を動的または静的に実行します。
 - 選択した API またはプログラミング言語の該当する値を使用して、LANGUAGE 節を指定します。例として、CLR、C、JAVA が含まれます。
 - PARAMETER STYLE 節に、ルーチン・コードでインプリメントした有効なパラメーター・スタイルの名前を指定します。
 - EXTERNAL 節に、ルーチンと関連したライブラリー、クラス、またはアセンブリー・ファイルの名前を指定します。そのためには、以下のいずれかの値を使用します。
 - ルーチン・ライブラリー、クラス、またはアセンブリー・ファイルの完全修飾パス名。
 - *function* ディレクトリーを基準にしたルーチン・ライブラリー、クラス、またはアセンブリー・ファイルの相対パス名。

EXTERNAL 節にライブラリー、クラス、またはアセンブリー・ファイルの完全修飾パス名または相対パス名を指定しない場合、DB2 はデフォルトで、function ディレクトリー内でそれらの名前を探します。

- ルーチンがプロシージャの場合に、DYNAMIC RESULT SETS に数値を指定すると、呼び出し元に 1 つ以上の結果セットが戻されます。
- ルーチンの特徴を指定するために必要な他の節を指定します。

外部ルーチンを呼び出すには、ルーチンの呼び出しを参照してください。

ルーチンの作成

3 つのタイプのルーチン (プロシージャ、UDF、メソッド) には、その作成法に関して多くの共通点があります。例えば、それら 3 つのルーチン・タイプは、同じパラメーター・スタイルをいくつか使用し、各種クライアント・インターフェース (組み込み SQL、CLI、JDBC) を介して SQL の使用をサポートします。また、いずれも他のルーチンを呼び出すことができます。その例として、以下のステップはルーチンを作成する 1 つのアプローチを示しています。

特定のルーチン・タイプだけに用意されている機能もあります。例えば、結果セットはストアド・プロシージャ独特のものであり、スクラッチパッドは UDF およびメソッド独自のものです。開発しようとしているルーチン・タイプに当てはまらないステップに行き当たったら、その後のステップに進んでください。

ルーチンを作成する場合は、事前に以下を決定する必要があります。

- 必要なルーチン・タイプ。
- 作成に使用するプログラム言語。
- ルーチン内に SQL ステートメントが必要な場合にどのインターフェースを使用するか。

『セキュリティ、ライブラリー、およびクラス管理』および『パフォーマンスの考慮事項』の項も参照してください。

ルーチン本体を作成するには、以下を行う必要があります。

1. 外部ルーチンにのみ当てはまります。呼び出し側のアプリケーションまたはルーチンからの入力パラメーターを受け入れて、出力パラメーターを宣言します。ルーチンがパラメーターをどのように受け入れるかは、ルーチンの作成に使用するパラメーター・スタイルによって異なります。各パラメーター・スタイルは、ルーチン本体に渡される一連のパラメーターと、パラメーターが渡される順序を定義します。

例えば、PARAMETER STYLE SQL 用に C で書かれた UDF 本体のシグニチャー (sqludf.h を使用して) を以下に示します。

```
SQL_API_RC SQL_API_FN product ( SQLUDF_DOUBLE *in1,
                                SQLUDF_DOUBLE *in2,
                                SQLUDF_DOUBLE *outProduct,
                                SQLUDF_NULLIND *in1NullInd,
                                SQLUDF_NULLIND *in2NullInd,
                                SQLUDF_NULLIND *productNullInd,
                                SQLUDF_TRAIL_ARGS )
```

2. ルーチンが実行するロジックを追加します。ルーチンの本体で使用できる機能には次のようなものがあります。
 - 他のルーチンの呼び出し (ネスティング)、または現在のルーチンの呼び出し (再帰)。
 - SQL (CONTAINS SQL、READS SQL、または MODIFIES SQL) を組み込むように定義されたルーチンでは、ルーチンから SQL ステートメントを発行することができます。呼び出すステートメントのタイプは、ルーチンの登録方法によって制御します。
 - 外部 UDF およびメソッドでは、スクラッチパッドを使用して、複数の呼び出しにまたがって状態を保管します。
 - SQL プロシージャでは、条件ハンドラーを使用して、指定の条件が発生したときの SQL プロシージャの動作を指定します。そのような条件は、SQLSTATE をベースにして定義することができます。
3. ストアード・プロシージャにのみ当てはまります。1 つ以上の結果セットを戻します。呼び出し側のアプリケーションとの間で交換される個々のパラメーターに加えて、複数の結果セットを戻す機能がストアード・プロシージャに備わっています。SQL ルーチンと、CLI、ODBC、JDBC、および SQLJ ルーチンとクライアントのみが、結果セットを受け入れることができます。

ルーチンの作成以外に、ルーチンを呼び出すにはまず登録する必要があります。それには、開発しているルーチンのタイプに合った CREATE ステートメントを使用します。一般的に、ルーチンを作成して登録する順序は問題にはなりません。ただし、ルーチンが自身を参照する SQL を発行する場合は、作成の前にルーチンを登録する必要があります。その場合にバインドを正常に完了するには、ルーチンの登録が事前に完了していなければなりません。

ルーチンのデバッグ

実動サーバーにルーチンを配置する前に、テスト・サーバーでそのルーチンを徹底的にテストしてデバッグする必要があります。これは、NOT FENCED として登録する必要のあるルーチンの場合には特に重要です。このルーチンは、データベース・マネージャーのメモリー、そのデータベース、およびデータベース制御構造に無制限でアクセスできるからです。FENCED THREADSAFE ルーチンにも厳重な注意が必要です。このルーチンは他のルーチンとメモリーを共用するからです。

ルーチンの一般的な問題のチェックリスト

ルーチンが正しく稼働していることを確認するには、以下を調べます。

- ルーチンが正しく登録されている。CREATE ステートメントに指定するパラメーターは、ルーチン本体で処理される引数に一致している必要があります。この点に留意しながら、以下の項目を 1 つずつチェックします。
 - ルーチン本体で使用されている引数のデータ・タイプは、CREATE ステートメントに定義されているパラメーター・タイプに適したタイプである。
 - CREATE ステートメント内の対応する結果に対して定義されているより大きいバイト数がルーチンによって出力変数に書き込まれていない。

- 対応する CREATE オプションを使用してルーチンが登録されていた場合は、SCRATCHPAD、FINAL CALL、DBINFO 用のルーチン引数が存在する。
- 外部ルーチンの場合、CREATE ステートメント内の EXTERNAL NAME 節の値は、ルーチン・ライブラリーおよびエントリー・ポイントに一致していなければならない (大文字小文字の区別はオペレーティング・システムによって異なります)。
- C++ ルーチンの場合、C++ コンパイラーはタイプ修飾をエントリー・ポイント名に適用します。タイプ修飾名を EXTERNAL NAME 節に指定する必要がありますが、そうでない場合、ユーザー・コード内でエントリー・ポイントを extern "C" と定義しなければなりません。
- 呼び出し時に指定するルーチン名は、そのルーチンの登録名 (CREATE ステートメント内で定義済みの名前) に一致していなければならない。デフォルトではルーチン ID は英大文字に変換されます。それは、区切り ID には当てはまりません。区切り ID の場合、大文字には変換されずに、大文字小文字が区別されます。

ルーチンは、CREATE ステートメントに指定されているディレクトリー・パスに置かれなければなりません。ただし、パスを指定しないと、デフォルトで DB2 がそれを探索します。UDF、メソッド、および fenced プロシージャの場合にはそれは、sqllib/function (UNIX) または sqllib¥function (Windows) となります。unfenced プロシージャの場合にはそれは、sqllib/function/unfenced (UNIX) または sqllib¥function¥unfenced (Windows) となります。

- 正しい呼び出しシーケンス、プリコンパイル (組み込み SQL の場合)、およびリンク・オプションを使用してルーチンが作成されている。
- アプリケーションが DB2 CLI、ODBC、または JDBC を使用して作成されている場合を除き、アプリケーションはデータベースにバインドされている。ルーチンに SQL が入っていて、しかもそのようなインターフェースを使用しない場合、ルーチンをバインドする必要もあります。
- そのルーチンは、クライアント・アプリケーションにエラー情報を正確に戻す。
- FINAL CALL を使用してルーチンが定義されている場合は、使用可能なすべての呼び出しタイプを考慮に入れる。
- ルーチンによって使用されるシステム・リソースが戻される。
- ルーチンを呼び出そうとしたときに、その操作を実行するための十分な特権がないことを示すエラー (SQLCODE -551、SQLSTATE 42501) を受け取った場合は、ルーチンに関する EXECUTE 特権がないことが原因になっている可能性が高いと言えます。この特権をルーチンの呼び出し側に付与できるのは、SECADM 特権、ACCESSCTRL 特権を持つユーザーか、ルーチンに関する EXECUTE WITH GRANT OPTION 特権を持つユーザーです。権限とルーチンについての関連トピックでは、この特権の使用を効率的に管理する方法を詳しく説明しています。

ルーチンのデバッグの技法

ルーチンをデバッグするには、次のような技法を使用します。

- Development Center には、SQL を本体とするプロシージャと Java プロシージャ用の包括的デバッグ・ツールが備えられています。
- ルーチンから画面に診断データを書き込むことはできません。診断データをファイルに書き込む場合、必ず %tmp などのグローバル・アクセスの可能なディレクトリーに書き込んでください。データベース・マネージャーやデータベースによって使用されるディレクトリーには書き込まないでください。

プロシージャの場合はその代わりに SQL 表に診断データを書き込むのが安全な方法です。SQL 表に書き込めるようにするには、MODIFIES SQL DATA 節を使用して、テストするプロシージャを登録する必要があります。既存のプロシージャでデータを SQL 表に書き込む (または書き込みをやめる) 必要がある場合、プロシージャをいったんドロップしてから、MODIFIES SQL DATA 節を使用して (または使用しないで) 再登録しなければなりません。プロシージャをドロップして再登録する場合、事前にその従属関係に配慮してください。

- ルーチンのエントリー・ポイントを直接呼び出す単純なアプリケーションを作成すれば、ルーチンをローカル側でデバッグすることができます。装備されているデバッガーの使用法の詳細は、コンパイラーの資料をご覧ください。

SQL プロシージャからのエラー・メッセージの戻り

SQL プロシージャの CREATE PROCEDURE ステートメントを発行した場合、DB2 は SQL プロシージャ本体の構文を受け入れたとしても、プリコンパイルまたはコンパイル段階において SQL プロシージャを作成しない可能性があります。このような状況下では、DB2 は通常、エラー・メッセージが含まれるログ・ファイルを作成します。

SQL プロシージャの DB2 および C コンパイラーで生成されたエラー・メッセージを検索するには、データベース・サーバー上の以下のディレクトリーでメッセージ・ログ・ファイルを表示します。

UNIX *instance/function/routine/sqlproc/db_name/schema_name/tmp*

ただし *instance* は DB2 インスタンスのパスを表し、*db_name* はデータベースの別名を表し、そして *schema_name* は、CREATE PROCEDURE ステートメントの発行に使用されたスキーマを表します。

Windows

instance%function%routine%sqlproc%db_name%schema_name%tmp

ただし *instance* は DB2 インスタンスのパスを表し、*db_name* はデータベースの別名を表し、そして *schema_name* は、CREATE PROCEDURE ステートメントの発行に使用されたスキーマを表します。

注: CREATE PROCEDURE ステートメントの一部として SQL プロシージャ・スキーマ名が発行されないと、DB2 は CURRENT SCHEMA 特殊レジスターの値を使用します。CURRENT SCHEMA 特殊レジスターの値を表示するには、CLP で以下のようなステートメントを発行してください。

```
VALUES CURRENT SCHEMA
```

外部ルーチンのライブラリーおよびクラスの管理

外部ルーチンを首尾よく開発し、呼び出すには、外部ルーチンのライブラリーおよびクラス・ファイルを適切にデプロイし、管理する必要があります。

外部ルーチンのライブラリーおよびクラス・ファイルの管理は、外部ルーチンを初めて作成し、ライブラリーおよびクラス・ファイルをデプロイするときに注意を払うことにより、最小限に抑えることができます。

外部ルーチンの管理に関する主な考慮事項は次のとおりです。

- 外部ルーチンのライブラリーおよびクラス・ファイルのデプロイメント
- 外部ルーチン・ライブラリーおよびクラス・ファイルのセキュリティ
- 外部ルーチンのライブラリーおよびクラスの解決
- 外部ルーチンのライブラリーおよびクラス・ファイルに対する変更
- 外部ルーチンのライブラリーおよびクラス・ファイルのバックアップおよびリストア

システム管理者、データベース管理者、およびデータベース・アプリケーション開発者のすべてが、ルーチンの開発中およびデータベース管理タスク実行時の外部ルーチンのライブラリーおよびクラス・ファイルの機密保護を確保し、それらが正しく保存されることに責任を持つ必要があります。

外部ルーチン・ライブラリーおよびクラスのデプロイメント

外部ルーチン・ライブラリーおよびクラスのデプロイメントは、外部ルーチン・ライブラリーおよびクラスをソース・コードからビルドされた後に、それらをデータベース・サーバーにコピーすることを指します。

外部ルーチン・ライブラリー、クラス、またはアセンブリー・ファイルは、データベース・サーバーの *DB2 function* ディレクトリーまたはこのディレクトリーのサブディレクトリーにコピーする必要があります。これは、外部ルーチンのデプロイメントの推奨される位置です。 *function* ディレクトリーの詳細については、SQL ステートメント (CREATE PROCEDURE または CREATE FUNCTION) のいずれかの *EXTERNAL* 節の説明を参照してください。

ルーチンのインプリメントに使用する API およびプログラミング言語に応じて、外部ルーチンのクラス、ライブラリー、またはアセンブリーをサーバー上の他のディレクトリーの位置にコピーすることができます。ただし、これは通常お勧めできません。これを実行した場合、ルーチンを正常に呼び出すために、完全修飾パス名に特に注意して、この値が *EXTERNAL NAME* 節で使用されていることを確認する必要があります。

ライブラリーおよびクラス・ファイルは、ごく普通に入手できるファイル転送ツールを使ってデータベース・サーバーのファイル・システムにコピーすることができます。特別なシステム定義のプロシージャを使って (特にこの目的のために設計された)、Java ルーチンを DB2 クライアントがインストールされているコンピューターから DB2 データベース・サーバーにコピーすることができます。詳しくは、Java ルーチンに関するトピックを参照してください。

該当するルーチン・タイプの SQL 言語 CREATE ステートメント (CREATE PROCEDURE または CREATE FUNCTION) を実行する場合、EXTERNAL NAME 節に特に注意して、該当する節を指定します。

- 選択した API またはプログラミング言語の該当する値を使用して、LANGUAGE 節を指定します。例として、CLR、C、JAVA が含まれます。
- PARAMETER STYLE 節に、ルーチン・コードでインプリメントした有効なパラメーター・スタイルの名前を指定します。
- EXTERNAL 節に、ルーチンと関連したライブラリー、クラス、またはアセンブリー・ファイルの名前を指定します。そのためには、以下のいずれかの値を使用します。
 - ルーチン・ライブラリー、クラス、またはアセンブリー・ファイルの完全修飾パス名。
 - function ディレクトリーを基準にしたルーチン・ライブラリー、クラス、またはアセンブリー・ファイルの相対パス名。

EXTERNAL 節にライブラリー、クラス、またはアセンブリー・ファイルの完全修飾パス名または相対パス名を指定しない場合、DB2 はデフォルトで、function ディレクトリー内でそれらの名前を探します。

外部ルーチン・ライブラリーまたはクラス・ファイルのセキュリティ

外部ルーチン・ライブラリーは、データベース・サーバーのファイル・システムに保管され、DB2 データベース・マネージャーではどんな方法でもバックアップまたは保護されることはありません。ルーチンが正常に呼び出され続けるようにするには、ルーチンに関連したライブラリーが、ルーチンの作成に使用される CREATE ステートメントの EXTERNAL 節で指定された位置に存在し続ける必要があります。ルーチンの作成後に、ルーチン・ライブラリーを移動したり削除しないでください。これを行うと、ルーチンの呼び出しは失敗します。

ルーチン・ライブラリーが意図せずにはまたは故意に削除されたり置き換えられたりしないようにするには、データベース・サーバー上のルーチン・ライブラリーを含むディレクトリーへのアクセス、およびルーチン・ライブラリー・ファイルへのアクセスを制限する必要があります。これは、オペレーティング・システムのコマンドを使用して、ディレクトリーおよびファイル許可を設定することによって行うことができます。

外部ルーチンのライブラリーおよびクラスの解決

DB2 外部ルーチン・ライブラリーの解決は、DB2 インスタンス・レベルで実行されます。これは、複数の DB2 データベースが含まれる DB2 インスタンスでは、外部ルーチンを、あるデータベースのルーチン用にすでに使用されている外部ルーチン・ライブラリーを使用する、別のデータベースに作成できることを意味します。

インスタンス・レベルの外部ルーチンの解決では、複数のルーチン定義が単一のライブラリーに関連付けられるようにして、コードの再利用をサポートします。外部ルーチン・ライブラリーをこの方法で再利用せず、代わりに外部ルーチン・ライブラリーのコピーがデータベース・サーバーのファイル・システムに存在する場合、ライブラリー名の競合が発生する可能性があります。このことが特に起きるのは、

単一のインスタンス内に複数のデータベースが存在し、各データベース内のルーチンがルーチン本体の独自のライブラリーおよびクラスのそれ自体のコピーに関連付けられている場合です。あるデータベースのルーチンが使用するライブラリーまたはクラスの名前が、(同じインスタンス内の)別のデータベースのルーチンが使用するライブラリーまたはクラスの名前と同一の場合には、競合が生じます。

これが起きる可能性を最小限に抑えるには、ルーチン・ライブラリーの単一コピーをインスタンス・レベルの `function` ディレクトリー (`sqllib/function` ディレクトリー) に保管し、各データベースのすべてのルーチン定義の `EXTERNAL` 節が固有のライブラリーを参照するようにすることをお勧めします。

機能的に異なる 2 つのルーチン・ライブラリーを同じ名前で作成する必要がある場合は、ライブラリー名が競合する可能性を最小限に抑えるために追加のステップを行うことが重要です。

C、C++、COBOL、および ADO.NET ルーチンの場合:

以下のようにして、ライブラリー名の競合を最小限に抑えるかあるいは解決することができます。

1. ライブラリーを、各データベースの別個のディレクトリーにルーチン本体と共に保管する。
2. 指定したライブラリーの (相対パスの代わりに) 絶対パスを指定する `EXTERNAL NAME` 節の値を使用してルーチンを作成する。

Java ルーチンの場合:

`CLASSPATH` 環境変数の有効範囲はインスタンス全体なので、疑わしいクラス・ファイルを別のディレクトリーに移動してもクラス名の競合は解決できません。 `CLASSPATH` で最初に出現するクラスが、使用されるクラスです。このため、同じ名前のクラスを参照する 2 つの異なる Java ルーチンがある場合には、このルーチンのいずれかが間違ったクラスを使用します。可能な 2 つの解決策があります。関係するクラスを名前変更するか、各データベースの別個のインスタンスを作成します。

外部ルーチンのライブラリーおよびクラス・ファイルに対する変更

既存の外部ルーチンのロジックに対する変更は、外部ルーチンをデプロイしており、それが実動データベース・システム環境で使用中心になってから必要になる場合があります。既存のルーチンに対する変更を行うことは可能ですが、更新のための明確なテークオーバーのポイント・イン・タイムを定義し、ルーチンの並行呼び出しが中断するリスクを最小限に抑えるように、注意深く行うことが重要です。

外部ルーチン・ライブラリーを更新する必要が生じたら、データベース・マネージャーの稼働中に現行ルーチンが使用しているのと同じターゲット・ファイル (例えば、`sqllib/function/foo.a`) にルーチンを再コンパイルおよび再リンクしないでください。ルーチンの現在の呼び出しがルーチン・プロセスのキャッシュ・バージョンにアクセスする場合、基本ライブラリーが置き換えられていると、ルーチンの呼び出しは失敗することがあります。DB2 の停止と再始動の過程を経ないでルーチンの本体を変更する必要が生じた場合、以下のステップを行ってください。

1. 別のライブラリーまたはクラス・ファイル名を使用して、新規の外部ルーチン・ライブラリーを作成します。

2. それが組み込み SQL ルーチンである場合、BIND コマンドを使用してルーチン・パッケージをデータベースにバインドします。
3. ALTER ROUTINE ステートメントを使用してルーチン定義を変更し、EXTERNAL NAME 節が更新されたルーチン・ライブラリーまたはクラスを参照できるようにします。更新されるルーチン本体が、複数のデータベースにカタログされたルーチンによって使用される場合には、このセクションで指示されたアクションを、関係する各データベースについて実行しなければなりません。
4. JAR ファイルに組み入れられた Java ルーチンの更新の場合、CALL SQLJ.REFRESH_CLASSES() ステートメントを発行して、DB2 で強制的に新規クラスをロードする必要があります。Java ルーチン・クラスを更新した後に CALL SQLJ.REFRESH_CLASSES() ステートメントを発行しないと、DB2 は以前のバージョンのクラスを使用し続けます。DB2 は、COMMIT または ROLLBACK が生じると、クラスをリフレッシュします。

ルーチン定義が更新されたら、ルーチンの後続の呼び出しはすべて、新規の外部ルーチン・ライブラリーまたはクラスをロードして実行します。

外部ルーチンのライブラリーおよびクラス・ファイルのバックアップおよびリストア

外部ルーチン・ライブラリーは、データベースのバックアップが実行されるときに、他のデータベース・オブジェクトとともにバックアップされません。同様に、データベースがリストアされる際に、外部ルーチン・ライブラリーはリストアされません。

データベースのバックアップおよびリストアの目的がデータベースの再デプロイである場合、外部ルーチンのライブラリー・ファイルを、外部ルーチン・ライブラリーの相対パス名を保存するのと同じ方法で、元のデータベース・サーバーのファイル・システムからターゲット・データベース・サーバーのファイル・システムにコピーする必要があります。

外部ルーチン・ライブラリー管理およびパフォーマンス

外部ルーチン・ライブラリー管理は、DB2 データベース・マネージャーが、ルーチンの使用法に従ってパフォーマンスを向上させる目的で外部ルーチン・ライブラリーを動的にキャッシュするため、ルーチンのパフォーマンスに影響を与えることがあります。外部ルーチンのパフォーマンスを最適なものにするには、以下の点を考慮してください。

- 各ライブラリー内のルーチン数を可能な限り少数に保ってください。少数の大きな外部ルーチン・ライブラリーよりも多数の小さい外部ルーチン・ライブラリーを用意したほうがよいと思われます。
- 一般的に一緒に呼び出されるルーチンのルーチン関数を、ソース・コード内で 1 つのグループにまとめます。コードを 1 つの外部ルーチン・ライブラリーにコンパイルすると、一般的に呼び出されるルーチンのエントリー・ポイントは互いに近くなるため、データベース・マネージャーはより優れたキャッシング・サポートを提供できるようになります。キャッシング・サポートが向上するのは、単一の外部ルーチン・ライブラリーを一度ロードして、複数の外部ルーチン関数をそのライブラリー内で呼び出すことによって効率が上がるためです。

C または C++ プログラミング言語でインプリメントされた外部ルーチンの場合、ライブラリーをロードする手間は、C ルーチンによって首尾一貫して使用されているライブラリーの場合は 1 回しかかかりません。ルーチンを最初に呼び出した後に、そのプロセスの同じスレッドから後続のすべての呼び出しを実行するときには、ルーチンのライブラリーを再ロードする必要がありません。

第 4 章 .NET 共通言語ランタイム (CLR) ルーチン

DB2 における共通言語ランタイム (CLR) ルーチンとは、.NET アセンブリーを外部コード本体として参照する外部ルーチンであり、`CREATE PROCEDURE` ステートメントまたは `CREATE FUNCTION` ステートメントの実行によって作成します。

CLR ルーチンのコンテキストで重要な用語は、以下のとおりです。

.NET Framework

CLR と .NET Framework クラス・ライブラリーから成る Microsoft アプリケーション開発環境。コード断片の開発と統合のための一貫したプログラミング環境を提供します。

共通言語ランタイム (CLR)

あらゆる .NET Framework アプリケーションのためのランタイム・インタープリター。

中間言語 (IL)

.NET Framework CLR によって解釈されるコンパイル済みバイトコードの一種。すべての .NET 互換言語のソース・コードが IL バイト・コードにコンパイルされます。

アセンブリー

IL バイト・コードを内容とするファイル。ライブラリーか実行可能ファイルのいずれかです。

CLR ルーチンは、IL アセンブリーにコンパイルできる言語であればどの言語でもインプリメントできます。例えば、Managed C++、C#、Visual Basic、J# などの言語があります。

CLR ルーチンを開発するには、ルーチンの基本と、CLR ルーチンにユニークなフィーチャーや特徴をあらかじめ理解しておくことが重要です。ルーチンと CLR ルーチンの詳細については、以下を参照してください。

- 3 ページの『ルーチン使用の利点』
- 121 ページの『.NET CLR ルーチンでの SQL データ・タイプ表記』
- 123 ページの『.NET CLR ルーチンのパラメーター』
- 126 ページの『.NET CLR プロシージャからの結果セットの戻り』
- 128 ページの『.NET CLR ルーチンに関する制約事項』
- 139 ページの『.NET CLR ルーチンに関連したエラー』

CLR ルーチンの開発は簡単です。CLR ルーチンの開発の方法に関する段階的な説明と完全な例については、以下を参照してください。

- 130 ページの『DB2 コマンド・ウィンドウから .NET CLR ルーチンを作成する』
- 143 ページの『C# .NET CLR プロシージャの例』
- 178 ページの『C# .NET CLR 関数の例』

.NET CLR 言語での外部ルーチン開発のサポート

.NET CLR 言語で外部ルーチンを開発し、それを正常に実行するには、いずれもサポートされているオペレーティング・システム、DB2 データベース・サーバーとクライアントのバージョン、および開発ソフトウェアを使用する必要があります。

.NET CLR 外部ルーチンは、IL アセンブリーにコンパイルできる言語であればどの言語でも Microsoft .NET Framework を使用してインプリメントできます。例えば、Managed C++、C#、Visual Basic、J# などの言語があります。

.NET CLR ルーチンは、以下のオペレーティング・システム上で開発できます。

- Windows 2000
- Windows XP (32 ビット版および 64 ビット版)
- Windows Server 2003 (32 ビット版および 64 ビット版)
- Windows Server 2008 (32 ビット版および 64 ビット版)

.NET CLR ルーチンの開発では、バージョン 9 以降のデータ・サーバー・クライアントがインストールされている必要があります。データベース・サーバーで、DB2 バージョン 9 以降のデータベース製品を実行している必要があります。

Microsoft .NET Framework ソフトウェアのサポートされているバージョンも DB2 データベース・サーバーと同じコンピューター上にインストールされている必要があります。Microsoft .NET Framework は、単独で、または Microsoft .NET Framework Software Development Kit の一部として入手可能です。

.NET CLR ルーチン開発のためのツール

ツールを使用して、DB2 データベースと対話する .NET CLR ルーチンの開発タスクの実行をより高速かつ簡単にできます。

.NET CLR ルーチンは、Microsoft Visual Studio .NET において以下の製品で有効なグラフィック・ツールを使用して開発することができます。

- IBM DB2 Development Add-In for Microsoft Visual Studio .NET 1.2

DB2 データベース・サーバー上での .NET CLR ルーチンの開発にも使用できる DB2 付属のコマンド行インターフェースには次のものがあります。

- DB2 コマンド行プロセッサ (DB2 CLP)
- DB2 コマンド・ウィンドウ

.NET CLR ルーチンの設計

.NET CLR ルーチンを設計するときは、一般的な外部ルーチンの設計上の考慮事項と .NET CLR 固有の設計上の考慮事項の両方を考慮に入れる必要があります。

.NET アプリケーション開発に関する知識と経験、および外部ルーチンに関する一般的な知識。以下のトピックは、必要な前提条件に関する情報の一部を提供しています。

外部ルーチンのフィーチャーおよび使用方法の詳細については、以下を参照してください。

- 24 ページの『外部ルーチンのインプリメンテーション』

.NET CLR ルーチンの特性の詳細については、以下を参照してください。

- 119 ページの『第 4 章 .NET 共通言語ランタイム (CLR) ルーチン』

すでに前提知識がある場合、組み込み SQL ルーチンの設計には、主に .NET CLR ルーチンの固有のフィーチャーおよび特性について習得することが関係しています。

- .NET CLR ルーチンでの SQL ステートメント実行のサポートを提供する組み込みアセンブリー (IBM.Data.DB2)
- .NET CLR ルーチンでサポートされている SQL データ・タイプ
- .NET CLR ルーチンのパラメーター
- .NET CLR ルーチンからの結果セットの戻り
- .NET CLR ルーチンのセキュリティーおよび実行制御モードの設定
- .NET CLR ルーチンに関する制約事項
- .NET CLR プロシージャからの結果セットの戻り

.NET CLR の特性について理解した後、129 ページの『.NET CLR ルーチンの作成』を参照してください。

.NET CLR ルーチンでの SQL データ・タイプ表記

.NET CLR ルーチンでは、ルーチン・パラメーターとして、SQL ステートメントの実行に使われるパラメーター値として、および変数として、SQL データ・タイプ (データ型) 値を参照できます。ただし、値のアクセス時または取得時にデータの切り捨てや喪失が発生しないよう、適切な IBM SQL データ・タイプ値、IBM Data Server Provider for .NET データ・タイプ値、および .NET Framework データ・タイプ値を使用する必要があります。

.NET CLR ルーチンの作成に使われる CREATE PROCEDURE または CREATE FUNCTION ステートメント内のルーチン・パラメーター指定では、DB2 SQL データ・タイプ値が使用されます。ほとんどの SQL データ・タイプをルーチン・パラメーターとして指定できますが、例外もあります。

SQL ステートメントの実行で使用されるパラメーター値を指定するには、IBM Data Server Provider for .NET オブジェクトを使用する必要があります。SQL ステートメントを表す DB2Command オブジェクトに追加されるパラメーターを表すために、DB2Parameter オブジェクトが使用されます。パラメーターのデータ・タイプ値を指定する際には、IBM.Data.DB2Types ネーム・スペース (名前空間) にある IBM Data Server Provider for .NET データ・タイプ値を使用する必要があります。

IBM.Data.DB2Types ネーム・スペースには、サポートされるそれぞれの IBM SQL データ・タイプを表すクラスおよび構造体があります。

SQL データ・タイプ値を一時的に保持する可能性のあるパラメーターとローカル変数に関しては、IBM.Data.DB2Types ネーム・スペースで定義された適切な IBM Data Server Provider for .NET データ・タイプを使用する必要があります。

注: dbinfo 構造は、パラメーターとして CLR の関数やプロシージャーに渡されます。 CLR UDF のスクラッチパッドと呼び出しタイプも、パラメーターとして CLR ルーチンに渡されます。これらのパラメーターに該当する CLR データ・タイプの詳細については、以下の関連トピックを参照してください。

- CLR ルーチンのパラメーター

次の表は、DB2Type データ・タイプ、DB2 データ・タイプ、 Informix® データ・タイプ、Microsoft .NET Framework タイプ、および DB2Types クラスと構造体の対応を示しています。

分類	DB2Types クラスと構造体	DB2Type データ・タイプ	DB2 データ・タイプ	Informix データ・タイプ	.NET データ・タイプ
Numeric	DB2Int16	SmallInt	SMALLINT	BOOLEAN, SMALLINT	Int16
	DB2Int32	Integer	INT	INTEGER、 INT、 SERIAL	Int32
	DB2Int64	BigInt	BIGINT	BIGINT, BIGSERIAL, INT8, SERIAL8	Int64
	DB2Real, DB2Real370	Real	REAL	REAL, SMALLFLOAT	Single
	DB2Double	Double	DOUBLE PRECISION	DECIMAL (≦31), DOUBLE PRECISION	Double
	DB2Double	Float	FLOAT	DECIMAL (32), FLOAT	Double
	DB2Decimal	Decimal	DECIMAL	MONEY	Decimal
	DB2DecimalFloat	DecimalFloat	DECFLOAT (16 34) ^{1,4}		Decimal
	DB2Decimal	Numeric	DECIMAL	DECIMAL (≦31), NUMERIC	Decimal
Date/Time	DB2Date	Date	DATE	DATETIME (日付精度)	Datetime
	DB2Time	Time	TIME	DATETIME (時刻精度)	TimeSpan
	DB2TimeStamp	Timestamp	TIMESTAMP	DATETIME (時刻と日付の精度)	DateTime
XML	DB2Xml	Xml ²	XML		Byte[]
文字データ	DB2String	Char	CHAR	CHAR	String
	DB2String	VarChar	VARCHAR	VARCHAR	String
	DB2String	LongVarChar ¹	LONG VARCHAR	LVARCHAR	String

1. これらのデータ・タイプは、DB2 .NET 共通言語ランタイム・ルーチンではパラメーターとしてサポートされていません。
2. DB2Type.Xml 型の DB2ParameterClass.ParameterName プロパティは、String、byte[]、DB2Xml、および XmlReader 型の変数を受け入れることができます。
3. これらのデータ・タイプは、DB2 for z/OS のみに該当します。
4. このデータ・タイプは DB2 for z/OS バージョン 9 以降のリリース、および DB2 for Linux, UNIX, and Windows バージョン 9.5 以降のリリースでのみサポートされます。

分類	DB2Types クラスと構造体	DB2Type データ・タイプ	DB2 データ・タイプ	Informix データ・タイプ	.NET データ・タイプ
バイナリー・データ	DB2Binary	Binary	CHAR FOR BIT DATA		Byte[]
	DB2Binary	Binary ³	BINARY		Byte[]
	DB2Binary	VarBinary ³	VARBINARY		Byte[]
	DB2Binary	LongVarBinary ¹	LONG VARCHAR FOR BIT DATA		Byte[]
グラフィック・データ	DB2String	Graphic	GRAPHIC		String
	DB2String	VarGraphic	VARGRAPHIC		String
	DB2String	LongVarGraphic ¹	LONG VARGRAPHIC		String
LOB データ	DB2Clob	Clob	CLOB	CLOB, TEXT	String
	DB2Blob	Blob	BLOB	BLOB, BYTE	Byte[]
	DB2Clob	DbClob	DBCLOB		String
Row ID	DB2RowId	RowId	ROWID		Byte[]

.NET CLR ルーチンのパラメーター

.NET CLR ルーチンのパラメーター宣言は、サポートされているいずれかのパラメーター・スタイルの要件と、ルーチンで使用している特定の .NET 言語のパラメーター・キーワードの要件を満たしている必要があります。ルーチンがスクラッチパッドを使用する場合や、dbinfo 構造を使用する場合や、PROGRAM TYPE MAIN パラメーター・インターフェースを使用する場合には、追加の考慮事項があります。このトピックでは、CLR パラメーターに関するすべての考慮事項を取り上げます。

CLR ルーチンでサポートされているパラメーター・スタイル

ルーチンのパラメーター・スタイルは、ルーチンの作成時にそのルーチンの CREATE ステートメントの EXTERNAL 節で指定する必要があります。外部 CLR ルーチン・コードのインプリメンテーションでは、そのパラメーター・スタイルを正確に反映しなければなりません。CLR ルーチンでは、以下の DB2 パラメーター・スタイルがサポートされています。

- SQL (プロシージャと関数に対応)
- GENERAL (プロシージャにのみ対応)
- GENERAL WITH NULLS (プロシージャにのみ対応)
- DB2SQL (プロシージャと関数に対応)

これらのパラメーター・スタイルの詳細については、以下を参照してください。

- 82 ページの『外部ルーチンのパラメーター・スタイル』

CLR ルーチン・パラメーターの NULL 標識

CLR ルーチンに対して選択したパラメーター・スタイルのパラメーターに NULL 標識を指定する必要がある場合は、パラメーター・スタイルが NULL 標識のベクトルを必要とするときに、NULL 標識を System.Int16 タイプ値または System.Int16[] 値として CLR ルーチンに渡します。

パラメーター・スタイル SQL のように、NULL 標識を特殊パラメーターとしてルーチンに渡すことが必要なパラメーター・スタイルの場合は、各パラメーターで 1 つの System.Int16 NULL 標識が必要になります。

.NET 言語の場合は、特殊パラメーターの前に、そのパラメーターを値によって渡すのか、参照によって渡すのかを示すキーワードを付ける必要があります。ルーチン・パラメーターに使用するのと同じキーワードを、関連した NULL 標識パラメーターで使用しなければなりません。引数を値によって渡すのか、参照によって渡すのかを示すキーワードについては、以下に詳しく取り上げます。

パラメーター・スタイル SQL や、他のサポートされているパラメーター・スタイルの詳細については、以下を参照してください。

- 82 ページの『外部ルーチンのパラメーター・スタイル』

CLR ルーチンのパラメーターを値によって渡すか、参照によって渡すか

中間言語 (IL) のバイト・コードにコンパイルする .NET 言語のルーチンの場合は、パラメーターを値によって渡すか、参照によって渡すか、入力専用パラメーターか、出力専用パラメーターか、といったパラメーターのプロパティを示すキーワードをパラメーターの前に置く必要があります。

パラメーター・キーワードは、それぞれの .NET 言語によって異なります。例えば、C# の場合、パラメーターを参照によって渡すことを示すパラメーター・キーワードは ref ですが、Visual Basic の場合は、byRef キーワードによって参照渡しのパラメーターであることを示します。ルーチンの CREATE ステートメントに指定する SQL パラメーターの使用法 (IN、OUT、INOUT) を示すために、キーワードを使用する必要があります。

DB2 ルーチンで .NET 言語ルーチン・パラメーターにパラメーター・キーワードを適用するときには以下の規則が適用されます。

- IN タイプ・パラメーターは、C# ではパラメーター・キーワードなしで宣言し、Visual Basic では byVal キーワードで宣言しなければなりません。
- INOUT タイプ・パラメーターは、参照渡しのパラメーターであることを示す言語固有のキーワードによって宣言しなければなりません。C# の場合、該当するキーワードは ref です。Visual Basic の場合、該当するキーワードは byRef です。
- OUT タイプ・パラメーターは、出力専用のパラメーターであることを示す言語固有のキーワードによって宣言しなければなりません。C# の場合は、out キーワードを使用します。Visual Basic の場合は、byRef キーワードによってパラメーターを宣言する必要があります。出力専用パラメーターには、ルーチンが呼び出し元に戻る前に値を代入する必要があります。ルーチンが出力専用パラメーターに値を代入しない場合は、.NET ルーチンのコンパイル時にエラーが発生します。

1 つの出力パラメーター language を戻すルーチンの C# のパラメーター・スタイル SQL のプロシージャのプロトタイプは、次のようになります。

```
public static void Counter (out String language,  
                           out Int16 languageNullInd,  
                           ref String sqlState,
```

```
String funcName,
String funcSpecName,
ref String sqlMsgString,
Byte[] scratchPad,
Int32 callType);
```

ここでは、出力パラメーター language に関連する追加の NULL 標識パラメーター languageNullInd と、SQLSTATE、ルーチン名、ルーチン固有名、オプションのユーザー定義 SQL エラー・メッセージを渡すパラメーターのために、パラメーター・スタイル SQL をインプリメントしています。また、パラメーターのパラメーター・キーワードを次のように指定しています。

- C# では、入力専用パラメーターにパラメーター・キーワードは必要ありません。
- C# では、'out' キーワードは、変数が出力パラメーター専用であり、その値が呼び出し元によって初期化されていないことを示します。
- C# では、'ref' キーワードは、パラメーターが呼び出し元によって初期化されており、ルーチンがオプションでこの値を変更できることを示します。

.NET 言語のパラメーター・キーワードについては、パラメーターの受け渡しに関する .NET 言語固有の資料を参照してください。

注: DB2 は、すべてのパラメーターに関するメモリーの割り振りを制御し、ルーチンとの間で受け渡しが行われるすべてのパラメーターへの CLR 参照を管理します。

プロシージャの結果セットのためのパラメーター・マーカーは不要

プロシージャのプロシージャ宣言内で、呼び出し元に戻される結果セットのためのパラメーター・マーカーは不要です。CLR ストアード・プロシージャ内部からクローズされないカーソル・ステートメントはすべて、その呼び出し元に結果セットとして戻されます。

CLR ルーチンの結果セットの詳細については、以下を参照してください。

- CLR プロシージャからの結果セットの戻り

CLR パラメーターとしての dbinfo 構造

CLR ルーチンでは、ルーチンとの間で追加のデータベース情報パラメーターを受け渡すための dbinfo 構造を、IL の dbinfo クラスの使用によってサポートしています。このクラスには、ストリングに関連した長さフィールドを除いて、C 言語の sqludf_dbinfo 構造にあるすべてのエレメントが含まれています。各ストリングの長さは、.NET 言語の各ストリングの Length プロパティによって検出できます。

dbinfo クラスにアクセスするには、単に、対象のルーチンが含まれるファイルに IBM.Data.DB2 アセンブリを含め、タイプ sqludf_dbinfo のパラメーターを、ルーチンのシグニチャーの中の、使用されているパラメーター・スタイルによって指定される位置に追加します。

CLR パラメーターとしての UDF スクラッチパッド

ユーザー定義関数のためのスクラッチパッドを要求する場合は、指定のサイズの System.Byte[] パラメーターとしてスクラッチパッドをルーチンに渡します。

CLR UDF の呼び出しタイプ・パラメーターまたは最終呼び出しパラメーター

最終呼び出しパラメーターまたは表関数を要求したユーザー定義関数の場合は、呼び出しタイプ・パラメーターを System.Int32 データ・タイプとしてルーチンに渡します。

CLR プロシージャでサポートされている PROGRAM TYPE MAIN

.NET CLR プロシージャでは、プログラム・タイプ MAIN がサポートされています。プログラム・タイプ MAIN を使用するプロシージャを定義する場合は、以下のシグニチャーが必要です。

```
void functionname(Int32 NumParams, Object[] Params)
```

.NET CLR プロシージャからの結果セットの戻り

呼び出し側のルーチンまたはアプリケーションに結果セットを戻す CLR プロシージャを開発できます。CLR 関数 (UDF) から結果セットを戻すことはできません。

結果セットの .NET 表現は、DB2DataReader オブジェクトです。このオブジェクトは、DB2Command オブジェクトのさまざまな実行呼び出しのいずれかから戻すことができます。戻すことができるのは、プロシージャの戻りの前に Close() メソッドが明示的に呼び出されなかった DB2DataReader オブジェクトです。結果セットが呼び出し元に戻される順序は、DB2DataReader オブジェクトがインスタンス化された順序と同じです。結果セットを戻すために、関数定義で追加のパラメーターを指定する必要はありません。

CLR ルーチンの作成方法を理解しておく、CLR プロシージャから結果を戻すための以下の手順のステップを容易に理解できます。

- .NET CLR ルーチンの作成

CLR プロシージャから結果セットを戻すには、次のようにします。

1. CLR ルーチンの CREATE PROCEDURE ステートメントでは、他にも適切な節を指定する中で、特に DYNAMIC RESULT SETS 節に、プロシージャによって戻される結果セットの数と等しい値を指定しなければなりません。
2. プロシージャ宣言内で、呼び出し元に戻される結果セットのためのパラメーター・マーカーは不要です。
3. CLR ルーチンの .NET 言語インプリメンテーションでは、DB2Connection オブジェクト、DB2Command オブジェクト、DB2Transaction オブジェクトを作成します。DB2Transaction オブジェクトは、データベース・トランザクションのロールバックとコミットを担当します。
4. DB2Transaction オブジェクトに対する DB2Command オブジェクトの Transaction プロパティを初期化します。
5. 戻したい結果セットを定義する DB2Command オブジェクトの CommandText プロパティにストリング照会を割り当てます。
6. DB2DataReader をインスタンス化し、DB2Command オブジェクトの ExecuteReader メソッドの呼び出しの結果をそのインスタンスに割り当てます。照会の結果セットは、DB2DataReader オブジェクトに組み込まれます。

7. DB2DataReader オブジェクトの Close() メソッドは、プロシージャが呼び出し元に戻る前に実行してはなりません。オープンしている DB2DataReader オブジェクトが、結果セットとして呼び出し元に戻されます。

プロシージャの戻り時に複数の DB2DataReader がオープンしたままになっていると、それぞれの DB2DataReader が作成順に呼び出し元に戻されます。

CREATE PROCEDURE ステートメントで指定した数の結果セットだけが呼び出し元に戻されます。

8. .NET CLR 言語プロシージャをコンパイルし、CREATE PROCEDURE ステートメントの EXTERNAL 節で指定するロケーションにアセンブリーをインストールします。CLR プロシージャの CREATE PROCEDURE ステートメントをまだ実行していない場合は、実行してください。
9. CLR プロシージャ・アセンブリーを適切な場所にインストールして、CREATE PROCEDURE ステートメントを正常に実行したなら、CALL ステートメントでプロシージャを呼び出し、結果セットが呼び出し元に戻されるのを確認してください。

CLR ルーチンのセキュリティおよび実行モード

データベース管理者またはアプリケーション開発者であれば、DB2 外部ルーチンに関連付けられているアセンブリーを、実行時のルーチンのアクションを制限する厄介な改ざんから保護することもできます。DB2 .NET 共通言語ランタイム (CLR) ルーチンは、実行制御モードの指定をサポートします。この指定により、実行時にルーチンが行えるアクションのタイプを識別できます。実行時に DB2 は、ルーチンがその指定された実行制御モードの有効範囲を越えてアクションを実行しようとしていないかどうかを検出できます。これはアセンブリーが破損していないかどうかを判別するときに役に立ちます。

CLR ルーチンの実行制御モードを設定するには、ルーチンの CREATE ステートメントにオプションの EXECUTION CONTROL 節を指定します。有効なモードは以下のとおりです。

- SAFE
- FILEREAD
- FILEWRITE
- NETWORK
- UNSAFE

既存の CLR ルーチンの実行制御モードに対して変更を加えるには、ALTER PROCEDURE または ALTER FUNCTION ステートメントを実行します。

CLR ルーチンで EXECUTION CONTROL 節が指定されない場合、デフォルトで CLR ルーチンは、制限度の最も高い実行制御モードである SAFE モードを使って実行されます。この実行制御モードを使って作成されるルーチンは、データベース・マネージャーによって制御されるリソースにのみアクセスできます。これより制限度の低い実行制御モードのルーチンでは、ファイルへのアクセス (FILEREAD または FILEWRITE)、または Web ページへのアクセスなどのネットワーク操作の実行 (NETWORK) が可能です。実行制御モード UNSAFE は、ルーチンの振る舞い

に制限を課さないことを指定します。 UNSAFE 実行制御モードを使って定義されるルーチンは、バイナリー・コードを実行することができます。

これらのモードは許容アクションの階層を表し、階層内の高いレベルのモードは、それよりも低いレベルの許容アクションを包含します。例えば、実行制御モードが NETWORK のルーチンは、インターネット上の Web ページへのアクセス、ファイルの読み取りおよび書き込み、およびデータベース・マネージャーが制御しているリソースへのアクセスが可能です。できる限り制限度の高い実行制御モードを使用することと、UNSAFE モードの使用を避けることをお勧めします。

CLR ルーチンがその実行制御モードの有効範囲を越えてアクションを試行していることを DB2 が実行時に検出した場合、DB2 はエラー (SQLSTATE 38501) を戻します。

EXECUTION CONTROL 節は、LANGUAGE CLR ルーチンに対してのみ指定することができます。 EXECUTION CONTROL 節の適用度の有効範囲は .NET CLR ルーチンそのものに限られ、他のいかなるルーチンにも (EXECUTION CONTROL 節によって呼び出せるとしても) 拡張されません。

サポートされる実行制御モードの詳細については、該当するルーチン・タイプの CREATE ステートメントの構文を参照してください。

.NET CLR ルーチンに関する制約事項

すべての外部ルーチンまたは特定のルーチン・クラス (プロシージャや UDF) のインプリメンテーションに当てはまる一般的な制約事項は、 CLR ルーチンにも当てはまります。また、CLR ルーチンだけに該当する制約事項もいくつかあります。ここでは、その種の制約事項を取り上げます。

LANGUAGE CLR 節付きの CREATE METHOD ステートメントはサポートされていない

CLR アセンブリーを参照する DB2 構造化タイプの外部メソッドは作成できません。 LANGUAGE 節に CLR という値を指定した CREATE METHOD ステートメントの使用は、サポートされていません。

CLR プロシージャは NOT FENCED プロシージャとしてインプリメントできない

CLR プロシージャを unfenced プロシージャとして実行することはできません。 CLR プロシージャを作成する CREATE PROCEDURE ステートメントでは、 NOT FENCED 節を指定できません。

EXECUTION CONTROL 節がルーチンに含まれているロジックを制限する

EXECUTION CONTROL 節および関連値は、.NET CLR ルーチンで実行できるロジックおよび操作のタイプを決定します。デフォルトでは、EXECUTION CONTROL 節の値は SAFE に設定されます。ファイルの読み取り、ファイルへの書き込みを行ったり、インターネットにアクセスするルーチン・ロジックの場合、EXECUTION

CONTROL 節にはデフォルト以外の値および制約の少ない値を指定する必要があります。

CLR ルーチンの 10 進数の最大精度は 29、最大スケールは 28

DB2 の DECIMAL データ・タイプは、31 桁の精度と 28 桁のスケールで表現します。 .NET CLR の System.Decimal データ・タイプは、29 桁の精度と 28 桁のスケールに制限されます。したがって、DB2 の外部 CLR ルーチンで、 $(2^{96})-1$ (29 桁の精度と 28 桁のスケールで表せる最大値) より大きい値を持つ System.Decimal データ・タイプに値を割り当てないでください。こうした割り当てが行われると、DB2 で実行時エラー (SQLSTATE 22003、SQLCODE -413) が発生します。ルーチンの CREATE ステートメントの実行時にスケールが 28 桁より大きい DECIMAL データ・タイプ・パラメーターが定義されていると、DB2 でエラー (SQLSTATE 42613、SQLCODE -628) が発生します。

DB2 でサポートされている精度とスケールの最大桁数を利用した 10 進数値をルーチンで操作する必要がある場合は、別のプログラム言語 (Java など) で外部ルーチンをインプリメントしてください。

CLR ルーチンでサポートされていないデータ・タイプ

CLR ルーチンでは、以下の DB2 SQL データ・タイプがサポートされていません。

- LONG VARCHAR
- LONG VARCHAR FOR BIT DATA
- LONG GRAPHIC
- ROWID

64 ビットのインスタンス上での 32 ビットの CLR ルーチンの実行

現時点で、64 ビットの実行環境には .NET Framework をインストールできないので、CLR ルーチンを 64 ビットのインスタンスで実行することはできません。

.NET CLR はセキュリティー・プラグインのインプリメントに非対応

セキュリティー・プラグイン・ライブラリーのソース・コードのコンパイルおよびリンクにおいて、.NET CLR はサポートされません。

.NET CLR ルーチンの作成

.NET CLR ルーチンの作成は、以下から構成されます。

- DB2 データベース・サーバーでルーチンを定義する CREATE ステートメントの実行
- ルーチン定義に対応するルーチン・インプリメンテーションの開発

.NET CLR ルーチンを作成する方法は、以下に従ってください。

- DB2 Database Development Add-In for Visual Studio .NET 1.2 のグラフィック・ツールを使用する
- DB2 コマンド・ウィンドウを使用する

一般に、最も簡単な方法は、DB2 Database Development Add-In for Visual Studio .NET 1.2 を使用して .NET CLR ルーチンを作成することです。これが使用できない場合は、DB2 コマンド・ウィンドウがコマンド行インターフェースを介して同様のサポートを提供します。

前提条件

- 119 ページの『第 4 章 .NET 共通言語ランタイム (CLR) ルーチン』を確認します。
- DB2 バージョン 9 サーバー (インスタンスおよびデータベースを含む) にアクセスしていることを確認します。
- オペレーティング・システムのバージョン・レベルが、DB2 データベース製品でサポートされているものであることを確認します。
- Microsoft .NET 開発ソフトウェアのバージョン・レベルが、.NET CLR ルーチン開発にサポートされるものであることを確認します。
- CREATE PROCEDURE または CREATE FUNCTION ステートメントを実行する権限。

CLR ルーチンに関連した制約事項のリストについては、以下を参照してください。

- 128 ページの『.NET CLR ルーチンに関する制約事項』

以下のいずれかのインターフェースからの .NET CLR ルーチンの作成:

- IBM DB2 Development Add-In for Microsoft Visual Studio .NET 1.2 もインストールされる場合には、Visual Studio .NET。 Add-In がインストールされているなら、DB2 データベース・サーバーで動作する .NET CLR ルーチンを作成するために、Visual Studio .NET に統合されたグラフィック・ツールを使用できます。
- DB2 コマンド・ウィンドウ

DB2 コマンド・ウィンドウから .NET CLR ルーチンを作成するには、以下を参照してください。

- 『DB2 コマンド・ウィンドウから .NET CLR ルーチンを作成する』

DB2 コマンド・ウィンドウから .NET CLR ルーチンを作成する

中間言語アセンブリを参照するプロシージャと関数の作成方法は、他の外部ルーチンの場合と同じです。 .NET 言語で外部ルーチンをインプリメントするのは、以下のような場合です。

- データベースにアクセスするルーチンや、データベースの外部でアクションを実行するルーチンに複雑なロジックをカプセル化したい場合。
- 複数のアプリケーション、CLP、他のルーチン (プロシージャ、関数 (UDF)、メソッド)、トリガーのいずれかから、カプセル化されたロジックを呼び出す必要がある場合。
- そのロジックのコーディングに .NET 言語が最も使いやすいと感じる場合。

前提条件

- CLR ルーチンのインプリメンテーションに関する知識。 CLR ルーチンの概要や CLR のフィーチャーについては、以下を参照してください。
 - 119 ページの『第 4 章 .NET 共通言語ランタイム (CLR) ルーチン』

- データベース・サーバーが Microsoft .NET Framework をサポートする Windows オペレーティング・システムを実行していること。
- サポートされているバージョンの Microsoft .NET Framework ソフトウェアがサーバー上にインストールされている必要があります。 .NET Framework は、単独で、または Microsoft .NET Framework Software Development Kit の一部として入手可能です。
- サポートされている DB2 データベース製品、または IBM Data Server Client をインストールする必要があります。「DB2 サーバー機能 インストール」の『DB2 データベース製品のインストール要件』を参照してください。
- 外部ルーチンを作成する CREATE ステートメントの実行権限。 CREATE PROCEDURE ステートメントまたは CREATE FUNCTION ステートメントの実行に必要な特権については、該当するステートメントの詳細を参照してください。

CLR ルーチンに関連した制約事項のリストについては、以下を参照してください。

- 128 ページの『.NET CLR ルーチンに関する制約事項』

手順

1. CLR でサポートされている言語でルーチン・ロジックをコーディングします。
 - .NET CLR ルーチンの概要や .NET CLR ルーチンのフィーチャーについては、「前提条件」のセクションで示したトピックを参照してください。
 - ルーチンで SQL を実行する場合は、IBM.Data.DB2 アセンブリーを使用するか、インポートします。
 - DB2 SQL データ・タイプにマップするデータ・タイプを使用して、ホスト変数とパラメーターを正しく宣言します。 DB2 データ・タイプと .NET データ・タイプ間のマッピングについては、以下を参照してください。
 - 121 ページの『.NET CLR ルーチンでの SQL データ・タイプ表記』
 - DB2 がサポートするパラメーター・スタイルのいずれかを使用し、.NET CLR ルーチンのパラメーター要件に従って、パラメーターとパラメーターの NULL 標識を宣言する必要があります。また、UDF のスクラッチパッドと DBINFO クラスをパラメーターとして CLR ルーチンに渡します。パラメーターとプロトタイプ宣言の詳細については、以下を参照してください。
 - 123 ページの『.NET CLR ルーチンのパラメーター』
 - ルーチンがプロシージャで、ルーチンの呼び出し元に結果セットを戻したい場合、結果セット用のパラメーターは必要ありません。 CLR ルーチンから結果セットを戻す方法の詳細については、以下を参照してください。
 - 126 ページの『.NET CLR プロシージャからの結果セットの戻り』
 - 必要に応じてルーチンの戻り値を設定します。 CLR スカラー関数の場合は、値を戻す前に戻り値を設定する必要があります。 CLR 表関数の戻りコードは、表関数の呼び出しごとに出力パラメーターとして指定しなければなりません。 CLR プロシージャは戻り値を戻しません。
2. CLR で実行できる中間言語 (IL) アセンブリーにコードをビルドします。 DB2 にアクセスする CLR .NET ルーチンの作成方法については、以下のトピックを参照してください。

- 「*ADO.NET* および *OLE DB* アプリケーションの開発」内の『Common Language Runtime (CLR) .NET ルーチンの構築』
3. そのアセンブリーをデータベース・サーバー上の *DB2 function* ディレクトリーにコピーします。 *DB2* ルーチンに関連したアセンブリーまたはライブラリーは、 *function* ディレクトリーに保管することをお勧めします。 *function* ディレクトリーの詳細については、 *CREATE PROCEDURE* ステートメントまたは *CREATE FUNCTION* ステートメントのいずれかの *EXTERNAL* 節を参照してください。

アセンブリーをサーバー上の別のディレクトリーにコピーすることもできますが、ルーチンを正常に呼び出すには、アセンブリーの完全修飾パス名をメモしておく必要があります。次のステップでこれが必要になるからです。

4. 該当するルーチン・タイプの *SQL* 言語 *CREATE* ステートメント (*CREATE PROCEDURE* または *CREATE FUNCTION*) を動的または静的に実行します。
 - *LANGUAGE* 節に、*CLR* という値を指定します。
 - *PARAMETER STYLE* 節に、ルーチン・コードでインプリメントした有効なパラメーター・スタイルの名前を指定します。
 - *EXTERNAL* 節に、ルーチンと関連したアセンブリーの名前を指定します。そのためには、以下のいずれかの値を使用します。
 - ルーチン・アセンブリーの完全修飾パス名。
 - *function* ディレクトリーを基準にしたルーチン・アセンブリーの相対パス名。

EXTERNAL 節にライブラリーの完全修飾パス名または相対パス名を指定しない場合、*DB2* はデフォルトで、*function* ディレクトリー内でアセンブリーの名前を探します。

CREATE ステートメントを実行した後、*EXTERNAL* 節に指定したアセンブリーを *DB2* が見つけられない場合は、理由コード 1 のエラー (*SQLCODE* -20282) が発生します。

- *DYNAMIC RESULT SETS* 節に、ルーチンによって戻される結果セットの最大数と等しい整数値を指定します。
- *CLR* プロシージャに *NOT FENCED* 節を指定することはできません。*CLR* プロシージャはデフォルトで、*FENCED* プロシージャとして実行されます。

.NET CLR ルーチン・コードのビルド

.NET CLR ルーチンのインプリメンテーション・コードを作成したなら、ルーチン・アセンブリーをデプロイしてルーチンを呼び出せるようにするには、その前にインプリメンテーション・コードをビルドする必要があります。 .NET CLR ルーチンのビルドに必要なステップは、外部ルーチンのビルドに必要なステップに似ていますが、いくつかの点で異なります。

.NET CLR ルーチンをビルドするには、次の 3 とおりの方法があります。

- *DB2 Database Development Add-In for Visual Studio .NET 1.2* のグラフィック・ツールを使用する

- DB2 サンプル・バッチ・ファイルを使用する
- DB2 コマンド・ウィンドウからコマンドを入力する

ルーチン用の DB2 サンプル・ビルド・スクリプトおよびバッチ・ファイルは、デフォルトでサポートされるコンパイラーを使用する特定のオペレーティング・システムのためのユーザー作成ルーチンだけでなく、DB2 サンプル・ルーチン (プロシージャおよびユーザー定義関数) 用に設計されています。

C# および Visual Basic 用の DB2 サンプル・ビルド・スクリプトおよびバッチ・ファイルには個別セットがあります。一般に、.NET CLR ルーチンをビルドするには、グラフィック・ツールまたはビルド・スクリプト (必要に応じて容易に変更できる) を使用するのが最も簡単な方法ですが、DB2 コマンド・ウィンドウからルーチンをビルドする方法も知っておくと、いろいろと役に立ちます。

サンプル・ビルド・スクリプトを使った .NET 共通言語ランタイム (CLR) ルーチン・コードのビルド

.NET 共通言語ランタイム (CLR) ルーチンのソース・コードのビルドは、.NET CLR ルーチン作成の副次作業です。この作業は DB2 サンプル・バッチ・ファイルを使用することにより、迅速かつ簡単に行うことができます。サンプル・ビルド・スクリプトは、SQL ステートメントを含むソース・コードにも含まないソース・コードにも使用できます。ビルド・スクリプトはビルド済みアセンブリーの function ディレクトリーへのコンパイル、リンク、およびデプロイメントを処理します。

別の方法として、Visual Studio .NET によって .NET CLR ルーチン・コードのビルド作業を単純化するか、または DB2 サンプル・ビルド・スクリプトのステップを手動で行うことができます。次のトピックを参照してください。

- Visual Studio .NET での .NET 共通言語ランタイム (CLR) ルーチンのビルド
- DB2 コマンド・ウィンドウによる .NET 共通言語ランタイム (CLR) ルーチンの作成

C# および Visual Basic .NET CLR ルーチンをビルドするためのプログラミング言語固有のサンプル・ビルド・スクリプトには bldrtn という名前があります。これを使用してビルドできるサンプル・プログラムと一緒に、次の DB2 ディレクトリーに置かれています。

- C: の場合 sqllib/samples/cs/
- C++: の場合 sqllib/samples/vb/

bldrtn スクリプトを使用して、プロシージャとユーザー定義関数の両方を含むソース・コード・ファイルをビルドすることができます。このスクリプトは、以下の処理を行います。

- ユーザーが指定したデータベースとの接続を確立する
- ソース・コードをコンパイルしてリンクし、.DLL というファイル接尾部を持つアセンブリーを生成する
- アセンブリーをデータベース・サーバーの DB2 function ディレクトリーにコピーする

bldrtn スクリプトは、次の 2 つの引数を受け入れます。

- ソース・コード・ファイルの名前 (ファイル接尾部なし)

- 接続が確立される先のデータベースの名前

データベース・パラメーターはオプションです。データベース名を指定しない場合は、プログラムはデフォルトの `sample` データベースを使用します。ルーチンはデータベースがあるインスタンスと同じインスタンス上で構築する必要があるため、ユーザー ID とパスワードのための引数は不要です。

前提条件

- 必要な .NET CLR ルーチンのオペレーティング・システムおよび開発ソフトウェアの前提要件を満たしている必要があります。『.NET CLR ルーチンの開発のサポート』を参照してください。
- 1 つ以上のルーチン・インプリメンテーションを含むソース・コード・ファイル
- 現行 DB2 インスタンス内の、ルーチンが作成されるデータベースの名前。

手順

1 つ以上のルーチン・コード・インプリメンテーションを含むソース・コード・ファイルをビルドするには、以下のステップを実行します。

1. DB2 コマンド・ウィンドウをオープンする。
2. ソース・コード・ファイルを `bldrtn` スクリプト・ファイルと同じディレクトリーにコピーする。
3. サンプル・データベースにルーチンを作成する場合は、次のように、ビルド・スクリプト名に続いて、ソース・コード・ファイルの名前 (`.cs` または `.vb` ファイル拡張子なし) を入力する。

```
bldrtn <file-name>
```

別のデータベースにルーチンを作成する場合は、次のように、ビルド・スクリプト名、ソース・コード・ファイル名 (ファイル拡張子なし)、データベース名を入力します。

```
bldrtn <file-name> <database-name>
```

スクリプトによってソース・コードのコンパイルとリンクが行われ、アセンブリーが生成されます。次に、スクリプトによってアセンブリーがデータベース・サーバーの `function` ディレクトリーにコピーされます。

4. ルーチン・インプリメンテーションを含むソース・コード・ファイルをビルドしたのが今回が初めてでない場合は、データベースを停止してから再始動することにより、DB2 で新しいバージョンの共用ライブラリーが使用されるようにする。これを行うには、コマンド行で `db2stop` に続けて `db2start` を入力します。

ルーチン共用ライブラリーのビルドとデータベース・サーバー上の `function` ディレクトリーへのデプロイが正常に終了したなら、C および C++ ルーチンの作成作業に関連したステップを完了する必要があります。

.NET CLR ルーチンの作成には、ソース・コード・ファイルにインプリメントされたルーチンごとに `CREATE` ステートメントを実行するというステップが含まれます。ルーチンの作成が完了すると、ルーチンを呼び出すことができます。

DB2 コマンド・ウィンドウからの .NET 共通言語ランタイム (CLR) ルーチン・コードのビルド

.NET CLR ルーチンのソース・コードのビルドは、.NET CLR ルーチン作成の副次作業です。この作業は、DB2 コマンド・ウィンドウから手動で行うことができます。ルーチン・コードに SQL ステートメントが含まれているかどうかにかかわらず、同じ手順を進めることができます。この作業ステップには、.NET CLR がサポートするプログラミング言語で作成されたソース・コードを、.DLL ファイル接尾部を持つアセンブリーにコンパイルする処理が含まれます。

別の方法として、Visual Studio .NET で行う方法と、DB2 サンプル・ビルド・スクリプトを使用する方法により、.NET CLR ルーチン・コードをビルド作業が簡単になります。次のトピックを参照してください。

- Visual Studio .NET での .NET 共通言語ランタイム (CLR) ルーチンのビルド
- サンプル・ビルド・スクリプトを使った .NET 共通言語ランタイム (CLR) ルーチンのビルド

前提条件

- 必須のオペレーティング・システムと .NET CLR ルーチン開発のソフトウェア前提条件が満たされていること。『.NET CLR ルーチンの開発のサポート』を参照してください。
- 1 つ以上の .NET CLR ルーチン・インプリメンテーションを含む、サポートされている .NET CLR プログラミング言語で作成されたソース・コード。
- 現行 DB2 インスタンス内の、ルーチンが作成されるデータベースの名前。
- .NET CLR ルーチンをビルドするのに必要な、オペレーティング環境固有のコンパイルおよびリンク・オプション。

1 つ以上の .NET CLR ルーチン・コード・インプリメンテーションを含むソース・コード・ファイルをビルドするには、以下のステップを実行します。各ステップを示した例が後に続きます。

1. DB2 コマンド・ウィンドウをオープンする。
2. ソース・コード・ファイルを含むディレクトリーにナビゲートする。
3. ルーチンが作成されるデータベースとの接続を確立する。
4. ソース・コード・ファイルをコンパイルする。
5. ソース・コード・ファイルをリンクして共用ライブラリーを生成する。ここでは、DB2 固有のコンパイルおよびリンク・オプションを使用する必要があります。
6. .DLL ファイル接尾部を持つアセンブリー・ファイルを、データベース・サーバーの DB2 function ディレクトリーにコピーする。
7. ルーチン・インプリメンテーションを含むソース・コード・ファイルをビルドしたのが今回が初めてでない場合は、データベースを停止してから再始動することにより、DB2 で新しいバージョンの共用ライブラリーが使用されるようになる。これを行うには、db2stop コマンドに続けて db2start コマンドを実行します。

ルーチン・ライブラリーのビルドとデプロイが正常に終了したなら、.NET CLR ルーチンの作成作業に関連したステップを完了する必要があります。 .NET CLR ルー

チンの作成には、ソース・コード・ファイルにインプリメントされたルーチンごとに CREATE ステートメントを実行するというステップが含まれます。ルーチンを呼び出せるようにするには、このステップも完了しなければなりません。

例

以下は、.NET CLR ソース・コード・ファイルの再ビルドを示した例です。ルーチン・インプリメンテーションを含む、myVBfile.vb という名前の Visual Basic コード・ファイルと myCSfile.cs という名前の C# コード・ファイルの、両方の場合のステップを示しています。ルーチンは Windows 2000 オペレーティング・システム上でビルドされ、Microsoft .NET Framework 1.1 を使用して 64 ビット・アセンブリーが生成されます。

1. DB2 コマンド・ウィンドウをオープンする。
2. ソース・コード・ファイルを含むディレクトリーにナビゲートする。
3. ルーチンが作成されるデータベースとの接続を確立する。

```
db2 connect to <database-name>
```
4. コンパイルとリンクの推奨オプションを使用してソース・コード・ファイルをコンパイルする (\$DB2PATH は DB2 インスタンスのインストール・パスです。コマンドを実行する前にこの値を置き換えてください)。

```
C# example
=====
csc /out:myCSfile.dll /target:library
    /reference:$DB2PATH%bin%netf11%IBM.Data.DB2.dll myCSfile.cs
```

```
Visual Basic example
=====
vbc /target:library /libpath:$DB2PATH%bin%netf11
    /reference:$DB2PATH%bin%netf11%IBM.Data.DB2.dll
    /reference:System.dll
    /reference:System.Data.dll myVBfile.vb
```

エラーがある場合は、コンパイラーによって出力が生成されます。このステップで、myfile.exp という名前のエクスポート・ファイルが生成されます。

5. 共用ライブラリーをデータベース・サーバーの DB2 function ディレクトリーにコピーする。

```
C# example
=====
rm -f ~HOME/sql1lib/function/myCSfile.DLL
cp myCSfile $HOME/sql1lib/function/myCSfile.DLL
```

```
Visual Basic example
=====
rm -f ~HOME/sql1lib/function/myVBfile.DLL
cp myVBfile $HOME/sql1lib/function/myVBfile.DLL
```

このステップで、DB2 がルーチン・ライブラリーを探すデフォルト・ディレクトリーにルーチン・ライブラリーが置かれます。ルーチン・ライブラリーのデプロイについての詳細は、.NET CLR ルーチンの作成に関するトピックを参照してください。

6. これは以前にビルドされたルーチン・ソース・コード・ファイルの再ビルドなので、データベースを停止して再始動する。

```
db2stop
db2start
```

.NET CLR ルーチンのビルドは一般に、オペレーティング環境固有のサンプル・ビルド・スクリプトを使用して行うのが最も簡単です。このスクリプトは、コマンド行からルーチンをビルドする方法の参考として使用できます。

CLR .NET ルーチンのコンパイルとリンクのオプション

以下は、`samples¥.NET¥cs¥bldrtn.bat` および `samples¥.NET¥vb¥bldrtn.bat` バッチ・ファイルに示されているように、Windows 上で Microsoft Visual Basic .NET コンパイラーまたは Microsoft C# コンパイラーのどちらかを使用して、Common Language Runtime (CLR) .NET ルーチンを構築するのに勧められるコンパイルとリンクのオプションです。

bldrtn のコンパイルとリンクのオプション	
Microsoft C# コンパイラーを使用したコンパイルとリンクのオプション	
csc	Microsoft C# コンパイラー
/out:%1.d11 /target:library	ダイナミック・リンク・ライブラリーをストアード・プロシージャのアセンブリ DLL として出力します。
/debug	デバッガーを使用します。
/lib: "%DB2PATH%¥bin¥netf20¥	.NET Framework バージョン 2.0 のライブラリー・パスを使用します。 アプリケーション用にサポートされている .NET Framework のバージョンはいくつかあります。バージョン 2.0、バージョン 3.0、およびバージョン 3.5 です。それぞれにダイナミック・リンク・ライブラリーがあります。 .NET Framework バージョン 1.1 の場合には "%DB2PATH%¥bin¥netf11 サブディレクトリーを使用します。 .NET Framework バージョン 2.0、3.0、および 3.5 の場合には "%DB2PATH%¥bin¥netf20 サブディレクトリーを使用します。
/reference:IBM.Data.DB2.d11	IBM Data Server Provider for .NET の DB2 ダイナミック・リンク・ライブラリーを使用します。
他のコンパイラー・オプションについては、コンパイラーの資料をご覧ください。	

Microsoft Visual Basic .NET コンパイラーを使用したコンパイルとリンクのオプション

vbc Microsoft Visual Basic .NET コンパイラー。

/out:%1.d11 /target:library

ダイナミック・リンク・ライブラリーをストアード・プロシージャのアセンブリ DLL として出力します。

/debug デバッガーを使用します。

/libpath:"%DB2PATH%\%bin%netf20%

.NET Framework バージョン 2.0 のライブラリー・パスを使用します。

アプリケーション用にサポートされている .NET Framework のバージョンはいくつかあります。バージョン 2.0、バージョン 3.0、およびバージョン 3.5 です。それぞれにダイナミック・リンク・ライブラリーがあります。 .NET Framework バージョン 1.1 の場合には "%DB2PATH%\%bin%netf11" サブディレクトリーを使用します。 .NET Framework バージョン 2.0、3.0、および 3.5 の場合には "%DB2PATH%\%bin%netf20" サブディレクトリーを使用します。

/reference:IBM.Data.DB2.d11

IBM Data Server Provider for .NET の DB2 ダイナミック・リンク・ライブラリーを使用します。

/reference:System.d11

Microsoft Windows の System ダイナミック・リンク・ライブラリーを参照します。

/reference:System.Data.d11

Microsoft Windows の System Data ダイナミック・リンク・ライブラリーを参照します。

他のコンパイラー・オプションについては、コンパイラーの資料をご覧ください。

.NET CLR ルーチンのデバッグ

ルーチンの作成やルーチンの呼び出しに失敗する場合、またはルーチンの呼び出し時にそれが期待どおりに動作または実行しない場合は、.NET CLR ルーチンのデバッグが必要となる可能性があります。

.NET CLR ルーチンをデバッグするときには、以下を考慮に入れてください。

- .NET CLR ルーチン開発用にサポートされるオペレーティング・システムが使用されていることを確認します。
- .NET CLR ルーチン開発用にサポートされる DB2 データベース・サーバーと DB2 クライアントの両方が使用されていることを確認します。
- サポートされる Microsoft .NET Framework 開発ソフトウェアが使用されていることを確認します。
- ルーチンの作成が失敗した場合、以下のようになります。
 - ユーザーが CREATE PROCEDURE または CREATE FUNCTION ステートメントを実行するのに必要な権限および特権を持っていることを確認します。
- ルーチンの呼び出しが失敗した場合、以下のようになります。

- ユーザーがルーチンを実行するための権限を持っていることを確認します。エラー (SQLCODE -551, SQLSTATE 42501) が発生する場合、呼び出し側に、ルーチンの EXECUTE 特権が付与されていないことが原因である可能性が高いと言えます。この特権を付与できるのは、SECADM 特権、ACCESSCTRL 特権を持つユーザーか、ルーチンに関する EXECUTE WITH GRANT OPTION 特権を持つユーザーです。
- CREATE ステートメントで使用されるルーチン用のルーチン・パラメーター・シグニチャーが、ルーチン・インプリメンテーションのルーチン・パラメーター・シグニチャーと一致することを確認します。
- ルーチン・インプリメンテーションで使用されるデータ・タイプが、CREATE ステートメントのルーチン・パラメーター・シグニチャーで指定されたデータ・タイプと互換性があることを確認します。
- ルーチン・インプリメンテーションでは、パラメーターを (値別または参照別に) 受け渡さなければならないメソッドを示すために使用される .NET CLR 言語固有キーワードが有効であることを確認します。
- CREATE PROCEDURE または CREATE FUNCTION ステートメントの EXTERNAL 節で指定される値が、ルーチン・インプリメンテーションを含む .NET CLR アセンブリが、DB2 データベース・サーバーがインストールされているコンピューターのファイル・システム上にある場所と一致することを確認します。
- ルーチンが関数の場合、適用できる呼び出しタイプのすべてがルーチン・インプリメンテーションで正しくプログラムされていることを確認します。 FINAL CALL 節を使用してルーチンが定義されている場合には、これが特に重要です。
- ルーチンが期待どおりに動作しない場合、以下のようにします。
 - ルーチンが診断情報をグローバル・アクセス可能なディレクトリーにあるファイルに出力するように、ルーチンを変更します。診断情報を画面に出力することは、.NET CLR ルーチンからは行えません。DB2 データベース・マネージャーまたは DB2 データベースによって使用されるディレクトリー内のファイルに出力を送らないでください。
 - ルーチンのエントリー・ポイントを直接呼び出す単純な .NET アプリケーションを作成して、ルーチンをローカル側でデバッグします。Microsoft Visual Studio .NET でのデバッグ・フィーチャーの使用方法については、Microsoft Visual Studio .NET コンパイラーの資料を調べてください。

.NET CLR ルーチンの作成および呼び出しに関連する共通エラーについては詳しくは、以下を参照してください。

- 『.NET CLR ルーチンに関連したエラー』

.NET CLR ルーチンに関連したエラー

外部ルーチンのインプリメンテーションは基本的に共通ですが、CLR ルーチンに固有の DB2 エラーもいくつか発生します。このリファレンスでは、最もよく発生する .NET CLR 関連のエラーを SQLCODE ごと、または動作ごとに取り上げ、それぞれのデバッグのための提案を示します。ルーチンに関連した DB2 エラーは、以下のように分類できます。

ルーチン作成時のエラー

ルーチンを作成する CREATE ステートメントの実行時に発生するエラーです。

ルーチン実行時のエラー

ルーチンの呼び出し時または実行時に発生するエラーです。

DB2 のルーチンに関連したエラーが DB2 によっていつ検出されるかにかかわらず、エラー・メッセージのテキストには、エラーの原因と、その問題を解決するためにユーザーが行うべき処置が詳しく説明されています。ルーチン・エラーのシナリオに関するその他の情報については、db2diag 診断ログ・ファイルを参照してください。

CLR ルーチン作成時のエラー

SQLCODE -451、SQLSTATE 42815

このエラーは、LANGUAGE 節に CLR という値を指定した外部メソッド宣言を含んだ CREATE TYPE ステートメントを実行しようとしたときに発生します。現時点で、CLR アセンブリーを参照する構造化タイプの DB2 外部メソッドは作成できません。LANGUAGE 節を変更して、そのメソッドでサポートされている言語を指定し、その代替言語でメソッドをインプリメントしてください。

SQLCODE -449、SQLSTATE 42878

CLR ルーチンを作成する CREATE ステートメントの EXTERNAL NAME 節に、無効な形式のライブラリー指定または関数指定が含まれています。CLR 言語の場合、EXTERNAL 節の値は、':!<c>' という厳密な形式でなければなりません。それぞれの意味は、次のとおりです。

- <a> は CLR アセンブリー・ファイルです。このファイルの中に対象のクラスが存在しています。
- はクラスです。このクラスの中に呼び出し対象のメソッドが含まれています。
- <c> は呼び出し対象のメソッドです。

単一引用符、オブジェクト ID、分離文字の間に先行ブランクと末尾ブランクを入れてはなりません (例えば、' <a> ! ' は無効です)。ただし、プラットフォームによっては、パス名とファイル名にブランクを組み込むことは可能です。どんなファイル名についても、短形式の名前 (例: math.dll) と完全修飾パス名 (例: d:\udfs\math.dll) のどちらを指定してもかまいません。短形式のファイル名を使用するか、プラットフォームが UNIX であるか、ルーチンが LANGUAGE CLR ルーチンである場合は、対象のファイルが function ディレクトリーに存在する必要があります。プラットフォームが Windows であり、ルーチンが LANGUAGE CLR ルーチンでない場合は、対象のファイルがシステム PATH に存在する必要があります。ファイル名には、常にファイル拡張子 (例: .a (UNIX の場合)、.dll (Windows の場合)) を付けてください。

CLR ルーチン実行時のエラー

SQLCODE -20282、SQLSTATE 42724、理由コード 1

ルーチンを作成する CREATE ステートメントの EXTERNAL 節で指定した外部アセンブリーが見つかりません。

- EXTERNAL 節で正しいルーチン・アセンブリー名を指定したかどうか、そのアセンブリーが指定のロケーションに存在するかどうかを確認してください。EXTERNAL 節で対象のアセンブリーの完全修飾パス名を指定していない場合、DB2 は、そのパス名を、DB2 の function ディレクトリーを基準にしたアセンブリーの相対パス名と見なします。

SQLCODE -20282、SQLSTATE 42724、理由コード 2

ルーチンを作成する CREATE ステートメントの EXTERNAL 節で指定したロケーションでアセンブリーが見つかりましたが、そのアセンブリーの中に、EXTERNAL 節で指定したクラスに一致するクラスが存在しません。

- EXTERNAL 節で指定したアセンブリー名がルーチンの正しいアセンブリー名かどうか、そのアセンブリーが指定のロケーションに存在するかどうかを確認してください。
- EXTERNAL 節で指定したクラス名が正しいクラス名かどうか、そのクラスが指定のアセンブリーの中に存在するかどうかを確認してください。

SQLCODE -20282、SQLSTATE 42724、理由コード 3

ルーチンを作成する CREATE ステートメントの EXTERNAL 節で指定したロケーションでアセンブリーが見つかり、そのアセンブリーの中に該当するクラス定義が含まれていましたが、ルーチンのメソッド・シグニチャーが、そのルーチンの CREATE ステートメントで指定されたルーチン・シグニチャーと一致しません。

- EXTERNAL 節で指定したアセンブリー名がルーチンの正しいアセンブリー名かどうか、そのアセンブリーが指定のロケーションに存在するかどうかを確認してください。
- EXTERNAL 節で指定したクラス名が正しいクラス名かどうか、そのクラスが指定のアセンブリーの中に存在するかどうかを確認してください。
- パラメーター・スタイルのインプリメンテーションが、ルーチンを作成する CREATE ステートメントで指定したパラメーター・スタイルと一致するかどうかを確認してください。
- パラメーター・インプリメンテーションの順序が、ルーチンを作成する CREATE ステートメントで指定したパラメーター宣言の順序と一致するかどうか、そのパラメーター・スタイルのその他のパラメーター要件を満たしているかどうかを確認してください。
- SQL パラメーターのデータ・タイプが、CLR .NET でサポートされているデータ・タイプに正しくマップされているかどうかを確認してください。

SQLCODE -4301、SQLSTATE 58004、理由コード 5 または 6

.NET インタープリターを開始しようとしたとき、または .NET インタープリターと通信しようとしたときに、エラーが発生しました。DB2 が従属の .NET ライブラリーをロードできなかったか [理由コード 5]、.NET インタープリターの呼び出しが失敗しました [理由コード 6]。

- DB2 インスタンスが .NET のプロシージャーまたは関数を実行するための正しい構成になっているかどうかを確認してください (システム PATH に mscorere.dll が存在している必要があります)。db2c1r.dll が sqllib/bin ディレクトリに存在するかどうか、IBM.Data.DB2 がグローバル・アセンブリー・キャッシュにインストールされているかどうかを確認してください。そのいずれかが存在しない場合は、.NET Framework バージョン 1.1 以降がデータベース・サーバーにインストールされているかどうか、そのデータベース・サーバーが DB2 バージョン 8.2 以降のリリースを実行しているかどうかを確認してください。

SQLCODE -4302、SQLSTATE 38501

ルーチンの実行中か、実行準備中か、実行後に、処理できない例外が発生しました。これは、未処理になっていたルーチン・ロジックのプログラミング・エラーの結果か、内部処理エラーの結果であると考えられます。このタイプのエラーの場合、処理できない例外が発生した場所を示す .NET スタック・トレースバックが db2diag ログ・ファイルに書き込まれます。

また、ルーチンに指定された実行モードに対して許可されたアクションの有効範囲を超えるアクションをルーチンが試みた場合にも、このエラーが発生します。この場合、実行制御違反により例外が発生したことを具体的に示す項目が db2diag ログ・ファイルに書き込まれます。違反が発生した場所を示す例外スタック・トレースバックもこれに含まれます。

ルーチンのアセンブリーが破損していたり、最近変更されたりしたことはないか、判別してください。ルーチンの変更が妥当である場合、この問題は、ルーチンの EXECUTION CONTROL モードの設定が、変更後のロジックに適したものでなくなったために生じている可能性があります。アセンブリーで正しい変更が行われていることが確かな場合、ALTER PROCEDURE または ALTER FUNCTION ステートメントを使用してルーチンの実行モードを必要に応じて変更することができます。詳細については、以下のトピックを参照してください。

- 127 ページの『CLR ルーチンのセキュリティおよび実行モード』

.NET CLR ルーチンの例

.NET CLR ルーチンを開発する際には、例を参照して、CREATE ステートメントと .NET CLR ルーチン・コードをどのようにすればよいかという感覚をつかむと役に立ちます。以下のトピックには、.NET CLR プロシージャーと .NET CLR 関数 (スカラー関数と表関数の両方を含む) の例を記載しています。

.NET CLR プロシージャー

- Visual Basic .NET CLR プロシージャーの例
- C# .NET CLR プロシージャーの例

.NET CLR 関数

- Visual Basic .NET CLR 関数の例
- C# .NET CLR 関数の例

C# .NET CLR プロシージャの例

プロシージャ (ストアド・プロシージャともいう) の基礎と .NET 共通言語ランタイム・ルーチンの基本を理解できたら、アプリケーションで CLR プロシージャをさっそく活用できます。

このトピックでは、C# でインプリメントした CLR プロシージャの例をいくつか紹介します。それぞれの例は、サポートされているパラメーター・スタイル、パラメーター (dbinfo 構造を含む) の受け渡し、結果セットの戻し方などを示しています。C# の CLR UDF の例については、以下を参照してください。

- 178 ページの『C# .NET CLR 関数の例』

CLR プロシージャの例を使用した作業を開始する前に、概念について説明している以下のトピックを参照することもできます。

- 119 ページの『第 4 章 .NET 共通言語ランタイム (CLR) ルーチン』
- 130 ページの『DB2 コマンド・ウィンドウから .NET CLR ルーチンを作成する』
- 「*ADO.NET* および *OLE DB* アプリケーションの開発」内の『Common Language Runtime (CLR) .NET ルーチンの構築』

以下の例では、SAMPLE データベースに含まれる EMPLOYEE という名前の表を使用しています。

独自の C# CLR プロシージャを作成するときには、以下の例を参考にしてください。

- C# 外部コード・ファイル
- 例 1: C# のパラメーター・スタイル GENERAL のプロシージャ
- 例 2: C# のパラメーター・スタイル GENERAL WITH NULLS のプロシージャ
- 例 3: C# のパラメーター・スタイル SQL のプロシージャ
- 例 4: 結果セットを戻す C# のプロシージャ
- 例 5: dbinfo 構造にアクセスする C# のプロシージャ
- 例 6: PROGRAM TYPE MAIN スタイルの C# プロシージャ

C# 外部コード・ファイル

以下の例では、C# プロシージャのさまざまなインプリメンテーションを示しています。それぞれの例は、CREATE PROCEDURE ステートメントと、関連アセンブリーのビルド元プロシージャの外部 C# コード・インプリメンテーションという 2 つの部分から成っています。

以下の例のプロシージャ・インプリメンテーションに含まれる C# ソース・ファイルは、gwenProc.cs という名前であり、以下の形式になっています。

表 16. C# 外部コード・ファイルの形式

```
using System;
using System.IO;
using IBM.Data.DB2;

namespace bizLogic
{
    class empOps
    {
        ...
        // C# procedures
        ...
    }
}
```

ファイルの先頭には、このファイルに組み込むものを示します。ファイル内のプロシージャのいずれかに SQL が含まれる場合は、IBM.Data.DB2 を含める必要があります。このファイルには、ネーム・スペース宣言を組み込み、プロシージャを内容とするクラス empOps を組み込みます。ネーム・スペースの使用はオプションです。ネーム・スペースを使用する場合は、CREATE PROCEDURE ステートメントの EXTERNAL 節に指定するアセンブリー・パス名の中にネーム・スペースを入れなければなりません。

ファイルの名前、ネームスペース、特定のプロシージャ・インプリメンテーションを含むクラスの名前をメモしておくことは重要です。各プロシージャの CREATE PROCEDURE ステートメントの EXTERNAL 節でその情報を指定して、DB2 がアセンブリーと CLR プロシージャのクラスを見つけられるようにする必要があります。

例 1: C# のパラメーター・スタイル GENERAL のプロシージャ

この例では、以下について説明します。

- パラメーター・スタイル GENERAL のプロシージャの CREATE PROCEDURE ステートメント
- パラメーター・スタイル GENERAL のプロシージャの C# コード

このプロシージャは、従業員 ID と現在のボーナスの額を入力値として取り、そして、従業員の名前と給与を検索します。現在のボーナスの額がゼロの場合は、従業員の給与に基づいて新しいボーナスを計算し、従業員の氏名と一緒に戻します。従業員が見つからない場合は、空ストリングを戻します。

表 17. C# のパラメーター・スタイル GENERAL のプロシージャを作成するためのコード

```
CREATE PROCEDURE setEmpBonusGEN(IN empID CHAR(6), INOUT bonus Decimal(9,2),
                                OUT empName VARCHAR(60))
SPECIFIC SetEmpBonusGEN
LANGUAGE CLR
PARAMETER STYLE GENERAL
MODIFIES SQL DATA
EXECUTION CONTROL SAFE
FENCED
THREADSAFE
DYNAMIC RESULT SETS 0
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenProc.dll:bizLogic.empOps!SetEmpBonusGEN' ;
```

表 17. C# のパラメーター・スタイル *GENERAL* のプロシージャを作成するためのコード
(続き)

```
public static void SetEmpBonusGEN(    String empID,
                                     ref Decimal bonus,
                                     out String empName)
{
    // Declare local variables
    Decimal salary = 0;

    DB2Command myCommand = DB2Context.GetCommand();
    myCommand.CommandText =
        "SELECT FIRSTNME, MIDINIT, LASTNAME, SALARY "
        + "FROM EMPLOYEE "
        + "WHERE EMPNO = '" + empID + "'";

    DB2DataReader reader = myCommand.ExecuteReader();

    if (reader.Read()) // If employee record is found
    {
        // Get the employee's full name and salary
        empName = reader.GetString(0) + " " +
            reader.GetString(1) + ". " +
            reader.GetString(2);

        salary = reader.GetDecimal(3);

        if (bonus == 0)
        {
            if (salary > 75000)
            {
                bonus = salary * (Decimal)0.025;
            }
            else
            {
                bonus = salary * (Decimal)0.05;
            }
        }
    }
    else // Employee not found
    {
        empName = ""; // Set output parameter
    }

    reader.Close();
}
```

例 2: C# のパラメーター・スタイル *GENERAL WITH NULLS* のプロシージャ
この例では、以下について説明します。

- パラメーター・スタイル *GENERAL WITH NULLS* のプロシージャの **CREATE PROCEDURE** ステートメント
- パラメーター・スタイル *GENERAL WITH NULLS* のプロシージャの **C#** コード

このプロシージャは、従業員 ID と現在のボーナスの額を入力値として取ります。入力パラメーターが **NULL** 以外の場合は、従業員の名前と給与を検索します。現在のボーナスの額がゼロの場合は、給与に基づいて新しいボーナスを計算し、従業員の氏名と一緒に戻します。従業員データが見つからない場合は、**NULL** スtringと整数を戻します。

表 18. C# のパラメーター・スタイル *GENERAL WITH NULLS* のプロシージャーを作成するためのコード

```
CREATE PROCEDURE SetEmpbonusGENNULL(IN empID CHAR(6),
                                     INOUT bonus Decimal(9,2),
                                     OUT empName VARCHAR(60))

SPECIFIC SetEmpbonusGENNULL
LANGUAGE CLR
PARAMETER STYLE GENERAL WITH NULLS
DYNAMIC RESULT SETS 0
MODIFIES SQL DATA
EXECUTION CONTROL SAFE
FENCED
THREADSAFE
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenProc.dll:bizLogic.empOps!SetEmpBonusGENNULL'
;
```

表 18. C# のパラメーター・スタイル *GENERAL WITH NULLS* のプロシージャーを作成するためのコード (続き)

```

public static void SetEmpBonusGENNULL(    String empID,
                                          ref Decimal bonus,
                                          out String empName,
                                          Int16[] NullInds)
{
    Decimal salary = 0;
    if (NullInds[0] == -1) // Check if the input is null
    {
        NullInds[1] = -1;    // Return a NULL bonus value
        empName = "";      // Set output value
        NullInds[2] = -1;    // Return a NULL empName value
    }
    else
    {
        DB2Command myCommand = DB2Context.GetCommand();
        myCommand.CommandText =
            "SELECT FIRSTNAME, MIDINIT, LASTNAME, SALARY "
            + "FROM EMPLOYEE "
            + "WHERE EMPNO = '" + empID + "'";
        DB2DataReader reader = myCommand.ExecuteReader();

        if (reader.Read()) // If employee record is found
        {
            // Get the employee's full name and salary
            empName = reader.GetString(0) + " "
            +
                reader.GetString(1) + ". " +
                reader.GetString(2);
            salary = reader.GetDecimal(3);

            if (bonus == 0)
            {
                if (salary > 75000)
                {
                    bonus = salary * (Decimal)0.025;
                    NullInds[1] = 0; // Return a non-NULL value
                }
                else
                {
                    bonus = salary * (Decimal)0.05;
                    NullInds[1] = 0; // Return a non-NULL value
                }
            }
        }
        else // Employee not found
        {
            empName = "*sdq;"; // Set output parameter
            NullInds[2] = -1; // Return a NULL value
        }

        reader.Close();
    }
}

```

例 3: C# のパラメーター・スタイル *SQL* のプロシージャー

この例では、以下について説明します。

- パラメーター・スタイル *SQL* のプロシージャーの *CREATE PROCEDURE* ステートメント
- パラメーター・スタイル *SQL* のプロシージャーの C# コード

このプロシージャーは、従業員 ID と現在のボーナスの額を入力値として取ります。そして、従業員の名前と給与を検索します。現在のボーナスの額がゼロの場合は、給与に基づいて新しいボーナスを計算し、従業員の氏名と一緒に戻します。従業員が見つからない場合は、空ストリングを戻します。

表 19. パラメーターを使用してパラメーター・スタイル SQL で C# プロシージャーを作成するためのコード

```
CREATE PROCEDURE SetEmpbonusSQL(IN empID CHAR(6),
                                INOUT bonus Decimal(9,2),
                                OUT empName VARCHAR(60))
SPECIFIC SetEmpbonusSQL
LANGUAGE CLR
PARAMETER STYLE SQL
DYNAMIC RESULT SETS 0
MODIFIES SQL DATA
FENCED
THREADSAFE
EXECUTION CONTROL SAFE
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenProc.dll:bizLogic.empOps!SetEmpBonusSQL' ;
```

表 19. パラメーターを使用してパラメーター・スタイル SQL で C# プロシージャを作成するためのコード (続き)

```

public static void SetEmpBonusSQL(    String empID,
                                     ref Decimal bonus,
                                     out String empName,
                                     Int16 empIDNullInd,
                                     ref Int16 bonusNullInd,
                                     out Int16 empNameNullInd,
                                     ref string sqlStateate,
                                     string funcName,
                                     string specName,
                                     ref string sqlMessageText)
{
    // Declare local host variables
    Decimal salary eq; 0;

    if (empIDNullInd == -1) // Check if the input is null
    {
        bonusNullInd = -1; // Return a NULL bonus value
        empName = "";
        empNameNullInd = -1; // Return a NULL empName value
    }
    else
    {
        DB2Command myCommand = DB2Context.GetCommand();
        myCommand.CommandText =
            "SELECT FIRSTNME, MIDINIT, LASTNAME, SALARY
            "
            + "FROM EMPLOYEE "
            + "WHERE EMPNO = '" + empID + "'";

        DB2DataReader reader = myCommand.ExecuteReader();

        if (reader.Read()) // If employee record is found
        {
            // Get the employee's full name and salary
            empName = reader.GetString(0) + " "
            +
            reader.GetString(1) + ". " +
            reader.GetString(2);
            empNameNullInd = 0;
            salary = reader.GetDecimal(3);

            if (bonus == 0)
            {
                if (salary > 75000)
                {
                    bonus = salary * (Decimal)0.025;
                    bonusNullInd = 0; // Return a non-NULL value
                }
                else
                {
                    bonus = salary * (Decimal)0.05;
                    bonusNullInd = 0; // Return a non-NULL value
                }
            }
        }
        else // Employee not found
        {
            empName = ""; // Set output parameter
            empNameNullInd = -1; // Return a NULL value
        }

        reader.Close();
    }
}

```

例 4: 結果セットを戻す C# のパラメーター・スタイル GENERAL のプロシージャ
この例では、以下について説明します。

- 結果セットを戻す外部 C# プロシージャの CREATE PROCEDURE ステートメント
- 結果セットを戻すパラメーター・スタイル GENERAL のプロシージャの C# コード

このプロシージャは、パラメーターとして表の名前を受け入れます。そして、入力パラメーターによって指定されている表の行すべてを含む結果セットを戻します。この処理のために、プロシージャの戻り時に特定の照会結果セットの DB2DataReader をオープンしておきます。具体的には、reader.Close() が実行されなければ、結果セットが戻されるということです。

表 20. 結果セットを戻す C# プロシージャを作成するためのコード

```
CREATE PROCEDURE ReturnResultSet(IN tableName
                                VARCHAR(20))
SPECIFIC ReturnResultSet
DYNAMIC RESULT SETS 1
LANGUAGE CLR
PARAMETER STYLE GENERAL
FENCED
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenProc.dll:bizLogic.empOps!ReturnResultSet' ;
```

```
public static void ReturnResultSet(string tableName)
{
    DB2Command myCommand = DB2Context.GetCommand();

    // Set the SQL statement to be executed and execute it.
    myCommand.CommandText = "SELECT * FROM " + tableName;
    DB2DataReader reader = myCommand.ExecuteReader();

    // The DB2DataReader contains the result of the query.
    // This result set can be returned with the procedure,
    // by simply NOT closing the DB2DataReader.
    // Specifically, do NOT execute reader.Close();
}
```

例 5: dbinfo 構造にアクセスする C# のパラメーター・スタイル SQL のプロシージャ
この例では、以下について説明します。

- dbinfo 構造にアクセスするプロシージャの CREATE PROCEDURE ステートメント
- dbinfo 構造にアクセスするパラメーター・スタイル SQL のプロシージャの C# コード

dbinfo 構造にアクセスするには、CREATE PROCEDURE ステートメントに DBINFO 節を指定する必要があります。CREATE PROCEDURE ステートメントの dbinfo 構造にパラメーターは必要ありませんが、外部ルーチン・コードでそのためのパラメーターを作成する必要があります。このプロシージャは、dbinfo 構造の dbname フィールドからの現行データベース名の値だけを戻します。

表 21. dbinfo 構造にアクセスする C# プロシージャを作成するためのコード

```
CREATE PROCEDURE ReturnDbName(OUT dbName VARCHAR(20))
SPECIFIC ReturnDbName
DYNAMIC RESULT SETS 0
LANGUAGE CLR
PARAMETER STYLE SQL
FENCED
THREADSAFE
EXECUTION CONTROL SAFE
DBINFO
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenProc.dll:bizLogic.empOps!ReturnDbName'
;
```

```
public static void ReturnDbName(out string dbName,
                                out Int16 dbNameNullInd,
                                ref string sqlState,
                                string funcName,
                                string specName,
                                ref string sqlMessageText,
                                sqludf_dbinfo dbinfo)
{
    // Retrieve the current database name from the
    // dbinfo structure and return it.
    // ** Note! ** dbinfo field names are case sensitive
    dbName = dbinfo.dbname;
    dbNameNullInd = 0; // Return a non-null value;

    // If you want to return a user-defined error in
    // the SQLCA you can specify a 5 digit user-defined
    // sqlState and an error message string text.
    // For example:
    //
    //     sqlState = "ABCDE";
    //     sqlMessageText = "A user-defined error has occurred"
    //
    // DB2 returns the above values to the client in the
    // SQLCA structure. The values are used to generate a
    // standard DB2 sqlState error.
}
```

例 6: PROGRAM TYPE MAIN スタイルの C# プロシージャ

この例では、以下について説明します。

- メインプログラム・スタイルを使用したプロシージャの CREATE PROCEDURE ステートメント
- メインプログラム・スタイルを使用した C# のパラメーター・スタイル GENERAL WITH NULLS のコード

メインプログラム・スタイルでルーチンをインプリメントするには、CREATE PROCEDURE ステートメントの PROGRAM TYPE 節に MAIN という値を指定する必要があります。CREATE PROCEDURE ステートメントにもパラメーターを指定しますが、コードのインプリメンテーションでは、ルーチンの argc 整数パラメーターと argv パラメーター配列にパラメーターを渡します。

表 22. メインプログラム・スタイルで C# プロシージャを作成するためのコード

```
CREATE PROCEDURE MainStyle( IN empID CHAR(6),
                           INOUT bonus Decimal(9,2),
                           OUT empName VARCHAR(60))

SPECIFIC MainStyle
DYNAMIC RESULT SETS 0
LANGUAGE CLR
PARAMETER STYLE GENERAL WITH NULLS
MODIFIES SQL DATA
FENCED
THREADSAFE
EXECUTION CONTROL SAFE
PROGRAM TYPE MAIN
EXTERNAL NAME 'gwenProc.dll:bizLogic.empOps!main' ;
```

表 22. メインプログラム・スタイルで C# プロシーチャーを作成するためのコード (続き)

```

public static void main(Int32 argc, Object[]
argv)
{
    String empID = (String)argv[0]; // argv[0] has nullInd:argv[3]
    Decimal bonus = (Decimal)argv[1]; // argv[1] has nullInd:argv[4]
                                        // argv[2] has nullInd:argv[5]

    Decimal salary = 0;
    Int16[] NullInds = (Int16[])argv[3];

    if ((NullInds[0]) == (Int16)(-1)) // Check if empID is null
    {
        NullInds[1] = (Int16)(-1); // Return a NULL bonus value
        argv[1] = (String)""; // Set output parameter empName
        NullInds[2] = (Int16)(-1); // Return a NULL empName value
        Return;
    }
    else
    {
        DB2Command myCommand = DB2Context.GetCommand();
        myCommand.CommandText =
            "SELECT FIRSTNAME, MIDINIT, LASTNAME, salary "
            + "FROM EMPLOYEE "
            + "WHERE EMPNO = '" + empID + "'";

        DB2DataReader reader = myCommand.ExecuteReader();

        if (reader.Read()) // If employee record is found
        {
            // Get the employee's full name and salary
            argv[2] = (String) (reader.GetString(0) + " " +
                reader.GetString(1) + ".
                " +
                reader.GetString(2));
            NullInds[2] = (Int16)0;
            salary = reader.GetDecimal(3);

            if (bonus == 0)
            {
                if (salary > 75000)
                {
                    argv[1] = (Decimal)(salary * (Decimal)0.025);
                    NullInds[1] = (Int16)0; // Return a non-NULL value
                }
                else
                {
                    argv[1] = (Decimal)(salary * (Decimal)0.05);
                    NullInds[1] = (Int16)0; // Return a non-NULL value
                }
            }
        }
        else // Employee not found
        {
            argv[2] = (String)(""); // Set output parameter
            NullInds[2] = (Int16)(-1); // Return a NULL value
        }

        reader.Close();
    }
}

```

Visual Basic .NET CLR 関数の例

ユーザー定義関数 (UDF) の基礎と CLR ルーチンの基本を理解できたら、アプリケーションやデータベース環境の中で CLR UDF をさっそく活用できます。このトピックでは、手始めとして CLR UDF の例をいくつか紹介します。Visual Basic の CLR プロシージャの例については、以下を参照してください。

- 159 ページの『Visual Basic .NET CLR プロシージャの例』

CLR UDF の例を使用した作業を開始する前に、概念について説明している以下のトピックを参照することもできます。

- 119 ページの『第 4 章 .NET 共通言語ランタイム (CLR) ルーチン』
- 130 ページの『DB2 コマンド・ウィンドウから .NET CLR ルーチンを作成する』
- 68 ページの『外部スカラー関数』
- 「ADO.NET および OLE DB アプリケーションの開発」内の『Common Language Runtime (CLR) .NET ルーチンの構築』

以下の例では、SAMPLE データベースに含まれる EMPLOYEE という名前の表を使用しています。

独自の Visual Basic CLR UDF を作成するときには、以下の例を参考にしてください。

- Visual Basic 外部コード・ファイル
- 例 1: Visual Basic のパラメーター・スタイル SQL の表関数
- 例 2: Visual Basic のパラメーター・スタイル SQL のスカラー関数

Visual Basic 外部コード・ファイル

次の例では、Visual Basic UDF のさまざまなインプリメンテーションを示しています。各 UDF ごとに、関連アセンブリーのビルド元になる Visual Basic ソース・コードとともに、CREATE FUNCTION ステートメントを用意します。以下の例で使用している関数宣言に含まれる Visual Basic ソース・ファイルは、gwenVbUDF.cs という名前であり、以下の形式になっています。

表 23. Visual Basic 外部コード・ファイルの形式

```
using System;
using System.IO;
using IBM.Data.DB2;

Namespace bizLogic

    ...
    ' Class definitions that contain UDF declarations
    ' and any supporting class definitions
    ...

End Namespace
```

Visual Basic ファイル内のクラスに関数宣言を組み込む必要があります。ネーム・スペースの使用はオプションです。ネーム・スペースを使用する場合

は、CREATE PROCEDURE ステートメントの EXTERNAL 節に指定するアセンブリー・パス名の中にネーム・スペースを入れなければなりません。関数に SQL が含まれる場合は、IBM.Data.DB2. を含める必要があります。

例 1: Visual Basic のパラメーター・スタイル SQL の表関数

この例では、以下について説明します。

- パラメーター・スタイル SQL の表関数の CREATE FUNCTION ステートメント
- パラメーター・スタイル SQL の表関数の Visual Basic コード

この表関数は、データ配列から作成された従業員データの行を含んだ表を戻します。この例には、2 つの関連クラスがあります。1 つは従業員を表すクラス person であり、もう 1 つはクラス person を使用するルーチン表 UDF を含んだクラス empOps です。従業員の給与情報は、入力パラメーターの値に基づいて更新されます。この例のデータ配列は、表関数を最初に呼び出したときに表関数そのものの中に作成されます。そのような配列は、ファイル・システム上のテキスト・ファイルからデータを読み取ることによっても作成できます。表関数のその後の呼び出しで配列データにアクセスするために、データの値がスクラッチパッドに書き込まれます。

表関数を呼び出すたびに、1 つのレコードが配列から読み取られ、1 つの行が関数によって戻される表の中に生成されます。行を表の中に生成する処理は、表関数の出力パラメーターを対象の行値に設定するという形で実行されます。表関数の最終呼び出しが行われた後、生成された行の表が戻されず。

表 24. Visual Basic のパラメーター・スタイル SQL の表関数を作成するためのコード

```
CREATE FUNCTION TableUDF(double)
RETURNS TABLE (name varchar(20),
                job varchar(20),
                salary double)
EXTERNAL NAME 'gwenVbUDF.dll:bizLogic.empOps!TableUDF'
LANGUAGE CLR
PARAMETER STYLE SQL
NOT DETERMINISTIC
FENCED
SCRATCHPAD 10
FINAL CALL
DISALLOW PARALLEL
NO DBINFO
EXECUTION CONTROL SAFE
```

表 24. Visual Basic のパラメーター・スタイル SQL の表関数を作成するためのコード (続き)

```
Class Person
' The class Person is a supporting class for
' the table function UDF, tableUDF, below.

Private name As String
Private position As String
Private salary As Int32

Public Sub New(ByVal newName As String, _
              ByVal newPosition As String, _
              ByVal newSalary As Int32)

    name = newName
    position = newPosition
    salary = newSalary
End Sub

Public Property GetName() As String
    Get
        Return name
    End Get

    Set (ByVal value As String)
        name = value
    End Set
End Property

Public Property GetPosition() As String
    Get
        Return position
    End Get

    Set (ByVal value As String)
        position = value
    End Set
End Property

Public Property GetSalary() As Int32
    Get
        Return salary
    End Get

    Set (ByVal value As Int32)
        salary = value
    End Set
End Property

End Class
```

表 24. Visual Basic のパラメーター・スタイル SQL の表関数を作成するためのコード (続き)

```

Class empOps

    Public Shared Sub TableUDF(byVal factor As Double, _
                               byRef name As String, _
                               byRef position As String, _
                               byRef salary As Double, _
                               byVal factorNullInd As Int16, _
                               byRef nameNullInd As Int16, _
                               byRef positionNullInd As Int16, _
                               byRef salaryNullInd As Int16, _
                               byRef sqlState As String, _
                               byVal funcName As String, _
                               byVal specName As String, _
                               byRef sqlMessageText As String, _
                               byVal scratchPad As Byte(), _
                               byVal callType As Int32)

        Dim intRow As Int16

        intRow = 0

        ' Create an array of Person type information
        Dim staff(2) As Person
        staff(0) = New Person("Gwen", "Developer", 10000)
        staff(1) = New Person("Andrew", "Developer", 20000)
        staff(2) = New Person("Liu", "Team Leader", 30000)

        ' Initialize output parameter values and NULL indicators
        salary = 0
        name = position = ""
        nameNullInd = positionNullInd = salaryNullInd = -1

        Select callType
            Case -2 ' Case SQLUDF_TF_FIRST:
            Case -1 ' Case SQLUDF_TF_OPEN:
                intRow = 1
                scratchPad(0) = intRow ' Write to scratchpad
            Case 0 ' Case SQLUDF_TF_FETCH:
                intRow = scratchPad(0)
                If intRow > staff.Length
                    sqlState = "02000" ' Return an error SQLSTATE
                Else
                    ' Generate a row in the output table
                    ' based on the staff array data.
                    name = staff(intRow).GetName()
                    position = staff(intRow).GetPosition()
                    salary = (staff(intRow).GetSalary()) * factor
                    nameNullInd = 0
                    positionNullInd = 0
                    salaryNullInd = 0
                End If
                intRow = intRow + 1
                scratchPad(0) = intRow ' Write scratchpad

            Case 1 ' Case SQLUDF_TF_CLOSE:
            Case 2 ' Case SQLUDF_TF_FINAL:
        End Select

    End Sub

End Class

```

例 2: Visual Basic のパラメーター・スタイル SQL のスカラー関数

この例では、以下について説明します。

- パラメーター・スタイル SQL のスカラー関数の CREATE FUNCTION ステートメント
- パラメーター・スタイル SQL のスカラー関数の Visual Basic コード

このスカラー関数は、操作対象の入力値ごとに 1 つのカウント値を返します。入力値セットの n 番目の桁にある入力値に対する出力スカラー値は n になります。スカラー関数の各呼び出しでは、行または値の入力セット内のそれぞれの行または値に 1 つの呼び出しが関連付けられており、呼び出しのたびにカウントが 1 つずつ増え、カウントの現行値が戻されます。そのカウントはスクラッチパッドのメモリー・バッファー内に保管されるので、スカラー関数の呼び出しと呼び出しの間でカウントの値が保たれるようになっています。

例えば、表を次のように定義している場合は、このスカラー関数を簡単に呼び出すことができます。

```
CREATE TABLE T (i1 INTEGER);
INSERT INTO T VALUES 12, 45, 16, 99;
```

このスカラー関数の呼び出しには、以下のような簡単な照会を使用できます。

```
SELECT my_count(i1) as count, i1 FROM T;
```

この照会の出力は次のようになります。

COUNT	I1
1	12
2	45
3	16
4	99

このスカラー UDF は非常に簡単です。スカラー関数を使用するときには、行のカウントだけを返す代わりに、データの形式を既存の列に合わせることもできます。例えば、住所列の各値にストリングを付加することや、一連の入力ストリングから複雑なストリングを組み立てることや、中間結果の保管先のデータ・セットに対して複雑な数値評価を行うことなども可能です。

表 25. Visual Basic のパラメーター・スタイル SQL のスカラー関数を作成するためのコード

```
CREATE FUNCTION mycount(INTEGER)
RETURNS INTEGER
LANGUAGE CLR
PARAMETER STYLE SQL
NO SQL
SCRATCHPAD 10
FINAL CALL
FENCED
EXECUTION CONTROL SAFE
NOT DETERMINISTIC
EXTERNAL NAME 'gwenUDF.dll:bizLogic.empOps!CountUp';
```


表 25. Visual Basic のパラメーター・スタイル SQL のスカラー関数を作成するためのコード (続き)

```

Class empOps
  Public Shared Sub CountUp(byVal input As Int32, _
                           byRef outCounter As Int32, _
                           byVal nullIndInput As Int16, _
                           byRef nullIndOutCounter As Int16, _
                           byRef sqlState As String, _
                           byVal qualName As String, _
                           byVal specName As String, _
                           byRef sqlMessageText As String, _
                           byVal scratchPad As Byte(), _
                           byVal callType As Int32)

    Dim counter As Int32
    counter = 1

    Select callType
      case -1          ' case SQLUDF_TF_OPEN_CALL
        scratchPad(0) = counter
        outCounter = counter
        nullIndOutCounter = 0
      case 0          'case SQLUDF_TF_FETCH_CALL:
        counter = scratchPad(0)
        counter = counter + 1
        outCounter = counter
        nullIndOutCounter = 0
        scratchPad(0) = counter
      case 1          'case SQLUDF_CLOSE_CALL:
        counter = scratchPad(0)
        outCounter = counter
        nullIndOutCounter = 0
      case Else      ' Should never enter here
        ' These cases won't occur for the following reasons:
        ' Case -2 (SQLUDF_TF_FIRST)    ->No FINAL CALL in CREATE stmt
        ' Case 2 (SQLUDF_TF_FINAL)    ->No FINAL CALL in CREATE stmt
        ' Case 255 (SQLUDF_TF_FINAL_CRA) ->No SQL used in the function
        '
        ' * Note!*
        ' -----
        ' The Else is required so that at compile time
        ' out parameter outCounter is always set *
        outCounter = 0
        nullIndOutCounter = -1
    End Select
  End Sub

End Class

```

Visual Basic .NET CLR プロシージャの例

プロシージャ (ストアド・プロシージャともいう) の基礎と .NET 共通言語ランタイム・ルーチンの基本を理解できたら、アプリケーションで CLR プロシージャをさっそく活用できます。

このトピックでは、Visual Basic でインプリメントした CLR プロシージャの例をいくつか紹介します。それぞれの例は、サポートされているパラメーター・ス

タイトル、パラメーター (dbinfo 構造を含む) の受け渡し、結果セットの戻し方などを示しています。Visual Basic の CLR UDF の例については、以下を参照してください。

- 154 ページの『Visual Basic .NET CLR 関数の例』

CLR プロシージャの例を使用した作業を開始する前に、概念について説明している以下のトピックを参照することもできます。

- 119 ページの『第 4 章 .NET 共通言語ランタイム (CLR) ルーチン』
- 130 ページの『DB2 コマンド・ウィンドウから .NET CLR ルーチンを作成する』
- 3 ページの『ルーチン使用の利点』
- 「ADO.NET および OLE DB アプリケーションの開発」内の『Common Language Runtime (CLR) .NET ルーチンの構築』

以下の例では、SAMPLE データベースに含まれる EMPLOYEE という名前の表を使用しています。

独自の Visual Basic CLR プロシージャを作成するときには、以下の例を参考にしてください。

- Visual Basic 外部コード・ファイル
- 例 1: Visual Basic のパラメーター・スタイル GENERAL のプロシージャ
- 例 2: Visual Basic のパラメーター・スタイル GENERAL WITH NULLS のプロシージャ
- 例 3: Visual Basic のパラメーター・スタイル SQL のプロシージャ
- 例 4: 結果セットを戻す Visual Basic のプロシージャ
- 例 5: dbinfo 構造にアクセスする Visual Basic のプロシージャ
- 例 6: PROGRAM TYPE MAIN スタイルの Visual Basic プロシージャ

Visual Basic 外部コード・ファイル

以下の例では、Visual Basic プロシージャのさまざまなインプリメンテーションを示しています。それぞれの例は、CREATE PROCEDURE ステートメントと、関連アセンブリーのビルド元プロシージャの外部 Visual Basic コード・インプリメンテーションという 2 つの部分から成っています。

以下の例のプロシージャ・インプリメンテーションに含まれる Visual Basic ソース・ファイルは、gwenVbProc.vb という名前であり、以下の形式になっています。

表 26. Visual Basic 外部コード・ファイルの形式

```
using System;
using System.IO;
using IBM.Data.DB2;

Namespace bizLogic

    Class empOps
        ...
        ' Visual Basic procedures
        ...
    End Class
End Namespace
```

ファイルの先頭には、このファイルに組み込むものを示します。ファイル内のプロシージャのいずれかに SQL が含まれる場合は、IBM.Data.DB2 を含める必要があります。このファイルには、ネーム・スペース宣言を組み込み、プロシージャを内容とするクラス empOps を組み込みます。ネーム・スペースの使用はオプションです。ネーム・スペースを使用する場合は、CREATE PROCEDURE ステートメントの EXTERNAL 節に指定するアセンブリー・パス名の中にネーム・スペースを入れなければなりません。

ファイルの名前、ネームスペース、特定のプロシージャ・インプリメンテーションを含むクラスの名前をメモしておくことは重要です。各プロシージャの CREATE PROCEDURE ステートメントの EXTERNAL 節でその情報を指定して、DB2 がアセンブリーと CLR プロシージャのクラスを見つけれられるようにする必要があります。

例 1: Visual Basic のパラメーター・スタイル GENERAL のプロシージャ

この例では、以下について説明します。

- パラメーター・スタイル GENERAL のプロシージャの CREATE PROCEDURE ステートメント
- パラメーター・スタイル GENERAL のプロシージャの Visual Basic コード

このプロシージャは、従業員 ID と現在のボーナスの額を入力値として取ります。そして、従業員の名前と給与を検索します。現在のボーナスの額がゼロの場合は、従業員の給与に基づいて新しいボーナスを計算し、従業員の氏名と一緒に戻します。従業員が見つからない場合は、空ストリングを戻します。

表 27. Visual Basic のパラメーター・スタイル GENERAL のプロシージャを作成するためのコード

```
CREATE PROCEDURE SetEmpBonusGEN(IN empId CHAR(6),
                                INOUT bonus Decimal(9,2),
                                OUT empName VARCHAR(60))

SPECIFIC setEmpBonusGEN
LANGUAGE CLR
PARAMETER STYLE GENERAL
DYNAMIC RESULT SETS 0
FENCED
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenVbProc.dll:bizLogic.empOps!SetEmpBonusGEN'
```

表 27. Visual Basic のパラメーター・スタイル GENERAL のプロシージャを作成するためのコード (続き)

```
Public Shared Sub SetEmpBonusGEN(ByVal empId As String, _
                                ByRef bonus As Decimal, _
                                ByRef empName As String)

    Dim salary As Decimal
    Dim myCommand As DB2Command
    Dim myReader As DB2DataReader

    salary = 0

    myCommand = DB2Context.GetCommand()
    myCommand.CommandText = _
        "SELECT FIRSTNAME, MIDINIT, LASTNAME, SALARY " _
        + "FROM EMPLOYEE " _
        + "WHERE EMPNO = '" + empId + "'"
    myReader = myCommand.ExecuteReader()

    If myReader.Read() ' If employee record is found
        ' Get the employee's full name and salary
        empName = myReader.GetString(0) + " " _
            + myReader.GetString(1) + ". " _
            + myReader.GetString(2)

        salary = myReader.GetDecimal(3)

        If bonus = 0
            If salary > 75000
                bonus = salary * 0.025
            Else
                bonus = salary * 0.05
            End If
        End If
    Else ' Employee not found
        empName = "" ' Set output parameter
    End If

    myReader.Close()

End Sub
```

例 2: Visual Basic のパラメーター・スタイル GENERAL WITH NULLS のプロシージャ

この例では、以下について説明します。

- パラメーター・スタイル GENERAL WITH NULLS のプロシージャの CREATE PROCEDURE ステートメント
- パラメーター・スタイル GENERAL WITH NULLS のプロシージャの Visual Basic コード

このプロシージャは、従業員 ID と現在のボーナスの額を入力値として取ります。入力パラメーターが NULL 以外の場合は、従業員の名前と給与を検索します。現在のボーナスの額がゼロの場合は、給与に基づいて新しいボーナスを計算し、従業員の氏名と一緒に戻します。従業員データが見つからない場合は、NULL ストリングと整数を戻します。

表 28. Visual Basic のパラメーター・スタイル GENERAL WITH NULLS のプロシージャーを作成するためのコード

```
CREATE PROCEDURE SetEmpBonusGENNULL(IN empId CHAR(6),
                                     INOUT bonus Decimal(9,2),
                                     OUT empName VARCHAR(60))

SPECIFIC SetEmpBonusGENNULL
LANGUAGE CLR
PARAMETER STYLE GENERAL WITH NULLS
DYNAMIC RESULT SETS 0
FENCED
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenVbProc.dll:bizLogic.empOps!SetEmpBonusGENNULL'
```

```
Public Shared Sub SetEmpBonusGENNULL(ByVal empId As String, _
                                     ByRef bonus As Decimal, _
                                     ByRef empName As String, _
                                     byVal nullInds As Int16())

    Dim salary As Decimal
    Dim myCommand As DB2Command
    Dim myReader As DB2DataReader

    salary = 0

    If nullInds(0) = -1 ' Check if the input is null
        nullInds(1) = -1 ' Return a NULL bonus value
        empName = "" ' Set output parameter
        nullInds(2) = -1 ' Return a NULL empName value
        Return
    Else
        myCommand = DB2Context.GetCommand()
        myCommand.CommandText =
            "SELECT FIRSTNAME, MIDINIT, LASTNAME, SALARY " _
            + "FROM EMPLOYEE " _
            + "WHERE EMPNO = '" + empId + "'"

        myReader = myCommand.ExecuteReader()

        If myReader.Read() ' If employee record is found
            ' Get the employee's full name and salary
            empName = myReader.GetString(0) + " " _
                + myReader.GetString(1) + ". " _
                + myReader.GetString(2)

            salary = myReader.GetDecimal(3)

            If bonus = 0
                If salary > 75000
                    bonus = Salary * 0.025
                    nullInds(1) = 0 'Return a non-NULL value
                Else
                    bonus = salary * 0.05
                    nullInds(1) = 0 ' Return a non-NULL value
                End If
            Else 'Employee not found
                empName = "" ' Set output parameter
                nullInds(2) = -1 ' Return a NULL value
            End If
        End If

        myReader.Close()

    End If

End Sub
```

例 3: Visual Basic のパラメーター・スタイル SQL のプロシージャ

この例では、以下について説明します。

- パラメーター・スタイル SQL のプロシージャの CREATE PROCEDURE ステートメント
- パラメーター・スタイル SQL のプロシージャの Visual Basic コード

このプロシージャは、従業員 ID と現在のボーナスの額を入力値として取ります。そして、従業員の名前と給与を検索します。現在のボーナスの額がゼロの場合は、給与に基づいて新しいボーナスを計算し、従業員の氏名と一緒に戻します。従業員が見つからない場合は、空ストリングを戻します。

表 29. パラメーターを使用してパラメーター・スタイル SQL で Visual Basic プロシージャを作成するためのコード

```
CREATE PROCEDURE SetEmpBonusSQL(IN empId CHAR(6),
                                INOUT bonus Decimal(9,2),
                                OUT empName VARCHAR(60))

SPECIFIC SetEmpBonusSQL
LANGUAGE CLR
PARAMETER STYLE SQL
DYNAMIC RESULT SETS 0
FENCED
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenVbProc.dll:bizLogic.empOps!SetEmpBonusSQL'
```

表 29. パラメーターを使用してパラメーター・スタイル SQL で Visual Basic プロシージャを作成するためのコード (続き)

```

Public Shared Sub SetEmpBonusSQL(byVal empId As String, _
                                byRef bonus As Decimal, _
                                byRef empName As String, _
                                byVal empIdNullInd As Int16, _
                                byRef bonusNullInd As Int16, _
                                byRef empNameNullInd As Int16, _
                                byRef sqlState As String, _
                                byVal funcName As String, _
                                byVal specName As String, _
                                byRef sqlMessageText As String)

    ' Declare local host variables
    Dim salary As Decimal
    Dim myCommand As DB2Command
    Dim myReader As DB2DataReader

    salary = 0

    If empIdNullInd = -1 ' Check if the input is null
        bonusNullInd = -1 ' Return a NULL Bonus value
        empName = ""
        empNameNullInd = -1 ' Return a NULL empName value
    Else
        myCommand = DB2Context.GetCommand()
        myCommand.CommandText = _
            "SELECT FIRSTNME, MIDINIT, LASTNAME, SALARY " _
            + "FROM EMPLOYEE "
            + "WHERE EMPNO = '" + empId + "'"

        myReader = myCommand.ExecuteReader()

        If myReader.Read() ' If employee record is found
            ' Get the employee's full name and salary
            empName = myReader.GetString(0) + " "
                + myReader.GetString(1)
                + ". " + myReader.GetString(2)
            empNameNullInd = 0
            salary = myReader.GetDecimal(3)

            If bonus = 0
                If salary > 75000
                    bonus = salary * 0.025
                    bonusNullInd = 0 ' Return a non-NULL value
                Else
                    bonus = salary * 0.05
                    bonusNullInd = 0 ' Return a non-NULL value
                End If
            End If
        Else ' Employee not found
            empName = "" ' Set output parameter
            empNameNullInd = -1 ' Return a NULL value
        End If

        myReader.Close()
    End If

End Sub

```

例 4: 結果セットを戻す Visual Basic のパラメーター・スタイル GENERAL のプロシージャ

この例では、以下について説明します。

- 結果セットを戻す外部 Visual Basic プロシージャの CREATE PROCEDURE ステートメント
- 結果セットを戻すパラメーター・スタイル GENERAL のプロシージャの Visual Basic コード

このプロシージャは、パラメーターとして表の名前を受け入れます。そして、入力パラメーターによって指定されている表の行すべてを含む結果セットを戻します。この処理のために、プロシージャの戻り時に特定の照会結果セットの DB2DataReader をオープンしておきます。具体的には、reader.Close() が実行されなければ、結果セットが戻されるということです。

表 30. 結果セットを戻す Visual Basic プロシージャを作成するためのコード

```
CREATE PROCEDURE ReturnResultSet(IN tableName VARCHAR(20))
SPECIFIC ReturnResultSet
DYNAMIC RESULT SETS 1
LANGUAGE CLR
PARAMETER STYLE GENERAL
FENCED
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenVbProc.dll:bizLogic.empOps!ReturnResultSet'
```

```
Public Shared Sub ReturnResultSet(byVal tableName As String)

    Dim myCommand As DB2Command
    Dim myReader As DB2DataReader

    myCommand = DB2Context.GetCommand()

    ' Set the SQL statement to be executed and execute it.
    myCommand.CommandText = "SELECT * FROM " + tableName
    myReader = myCommand.ExecuteReader()

    ' The DB2DataReader contains the result of the query.
    ' This result set can be returned with the procedure,
    ' by simply NOT closing the DB2DataReader.
    ' Specifically, do NOT execute reader.Close()

End Sub
```

例 5: dbinfo 構造にアクセスする Visual Basic のパラメーター・スタイル SQL のプロシージャ

この例では、以下について説明します。

- dbinfo 構造にアクセスするプロシージャの CREATE PROCEDURE ステートメント
- dbinfo 構造にアクセスするパラメーター・スタイル SQL のプロシージャの Visual Basic コード

dbinfo 構造にアクセスするには、CREATE PROCEDURE ステートメントに DBINFO 節を指定する必要があります。CREATE PROCEDURE ステートメントの dbinfo 構造にパラメーターは必要ありませんが、外部ルーチン・コードでそのためのパラメーターを作成する必要があります。このプロシージャは、dbinfo 構造の dbname フィールドからの現行データベース名の値だけを戻します。

表 31. dbinfo 構造にアクセスする Visual Basic プロシージャを作成するためのコード

```

CREATE PROCEDURE ReturnDbName(OUT dbName VARCHAR(20))
SPECIFIC ReturnDbName
LANGUAGE CLR
PARAMETER STYLE SQL
DBINFO
FENCED
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenVbProc.dll:bizLogic.empOps!ReturnDbName'

Public Shared Sub ReturnDbName(byRef dbName As String, _
                               byRef dbNameNullInd As Int16, _
                               byRef sqlState As String, _
                               byVal funcName As String, _
                               byVal specName As String, _
                               byRef sqlMessageText As String, _
                               byVal dbinfo As sqludf_dbinfo)

    ' Retrieve the current database name from the
    ' dbinfo structure and return it.
    dbName = dbinfo.dbname
    dbNameNullInd = 0 ' Return a non-null value

    ' If you want to return a user-defined error in
    ' the SQLCA you can specify a 5 digit user-defined
    ' SQLSTATE and an error message string text.
    ' For example:
    '
    ' sqlState = "ABCDE"
    ' msg_token = "A user-defined error has occurred"
    '
    ' These will be returned by DB2 in the SQLCA. It
    ' will appear in the format of a regular DB2 sqlState
    ' error.
End Sub

```

例 6: PROGRAM TYPE MAIN スタイルの Visual Basic プロシージャ

この例では、以下について説明します。

- メインプログラム・スタイルを使用したプロシージャの CREATE PROCEDURE ステートメント
- メインプログラム・スタイルを使用した Visual Basic のパラメーター・スタイル GENERAL WITH NULLS のコード

メインプログラム・スタイルでルーチンをインプリメントするには、CREATE PROCEDURE ステートメントの PROGRAM TYPE 節に MAIN という値を指定する必要があります。CREATE PROCEDURE ステートメントにもパラメーターを指定しますが、コードのインプリメンテーションでは、ルーチンの argc 整数パラメーターと argv パラメーター配列にパラメーターを渡します。

表 32. メインプログラム・スタイルで *Visual Basic* プロシージャを作成するためのコード

```
CREATE PROCEDURE MainStyle(IN empId CHAR(6),
                           INOUT bonus Decimal(9,2),
                           OUT empName VARCHAR(60))
SPECIFIC mainStyle
DYNAMIC RESULT SETS 0
LANGUAGE CLR
PARAMETER STYLE GENERAL WITH NULLS
FENCED
PROGRAM TYPE MAIN
EXTERNAL NAME 'gwenVbProc.dll:bizLogic.empOps!Main'
```

表 32. メインプログラム・スタイルで Visual Basic プロシージャを作成するためのコード
(続き)

```

Public Shared Sub Main( ByVal argc As Int32,
                        ByVal argv As Object())

    Dim myCommand As DB2Command
    Dim myReader As DB2DataReader
    Dim empId As String
    Dim bonus As Decimal
    Dim salary As Decimal
    Dim nullInds As Int16()

    empId = argv(0) ' argv[0] (IN)    nullInd = argv[3]
    bonus = argv(1) ' argv[1] (INOUT) nullInd = argv[4]
                        ' argv[2] (OUT)  nullInd = argv[5]

    salary = 0
    nullInds = argv(3)

    If nullInds(0) = -1 ' Check if the empId input is null
        nullInds(1) = -1 ' Return a NULL Bonus value
        argv(1) = "" ' Set output parameter empName
        nullInds(2) = -1 ' Return a NULL empName value
        Return
    Else
        ' If the employee exists and the current bonus is 0,
        ' calculate a new employee bonus based on the employee's
        ' salary. Return the employee name and the new bonus
        myCommand = DB2Context.GetCommand()
        myCommand.CommandText =
            "SELECT FIRSTNME, MIDINIT, LASTNAME, SALARY " _
            + " FROM EMPLOYEE " _
            + " WHERE EMPNO = '" + empId + "'"

        myReader = myCommand.ExecuteReader()

        If myReader.Read() ' If employee record is found
            ' Get the employee's full name and salary
            argv(2) = myReader.GetString(0) + " " _
                + myReader.GetString(1) + ". " _
                + myReader.GetString(2)
            nullInds(2) = 0
            salary = myReader.GetDecimal(3)

            If bonus = 0
                If salary > 75000
                    argv(1) = salary * 0.025
                    nullInds(1) = 0 ' Return a non-NULL value
                Else
                    argv(1) = Salary * 0.05
                    nullInds(1) = 0 ' Return a non-NULL value
                End If
            End If
        Else ' Employee not found
            argv(2) = "" ' Set output parameter
            nullInds(2) = -1 ' Return a NULL value
        End If

        myReader.Close()
    End If

End Sub

```

例: C# .NET CLR プロシージャでの XML および XQuery サポート

プロシージャの基本、.NET 共通言語ランタイム・ルーチンの本質部分、XQuery および XML を理解したなら、XML フィーチャーを持つ CLR プロシージャの作成および使用を始めることができます。

以下の例は、XML データの更新および照会方法に加えて、タイプ XML のパラメーターを使用する C# .NET CLR プロシージャを示します。

前提条件

CLR プロシージャの例を使用した作業を開始する前に、概念について説明している以下のトピックを参照することもできます。

- .NET 共通言語ランタイム (CLR) ルーチン
- DB2 コマンド・ウィンドウから .NET CLR ルーチンを作成する
- ルーチン使用の利点
- 「*ADO.NET* および *OLE DB* アプリケーションの開発」内の『Common Language Runtime (CLR) .NET ルーチンの構築』

この下の例では、以下のように定義された `xmlDataTable` という名前の表を使用します。

```
CREATE TABLE xmlDataTable
(
    num INTEGER,
    xdata XML
)

INSERT INTO xmlDataTable VALUES
(1, XMLPARSE(DOCUMENT '<doc>
                    <type>car</type>
                    <make>Pontiac</make>
                    <model>Sunfire</model>
                    </doc>' PRESERVE WHITESPACE)),
(2, XMLPARSE(DOCUMENT '<doc>
                    <type>car</type>
                    <make>Mazda</make>
                    <model>Miata</model>
                    </doc>' PRESERVE WHITESPACE)),
(3, XMLPARSE(DOCUMENT '<doc>
                    <type>person</type>
                    <name>Mary</name>
                    <town>Vancouver</town>
                    <street>Waterside</street>
                    </doc>' PRESERVE WHITESPACE)),
(4, XMLPARSE(DOCUMENT '<doc>
                    <type>person</type>
                    <name>Mark</name>
                    <town>Edmonton</town>
                    <street>Oak</street>
                    </doc>' PRESERVE WHITESPACE)),
(5, XMLPARSE(DOCUMENT '<doc>
                    <type>animal</type>
                    <name>dog</name>
                    </doc>' PRESERVE WHITESPACE)),
(6, NULL),
(7, XMLPARSE(DOCUMENT '<doc>
                    <type>car</type>
                    <make>Ford</make>
                    <model>Taurus</model>
                    </doc>' PRESERVE WHITESPACE)),
```

```

(8, XMLPARSE(DOCUMENT '<doc>
    <type>person</type>
    <name>Kim</name>
    <town>Toronto</town>
    <street>Elm</street>
</doc>' PRESERVE WHITESPACE)),
(9, XMLPARSE(DOCUMENT '<doc>
    <type>person</type>
    <name>Bob</name>
    <town>Toronto</town>
    <street>Oak</street>
</doc>' PRESERVE WHITESPACE)),
(10, XMLPARSE(DOCUMENT '<doc>
    <type>animal</type>
    <name>bird</name>
</doc>' PRESERVE WHITESPACE))@

```

手順 独自の C# CLR プロシージャを作成するときには、以下の例を参考にしてください。

- C# 外部コード・ファイル
- 例 1: XML フィーチャーを持つ C# パラメーター・スタイル GENERAL プロシージャ

C# 外部コード・ファイル

例は、CREATE PROCEDURE ステートメントと、関連アセンブリーのビルド元プロシージャの外部 C# コード・インプリメンテーションという 2 つの部分から成っています。

以下の例のプロシージャ・インプリメンテーションに含まれる C# ソース・ファイルは、gwenProc.cs という名前であり、以下の形式になっています。

表 33. C# 外部コード・ファイルの形式

```

using System;
using System.IO;
using System.Data;
using IBM.Data.DB2;
using IBM.Data.DB2Types;

namespace bizLogic
{
    class empOps
    {
        ...
        // C# procedures
        ...
    }
}

```

ファイルの先頭には、このファイルに組み込むものを示します。ファイル内のプロシージャのいずれかに SQL が含まれる場合は、IBM.Data.DB2 を含める必要があります。ファイル内のプロシージャのいずれかにタイプ XML のパラメーターまたは変数が含まれる場合は、IBM.Data.DB2Types を含める必要があります。このファイルには、ネーム・スペース宣言を組み込み、プロシージャを内容とするクラス empOps を組み込みます。ネーム・スペースの使用はオプションです。ネーム・スペースを使用する場合は、CREATE PROCEDURE ステートメントの EXTERNAL 節に指定するアセンブリー・パス名の中にネーム・スペースを入れなければなりません。

ファイルの名前、ネームスペース、特定のプロシージャ・インプリメンテーションを含むクラスの名前をメモしておくことは重要です。各プロシージャの CREATE PROCEDURE ステートメントの EXTERNAL 節でその情報を指定して、DB2 がアセンブリーと CLR プロシージャのクラスを見つけられるようにする必要があります。

例 1: XML フィーチャーを持つ C# パラメーター・スタイル GENERAL プロシージャ

この例では、以下について説明します。

- パラメーター・スタイル GENERAL のプロシージャの CREATE PROCEDURE ステートメント
- XML パラメーターを使用するパラメーター・スタイル GENERAL プロシージャの C# コード

このプロシージャは、整数 inNum と inXML という 2 つのパラメーターを取ります。これらの値は表 xmlDataTable に挿入されます。次に、XML 値が XQuery を使用して検索されます。もう 1 つの XML 値が SQL を使用して検索されます。検索された XML 値は 2 つの出力パラメーター、outXML1 と outXML2 に割り当てられます。結果セットは戻されません。

表 34. C# のパラメーター・スタイル GENERAL のプロシージャを作成するためのコード

```
CREATE PROCEDURE xmlProc1 ( IN inNUM INTEGER,
                           IN inXML XML as CLOB (1K),
                           OUT inXML XML as CLOB (1K),
                           OUT inXML XML as CLOB (1K)
                           )

LANGUAGE CLR
PARAMETER STYLE GENERAL
DYNAMIC RESULT SETS 0
FENCED
THREADSAFE
DETERMINISTIC
NO DBINFO
MODIFIES SQL DATA
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenProc.dll:bizLogic.empOps!xmlProc1' ;

//*****
// Stored Procedure: xmlProc1
//
// Purpose:  insert XML data into XML column
//
// Parameters:
//
// IN:      inNum -- the sequence of XML data to be insert in xmldata table
//          inXML -- XML data to be inserted
// OUT:     outXML1 -- XML data returned - value retrieved using XQuery
//          outXML2 -- XML data returned - value retrieved using SQL
//*****
```

表 34. C# のパラメーター・スタイル *GENERAL* のプロシージャーを作成するためのコード
(続き)

```

public static void xmlProc1 (    int    inNum, DB2Xml  inXML,
                               out DB2Xml  outXML1, out DB2Xml  outXML2 )
{
    // Create new command object from connection context
    DB2Parameter parm;
    DB2Command cmd;
    DB2DataReader reader = null;
    outXML1 = DB2Xml.Null;
    outXML2 = DB2Xml.Null;

    // Insert input XML parameter value into a table
    cmd = DB2Context.GetCommand();
    cmd.CommandText = "INSERT INTO "
        + "xmlDataTable( num , xdata ) "
        + "VALUES( ?, ?)";

    parm = cmd.Parameters.Add("@num", DB2Type.Integer );
    parm.Direction = ParameterDirection.Input;
    cmd.Parameters["@num"].Value = inNum;
    parm = cmd.Parameters.Add("@data", DB2Type.Xml);
    parm.Direction = ParameterDirection.Input;
    cmd.Parameters["@data"].Value = inXML ;
    cmd.ExecuteNonQuery();
    cmd.Close();

    // Retrieve XML value using XQuery
    // and assign value to an XML output parameter
    cmd = DB2Context.GetCommand();
    cmd.CommandText = "XQUERY for $x " +
        "in db2-fn:xmlcolumn(¥'xmlDataTable.xdata¥')/doc "+
        "where $x/make = ¥'Mazda¥' " +
        "return <carInfo>{$x/make}{¥$x/model¥}</carInfo>";
    reader = cmd.ExecuteReader();
    reader.CacheData= true;

    if (reader.Read())
    { outXML1 = reader.GetDB2Xml(0); }
    else
    { outXML1 = DB2Xml.Null; }

    reader.Close();
    cmd.Close();

    // Retrieve XML value using SQL
    // and assign value to an XML output parameter value
    cmd = DB2Context.GetCommand();
    cmd.CommandText = "SELECT xdata "
        + "FROM xmlDataTable "
        + "WHERE num = ?";

    parm = cmd.Parameters.Add("@num", DB2Type.Integer );
    parm.Direction = ParameterDirection.Input;
    cmd.Parameters["@num"].Value = inNum;
    reader = cmd.ExecuteReader();
    reader.CacheData= true;

    if (reader.Read())
    { outXML2 = reader.GetDB2Xml(0); }
    else
    { outXML2 = DB2Xml.Null; }

    reader.Close();
    cmd.Close();

    return;
}

```

例: C プロシージャでの XML および XQuery サポート

プロシージャの基本、C ルーチンの本質部分、XQuery および XML を理解したなら、XML 機能を持つ C プロシージャの作成および使用を始めることができます。

以下の例は、XML データの更新および照会方法に加えて、タイプ XML のパラメーターを使用する C プロシージャを示します。

前提条件

C プロシージャの例を使用した作業を開始する前に、概念について説明している以下のトピックを参照することもできます。

- ルーチン使用の利点

この下の例では、以下のように定義された `xmlDataTable` という名前の表を使用します。

```
CREATE TABLE xmlDataTable
(
    num INTEGER,
    xdata XML
)

INSERT INTO xmlDataTable VALUES
(1, XMLPARSE(DOCUMENT '<doc>
                    <type>car</type>
                    <make>Pontiac</make>
                    <model>Sunfire</model>
                    </doc>' PRESERVE WHITESPACE)),
(2, XMLPARSE(DOCUMENT '<doc>
                    <type>car</type>
                    <make>Mazda</make>
                    <model>Miata</model>
                    </doc>' PRESERVE WHITESPACE)),
(3, XMLPARSE(DOCUMENT '<doc>
                    <type>person</type>
                    <name>Mary</name>
                    <town>Vancouver</town>
                    <street>Waterside</street>
                    </doc>' PRESERVE WHITESPACE)),
(4, XMLPARSE(DOCUMENT '<doc>
                    <type>person</type>
                    <name>Mark</name>
                    <town>Edmonton</town>
                    <street>Oak</street>
                    </doc>' PRESERVE WHITESPACE)),
(5, XMLPARSE(DOCUMENT '<doc>
                    <type>animal</type>
                    <name>dog</name>
                    </doc>' PRESERVE WHITESPACE)),
(6, NULL),
(7, XMLPARSE(DOCUMENT '<doc>
                    <type>car</type>
                    <make>Ford</make>
                    <model>Taurus</model>
                    </doc>' PRESERVE WHITESPACE)),
(8, XMLPARSE(DOCUMENT '<doc>
                    <type>person</type>
                    <name>Kim</name>
                    <town>Toronto</town>
                    <street>Elm</street>
                    </doc>' PRESERVE WHITESPACE)),
```



```
(9, XMLPARSE(DOCUMENT '<doc>
                                <type>person</type>
                                <name>Bob</name>
                                <town>Toronto</town>
                                <street>Oak</street>
                                </doc>' PRESERVE WHITESPACE)),
(10, XMLPARSE(DOCUMENT '<doc>
                                <type>animal</type>
                                <name>bird</name>
                                </doc>' PRESERVE WHITESPACE))
```

手順 独自の C プロシージャを作成するときには、以下の例を参考にしてください。

- C 外部コード・ファイル
- 例 1: XML フィーチャーを持つ C パラメーター・スタイル SQL プロシージャ

C 外部コード・ファイル

例は、CREATE PROCEDURE ステートメントと、関連アセンブリーのビルド元プロシージャの外部 C コード・インプリメンテーションという 2 つの部分から成っています。

以下の例のプロシージャ・インプリメンテーションに含まれる C ソース・ファイルは、gwenProc.SQC という名前であり、以下の形式になっています。

表 35. C 外部コード・ファイルの形式

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlda.h>
#include <sqlca.h>
#include <sqludf.h>
#include <sql.h>
#include <memory.h>

// C procedures
...
```

ファイルの先頭には、このファイルに組み込むものを示します。組み込み SQL ルーチンには、XML サポートに必要な余分の組み込みファイルはありません。

ファイルの名前、およびプロシージャ・インプリメンテーションに対応する関数の名前をメモしておくことは重要です。各プロシージャの CREATE PROCEDURE ステートメントの EXTERNAL 節でその情報を指定して、DB2 データベース・マネージャーがその C プロシージャに該当するライブラリーとエンタリー・ポイントを見つけられるようにする必要があります。

例 1: XML フィーチャーを持つ C パラメーター・スタイル SQL プロシージャ

この例では、以下について説明します。

- パラメーター・スタイル SQL のプロシージャの CREATE PROCEDURE ステートメント

- XML パラメーターを使用するパラメーター・スタイル SQL プロシージャの C コード

このプロシージャは 2 つの入力パラメーターを取ります。最初の入力パラメーターの名前は inNum で、タイプは INTEGER です。2 番目の入力パラメーターの名前は inXML で、タイプは XML です。入力パラメーターの値を使用して、行を表 xmlDataTable に挿入します。次に、XML 値が SQL ステートメントを使用して検索されます。もう 1 つの XML 値が XQuery 式を使用して検索されます。検索された XML 値はそれぞれ 2 つの出力パラメーター、out1XML と out2XML に割り当てられます。結果セットは戻されません。

表 36. C のパラメーター・スタイル SQL のプロシージャを作成するためのコード

```
CREATE PROCEDURE xmlProc1 ( IN inNUM INTEGER,
                           IN inXML XML as CLOB (1K),
                           OUT inXML XML as CLOB (1K),
                           OUT inXML XML as CLOB (1K)
                           )

LANGUAGE C
PARAMETER STYLE SQL
DYNAMIC RESULT SETS 0
FENCED
THREADSAFE
DETERMINISTIC
NO DBINFO
MODIFIES SQL DATA
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenProc!xmlProc1' ;

//*****
// Stored Procedure: xmlProc1
//
// Purpose:  insert XML data into XML column
//
// Parameters:
//
// IN:    inNum -- the sequence of XML data to be insert in xmldata table
//        inXML -- XML data to be inserted
// OUT:   out1XML -- XML data returned - value retrieved using XQuery
//        out2XML -- XML data returned - value retrieved using SQL
//*****
```

表 36. C のパラメーター・スタイル SQL のプロシーチャーを作成するためのコード (続き)

```

#ifdef __cplusplus
extern "C"
#endif
SQL_API_RC SQL_API_FN testSecA1(sqlint32* inNum,
                                SQLUDF_CLOB* inXML,
                                SQLUDF_CLOB* out1XML,
                                SQLUDF_CLOB* out2XML,
                                SQLUDF_NULLIND *inNum_ind,
                                SQLUDF_NULLIND *inXML_ind,
                                SQLUDF_NULLIND *out1XML_ind,
                                SQLUDF_NULLIND *out2XML_ind,
                                SQLUDF_TRAIL_ARGS)
{
    char *str;
    FILE *file;

    EXEC SQL INCLUDE SQLCA;

    EXEC SQL BEGIN DECLARE SECTION;
        sqlint32 hvNum1;
        SQL TYPE IS XML AS CLOB(200) hvXML1;
        SQL TYPE IS XML AS CLOB(200) hvXML2;
        SQL TYPE IS XML AS CLOB(200) hvXML3;
    EXEC SQL END DECLARE SECTION;

    /* Check null indicators for input parameters */
    if ((*inNum_ind < 0) || (*inXML_ind < 0)) {
        strcpy(sqludf_sqlstate, "38100");
        strcpy(sqludf_msgtext, "Received null input");
        return 0;
    }

    /* Copy input parameters to host variables */
    hvNum1 = *inNum;
    hvXML1.length = inXML->length;
    strncpy(hvXML1.data, inXML->data, inXML->length);

    /* Execute SQL statement */
    EXEC SQL
        INSERT INTO xmlDataTable (num, xdata) VALUES (:hvNum1, :hvXML1);

    /* Execute SQL statement */
    EXEC SQL
        SELECT xdata INTO :hvXML2
        FROM xmlDataTable
        WHERE num = :hvNum1;

    sprintf(stmt5, "SELECT XMLQUERY('for $x in $xmldata/doc
                                return <carInfo>{$x/model}</carInfo>'
                                passing by ref xmlDataTable.xdata
                                as ¥"xmldata¥" returning sequence)
        FROM xmlDataTable WHERE num = ?");

    EXEC SQL PREPARE selstmt5 FROM :stmt5 ;
    EXEC SQL DECLARE c5 CURSOR FOR selstmt5;
    EXEC SQL OPEN c5 using :hvNum1;
    EXEC SQL FETCH c5 INTO :hvXML3;

    exit:

    /* Set output return code */
    *outReturnCode = sqlca.sqlcode;
    *outReturnCode_ind = 0;

    return 0;
}

```

C# .NET CLR 関数の例

ユーザー定義関数 (UDF) の基礎と CLR ルーチンの基本を理解できたら、アプリケーションやデータベース環境の中で CLR UDF をさっそく活用できます。このトピックでは、手始めとして CLR UDF の例をいくつか紹介します。C# の CLR プロシージャの例については、以下を参照してください。

- 143 ページの『C# .NET CLR プロシージャの例』

CLR UDF の例を使用した作業を開始する前に、概念について説明している以下のトピックを参照することもできます。

- 119 ページの『第 4 章 .NET 共通言語ランタイム (CLR) ルーチン』
- 130 ページの『DB2 コマンド・ウィンドウから .NET CLR ルーチンを作成する』
- 68 ページの『外部スカラー関数』
- 「ADO.NET および OLE DB アプリケーションの開発」内の『Common Language Runtime (CLR) .NET ルーチンの構築』

以下の例では、SAMPLE データベースに含まれる EMPLOYEE という名前の表を使用しています。

独自の C# CLR UDF を作成するときには、以下の例を参考にしてください。

- C# 外部コード・ファイル
- 例 1: C# のパラメーター・スタイル SQL の表関数
- 例 2: C# のパラメーター・スタイル SQL のスカラー関数

C# 外部コード・ファイル

以下の例では、C# UDF のさまざまなインプリメンテーションを示しています。各 UDF ごとに、関連アセンブリーのビルド元になる C# ソース・コードとともに、CREATE FUNCTION ステートメントを用意します。以下の例で使用している関数宣言に含まれる C# ソース・ファイルは、gwenUDF.cs という名前であり、以下の形式になっています。

表 37. C# 外部コード・ファイルの形式

```
using System;
using System.IO;
using IBM.Data.DB2;

namespace bizLogic
{
    ...
    // Class definitions that contain UDF declarations
    // and any supporting class definitions
    ...
}
```

C# ファイル内のクラスに関数宣言を組み込む必要があります。ネーム・スペースの使用はオプションです。ネーム・スペースを使用する場合は、CREATE PROCEDURE ステートメントの EXTERNAL 節に指定するアセン

ブリー・パス名の中にネーム・スペースを入れなければなりません。関数に SQL が含まれる場合は、IBM.Data.DB2. を含める必要があります。

例 1: C# のパラメーター・スタイル SQL の表関数

この例では、以下について説明します。

- パラメーター・スタイル SQL の表関数の CREATE FUNCTION ステートメント
- パラメーター・スタイル SQL の表関数の C# コード

この表関数は、データ配列から作成された従業員データの行を含んだ表を戻します。この例には、2 つの関連クラスがあります。1 つは従業員を表すクラス `person` であり、もう 1 つはクラス `person` を使用するルーチン表 UDF を含んだクラス `empOps` です。従業員の給与情報は、入力パラメーターの値に基づいて更新されます。この例のデータ配列は、表関数を最初に呼び出したときに表関数そのものの中に作成されます。そのような配列は、ファイル・システム上のテキスト・ファイルからデータを読み取ることによっても作成できます。表関数のその後の呼び出しで配列データにアクセスするために、データの値がスクラッチパッドに書き込まれます。

表関数を呼び出すたびに、1 つのレコードが配列から読み取られ、1 つの行が関数によって戻される表の中に生成されます。行を表の中に生成する処理は、表関数の出力パラメーターを対象の行値に設定するという形で実行されます。表関数の最終呼び出しが行われた後、生成された行の表が戻されず。

表 38. C# のパラメーター・スタイル SQL の表関数を作成するためのコード

```
CREATE FUNCTION tableUDF(double)
RETURNS TABLE (name varchar(20),
                job varchar(20),
                salary double)
EXTERNAL NAME 'gwenUDF.dll:bizLogic.empOps!tableUDF'
LANGUAGE CLR
PARAMETER STYLE SQL
NOT DETERMINISTIC
FENCED
THREADSAFE
SCRATCHPAD 10
FINAL CALL
EXECUTION CONTROL SAFE
DISALLOW PARALLEL
NO DBINFO
```

表 38. C# のパラメーター・スタイル SQL の表関数を作成するためのコード (続き)

```
// The class Person is a supporting class for
// the table function UDF, tableUDF, below.
class Person
{
    private String name;
    private String position;
    private Int32 salary;

    public Person(String newName, String newPosition, Int32
newSalary)
    {
        this.name = newName;
        this.position = newPosition;
        this.salary = newSalary;
    }

    public String getName()
    {
        return this.name;
    }

    public String getPosition()
    {
        return this.position;
    }

    public Int32 getSalary()
    {
        return this.salary;
    }
}
```

表 38. C# のパラメーター・スタイル SQL の表関数を作成するためのコード (続き)

```

class empOps
{
    public static void TableUDF( Double factor, out String name,
                                out String position, out Double salary,
                                Int16 factorNullInd, out Int16 nameNullInd,
                                out Int16 positionNullInd, out Int16 salaryNullInd,
                                ref String sqlState, String funcName,
                                String specName, ref String sqlMessageText,
                                Byte[] scratchPad, Int32 callType)
    {
        Int16 intRow = 0;

        // Create an array of Person type information
        Person[] Staff = new
        Person[3];
        Staff[0] = new Person("Gwen", "Developer", 10000);
        Staff[1] = new Person("Andrew", "Developer", 20000);
        Staff[2] = new Person("Liu", "Team Leader", 30000);

        salary = 0;
        name = position = "";
        nameNullInd = positionNullInd = salaryNullInd = -1;

        switch(callType)
        {
            case (-2): // Case SQLUDF_TF_FIRST:
                break;

            case (-1): // Case SQLUDF_TF_OPEN:
                intRow = 1;
                scratchPad[0] = (Byte)intRow; // Write to scratchpad
                break;
            case (0): // Case SQLUDF_TF_FETCH:
                intRow = (Int16)scratchPad[0];
                if (intRow > Staff.Length)
                {
                    sqlState = "02000"; // Return an error SQLSTATE
                }
                else
                {
                    // Generate a row in the output table
                    // based on the Staff array data.
                    name =
                    Staff[intRow-1].getName();
                    position = Staff[intRow-1].getPosition();
                    salary = (Staff[intRow-1].getSalary()) * factor;
                    nameNullInd = 0;
                    positionNullInd = 0;
                    salaryNullInd = 0;
                }
                intRow++;
                scratchPad[0] = (Byte)intRow; // Write scratchpad
                break;

            case (1): // Case SQLUDF_TF_CLOSE:
                break;

            case (2): // Case SQLUDF_TF_FINAL:
                break;
        }
    }
}

```

例 2: C# のパラメーター・スタイル SQL のスカラー関数

この例では、以下について説明します。

- パラメーター・スタイル SQL のスカラー関数の CREATE FUNCTION ステートメント
- パラメーター・スタイル SQL のスカラー関数の C# コード

このスカラー関数は、操作対象の入力値ごとに 1 つのカウント値を返します。入力値セットの *n* 番目の桁にある入力値に対する出力スカラー値は *n* になります。スカラー関数の各呼び出しでは、行または値の入力セット内のそれぞれの行または値に 1 つの呼び出しが関連付けられており、呼び出しのたびにカウントが 1 つずつ増え、カウントの現行値が戻されます。そのカウントはスクラッチパッドのメモリー・バッファー内に保管されるので、スカラー関数の呼び出しと呼び出しの間でカウントの値が保たれるようになっています。

例えば、表を次のように定義している場合は、このスカラー関数を簡単に呼び出すことができます。

```
CREATE TABLE T (i1 INTEGER);
INSERT INTO T VALUES 12, 45, 16, 99;
```

このスカラー関数の呼び出しには、以下のような簡単な照会を使用できます。

```
SELECT countUp(i1) as count, i1 FROM T;
```

この照会の出力は次のようになります。

COUNT	I1
1	12
2	45
3	16
4	99

このスカラー UDF は非常に簡単です。スカラー関数を使用するときには、行のカウントだけを返す代わりに、データの形式を既存の列に合わせることもできます。例えば、住所列の各値にストリングを付加することや、一連の入力ストリングから複雑なストリングを組み立てることや、中間結果の保管先のデータ・セットに対して複雑な数値評価を行うことなども可能です。

表 39. C# のパラメーター・スタイル SQL のスカラー関数を作成するためのコード

```
CREATE FUNCTION countUp(INTEGER)
RETURNS INTEGER
LANGUAGE CLR
PARAMETER STYLE SQL
SCRATCHPAD 10
FINAL CALL
NO SQL
FENCED
THREADSAFE
NOT DETERMINISTIC
EXECUTION CONTROL SAFE
EXTERNAL NAME 'gwenUDF.d11:bizLogic.empOps!CountUp' ;
```


表 39. C# のパラメーター・スタイル SQL のスカラー関数を作成するためのコード (続き)

```

class empOps
{
    public static void CountUp(
        Int32 input,
        out Int32 outCounter,
        Int16 inputNullInd,
        out Int16 outCounterNullInd,
        ref String sqlState,
        String funcName,
        String specName,
        ref String sqlMessageText,
        Byte[] scratchPad,
        Int32 callType)
    {
        Int32 counter = 1;

        switch(callType)
        {
            case -1: // case SQLUDF_FIRST_CALL
                scratchPad[0] = (Byte)counter;
                outCounter = counter;
                outCounterNullInd = 0;
                break;
            case 0: // case SQLUDF_NORMAL_CALL:
                counter = (Int32)scratchPad[0];
                counter = counter + 1;
                outCounter = counter;
                outCounterNullInd = 0;
                scratchPad[0] =
                    (Byte)counter;
                break;
            case 1: // case SQLUDF_FINAL_CALL:
                counter =
                    (Int32)scratchPad[0];
                outCounter = counter;
                outCounterNullInd = 0;
                break;
            default: // Should never enter here
                // * Required so that at compile time
                //   out parameter outCounter is always set *
                outCounter = (Int32)(0);
                outCounterNullInd = -1;
                sqlState="ABCDE";
                sqlMessageText = "Should not get here: Default
                case!";
                break;
        }
    }
}

```

第 5 章 C および C++ ルーチン

C および C++ ルーチンとは、C または C++ ソース・コードでビルドされたライブラリーを外部コード本体として参照する CREATE PROCEDURE ステートメント、CREATE FUNCTION ステートメント、または CREATE METHOD ステートメントの実行によって作成される外部ルーチンです。

C および C++ ルーチンは、組み込み SQL ステートメントをオプションで SQL ステートメントを実行することができます。

C および C++ ルーチンのコンテキストで重要な用語は、以下のとおりです。

CREATE ステートメント

データベース内にルーチンを作成するために使用される SQL 言語の CREATE ステートメント。

ルーチン本体のソース・コード

CREATE ステートメントの EXTERNAL 節の仕様に対応する、C または C++ ルーチンのインプリメンテーションを含むソース・コード・ファイル。

プリコンパイラー

ルーチンのソース・コード・インプリメンテーションを事前構文解析する DB2 ユーティリティー。コードに含まれる SQL ステートメントを検証し、パッケージを生成します。

コンパイラー

ソース・コード・インプリメンテーションをコンパイルおよびリンクするために必要なプログラミング言語固有のソフトウェア。

パッケージ

DB2 がルーチンの実行時にルーチン・コード・インプリメンテーションに含まれる SQL ステートメントを実行するために使用する、実行時のアクセス・パス情報を含むファイル。

ルーチン・ライブラリー

コンパイルされた形式のルーチンのソース・コードが含まれるファイル。Windows の場合、これらのファイルには .dll ファイル拡張子が含まれるため、DLL と呼ばれることもあります。

C または C++ ルーチンを開発するには、ルーチンの基本と、C および C++ ルーチンにユニークなフィーチャーや特徴をあらかじめ理解しておくことが重要です。組み込み SQL API と、組み込み SQL アプリケーション開発の基本について理解しておくことも重要です。これらのテーマについてより理解を深めるには、以下のトピックを参照してください。

- 外部ルーチン
- 組み込み SQL
- C および C++ ルーチンの組み込みファイル
- C および C++ ルーチンでのパラメーター

- C および C++ ルーチンに関する制約事項

C または C++ ルーチンの開発には、一連の段階的な指示に従って、C または C++ ルーチン例を参照することが含まれます。次のトピックを参照してください。

- C および C++ ルーチンの作成
- C プロシージャの例
- C ユーザー定義関数の例

C での外部ルーチン開発のサポート

C で外部ルーチンを開発するには、サポートされるコンパイラおよび開発ソフトウェアを使用する必要があります。

C での DB2 データベース・アプリケーション開発のためのサポートされるコンパイラおよび開発ソフトウェアは、すべて C での外部ルーチン開発に使用できます。

C++ での外部ルーチン開発のサポート

C++ で外部ルーチンを開発するには、サポートされるコンパイラおよび開発ソフトウェアを使用する必要があります。

C での DB2 データベース・アプリケーション開発のためのサポートされるコンパイラおよび開発ソフトウェアは、すべて C++ での外部ルーチン開発に使用できます。

C および C++ ルーチン開発のためのツール

C および C++ ルーチンでサポートされるツールは、組み込み SQL C および C++ アプリケーションでサポートされるものと同じです。

組み込み SQL アプリケーションまたはルーチンの開発、デバッグ、またはデプロイのための DB2 開発環境またはグラフィカル・ユーザー・インターフェース・ツールはありません。

以下のコマンド行インターフェースは、一般に組み込み SQL アプリケーションおよびルーチンの開発、デバッグ、およびデプロイに使用されます。

- DB2 コマンド行プロセッサ
- DB2 コマンド・ウィンドウ

これらのインターフェースは、データベース内でルーチンを作成するのに必要な SQL ステートメントの実行をサポートします。C および C++ ルーチン (組み込み SQL が含まれる) をビルドするのに必要な PREPARE コマンドおよび BIND コマンドも、これらのインターフェースから発行することができます。

C および C++ ルーチンの設計

C および C++ ルーチンの設計は、C および C++ ルーチンを作成する前に行うべき作業です。C および C++ ルーチンの設計は一般に、他のプログラミング言語でインプリメントされる外部ルーチンの設計と、組み込み SQL アプリケーションの設計の両方に関連します。

前提条件

- 外部ルーチンの一般的な知識
- C または C++ プログラミングの経験
- オプション: 組み込み SQL または CLI アプリケーション開発の知識および経験 (ルーチンが SQL ステートメントを実行する場合)

以下のトピックは、必要な前提条件に関する情報の一部を提供しています。

外部ルーチンのフィーチャーおよび使用方法の詳細

- 24 ページの『外部ルーチンのインプリメンテーション』のトピックを参照してください。

組み込み SQL API の特性についての詳細

- 「組み込み SQL アプリケーションの開発」にある『組み込み SQL の概要』というトピックを参照してください。

すでに前提知識がある場合、組み込み SQL ルーチンの設計には、主に C および C++ ルーチンの固有のフィーチャーおよび特性について習得することが関係しています。

- 『C および C++ ルーチン開発に必要な組み込みファイル (sqludf.h)』
- 189 ページの『C および C++ ルーチンでのパラメーター』
- 191 ページの『パラメーター・スタイル SQL C および C++ プロシージャー』
- 194 ページの『パラメーター・スタイル SQL C および C++ 関数』
- 205 ページの『C および C++ ルーチンでの SQL データ・タイプ処理』
- 229 ページの『C および C++ ルーチンでのグラフィック・ホスト変数』
- 232 ページの『C および C++ プロシージャーからの結果セットの戻り』
- 230 ページの『C++ のタイプ修飾』
- 104 ページの『外部ルーチンに関する制約事項』

C および C++ の特性について理解した後、以下を参照することができます。

- 233 ページの『C および C++ ルーチンの作成』

C および C++ ルーチン開発に必要な組み込みファイル (sqludf.h)

sqludf.h 組み込みファイルには、ルーチン実装をコーディングする際に必要な構造体、定義、および値が入っています。過去の経緯のために、このファイルの名前には `udf` が含まれていますが、ストアード・プロシージャーやメソッドにも役立ちます。ルーチンのコンパイル時には、このファイルが含まれる DB2 インストール include ディレクトリーを参照する必要があります。

特定のオペレーティング・システム用およびオペレーティング・システム・ビット幅用の適切な C データ・タイプを確実に使用するために、このファイル内のオブジェクトを使用することをお勧めします。

sqludf.h ファイルには、構造体の定義、および構造体定義の記述が含まれています。以下にその内容について簡単に要約します。

- C または C++ 構造体としての表記を必要としない外部ルーチンへのパラメーターとしてサポートされる SQL データ・タイプ用のマクロ定義。ファイル内では、定義の名前は SQLUDF_x および SQLUDF_x_FBD という形式です。ここで x は SQL データ・タイプ名、FBD はバイナリー形式で保管されるデータ・タイプ用の FOR BIT DATA を表しています。

また、AS LOCATOR 節を使用して定義される引数や結果の C 言語タイプも含まれます。これは、UDF およびメソッドにのみ当てはまります。

- 以下の SQL データ・タイプおよび特殊パラメーターを表すために必要な C 構造体の定義。
 - VARCHAR FOR BIT DATA データ・タイプ
 - LONG VARCHAR データ・タイプ
 - LONG VARCHAR FOR BIT DATA データ・タイプ
 - LONG VARGRAPHIC データ・タイプ
 - BLOB データ・タイプ
 - CLOB データ・タイプ
 - DBCLOB データ・タイプ
 - スクラッチパッド構造
 - dbinfo 構造

これらは、単純な C データ・タイプではなく、複数のフィールド値を持つ構造としてそれぞれ表記されます。

スクラッチパッド構造はバッファーを定義し、これは関数呼び出し時にユーザー定義関数に渡されて使用されます。ただし、変数とは異なり、スクラッチパッドに保管されるデータは、1 つの呼び出し内の複数のユーザー定義関数呼び出しにわたって持続します。これは、集約値を戻す関数と、ロジックの初期セットアップを必要とする関数の両方において便利です。

dbinfo 構造とは、DBINFO 節がルーチンの CREATE ステートメントに含まれている場合に限って、追加の引数としてルーチン・インプリメンテーションと受け渡しできる、データベースおよびルーチンの情報が含まれる構造のことです。

- スクラッチパッドおよび呼び出しタイプ引数に対する C 言語型定義。呼び出しタイプ引数に対しては enum 型定義が指定されます。

外部ユーザー定義関数は、値のセットに対して複数回呼び出されます。呼び出しタイプは、個々の外部関数の呼び出しを識別するために使用されます。それぞれの呼び出しは、関数ロジック内で参照可能な呼び出しタイプの値で識別されます。例えば、ある関数の最初の呼び出し、データのフェッチ呼び出し、および最終呼び出しに関して特別な呼び出しタイプが存在します。呼び出しタイプは、特

定のロジックを特定の呼び出しタイプに関連付けることができるため便利です。呼び出しタイプには、例えば FIRST 呼び出し、FETCH 呼び出し、FINAL 呼び出しがあります。

- ユーザー定義関数 (UDF) プロトタイプで必要とされる標準的な末尾引数を定義するためのマクロ。末尾引数には、SQL 状態 (SQL-state)、関数名 (function-name)、特定名 (specific-name)、診断メッセージ (diagnostic-message)、スクラッチパッド (scratchpad)、および呼び出しタイプ (call-type) という UDF 呼び出し引数があります。また、これらの構造の参照、およびさまざまな SQLSTATE 有効値に対する定義も含まれます。マクロ定義はさまざまに異なり、スクラッチパッドや呼び出しタイプの引数を含んでいるマクロ定義もあれば、除外しているマクロ定義もあります。これらは、関数定義で SCRATCHPAD 節や FINAL CALL 節が使用される場合、および使用されない場合にそれぞれ対応しています。

通常、ユーザー定義関数を定義する際には、マクロ SQLUDF_TRAIL_ARGS を使用して関数プロトタイプを単純化することが推奨されます。例えば、次のようにします。

```
void SQL_API_FN ScalarUDF(SQLUDF_CHAR *inJob,
                          SQLUDF_DOUBLE *inSalary,
                          SQLUDF_DOUBLE *outNewSalary,
                          SQLUDF_SMALLINT *jobNullInd,
                          SQLUDF_SMALLINT *salaryNullInd,
                          SQLUDF_SMALLINT *newSalaryNullInd,
                          SQLUDF_TRAIL_ARGS)
```

- SQL 引数に NULL 値が含まれるかどうか検査するために使用できるマクロ定義。

ファイル sqludf.h に定義されているさまざまな定義、マクロ、構造体の使用方法については、C および C++ のサンプル・アプリケーションとルーチンを参照してください。

C および C++ ルーチンでのパラメーター

C および C++ ルーチンのパラメーター宣言は、サポートされているいずれかのパラメーター・スタイルまたはプログラム・タイプの要件を満たしている必要があります。ルーチンがスクラッチパッドを使用する場合や、dbinfo 構造を使用する場合や、PROGRAM TYPE MAIN パラメーター・インターフェースを使用する場合には、次の追加の考慮事項があります。

- 190 ページの『サポートされている C および C++ ルーチンのパラメーター・スタイル』
- 190 ページの『C および C++ ルーチンでのパラメーター NULL 標識』
- 191 ページの『パラメーター・スタイル SQL C および C++ プロシージャ』
- 194 ページの『パラメーター・スタイル SQL C および C++ 関数』
- 196 ページの『C および C++ ルーチンのパラメーターを値によって渡すか、参照によって渡すか』
- 197 ページの『パラメーターは C および C++ プロシージャの結果セットには必要ではない』
- 197 ページの『C または C++ ルーチン・パラメーターとしての Dbinfo 構造』
- 199 ページの『C または C++ 関数パラメーターとしてのスクラッチパッド』

- 200 ページの『C および C++ プロシージャでのプログラム・タイプ MAIN のサポート』

C および C++ ルーチンのパラメーター・インターフェースを正しくインプリメントすることは非常に重要です。これは、ほんの少し注意を払い、指定されたとおりにパラメーター・スタイルおよびデータ・タイプが確実に正しく選択され、インプリメントされるように見届けるだけで簡単に行えます。

サポートされている C および C++ ルーチンのパラメーター・スタイル

C および C++ ルーチンでは、以下のパラメーター・スタイルがサポートされます。

- SQL (プロシージャと関数に対応、推奨)
- GENERAL (プロシージャに対応)
- GENERAL WITH NULLS (プロシージャに対応)

すべての C および C++ ルーチンにパラメーター・スタイル SQL を使用することを強くお勧めします。このパラメーター・スタイルは NULL 値をサポートし、エラーの報告や、スクラッチパッドと呼び出しタイプのサポートなどのための標準インターフェースを備えています。

ルーチンで使用するパラメーター・スタイルを指定するには、ルーチン作成時にそのルーチンの CREATE ステートメント内に PARAMETER STYLE 節を指定する必要があります。

C または C++ ルーチン・コードのインプリメンテーションでは、そのパラメーター・スタイルを正確に反映しなければなりません。

これらのパラメーター・スタイルについて詳しくは、『Syntax for passing parameters to C and C++ routines』を参照してください。

C および C++ ルーチンでのパラメーター NULL 標識

C または C++ ルーチン (プロシージャまたは関数) 用に選択されたパラメーター・スタイルで、パラメーター・スタイル SQL および GENERAL で必要とされるように、NULL 標識パラメーターをそれぞれの SQL パラメーターに指定する必要がある場合、NULL 標識はデータ・タイプ SQLUDF_NULLIND* のパラメーターとして渡されます。パラメーター・スタイル GENERAL WITH NULLS の場合、タイプ SQLUDF_NULLIND の配列として渡す必要があります。このデータ・タイプは組み込み SQL アプリケーションおよびルーチン組み込みファイル sqludf.h で定義されます。

NULL 標識パラメーターは、対応するパラメーター値が SQL 内の NULL と同等かどうか、あるいはリテラル値が含まれるのかを示します。パラメーターの NULL 標識値が 0 の場合、パラメーター値が NULL ではないことを示します。パラメーターの NULL 標識値が -1 の場合、パラメーターは SQL 値 NULL と同等の値を持っていると見なされます。

NULL 標識を使用する場合、ルーチン内に以下を行うコードを含めることが重要です。

- 使用する前に、入力パラメーターの NULL 標識値をチェックする。
- ルーチンが戻す前に、出力パラメーターの NULL 標識値を設定する。

パラメーター SQL の詳細については、以下を参照してください。

- 82 ページの『外部ルーチンのパラメーター・スタイル』
- 『パラメーター・スタイル SQL C および C++ プロシージャ』
- 194 ページの『パラメーター・スタイル SQL C および C++ 関数』

パラメーター・スタイル SQL C および C++ プロシージャ

C および C++ プロシージャは、CREATE PROCEDURE ステートメント内の PARAMETER STYLE SQL 節を使用して作成する必要があります。このパラメーター・スタイルのパラメーター引き渡し規則は、対応するプロシージャ・コードのインプリメンテーションでインプリメントする必要があります。

プロシージャに必要な C および C++ PARAMETER STYLE SQL シグニチャーのインプリメンテーションは、以下の形式に従います。

```
SQL_API_RC SQL_API_FN function-name (
    SQL-arguments,
    SQL-argument-inds,
    sqlstate,
    routine-name,
    specific-name,
    diagnostic-message )
```

SQL_API_RC SQL_API_FN

SQL_API_RC および SQL_API_FN は、サポートされているオペレーティング・システムによって異なる可能性のある C または C++ プロシージャの戻りのタイプと呼び出し規則を指定するマクロです。C および C++ ルーチンには、マクロを使用する必要があります。このマクロは、組み込み SQL アプリケーションおよびルーチン組み込みファイル `sqlsystem.h` で宣言されます。

function-name

コード・ファイル内の C または C++ 関数の名前。この値は、対応する CREATE PROCEDURE ステートメント内で指定されたプロシージャの名前と同じである必要はありません。ただし、ライブラリー名と組み合わせるとこの値は、使用するライブラリー内で正しい関数エントリー・ポイントを識別するために、EXTERNAL NAME 節で指定する必要があります。C++ ルーチンの場合、C++ コンパイラーはタイプ修飾をエントリー・ポイント名に適用します。タイプ修飾名を EXTERNAL NAME 節に指定する必要がありますが、そうでない場合、ユーザー・コード内でエントリー・ポイントを `extern "C"` と定義しなければなりません。関数名は、明示的にエクスポートする必要があります。

SQL-arguments

CREATE PROCEDURE ステートメントで指定された SQL パラメーターのセットに対応する C または C++ 引数。IN、OUT、および INOUT モード・パラメーターが、個別のポインター値を使用して渡されます。

SQL-argument-inds

CREATE PROCEDURE ステートメントで指定された SQL パラメーターのセットに対応する C または C++ NULL 標識。それぞれの IN、OUT、お

よび INOUT モード・パラメーターに関して、関連した NULL 標識パラメーターがなければなりません。Null 標識は、タイプ SQLUDF_NULLIND の個別の引数として、または SQLUDF_NULLIND* として定義された NULL 標識の単一配列の一部として渡すことができます。

sqlstate

ルーチンで使用して、警告またはエラー条件を発信することができる入出力パラメーター値。通常この引数は、ユーザー定義の SQLSTATE 値を割り当てるのに使用し、その値は呼び出し側に戻すことのできるエラーまたは警告に対応しています。形式 38xxx の SQLSTATE 値 (xxx は任意の数値) を、ユーザー定義 SQLSTATE エラー値に使用できます。形式 01Hxx の SQLSTATE 値 (xx は任意の数値) を、ユーザー定義 SQLSTATE 警告値に使用できます。

routine-name

修飾ルーチン名を含む入力パラメーター値。この値は、DB2 によって生成され、<schema-name>.<routine-name> という形式でルーチンに渡されます。ここで、<schema-name> および <routine-name> は、SYSCAT.ROUTINES カタログ・ビュー内のルーチンの ROUTINESCHEMA 列値および ROUTINENAME 列値にそれぞれ相当します。この値は、複数の異なるルーチン定義で単一のルーチン・インプリメンテーションが使用される場合に役立ちます。ルーチン定義名がルーチンに渡されると、使用された定義に基づいてロジックを条件付きで実行できます。エラー・メッセージが含まれる診断情報を編成したり、ログ・ファイルに書き込んだりする場合には、ルーチン名も役立ちます。

specific-name

固有のルーチン特有の名前を含む入力パラメーター値。この値は DB2 によって生成されてルーチンに渡されます。この値は、SYSCAT.ROUTINES ビューのルーチンの SPECIFICNAME 列の値に対応します。ルーチン名と同じ方法で、使用できます。

diagnostic-message

呼び出し元のアプリケーションまたはルーチンにメッセージ・テキストを戻すのにルーチンがオプションで使用する出力パラメーター値。このパラメーターは、SQLSTATE 引数を補足するものとして使用することを目的とするものです。ユーザー定義のエラー・メッセージを割り当ててユーザー定義の SQLSTATE 値と一緒に使用できます。それにより、ルーチンの呼び出し側に詳細な診断エラー情報や警告情報を提供できます。

注: C および C++ プロシージャのシグニチャーの作成を単純化するには、sqludf.h で定義されたマクロ定義 SQLUDF_TRAIL_ARGS を、個々の引数を使用する代わりに、プロシージャのシグニチャーで使用することによって、非 SQL データ・タイプの引数をインプリメントできます。

以下に、単一の入力パラメーターを受け入れて、単一の出力パラメーターと結果セットを戻す C または C++ プロシージャのインプリメンテーションの例を示します。

```
/******  
Routine:  cstp  
  
Purpose:  Returns an output parameter value based on an input
```

parameter value

Shows how to:

- define a procedure using PARAMETER STYLE SQL
- define NULL indicators for the parameter
- execute an SQL statement
- how to set a NULL indicator when parameter is not null

Parameters:

IN: inParm
OUT: outParm

When PARAMETER STYLE SQL is defined for the routine (see routine registration script spcreate.db2), in addition to the parameters passed during invocation, the following arguments are passed to the routine in the following order:

- one null indicator for each IN/INOUT/OUT parameter ordered to match order of parameter declarations
- SQLSTATE to be returned to DB2 (output)
- qualified name of the routine (input)
- specific name of the routine (input)
- SQL diagnostic string to return an optional error message text to DB2 (output)

See the actual parameter declarations below to see the recommended datatypes and sizes for them.

CODE TIP:

Instead of coding the 'extra' parameters: sqlstate, qualified name of the routine, specific name of the routine, diagnostic message, a macro SQLUDF_TRAIL_ARGS can be used instead. This macro is defined in DB2 include file sqludf.h

TIP EXAMPLE:

The following is equivalent to the actual prototype used that makes use of macro definitions included in sqludf.h. The form actually implemented is simpler and removes datatype concerns.

```
extern "C" SQL_API_RC SQL_API_FN OutLanguage(
    sqlint16 *inParm,
    double *outParm,
    sqlint16 *inParmNullInd,
    sqlint16 *outParmNullInd,
    char sqlst[6],
    char qualName[28],
    char specName[19],
    char diagMsg[71])
)

*****/

extern "C" SQL_API_RC SQL_API_FN cstp ( sqlint16 *inParm,
    double *outParm,
    SQLUDF_NULLIND *inParmNullInd,
    SQLUDF_NULLIND *outParmNullInd,
    SQLUDF_TRAIL_ARGS )
{
    EXEC SQL INCLUDE SQLCA;
}
```

```

EXEC SQL BEGIN DECLARE SECTION;
    sqlint16 sql_inParm;
EXEC SQL END DECLARE SECTION;

sql_inParm = *inParm;

EXEC SQL DECLARE cur1 CURSOR FOR
    SELECT value
    FROM table01
    WHERE index = :sql_inParm;

*outParm = (*inParm) + 1;
*outParmNullInd = 0;

EXEC SQL OPEN cur1;

return (0);
}

```

このプロシーチャーの対応する CREATE PROCEDURE ステートメントは次のとおりです。

```

CREATE PROCEDURE cproc( IN inParm INT, OUT outParm INT )
LANGUAGE c
PARAMETER STYLE sql
DYNAMIC RESULT SETS 1
FENCED
THREADSAFE
RETURNS NULL ON NULL INPUT
EXTERNAL NAME 'c_rtms!cstp'

```

上記のステートメントでは、C または C++ プロシーチャーのインプリメンテーションは、Linux または UNIX 環境内では c_rtms というライブラリー・ファイル内に、および cstp という関数内にあることが前提になっています。

パラメーター・スタイル SQL C および C++ 関数

C および C++ ユーザー定義関数は、CREATE FUNCTION ステートメント内の PARAMETER STYLE SQL 節を使用して作成する必要があります。このパラメーター・スタイルのパラメーター引き渡し規則は、対応するソース・コードのインプリメンテーションでインプリメントする必要があります。ユーザー定義関数に必要な C および C++ PARAMETER STYLE SQL シグニチャーのインプリメンテーションは、以下の形式に従います。

```

SQL_API_RC SQL_API_FN function-name ( SQL-arguments,
    SQL-argument-inds,
    SQLUDF_TRAIL_ARGS )

```

SQL_API_RC SQL_API_FN

SQL_API_RC および SQL_API_FN は、サポートされているオペレーティング・システムによって異なる可能性のある C または C++ ユーザー定義関数の戻りのタイプと呼び出し規則を指定するマクロです。C および C++ ルーチンには、マクロを使用する必要があります。このマクロは、組み込み SQL アプリケーションおよびルーチン組み込みファイル sqlsystem.h で宣言されます。

function-name

コード・ファイル内の C または C++ 関数の名前。この値は、対応する CREATE FUNCTION ステートメント内で指定された関数の名前と同じである必要はありません。ただし、ライブラリー名と組み合わせたこの値は、使

用するライブラリー内で正しい関数エントリー・ポイントを識別するために、EXTERNAL NAME 節で指定する必要があります。C++ ルーチンの場合、C++ コンパイラーはタイプ修飾をエントリー・ポイント名に適用します。タイプ修飾名を EXTERNAL NAME 節で指定するか、または以下の例で示されるようにソース・コード・ファイル内の関数宣言に extern "C" という接頭部を付ける必要があります。extern "C" SQL_API_RC SQL_API_FN OutLanguage(char *, sqlint16 *, char *, char *, char *, char *);

SQL-arguments

CREATE FUNCTION ステートメントで指定された SQL パラメーターのセットに対応する C または C++ 引数。

SQL-argument-inds

SQL-argument ごとに、ルーチンのインプリメンテーション内でパラメーター値を SQL の NULL 値として解釈するかどうかを指定するために、NULL 標識パラメーターが必要になります。NULL 標識は、データ・タイプ SQLUDF_NULLIND を使用して指定する必要があります。このデータ・タイプは、組み込み SQL ルーチンの組み込みファイル sqludf.h で定義されます。

SQLUDF_TRAIL_ARGS

すでに拡張された、組み込み SQL ルーチンの組み込みファイル sqludf.h で定義されるマクロ。完全なパラメーター・スタイル SQL シグニチャーに必要な追加の後続の引数を定義します。使用できるマクロには、SQLUDF_TRAIL_ARGS および SQLUDF_TRAIL_ARGS_ALL の 2 つがあります。sqludf.h で定義されたように、SQLUDF_TRAIL_ARGS は拡張されると、以下のルーチン引数を追加することに相当します。

```
SQLUDF_CHAR *sqlState,  
SQLUDF_CHAR qualName,  
SQLUDF_CHAR specName,  
SQLUDF_CHAR *sqlMessageText,
```

一般に、これらの引数は必要ではないか、あるいはユーザー定義関数ロジックの一部として使用されます。これらは、関数呼び出し側に渡される出力 SQLSTATE 値、入力の完全修飾関数名、入力の関数固有名、および SQLSTATE を使用して戻される出力メッセージ・テキストを表します。sqludf.h で定義されたように、SQLUDF_TRAIL_ARGS_ALL は拡張されると、以下のルーチン引数を追加することに相当します。

```
SQLUDF_CHAR qualName,  
SQLUDF_CHAR specName,  
SQLUDF_CHAR sqlMessageText,  
SQLUDF_SCRAT *scratchpad  
SQLUDF_CALLT *callType
```

UDF CREATE ステートメントに SCRATCHPAD 節または FINAL CALL 節が含まれる場合、マクロ SQLUDF_TRAIL_ARGS_ALL を使用する必要があります。SQLUDF_TRAIL_ARGS で提供される引数に加えて、このマクロには、スクラッチパッド構造へのポインター、および呼び出しタイプの値も含まれません。

以下に、2 つの入力パラメーター値の積の値を出力パラメーターに戻す、簡単な C または C++ UDF の例を示します。

```

SQL_API_RC SQL_API_FN product ( SQLUDF_DOUBLE *in1,
                                SQLUDF_DOUBLE *in2,
                                SQLUDF_DOUBLE *outProduct,
                                SQLUDF_NULLIND *in1NullInd,
                                SQLUDF_NULLIND *in2NullInd,
                                SQLUDF_NULLIND *productNullInd,
                                SQLUDF_TRAIL_ARGS )
{
    /* Check that input parameter values are not null
       by checking the corresponding null indicator values
       0 : indicates parameter value is not NULL
       -1 : indicates parameter value is NULL

       If values are not NULL, calculate the product.
       If values are NULL, return a NULL output value. */

    if ((*in1NullInd != -1) &&
        *in2NullInd != -1)
    {
        *outProduct = (*in1) * (*in2);
        *productNullInd = 0;
    }
    else
    {
        *productNullInd = -1;
    }
    return (0);
}

```

この UDF の作成に使用できる、対応する CREATE FUNCTION ステートメントは以下のとおりです。

```

CREATE FUNCTION product( in1 DOUBLE, in2 DOUBLE )
  RETURNS DOUBLE
  LANGUAGE C
  PARAMETER STYLE SQL
  NO SQL
  FENCED THREADSAFE
  DETERMINISTIC
  RETURNS NULL ON NULL INPUT
  NO EXTERNAL ACTION
  EXTERNAL NAME 'c_rtms!product'

```

上記の SQL ステートメントでは、C または C++ 関数は c_rtms という function ディレクトリーのライブラリー・ファイル内にあることが前提になっています。

C および C++ ルーチンのパラメーターを値によって渡すか、参照によって渡すか

C および C++ ルーチンの場合、パラメーター値は、必ずポインターを使用してルーチンへの参照によって渡す必要があります。これは、参照によって、入力専用パラメーター、入出力パラメーター、および出力パラメーターで必要となります。

NULL 標識パラメーターも、ポインターを使用してルーチンへの参照によって渡さなければなりません。

注: DB2 は、すべてのパラメーターに関するメモリーの割り振りを制御し、ルーチンとの間で受け渡しが行われるすべてのパラメーターへの C または C++ 参照を管理します。ルーチン・パラメーターおよび NULL 標識に関連したメモリーを割り振ったり解放したりする必要はありません。

パラメーターは C および C++ プロシージャの結果セットには必要ではない

プロシージャの CREATE PROCEDURE ステートメントのシグニチャー、または関連したプロシージャのインプリメンテーションでは、呼び出し側に結果セットを戻すためにパラメーターは必要ありません。

C プロシージャから戻される結果セットは、カーソルを使用して戻されます。

LANGUAGE C プロシージャから結果セットを戻す方法の詳細については、以下を参照してください。

- 232 ページの『C および C++ プロシージャからの結果セットの戻り』

C または C++ ルーチン・パラメーターとしての Dbinfo 構造

dbinfo 構造とは、DBINFO 節がルーチンの CREATE ステートメントに含まれている場合に限って、追加の引数としてルーチン・インプリメンテーションと受け渡しできる、データベースおよびルーチンの情報が含まれる構造のことです。

dbinfo 構造は、sqludf_dbinfo 構造を使用してサポートされている LANGUAGE C ルーチンです。この C 構造体は、sqllib¥include ディレクトリーにある DB2 組み込みファイル sqludf.h で定義されます。

sqludf_dbinfo 構造は、以下のように定義されます。

```
SQL_STRUCTURE sqludf_dbinfo
{
    unsigned short  dbnamelen;           /* Database name length */
    unsigned char   dbname[SQLUDF_MAX_IDENT_LEN]; /* Database name */
    unsigned short  authidlen;          /* Authorization ID length */
    unsigned char   authid[SQLUDF_MAX_IDENT_LEN]; /* Authorization ID */
    union db_cdpq   codepg;             /* Database code page */
    unsigned short  tbschemalen;        /* Table schema name length */
    unsigned char   tbschema[SQLUDF_MAX_IDENT_LEN]; /* Table schema name */
    unsigned short  tbnamelen;          /* Table name length */
    unsigned char   tbname[SQLUDF_MAX_IDENT_LEN]; /* Table name */
    unsigned short  colnamelen;         /* Column name length */
    unsigned char   colname[SQLUDF_MAX_IDENT_LEN]; /* Column name */
    unsigned char   ver_rel[SQLUDF_SH_IDENT_LEN]; /* Database version/release */
    unsigned char   resd0[2];           /* Alignment */
    sqluint32       platform;           /* Platform */
    unsigned short  numtfcol;           /* # of entries in TF column */
                                           /* List array */
    unsigned char   resd1[2];           /* Reserved */
    sqluint32       procid;             /* Current procedure ID */
    unsigned char   resd2[32];          /* Reserved */
    unsigned short  *tfcolumn;          /* Tfcolumn to be allocated */
                                           /* dynamically if a table */
                                           /* function is defined; */
                                           /* else a NULL pointer */
    char            *appl_id;           /* Application identifier */
    sqluint32       dbpartitionnum;    /* Database partition number */
                                           /* where routine executed */
    unsigned char   resd3[16];          /* Reserved */
};
```

dbinfo 構造内のすべてのフィールドがルーチンで役に立つわけではありませんが、この構造のフィールドにおける幾つかの値は診断エラー・メッセージ情報を編成する際に役立つ可能性があります。例えば、ルーチンでエラーが発生する場合、デー

データベース名、データベース名の長さ、データベース・コード・ページ、現行の許可 ID、およびその現行許可 ID の長さが戻されると有用な場合があります。

LANGUAGE C ルーチン・インプリメンテーションで sqludf_dbinfo 構造を参照するには、以下のようにします。

- DBINFO 節を、ルーチンを定義する CREATE ステートメントに追加します。
- ルーチン・インプリメンテーションが含まれているファイルの先頭に、sqludf.h ヘッダー・ファイルを組み込みます。
- タイプ sqludf_dbinfo のパラメーターを、使用するパラメーター・スタイルで指定された位置にあるルーチンのシグニチャーに追加します。

以下は、PARAMETER STYLE GENERAL を使用した C プロシージャの例で、dbinfo 構造の使用法を示しています。そのプロシージャの CREATE PROCEDURE ステートメントを以下に記します。EXTERNAL NAME 節で指定されているように、プロシージャ・インプリメンテーションは spserver という名前のライブラリー・ファイルに入れられ、DbinfoExample という C 関数が含まれています。

```
CREATE PROCEDURE DBINFO_EXAMPLE (IN job CHAR(8),
                                OUT salary DOUBLE,
                                OUT dbname CHAR(128),
                                OUT dbversion CHAR(8),
                                OUT errorcode INTEGER)

DYNAMIC RESULT SETS 0
LANGUAGE C
PARAMETER STYLE GENERAL
DBINFO
FENCED NOT THREADSAFE
READS SQL DATA
PROGRAM TYPE SUB
EXTERNAL NAME 'spserver!DbinfoExample'@
```

以下は、このプロシージャ定義に対応した C プロシージャ・インプリメンテーションです。

```

/*****
Routine:    DbinfoExample

IN:         inJob          - a job type, used in a SELECT predicate
OUT:        salary         - average salary of employees with job injob
           dbname         - database name retrieved from DBINFO
           dbversion      - database version retrieved from DBINFO
           outSqlError    - sqlcode of error raised (if any)
           sqludf_dbinfo - pointer to DBINFO structure

Purpose:    This routine takes in a job type and returns the
           average salary of all employees with that job, as
           well as information about the database (name,
           version of database). The database information
           is retrieved from the dbinfo object.

Shows how to:
           - define IN/OUT parameters in PARAMETER STYLE GENERAL
           - declare a parameter pointer to the dbinfo structure
           - retrieve values from the dbinfo structure
*****/
SQL_API_RC SQL_API_FN DbinfoExample(char inJob[9],
                                     double *salary,
                                     char dbname[129],
                                     char dbversion[9],
                                     sqlint32 *outSqlError,
```



```

                                struct sqludf_dbinfo * dbinfo
                                )
{
    /* Declare a local SQLCA */
    struct sqlca sqlca;

    EXEC SQL WHENEVER SQLERROR GOTO return_error;

    /* SQL host variable declaration section */
    /* Each host variable names must be unique within a code
       file, or the the precompiler raises SQL0307 error */
    EXEC SQL BEGIN DECLARE SECTION;
    char dbinfo_injob[9];
    double dbinfo_outsalary;
    sqlint16 dbinfo_outsalaryind;
    EXEC SQL END DECLARE SECTION;

    /* Initialize output parameters - se strings to NULL */
    memset(dbname, '%0', 129);
    memset(dbversion, '%0', 9);
    *outSqlError = 0;

    /* Copy input parameter into local host variable */
    strcpy(dbinfo_injob, inJob);

    EXEC SQL SELECT AVG(salary) INTO:dbinfo_outsalary
              FROM employee
              WHERE job =:dbinfo_injob;

    *salary = dbinfo_outsalary;

    /* Copy values from the DBINFO structure into the output parameters
       You must explicitly null-terminate the strings.
       Information such as the database name, and the version of the
       database product can be found in the DBINFO structure as well as
       other information fields. */

    strncpy(dbname, (char *) (dbinfo->dbname), dbinfo->dbnamelen);
    dbname[dbinfo->dbnamelen] = '%0';
    strncpy(dbversion, (char *) (dbinfo->ver_rel), 8);
    dbversion[8] = '%0';

    return 0;

    /* Copy SQLCODE to OUT parameter if SQL error occurs */

    return_error:
    {
        *outSqlError = SQLCODE;
        EXEC SQL WHENEVER SQLERROR CONTINUE;
        return 0;
    }
} /* DbinfoExample function */

```

C または C++ 関数パラメーターとしてのスクラッチパッド

C および C++ ルーチンでは、UDF 入力値ごとの呼び出しの間で UDF 値を保管するのに使用するスクラッチパッド構造、sqludf_scrat 構造の使用によってサポートされています。この C 構造体は、DB2 組み込みファイル sqludf.h で定義されます。

sqludf_scrat 構造を参照するには、sqludf.h ヘッダー・ファイルを C または C++ 関数のインプリメンテーションを含むファイルの上部に組み込み、ルーチンのインプリメンテーションのシグニチャー内で SQLUDF_TRAIL_ARGS_ALL マクロを使用します。

以下の例は、SQLUDF_TRAIL_ARGS_ALL というタイプのパラメーターを含む C スカラー関数のインプリメンテーションを示しています。

```
#ifdef __cplusplus
extern "C"
#endif
void SQL_API_FN ScratchpadScUDF(SQLUDF_INTEGER *outCounter,
                               SQLUDF_SMALLINT *counterNullInd,
                               SQLUDF_TRAIL_ARGS_ALL)
{
    struct scalar_scratchpad_data *pScratData;

    /* SQLUDF_CALLT and SQLUDF_SCRAT are */
    /* parts of SQLUDF_TRAIL_ARGS_ALL */

    pScratData = (struct scalar_scratchpad_data *)SQLUDF_SCRAT->data;
    switch (SQLUDF_CALLT)
    {
        case SQLUDF_FIRST_CALL:
            pScratData->counter = 1;
            break;
        case SQLUDF_NORMAL_CALL:
            pScratData->counter = pScratData->counter + 1;
            break;
        case SQLUDF_FINAL_CALL:
            break;
    }

    *outCounter = pScratData->counter;
    *counterNullInd = 0;
} /* ScratchpadScUDF */
```

SQLUDF_TRAIL_ARGS_ALL マクロは展開して他のパラメーター値を定義します。これには SQLUDF_SCRAT という名前の、バッファー・パラメーターがスクラッチパッドとして使用されることを定義するパラメーター値が含まれます。値のセットに対してスカラー関数が呼び出されると、そのスカラー関数が呼び出されるたびにバッファーがパラメーターとして関数に渡されます。アクセスにはバッファーを使用できます。

SQLUDF_TRAIL_ARGS_ALL マクロ値は別の SQLUDF_CALLT というパラメーターの定義も行います。このパラメーターは呼び出しタイプの値を示すために使用されます。呼び出しタイプの値を使用することにより、関数が値のセットについてその最初に呼び出されているのか、最後に呼び出されているのか、それとも処理の途中で呼び出されているのかを識別できます。

C および C++ プロシージャでのプログラム・タイプ MAIN のサポート

通常 C プロシージャの PROGRAM TYPE 節ではデフォルト値 SUB が推奨されますが、LANGUAGE 節の値が C の CREATE PROCEDURE ステートメントでは、PROGRAM TYPE 節の値として MAIN がサポートされています。

PROGRAM TYPE 節の値 MAIN は、91 個以上のパラメーターを持つルーチンで必要になります。

PROGRAM TYPE MAIN 節を指定する場合は、C ソース・コード・ファイルのメインルーチンのデフォルト・スタイルと整合したシグニチャーを使用して、プロシージャをインプリメントする必要があります。これは、main という名前の関数によってルーチンをインプリメントするという意味ではなく、通常は C プログラミング

の argc および argv 引数を使用して、デフォルト・タイプのメインルーチンのアプリケーション・インプリメンテーションに多くの場合関連付けられた形式でパラメーターを渡す必要があるということです。

以下は、PGRAM TYPE MAIN 仕様に従った C または C++ ルーチンのシグニチャーの例です。

```
SQL_API_RC SQL_API_FN functionName(int argc, char **argv)
{
    ...
}
```

関数の引数の総数は、argc の値で指定されます。引数値は、argv 配列内の配列エレメントとして渡されます。引数の数と順序は、CREATE PROCEDURE ステートメントで指定された PARAMETER STYLE 節の値によって決まります。

例として、PROGRAM TYPE MAIN スタイルと推奨された PARAMETER STYLE SQL を持つように指定した、C プロシージャー用の次の CREATE PROCEDURE ステートメントを考えます。

```
CREATE PROCEDURE MAIN_EXAMPLE (
    IN job CHAR(8),
    OUT salary DOUBLE)
SPECIFIC CPP_MAIN_EXAMPLE
DYNAMIC RESULT SETS 0
NOT DETERMINISTIC
LANGUAGE C
PARAMETER STYLE SQL
NO DBINFO
FENCED NOT THREADSAFE
READS SQL DATA
PROGRAM TYPE MAIN
EXTERNAL NAME 'spserver!MainExample'@
```

この CREATE PROCEDURE ステートメントに対応するルーチン・シグニチャーのインプリメンテーションは、次のとおりです。

```
/**
// Stored Procedure: MainExample
//
// SQL parameters:
//   IN:   argv[1] - job   (char[8])
//   OUT:  argv[2] - salary (double)
//
SQL_API_RC SQL_API_FN MainExample(int argc, char **argv)
{
    ...
}
```

プロシージャー呼び出し時に渡された SQL パラメーター値に加えて、PARAMETER STYLE SQL が使用されているため、そのスタイルに必要な追加パラメーターもルーチンに渡されます。

パラメーター値は、ソース・コード内の目的の argv 配列エレメントを参照することによりアクセスできます。上の例の場合、argc および argv 配列エレメントには、以下の値が含まれます。

```
argc   : Number of argv array elements
argv[0]: The function name
argv[1]: Value of parameter job (char[8], input)
argv[2]: Value of parameter salary (double, output)
```

```

argv[3]: null indicator for parameter job
argv[4]: null indicator for parameter salary
argv[5]: sqlstate (char[6], output)
argv[6]: qualName (char[28], output)
argv[7]: specName (char[19], output)
argv[8]: diagMsg (char[71], output)

```

C および C++ ルーチンにおいてサポートされる SQL データ・タイプ

次の表は、ルーチンの SQL データ・タイプと C データ・タイプの間でサポートされるマッピングをリストしています。C/C++ の各データ・タイプには、sqludf.h で定義されている対応するタイプが付記されています。

表 40. C/C++ 宣言にマップされた SQL データ・タイプ

SQL 列名	C/C++ データ・タイプ	SQL 列タイプ記述
SMALLINT	sqlint16 SQLUDF_SMALLINT	16 ビットの符号付き整数
INTEGER	sqlint32 SQLUDF_INTEGER	32 ビットの符号付き整数
BIGINT	sqlint64 SQLUDF_BIGINT	64 ビットの符号付き整数
REAL FLOAT(<i>n</i>)。ただし $1 \leq n \leq 24$ 。	float SQLUDF_REAL	単精度浮動小数点
DOUBLE FLOAT FLOAT(<i>n</i>)。ただし $25 \leq n \leq 53$ 。	double SQLUDF_DOUBLE	倍精度浮動小数点
DECIMAL(<i>p</i> , <i>s</i>)	サポートされていません。	10 進数値を渡すには、パラメーターを DECIMAL からキャスト可能なデータ・タイプ (例えば CHAR または DOUBLE) に定義し、明示的に引数をこのタイプにキャストします。
CHAR(<i>n</i>)	char[<i>n</i> +1] (<i>n</i> はデータを収容するのに十分な大きさ) $1 \leq n \leq 254$ SQLUDF_CHAR	固定長、NULL 終了文字ストリング
CHAR(<i>n</i>) FOR BIT DATA	char[<i>n</i>] (<i>n</i> はデータを収容するのに十分な大きさ) $1 \leq n \leq 254$ SQLUDF_CHAR	固定長、非 NULL 終了文字ストリング
VARCHAR(<i>n</i>)	char[<i>n</i> +1] (<i>n</i> はデータを収容するのに十分な大きさ) $1 \leq n \leq 32\ 672$ SQLUDF_VARCHAR	NULL 終了可変長ストリング

表 40. C/C++ 宣言にマップされた SQL データ・タイプ (続き)

SQL 列名	C/C++ データ・タイプ	SQL 列タイプ記述
VARCHAR(<i>n</i>) FOR BIT DATA	<pre>struct { sqluint16 length; char[n] }</pre> <p>1<=<i>n</i><=32 672</p> <p>SQLUDF_VARCHAR_FBD</p>	非 NULL 終了の可変長文字ストリング
LONG VARCHAR	<pre>struct { sqluint16 length; char[n] }</pre> <p>1<=<i>n</i><=32 700</p> <p>SQLUDF_LONG</p>	非 NULL 終了の可変長文字ストリング
CLOB(<i>n</i>)	<pre>struct { sqluint32 length; char data[n]; }</pre> <p>1<=<i>n</i><=2 147 483 647</p> <p>SQLUDF_CLOB</p>	4 バイト・ストリング長標識をもった、非 NULL 終了の可変長文字ストリング
BLOB(<i>n</i>)	<pre>struct { sqluint32 length; char data[n]; }</pre> <p>1<=<i>n</i><=2 147 483 647</p> <p>SQLUDF_BLOB</p>	4 バイト・ストリング長標識をもった、非 NULL 終了の可変バイナリー・ストリング
DATE	<pre>char[11] SQLUDF_DATE</pre>	以下のフォーマットの NULL 終了文字ストリング。 yyyy-mm-dd
TIME	<pre>char[9] SQLUDF_TIME</pre>	以下のフォーマットの NULL 終了文字ストリング。 hh.mm.ss

表 40. C/C++ 宣言にマップされた SQL データ・タイプ (続き)

SQL 列名	C/C++ データ・タイプ	SQL 列タイプ記述
TIMESTAMP	char[20] - char[33] SQLUDF_STAMP	<p>以下のフォーマットの NULL 終了文字スト リング。</p> <p>yyyy-mm-dd-hh.mm.ss.nnnnnnnnnnn</p> <p>文字ストリングは、秒の小数部分の指定され た桁数に応じて、19 から 32 バイトの長さ にすることができます。TIMESTAMP デー タ・タイプの秒の小数部分には、オプション で、0 から 12 桁のタイム・スタンプ精度を 指定できます。</p> <p>以下に例を示します。</p> <pre>(VALUES(CURRENT_TIMESTAMP(0)) 1 ----- 2008-07-09-14.48.36 1 record(s) selected.</pre> <pre>LENGTH (VALUES(CURRENT_TIMESTAMP(0)) 1 ----- 19 1 record(s) selected. (VALUE(CURRENT_TIMESTAMP(12)) 1 ----- 2008-07-09-14.48.36.123456789012 1 record(s) selected.</pre> <pre>LENGTH (VALUES(CURRENT_TIMESTAMP(0)) 1 ----- 32</pre> <p>タイム・スタンプ値が、秒の小数部分の桁数 が異なるタイム・スタンプ変数に割り当てら れると、その値はタイム・スタンプ変数のフ ォーマットに合うように切り捨てられるか、0 を埋め込むことによって調整されます。</p>
LOB LOCATOR	sqluint32 SQLUDF_LOCATOR	32 ビットの符号付き整数
GRAPHIC(<i>n</i>)	sqlbchar[<i>n</i> +1] (<i>n</i> はデータを収容 するのに十分な大きさ) 1<= <i>n</i> <=127 SQLUDF_GRAPH	固定長、NULL 終了 2 バイト文字ストリン グ

表 40. C/C++ 宣言にマップされた SQL データ・タイプ (続き)

SQL 列名	C/C++ データ・タイプ	SQL 列タイプ記述
VARGRAPHIC(<i>n</i>)	sqldbchar[<i>n</i> +1] (<i>n</i> はデータを収容するの十分な大きさ) $1 \leq n \leq 16\ 336$ SQLUDF_GRAPH	NULL 終了可変長 2 バイト文字ストリング
LONG VARGRAPHIC	<pre>struct { sqluint16 length; sqldbchar[n] }</pre> $1 \leq n \leq 16\ 350$ SQLUDF_LONGVARG	非 NULL 終了の可変長 2 バイト文字ストリング
DBCLOB(<i>n</i>)	<pre>struct { sqluint32 length; sqldbchar data[n]; }</pre> $1 \leq n \leq 1\ 073\ 741\ 823$ SQLUDF_DBCLOB	4 バイト・ストリング長標識をもった、非 NULL 終了の可変長文字ストリング
XML AS CLOB	<pre>struct { sqluint32 length; char data[n]; }</pre> $1 \leq n \leq 2\ 147\ 483\ 647$ SQLUDF_CLOB	4 バイト・ストリング長標識をもった、非 NULL 終了の可変長シリアライズ化文字ストリング

注: XML データ・タイプは、C または C++ でインプリメントされている外部ルーチン内で、CLOB データ・タイプとしてのみインプリメントすることができます。

注: 以下のデータ・タイプは、WCHARTYPE NOCONVERT オプションを指定してプリコンパイルした場合に DBCS または EUC 環境でのみ使用できます。

- GRAPHIC(*n*)
- VARGRAPHIC(*n*)
- LONG VARGRAPHIC
- DBCLOB(*n*)

C および C++ ルーチンでの SQL データ・タイプ処理

この項では、ルーチンのパラメーターと結果の有効なタイプを明らかにし、それに対応する引数を C や C++ 言語のルーチンでどのように定義すればよいかを指定します。ルーチンのすべての引数は、該当するデータ・タイプにポインターとして渡す必要があります。sqludf.h 組み込みファイルとそこで定義されるタイプを使用

すると、さまざまなデータ・タイプおよびコンパイラーに当てはまる言語変数および構造を自動的に生成できます。例えば、BIGINT では、SQLUDF_BIGINT データ・タイプを使用すれば、コンパイラーが異なっても、BIGINT の表現に必要なタイプの違いを隠すことができます。

それは、引数値のフォーマットを統括するルーチンの CREATE ステートメントに定義される各パラメーターのデータ・タイプです。適切なフォーマットで値を受け取るには、引数のデータ・タイプからのプロモーションが必要なことがあります。DB2 は、引数値に対してこのようなプロモーションを自動的に実行します。ただし、ルーチン・コードで誤ったデータ・タイプを指定すると、データの消失や異常終了などの不測の振る舞いが発生します。

スカラー関数またはメソッドの結果の場合、フォーマットを定義するのは、CREATE FUNCTION ステートメントの CAST FROM 節で指定するデータ・タイプです。CAST FROM 節がない場合は、RETURNS 節で指定されるデータ・タイプがフォーマットを定義します。

以下の例での CAST FROM 節は、ルーチン本体が SMALLINT を戻し、DB2 がその値を関数の参照を行うステートメントに渡す前に INTEGER にキャストすることを意味します。

```
... RETURNS INTEGER CAST FROM SMALLINT ...
```

この場合、この項の後半で定義されているとおり、SMALLINT を生成するようにルーチンを作成しなければなりません。CAST FROM データ・タイプは RETURNS データ・タイプに対してキャスト可能 でなければならないため、任意に他のデータ・タイプを選ぶことはできません。

以下に、SQL タイプとその C/C++ 言語での表示を示します。また、それぞれのタイプがパラメーターや結果として有効かどうかを説明します。さらに、そのタイプを C や C++ 言語のルーチンで定義される引数として表した例も示します。

- SMALLINT

正しい例。C で SQLUDF_SMALLINT または sqlint16 で表します。

以下に例を示します。

```
sqlint16 *arg1; /* example for SMALLINT */
```

整数のルーチン・パラメーターを定義する際は、SMALLINT ではなく INTEGER の使用をご検討ください。それは、DB2 は INTEGER 引数を SMALLINT にはプロモートしないからです。例えば、UDF を次のように定義するとします。

```
CREATE FUNCTION SIMPLE(SMALLINT)...
```

INTEGER データ (... SIMPLE(1)...) を使用して SIMPLE 関数を呼び出すと、関数が見つからないことを示す SQLCODE -440 (SQLSTATE 42884) エラーが出されますが、この関数のエンド・ユーザーはそのメッセージの原因を理解できないことがあります。前の例では 1 は INTEGER であるため、それを SMALLINT にキャストすることも INTEGER としてパラメーターを定義することもできます。

- INTEGER または INT

正しい例。C で SQLUDF_INTEGER または sqlint32 として表します。#include sqludf.h または #include sqlsystem.h を指定して、定義を選出する必要があります。

以下に例を示します。

```
sqlint32 *arg2;          /* example for INTEGER */
```

- BIGINT

正しい例。C で SQLUDF_BIGINT または sqlint64 として表します。

以下に例を示します。

```
sqlint64 *arg3;          /* example for INTEGER */
```

DB2 では、sqlint64 C 言語タイプが定義されるので、コンパイラーとオペレーティング・システムの 64 ビットの符号付き整数の定義の違いはなくなります。#include sqludf.h または #include sqlsystem.h を指定して、定義を選出する必要があります。

- REAL または FLOAT(*n*)。ただし $1 \leq n \leq 24$ 。

正しい例。C で SQLUDF_REAL または float として表します。

以下に例を示します。

```
float *result;          /* example for REAL */
```

- DOUBLE または DOUBLE PRECISION または FLOAT または FLOAT(*n*)。ただし $25 \leq n \leq 53$ 。

正しい例。C で SQLUDF_DOUBLE または double として表します。

以下に例を示します。

```
double *result;         /* example for DOUBLE */
```

- DECIMAL(*p,s*) または NUMERIC(*p,s*)

誤った例。これは C 言語の表記ではありません。10 進数の値を渡す場合は、パラメーターを DECIMAL からキャスト可能なデータ・タイプ (CHAR や DOUBLE など) に定義して、引数をこのタイプに明示的にキャストしなければなりません。DOUBLE の場合は、DB2 が自動的にプロモーションするので、10 進値引数を明示的に DOUBLE パラメーターにキャストする必要はありません。

以下に例を示します。

DECIMAL(5,2) の WAGE と、DECIMAL(4,1) の HOURS という 2 つの列があり、賃金、労働時間、および他の要素に基づいて週給を計算する UDF を作成するとします。UDF は次のようになります。

```
CREATE FUNCTION WEEKLY_PAY (DOUBLE, DOUBLE, ...)
  RETURNS DECIMAL(7,2) CAST FROM DOUBLE
  ...;
```

上記の UDF では、最初の 2 つのパラメーターは賃金と時間に当たります。次のように SQL SELECT ステートメントで UDF WEEKLY_PAY を呼び出します。

```
SELECT WEEKLY_PAY (WAGE, HOURS, ...) ...;
```

DECIMAL 引数は DOUBLE にキャスト可能なので、明示的にキャストする必要はありません。

別の方法として、CHAR 引数を持つ WEEKLY_PAY を次のように定義できます。

```
CREATE FUNCTION WEEKLY_PAY (VARCHAR(6), VARCHAR(5), ...)
    RETURNS DECIMAL (7,2) CAST FROM VARCHAR(10)
    ...;
```

これは、次のように呼び出します。

```
SELECT WEEKLY_PAY (CHAR(WAGE), CHAR(HOURS), ...) ...;
```

DECIMAL 引数は VARCHAR にプロモーションできないので、明示的にキャストすることが必要であることに注意してください。

浮動小数点パラメーターを使用する利点は、ルーチン内の値の算術計算を実行しやすいことです。一方、文字パラメーターを使用する利点は、正確に 10 進数の値を表すことが常に可能であるということです。浮動小数点の場合、これは常に可能というわけではありません。

- FOR BIT DATA 修飾子を持つ、または持たない CHAR(n) または CHARACTER(n)

正しい例。C では SQLUDF_CHAR または char...[n+1] と表します (これは、C の NULL 終了ストリングです)。

以下に例を示します。

```
char    arg1[14];    /* example for CHAR(13) */
char    *arg1;      /* also acceptable */
```

データ・タイプ CHAR の入力ルーチン・パラメーターは常に自動的に NULL 終了します。CHAR(n) 入力パラメーターの場合 (n は CHAR データ・タイプの長さ)、n バイトのデータがルーチンのインプリメンテーションのバッファーに移され、n + 1 の位置にある文字が ASCII NULL 終止符の文字に設定されます (X'00')。

プロシージャの出力パラメーターおよびデータ・タイプ CHAR の関数の戻り値は、ルーチンによって明示的に NULL 終了されなければなりません。RETURNS CHAR(n) や、CHAR(n) として指定されるプロシージャの出力パラメーターなどの (n は CHAR 値の長さ)、RETURNS 節によって指定される UDF の戻り値の場合、バッファーの最初の n+1 バイトの中に NULL 終止符が存在していなければなりません。バッファーの最初の n+1 バイトの中に NULL 終止符が見つかった場合、n バイトまでの残りのバイトが ASCII ブランク文字に設定されます (X'20')。NULL 終止符が見つからなかった場合、結果は SQL エラー (SQLSTATE 39501) になります。

データ・タイプ CHAR のプロシージャまたは関数の戻り値の入出力パラメーターに FOR BIT DATA 節も指定されると、データがバイナリー形式で扱われることになり、パラメーター値の終了を示す際に NULL 終止符は使用されません。RETURNS CHAR(n) FOR BIT DATA 関数の戻り値または CHAR(n) FOR BIT DATA 出力パラメーターのいずれかの場合には、最初の n バイトの中にストリングの NULL 終止符が現れるかどうかに関係なく、バッファーの最初の n バイ

トが上書きコピーされます。バッファ内で識別される NULL 終止符の文字は NULL 終止符としては無視され、単純に通常のデータとして扱われます。

FOR BIT DATA 値を操作するルーチン内で通常のストリングを処理する C 関数を使用するときには、注意が必要です。NULL 終止符 (X'00') を FOR BIT DATA 値の中で使用するのとは正常なことです、この種の関数の多くは、ストリング引数を区切る NULL 終止符を探します。FOR BIT DATA 値に対して C 関数を使用すると、予期せずにデータ値が切り捨てられてしまうことがあります。

文字のルーチン・パラメーターを定義する際は、CHAR ではなく VARCHAR を使用するようしてください。これは、DB2 は VARCHAR 引数を CHAR にプロモートしないためと、ストリング・リテラルは自動的に VARCHAR とみなされるためです。例えば、UDF を次のように定義するとします。

```
CREATE FUNCTION SIMPLE(INT,CHAR(1))...
```

VARCHAR データ (... SIMPLE(1,'A')...) を使用して SIMPLE 関数を呼び出すと、関数が見つからないことを示す SQLCODE -440 (SQLSTATE 42884) エラーが出されますが、この関数のエンド・ユーザーはそのメッセージの原因を理解できないことがあります。上記の例では、'A' は VARCHAR であるため、それを CHAR にキャストすることも VARCHAR としてパラメーターを定義することもできます。

- FOR BIT DATA 修飾子を持つ、または持たない VARCHAR(n) FOR BIT DATA または LONG VARCHAR

正しい例。 VARCHAR(n) FOR BIT DATA を C では SQLUDF_VARCHAR_FBD と表します。 LONG VARCHAR を C では SQLUDF_LONG と表します。そうでない場合、この 2 つの SQL タイプを C では、 sqludf.h 組み込みファイルの中の以下のものに似た構造で表します。

```
struct sqludf_vc_fbd
{
    unsigned short length;      /* length of data */
    char          data[1];     /* first char of data */
};
```

[1] は、コンパイラーに対する配列を示しています。1 文字だけが渡されることを意味しているわけではありません。すなわち構造のアドレスが渡されますが、それは実際の構造ではないため、配列論理を使用する方法を提供するだけです。

これらの値は、C の NULL 終了ストリングとしては表されません。これは、NULL 文字がデータ値の一部として正しく認識されることがあるためです。その長さは、構造変数 length を使用してルーチンにパラメーターとして正しく渡されます。 RETURNS 節の場合、ルーチンに渡される長さはバッファの長さです。ルーチン本体は、構造変数 length を使用してデータ値の実際の長さを戻す必要があります。

以下に例を示します。

```
struct sqludf_vc_fbd *arg1; /* example for VARCHAR(n) FOR BIT DATA */
struct sqludf_vc_fbd *result; /* also for LONG VARCHAR FOR BIT DATA */
```

- FOR BIT DATA を持たない VARCHAR(n)

正しい例。C では SQLUDF_VARCHAR または char...[n+1] と表されます。(これは C の NULL 終了ストリングです。)

VARCHAR(n) パラメーターの場合、DB2 は NULL を (k+1) の位置に置きます。この場合の k は特定のストリングの長さです。このため、C ストリング処理関数はこれらの値の操作に適しています。RETURNS VARCHAR(n) 値またはストアード・プロシージャの出力パラメーターの場合、ルーチン本体は実際の値を NULL を使用して区切る必要があります。これは、DB2 がこの NULL 文字から結果の長さを決めるためです。

以下に例を示します。

```
char    arg2[51];    /* example for VARCHAR(50) */
char    *result;    /* also acceptable */
```

- DATE

正しい例。C で SQLUDF_DATE または CHAR(10) として、つまり char...[11] として表します。日付の値は、常に以下の ISO 書式でルーチンに渡されます。

yyyy-mm-dd

以下に例を示します。

```
char    arg1[11];    /* example for DATE      */
char    *result;    /* also acceptable */
```

注: DATE、TIME、および TIMESTAMP の戻り値の場合、DB2 では文字が定義済みの形式になっていなければなりません。そうでないと、その値は DB2 で誤解される可能性があったり (例えば、3 月 4 日であるはずの 2001-04-03 は 4 月 3 日と解釈されます)、エラーを生じたりします (SQLCODE -493、SQLSTATE 22007)。

- TIME

正しい例。C で SQLUDF_TIME または CHAR(8) として、つまり char...[9] として表します。時間の値は、常に以下の ISO 書式でルーチンに渡されます。

hh.mm.ss

以下に例を示します。

```
char    *arg;        /* example for TIME      */
char    result[9];   /* also acceptable */
```

- TIMESTAMP

正しい例。C で SQLUDF_STAMP または CHAR(19) から CHAR(32) として、つまり char[20] to char[33] として表します。タイム・スタンプの値の形式は以下のとおりです。

yyyy-mm-dd-hh.mm.ss.nnnnnnnnnnn

詳細は次のとおりです。

yyyy 年を表します。

mm 月を表します。

dd 日を表します。

hh 時を表します。

mm 分を表します。

ss 秒を表します。

nnnnnnnnnnnnnn

秒の小数部分を表します。TIMESTAMP データ・タイプの秒の小数部分には、オプションで、0 から 12 桁のタイム・スタンプ精度を指定できます。

タイム・スタンプ値が、秒の小数部分の桁数が異なるタイム・スタンプ変数に割り当てられると、その値はタイム・スタンプ変数のフォーマットに合うように切り捨てられるか、0 を埋め込むことによって調整されます。

文字ストリングは、秒の小数部分の指定された桁数に応じて、19 から 32 バイトの長さにすることができます。TIMESTAMP データ・タイプの秒の小数部分には、オプションで、0 から 12 桁のタイム・スタンプ精度を指定できます。

以下に例を示します。

```
(VALUES(CURRENT TIMESTAMP(0))
```

```
1
-----
2008-07-09-14.48.36
```

1 record(s) selected.

```
LENGTH (VALUES(CURRENT TIMESTAMP(0))
```

```
1
-----
19
```

1 record(s) selected.

```
(VALUES(CURRENT TIMESTAMP(12))
```

```
1
-----
2008-07-09-14.48.36.123456789012
```

1 record(s) selected.

```
LENGTH (VALUES(CURRENT TIMESTAMP(0))
```

```
1
-----
32
```

以下は、TIMESTAMP(12) 値を保持できる変数宣言です。

```
char    arg1[33];      /* example for TIMESTAMP */
char    *result;      /* also acceptable */
```

• GRAPHIC(n)

正しい例。C では SQLUDF_GRAPH または sqldbchar[n+1] と表されます。(これは NULL 終了 GRAPHIC ストリングです。) wchar_t が長さ 2 バイトとして定義されているオペレーティング・システム上では、wchar_t[n+1] を使用できますが、sqldbchar を使用することをお勧めします。

GRAPHIC(n) パラメーターの場合、DB2 は n 個の 2 バイト文字をバッファーに移動し、次の 2 バイトを NULL に設定します。DB2 からルーチンに渡されるデータは DBCS 書式であり、戻される結果も DBCS 書式であると見なされます。この動作は、WCHARTYPE NOCONVERT プリコンパイラー・オプションを使うことと同じです。RETURNS GRAPHIC(n) 値またはストアード・プロシージャ

ャーの出力パラメーターの場合、DB2 は組み込みの GRAPHIC NULL CHAR を探します。見つかった場合、値の *n* まで GRAPHIC ブランク文字を埋め込みます。

GRAPHIC ルーチン・パラメーターを定義する際は、GRAPHIC よりも VARGRAPHIC を使用するようしてください。これは、DB2 が VARGRAPHIC 引数を GRAPHIC にプロモートしないためです。例えば、ルーチンを次のように定義するとします。

```
CREATE FUNCTION SIMPLE(GRAPHIC)...
```

VARGRAPHIC データ (... SIMPLE('graphic_literal')...) を使用して SIMPLE 関数を呼び出すと、関数が見つからないことを示す SQLCODE -440 (SQLSTATE 42884) エラーが出されますが、この関数のエンド・ユーザーはそのメッセージの原因を理解できないことがあります。上記の例では、*graphic_literal* は VARGRAPHIC データとして解釈されるリテラル DBCS ストリングであるため、それを GRAPHIC にキャストすることも VARGRAPHIC としてパラメーターを定義することもできます。

以下に例を示します。

```
sqldbchar  arg1[14];      /* example for GRAPHIC(13) */
sqldbchar  *arg1;        /* also acceptable */
```

- VARGRAPHIC(*n*)

正しい例。 C では SQLUDF_GRAPH または sqldbchar[*n*+1] と表されます。(これは NULL 終了 GRAPHIC ストリングです。) wchar_t が長さ 2 バイトとして定義されているオペレーティング・システム上では、wchar_t[*n*+1] を使用できますが、sqldbchar を使用することをお勧めします。

VARGRAPHIC(*n*) パラメーターの場合、DB2 は GRAPHIC NULL を (*k*+1) の位置に置きます。この場合の *k* は個々に発生する長さです。GRAPHIC NULL は、GRAPHIC ストリングの最後の文字の全バイトにバイナリー・ゼロ ('¥0's) が含まれていることを示します。DB2 からルーチンに渡されるデータは DBCS 書式であり、戻される結果も DBCS 書式であると見なされます。この動作は、WCHARTYPE NOCONVERT プリコンパイラー・オプションを使うことと同じです。RETURNS VARGRAPHIC(*n*) 値またはストアード・プロシージャの出力パラメーターの場合、ルーチン本体の実際の値を GRAPHIC NULL を使用して区切る必要があります。これは、DB2 はこの GRAPHIC NULL 文字から結果の長さを決めるためです。

以下に例を示します。

```
sqldbchar  args[51],     /* example for VARGRAPHIC(50) */
sqldbchar  *result,     /* also acceptable */
```

- LONG VARGRAPHIC

正しい例。 C で SQLUDF_LONGVARG または次のような構造として表します。

```
struct sqludf_vg
{
    unsigned short length;      /* length of data */
    sqldbchar      data[1];     /* first char of data */
};
```

上記の構造体において、`wchar_t` が 2 バイトの長さで定義されているオペレーティング・システムでは `sqldbchar` に代えて `wchar_t` を使用できますが、`sqldbchar` の方を使用するようお勧めします。

[1] は、単にコンパイラーに対する配列を示しています。GRAPHIC 文字を 1 つだけ渡すことを意味しているのではありません。すなわち、渡されるのは構造のアドレスであり、実際の構造ではないため、配列論理を使用する方法が提供されます。

これらは NULL 終了 GRAPHIC ストリングとしては表されません。その長さ (2 バイト文字単位) は、構造変数 `length` を使用してルーチンにパラメーターとして明示的に渡されます。DB2 からルーチンに渡されるデータは DBCS 書式であり、戻される結果も DBCS 書式であると見なされます。この動作は、`WCHARTYPE NOCONVERT` プリコンパイラー・オプションを使うことと同じです。RETURNS 節またはストアード・プロシージャの出力パラメーターの場合、ルーチンに渡される長さはバッファの長さです。ルーチン本体は、構造変数 `length` を使用してデータ値の実際の長さを 2 バイト文字で戻す必要があります。

以下に例を示します。

```
struct sqludf_vg *arg1; /* example for VARGRAPHIC(n) */
struct sqludf_vg *result; /* also for LONG VARGRAPHIC */
```

- BLOB(n) と CLOB(n)

正しい例。C で `SQLUDF_BLOB`、`SQLUDF_CLOB`、または次のような構造として表します。

```
struct sqludf_lob
{
    sqluint32    length;    /* length in bytes */
    char         data[1];  /* first byte of lob */
};
```

[1] は、単にコンパイラーに対する配列を示しています。1 文字だけが渡されることを意味しているのではありません。すなわち構造のアドレスが渡されますが、それは実際の構造ではないため、配列論理を使用する方法を提供するだけです。

これらは C の NULL 終了ストリングとして表されません。その長さは、構造変数 `length` を使用してルーチンにパラメーターとして正しく渡されます。RETURNS 節またはストアード・プロシージャの出力パラメーターの場合、ルーチンに返送される長さは、バッファの長さです。ルーチン本体は、構造変数 `length` を使用してデータ値の実際の長さを戻す必要があります。

以下に例を示します。

```
struct sqludf_lob *arg1; /* example for BLOB(n), CLOB(n) */
struct sqludf_lob *result;
```

- DBCLOB(n)

正しい例。C で `SQLUDF_DBCLOB` または次のような構造として表します。

```

struct sqludf_lob
{
    sqluint32 length;    /* length in graphic characters */
    sqldbchar data[1]; /* first byte of lob */
};

```

上記の構造体において、wchar_t が 2 バイトの長さで定義されているオペレーティング・システムでは sqldbchar に代えて wchar_t を使用できますが、sqldbchar の方を使用するようお勧めします。

[1] は、単にコンパイラーに対する配列を示しています。GRAPHIC 文字を 1 つだけ渡すことを意味しているのではありません。すなわち、構造のアドレスが渡されますが、それは実際の構造ではないため、配列論理を使用する方法を提供するだけです。

これらは NULL 終了 GRAPHIC ストリングとしては表されません。その長さは、構造変数 length を使用してルーチンにパラメーターとして正しく渡されます。DB2 からルーチンに渡されるデータは DBCS 書式であり、戻される結果も DBCS 書式であると見なされます。この動作は、WCHARTYPE NOCONVERT プリコンパイラー・オプションを使うことと同じです。RETURNS 節またはストアド・プロシージャの出力パラメーターの場合、ルーチンに渡される長さはバッファの長さです。ルーチン本体は、構造変数 length を使用してデータ値の実際の長さを戻す必要がありますが、その際にこれらすべての長さを 2 バイト文字で表していなければなりません。

以下に例を示します。

```

struct sqludf_lob *arg1; /* example for DBCLOB(n) */
struct sqludf_lob *result;

```

- 特殊タイプ

正しい例または誤った例 (基本タイプにより異なる)。特殊タイプは、UDT の基本タイプの書式で UDF に渡されるため、基本タイプが有効な場合に限り指定されます。

以下に例を示します。

```

struct sqludf_lob *arg1; /* for distinct type based on BLOB(n) */
double *arg2; /* for distinct type based on DOUBLE */
char res[5]; /* for distinct type based on CHAR(4) */

```

- XML

正しい例。C で SQLUDF_XML として表すか、または以下の構造を使用して、CLOB データ・タイプを表す方法で表します。

```

struct sqludf_lob
{
    sqluint32 length;    /* length in bytes */
    char data[1]; /* first byte of lob */
};

```

[1] は、単にコンパイラーに対する配列を示しています。1 文字だけが渡されることを意味しているのではありません。すなわち構造のアドレスが渡されますが、それは実際の構造ではないため、配列論理を使用する方法を提供するだけです。

これらは C の NULL 終了ストリングとして表されません。その長さは、構造変数 length を使用してルーチンにパラメーターとして正しく渡されます。RETURNS 節またはストアード・プロシージャの出力パラメーターの場合、ルーチンに返送される長さは、バッファの長さです。ルーチン本体は、構造変数 length を使用してデータ値の実際の長さを戻す必要があります。

以下に例を示します。

```
struct sqludf_lob *arg1; /* example for XML(n) */
struct sqludf_lob *result;
```

C および C++ 外部ルーチン・コード中の XML パラメーターおよび変数値の割り当てとアクセスは、CLOB 値の場合と同じ方法で行われます。

- 特殊タイプ AS LOCATOR、または任意の LOB タイプ AS LOCATOR

UDF およびメソッドのパラメーターと結果には有効です。 これは、LOB タイプか、LOB タイプに基づく特殊タイプを修正する場合にのみ使用できます。C では SQLUDF_LOCATOR または 4 バイトの整数で表されます。

互換性のあるタイプをもった任意のロケーター・ホスト変数にロケーター値を割り当ててから、SQL ステートメント内でそれを使用することができます。つまり、ロケーター変数が有用であるのは、CONTAINS SQL 以上の SQL アクセス標識を使用して定義されている UDF およびメソッドにおいてのみであるということです。既存の UDF およびメソッドとの互換性に関しては、NOT FENCED NO SQL UDF ではロケーター API はこれまでどおりサポートされます。新規の関数の場合はこの API の使用はお勧めしません。

以下に例を示します。

```
sqludf_locator      *arg1; /* locator argument */
sqludf_locator      *result; /* locator result */
```

```
EXEC SQL BEGIN DECLARE SECTION;
SQL TYPE IS CLOB LOCATOR arg_loc;
SQL TYPE IS CLOB LOCATOR res_loc;
EXEC SQL END DECLARE SECTION;
```

```
/* Extract some characters from the middle */
/* of the argument and return them          */
*arg_loc = arg1;
EXEC SQL VALUES SUBSTR(arg_loc, 10, 20) INTO :res_loc;
*result = res_loc;
```

- 構造化タイプ

適切な Transform 関数が存在する場合の UDF およびメソッドのパラメーターと結果に有効です。構造化タイプ・パラメーターは、FROM SQL Transform 関数の結果タイプの形で関数またはメソッドに渡されます。構造化タイプの結果は、TO SQL Transform 関数のパラメーター・タイプで渡されます。

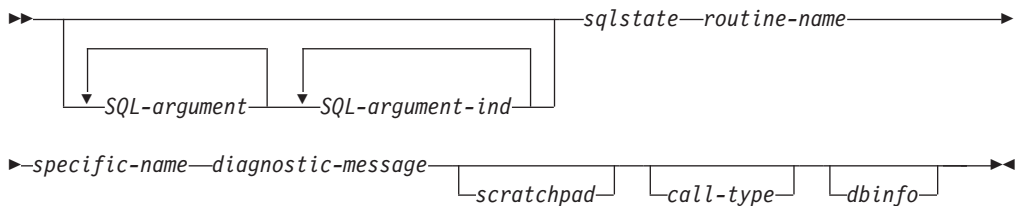
C、C++、OLE、または COBOL ルーチンへの引数の引き渡し

ルーチンの DML 参照で指定された SQL 引数に加えて、DB2 は追加の引数を外部ルーチン本体に渡します。そのような引数の特性と順序は、ルーチンの登録時に指定したパラメーター・スタイルで決まります。呼び出し側とルーチン本体が必ず正

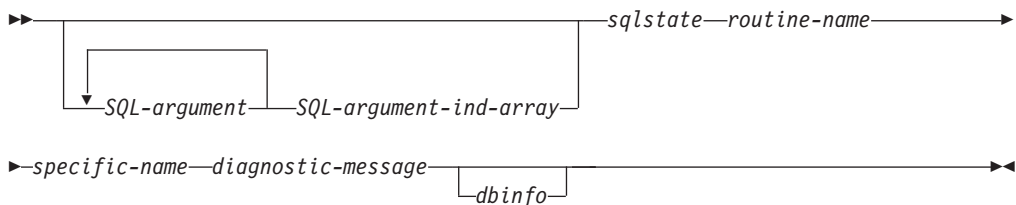
しく情報を交換できるようにするには、ルーチンが、使用しているパラメーター・スタイルに従って、渡されたとおりの順序で引数を受け入れることを確認しなければなりません。sqludf 組み込みファイルを使用すれば、そのような引数を簡単に処理して使用することができます。

以下に示すパラメーター・スタイルは LANGUAGE C、LANGUAGE OLE、および LANGUAGE COBOL ルーチンにのみ適用されます。

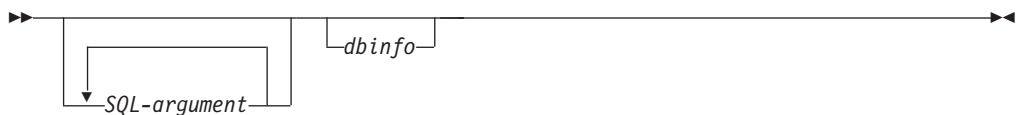
PARAMETER STYLE SQL ルーチン



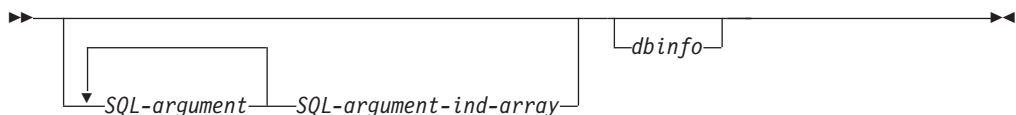
PARAMETER STYLE DB2SQL プロシージャ



PARAMETER STYLE GENERAL プロシージャ



PARAMETER STYLE GENERAL WITH NULLS プロシージャ



注: UDF およびメソッドの場合、PARAMETER STYLE SQL は PARAMETER STYLE DB2SQL と同等です。

上記のパラメーター・スタイルの引数について、以下に説明してあります。

SQL-argument...

各 SQL-argument は、ルーチンの作成時に定義された 1 つの入力値または出力値を表します。引数リストは次のように判別されます。

- スカラー関数の場合、関数への各入力パラメーターごとに 1 つの引数の後に、関数の結果の 1 つの SQL-argument が続きます。

- 表関数の場合、関数への各入力パラメーターごとに 1 つの引数の後に、関数の結果表内の各列ごとに 1 つの *SQL-argument* が続きます。
- メソッドの場合、メソッドのサブジェクト・タイプごとに 1 つの *SQL-argument* と、さらにメソッドへの各入力パラメーターごとに 1 つの引数の後に、メソッドの結果ごとに 1 つの *SQL-argument* が続きます。
- ストアード・プロシージャの場合、ストアード・プロシージャへの各パラメーターごとに 1 つの *SQL-argument*。

各 *SQL-argument* は、次のように使用します。

- 関数またはメソッドの入力パラメーター、メソッドのサブジェクト・タイプ、またはストアード・プロシージャの IN パラメーター。

この引数は、ルーチンを呼び出す前に DB2 によって設定されます。これらの各引数の値は、ルーチン呼び出しで指定された式から取得されます。その値は、CREATE ステートメント中で該当するパラメーター定義のデータ・タイプで表されます。

- 関数またはメソッドの結果、またはストアード・プロシージャの OUT パラメーター。

この引数は、DB2 に戻る前にルーチンによって設定されます。DB2 はバッファを割り振り、そのアドレスをルーチンに渡します。ルーチンは結果の値をバッファに入れます。DB2 は、データ・タイプで示される値を収容するのに十分なバッファ・スペースを割り振ります。文字タイプおよび LOB の場合はこれは、CREATE ステートメントでの定義どおりの最大サイズが割り振られることを意味します。

スカラー関数およびメソッドの場合、結果データ・タイプは、CAST FROM 節があればそこで定義され、CAST FROM 節がなければ、RETURNS 節で定義されます。

表関数の場合、DB2 は、定義されているすべての列を DB2 に戻さなくてもよいように、パフォーマンスの最適化を定義します。このフィーチャーを利用するように UDF を作成する場合、UDF は、表関数を参照しているステートメントが必要とする列のみを戻します。例えば、100 個の結果列が定義されている表関数の CREATE FUNCTION ステートメントを考えてみましょう。この関数を参照するステートメントに関係するものが、これらの結果列のうちの 2 つだけであるならば、この最適化により、UDF は各行にこれらの 2 つの列だけを戻し、他の 98 列には時間を費やしません。この最適化の詳細については、後で説明する dbinfo 引数を参照してください。

戻される各値については、ルーチンが、結果のデータ・タイプと長さに必要なバイト数よりも多くのバイトを戻さないようにしてください。最大値は、ルーチンのカタログ項目の作成時に定義します。ルーチンによって上書きされると、想定外の結果が生じたり、異常終了が起きることがあります。

- ストアード・プロシージャの INOUT パラメーター。

この引数は、IN と OUT の両方のパラメーターとして働くので、上記の両方の一連の規則に従います。DB2 は、ストアド・プロシージャを呼び出す前に引数を設定します。引数用に DB2 で割り振られるバッファは、CREATE PROCEDURE ステートメントに定義されているパラメーターのデータ・タイプの最大サイズを収容するのに十分な大きさです。例えば、CHAR タイプの INOUT パラメーターは、ストアド・プロシージャに送られる 10 バイトの VARCHAR をもっていたり、ストアド・プロシージャから送出される 100 バイトの VARCHAR をもっていたりすることがあります。バッファは、DB2 に戻る前にストアド・プロシージャによって設定されます。

DB2 は、データ・タイプとサーバーのオペレーティング・システム (プラットフォームともいう) に応じて、*SQL-argument* が表すデータの位置を調整します。

SQL-argument-ind...

ルーチンに渡される各 *SQL-argument* ごとに *SQL-argument-ind* があります。n 番目の *SQL-argument-ind* は、n 番目の *SQL-argument* に対応し、*SQL-argument* が値を持っているかまたは NULL であることを示します。

各 *SQL-argument-ind* は、次のように使用します。

- 関数またはメソッドの入力パラメーター、メソッドのサブジェクト・タイプ、またはストアド・プロシージャの IN パラメーター。

この引数は、ルーチンを呼び出す前に DB2 によって設定されます。この引数には以下の値のうちの 1 つが入ります。

- 0** NULL 以外の引数があります。
- 1** NULL の引数があります。

ルーチンを RETURNS NULL ON NULL INPUT で定義すると、ルーチン本体は NULL 値に対する検査を行う必要がありません。ただし、CALLED ON NULL INPUT で定義されると引数はいずれも NULL になる可能性があるため、ルーチンは *SQL-argument-ind* を検査してから、それに対応する *SQL-argument* を使用する必要があります。

- 関数またはメソッドの結果、またはストアド・プロシージャの OUT パラメーター。

この引数は、DB2 に戻る前にルーチンによって設定されます。ルーチンは、この引数を使用して特定の結果値が NULL かどうかを示します。

- 0** 結果は NULL ではありません。
- 1** 結果は NULL 値です。

ルーチンが RETURNS NULL ON NULL INPUT で定義されていても、ルーチン本体は結果の *SQL-argument-ind* を設定する必要があります。例えば、分母がゼロである場合、除算関数は結果を NULL に設定することができます。

スカラー関数およびメソッドの場合、以下が真の場合は DB2 は NULL 結果を算術計算エラーとして扱います。

- データベース構成パラメーター *dft_sqlmathwarn* が YES の場合
- 入力引数の 1 つが、算術計算エラーのため NULL になっている場合

これは、関数を RETURNS NULL ON NULL INPUT オプションで定義した場合にも当てはまります。

表関数の場合、列リストを用いて最適化の利点を UDF で活用する場合、必要な列に対応する標識のみを設定する必要があります。

- ストアド・プロシージャの INOUT パラメーター。

この引数は、IN と OUT の両方のパラメーターとして働くので、上記の両方の一連の規則に従います。DB2 は、ストアド・プロシージャを呼び出す前に引数を設定します。SQL-argument-ind は、DB2 に戻る前にストアド・プロシージャによって設定されます。

各 SQL-argument-ind は、SMALLINT 値の形式をとります。DB2 は、データ・タイプとサーバーのオペレーティング・システムに応じて、SQL-argument-ind が表すデータの位置を調整します。

SQL-argument-ind-array

ストアド・プロシージャに渡される各 SQL 引数ごとに、SQL-argument-ind-array 内にエレメントが 1 つずつあります。SQL-argument-ind-array 内の n 番目のエレメントは、 n 番目の SQL-argument に対応し、SQL-argument が値をもっているかまたは NULL であるかを示します。

SQL-argument-ind-array 内の各エレメントは、次のように使用します。

- ストアド・プロシージャの IN パラメーター。

このエレメントは、ルーチンを呼び出す前に DB2 によって設定されます。この引数には以下の値のうちの 1 つが入ります。

- 0 NULL 以外の引数があります。
- 1 NULL の引数があります。

ストアド・プロシージャを RETURNS NULL ON NULL INPUT で定義すると、ストアド・プロシージャ本体は NULL 値に対する検査を行う必要がありません。ただし、CALLED ON NULL INPUT で定義されると引数はいずれも NULL になる可能性があるため、ストアド・プロシージャは SQL-argument-ind を検査してから、それに対応する SQL-argument を使用する必要があります。

- ストアド・プロシージャの OUT パラメーター。

このエレメントは、DB2 に戻る前にルーチンによって設定されます。ルーチンは、この引数を使用して特定の結果値が NULL かどうかを示します。

0 または正の整数

結果は NULL ではありません。

負の整数

結果は NULL 値です。

- ストアド・プロシージャの INOUT パラメーター。

このエレメントは、IN と OUT の両方のパラメーターとして働くので、上記の両方の一連の規則に従います。DB2 は、ストアド・プロシージャ

ャーを呼び出す前に引数を設定します。 *SQL-argument-ind-array* のエレメントは、 DB2 に戻る前にストアード・プロシージャによって設定されます。

SQL-argument-ind-array の各エレメントは、 SMALLINT 値の形式をとります。 DB2 は、データ・タイプとサーバーのオペレーティング・システムに応じて、 *SQL-argument-ind-array* が表すデータの位置を調整します。

sqlstate

この引数は、DB2 に戻る前にルーチンによって設定されます。これをルーチンで使用して、警告またはエラー条件を発信することができます。ルーチンは、この引数を任意の値に設定することができます。 '00000' の値は、警告またはエラーの状態はまったく検出されなかったことを意味します。 '01' で始まる値は警告条件です。 '00' または '01' 以外のもので始まる値はすべて、エラー条件です。ルーチンを呼び出したときにはこの引数には値 '00000' が入っています。

エラー条件の場合、ルーチンは -443 の SQLCODE を戻します。警告条件の場合、ルーチンは +462 の SQLCODE を戻します。 SQLSTATE が 38001 または 38502 の場合、SQLCODE は -487 になります。

sqlstate は、CHAR(5) 値の形式をとります。 DB2 は、データ・タイプとサーバーのオペレーティング・システムに応じて、 *sqlstate* が表すデータの位置を調整します。

routine-name

この引数は、ルーチンを呼び出す前に DB2 によって設定されます。これは、DB2 からルーチンに渡される、修飾された関数名です。

渡される *routine-name* の形式は次のとおりです。

schema.routine

各部分はピリオドで区切られます。以下に例を 2 つ示します。

PABLO.BLOOP WILLIE.FINDSTRING

この形式を使うと、複数の外部ルーチンに同じルーチン本体を使用してもその呼び出し時にはそれらのルーチンを区別することができます。

注: オブジェクト名およびスキーマ名にはピリオドを付けることができますが、付けない方がよいでしょう。例えば、関数 ROTATE がスキーマ OBJ.OP 内にあって、関数に渡されるルーチン名が OBJ.OP.ROTATE である場合に、スキーマ名が OBJ または OBJ.OP のどちらなのかあいまいであるとします。

routine-name は VARCHAR(257) 値の形式をとります。 DB2 は、データ・タイプとサーバーのオペレーティング・システムに応じて、 *routine-name* が表すデータの位置を調整します。

specific-name

この引数は、ルーチンを呼び出す前に DB2 によって設定されます。これは、DB2 からルーチンに渡されるルーチンの特定の名前です。

以下に例を 2 つ示します。

WILLIE_FIND_FEB99 SQL9904281052440430

この例の最初の値は、ユーザーが CREATE ステートメントで定義します。2 番目の値は、ユーザーが値を指定しなかった場合に DB2 によって現行タイム・スタンプから生成される値です。

routine-name 引数の場合と同じように、この値が渡されるのは、どのルーチンが呼び出しているかをはっきり区別するための手段をルーチンに対して提供するためです。

specific-name は VARCHAR(18) 値の形式をとります。DB2 は、データ・タイプとサーバーのオペレーティング・システムに応じて、*specific-name* が表すデータの位置を調整します。

diagnostic-message

この引数は、DB2 に戻る前にルーチンによって設定されます。ルーチンは、この引数を用いて DB2 メッセージにメッセージ・テキストを挿入します。

上述の *sqlstate* 引数を用いてルーチンがエラーまたは警告のいずれかを戻す場合、ここに記述情報を組み込むことができます。DB2 はこの情報をトークンとしてメッセージ内に組み込みます。

DB2 は、ルーチンを呼び出す前に最初の文字を NULL に設定します。そして、戻ったときにそのストリングを C の NULL 終了ストリングとして扱います。このストリングは、エラー状態のトークンとして SQLCA 内に組み込まれます。このストリングの少なくとも最初の一部は、SQLCA または DB2 CLP メッセージに表示されます。ただし、表示される実際の文字数は、その他のトークンの長さで決まります。これは DB2 が、SQLCA で定められている合計トークン長に合わせてトークンを切り捨てるからです。SQLCA 内のトークンを区切るために使用されるため、テキストでは X'FF' という文字を使用しないでください。

ルーチンは、そのコードに渡される VARCHAR(70) バッファーに入らないほど多くのテキストを戻すべきではありません。ルーチンによって上書きされると、想定外の結果が生じたり、異常終了が起きることがあります。

DB2 では、ルーチンから DB2 に戻されるメッセージ・トークンがルーチンと同じコード・ページにあることを前提とします。ご使用のルーチンがそれに当てはまるかどうかを確認してください。7 ビットの不変の ASCII サブセットを使うと、ルーチンは任意のコード・ページのメッセージ・トークンを戻します。

diagnostic-message は VARCHAR(70) 値の形式をとります。DB2 は、データ・タイプとサーバーのオペレーティング・システムに応じて、*diagnostic-message* が表すデータの位置を調整します。

scratchpad

この引数は、UDF またはメソッドの呼び出しの前に DB2 によって設定されます。これは、登録時に SCRATCHPAD キーワードを指定した関数とメソッドの場合にのみ示されます。この引数は、以下のエレメントを持ち、任意の LOB データ・タイプの値を渡すために使用される構造とまったく同じ構造です。

- スクラッチパッドの長さを含む INTEGER。スクラッチパッドの長さを変更すると、SQLCODE -450 (SQLSTATE 39501) になります。

- 実際のスクラッチパッド。以下のようにすべてバイナリー数の 0 に初期化されます。
 - スカラー関数およびメソッドの場合、スクラッチパッドは最初の呼び出し前に初期化され、その後は通常は DB2 による参照や修正は行われません。
 - 表関数の場合、FINAL CALL が CREATE FUNCTION で指定されているなら、スクラッチパッドは UDF への FIRST 呼び出しより前に初期化されます。この呼び出しの後、スクラッチパッドの内容は、完全に表関数の制御下に置かれます。表関数で NO FINAL CALL が明示的に指定されたか、デフォルトとして使用された場合、スクラッチパッドは各 OPEN 呼び出しごとに初期化され、スクラッチパッドの内容は回目の OPEN 呼び出しまで完全に表関数の制御下に置かれます。(これは、結合または副照会で使用される表関数ではかなり重要である場合があります。複数の OPEN 呼び出しにまたがってスクラッチパッドの内容を保守する必要がある場合、CREATE FUNCTION ステートメントで FINAL CALL を指定しなければなりません。通常の OPEN、FETCH、および CLOSE 呼び出しに加え、FINAL CALL を指定すると、表関数は、スクラッチパッド保守およびリソース解放のために、FIRST および FINAL 呼び出しも受け取ります。)

スクラッチパッドは、CLOB か BLOB と同じタイプを使用してルーチンにマップすることができます。これは、渡される引数が同じ構造であるためです。

ルーチン・コードがスクラッチパッド・バッファ外で変更を行わないことを確認してください。ルーチンによって上書きされると、想定外の結果が生じたり異常終了したりして、DB2 による軽度の障害では済まない可能性があります。

スクラッチパッドを使用するスカラー UDF またはメソッドが副照会で参照される場合、DB2 は副照会の呼び出しと呼び出しの間にスクラッチパッドをリフレッシュすることに決めることがあります。UDF で FINAL CALL が指定されている場合、このリフレッシュは、最終呼び出しが行われた後に起こります。

DB2 は、データ・フィールドの位置がどのデータ・タイプのストレージでも合うよう、スクラッチパッドを初期化します。その結果、スクラッチパッド構造全体 (長さフィールドを含む) が正しく位置合わせされない場合があります。

call-type

この引数 (存在する場合) は、UDF またはメソッドの呼び出しの前に DB2 によって設定されます。この引数が存在するのは、すべての表関数の場合と、登録時に FINAL CALL を指定したスカラー関数およびメソッドの場合です。

現在 *call-type* に指定できるすべての値が以下に示されています。UDF またはメソッドには、「A ならば AA を実行、さもなければ B ならば BB を実行、さもなければ必ず C なので CC を実行 (if A do AA, else if B do BB, else it must be C so do CC)」といったタイプの論理を組み込むのではなく、想定されるすべての値のテストを明示的に指示するスイッチまたは

CASE ステートメントを UDF に組み込む必要があります。これは、将来さらに別の呼び出しタイプが追加される可能性に対する措置です。条件 C のテストを明示的に指示していないと、新しい呼び出しタイプが追加されたときに問題が発生するからです。

注:

1. *call-type* のどの値の場合も、*sqlstate* と *diagnostic-message* 戻り値をルーチンで設定するのが適切です。この解説は、以下の各 *call-type* の説明の中では繰り返されていません。すべての呼び出しで、DB2 は、これらの引数について前に説明したように、指示されたアクションを行います。
2. 組み込みファイル *sqludf.h* は、ルーチンで使用するためのものです。このファイルには以下の *call-type* の値のシンボリック定義が入っていて、それらは定数として読み取られます。

スカラー関数およびメソッドの場合、*call-type* には次のものが入ります。

SQLUDF_FIRST_CALL (-1)

これは、このステートメントに対するルーチンへの **FIRST** 呼び出しです。*scratchpad* (存在する場合) は、ルーチンが呼び出されるときにバイナリー数のゼロに設定されます。すべての引数値が渡され、ルーチンは 1 回の初期化処理に必要なことを行います。加えて、スカラー UDF またはメソッドに対する **FIRST** 呼び出しは、応答を作成して戻すことになるとみなされるため、**NORMAL** 呼び出しに似ています。

注: **SCRATCHPAD** が指定されていても **FINAL CALL** が指定されていない場合、ルーチンは最初の呼び出しを識別するためにこの *call-type* 引数を使用しません。その代わりに、スクラッチパッドのすべてゼロの状態に依存する必要があります。

SQLUDF_NORMAL_CALL (0)

これは **NORMAL** 呼び出しです。すべての SQL 入力値が渡され、ルーチンが結果を作成して戻すことが期待されています。ルーチンは、*sqlstate* と *diagnostic-message* 情報も戻すことがあります。

SQLUDF_FINAL_CALL (1)

これは **FINAL** 呼び出しです。すなわち、*SQL-argument* の値も *SQL-argument-ind* の値も渡されず、これらの値を検査しようとする、予測不能な結果が生じます。*scratchpad* も渡される場合は、この値は前の呼び出し時のままです。ルーチンはこの時点でリソースを解放することになります。

SQLUDF_FINAL_CRA (255)

これは **FINAL** 呼び出しであり、上記の **FINAL** 呼び出しと同一ですが、さらに別の特性が 1 つ追加されています。つまりこれは、SQL を発行できると定義されているルーチンに対する追加であり、ルーチンは **CLOSE** カーソル以外のどの SQL も発行してはならない場合のための追加です。(SQLCODE -396、SQLSTATE 38505) 例えば、DB2 が **COMMIT** 処理の途中にあるときは、新規の SQL を受け入れることができないので、その時点でルーチンに対して

FINAL 呼び出しが発行された場合はすべて、255 FINAL 呼び出しになります。どのレベルの SQL アクセス権ももっていないと定義されたルーチンが 255 FINAL 呼び出しを受信することはないのに対して、SQL を使用するルーチンは、いずれかのタイプの FINAL 呼び出しを受け取ることがあります。

リソースの解放

スカラー UDF またはメソッドは、例えばメモリーのような、必要なリソースを解放するものとみなされます。SCRATCHPAD が同時に指定されていて、リソースを追跡するために使用されている場合に限っては、FINAL CALL がルーチンに指定されると、FINAL 呼び出しがリソースを解放する順当な時点になります。FINAL CALL が指定されていない場合には、獲得されたいずれかのリソースをその同じ呼び出し時に解放する必要があります。

表関数の場合、*call-type* には次のものが入ります。

SQLUDF_TF_FIRST (-2)

これは、UDF に対して FINAL CALL キーワードが指定された場合にだけ生じる、FIRST 呼び出しです。この呼び出しの前に、*scratchpad* はバイナリー・ゼロに設定されます。引数値は、表関数に渡されます。表関数はメモリーを取得するか、別のリソースの初期化を一度限り実行できます。これは OPEN 呼び出しではありません。OPEN 呼び出しは、この呼び出しの後に続きます。FIRST 呼び出し時には、DB2 がデータを無視するため、表関数は DB2 にデータを戻しません。

SQLUDF_TF_OPEN (-1)

これは、OPEN 呼び出しです。NO FINAL CALL が指定される場合には、*scratchpad* は初期化されますが、指定されない場合には、初期化する必要はありません。すべての SQL 引数値は、OPEN 時の表関数に渡されます。OPEN 呼び出し時には、表関数は DB2 にデータを戻しません。

SQLUDF_TF_FETCH (0)

これは FETCH 呼び出しで、通常 DB2 では、表関数が戻り値のセットから成る行か、SQLSTATE 値 '02000' によって指定された表の終わりの条件を戻します。*scratchpad* が UDF に渡される場合、入力時のスクラッチパッドは前の呼び出しのままです。

SQLUDF_TF_CLOSE (1)

これは、表関数への CLOSE 呼び出しです。これは、OPEN 呼び出しと同じように、外部 CLOSE 処理 (例えば、ソース・ファイルのクローズ) と、リソースの解放 (特に NO FINAL CALL ケース) を実行するために使用することができます。

結合や副照会が関係している場合、OPEN/FETCH.../CLOSE 呼び出しはステートメントの実行内で繰り返すことができますが、FIRST 呼び出しと FINAL 呼び出しはそれぞれ 1 回ずつしか実行できません。FIRST 呼び出しと FINAL 呼び出しが現れるのは、表関数に対して FINAL CALL が指定される場合だけです。

SQLUDF_TF_FINAL (2)

これは FINAL 呼び出しで、表関数に対して FINAL CALL が指定された場合にだけ現れます。これは FIRST 呼び出しのように、ステートメントの実行につき 1 回だけ現れます。この呼び出しの目的は、リソースの解放にあります。

SQLUDF_TF_FINAL_CRA (255)

これは FINAL 呼び出しであり、上記の FINAL 呼び出しと同一ですが、さらに別の特性が 1 つ追加されています。つまりこれは、SQL を発行できると定義されている UDF に対する追加であり、UDF は CLOSE カーソル以外のどの SQL も発行してはならない場合のための追加です。(SQLCODE -396、SQLSTATE 38505) 例えば、DB2 が COMMIT 処理の途中にあるときは、新規の SQL を受け入れることができないので、その時点で UDF に対して FINAL 呼び出しが発行された場合はすべて、255 FINAL 呼び出しになります。どのレベルの SQL アクセス権ももっていないと定義された UDF が 255 FINAL 呼び出しを受信することはないのに対して、SQL を使用する UDF は、いずれかのタイプの FINAL 呼び出しを受け取ることがあります。

リソースの解放

獲得したリソースを解放するルーチンを作成します。表関数の場合、CLOSE 呼び出しと FINAL 呼び出しの 2 つで通常この解放を行うことができます。CLOSE 呼び出しは、OPEN 呼び出しと対になり、ステートメントの実行内で複数回実行することができます。FINAL 呼び出しが行われるのは、UDF に FINAL CALL が指定される場合だけで、ステートメントにつき 1 回です。

UDF のすべての OPEN/FETCH/CLOSE シーケンスに 1 つのリソースを適用できる場合、FIRST 呼び出し時にこのリソースを獲得し、FINAL 呼び出し時にそれを解放する UDF を作成します。スクラッチパッドが通常このリソースを追跡します。表関数では、FINAL CALL が指定される場合、スクラッチパッドが初期化されるのは FIRST 呼び出しの前だけです。FINAL CALL が指定されていない場合には、各 OPEN 呼び出しの前に再初期化されます。

リソースがそれぞれの OPEN/FETCH/CLOSE シーケンスに対して固有である場合には、CLOSE 呼び出し時にリソースを解放する UDF を作成します。

注: 表関数が副照会または結合関数中にある場合、DB2 オプティマイザがステートメントの実行を編成する方法に応じて、OPEN/FETCH/CLOSE シーケンスが複数回出現する可能性が高くなります。

call-type は、INTEGER 値の形式をとります。DB2 は、データ・タイプとサーバーのオペレーティング・システムに応じて、*call-type* が表すデータの位置を調整します。

dbinfo この引数は、ルーチンを呼び出す前に DB2 によって設定されます。これは、ルーチンに対する CREATE ステートメントに DBINFO キーワードを指定した場合にのみ存在します。引数は、ヘッダー・ファイル `sqludf.h` に定義されている `sqludf_dbinfo` 構造です。この構造内で名前と ID の入れ

られる変数は、本リリースの DB2 で指定できる最長の値より長くなること
がありますが、将来のリリースとの互換性を確保するためにこのように定義
されています。それぞれの名前および ID 変数を補完する長さ変数を使用し
て、実際に使用される変数の一部を読み取るか抽出することができます。

dbinfo 構造には以下のエレメントが含まれます。

1. データベース名の長さ (dbnamelen)

次に挙げるデータベース名 の長さ。このフィールドは符号なし短整数
です。

2. データベース名 (dbname)

現在接続されているデータベースの名前。このフィールドは、128 文字
の長 ID です。上記のデータベース名の長さ フィールドは、このフィ
ールドの実際の長さを示します。NULL 終了符や埋め込みは含まれま
せん。

3. アプリケーション許可 ID の長さ (authidlen)

次に挙げるアプリケーション許可 ID の長さ。このフィールドは符号なし
短整数です。

4. アプリケーション許可 ID (authid)

アプリケーションのランタイム許可 ID。このフィールドは、128 文字
の長 ID です。NULL 終了符や埋め込みは含まれません。上述のアプ
リケーション許可 ID の長さ フィールドは、このフィールドの実際の
長さを示します。

5. 環境コード・ページ (codepg)

これは、すべての DB2 データベース製品に共通の構造 (cdpg_db2)、旧
バージョンの DB2 データベース用に作成されたルーチンによって使用
される構造 (cdpg_cs)、旧バージョンの DB2 for z/OS 用の構造
(cdpg_mvs) という、3 つの 48 バイト構造の共用体です。移植性を考
えて、すべてのルーチンで共通構造 cdpd_db2 を使用することをお勧め
します。

cdpg_db2 構造は、以下に示すように、データベース内で有効なコード
化スキームを表す 3 つのコード・ページ情報セットの配列
(db2_ccsids_triplet) です。

- a. ASCII コード化スキーム。DB2 データベースの旧バージョンとの
互換性を保つために、データベースが Unicode データベースの場合
は、Unicode コード化スキームの情報がここに置かれるとともに、
3 つ目のエレメントに現れます。
- b. EBCDIC コード化スキーム
- c. Unicode コード化スキーム

コード化スキームの情報に続いて、ルーチン用のコード化スキーム
(db2_encoding_scheme) の添字を示します。配列の各エレメントは、次
の 3 つのフィールドで構成されます。

- db2_sbcs。 1 バイトのコード・ページで、符号なし長整数です。

- `db2_dbcs`。 2 バイトのコード・ページで、符号なし長整数です。
- `db2_mixed`。複合コード・ページ (混合コード・ページともいう) で、符号なし長整数です。

6. スキーマ名の長さ (`tbschemalen`)

次に挙げるスキーマ名 の長さ。表名が渡されない場合は 0 (ゼロ) が入ります。このフィールドは符号なし短整数です。

7. スキーマ名 (`tbschema`)

後に挙げる表名 のスキーマ。このフィールドは、128 文字の長 ID です。 NULL 終了符や埋め込みは含まれません。上記のスキーマ名の長さ フィールドは、このフィールドの実際の長さを示します。

8. 表名の長さ (`tbnamelen`)

後に挙げる表名 の長さ。表名が渡されない場合は 0 (ゼロ) が入ります。このフィールドは符号なし短整数です。

9. 表名 (`tbname`)

更新中または挿入中の表の名前です。このフィールドが設定されるのは、ルーチン参照が UPDATE ステートメントで SET 節の右側にあるか、 INSERT ステートメントの VALUES リスト内の項目になっている場合だけです。このフィールドは、128 文字の長 ID です。 NULL 終了符や埋め込みは含まれません。上記の表名の長さ フィールドは、このフィールドの実際の長さを示します。上記のスキーマ名 とこのフィールドがまとまって、完全修飾表名を形成します。

10. 列名の長さ (`colnamelen`)

次に挙げる列名 の長さ。列名が渡されない場合は 0 (ゼロ) が入ります。このフィールドは符号なし短整数です。

11. 列名 (`colname`)

表名の場合とまったく同じ条件の下では、このフィールドには更新中または挿入中の列の名前が入ります。それ以外の場合は、予想できません。このフィールドは、128 文字の長 ID です。 NULL 終了符や埋め込みは含まれません。上述の列名の長さ フィールドは、このフィールドの実際の長さを示します。

12. バージョン/リリース番号 (`ver_rel`)8 文字のフィールドで、製品およびそのバージョン、リリース、修正レベルを、 `pppvrrm` の書式で識別します。この書式は次のとおりです。

- `ppp` は、次のように製品を識別します。

DSN DB2 for z/OS または OS/390®

ARI SQL/DS または DB2 for VM and VSE

QSQ DB2 Database for System i

SQL DB2 Database for Linux, UNIX, and Windows

- `vv` は、2 桁のバージョン ID です。
- `rr` は、2 桁のリリース ID です。

• *m* は、1 桁の修正レベル ID です。

13. 予約済みのフィールド (resd0)

このフィールドは将来の利用のためのものです。

14. プラットフォーム (platform)

アプリケーション・サーバーのオペレーティング・システム (プラットフォーム) は、以下のとおりです。

SQLUDF_PLATFORM_AIX
AIX®

SQLUDF_PLATFORM_HP
HP-UX

SQLUDF_PLATFORM_LINUX
Linux

SQLUDF_PLATFORM_MVS
OS/390

SQLUDF_PLATFORM_NT
Windows 2000、Windows XP

SQLUDF_PLATFORM_SUN
Solaris オペレーティング・システム

SQLUDF_PLATFORM_WINDOWS95
Windows 95、Windows 98、Windows Me

SQLUDF_PLATFORM_UNKNOWN
不明なオペレーティング・システムまたはプラットフォーム

上記のリストに含まれていないその他のオペレーティング・システムについては、`sqludf.h` ファイルの内容を参照してください。

15. 表関数列リストの項目数 (numtfcoll)

後に挙げる「表関数列リスト」フィールドで指定された表関数列リストにある非ゼロ項目の数。

16. 予約済みのフィールド (resd1)

このフィールドは将来の利用のためのものです。

17. 現在のルーチン呼び出ししたストアード・プロシージャのルーチン ID (procid)。

ストアード・プロシージャのルーチン ID は、呼び出し元ストアード・プロシージャの名前を検索するのに使用できる `SYSCAT.ROUTINES` 内の `ROUTINEID` 列に一致します。このフィールドは 32 ビットの符号付き整数です。

18. 予約済みのフィールド (resd2)

このフィールドは将来の利用のためのものです。

19. 表関数列リスト (tfcoll)

これが表関数である場合、このフィールドは、DB2 が動的に割り振った短整数の配列へのポインターです。これが他のいずれかのタイプのルーチンである場合、このポインターは NULL になります。

このフィールドは表関数にのみ使用されます。最初の n 個の項目 (n は、表関数列リストの項目の数 (*number of table function column list*) フィールドで指定される)、`numtfc1` のみに関係します。 n は 0 のこともあります。いずれにしても、CREATE FUNCTION ステートメントの RETURNS TABLE(...) 節内の関数に定義される結果列の数以下になります。これらの値は、このステートメントが表関数から取得する必要のある列の序数に対応します。値が「1」の場合は最初に定義された結果列を表し、「2」の場合は 2 番目に定義された結果列を表し、3 番目以降も同様です。値は任意の順序にすることができます。 n はゼロのこともあります。SELECT COUNT(*) FROM TABLE(TF(...)) AS QQ に類似したステートメント (ただし実際の列値は照会には必要ない) の場合、変数 `numtfc1` がゼロになることがあるからです。

この配列は、最適化の機会を表します。UDF は、表関数のすべての結果列のすべての値を戻す必要はなく、特定のコンテキストに必要なものだけを戻します。戻されるのは、配列で (番号によって) 識別される列です。この最適化は、パフォーマンスを向上させるために UDF 論理を複雑にする場合があるので、UDF では、定義されたすべての列を戻すように選択することができます。

20. ユニークなアプリケーション ID (`appl_id`)

このフィールドは、NULL 文字で終了する C のストリングを指すポインターであり、アプリケーションの DB2 への接続を固有識別します。これは、接続時に DB2 によって生成されます。

ストリングの最大長は 32 文字であり、その形式は、クライアントと DB2 の間で設定された接続タイプによって決まります。通常は以下のような形式です。

`x.y.ts`

ここで、 x と y は接続タイプに応じて変わりますが、 ts は YYMMDDHHMMSS という形式の 12 文字のタイム・スタンプで、固有性を確実にするために DB2 によって調整されることがあります。

Example: *LOCAL.db2inst.980707130144

21. 予約済みのフィールド (`resd3`)

このフィールドは将来の利用のためのものです。

C および C++ ルーチンでのグラフィック・ホスト変数

パラメーター入力や出力によって GRAPHIC データを送受信する、C または C++ で書かれたルーチンはすべて、通常 WCHARTYPE NOCONVERT オプションを指定してプリコンパイルしなければなりません。これは、そのようなパラメーターによって渡される GRAPHIC データが、`wchar_t` 処理コード形式ではなく、DBCS 形式であると見なされるからです。NOCONVERT を使用すると、ルーチン内の

SQL ステートメントで操作される GRAPHIC データも DBCS 形式であると見なされ、パラメーター・データの形式と一致します。

WCHARTYPE NOCONVERT を使用すると、GRAPHIC ホスト変数とデータベース・マネージャーの間では文字コード変換は実行されません。GRAPHIC ホスト変数を用いたデータは、無変換の DBCS 文字としてデータベース・マネージャーに送受信されます。WCHARTYPE NOCONVERT を使用しなくても、ルーチン内の `wchar_t` 形式の GRAPHIC データを操作できますが、入出力変換は手動で実行しなければなりません。

CONVERT は、FENCED ルーチン内で使用することができますが、ルーチンの SQL ステートメント内の GRAPHIC データに影響を及ぼします。ただし、ルーチンのパラメーターを介して渡されたデータには影響を与えません。NOT FENCED ルーチンは、必ず NOCONVERT オプションを使用して作成してください。

要約すると、入力または出力パラメーターによってルーチンに渡したり、ストアード・プロシージャから送られてくる GRAPHIC データは、WCHARTYPE オプションによってどのようにプリコンパイルされたかに関係なく、DBCS 形式になります。

C++ のタイプ修飾

C++ 関数名は、多重定義することができます。2 つの C++ 関数に同一名が付いていても、以下のようにそれぞれ異なる引数をもっていれば共存することができます。

```
int func( int i )
```

および

```
int func( char c )
```

C++ コンパイラーは、デフォルトで関数名をタイプ修飾つまり「マングル」します。その意味するところは、前記の 2 つの例の `func_Fi` と `func_Fc` の場合のように、引数タイプ名がその関数名に付加されて解決されるということです。マングルされた名前はオペレーティング・システムごとに異なるため、マングル名を明示的に使用するコードは移植可能ではありません。

Windows オペレーティング・システムでは、`.obj` (オブジェクト) ファイルからタイプ修飾関数名を判別することができます。

Windows 上の Microsoft Visual C++ コンパイラーの場合、以下のように `dumpbin` コマンドを使用して、`.obj` (オブジェクト) ファイルからタイプ修飾関数名を判別することができます。

```
dumpbin /symbols myprog.obj
```

ただし `myprog.obj` は、プログラム・オブジェクト・ファイルです。

UNIX オペレーティング・システムでは、`nm` コマンドを使用して、`.o` (オブジェクト) ファイルからかまたは共用ライブラリーからタイプ修飾関数名を判別することができます。このコマンドは大量の出力を生成することがあるので、次のように `grep` を介して出力をパイピングして正しい行を探すことをお勧めします。

```
nm myprog.o | grep myfunc
```


ただし myprog.o はプログラム・オブジェクト・ファイル、そして myfunc はプログラムのソース・ファイル内の関数です。

これらのコマンドで生成される出力ではすべて、マングルされた関数名の入った行が示されます。例えば UNIX 上では、この行は次のようになります。

```
myfunc__FP1T1PsT3PcN35| 3792|unamex| | ...
```

上記のコマンドのうちのいずれかでマングルされた関数名を取得し終わったら、該当するコマンド内でそれを使用することができます。それについては、上記の UNIX の例で得たマングル関数名を使った解説がこの項の後半で述べられています。Windows の場合も、取得したマングル関数名を同じように使用することができます。

CREATE ステートメントを使用してルーチンを登録するときは、マングルされた関数名を EXTERNAL NAME 節内に指定する必要があります。以下に例を示します。

```
CREATE FUNCTION myfunco(...) RETURNS...
...
EXTERNAL NAME '/whatever/path/myprog!myfunc__FP1T1PsT3PcN35'
...
```

多重定義された C++ 関数名がルーチン・ライブラリー内にはない場合、extern "C" を使用すれば、コンパイラで関数名がタイプ修飾されないようにするオプションを利用できます。(UDF に付ける SQL 関数名は常に多重定義できることに注意してください。DB2 では、その名前および指定されているパラメーターに基づいてどのライブラリー関数を呼び出すかを決められるからです。)

```
#include <string.h>
#include <stdlib.h>
#include "sqludf.h"

/*-----*/
/* function fold: output = input string is folded at point indicated */
/*                               by the second argument. */
/*           inputs: CLOB,          input string */
/*                   LONG           position to fold on */
/*           output: CLOB          folded string */
/*-----*/
extern "C" void fold(
    SQLUDF_CLOB      *in1,          /* input CLOB to fold */
    ...
    ...
)
/* end of UDF: fold */

/*-----*/
/* function find_vowel: */
/*           returns the position of the first vowel. */
/*           returns error if no vowel. */
/*           defined as NOT NULL CALL */
/*           inputs: VARCHAR(500) */
/*           output: INTEGER */
/*-----*/
extern "C" void findvwl(
    SQLUDF_VARCHAR  *in,          /* input smallint */
    ...
    ...
)
/* end of UDF: findvwl */
```

この例では、fold と findvwl という UDF はコンパイラーによってタイプ修飾されないで、それぞれの実際の名前を使用して CREATE FUNCTION ステートメント中で登録する必要があります。同様に、extern "C" を使用して C++ ストアド・プロシージャまたはメソッドをコード化すると、その未修飾の関数名が CREATE ステートメント内で使用されます。

C および C++ プロシージャからの結果セットの戻り

プロシージャの結果セットの検索をサポートする API を使ってインプリメントされる呼び出しルーチンまたはアプリケーションに結果セットを戻す C および C++ プロシージャを開発することができます。ほとんどの API はプロシージャの結果セットの検索をサポートしていますが、組み込み SQL はサポートしていません。

結果セットの C および C++ の表現は、SQL カーソルです。プロシージャ内に宣言およびオープンされたものの、明示的にクローズされなかった SQL カーソルがある場合、それらはプロシージャの戻りの前に呼び出し元に戻すことができます。結果セットが呼び出し元に戻される順序は、カーソル・オブジェクトがルーチン内でオープンされる順序と同じです。CREATE PROCEDURE ステートメントまたはプロシージャのインプリメンテーションでは、結果セットを戻すための追加のパラメーターは必要がありません。

前提条件

C および C++ ルーチンの作成方法の概略を理解しておく、C および C++ プロシージャから結果を戻すための以下の手順のステップを容易に理解できます。

C および C++ ルーチンの作成

C または C++ の組み込み SQL プロシージャで宣言されるカーソルは、両方向スクロール・カーソルではありません。

手順

C または C++ プロシージャから結果セットを戻すには、次のようにします。

1. C または C++ プロシージャの CREATE PROCEDURE ステートメントでは、他にも適切な節を指定する中で、特に DYNAMIC RESULT SETS 節に、プロシージャによって戻される結果セットの最大数と等しい値を指定しなければなりません。
2. プロシージャ宣言内で、呼び出し元に戻される結果セットのためのパラメーター・マーカは不要です。
3. ルーチンの C または C++ プロシージャのインプリメンテーションでは、ホスト変数を宣言する宣言セクション内の DECLARE CURSOR ステートメントを使ってカーソルを宣言します。このカーソル宣言により SQL がカーソルに関連付けられます。
4. C または C++ ルーチン・コード内で、OPEN ステートメントを実行することによりカーソルをオープンします。これにより DECLARE CURSOR ステートメントで指定した照会が実行され、照会の結果がカーソルに関連付けられます。

5. オプション: FETCH ステートメントを使って、カーソルに関連付けられた結果セット内の行を取り出します。
- 6.

カーソルのクローズに使用する CLOSE ステートメントは、プロシージャが呼び出し元に戻る前に実行してはなりません。オープン・カーソルはプロシージャが戻るときに結果セットとして呼び出し元に戻されます。

プロシージャの戻り時に複数のカーソルがオープンしたままになっていると、カーソルに関連付けられた結果セットが、オープンされた順番で呼び出し元に戻されます。プロシージャによって戻ることができるのは、DYNAMIC RESULT SETS 節の値に指定した結果セットの最大数までです。プロシージャのインプリメンテーションでオープンになったままのカーソルの数が DYNAMIC RESULT SETS 節で指定した値よりも大きい場合は、超過した結果セットが単に戻されないだけです。この状況では DB2 はエラーまたは警告を発生しません。

C または C++ プロシージャの作成が正常に完了したなら、DB2 コマンド行プロセッサまたは DB2 コマンド・ウィンドウから CALL ステートメントを使ってプロシージャを呼び出し、結果セットが正常に呼び出し元に戻されているかどうかを確認することができます。

プロシージャや他のタイプのルーチンの呼び出しの詳細については、以下を参照してください。

- ルーチンの呼び出し

C および C++ ルーチンの作成

C または C++ ライブラリーを参照するプロシージャまたは関数は、他のインプリメンテーションを持つ外部ルーチンと類似の方法で作成されます。このタスクはいくつかのステップで構成されますが、それにはルーチンの CREATE ステートメントの公式化、ルーチンのインプリメンテーションのコーディング、プリコンパイル、コードのコンパイルとリンク、およびソース・コードのデプロイメントが含まれます。

C または C++ ルーチンをインプリメントするのは、以下のような場合です。

- データベースにアクセスするルーチンや、データベースの外部でアクションを実行するルーチンに複雑なロジックをカプセル化したい場合。
- 複数のアプリケーション、CLP、他のルーチン (プロシージャ、関数 (UDF)、メソッド)、トリガーのいずれかから、カプセル化されたロジックを呼び出す必要がある場合。
- このロジックは、C または C++ などの組み込み SQL プログラミング言語を使用してコーディングするのが最適です。

前提条件

- C および C++ ルーチンのインプリメンテーションに関する知識。C および C++ ルーチン全般を学習するには、以下を参照してください。

- 185 ページの『第 5 章 C および C++ ルーチン』

- アプリケーション開発サポートを組み込んでいる IBM Data Server Clientが、クライアント・コンピューター上にインストールされていること。
- データベース・サーバーが、ルーチン開発用の DB2 サポートの C または C++ コンパイラーをサポートするオペレーティング・システムを実行していること。
- 必要なコンパイラーがデータベース・サーバー上にインストールされている必要があります。
- 外部ルーチンを作成する CREATE ステートメントの実行権限。CREATE PROCEDURE ステートメントまたは CREATE FUNCTION ステートメントの実行に必要な特権については、ステートメントの資料を参照してください。

手順

1. 選択したプログラミング言語 (C または C++) でルーチン・ロジックをコーディングします。
 - C と C++ ルーチンおよび C と C++ ルーチン・フィーチャーの一般情報については、『前提条件』のセクションで参照されているトピックを参照してください。
 - 追加の C 機能に必要な C または C++ ヘッダー・ファイル、および SQL データ・タイプと SQL 実行サポートに必要な DB2 C または C++ ヘッダー・ファイルを組み込みます。sqludf.h、sql.h、sqlda.h、sqlca.h、および memory.h の各ヘッダー・ファイルを組み込みます。
 - ルーチン・パラメーター・シグニチャーが、以下のいずれかのサポートされるパラメーター・スタイルを使用してインプリメントされている必要があります。すべての C および C++ ルーチンにパラメーター・スタイル SQL を使用することを強くお勧めします。スクラッチパッドおよび DBINFO 構造は、パラメーターとして C および C++ ルーチンに渡されます。パラメーター・シグニチャーおよびパラメーター・インプリメンテーションの詳細については、以下を参照してください。
 - 189 ページの『C および C++ ルーチンでのパラメーター』
 - 191 ページの『パラメーター・スタイル SQL C および C++ プロシージャ』
 - 194 ページの『パラメーター・スタイル SQL C および C++ 関数』
 - ホスト変数およびパラメーター・マーカを、組み込み SQL C および C++ アプリケーションに対するのと同じ方法で宣言します。DB2 SQL データ・タイプにマップするデータ・タイプを正しく使用するよう注意してください。DB2 と C または C++ データ・タイプとの間のデータ・タイプ・マッピングの詳細については、以下を参照してください。
 - 202 ページの『C および C++ ルーチンにおいてサポートされる SQL データ・タイプ』
 - ルーチン・ロジックを組み込みます。ルーチン・ロジックは、C または C++ プログラミング言語でサポートされる任意のコードで構成できます。これには、組み込み SQL アプリケーションと同じ方法でインプリメントされる組み込み SQL ステートメントの実行を含めることもできます。組み込み SQL における SQL ステートメントの実行の詳細については、以下を参照してください。

- 「組み込み SQL アプリケーションの開発」にある『組み込み SQL アプリケーションにおける SQL ステートメントの実行』
 - ルーチンがプロシージャで、ルーチンの呼び出し元に結果セットを戻したい場合、結果セット用のパラメーターは必要ありません。ルーチンから結果セットを戻す方法の詳細については、以下を参照してください。
 - 232 ページの『C および C++ プロシージャからの結果セットの戻り』
 - ルーチンの末尾にルーチンの戻り値を設定します。
2. ライブラリー・ファイルを生成するコードを作成します。組み込み SQL C および C++ ルーチンを作成する方法の詳細については、以下を参照してください。
 - 236 ページの『C および C++ ルーチン・コードのビルド』
 3. そのライブラリーをデータベース・サーバー上の DB2 *function* ディレクトリーにコピーします。DB2 ルーチンに関連したライブラリーは、*function* ディレクトリーに保管することをお勧めします。*function* ディレクトリーの詳細については、CREATE PROCEDURE ステートメントまたは CREATE FUNCTION ステートメントのいずれかの EXTERNAL 節を参照してください。

ライブラリーをサーバー上の別のディレクトリーにコピーすることもできますが、ルーチンを正常に呼び出すには、ライブラリーの完全修飾パス名をメモしておく必要があります。次のステップでこれが必要になるからです。

4. 該当するルーチン・タイプの SQL 言語 CREATE ステートメント (CREATE PROCEDURE または CREATE FUNCTION) を動的または静的に実行します。
 - LANGUAGE 節に、C という値を指定します。
 - PARAMETER STYLE 節に、ルーチン・コードでインプリメントした有効なパラメーター・スタイルの名前を指定します。PARAMETER STYLE SQL を使用することを強くお勧めします。
 - EXTERNAL 節に、ルーチンと関連したライブラリーの名前を指定します。そのためには、以下のいずれかの値を使用します。
 - ルーチン・ライブラリーの完全修飾パス名。
 - *function* ディレクトリーを基準にしたルーチン・ライブラリーの相対パス名。

EXTERNAL 節にライブラリーの完全修飾パス名または相対パス名を指定しない場合、DB2 はデフォルトで、*function* ディレクトリー内でライブラリーを探します。

- ルーチンがプロシージャの場合に、DYNAMIC RESULT SETS に数値を指定すると、呼び出し元に 1 つ以上の結果セットが戻されます。
 - ルーチンの特徴を指定するために使用する CREATE ステートメントで、その他のデフォルト以外の節値を指定する。
- C または C++ ルーチンを呼び出すには、337 ページの『第 10 章 ルーチンの呼び出し』を参照してください。

C および C++ ルーチン・コードのビルド

組み込み SQL 用の C または C++ ルーチンのインプリメンテーション・コードを作成したなら、ルーチン呼び出せるようにするには、その前にインプリメンテーション・コードをライブラリーにビルドしてデプロイする必要があります。組み込み SQL 用 C および C++ ルーチンのビルドに必要なステップは、組み込み SQL 用 C および C++ アプリケーションの構築に必要なステップに似ていますが、いくつかの点で異なります。ルーチンに組み込み SQL ステートメントが含まれていない場合は、同じステップを進めることができます。手順は高速かつ簡単です。

C および C++ ルーチンをビルドするには、次の 2 とおりの方法があります。

- DB2 サンプル・ビルド・スクリプト (UNIX) またはビルド・バッチ・ファイル (Windows) を使用する。
- DB2 コマンド・ウィンドウから、DB2 および C または C++ コンパイラーのコマンドを入力する。

ルーチン用の DB2 サンプル・ビルド・スクリプトおよびバッチ・ファイルは、デフォルトでサポートされるコンパイラーを使用する特定のオペレーティング・システムのためのユーザー作成ルーチンだけでなく、DB2 サンプル・ルーチン (プロシージャおよびユーザー定義関数) 用に設計されています。

C および C++ 用の DB2 サンプル・ビルド・スクリプトおよびバッチ・ファイルには個別セットがあります。一般に、組み込み SQL ルーチンをビルドするには、ビルド・スクリプトまたはバッチ・ファイル (いずれも必要に応じて容易に変更できる) を使用するのが最も簡単な方法ですが、DB2 コマンド・ウィンドウからルーチンをビルドする方法も知っておくと、いろいろと役に立ちます。

サンプル bldrtn スクリプトを使用した C および C++ ルーチン・コードのビルド

C および C++ ルーチンのソース・コードのビルドは、C および C++ ルーチン作成の副次作業です。この作業は、DB2 サンプル・ビルド・スクリプト (UNIX) およびバッチ・ファイル (Windows) を使用することにより、迅速で簡単に行うことができます。サンプル・ビルド・スクリプトは、組み込み SQL ステートメントを含むソース・コードにも含まないソース・コードにも使用できます。ビルド・スクリプトは、C および C++ ソース・コードのプリコンパイル、コンパイル、リンクを処理します。これを使用しない場合は、コマンド行から個別のステップを実行する必要があります。また、指定されたデータベースへのパッケージのバインドも行います。

C および C++ ルーチンをビルドするためのサンプル・ビルド・スクリプトの名前は、bldrtn です。これを使用してビルドできるサンプル・プログラムと一緒に、次の DB2 ディレクトリーに置かれています。

- C の場合: `sqllib/samples/c/`
- C++ の場合: `sqllib/samples/cpp/`

bldrtn スクリプトを使用して、プロシージャと関数インプリメンテーションの両方を含むソース・コード・ファイルをビルドすることができます。このスクリプトは、以下の処理を行います。

- ユーザーが指定したデータベースとの接続を確立する

- ユーザーが指定したソース・コード・ファイルをプリコンパイルする
- パッケージを現行データベースにバインドする
- ソース・コードをコンパイルしてリンクし、共用ライブラリーを生成する
- 共用ライブラリーをデータベース・サーバーの DB2 function ディレクトリーにコピーする

bldrtn スクリプトは、次の 2 つの引数を受け入れます。

- ソース・コード・ファイルの名前 (ファイル拡張子なし)
- 接続が確立される先のデータベースの名前

データベース・パラメーターはオプションです。データベース名を指定しない場合は、プログラムはデフォルトの sample データベースを使用します。ルーチンはデータベースがあるインスタンスと同じインスタンス上で構築する必要があるため、ユーザー ID とパスワードのための引数は不要です。

前提条件

- 1 つ以上のルーチン・インプリメンテーションを含むソース・コード・ファイル
- 現行 DB2 インスタンス内の、ルーチンが作成されるデータベースの名前。

1 つ以上のルーチン・コード・インプリメンテーションを含むソース・コード・ファイルをビルドするには、以下のステップを実行します。

1. DB2 コマンド・ウィンドウをオープンする。
2. ソース・コード・ファイルを bldrtn スクリプト・ファイルと同じディレクトリーにコピーする。
3. サンプル・データベースにルーチンを作成する場合は、次のように、ビルド・スクリプト名に続いて、ソース・コード・ファイルの名前 (.sqc または .sqc ファイル拡張子なし) を入力する。

```
bldrtn <file-name>
```

別のデータベースにルーチンを作成する場合は、次のように、ビルド・スクリプト名、ソース・コード・ファイル名 (ファイル拡張子なし)、データベース名を入力します。

```
bldrtn <file-name> <database-name>
```

スクリプトによってソース・コードのプリコンパイル、コンパイル、リンクが行われ、共用ライブラリーが生成されます。次に、スクリプトによって共用ライブラリーがデータベース・サーバーの function ディレクトリーにコピーされます。

4. ルーチン・インプリメンテーションを含むソース・コード・ファイルをビルドしたのが今回が初めてでない場合は、データベースを停止してから再始動することにより、DB2 で新しいバージョンの共用ライブラリーが使用されるようにする。これを行うには、コマンド行で db2stop に続けて db2start を入力します。

ルーチン共用ライブラリーのビルドとデータベース・サーバー上の function ディレクトリーへのデプロイが正常に終了したなら、C および C++ ルーチンの作成作業に関連したステップを完了する必要があります。ルーチンの作成が完了すると、ルーチンを呼び出すことができます。

サンプル・ビルド・スクリプトを使用した C または C++ でのルーチンの構築 (UNIX)

DB2 には、C および C++ プログラムをコンパイルしてリンクするためのビルド・スクリプトが用意されていて、C のルーチン用は `sqllib/samples/c` ディレクトリに、C++ のルーチン用は `sqllib/samples/cpp` ディレクトリに置かれており、それと一緒に、それらのファイルを使用してビルドできるサンプル・プログラムも置かれています。

スクリプト `bldrtn` には、ルーチン (ストアード・プロシージャとユーザー定義関数) を構築するためのコマンドが入っています。このスクリプトは、データベース・マネージャーがロードできてしかもクライアント・アプリケーションから呼び出せるルーチンを共用ライブラリー中でコンパイルします。

第 1 パラメーター `$1` には、ソース・ファイルの名前を指定します。第 2 パラメーター `$2` には、接続先のデータベースの名前を指定します。

データベース・パラメーターはオプションです。データベース名を指定しない場合は、プログラムはデフォルトの `sample` データベースを使用します。ストアード・プロシージャは、データベースが置かれているのと同じインスタンス上で構築する必要があるため、ユーザー ID やパスワードを指定するパラメーターはありません。

この後の例は、次のものを使用してルーチンの共用ライブラリーを構築する方法を示しています。

- ストアード・プロシージャ
- 非組み込み SQL ユーザー定義関数 (UDF)
- 組み込み SQL ユーザー定義関数 (UDF)

ストアード・プロシージャの共用ライブラリー

C 用のソース・ファイル `spserver.sqc` および C++ 用のソース・ファイル `spserver.sqC` からサンプル・プログラム `spserver` を構築するには、次のように入力します。

1. `sample` データベースに接続している場合は、次のようにビルド・スクリプト名とプログラム名を入力します。

```
bldrtn spserver
```

他のデータベースに接続しているときは、さらにデータベース名も入力します。

```
bldrtn spserver database
```

スクリプトは、共用ライブラリーをサーバー上の `sqllib/function` というパスにコピーします。

2. 次に、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
```

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、個別に呼び出すこともできます。

- 次に、ストアード・プロシージャを構築するのが今回が初めてではない場合は、データベースをいったん停止してから再始動し、新しいバージョンの共有ライブラリーが認識されるようにします。これを行うには、コマンド行で db2stop に続けて db2start を入力します。

共有ライブラリー spserver の構築が完了したなら、共有ライブラリーにアクセスするクライアント・アプリケーション spclient を構築することができます。

spclient は、スクリプト bldapp を使用して構築することができます。

共有ライブラリーのストアード・プロシージャを呼び出すには、spclient *database userid password* と入力してサンプル・クライアント・アプリケーションを実行します。

ここで、

database

接続先のデータベースの名前です。名前は、sample かその別名、またはその他のデータベース名にすることができます。

userid 有効なユーザー ID です。

password

ユーザー ID の有効なパスワードです。

クライアント・アプリケーションは共有ライブラリー spserver にアクセスし、さまざまなストアード・プロシージャ関数をサーバー・データベース上で実行します。出力は、クライアント・アプリケーションに戻されます。

組み込み SQL UDF の共有ライブラリー

sample データベースに接続しているときに、組み込み SQL ユーザー定義関数プログラム udfemsrv を、C 用のソース・ファイル udfemsrv.sqc および C++ 用のソース・ファイル udfemsrv.sqc から構築するには、次のようにビルド・スクリプト名とプログラム名を入力します。

```
bldrtn udfemsrv
```

他のデータベースに接続しているときは、さらにデータベース名も入力します。

```
bldrtn udfemsrv database
```

スクリプトは、UDF を sqllib/function ディレクトリーにコピーします。

udfemsrv の構築が完了したなら、それを呼び出すクライアント・アプリケーション udfemcli を構築できます。スクリプト bldapp を使用して、sqllib/samples/c 内のソース・ファイル udfemcli.sqc から udfemcli クライアント・プログラムを構築することができます。

共有ライブラリー内の UDF を呼び出すには、udfemcli *database userid password* と入力してクライアント・アプリケーションを実行します。

ここで、

database

接続先のデータベースの名前です。名前は、`sample` かその別名、またはその他のデータベース名にすることができます。

userid 有効なユーザー ID です。

password

ユーザー ID の有効なパスワードです。

クライアント・アプリケーションは共用ライブラリー `udfemsrv` にアクセスし、ユーザー定義関数をサーバー・データベース上で実行します。出力は、クライアント・アプリケーションに戻されます。

注: 組み込み SQL プロシージャを構築した場合、`LD_LIBRARY_PATH` オペレーティング・システム環境変数は無視されます。

Windows での C/C++ ルーチンの構築

DB2 には、C および C++ の DB2 API と組み込み SQL プログラムをコンパイルおよびリンクするためのビルド・スクリプトが用意されています。このファイルを使用して構築できるサンプル・プログラムと一緒に `sqllib%samples%c` および `sqllib%samples%c.cpp` ディレクトリーに置かれています。

バッチ・ファイル `bldrtn.bat` には、組み込み SQL ルーチン (ストアード・プロシージャとユーザー定義関数) を構築するためのコマンドが入っています。バッチ・ファイルは、サーバー上に DLL を構築します。これは 2 個のパラメーターをとります。それらは、バッチ・ファイル内では変数 `%1` と `%2` で表されます。

最初のパラメーター `%1` には、ソース・ファイルの名前を指定します。このバッチ・ファイルは、DLL 名としてソース・ファイル名を使用します。第 2 パラメーター `%2` には、接続先のデータベースの名前を指定します。DLL は、データベースが置かれているのと同じインスタンス上で構築する必要があるため、ユーザー ID やパスワードを指定するパラメーターはありません。

第 1 パラメーター (ソース・ファイル名) だけが必須です。データベース名は任意で指定します。データベース名を指定しない場合は、プログラムはデフォルトの `sample` データベースを使用します。

この後の例は、次のものを使用してルーチンの DLL を構築する方法を示しています。

- ストアード・プロシージャ
- 非組み込み SQL ユーザー定義関数 (UDF)
- 組み込み SQL ユーザー定義関数 (UDF)

ストアード・プロシージャの DLL

C ソース・ファイル `spserver.sqc` または C++ ソース・ファイル `spserver.sqx` から `spserver` DLL を構築するには、次のようにします。

1. 次のように、バッチ・ファイル名とプログラム名を入力します。

```
bldrtn spserver
```

他のデータベースに接続しているときは、さらにデータベース名も入力します。

```
bldrtn spserver database
```

このバッチ・ファイルは、サンプル・プログラムと同じディレクトリーに入っている、モジュール定義ファイル `spserver.def` を使用して DLL を構築します。このバッチ・ファイルは、DLL の `spserver.dll` をパス `sqllib%function` のサーバーにコピーします。

- 次に、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
```

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、個別に呼び出すこともできます。

- 次に、データベースを一度停止してから再始動し、新しい DLL が認識されるようにします。必要であれば、DLL にファイル・モードを設定して、DB2 インスタンスからアクセスできるようにします。

DLL `spserver` を構築し終わったら、それを呼び出すクライアント・アプリケーション `spclient` を構築することができます。

`spclient` は、バッチ・ファイル `bldapp.bat` を使用して構築することができます。

DLL を呼び出すには、次のように入力してサンプル・クライアント・アプリケーションを実行します。

```
spclient database userid password
```

ここで、

database

接続先のデータベースの名前です。名前は、`sample` かその別名、またはその他のデータベース名にすることができます。

userid 有効なユーザー ID です。

password

ユーザー ID の有効なパスワードです。

クライアント・アプリケーションは、DLL `spserver` にアクセスしてサーバー・データベース上のいくつかのルーチンを実行します。出力は、クライアント・アプリケーションに戻されます。

非組み込み SQL UDF の DLL

ユーザー定義関数 `udfsrv` をソース・ファイル `udfsrv.c` から構築するには、次のように入力します。

```
bldrtn udfsrv
```

このバッチ・ファイルは、サンプル・プログラム・ファイルと同じディレクトリーに入っているモジュール定義ファイル `udfsrv.def` を使用して、ユーザー定義関数

DLL を構築します。このバッチ・ファイルは、ユーザー定義関数 DLL の `udfsrv.dll` を `sqllib¥function` というパスのサーバーにコピーします。

`udfsrv` の構築が完了したなら、それを呼び出すクライアント・アプリケーション `udfcli` を構築できます。DB2 CLI が、このプログラムの組み込み SQL C および C++ バージョンとともに提供されます。

DB2 CLI `udfcli` プログラムは、`sqllib¥samples¥cli` のバッチ・ファイル `bldapp` を使用して、`udfcli.c` ソース・ファイルから構築できます。

組み込み SQL C `udfcli` プログラムは、`sqllib¥samples¥c` のバッチ・ファイル `bldapp` を使用して、`udfcli.sqc` ソース・ファイルから構築できます。

組み込み SQL C++ `udfcli` プログラムは、`sqllib¥samples¥cpp` のバッチ・ファイル `bldapp` を使用して、`udfcli.sqx` ソース・ファイルから構築できます。

UDF を実行するには、次のように入力します。

```
udfcli
```

この呼び出しアプリケーションは、`udfsrv` DLL から `ScalarUDF` 関数を呼び出します。

組み込み SQL UDF の DLL

`sqllib¥samples¥c` 内の C ソース・ファイル `udfemsrv.sqc` からか、または `sqllib¥samples¥cpp` 内の C++ ソース・ファイル `udfemsrv.sqx` から、組み込み SQL ユーザー定義関数ライブラリー `udfemsrv` を構築するには、次のように入力します。

```
bldrtn udfemsrv
```

他のデータベースに接続しているときは、さらにデータベース名も入力します。

```
bldrtn udfemsrv database
```

このバッチ・ファイルは、サンプル・プログラムと同じディレクトリーに入っているモジュール定義ファイル `udfemsrv.def` を使用してユーザー定義関数 DLL を構築します。このバッチ・ファイルは、ユーザー定義関数 DLL の `udfemsrv.dll` をパス `sqllib¥function` のサーバーにコピーします。

`udfemsrv` の構築が完了したなら、それを呼び出すクライアント・アプリケーション `udfemcli` を構築できます。バッチ・ファイル `bldapp` を使用して、`sqllib¥samples¥c` 内の C ソース・ファイル `udfemcli.sqc` からか、または `sqllib¥samples¥cpp` 内の C++ ソース・ファイル `udfemcli.sqx` から、`udfemcli` を構築することができます。

UDF を実行するには、次のように入力します。

```
udfemcli
```

呼び出し側アプリケーションは、`udfemsrv` DLL 内の UDF を呼び出します。

DB2 コマンド・ウィンドウからの C および C++ ルーチン・コードのビルド

C および C++ ルーチンのソース・コードのビルドは、C および C++ ルーチン作成の副次作業です。この作業は、コマンド行から手動で行うことができます。C または C++ ルーチン・コードに組み込み SQL ステートメントが含まれているかどうかにかかわらず、同じ手順を進めることができます。この作業のステップには、ルーチン・インプリメンテーションを含む C および C++ ソース・コードのプリコンパイル、コンパイル、リンクのほかに、生成されたパッケージのバインド (組み込み SQL ステートメントがあった場合) とルーチン・ライブラリーのデプロイが含まれます。ルーチン・パッケージのバインドを後に延ばす場合や、カスタマイズ済みビルド・スクリプトを作成する場合は、この作業を、プリコンパイラー、コンパイラー、またはバインド・オプションの使用テストの一環として、DB2 コマンド・ウィンドウから行うことも可能です。

別の方法として、DB2 サンプル・ビルド・スクリプトを使用して、この作業を簡単にすることができます。『サンプル・ビルド・スクリプトによる組み込み SQL C および C++ ルーチン・コードの作成』を参照してください。

前提条件

- 組み込み SQL 用 C または C++ ルーチン・インプリメンテーションを 1 つ以上含むソース・コード・ファイル。
- 現行 DB2 インスタンス内の、ルーチンが作成されるデータベースの名前。
- C および C++ ルーチンをビルドするのに必要な、オペレーティング環境固有のコンパイルおよびリンク・オプション。このトピックの下部にある関連リンクに記載されているトピックを参照してください。

1 つ以上のルーチン・コード・インプリメンテーションを含むソース・コード・ファイルをビルドするには、以下のステップを実行します。各ステップを示した例が後に続きます。

1. DB2 コマンド・ウィンドウをオープンする。
2. ソース・コード・ファイルを含むディレクトリーにナビゲートする。
3. ルーチンが作成されるデータベースとの接続を確立する。
4. ソース・コード・ファイルをプリコンパイルする。
5. 生成されたパッケージをデータベースにバインドする。
6. ソース・コード・ファイルをコンパイルする。
7. ソース・コード・ファイルをリンクして共用ライブラリーを生成する。ここでは、使用されるコンパイラーに合った、DB2 固有のコンパイルおよびリンク・オプションを使用する必要があります。
8. 共用ライブラリーをデータベース・サーバーの DB2 function ディレクトリーにコピーする。
9. ルーチン・インプリメンテーションを含むソース・コード・ファイルをビルドしたのが今回が初めてでない場合は、データベースを停止してから再始動することにより、DB2 で新しいバージョンの共用ライブラリーが使用されるようにする。これを行うには、db2stop コマンドに続けて db2start コマンドを実行します。

ルーチン・ライブラリーのビルドとデプロイを正常に終了したなら、C および C++ ルーチンの作成作業に関連したステップを完了する必要があります。C および C++ ルーチンの作成には、ソース・コード・ファイルにインプリメントされたルーチンごとに CREATE ステートメントを実行するというステップが含まれます。ルーチン呼び出せるようにするには、このステップも完了しなければなりません。

例

以下は、ルーチン・インプリメンテーションを含む、myfile.sqC という名前の組み込み SQL C++ ソース・コード・ファイルの再ビルドを示した例です。ルーチンは AIX オペレーティング・システム上でビルドされ、デフォルトでサポートされている IBM VisualAge® C++ コンパイラーを使用して 32 ビット・ルーチン・ライブラリーが生成されます。

1. DB2 コマンド・ウィンドウをオープンする。
2. ソース・コード・ファイルを含むディレクトリーにナビゲートする。
3. ルーチンが作成されるデータベースとの接続を確立する。

```
db2 connect to <database-name>
```

4. PREPARE コマンドを使用してソース・コード・ファイルをプリコンパイルする。

```
db2 prep myfile.sqC bindfile
```

プリコンパイルが正常に進行したかどうか、またはエラーがあったかどうかを示す出力が、プリコンパイラーによって生成されます。このステップで myfile.bnd という名前のバインド・ファイルが生成されます。次のステップでパッケージを生成するのにこのファイルを使用します。

5. BIND コマンドを使用して、生成されたパッケージをデータベースにバインドする。

```
db2 bind myfile.bnd
```

バインドが正常に進行したかどうか、またはエラーがあったかどうかを示す出力が、バインド・ユーティリティーによって生成されます。

6. コンパイルとリンクの推奨オプションを使用してソース・コード・ファイルをコンパイルする。

```
x1C_r -qstaticinline -I$HOME/sqlllib/include -c $myfile.C
```

エラーがある場合は、コンパイラーによって出力が生成されます。このステップで、myfile.exp という名前のエクスポート・ファイルが生成されます。

7. ソース・コード・ファイルをリンクして共用ライブラリーを生成する。

```
x1C_r -qmkshrobj -o $1 $1.o -L$ HOME/sqlllib/include/lib32 -lDB2
```

エラーがある場合は、リンカーによって出力が生成されます。このステップで、ファイル名が myfile の共用ライブラリーが生成されます。

8. 共用ライブラリーをデータベース・サーバーの DB2 function ディレクトリーにコピーする。

```
rm -f ~HOME/sqlllib/function/myfile
cp myfile $HOME/sqlllib/function/myfile
```

このステップで、DB2 がルーチン・ライブラリーを探すデフォルト・ディレクトリーにルーチン・ライブラリーが置かれます。ルーチン・ライブラリーのデブroyについての詳細は、C および C++ ルーチンの作成に関するトピックを参照してください。

9. これは以前にビルドされたルーチン・ソース・コード・ファイルの再ビルドなので、データベースを停止して再始動する。

```
db2stop  
db2start
```

C および C++ ルーチンのビルドは一般に、オペレーティング環境固有のサンプル・ビルド・スクリプトを使用して行うのが最も簡単です。このスクリプトは、コマンド行からルーチンをビルドする方法の参考として使用できます。

注: 組み込み SQL プロシージャを構築した場合、LD_LIBRARY_PATH オペレーティング・システム環境変数は無視されます。

C および C++ ルーチンのコンパイルとリンクのオプション

AIX C ルーチンのコンパイルとリンクのオプション

以下は、bldrtn ビルド・スクリプトに示されているように、AIX IBM C コンパイラーを使用して、C ルーチン (ストアド・プロシージャとユーザー定義関数) を構築するのにお勧めするコンパイルとリンクのオプションです。

bldrtn のコンパイルとリンクのオプション

コンパイル・オプション:

xlc_r ルーチンを他のルーチンと同じプロセスで実行する (THREADSAFE) か、またはエンジンそのもので実行する (NOT FENCED) かに応じて、マルチスレッド・バージョンの IBM C コンパイラーを使用してください。

\$EXTRA_CFLAG

64 ビット・サポートが使用可能なインスタンスの場合は「-q64」が入り、それ以外の場合は値は入りません。

-I\$DB2PATH/include

DB2 組み込みファイルのロケーションを指定します。例えば、`$HOME/sql1lib/include` のように指定します。

-c

コンパイルのみを実行し、リンクは実行しません。コンパイルとリンクは別個のステップです。

リンク・オプション:

x1c_r リンカーのフロントエンドとしてマルチスレッド・バージョンのコンパイラーを使用します。

\$EXTRA_CFLAG

64 ビット・サポートが使用可能なインスタンスの場合は「-q64」が入り、それ以外の場合は値は入りません。

-qmkshrobj

共用ライブラリーを作成します。

-o \$1 出力ファイル名を指定します。

\$1.o オブジェクト・ファイルを指定します。

-ldb2 DB2 ライブラリーとリンクします。

-L\$DB2PATH/\$LIB

DB2 ランタイム共用ライブラリーのロケーションを指定します。例えば、`$HOME/sqllib/$LIB`。 `-L` オプションを指定しないと、コンパイラーは次のパスを想定します。 `/usr/lib:/lib`。

-bE:\$1.exp

エクスポート・ファイルを指定します。エクスポート・ファイルには、ルーチンの一覧が入っています。

他のコンパイラー・オプションについては、コンパイラーの資料をご覧ください。

AIX C++ ルーチンのコンパイルとリンクのオプション

以下は、`bldrtn` ビルド・スクリプトに示されているように、AIX IBM XL C/C++ コンパイラーを使用して、C++ ルーチン (ストアード・プロシージャとユーザー定義関数) を構築するのにお勧めするコンパイルとリンクのオプションです。

`bldrtn` のコンパイルとリンクのオプション

コンパイル・オプション:

x1c_r ルーチンを他のルーチンと同じプロセスで実行する (THREADSAFE) か、またはエンジンそのもので実行する (NOT FENCED) かに応じたマルチスレッド・バージョンの IBM XL C/C++ コンパイラー。

\$EXTRA_CFLAG

64 ビット・サポートが使用可能なインスタンスの場合は「-q64」が入り、それ以外の場合は値は入りません。

-I\$DB2PATH/include

DB2 組み込みファイルのロケーションを指定します。例えば、`$HOME/sqllib/include` のように指定します。

-c

コンパイルのみを実行し、リンクは実行しません。コンパイルとリンクは別個のステップです。

リンク・オプション:

x1C_r リンカーのフロントエンドとしてマルチスレッド・バージョンのコンパイラーを使用します。

\$EXTRA_CFLAG

64 ビット・サポートが使用可能なインスタンスの場合は「-q64」が入り、それ以外の場合は値は入りません。

-qmksbobj

共用ライブラリーを生成します。

-o \$1 出力を、共用ライブラリー・ファイルとして指定します。

\$1.o プログラム・オブジェクト・ファイルを指定します。

-L\$DB2PATH/\$LIB

DB2 ランタイム共用ライブラリーのロケーションを指定します。例えば、`$HOME/sqllib/$LIB`。 **-L** オプションを指定しないと、コンパイラーは次のパスを想定します。 `/usr/lib:/lib`。

-ldb2 DB2 ライブラリーとリンクします。

-bE:\$1.exp

エクスポート・ファイルを指定します。エクスポート・ファイルには、ルーチンの一覧が入っています。

他のコンパイラー・オプションについては、コンパイラーの資料をご覧ください。

HP-UX C ルーチンのコンパイルとリンクのオプション

以下は、`bldrtn` ビルド・スクリプトに示されているように、HP-UX C コンパイラーを使用して、C ルーチン (ストアード・プロシージャとユーザー定義関数) を構築するのにお勧めするコンパイルとリンクのオプションです。

bldrtn のコンパイルとリンクのオプション

コンパイル・オプション:

cc C コンパイラー。

\$EXTRA_CFLAG

HP-UX プラットフォームが IA64 で、64 ビット・サポートが使用可能な場合は、このフラグには値 **+DD64** が入り、32 ビット・サポートが使用可能な場合は、値 **+DD32** が入ります。HP-UX プラットフォームが PA-RISC で、64 ビット・サポートが使用可能な場合は、これには値 **+DA2.0W** が入ります。PA-RISC プラットフォームでの 32 ビット・サポートの場合は、このフラグには値 **+DA2.0N** が入ります。

+DD64 IA64 版の HP-UX 用の 64 ビット・コードを生成する場合に使用する必要があります。

+DD32 IA64 版の HP-UX 用の 32 ビット・コードを生成する場合に使用する必要があります。

+DA2.0W

PA-RISC 版の HP-UX 用の 64 ビット・コードを生成する場合に使用する必要があります。

+DA2.0N

PA-RISC 版の HP-UX 用の 32 ビット・コードを生成する場合に使用する必要があります。

+u1 位置合わせしないデータ・アクセスを認めます。アプリケーションが位置合わせしないデータを使用する場合にのみ使用します。

+z 位置に依存しないコードを生成します。

-Ae HP ANSI 拡張モードを使用可能にします。

-\$DB2PATH/include

DB2 組み込みファイルのロケーションを指定します。例えば、**-\$DB2PATH/include**。

-D_POSIX_C_SOURCE=199506L

ルーチンを他のルーチンと同じプロセスで実行する (THREADSAFE) か、またはエンジンそのもので実行する (NOT FENCED) ときに必要な、**_REENTRANT** が必ず定義されるようにするための POSIX スレッド・ライブラリー・オプション。

-c コンパイルのみを実行し、リンクは実行しません。コンパイルとリンクは別個のステップです。

リンク・オプション:

ld リンクにリンカーを使用します。

-b 通常の実行可能ファイルではなく、共用ライブラリーを作成します。

-o \$1 出力を、共用ライブラリー・ファイルとして指定します。

\$1.o プログラム・オブジェクト・ファイルを指定します。

\$EXTRA_LFLAG

ランタイム・パスを指定します。設定する場合、32 ビットの場合は値 "+b\$HOME/sql1lib/lib32"、64 ビットの場合は "+b\$HOME/sql1lib/lib64" が入ります。設定しない場合は、これには値が入りません。

-L\$DB2PATH/\$LIB

DB2 ランタイム共用ライブラリーのロケーションを指定します。32 ビットの場合は \$HOME/sql1lib/lib32、64 ビットの場合は \$HOME/sql1lib/lib64 です。

-ldb2 DB2 ライブラリーとリンクします。

-lpthread

POSIX スレッド・ライブラリーとリンクします。

他のコンパイラー・オプションについては、コンパイラーの資料をご覧ください。

HP-UX C++ ルーチンのコンパイルとリンクのオプション

以下は、bldrtn ビルド・スクリプトに示されているように、HP-UX C++ コンパイラーを使用して、C++ ルーチン (ストアード・プロシージャとユーザー定義関数) を構築するのにお勧めするコンパイルとリンクのオプションです。

bldrtn のコンパイルとリンクのオプション

コンパイル・オプション:

aCC HP aC++ コンパイラー。

\$EXTRA_CFLAG

HP-UX プラットフォームが IA64 で、64 ビット・サポートが使用可能な場合は、このフラグには値 **+DD64** が入り、32 ビット・サポートが使用可能な場合は、値 **+DD32** が入ります。HP-UX プラットフォームが PA-RISC で、64 ビット・サポートが使用可能な場合は、これには値 **+DA2.0W** が入ります。PA-RISC プラットフォームでの 32 ビット・サポートの場合は、このフラグには値 **+DA2.0N** が入ります。

+DD64 IA64 版の HP-UX 用の 64 ビット・コードを生成する場合に使用する必要があります。

+DD32 IA64 版の HP-UX 用の 32 ビット・コードを生成する場合に使用する必要があります。

+DA2.0W

PA-RISC 版の HP-UX 用の 64 ビット・コードを生成する場合に使用する必要があります。

+DA2.0N

PA-RISC 版の HP-UX 用の 32 ビット・コードを生成する場合に使用する必要があります。

+u1 位置合わせしないデータ・アクセスを認めます。

+z 位置に依存しないコードを生成します。

-ext "long long" サポートを含むさまざまな C++ 拡張子を許可します。

-mt ルーチンを他のルーチンと同じプロセスで実行する (THREADSAFE) か、またはエンジンそのもので実行する (NOT FENCED) かに応じて、HP aC++ コンパイラーのマルチスレッド・サポートを使用可能にします。

-I\$DB2PATH/include

DB2 組み込みファイルのロケーションを指定します。例えば、\$DB2PATH/include。

-c コンパイルのみを実行し、リンクは実行しません。コンパイルとリンクは別個のステップです。

リンク・オプション:

aCC HP aC++ コンパイラーをリンカーのフロントエンドとして使用します。

\$EXTRA_CFLAG

HP-UX プラットフォームが IA64 で、64 ビット・サポートが使用可能な場合は、このフラグには値 **+DD64** が入り、32 ビット・サポートが使用可能な場合は、値 **+DD32** が入ります。HP-UX プラットフォームが PA-RISC で、64 ビット・サポートが使用可能な場合は、これには値 **+DA2.0W** が入ります。PA-RISC プラットフォームでの 32 ビット・サポートの場合は、このフラグには値 **+DA2.0N** が入ります。

+DD64 IA64 版の HP-UX 用の 64 ビット・コードを生成する場合に使用する必要があります。

+DD32 IA64 版の HP-UX 用の 32 ビット・コードを生成する場合に使用する必要があります。

+DA2.0W

PA-RISC 版の HP-UX 用の 64 ビット・コードを生成する場合に使用する必要があります。

+DA2.0N

PA-RISC 版の HP-UX 用の 32 ビット・コードを生成する場合に使用する必要があります。

-mt ルーチンを他のルーチンと同じプロセスで実行する (THREADSAFE) か、またはエンジンそのもので実行する (NOT FENCED) かに応じて、HP aC++ コンパイラーのマルチスレッド・サポートを使用可能にします。

-b 通常の実行可能ファイルではなく、共用ライブラリーを作成します。

-o \$1 実行可能ファイルを指定します。

\$1.o プログラム・オブジェクト・ファイルを指定します。

\$EXTRA_LFLAG

ランタイム・パスを指定します。設定する場合、32 ビットの場合は値 **-Wl,+b\$HOME/sql1lib/lib32**、64 ビットの場合は **-Wl,+b\$HOME/sql1lib/lib64** が入ります。設定しない場合は、これには値が入りません。

-L\$DB2PATH/\$LIB

DB2 ランタイム共用ライブラリーのロケーションを指定します。32 ビットの場合は **"\$HOME/sql1lib/lib32"**、64 ビットの場合は **"\$HOME/sql1lib/lib64"** です。

-ldb2 DB2 ライブラリーとリンクします。

他のコンパイラー・オプションについては、コンパイラーの資料をご覧ください。

Linux C ルーチンのコンパイルとリンクのオプション

以下は、`bldrtn` ビルド・スクリプトに示されているように、Linux C コンパイラーを使用して、C ルーチン (ストアード・プロシージャとユーザー定義関数) を構築するのにお勧めするコンパイルとリンクのオプションです。

bldrtn のコンパイルとリンクのオプション

コンパイル・オプション:

\$CC gcc または xlc_r コンパイラー

\$EXTRA_C_FLAGS

次のうちの 1 つが含まれています。

- Linux for zSeries® の場合のみ -m31 を指定して、32 ビット・ライブラリーを作成します。
- Linux for x86, x64 and POWER® の場合には -m32 を指定して、32 ビット・ライブラリーを作成します。
- Linux for zSeries, POWER, x64 の場合には -m64 を指定して、64 ビット・ライブラリーを作成します。
- Linux for IA64 の場合には値を指定しないで、64 ビット・ライブラリーを作成します。

-\$DB2PATH/include

DB2 組み込みファイルのロケーションを指定します。

-c コンパイルのみを実行し、リンクは実行しません。このスクリプト・ファイルでは、コンパイルとリンクは別個のステップです。

-D_REENTRANT

ルーチンを他のルーチンと同じプロセスで実行する (THREADSAFE) か、またはエンジンそのもので実行する (NOT FENCED) かに応じて、_REENTRANT を定義します。

リンク・オプション:

\$CC gcc または xlc_r コンパイラー。コンパイラーをリンカーのフロントエンドとして使用します。

\$LINK_FLAGS

値 "\$EXTRA_C_FLAGS \$SHARED_LIB_FLAG" を入れます。

\$EXTRA_C_FLAGS

次のうちの 1 つが含まれています。

- Linux for zSeries の場合のみ -m31 を指定して、32 ビット・ライブラリーを作成します。
- Linux for x86, x64 and POWER の場合には -m32 を指定して、32 ビット・ライブラリーを作成します。
- Linux for zSeries, POWER, x64 の場合には -m64 を指定して、64 ビット・ライブラリーを作成します。
- Linux for IA64 の場合には値を指定しないで、64 ビット・ライブラリーを作成します。

\$SHARED_LIB_FLAG

gcc コンパイラーの場合には -shared、xlc_r コンパイラーの場合には -qmksrobj を入れます。

-o \$1 実行可能ファイルを指定します。

\$1.o プログラム・オブジェクト・ファイルを組み込みます。

\$EXTRA_LFLAG

実行時の DB2 共用ライブラリーのロケーションを示します。32 ビットの場合は値「-Wl,-rpath,\$DB2PATH/lib32」が入ります。64 ビットの場合は値「-Wl,-rpath,\$DB2PATH/lib64」が入ります。

-L\$DB2PATH/\$LIB

リンク時の DB2 静的ライブラリーおよび共用ライブラリーのロケーションを示します。例えば、32 ビットの場合は \$HOME/sql1lib/lib32、64 ビットの場合は \$HOME/sql1lib/lib64 です。

-ldb2 DB2 ライブラリーとリンクします。

-lpthread

POSIX スレッド・ライブラリーとリンクします。

他のコンパイラー・オプションについては、コンパイラーの資料をご覧ください。

Linux C++ ルーチンのコンパイルとリンクのオプション

以下は、bldrtn ビルド・スクリプトに示されているように、Linux C++ コンパイラーを使用して、C++ ルーチン (ストアード・プロシージャとユーザー定義関数) を構築するのにお勧めするコンパイルとリンクのオプションです。

bldrtn のコンパイルとリンクのオプション

コンパイル・オプション:

g++ GNU/Linux C++ コンパイラ。

\$EXTRA_C_FLAGS

次のうちの 1 つが含まれています。

- Linux for zSeries の場合のみ **-m31** を指定して、32 ビット・ライブラリーを作成します。
- Linux for x86, x64 and POWER の場合には **-m32** を指定して、32 ビット・ライブラリーを作成します。
- Linux for zSeries, POWER, x64 の場合には **-m64** を指定して、64 ビット・ライブラリーを作成します。
- Linux for IA64 の場合には値を指定しないで、64 ビット・ライブラリーを作成します。

-fpic 位置に依存しないコードを生成します。

-I\$DB2PATH/include

DB2 組み込みファイルのロケーションを指定します。

-c コンパイルのみを実行し、リンクは実行しません。このスクリプト・ファイルでは、コンパイルとリンクは別個のステップです。

-D_REENTRANT

ルーチンを他のルーチンと同じプロセスで実行する (THREADSAFE) か、またはエンジンそのもので実行する (NOT FENCED) かに応じて、**_REENTRANT** を定義します。

リンク・オプション:

g++ コンパイラーをリンカーのフロントエンドとして使用します。

\$EXTRA_C_FLAGS

次のうちの 1 つが含まれています。

- Linux for zSeries の場合のみ **-m31** を指定して、32 ビット・ライブラリーを作成します。
- Linux for x86, x64 and POWER の場合には **-m32** を指定して、32 ビット・ライブラリーを作成します。
- Linux for zSeries, POWER, x64 の場合には **-m64** を指定して、64 ビット・ライブラリーを作成します。
- Linux for IA64 の場合には値を指定しないで、64 ビット・ライブラリーを作成します。

-shared

共用ライブラリーを生成します。

-o \$1 実行可能ファイルを指定します。

\$1.o プログラム・オブジェクト・ファイルを組み込みます。

\$EXTRA_LFLAG

実行時の DB2 共用ライブラリーのロケーションを示します。32 ビットの場合は値「**-Wl,-rpath,\$DB2PATH/lib32**」が入ります。64 ビットの場合は値「**-Wl,-rpath,\$DB2PATH/lib64**」が入ります。

-L\$DB2PATH/\$LIB

リンク時の DB2 静的ライブラリーおよび共用ライブラリーのロケーションを示します。例えば、32 ビットの場合は **\$HOME/sql11ib/lib32**、64 ビットの場合は **\$HOME/sql11ib/lib64** です。

-ldb2 DB2 ライブラリーとリンクします。

-lpthread

POSIX スレッド・ライブラリーとリンクします。

他のコンパイラー・オプションについては、コンパイラーの資料をご覧ください。

Solaris C ルーチンのコンパイルとリンクのオプション

以下は、**bldrtn** ビルド・スクリプトに示されているように、Forte C コンパイラーを使用して、C ルーチン (ストアード・プロシージャーとユーザー定義関数) を構築するのにお勧めするコンパイルとリンクのオプションです。

bldrtn のコンパイルとリンクのオプション

コンパイル・オプション:

cc C コンパイラー。

-xarch=\$CFLAG_ARCH

このオプションを使用すると、libdb2.so へのリンク時に必ず正しい実行可能ファイルがコンパイラーで生成されるようにすることができます。\$CFLAG_ARCH の値は、次のように設定されます。

- Solaris SPARC の 32 ビット・アプリケーションの場合は「v8plusa」
- Solaris SPARC の 64 ビット・アプリケーションの場合は「v9」
- Solaris x64 の 32 ビット・アプリケーションの場合は「sse2」
- Solaris x64 の 64 ビット・アプリケーションの場合は「amd64」

-mt ルーチンを他のルーチンと同じプロセスで実行する (THREADSAFE) か、またはエンジンそのもので実行する (NOT FENCED) かに応じて、マルチスレッド・サポートを使用可能にします。

-DUSE_UI_THREADS

Sun の「UNIX International」スレッド API を使用可能にします。

-Kpic 共用ライブラリー用の位置に依存しないコードを生成します。

-I\$DB2PATH/include

DB2 組み込みファイルのロケーションを指定します。

-c コンパイルのみを実行し、リンクは実行しません。このスクリプトでは、コンパイルとリンクは別個のステップです。

リンク・オプション:

cc コンパイラーをリンカーのフロントエンドとして使用します。

-xarch=\$CFLAG_ARCH

このオプションを使用すると、libdb2.so へのリンク時に必ず正しい実行可能ファイルがコンパイラーで生成されるようにすることができます。\$CFLAG_ARCH の値は、32 ビットの場合は v8plusa に、64 ビットの場合は v9 に設定されます。

-mt これが必要なのは、DB2 ライブラリーは -mt にリンクされているからです。

-G 共用ライブラリーを生成します。

-o \$1 実行可能ファイルを指定します。

\$1.o プログラム・オブジェクト・ファイルを組み込みます。

-L\$DB2PATH/\$LIB

リンク時の DB2 静的ライブラリーおよび共用ライブラリーのロケーションを示します。例えば、32 ビットの場合は \$HOME/sql11ib/lib32、64 ビットの場合は \$HOME/sql11ib/lib64 です。

\$EXTRA_LFLAG

実行時の DB2 共用ライブラリーのロケーションを示します。32 ビットの場合は値 "-R\$DB2PATH/lib32" が入り、64 ビットの場合は値 "-R\$DB2PATH/lib64" が入ります。

-ldb2 DB2 ライブラリーとリンクします。

他のコンパイラー・オプションについては、コンパイラーの資料をご覧ください。

Solaris C++ ルーチンのコンパイルとリンクのオプション

以下は、bldrtn ビルド・スクリプトに示されているように、Forte C++ コンパイラーを使用して、C++ ルーチン (ストアド・プロシージャとユーザー定義関数) を構築するのにお勧めするコンパイルとリンクのオプションです。

bldrtn のコンパイルとリンクのオプション

コンパイル・オプション:

CC C++ コンパイラー。

-xarch=\$CFLAG_ARCH

このオプションを使用すると、libdb2.so へのリンク時に必ず正しい実行可能ファイルがコンパイラーで生成されるようにすることができます。\$CFLAG_ARCH の値は、次のように設定されます。

- Solaris SPARC の 32 ビット・アプリケーションの場合は「v8plusa」
- Solaris SPARC の 64 ビット・アプリケーションの場合は「v9」
- Solaris x64 の 32 ビット・アプリケーションの場合は「sse2」
- Solaris x64 の 64 ビット・アプリケーションの場合は「amd64」

-mt ルーチンを他のルーチンと同じプロセスで実行する (THREADSAFE) か、またはエンジンそのもので実行する (NOT FENCED) かに応じて、マルチスレッド・サポートを使用可能にします。

-DUSE_UI_THREADS

Sun の「UNIX International」スレッド API を使用可能にします。

-Kpic 共用ライブラリー用の位置に依存しないコードを生成します。

-I\$DB2PATH/include

DB2 組み込みファイルのロケーションを指定します。

-c コンパイルのみを実行し、リンクは実行しません。このスクリプトでは、コンパイルとリンクは別個のステップです。

リンク・オプション:

CC コンパイラーをリンカーのフロントエンドとして使用します。

-xarch=\$CFLAG_ARCH

このオプションを使用すると、libdb2.so へのリンク時に必ず正しい実行可能ファイルがコンパイラーで生成されるようにすることができます。\$CFLAG_ARCH の値は、32 ビットの場合は v8plusa に、64 ビットの場合は v9 に設定されます。

-mt これが必要なのは、DB2 ライブラリーは -mt にリンクされているからです。

-G 共用ライブラリーを生成します。

-o \$1 実行可能ファイルを指定します。

\$1.o プログラム・オブジェクト・ファイルを組み込みます。

-L\$DB2PATH/\$LIB

リンク時の DB2 静的ライブラリーおよび共用ライブラリーのロケーションを示します。例えば、32 ビットの場合は \$HOME/sql1lib/lib32、64 ビットの場合は \$HOME/sql1lib/lib64 です。

\$EXTRA_LFLAG

実行時の DB2 共用ライブラリーのロケーションを示します。32 ビットの場合は値 "-R\$DB2PATH/lib32" が入り、64 ビットの場合は値 "-R\$DB2PATH/lib64" が入ります。

-ldb2 DB2 ライブラリーとリンクします。

他のコンパイラー・オプションについては、コンパイラーの資料をご覧ください。

Windows C および C++ ルーチンのコンパイルとリンクのオプション

以下は、bldrtn.bat バッチ・ファイルに示されているように、Windows 上で Microsoft Visual C++ コンパイラーを使用して、C および C++ ルーチン (ストアード・プロシージャとユーザー定義関数) を構築するのにお勧めするコンパイルとリンクのオプションです。

bldrtn のコンパイルとリンクのオプション

コンパイル・オプション:

%BLDCOMP%

コンパイラー用の変数です。デフォルトは、c1 (Microsoft Visual C++ コンパイラー) です。またこれは、ic1 (32 ビットおよび 64 ビット・アプリケーション用の Intel® C++ コンパイラー)、または ec1 (Itanium® 64 ビット・アプリケーション用の Intel C++ コンパイラー) に設定することもできます。

-Zi デバッグ情報を使用可能にします。

-Od 最適化なし。

-c コンパイルのみを実行し、リンクは実行しません。コンパイルとリンクは別個のステップです。

-W2 警告、エラー、重大、およびリカバリー不能エラー・メッセージを出力します。

-DWIN32

Windows オペレーティング・システムに必要なコンパイラー・オプション。

-MD MSVCRT.LIB を使用するリンク。

リンク・オプション:

link リンクにリンカーを使用します。

-debug デバッグ情報を組み込みます。

-out:%1.dll

.DLL ファイルをビルドします。

%1.obj オブジェクト・ファイルを組み込みます。

db2api.lib

DB2 ライブラリーとリンクします。

-def:%1.def

モジュール定義ファイル。

他のコンパイラー・オプションについては、コンパイラーの資料をご覧ください。

構成ファイルを使用した C または C++ の組み込み SQL ストアド・プロシージャの構築

AIX 上では、sql1lib/samples/c および sql1lib/samples/cpp にある構成ファイル stp.icc を使用することによって、DB2 組み込み SQL ストアド・プロシージャを C および C++ で作成できます。

構成ファイルを使用して、ソース・ファイル spserver.sqc から組み込み SQL ストアド・プロシージャの共用ライブラリー spserver を構築するには、以下のようになります。

1. 次のように入力して、STP 環境変数をプログラム名に設定します。

- bash または Korn シェルの場合

```
export STP=spserver
```

- C シェルの場合

```
setenv STP spserver
```

2. `stp.icc` ファイルを使用して異なるプログラムを構築することによって生成された `stp.ics` ファイルが作業ディレクトリーにある場合は、次のコマンドで `stp.ics` ファイルを削除してください。

```
rm stp.ics
```

既存の `stp.ics` ファイルが、再構築するその同じプログラム用に生成されているのであれば、削除する必要はありません。

3. サンプル・プログラムを以下のように入力してコンパイルします。

```
vacbld stp.icc
```

注: `vacbld` コマンドは、 VisualAge C++ で提供されます。

このストアード・プロシージャの共用ライブラリーは、パス `sqllib/function` のサーバーにコピーされます。

次に、次のようにサーバーで `spcat` スクリプトを実行して、共用ライブラリーのストアード・プロシージャをカタログします。

```
spcat
```

このスクリプトは、サンプル・データベースに接続し、ストアード・プロシージャが既にカタログ済みであれば `spdrop.db2` を呼び出してそれをアンカタログし、次に `spcreate.db2` を呼び出してそのストアード・プロシージャをカタログし、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、個別に呼び出すこともできます。

次に、データベースを一度停止してから再始動し、新しい共用ライブラリーが認識されるようにします。必要であれば、共用ライブラリーにファイル・モードを設定して、DB2 インスタンスからアクセスできるようにします。

ストアード・プロシージャの共用ライブラリー `spserver` を作成し終わったら、その中のストアード・プロシージャを呼び出すクライアント・アプリケーション `spclient` を構築することができます。 `spclient` は、構成ファイル `emb.icc` を使用して構築することができます。

ストアード・プロシージャを呼び出すためには、次のように入力してサンプル・クライアント・アプリケーションを実行します。 `spclient database userid password`

ここで、

database

接続先のデータベースの名前です。 名前は、`sample` またはそのリモート別名、あるいはその他の名前にすることができます。

userid 有効なユーザー ID です。

password

有効なパスワードです。

クライアント・アプリケーションは共用ライブラリー `spserver` にアクセスし、さまざまなストアード・プロシージャ関数をサーバー・データベース上で実行します。出力は、クライアント・アプリケーションに戻されます。

構成ファイルを使用した C または C++ のユーザー定義関数の構築 (AIX)

sqllib/samples/c と sqllib/samples/cpp の中の構成ファイル udf.icc を使用すれば、AIX 上で C および C++ のユーザー定義関数を構築することができます。

構成ファイルを使用して、ソース・ファイル udfsrv.c からユーザー定義関数プログラム udfsrv を構築するには、以下のようにします。

1. 次のように入力して、UDF 環境変数をプログラム名に設定します。

- bash または Korn シェルの場合

```
export UDF=udfsrv
```

- C シェルの場合

```
setenv UDF udfsrv
```

2. udf.icc ファイルを使用して異なるプログラムを構築することによって生成された udf.ics ファイルが作業ディレクトリーにある場合は、次のコマンドで udf.ics ファイルを削除してください。

```
rm udf.ics
```

既存の udf.ics ファイルが、再構築するその同じプログラム用に生成されているのであれば、削除する必要はありません。

3. サンプル・プログラムを以下のように入力してコンパイルします。

```
vacbld udf.icc
```

注: vacbld コマンドは、VisualAge C++ で提供されます。

UDF ライブラリーは、サーバー上の sqllib/function というパスにコピーされます。

必要であれば、ユーザー定義関数にファイル・モードを設定して DB2 インスタンスがそれを実行できるようにしてください。

udfsrv の構築が完了したなら、それを呼び出すクライアント・アプリケーション udfcli を構築できます。このプログラムの DB2 CLI および組み込み SQL バージョンが提供されています。

構成ファイル cli.icc を使用して、sqllib/samples/cli 内のソース・ファイル udfcli.c から DB2 CLI udfcli プログラムを構築することができます。

構成ファイル emb.icc を使用して、sqllib/samples/c 内のソース・ファイル udfcli.sqc から組み込み SQL udfcli プログラムを構築することができます。

UDF を呼び出すには、次の実行可能ファイル名を入力して、サンプルの呼び出しアプリケーションを実行します。

```
udfcli
```

この呼び出しアプリケーションは、udfsrv ライブラリーから ScalarUDF 関数を呼び出します。

DB2 ルーチンの共用ライブラリーの再構築

DB2 は、ストアド・プロシージャとユーザー定義関数に使用される共用ライブラリーがロードされると、それをキャッシュに入れます。ルーチンを開発する場合、同じ共用ライブラリーのロードを何度もテストする場合があります。そのキャッシングの際に、最新バージョンの共用ライブラリーを取得できないこともあります。キャッシングに関連する問題を回避するには、ルーチンのタイプによって方法が異なります。

1. **fenced** された、スレッド・セーフではないルーチン。 データベース・マネージャー構成キーワード `KEEPFENCED` のデフォルト値は、`YES` です。そのため、`fenced` モード・プロセスは存続し続けます。このデフォルト設定によって、ライブラリーの再ロードが妨げられることがあります。 `fenced` された、スレッド・セーフではないルーチンの開発中はこのキーワードの値を `NO` に変更しておいて、最終バージョンの共用ライブラリーをロードする準備ができたなら、この値を `YES` に戻すのが最善の方法です。詳しくは、『データベース・マネージャー構成ファイルの更新』を参照してください。
2. 信頼できる、またはスレッド・セーフのルーチン。 SQL ルーチン (SQL プロシージャを含む) 以外で、DB2 ルーチン・ライブラリーが信頼できるルーチンまたはスレッド・セーフ・ルーチンに使用される場合、更新されたバージョンが確実に選び出されるようにする唯一の方法は、コマンド行で `db2stop` の後に `db2start` と入力して、DB2 インスタンスをリサイクルすることです。SQL ルーチンではこれは必要ありません。SQL ルーチンが再作成される際には、コンパイラーは新しいユニークなライブラリー名を使用して、競合の可能性を回避するからです。

また SQL ルーチン以外のルーチンの場合、ライブラリーに別の名前 (例えば、`foo.a` を `foo.1.a` にする) を付けて新規バージョンのルーチンを作成し、その新しいライブラリーで `ALTER PROCEDURE` または `ALTER FUNCTION SQL` ステートメントのいずれかを使用すると、キャッシングの問題を避けることができます。

データベース・マネージャー構成ファイルの更新

このファイルには、アプリケーション開発のための重要な設定が収められています。

キーワード `KEEPFENCED` のデフォルト値は `YES` です。 `fenced` された、スレッド・セーフではないルーチン (ストアド・プロシージャと UDF) の場合、ルーチン・プロセスは存続し続けます。こうしたルーチンの開発中はこのキーワードの値を `NO` に変更しておいて、最終バージョンの共用ライブラリーをロードする準備ができたなら、この値を `YES` に戻すのが最善の方法です。詳しくは、『DB2 ルーチンの共用ライブラリーの再構築』を参照してください。

注: `KEEPFENCED` は、旧バージョンの DB2 では `KEEPDARI` という名称でした。

Java アプリケーション開発の場合、Java Development Kit のインストール先のパスを使用して `JDK_PATH` キーワードを更新する必要があります。

注: `JDK_PATH` は、旧バージョンの DB2 では `JDK11_PATH` という名称でした。

この設定を変更するには、次のように入力します。


```
db2 update dbm cfg using <keyword> <value>
```

例えば、キーワード `KEEPFENCED` を `NO` に設定するには、次のようにします。

```
db2 update dbm cfg using KEEPFENCED NO
```

`JDK_PATH` キーワードをディレクトリー `/home/db2inst/jdk13` に設定するには、次のようにします。

```
db2 update dbm cfg using JDK_PATH /home/db2inst/jdk13
```

データベース・マネージャー構成ファイル内の現在の設定を表示するには、以下を入力します。

```
db2 get dbm cfg
```

注: Windows では、このコマンドを DB2 コマンド・ウィンドウに入力する必要があります。

第 6 章 COBOL プロシージャ

COBOL プロシージャは、COBOL サブプログラムと似た方法で書き込まれることになっています。

COBOL プロシージャ内でのパラメーターの処理

プロシージャによって受け入れられる、または渡される各パラメーターは、LINKAGE SECTION 内で宣言されていなければなりません。例えば、このコード断片は 2 つの IN パラメーター (CHAR(15) と INT) を受け取り、OUT パラメーター (INT) を渡すプロシージャから来ています。

```
LINKAGE SECTION.  
01 IN-SPERSON PIC X(15).  
01 IN-SQTY PIC S9(9) USAGE COMP-5.  
01 OUT-SALESSUM PIC S9(9) USAGE COMP-5.
```

宣言した COBOL データ・タイプが正しく SQL データ・タイプにマップされるようにします。SQL と COBOL の間のデータ・タイプ・マッピングの詳細なリストについては、『COBOL でサポートされる SQL データ・タイプ』を参照してください。

次いで、各パラメーターを PROCEDURE DIVISION にリストしなければなりません。以下に、前の LINKAGE SECTION の例にあるパラメーター定義に対応する PROCEDURE DIVISION の例を示します。

```
PROCEDURE DIVISION USING IN-SPERSON  
IN-SQTY  
OUT-SALESSUM.
```

COBOL プロシージャの終了

プロシージャを適切に終了するには、次のコマンドを使用します。

```
MOVE SQLZ-HOLD-PROC TO RETURN-CODE.  
GOBACK.
```

これらのコマンドを使用して、プロシージャはクライアント・アプリケーションに正しく戻ります。プロシージャがローカルの COBOL クライアント・アプリケーションによって呼び出された場合に、これは特に重要です。

COBOL プロシージャを構築する場合、使用するオペレーティング・システムおよびコンパイラ用に書かれたビルド・スクリプトを使用するよう強くお勧めします。Micro Focus COBOL 用のビルド・スクリプトは、sqllib/samples/cobol_mf ディレクトリにあります。IBM COBOL 用のビルド・スクリプトは、sqllib/samples/cobol ディレクトリにあります。

以下に、2 つの入力パラメーターを受け入れて、出力パラメーターと結果セットを戻す COBOL のプロシージャの例を示します。

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. "NEWSALE".  
DATA DIVISION.
```

```
WORKING-STORAGE SECTION.
```

```

01 INSERT-STMT.
   05 FILLER PIC X(24) VALUE "INSERT INTO SALES (SALES".
   05 FILLER PIC X(24) VALUE "_PERSON,SALES) VALUES ('".
   05 SPERSON PIC X(16).
   05 FILLER PIC X(2) VALUE "','".
   05 SQTY PIC S9(9).
   05 FILLER PIC X(1) VALUE ")".
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 INS-SMT-INF.
   05 INS-STMT.
   49 INS-LEN PIC S9(4) USAGE COMP.
   49 INS-TEXT PIC X(100).
01 SALESSUM PIC S9(9) USAGE COMP-5.
EXEC SQL END DECLARE SECTION END-EXEC.
EXEC SQL INCLUDE SQLCA END-EXEC.

LINKAGE SECTION.
01 IN-SPERSON PIC X(15).
01 IN-SQTY PIC S9(9) USAGE COMP-5.
01 OUT-SALESSUM PIC S9(9) USAGE COMP-5.

PROCEDURE DIVISION USING IN-SPERSON
                        IN-SQTY
                        OUT-SALESSUM.

MAINLINE.
  MOVE 0 TO SQLCODE.
  PERFORM INSERT-ROW.
  IF SQLCODE IS NOT EQUAL TO 0
    GOBACK
  END-IF.
  PERFORM SELECT-ROWS.
  PERFORM GET-SUM.
  GOBACK.
INSERT-ROW.
  MOVE IN-SPERSON TO SPERSON.
  MOVE IN-SQTY TO SQTY.
  MOVE          INSERT-STMT TO INS-TEXT.
  MOVE LENGTH OF INSERT-STMT TO INS-LEN.
  EXEC SQL EXECUTE IMMEDIATE :INS-STMT END-EXEC.
GET-SUM.
  EXEC SQL
    SELECT SUM(SALES) INTO :SALESSUM FROM SALES
  END-EXEC.
  MOVE SALESSUM TO OUT-SALESSUM.
SELECT-ROWS.
  EXEC SQL
    DECLARE CUR CURSOR WITH RETURN FOR SELECT * FROM SALES
  END-EXEC.
  IF SQLCODE = 0
    EXEC SQL OPEN CUR END-EXEC
  END-IF.

```

このプロシージャの対応する CREATE PROCEDURE ステートメントは次のとおりです。

```

CREATE PROCEDURE NEWSALE ( IN SALESPERSON CHAR(15),
                          IN SALESQTY INT,
                          OUT SALESSUM INT)

RESULT SETS 1
EXTERNAL NAME 'NEWSALE!NEWSALE'
FENCED
LANGUAGE COBOL
PARAMETER STYLE SQL
MODIFIES SQL DATA

```

上記のステートメントでは、COBOL 関数は NEWSALE というライブラリー内に
ることが前提になっています。

注: COBOL プロシージャを Windows オペレーティング・システムで登録する場
合、CREATE ステートメントの EXTERNAL NAME 節でストアード・プロシージ
ャー本体を識別するときは以下のように気を付けてください。プロシージャ本体
を識別するのに絶対パス ID を使用する場合、.d11 拡張子を付加する必要があり
ます。以下に例を示します。

```
CREATE PROCEDURE NEWSALE ( IN SALESPERSON CHAR(15),
                           IN SALESQTY INT,
                           OUT SALESSUM INT)
  RESULT SETS 1
  EXTERNAL NAME 'NEWSALE!NEWSALE'
  FENCED
  LANGUAGE COBOL
  PARAMETER STYLE SQL
  MODIFIES SQL DATA
  EXTERNAL NAME 'd:\mylib\NEWSALE.d11'
```

COBOL での外部プロシージャ開発のサポート

COBOL で外部プロシージャを開発するには、サポートされる COBOL 開発ソフ
トウェアを使用する必要があります。

COBOL でのデータベース・アプリケーション開発の場合にサポートされている開
発ソフトウェアはすべて、COBOL での外部プロシージャの開発にも使用できま
す。

COBOL 組み込み SQL アプリケーションでサポートされている SQL デー タ・タイプ

特定の事前定義 COBOL データ・タイプは、DB2 データベースの列タイプに対応し
ています。ホスト変数として宣言できるのは、その種の COBOL データ・タイプだ
けです。

以下の表に、各列タイプに対応する COBOL データ・タイプを示します。プリコン
パイラーはホスト変数宣言を検出すると、該当する SQL タイプの値を判別しま
す。データベース・マネージャーはこの値を使用して、アプリケーションとの間で
やり取りするデータを変換します。

ホスト変数のすべての有効なデータ記述が認識されるわけではありません。
COBOL データ項目は、以下の表に示す項目と一致していなければなりません。別
のデータ項目を使用すると、エラーになる場合があります。

表 41. COBOL 宣言にマップされる SQL データ・タイプ

SQL 列タイプ ¹	COBOL データ・タイプ	SQL 列タイプ記述
SMALLINT (500 または 501)	01 name PIC S9(4) COMP-5	16 ビットの符号付き整数

表 41. COBOL 宣言にマップされる SQL データ・タイプ (続き)

SQL 列タイプ ¹	COBOL データ・タイプ	SQL 列タイプ記述
INTEGER (496 または 497)	01 name PIC S9(9) COMP-5	32 ビットの符号付き整数
BIGINT (492 または 493)	01 name PIC S9(18) COMP-5	64 ビットの符号付き整数
DECIMAL(<i>p,s</i>) (484 または 485)	01 name PIC S9(<i>m</i>)V9(<i>n</i>) COMP-3	パック 10 進数
REAL ² (480 または 481)	01 name USAGE IS COMP-1	単精度浮動小数点
DOUBLE ³ (480 または 481)	01 name USAGE IS COMP-2	倍精度浮動小数点
CHAR(<i>n</i>) (452 または 453)	01 name PIC X(<i>n</i>)	固定長文字ストリング
VARCHAR(<i>n</i>) (448 または 449)	01 name 49 length PIC S9(4) COMP-5 49 name PIC X(<i>n</i>) 1<= <i>n</i> <=32 672	可変長文字ストリング
LONG VARCHAR ⁶ (456 または 457)	01 name 49 length PIC S9(4) COMP-5 49 data PIC X(<i>n</i>) 32 673<= <i>n</i> <=32 700	long 可変長文字ストリング
CLOB(<i>n</i>) (408 または 409)	01 MY-CLOB USAGE IS SQL TYPE IS CLOB(<i>n</i>) 1<= <i>n</i> <=2 147 483 647	ラージ・オブジェクト可変長文字ストリング
CLOB ロケーター変数 ⁴ (964 または 965)	01 MY-CLOB-LOCATOR USAGE IS SQL TYPE IS CLOB-LOCATOR	サーバー上の CLOB エンティティを識別する
CLOB ファイル参照変数 ⁴ (920 または 921)	01 MY-CLOB-FILE USAGE IS SQL TYPE IS CLOB-FILE	CLOB データを含むファイルの記述子
BLOB(<i>n</i>) (404 または 405)	01 MY-BLOB USAGE IS SQL TYPE IS BLOB(<i>n</i>) 1<= <i>n</i> <=2 147 483 647	ラージ・オブジェクト可変長バイナリー・ストリング
BLOB ロケーター変数 ⁴ (960 または 961)	01 MY-BLOB-LOCATOR USAGE IS SQL TYPE IS BLOB-LOCATOR	サーバー上の BLOB エンティティを識別する

表 41. COBOL 宣言にマップされる SQL データ・タイプ (続き)

SQL 列タイプ ¹	COBOL データ・タイプ	SQL 列タイプ記述
BLOB ファイル参照変数 ⁴ (916 または 917)	01 MY-BLOB-FILE USAGE IS SQL TYPE IS BLOB-FILE.	BLOB データを含むファイルの記述子
DATE (384 または 385)	01 identifier PIC X(10)	10 バイトの文字ストリング
TIME (388 または 389)	01 identifier PIC X(8)	8 バイトの文字ストリング
TIMESTAMP(<i>p</i>) (392 または 393)	01 identifier PIC X(<i>p</i> +20) 0<= <i>p</i> <=12	19 から 32 バイトの文字ストリング <i>p</i> が 0 の場合、19 バイトの文字ストリングを使用できます。
XML ⁵ (988 または 989)	01 name USAGE IS SQL TYPE IS XML AS CLOB (size).	XML 値

以下のデータ・タイプは、DBCS 環境でのみ使用可能です。

表 42. COBOL 宣言にマップされる SQL データ・タイプ

SQL 列タイプ ¹	COBOL データ・タイプ	SQL 列タイプ記述
GRAPHIC(<i>n</i>) (468 または 469)	01 name PIC G(<i>n</i>) DISPLAY-1	固定長 2 バイト文字ストリング
VARGRAPHIC(<i>n</i>) (464 または 465)	01 name 49 length PIC S9(4) COMP-5 49 name PIC G(<i>n</i>) DISPLAY-1 1<= <i>n</i> <=16 336	2 バイトのストリング長標識を持つ、可変長 2 バイト文字ストリング
LONG VARGRAPHIC ⁶ (472 または 473)	01 name 49 length PIC S9(4) COMP-5 49 name PIC G(<i>n</i>) DISPLAY-1 16 337<= <i>n</i> <=16 350	2 バイトのストリング長標識を持つ、可変長 2 バイト文字ストリング
DBCLOB(<i>n</i>) (412 または 413)	01 MY-DBCLOB USAGE IS SQL TYPE IS DBCLOB(<i>n</i>) 1<= <i>n</i> <=1 073 741 823	4 バイトのストリング長標識を持つ、ラージ・オブジェクト可変長 2 バイト文字ストリング
DBCLOB ロケータ変数 ⁴ (968 または 969)	01 MY-DBCLOB-LOCATOR USAGE IS SQL TYPE IS DBCLOB-LOCATOR	サーバー上の DBCLOB エンティティーを識別する

表 42. COBOL 宣言にマップされる SQL データ・タイプ (続き)

SQL 列タイプ ¹	COBOL データ・タイプ	SQL 列タイプ記述
DBCLOB ファイル参照変数 ⁴ (924 または 925)	01 MY-DBCLOB-FILE USAGE IS SQL TYPE IS DBCLOB-FILE	DBCLOB データを含むファイルの記述子

注:

1. **SQL 列タイプ**の欄にある最初の番号は、標識変数が提供されていないこと、2 番目の番号は標識変数が提供されていることを示します。標識変数は、NULL 値を示したり、切り捨てられたストリングの長さを保持するのに必要です。これらの値は、それぞれのデータ・タイプの SQLDA の SQLTYPE フィールドに表示されます。
2. **FLOAT(n)**。ここで $0 < n < 25$ の場合、REAL と同義。SQLDA での REAL と DOUBLE の違いは長さの値です (4 または 8)。
3. 以下の SQL タイプは、DOUBLE と同義です。
 - FLOAT
 - **FLOAT(n)** ($24 < n < 54$) は DOUBLE と同義。
 - DOUBLE PRECISION
4. これは列タイプではなく、ホスト変数タイプである。
5. **SQL_TYP_XML/SQL_TYP_NXML** 値は、DESCRIBE 要求によってのみ戻されます。アプリケーション・リソースを XML 値にバインドするために、アプリケーションが直接この値を使用することはできません。
6. **LONG VARCHAR** および **LONG VARGRAPHIC** データ・タイプは推奨されておらず、将来のリリースで除去される可能性があります。代わりに CLOB または DBCLOB データ・タイプを選択します。

サポートされる COBOL データ・タイプについては、さらに次の規則があります。

- **PIC S9** と **COMP-3/COMP-5** が明示されている場合、これらは必須です。
- **VARCHAR**、**LONG VARCHAR**、**VARGRAPHIC**、**LONG VARGRAPHIC**、すべての **LOB** 変数タイプ以外の列タイプについては、レベル番号として 01 の代わりに 77 を使用できます。
- **DECIMAL(p,s)** 列タイプのホスト変数を宣言する際には、以下の規則を使用します。以下のサンプルを参照してください。


```
01 identifier PIC S9(m)V9(n) COMP-3
```

 - 小数点の表記に **V** を使用します。
 - n と m の値は 1 以上でなければなりません。
 - $n + m$ の値は 31 以下でなければなりません。
 - s の値は n の値と等しくなります。
 - p の値は $n + m$ の値と等しくなります。
 - 反復因数 (n) と (m) はオプションです。以下の例はすべて有効です。


```
01 identifier PIC S9(3)V COMP-3
01 identifier PIC SV9(3) COMP-3
01 identifier PIC S9V COMP-3
01 identifier PIC SV9 COMP-3
```
 - **COMP-3** の代わりに **PACKED-DECIMAL** を使用できます。
- 配列は、COBOL プリコンパイラーではサポートされていません。

COBOL ルーチンのビルド

COBOL ルーチンのコンパイルとリンクのオプション

AIX IBM COBOL ルーチンのコンパイルとリンクのオプション

以下は、bldrtn ビルド・スクリプトに示されているように、AIX 上で IBM COBOL for AIX コンパイラーを使用して、COBOL ルーチン (ストアード・プロシージャ) を構築するのにお勧めするコンパイルとリンクのオプションです。

bldrtn のコンパイルとリンクのオプション

コンパイル・オプション:

cob2 IBM COBOL for AIX コンパイラー。

-qpgmname¥(mixed¥)

コンパイラーに、大/小文字混合の名前を持つライブラリーのエンタリー・ポイントの CALL を許可するように指示します。

-qlib コンパイラーに COPY ステートメントを処理するように指示します。

-c コンパイルのみを実行し、リンクは実行しません。コンパイルとリンクは別個のステップです。

-I\$DB2PATH/include/cobol_a

DB2 組み込みファイルのロケーションを指定します。例えば、
\$HOME/sql1lib/include/cobol_a。

リンク・オプション:

cob2 コンパイラーを使用してリンク・エディットします。

-o \$1 出力を、共用ライブラリー・ファイルとして指定します。

\$1.o ストアード・プロシージャ・オブジェクト・ファイルを指定します。

checkerr.o

エラー・チェック用のユーティリティー・オブジェクト・ファイルを組み込みます。

-bnoentry

共用ライブラリーへのデフォルトのエンタリー・ポイントを指定しません。

-bE:\$1.exp

エクスポート・ファイルを指定します。エクスポート・ファイルには、ストアード・プロシージャのリストが含まれています。

-L\$DB2PATH/\$LIB

DB2 ランタイム共用ライブラリーのロケーションを指定します。例えば、
\$HOME/sql1lib/lib32 のように指定します。

-ldb2 データベース・マネージャー・ライブラリーとリンクします。

他のコンパイラー・オプションについては、コンパイラーの資料をご覧ください。

AIX Micro Focus COBOL ルーチンのコンパイルとリンクのオプション

以下は、bldrtn ビルド・スクリプトに示されているように、AIX 上で Micro Focus COBOL コンパイラーを使用して、COBOL ルーチン (ストアード・プロシージャ) を構築するのにお勧めするコンパイルとリンクのオプションです。DB2 MicroFocus COBOL 組み込みファイルは、COBCPY 環境変数をセットアップすることによって検出されます。したがって、コンパイルのステップで -I フラグは必要ありません。この例は、bldapp スクリプトを参照してください。

bldrtn のコンパイルとリンクのオプション

コンパイル・オプション:	
cob	MicroFocus COBOL コンパイラー。
-c	コンパイルのみを実行し、リンクは実行しません。コンパイルとリンクは別個のステップです。
\$EXTRA_COBOL_FLAG="-C MFSYNC" 64 ビット・サポートを使用可能にします。	
-x	-c オプションと一緒に使用すると、オブジェクト・モジュールへとコンパイルします。
リンク・オプション:	
cob	コンパイラーをリンカーのフロントエンドとして使用します。
-x	共用ライブラリーを作成します。
-o \$1	実行可能プログラムを指定します。
\$1.o	プログラム・オブジェクト・ファイルを指定します。
-L\$DB2PATH/\$LIB DB2 ランタイム共用ライブラリーのロケーションを指定します。例えば、\$HOME/sql1lib/lib32 のように指定します。	
-ldb2	DB2 ライブラリーとリンクします。
-ldb2gmf	Micro Focus COBOL 用 DB2 例外ハンドラー・ライブラリーとリンクします。
他のコンパイラー・オプションについては、コンパイラーの資料をご覧ください。	

HP-UX Micro Focus COBOL ルーチンのコンパイルとリンクのオプション

以下は、bldrtn ビルド・スクリプトに示されているように、HP-UX 上で Micro Focus COBOL コンパイラーを使用して、COBOL ルーチン (ストアード・プロシージャ) を構築するのにお勧めするコンパイルとリンクのオプションです。

bldrtn のコンパイルとリンクのオプション

コンパイル・オプション:

cob COBOL コンパイラー。

\$EXTRA_COBOL_FLAG

HP-UX プラットフォームが IA64 で、64 ビット・サポートが有効な場合は、"-C MFSYNC" が入ります。

リンク・オプション:

-y 目的の出力が共用ライブラリーであることを指定します。

-o \$1 実行可能ファイルを指定します。

-L\$DB2PATH/\$LIB

DB2 ランタイム共用ライブラリーのロケーションを指定します。

-ldb2 DB2 共用ライブラリーにリンクします。

-ldb2gmf

Micro Focus COBOL 用 DB2 例外ハンドラー・ライブラリーとリンクします。

他のコンパイラー・オプションについては、コンパイラーの資料をご覧ください。

Solaris Micro Focus COBOL ルーチンのコンパイルとリンクのオプション

以下は、bldrtn ビルド・スクリプトに示されているように、Solaris 上で Micro Focus COBOL コンパイラーを使用して、COBOL ルーチン (ストアード・プロシージャ) を構築するのにお勧めするコンパイルとリンクのオプションです。

bldrtn のコンパイルとリンクのオプション

コンパイル・オプション:

cob COBOL コンパイラー。

-cx オブジェクト・モジュールにコンパイルします。

\$EXTRA_COBOL_FLAG

64 ビット・サポート用には、値 "-C MFSYNC" を入れます。その他の場合には、値を入れません。

リンク・オプション:

- cob** コンパイラーをリンカーのフロントエンドとして使用します。
- y** 必要なものを完備したスタンドアロン共用ライブラリーを作成します。
- o \$1** 実行可能プログラムを指定します。
- \$1.o** プログラム・オブジェクト・ファイルを指定します。
- L\$DB2PATH/\$LIB**
DB2 ランタイム共用ライブラリーのロケーションを指定します。例えば、
\$HOME/sql1lib/lib64。
- ldb2** DB2 ライブラリーとリンクします。
- ldb2gmf**
Micro Focus COBOL 用 DB2 例外ハンドラー・ライブラリーとリンクします。

他のコンパイラー・オプションについては、コンパイラーの資料をご覧ください。

Linux Micro Focus COBOL ルーチンのコンパイルとリンクのオプション

以下は、bldrtn ビルド・スクリプトに示されているように、Linux 上で Micro Focus COBOL コンパイラーを使用して、COBOL ルーチン (ストアード・プロシージャ) を構築するのにお勧めするコンパイルとリンクのオプションです。

bldrtn のコンパイルとリンクのオプション

コンパイルおよびリンクのオプション:

- cob** COBOL コンパイラー
- \$EXTRA_COBOL_FLAG**
64 ビット・サポート用には、値 "-C MFSYNC" を入れます。その他の場合には、
値を入れません。
- y** 内蔵タイプの呼び出し可能共有オブジェクトにコンパイルすることを指定します。
- o \$1** 実行可能ファイルを指定します。
- \$1.cb1** ソース・ファイルを指定します
- L\$DB2PATH/\$LIB**
DB2 ランタイム共用ライブラリーのロケーションを指定します。
- ldb2** DB2 ライブラリーとリンクします。
- ldb2gmf**
Micro Focus COBOL 用 DB2 例外ハンドラー・ライブラリーとリンクします。

他のコンパイラー・オプションについては、コンパイラーの資料をご覧ください。

注: ストアード・プロシージャをコンパイルした後、コピーするか、または /usr/lib へリンクする必要があります。 ld.so.conf を編集するだけでは不十分です。

Windows IBM COBOL ルーチンのコンパイルとリンクのオプション

以下は、bldrtn.bat バッチ・ファイルに示されているように、Windows 上で IBM VisualAge COBOL コンパイラーを使用して、COBOL ルーチン (ストアード・プロシージャとユーザー定義関数) を構築するのにお勧めするコンパイルとリンクのオプションです。

bldrtn のコンパイルとリンクのオプション

コンパイル・オプション:

cob2 IBM VisualAge COBOL コンパイラー。

-qpgmname(mixed)

コンパイラーに、大/小文字混合の名前を持つライブラリーのエン트리・ポイントの CALL を許可するように指示します。

-c コンパイルのみを実行し、リンクは実行しません。このバッチ・ファイルでは、コンパイルとリンクは別個のステップです。

-qlib コンパイラーに COPY ステートメントを処理するように指示します。

-Ipath DB2 組み込みファイルのロケーションを指定します。例えば、**-I"%DB2PATH %\include\cobol_a"**。

%EXTRA_COMPFLAG%

"set IBMCOB_PRECOMP=true" がコメント化されていない場合、IBM COBOL プリコンパイラーを使用して組み込み SQL がプリコンパイルされます。入力パラメータに応じて、以下のいずれかの公式で呼び出されます。

-q"SQL('database sample CALL_RESOLUTION DEFERRED')"

デフォルトのサンプル・データベースを使用してプリコンパイルし、呼び出し解決を据え置きます。

-q"SQL('database %2 CALL_RESOLUTION DEFERRED')"

ユーザー指定のデータベースを使用してプリコンパイルし、呼び出し解決を据え置きます。

リンク・オプション:

ilink IBM VisualAge COBOL リンカーを使用します。

/free 自由書式。

/no1 ログなし。

/d11 DLL をソース・プログラム名を使用して作成します。

db2api.lib

DB2 ライブラリーとリンクします。

%1.exp エクスポート・ファイルを組み込みます。

%1.obj プログラム・オブジェクト・ファイルを組み込みます。

iwzrwin3.obj

IBM VisualAge COBOL が提供するオブジェクト・ファイルを組み込みます。

他のコンパイラー・オプションについては、コンパイラーの資料をご覧ください。

Windows Micro Focus COBOL ルーチンのコンパイルとリンクのオプション

以下は、bldrtn.bat バッチ・ファイルに示されているように、Windows 上で Micro Focus COBOL コンパイラーを使用して、COBOL ルーチン (ストアード・プロシージャとユーザー定義関数) を構築するのにお勧めするコンパイルとリンクのオプションです。

bldrtn のコンパイルとリンクのオプション

コンパイル・オプション:

cobol Micro Focus COBOL コンパイラー。

/case 外部シンボルが大文字に変換されないようにします。

リンク・オプション:

cbllink

リンク・エディットのために Micro Focus COBOL リンカーを使用します。

/d .DLL ファイルを作成します。

db2api.lib

DB2 API ライブラリーとリンクします。

他のコンパイラー・オプションについては、コンパイラーの資料をご覧ください。

AIX での IBM COBOL ルーチンの構築

DB2 には、COBOL 組み込み SQL と DB2 管理 API プログラムをコンパイルしてリンクするためのビルド・スクリプトが用意されていて、このファイルを使用して構築できるサンプル・プログラムと一緒に `sql1lib/samples/cobol` ディレクトリーに置かれています。

`sql1lib/samples/cobol` にあるスクリプト・ファイル `bldrtn` には、ルーチン (ストアード・プロシージャ) を構築するためのコマンドが入っています。このスクリプトは、クライアント・アプリケーションから呼び出せるルーチンを共用ライブラリーの中でコンパイルします。

第 1 パラメーター \$1 には、ソース・ファイルの名前を指定します。第 2 パラメーター \$2 には、接続先のデータベースの名前を指定します。共用ライブラリーは、データベースが置かれているのと同じインスタンス上に作成する必要があるため、ユーザー ID やパスワードを指定するパラメーターはありません。

最初のパラメーター (ソース・ファイル名) だけが、必須です。スクリプトは、ソース・ファイル名 \$1 を共用ライブラリー名として使用します。データベース名は任意で指定します。データベース名を指定しない場合は、プログラムはデフォルトの `sample` データベースを使用します。

サンプル・データベースに接続して、ソース・ファイル `outsrv.sqb` からサンプル・プログラム `outsrv` を構築するには、次のように入力します。

```
bldrtn outsrv
```

他のデータベースに接続しているときは、さらにデータベース名も含めます。

```
bldrtn outsrv database
```

スクリプト・ファイルは、共用ライブラリーをサーバー上の `sqllib/function` というパスにコピーします。

ルーチンの共用ライブラリー `outsrv` を構築し終わったら、クライアント・アプリケーション `outcli` を構築することができます。これは、ライブラリー内のルーチンと呼び出すアプリケーションです。`outcli` は、スクリプト・ファイル `bldapp` を使用して構築することができます。

ルーチンと呼び出すには、次のように入力してサンプル・クライアント・アプリケーションを実行します。

```
outcli database userid password
```

ここで、

database

接続先のデータベースの名前です。名前は、`sample` またはそのリモート別名、あるいはその他の名前にすることができます。

userid 有効なユーザー ID です。

password

ユーザー ID の有効なパスワードです。

クライアント・アプリケーションは共用ライブラリー `outsrv` にアクセスし、同一名のルーチンをサーバー・データベース上で実行します。この出力は、クライアント・アプリケーションに戻されます。

UNIX Micro Focus COBOL ルーチンの構築

DB2 データベース製品では、Micro Focus COBOL 組み込み SQL および DB2 API プログラムをコンパイルしてリンクするためのビルド・スクリプトが用意されています。このファイルを使用して構築できるサンプル・プログラムと一緒に `sqllib/samples/cobol_mf` ディレクトリーに置かれています。

このタスクについて

スクリプト `bldrtn` には、ルーチン (ストアド・プロシージャ) を構築するためのコマンドが入っています。このスクリプトは、クライアント・アプリケーションから呼び出せるルーチンのソース・ファイルを共用ライブラリーの中でコンパイルします。

第 1 パラメーター `$1` には、ソース・ファイルの名前を指定します。スクリプトは、そのソース・ファイル名を共用ライブラリー名として使用します。第 2 パラメーター `$2` には、接続先のデータベースの名前を指定します。共用ライブラリーは、データベースが置かれているのと同じインスタンス上で構築する必要があるため、ユーザー ID やパスワードを指定するパラメーターはありません。

最初のパラメーター (ソース・ファイル名) だけが、必須です。データベース名は任意で指定します。データベース名を指定しない場合は、プログラムはデフォルトの `sample` データベースを使用します。

手順

1. Micro Focus COBOL ルーチンをビルドする場合は、事前に以下のコマンドを実行する必要があります。

```
db2stop
db2set DB2LIBPATH=$LD_LIBRARY_PATH
db2set DB2ENVLIST="COBDIR LD_LIBRARY_PATH"
db2set
db2start
```

db2stop がデータベースを確実に停止するようにしてください。最後の db2set コマンドが設定値をチェックするために出されます。 **DB2LIBPATH** および **DB2ENVLIST** が正しく設定されるようにしてください。

2. サンプル・データベースに接続している場合、ソース・ファイル `outsrv.sqb` からサンプル・プログラム `outsrv` を構築するには、`bldrtn outsrv` を入力します。

他のデータベースに接続しているときは、さらにデータベース名も入力します。以下に例を示します。

```
bldrtn outsrv database
```

スクリプト・ファイルは、共用ライブラリーをサーバー上の `sqllib/function` というパスにコピーします。

ストアード・プロシージャ `outsrv` を構築すると、そのストアード・プロシージャを呼び出すクライアント・アプリケーション `outcli` を構築できます。スクリプト・ファイル `bldapp` を使用して、`outcli` を構築することができます。

3. ストアード・プロシージャを呼び出すためには、次のように入力してサンプル・クライアント・アプリケーションを実行します。`outcli database userid password`

ここで、

database

接続先のデータベースの名前です。名前は、`sample` かその別名、またはその他の名前にすることができます。

userid 有効なユーザー ID です。

password

ユーザー ID の有効なパスワードです。

クライアント・アプリケーションは共用ライブラリー `outsrv` にアクセスし、サーバー・データベース上の同じ名前のストアード・プロシージャ関数を実行します。出力は、クライアント・アプリケーションに戻されます。

Windows での IBM COBOL ルーチンの構築

DB2 には、IBM COBOL の DB2 API と組み込み SQL プログラムをコンパイルおよびリンクするためのビルド・スクリプトが用意されています。このファイルを使用して構築できるサンプル・プログラムと一緒に `sqllib\samples\cobol` ディレクトリーに置かれています。

DB2 は、Windows での IBM COBOL アプリケーションの構築用に、DB2 プリコンパイラーと IBM COBOL プリコンパイラーの 2 種類のプリコンパイラーをサポートします。デフォルトは DB2 プリコンパイラーです。使用するバッチ・ファイルの該当する行のコメントを外すことにより、IBM COBOL プリコンパイラーを選択できます。IBM COBOL でのプリコンパイルは、特定のプリコンパイル・オプションを使用して、コンパイラー単体で実行できます。

バッチ・ファイル `bldrtn.bat` には、組み込み SQL ルーチン (ストアド・プロシージャ) を構築するためのコマンドが入っています。このバッチ・ファイルは、サーバー上の DLL 内でルーチンをコンパイルします。これは 2 個のパラメーターをとります。それらは、バッチ・ファイル内では変数 `%1` と `%2` で表されます。

最初のパラメーター `%1` には、ソース・ファイルの名前を指定します。バッチ・ファイルでは、ソース・ファイル名 `%1` を DLL 名に使用します。第 2 パラメーター `%2` には、接続先のデータベースの名前を指定します。ストアド・プロシージャは、データベースが置かれているのと同じインスタンス上で構築する必要があるため、ユーザー ID やパスワードを指定するパラメーターはありません。

最初のパラメーター (ソース・ファイル名) だけが、必須です。データベース名は任意で指定します。データベース名を指定しない場合は、プログラムはデフォルトの `sample` データベースを使用します。

デフォルトの DB2 プリコンパイラーを使用している場合、`bldrtn.bat` は、プリコンパイルおよびバインドのファイル `embprep.bat` にパラメーターを渡します。

IBM COBOL プリコンパイラーを使用している場合、`bldrtn.bat` は `.sqb` ソース・ファイルを `.cbl` ソース・ファイルにコピーします。コンパイラーは `.cbl` ソース・ファイルに対して特定のプリコンパイル・オプションを使用して、プリコンパイルを実行します。

サンプル・データベースに接続して、ソース・ファイル `outsrv.sqb` からサンプル・プログラム `outsrv` を構築するには、次のように入力します。

```
bldrtn outsrv
```

他のデータベースに接続しているときは、さらにデータベース名も含めます。

```
bldrtn outsrv database
```

バッチ・ファイルは、DLL をサーバー上の `sqllib%function` というパスにコピーします。

DLL `outsrv` の構築が完了したなら、DLL 内のルーチン (DLL と同名) を呼び出すクライアント・アプリケーション `outcli` を構築することができます。 `outcli` は、`bldapp.bat` バッチ・ファイルを使用して構築することができます。

`outsrv` ルーチンを呼び出すには、次のように入力してサンプル・クライアント・アプリケーションを実行します。

```
outcli database userid password
```

ここで、

database

接続先のデータベースの名前です。名前は、`sample` またはそのリモート別名、あるいはその他の名前にすることができます。

userid 有効なユーザー ID です。

password

ユーザー ID の有効なパスワードです。

クライアント・アプリケーションは DLL `outsrv` にアクセスし、同一名のルーチンをサーバー・データベース上で実行します。この出力は、クライアント・アプリケーションに戻されます。

Windows での Micro Focus COBOL ルーチンの構築

DB2 には、Micro Focus COBOL の DB2 API と組み込み SQL プログラムをコンパイルおよびリンクするためのビルド・スクリプトが用意されています。このファイルを使用して構築できるサンプル・プログラムと一緒に `sqllib\samples\cobol_mf` ディレクトリーに置かれています。

バッチ・ファイル `bldrtn.bat` には、組み込み SQL ルーチン (ストアード・プロシージャ) を構築するためのコマンドが入っています。このバッチ・ファイルは、サーバー上の DLL 内でルーチンをコンパイルします。このバッチ・ファイルは 2 個のパラメーターをとります。それらは、バッチ・ファイル内では変数 `%1` と `%2` で表されます。

最初のパラメーター `%1` には、ソース・ファイルの名前を指定します。バッチ・ファイルでは、ソース・ファイル名 `%1` を DLL 名に使用します。第 2 パラメーター `%2` には、接続先のデータベースの名前を指定します。ストアード・プロシージャは、データベースが置かれているのと同じインスタンス上で構築する必要があるため、ユーザー ID やパスワードを指定するパラメーターはありません。

最初のパラメーター (ソース・ファイル名) だけが、必須です。データベース名は任意で指定します。データベース名を指定しない場合は、プログラムはデフォルトの `sample` データベースを使用します。

サンプル・データベースに接続している場合、ソース・ファイル `outsrv.sqb` からサンプル・プログラム `outsrv` を構築するには、次のように入力します。

```
bldrtn outsrv
```

他のデータベースに接続しているときは、さらにデータベース名も入力します。

```
bldrtn outsrv database
```

スクリプト・ファイルは、DLL をサーバー上の `sqllib/function` というパスにコピーします。

DLL `outsrv` の構築が完了したなら、DLL 内のルーチン (DLL と同名) を呼び出すクライアント・アプリケーション `outcli` を構築できます。 `outcli` は、 `bldapp.bat` バッチ・ファイルを使用して構築することができます。

`outsrv` ルーチンを呼び出すには、次のように入力してサンプル・クライアント・アプリケーションを実行します。

`outcli database userid password`

ここで、

database

接続先のデータベースの名前です。名前は、`sample` かその別名、またはその他の名前にすることができます。

userid 有効なユーザー ID です。

password

ユーザー ID の有効なパスワードです。

クライアント・アプリケーションは DLL `outsrv` にアクセスします。これは、同一名のルーチンをサーバー・データベース上で実行します。出力は、クライアント・アプリケーションに戻されます。

第 7 章 Java ルーチン

Java ルーチンは、Java プログラミング言語インプリメンテーションを持つ外部ルーチンです。Java ルーチンは、CREATE PROCEDURE または CREATE FUNCTION ステートメントを実行することによって、データベースに作成されます。このステートメントは、LANGUAGE JAVA 節を使って Java でルーチンをインプリメントすることを指定するものでなければなりません。また、EXTERNAL 節を使用して、それをインプリメントする Java クラスを指定する必要もあります。

外部プロシージャ、関数、およびメソッドは、Java で作成することができます。

Java ルーチンは SQL ステートメントを実行できます。

Java ルーチンのコンテキストで重要な用語は、以下のとおりです。

JDBC データベースのデータにアクセスするためのアプリケーション・プログラミング・インターフェースの一種で、動的 SQL 実行をサポートする。

SQLJ データベースのデータにアクセスするためのアプリケーション・プログラミング・インターフェースの一種で、動的および静的 SQL 実行をサポートする。

CREATE ステートメント

データベース内にルーチンを作成するために使用される SQL 言語の CREATE ステートメント。

ルーチン本体のソース・コード

Java ルーチン・インプリメンテーションを含むソース・コード・ファイル。Java ルーチンは、JDBC または SQLJ アプリケーション・プログラミング・インターフェースのいずれかを使って、データベースにアクセスできます。

JDBC Java コードの動的 SQL ステートメントの実行に対するサポートを提供するアプリケーション・プログラミング・インターフェース。

SQLJ Java コードの静的 SQL ステートメントの実行に対するサポートを提供するアプリケーション・プログラミング・インターフェース。

SDK for Java

Java ソース・コードのコンパイルに必要なために提供されている Software Development Kit for Java。

ルーチン・クラス

コンパイルされた形式の Java ルーチン・ソース・コードが含まれる Java ソース・コード・ファイル。Java クラス・ファイルはその独自のものが存在する場合もあれば、JAR ファイル内の Java クラス・ファイルの集合の 1 つである場合もあります。

サポートされる Java ルーチン開発ソフトウェア

Java で外部ルーチンを開発してデプロイするには、サポートされる Java 開発ソフトウェアを使用する必要があります。

Java ルーチン開発用の Software Development Kit (SDK) のうち、サポートされている最小のバージョンは、IBM SDK for Java 1.4.2 です。ただし、IBM SDK for Java 1.4.2 のサポートは推奨されていないため、今後のリリースでは廃止される可能性があります。

Java ルーチン開発のため SDK のうち、サポートされている最大のバージョンは、次のとおりです。

- IBM SDK for Java 6

IBM SDK for Java 6 は、以下のオペレーティング・システム上でストアド・プロシージャおよびユーザー定義関数をサポートします。

- AIX 5
- Itanium ベースのシステム上の HP-UX
- Linux (x86 版)
- Linux (AMD64/EM64T 版)
- Linux (POWER 版)
- Windows (x86 版)
- Windows (x64 上の AMD64/EM64T 版)

DB2 Database for Linux, UNIX, and Windows 製品によってインストールされる SDK for Java を使用することをお勧めします。代替の SDK for Java を指定することもできます。代替の SDK for Java を指定する場合は、DB2 インスタンスと同じビット幅でなければなりません。

Java での DB2 データベース・アプリケーション開発のためにサポートされるこれ以外の開発ソフトウェアはすべて、Java での外部ルーチン開発に使用できます。

Java ルーチンでの JDBC および SQLJ アプリケーション・プログラミング・インターフェースのサポート

Java で作成された外部ルーチンは、以下のアプリケーション・プログラミング・インターフェース (API) で使用できます。

- JDBC
- SQLJ

DB2 バージョン 9.5 以降、IBM Data Server Driver for JDBC and SQLJ がデフォルトのドライバーとなりました。これは JDBC と SQLJ API の両方をサポートしており、外部 Java ルーチンの開発に使用することができます。

IBM Data Server Type 2 Driver も Java ルーチンの開発に使用できますが、JDBC Type 2 しかサポートしていません。IBM Data Server Type 2 Driver は推奨されていないため、IBM Data Server Driver for JDBC and SQLJ を使用することを強くお勧めします。

Java ルーチンのインプリメントのための手順は、使用する API またはドライバーにかかわらず同じです。

SDK for Java ルーチン開発の仕様 (UNIX)

UNIX 環境で Java ルーチン・コードを作成して実行するために、DB2 データベース・マネージャー構成パラメーター `JDK_PATH` を、DB2 データベース・サーバー上にインストールされた SDK for Java のパスに設定する必要があります。DB2 Database for Linux, UNIX, and Windows 製品のインストール・プロセスが SDK for Java をインストールする場合、デフォルトで、`JDK_PATH` パラメーターはその SDK for Java のパスに設定されます。

- `$INSTDIR/sql/lib/java/jdk32` Linux on x86 環境にインストールされた DB2 データベース・サーバーの場合
- `$INSTDIR/sql/lib/java/jdk64` その他のすべての環境にインストールされた SDK を備えた DB2 データベース・サーバーの場合

`JDK_PATH` パラメーター値は、コンピューター上にインストールされた別の SDK for Java を指定するように変更できますが、SDK は DB2 インスタンスと同じビット幅でなければなりません。

前提条件

- ユーザーは DB2 データベース・サーバーにアクセスする必要があります。
- ユーザーはデータベース・マネージャー構成ファイルの読み取りおよび更新権限を持っている必要があります。
- ユーザーは、DB2 インスタンスがインストールされたファイル・システムで SDK for Java をインストールする権限を持っている必要があります。

手順

1. 以下のコマンドを DB2 コマンド・ウィンドウから発行して、`JDK_PATH` パラメーター値を調べます。

```
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` パラメーター値は、出力の先頭近くに表示されます。

2. 別の SDK for Java を使用する場合、それを DB2 データベース・サーバー上にインストールし、インストール・パスをメモします。インストール・パスの値は次のステップで必要になります。
3. 以下のコマンドを DB2 コマンド・ウィンドウから発行して、`JDK_PATH` パラメーター値を更新します。`<path>` はもう一方の SDK for Java がインストールされているパスです。

```
db2 update dbm cfg using JDK_PATH <path>
```

4. 以下の DB2 コマンドを DB2 コマンド・ウィンドウから発行して、DB2 インスタンスを停止して、再始動します。

```
db2stop;
```

```
db2start;
```

5. 以下のコマンドを DB2 コマンド・ウィンドウから発行して、JDK_PATH パラメーター値が正しく設定されていることを確認します。

```
db2 get dbm cfg
```

これらのステップが完了したなら、指定された SDK for Java が Java ルーチンの作成および実行に使用されます。環境内の CLASSPATH、PATH、および LIBPATH 環境変数は自動的に設定されます。

Java ルーチン用のドライバーの指定

Java ルーチンの開発および呼び出しには、JDBC または SQLJ ドライバーを指定する必要があります。Java ルーチンは、以下の 2 つのドライバーのいずれかを使用することができます。

- IBM Data Server Driver for JDBC and SQLJ バージョン 3.0
- DB2 Type 2 Driver

IBM Data Server Driver for JDBC and SQLJ バージョン 4.0 db2jcc4.jar には、JDBC バージョン 4.0 のさまざまな機能が含まれています。ドライバーは、DB2 バージョン 9.5 以上でサポートされています。しかし、Java ルーチンでは、IBM Data Server Driver for JDBC and SQLJ バージョン 4.0 はサポートされていません。

デフォルトでは、DB2 は、IBM Data Server Driver for JDBC and SQLJ を使用します。このドライバーが望ましい理由は、このドライバーがより強じんである上に、DB2 Type 2 Driver が使用すべきでないドライバーとされていること、さらに、このドライバーが Java ルーチンで以下のものが使用されている場合の前提条件であることが挙げられます。

- データ・タイプ XML のパラメーター
- データ・タイプ XML の変数
- XML データへの参照
- XML 関数への参照
- 他の任意のネイティブ XML フィーチャー

既存の Java ルーチンをアップグレードするときに問題が発生する場合、DB2_USE_DB2JCCT2_JROUTINE DB2 環境変数を値 NO に設定することにより、レガシーである IBM DB2 Type 2 Driver を使用することができます。設定するには、以下のコマンドを DB2 コマンド・ウィンドウから発行します。

```
db2set DB2_USE_DB2JCCT2_JROUTINE=NO
```

このコマンドの発行後、DB2 インスタンスをいったん停止してから再始動し、変更内容を有効化する必要があります。

Java (JDBC および SQLJ) ルーチン開発のためのツール

ツールの使用により Java ルーチン開発を短時間で容易に行うことができます。以下の DB2 ツールは、Java ルーチンの開発、デバッグ、およびデプロイのためのグラフィカル・ユーザー・インターフェース・サポートを提供します。

- IBM Data Studio

以下のコマンド行インターフェースを、Java ルーチンの開発、デバッグ、およびデプロイに使用することもできます。

- DB2 コマンド行プロセッサ
- DB2 コマンド・ウィンドウ

その他の IBM ソフトウェア製品により、以下のような Java ルーチン開発のためのグラフィック・ツールが提供されています。

- IBM Optim Development Studio
- IBM Rational Application Developer
- 分散統一デバッガー

Java ルーチンの設計

Java ルーチンの設計は、Java ルーチンを作成する前に行うべき作業です。Java ルーチンの設計は、他のプログラミング言語でインプリメントされる外部ルーチンの設計と、Java データベース・アプリケーションの設計の両方に関連します。

組み込み SQL アプリケーション開発に関する知識と経験、および外部ルーチンに関する一般的な知識。以下のトピックは、必要な前提条件に関する情報の一部を提供しています。

外部ルーチンのフィーチャーおよび使用方法の詳細については、以下を参照してください。

- 24 ページの『外部ルーチンのインプリメンテーション』

JDBC または SQLJ のアプリケーション・プログラミング・インターフェースを使用して基本の Java アプリケーションをプログラムする方法については詳しくは、以下を参照してください。

- 「Java アプリケーションの開発」内の『簡単な JDBC アプリケーションの例』
- 「Java アプリケーションの開発」内の『簡単な SQLJ アプリケーションの例』

すでに前提知識がある場合、Java ルーチンの設計には、主に Java ルーチンの固有のフィーチャーおよび特性について習得することが関係しています。

- 288 ページの『Java ルーチンでサポートされている SQL データ・タイプ』
- 290 ページの『Java ルーチンのパラメーター』
- 291 ページの『パラメーター・スタイル JAVA プロシージャ』
- 292 ページの『PARAMETER STYLE JAVA の Java 関数およびメソッド』
- 303 ページの『JDBC プロシージャからの結果セットの戻り』
- 303 ページの『SQLJ プロシージャからの結果セットの戻り』
- 307 ページの『Java ルーチンに関する制約事項』
- 73 ページの『Java の表関数実行モデル』

Java の特性について理解した後、以下を参照することができます。

- 310 ページの『コマンド行からの Java ルーチンの作成』

Java ルーチンでサポートされている SQL データ・タイプ

JDBC および SQLJ のアプリケーション・プログラミング・インターフェース仕様に従って SQL データ・タイプ値を保管するときは、Java プログラミング言語データ・タイプを Java ソース・コードで使用する必要があります。DB2 Driver for JDBC and SQLJ は、特定のデータ・タイプ・マッピングに従って、Java ソース・コードと DB2 データベースの間でやり取りされるデータを変換します。

データ・マッピングが有効なのは、以下のものに対してです。

- Java データベース・アプリケーション
- PARAMETER STYLE JAVA を使用して定義およびインプリメントされた Java ルーチン

SQL データ・タイプにマップされる Java データ・タイプは次のとおりです。

表 43. Java 宣言にマップされる SQL データ・タイプ

SQL 列名	Java データ・タイプ	SQL 列タイプ記述
SMALLINT (500 または 501)	short, boolean	16 ビットの符号付き整数
INTEGER (496 または 497)	int	32 ビットの符号付き整数
BIGINT ¹ (492 または 493)	long	64 ビットの符号付き整数
REAL (480 または 481)	float	単精度浮動小数点
DOUBLE (480 または 481)	double	倍精度浮動小数点
DECIMAL(<i>p,s</i>) (484 または 485)	java.math.BigDecimal	パック 10 進数
CHAR(<i>n</i>) (452 または 453)	java.lang.String	長さが <i>n</i> の固定長文字ストリング (<i>n</i> の範囲は 1 から 254 まで)
CHAR(<i>n</i>) FOR BIT DATA	byte[]	長さが <i>n</i> の固定長文字ストリング (<i>n</i> の範囲は 1 から 254 まで)
VARCHAR(<i>n</i>) (448 または 449)	java.lang.String	可変長文字ストリング
VARCHAR(<i>n</i>) FOR BIT DATA	byte[]	可変長文字ストリング
LONG VARCHAR (456 または 457)	java.lang.String	long 可変長文字ストリング
LONG VARCHAR FOR BIT DATA	byte[]	long 可変長文字ストリング
BLOB(<i>n</i>) (404 または 405)	java.sql.Blob	ラージ・オブジェクト可変長バイナリー・ストリング
CLOB(<i>n</i>) (408 または 409)	java.sql.Clob	ラージ・オブジェクト可変長文字ストリング
DBCLOB(<i>n</i>) (412 または 413)	java.sql.Clob	ラージ・オブジェクト可変長 2 バイト文字ストリング

表 43. Java 宣言にマップされる SQL データ・タイプ (続き)

SQL 列名	Java データ・タイプ	SQL 列タイプ記述
DATE (384 または 385)	java.sql.Date	10 バイトの文字ストリング
TIME (388 または 389)	java.sql.Time	8 バイトの文字ストリング
TIMESTAMP (392 または 393)	java.sql.Timestamp	文字ストリングは、秒の小数部分の指定された桁数に応じて、19 から 32 バイトの長さにするすることができます。 TIMESTAMP データ・タイプの秒の小数部分には、オプションで、0 から 12 桁のタイム・スタンプ精度を指定できます。 タイム・スタンプ値が、秒の小数部分の桁数が異なるタイム・スタンプ変数に割り当てられると、その値は変数のフォーマットに合うように切り捨てられるか、0 を埋め込むことによって調整されます。
GRAPHIC(<i>n</i>) (468 または 469)	java.lang.String	固定長 2 バイト文字ストリング
VARGRAPHIC(<i>n</i>) (464 または 465)	java.lang.String	NULL 終了ではない可変長 2 バイト文字ストリング (2 バイトのストリング長指定子を含む)
LONGVARGRAPHIC (472 または 473)	java.lang.String	NULL 終了ではない可変長 2 バイト文字ストリング (2 バイトのストリング長指定子を含む)
XML(<i>n</i>) (408 または 409)	java.sql.Clob	XML データ・タイプは、CLOB データ・タイプと同じ方法で表されます。つまり、ラージ・オブジェクトの可変長文字ストリングとして表されます。
ARRAY	java.sql.Array	SQL データの配列。

注:

1. DB2[®] Universal Database™ バージョン 8.1 クライアントから DB2 Universal Database バージョン 7.1 (または 7.2) サーバーに接続している Java アプリケーションでは、以下のことに注意してください。BIGINT 値の検索に getObject() 方式が使用された場合、java.math.BigDecimal オブジェクトが返されます。
2. SQL 配列データ・タイプのパラメーターはクラス com.ibm.db2.ARRAY にマップされます。
3. LONG VARCHAR、LONG VARGRAPHIC、XML、REFERENCE、UDT および ARRAY は、ARRAY データ・タイプとしてはサポートされません。

SQLJ ルーチン内の接続コンテキスト

DB2 Universal Database バージョン 8 でマルチスレッド・ルーチンが採り入れられたため、SQLJ ルーチンでデフォルトの接続コンテキストの使用を避けることが重要

です。つまり、各 SQL ステートメントは `ConnectionContext` オブジェクトを明示的に指定し、コンテキストは Java 方式で明示的にインスタンス化する必要があります。例えば、前のリリースの DB2 では、SQLJ ルーチンを以下のように書くことができました。

```
class myClass
{
    public static void myRoutine( short myInput )
    {
        DefaultContext ctx = DefaultContext.getDefaultContext();
        #sql { some SQL statement };
    }
}
```

このデフォルト・コンテキストの使用は、マルチスレッド環境のすべてのスレッドが同じ接続コンテキストを使用することになり、予期しない障害が発生します。

上記の SQLJ ルーチンを、以下のように変更する必要があります。

```
#context MyContext;

class myClass
{
    public static void myRoutine( short myInput )
    {
        MyContext ctx = new MyContext( "jdbc:default:connection", false );
        #sql [ctx] { some SQL statement };
        ctx.close();
    }
}
```

このように、ルーチンのそれぞれの呼び出しで、それぞれ固有の `ConnectionContext` (および基礎となる JDBC 接続) を作成することにより、並行スレッドによる予期しない干渉が回避されます。

Java ルーチンのパラメーター

Java ルーチンでのパラメーター宣言は、サポートされているいずれかのパラメーター・スタイルの要件を満たしている必要があります。Java ルーチンでは、以下の 2 つのパラメーター・スタイルがサポートされています。

- PARAMETER STYLE JAVA
- PARAMETER STYLE DB2GENERAL

ルーチンの CREATE ステートメント内で、PARAMETER STYLE JAVA 節を指定することを強くお勧めします。PARAMETER STYLE JAVA ではルーチンは、Java 言語と SQLJ ルーチンの仕様に準拠したパラメーター引き渡し規則に従います。

一部の Java ルーチン・フィーチャーは、PARAMETER STYLE JAVA ではインプリメントまたは使用できません。それらは次のとおりです。

- 表関数
- 関数内のスクラッチパッド
- 関数内の DBINFO 構造へのアクセス
- 関数またはメソッドの FINAL CALL (および別個の最初の呼び出し) を行うための機能。

上記のフィーチャーをインプリメントする必要がある場合、パラメーター・スタイル DB2GENERAL を使用して、C でルーチンをインプリメントするか、Java で作成できます。

上記のような特定の場合とは別に、常に PARAMETER STYLE JAVA を使用して、Java ルーチンを作成およびインプリメントする必要があります。

パラメーター・スタイル JAVA プロシージャ

Java プロシージャの設定で推奨されているパラメーター・スタイルは、PARAMETER STYLE JAVA です。

PARAMETER STYLE JAVA ストアード・プロシージャのシグニチャーは以下のフォーマットに準拠します。

```
public static void method-name ( SQL-arguments, ResultSet[] result-set-array )
                                throws SQLException
```

method-name

メソッドの名前。ルーチンの登録時にこの値は、CREATE PROCEDURE ステートメントの EXTERNAL NAME 節内のクラス名を使用して指定されます。

SQL-arguments

CREATE PROCEDURE ステートメント内の入力パラメーターのリストに対応します。OUT または INOUT モード・パラメーターが、単一エレメント配列で渡されます。CREATE PROCEDURE ステートメントの DYNAMIC RESULT SETS 節に指定された各結果セットごとに、ResultSet タイプの単一エレメント配列がパラメーター・リストに追加されます。

result-set-array

ResultSet オブジェクトの配列の名前。CREATE PROCEDURE ステートメントの DYNAMIC RESULT SETS パラメーターに宣言された各結果セットごとに、Java メソッド・シグニチャー内で ResultSet[] タイプのパラメーターを宣言する必要があります。

以下に、入力パラメーターを受け入れて、出力パラメーターと結果セットを戻す Java のストアード・プロシージャの例を示します。

```
public static void javastp( int inparm,
                           int[] outparm,
                           ResultSet[] rs
                           )
    throws SQLException
{
    Connection con = DriverManager.getConnection( "jdbc:default:connection" );
    PreparedStatement stmt = null;
    String sql = "SELECT value FROM table01 WHERE index = ?";

    //Prepare the query with the value of index
    stmt = con.prepareStatement( sql );
    stmt.setInt( 1, inparm );

    //Execute query and set output parm
    rs[0] = stmt.executeQuery();
    outparm[0] = inparm + 1;

    //Close open resources
    if (stmt != null) stmt.close();
}
```

```

    if (con != null) con.close();
    return;
}

```

このストアード・プロシージャの対応する CREATE PROCEDURE ステートメントは次のとおりです。

```

CREATE PROCEDURE javaproc( IN in1 INT, OUT out1 INT )
  LANGUAGE java
  PARAMETER STYLE java
  DYNAMIC RESULT SETS 1
  FENCED THREADSAFE
  EXTERNAL NAME 'myjar:stpclass.javastp'

```

上記のステートメントでは、stpclass というクラス内にメソッドがあることが前提になっています。なおこのメソッドは、myjar という Jar ID をもったデータベースにカタログされている JAR ファイル内に置かれます。

注:

1. PARAMETER STYLE JAVA ルーチンは、例外を使用してエラー・データを呼び出し側に返送します。例外呼び出しスタックなどの詳細については、管理通知ログを参照してください。この詳細の他に、PARAMETER STYLE JAVA ルーチンを呼び出すことに関するその他の特別な考慮事項はありません。
2. JNI 呼び出しは、Java ルーチンではサポートされていません。ただし、C ルーチンの呼び出しをネストすれば、Java ルーチンから C の機能呼び出しすることは可能です。その作業の一環として、任意の C 機能をルーチンに移動して登録してから、Java ルーチンから呼び出します。

PARAMETER STYLE JAVA の Java 関数およびメソッド

Java 関数およびメソッドに対して推奨されているパラメーター・スタイルは、PARAMETER STYLE JAVA です。

PARAMETER STYLE JAVA の関数およびメソッドのシグニチャーは以下のフォーマットに準拠します。

```
public static return-type method-name ( SQL-arguments ) throws SQLException
```

return-type

スカラー・ルーチンから戻される値のデータ・タイプ。ルーチン内では、戻り値は RETURN ステートメントを介して呼び出し側に返送されます。

method-name

メソッドの名前。ルーチンの登録時にこの値は、ルーチンの CREATE ステートメントの EXTERNAL NAME 節内のクラス名を使用して指定されます。

SQL-arguments

ルーチンの CREATE ステートメント内の入力パラメーターのリストに対応します。

以下に、2 つの入力引数の積を戻す Java の例を示します。

```

public static double product( double in1, double in2 ) throws SQLException
{
    return in1 * in2;
}

```

このスカラー関数に対応する CREATE FUNCTION ステートメントは次のとおりです。

```
CREATE FUNCTION product( in1 DOUBLE, in2 DOUBLE )
  RETURNS DOUBLE
  LANGUAGE java
  PARAMETER STYLE java
  NO SQL
  FENCED THREADSAFE
  DETERMINISTIC
  RETURNS NULL ON NULL INPUT
  NO EXTERNAL ACTION
  EXTERNAL NAME 'myjar:udfclass.product'
```

上記のステートメントでは、udfclass というクラス内にメソッドがあることが前提になっています。なおこのメソッドは、myjar という Jar ID を持ったデータベース・サーバーにインストールされている JAR ファイル内に置かれます。JAR ファイルは、INSTALL_JAR システム定義プロシージャを使ってデータベース・サーバーにインストールできます。

DB2GENERAL ルーチン

PARAMETER STYLE DB2GENERAL ルーチンは、Java で作成します。

DB2GENERAL ルーチンの作成は、他のサポートされているプログラム言語でのルーチンの作成によく似ています。いったんそれらを作成して登録すると、どの言語のプログラムからでも呼び出すことができます。一般的に、ストアード・プロシージャから JDBC API を呼び出すことはできますが、UDF からそれらを読み出すことはできません。

ルーチンを Java で開発する場合、CREATE ステートメント内で PARAMETER STYLE JAVA 節を使用してルーチンを登録することを強くお勧めします。これまでどおり PARAMETER STYLE DB2GENERAL を使用して、Java ルーチン内で次のようなフィーチャーのインプリメンテーションを実現することもできます。

- 表関数
- スクラッチパッド
- DBINFO 構造へのアクセス
- 関数またはメソッドの FINAL CALL (および別個の最初の呼び出し) を行うための機能。

上記のフィーチャーのどれも使用しない PARAMETER STYLE DB2GENERAL ルーチンの場合、それを PARAMETER STYLE JAVA に移植することをお勧めします。

DB2GENERAL UDF: 他の言語の場合と同じように Java の UDF を作成して使用できますが、C の UDF と比較するといくつかの小さな違いがあります。UDF のコーディングが終了したなら、その UDF をデータベースに登録します。その後、アプリケーションでその UDF を参照することができます。

一般的に、SQL タイプの引数 *t1*、*t2*、および *t3*、戻りタイプの引数 *t4* を取る UDF を宣言した場合、次のような所定の Java シグニチャーを指定して、Java メソッドとしてその UDF を呼び出します。

```
public void name ( T1 a, T2 b, T3 c, T4 d) {.....}
```

ここで、

- *name* は、Java メソッド名
- *T1* から *T4* までは、SQL タイプの *t1* から *t4* までに対応する Java タイプ
- *a*、*b*、および *c* は、入力引数のための変数名
- *d* は、出力引数を表す変数名

例えば、INTEGER を戻し、CHAR(5)、BLOB(10K)、および DATE タイプの引数を取る sample!test3 という UDF があるとする、DB2 では、UDF の Java インプリメンテーションは必然的に以下のシグニチャーを持つことになります。

```
import COM.ibm.db2.app.*;
public class sample extends UDF {
    public void test3(String arg1, Blob arg2, String arg3,
                     int result) {... }
}
```

表関数をインプリメントする Java ルーチンには、さらに多くの引数が必要になります。変数が入力を表すのに比べて、追加の変数は結果行の各列を表しています。例えば、表関数は次のように宣言されます。

```
public void test4(String arg1, int result1,
                  Blob result2, String result3);
```

SQL NULL 値は、初期化されていない Java 変数によって示されます。これらの変数は、それらがプリミティブ・タイプの場合、ゼロ値です。それらがオブジェクト・タイプの場合、Java 規則と一致して、Java null です。SQL NULL に普通のゼロ以外を知らせるには、どんな入力引数でも関数 isNull を呼び出します。

```
{ ....
  if (isNull(1)) { /* argument #1 was a SQL NULL */ }
  else           { /* not NULL */ }
}
```

上記の例では、引数番号は 1 から始まります。以下の他の関数のように isNull() 関数は、COM.ibm.db2.app.UDF クラスから継承されます。

スカラーまたは表 UDF から結果を戻すには、次のように UDF の set() メソッドを使用します。

```
{ ....
  set(2, value);
}
```

ここで、'2' は出力引数の索引で、value は互換タイプのリテラルまたは変数です。引数の番号は、選択された出力の引数リストの索引になっています。この項の最初の例の int result 変数の索引は 4 です。2 番目の例の result1 から result3 までの索引はそれぞれ 2 から 4 までです。

UDF とストアード・プロシージャで使用される C モジュールのように、Java ルーチンでは Java 標準入出力ストリーム (System.in、System.out、および System.err) を使用できません。

ルーチンのインプリメントに使用するすべての Java クラス・ファイル (またはクラスを収容する JAR) は、sqllib/function ディレクトリー内か、またはデータベース・マネージャの CLASSPATH に指定されているディレクトリー内に置かれていなければなりません。

一般的に DB2 は照会の結果セットまたは入力の行ごとに一度 UDF を呼び出し、それを何回も繰り返します。UDF の CREATE FUNCTION ステートメント中で SCRATCHPAD が指定される場合、UDF の連続した呼び出しには何らかの「連続性」が必要であるので、Java クラスのインプリメントが呼び出しのたびにではなく、一般的に言ってステートメントの UDF 参照ごとに 1 回インスタンス化されることを DB2 は識別します。通常、それは最初の呼び出しの前にインスタンス化され、その後使用されますが、表関数ではもっと頻繁にインスタンス化されることがあります。ただし、スカラー関数か表関数のどちらかで、UDF に対して NO SCRATCHPAD が指定されている場合、UDF の呼び出しごとに新しいインスタンスがインスタンス化されます。

スクラッチパッドは、UDF の次の呼び出しまで情報を保管するのに役立つことがあります。Java および OLE UDF では、呼び出し間の連続性をもたせるためにインスタンス変数を使用するかスクラッチパッドを設定することができますが、C および C++ UDF では、スクラッチパッドを使用する必要があります。Java UDF は、COM.ibm.db2.app.UDF で入手可能な getScratchPad() および setScratchPad() 方式を使用してスクラッチパッドにアクセスします。

スクラッチパッドを使用する Java の表関数の場合、CREATE FUNCTION ステートメント上で FINAL CALL または NO FINAL CALL オプションを使用して、新しいスクラッチパッド・インスタンスをいつ取得するかを制御してください。

スクラッチパッドによって UDF の呼び出し間の連続性をもたせる機能は、DB2 スクラッチパッドまたはインスタンス変数のどちらが使用されるかにかかわらず、CREATE FUNCTION の SCRATCHPAD および NO SCRATCHPAD オプションによって制御されます。

スカラー関数の場合、全ステートメントで同じインスタンスが使用されます。

同じ UDF が複数回参照されても、照会内の Java UDF に対するすべての参照は別個に扱われることに注意してください。これは、OLE、C、および C++ の UDF でも同じです。照会の終わりに、スカラー関数に FINAL CALL オプションを指定すると、オブジェクトの close() メソッドが呼び出されます。表関数の場合、この次のサブセクションに示されているように、close() メソッドが必ず呼び出されます。UDF クラスに close() メソッドを定義していない場合、スタブ関数が引き継ぎ、イベントは無視されます。

CREATE FUNCTION ステートメントで Java UDF に ALLOW PARALLEL 節を指定する場合、DB2 は並列で UDF を評価するよう選択します。このようになる場合、別のパーティションに別個の Java オブジェクトを作成できます。各オブジェクトは、行のサブセットを受け取ります。

他の UDF のように、Java UDF では FENCED または NOT FENCED を使用することができます。NOT FENCED を使用した UDF は、データベース・エンジンのアドレス・スペース内部で実行されます。FENCED を使用した UDF は、分割されたプロセスで実行されます。Java UDF は、その組み込み処理で偶然にアドレス・スペースを破壊することはありませんが、処理を終了したり、低下させたりする場合があります。したがって、Java で作成された UDF をデバッグする場合、FENCED を使用した UDF として実行する必要があります。

DB2GENERAL ルーチンでサポートされている SQL データ・タイプ:

PARAMETER STYLE DB2GENERAL ルーチンが呼び出されると、DB2 は SQL タイプと Java タイプの変換を行います。これらのクラスのいくつかは、Java パッケージの `COM.ibm.db2.app` にあります。

表 44. DB2 SQL タイプおよび Java オブジェクト

SQL 列名	Java データ・タイプ
SMALLINT	short
INTEGER	int
BIGINT	long
REAL ¹	float
DOUBLE	double
DECIMAL(p,s)	java.math.BigDecimal
NUMERIC(p,s)	java.math.BigDecimal
CHAR(n)	java.lang.String
CHAR(n) FOR BIT DATA	COM.ibm.db2.app.Blob
VARCHAR(n)	java.lang.String
VARCHAR(n) FOR BIT DATA	COM.ibm.db2.app.Blob
LONG VARCHAR	java.lang.String
LONG VARCHAR FOR BIT DATA	COM.ibm.db2.app.Blob
GRAPHIC(n)	java.lang.String
VARGRAPHIC(n)	String
LONG VARGRAPHIC ²	String
BLOB(n) ²	COM.ibm.db2.app.Blob
CLOB(n) ²	COM.ibm.db2.app.Clob
DBCLOB(n) ²	COM.ibm.db2.app.Clob
DATE ³	String
TIME ³	String
TIMESTAMP ³	String
注:	
1. SQLDA での REAL と DOUBLE の違いは長さの値です (4 または 8)。	
2. Blob および Clob クラスは COM.ibm.db2.app パッケージ中にあります。それらのインターフェースはルーチンを組み込んで、Blob に対しては読み書きを行い、Clob には Reader および Writer となる InputStream および OutputStream を生成します。	
3. C でコード化される UDF と同様に、SQL DATE、TIME、および TIMESTAMP 値は、Java でエンコードされる ISO ストリングを使用します。	

COM.ibm.db2.app.Blob および COM.ibm.db2.app.Clob クラスの例は、LOB データ・タイプ (BLOB、CLOB、および DBCLOB) を示します。これらのクラスは、入力として渡される LOB を読み込み、出力として戻される LOB を書き込む限定インターフェースを提供します。LOB の読み込みおよび書き込みは、標準 Java I/O ストリーム・オブジェクトを通して起こります。Blob クラスの場合、`getInputStream()` および `getOutputStream()` ルーチンは、BLOB の内容を一度にバイト単位で処理する、InputStream または OutputStream オブジェクトを戻しま

す。Clob の場合、`getReader()` および `getWriter()` の各ルーチンは、CLOB または DBCLOB の内容を一度に文字単位で処理する、Reader または Writer オブジェクトを戻します。

`set()` メソッドを使用して、そのようなオブジェクトが出力として戻される場合、データベースのコード・ページ中の Java Unicode 文字を表示する目的で、コード・ページ変換が適用される場合があります。

DB2GENERAL ルーチン用の Java クラス:

このインターフェースは、JDBC 接続を組み込みアプリケーション・コンテキストに取り出すための以下のルーチンを提供します。

```
public java.sql.Connection getConnection()
```

SQL ステートメントを実行するためにこの処理を使用できます。StoredProc インターフェースの他のメソッドは、`sqllib/samples/java/StoredProc.java` ファイルにリストされています。

Java ストアド・プロシージャまたは UDF で使用できるクラス/インターフェースは、以下の 5 つです。

- `COM.ibm.db2.app.StoredProc`
- `COM.ibm.db2.app.UDF`
- `COM.ibm.db2.app.Lob`
- `COM.ibm.db2.app.Blob`
- `COM.ibm.db2.app.Clob`

DB2GENERAL Java クラス: *COM.IBM.db2.app.StoredProc*:

PARAMETER STYLE DB2GENERAL ストアド・プロシージャとして呼び出されるメソッドが含まれる Java クラスは、パブリックでなければならず、この Java インターフェースを実現するものでなければなりません。そのようなクラスを次のように宣言する必要があります。

```
public class user-STP-class extends COM.ibm.db2.app.StoredProc{ ... }
```

現在実行しているストアド・プロシージャのコンテキストでは、`COM.ibm.db2.app.StoredProc` インターフェースの継承メソッドだけを呼び出せません。例えば、ストアド・プロシージャが戻った後には、LOB 引数に対する操作（結果設定呼び出しまたは状況設定呼び出し）を実行できません。この規則に違反すると、Java 例外がスローされます。

引数関連の呼び出しは、列索引を使用して参照する列を識別します。これは、最初の引数の 1 から開始します。PARAMETER STYLE DB2GENERAL ストアド・プロシージャのすべての引数は INOUT、つまり入出力であると見なされます。

ストアド・プロシージャから例外が戻されると、データベースによって捕そくされ、SQLCODE -4302、SQLSTATE 38501 と共に呼び出し元に戻されます。JDBC `SQLException` または `SQLWarning` が特別に処理され、その SQLCODE、SQLSTATE などが呼び出しアプリケーションに逐次渡されます。

次のメソッドは、`COM.ibm.db2.app.StoredProc` クラスに関連付けられています。

```
public StoredProc() [default constructor]
```

このコンストラクターは、ストアード・プロシージャー呼び出しの前にデータベースによって呼び出されます。

```
public boolean isNull(int) throws Exception
```

この関数は、所定の索引の付いた入力引数が SQL NULL であるかどうかをテストします。

```
public void set(int, short) throws Exception
public void set(int, int) throws Exception
public void set(int, double) throws Exception
public void set(int, float) throws Exception
public void set(int, java.math.BigDecimal) throws Exception
public void set(int, String) throws Exception
public void set(int, COM.ibm.db2.app.Blob) throws Exception
public void set(int, COM.ibm.db2.app.Clob) throws Exception
```

この関数は、所定の索引の付いた出力引数を所定の値に設定します。この索引は有効な出力引数を参照し、データ・タイプは一致し、値は有効な長さで内容である必要があります。Unicode 文字のストリングは、データベース・コード・ページで表せるストリングでなければなりません。エラーがあると、例外が生じます。

```
public java.sql.Connection getConnection() throws Exception
```

この関数は、呼び出しアプリケーションとデータベースの接続を示す JDBC オブジェクトを戻します。これは、C ストアード・プロシージャーでの NULL SQLConnect() 呼び出しの結果と似ています。

DB2GENERAL Java クラス: COM.IBM.db2.app.UDF:

PARAMETER STYLE DB2GENERAL UDF として呼び出されるメソッドが含まれる Java クラスは、パブリックでなければならず、この Java インターフェースを実現するものでなければなりません。そのようなクラスを次のように宣言する必要があります。

```
public class user-UDF-class extends COM.ibm.db2.app.UDF{ ... }
```

現在実行している UDF のコンテキストでは、COM.ibm.db2.app.UDF インターフェースのメソッドだけを呼び出せます。例えば、UDF が戻った後には、結果設定呼び出しあるいは状況設定呼び出しなど、LOB 引数での操作は行えません。この規則に違反すると、Java 例外がスローされます。

引数関連の呼び出しは、列索引を使用して設定する列を識別します。これは、最初の引数の 1 から開始します。出力引数は、入力引数よりも大きな番号が付けられます。例えば、3 つの入力があるスカラー UDF の場合は、出力には索引 4 が使用されます。

UDF から例外が戻されると、データベースによって捕そくされ、SQLCODE -4302、SQLSTATE 38501 と共に呼び出し元に戻されます。

次のメソッドは、COM.ibm.db2.app.UDF クラスに関連付けられています。

```
public UDF() [default constructor]
```

このコンストラクターは、一連の UDF 呼び出しの最初にデータベースによって呼び出されます。これは、UDF への最初の呼び出しの前に行われます。

```
public void close()
```

この関数は、FINAL CALL オプションで UDF が作成された場合、UDF の計算の最後にデータベースによって呼び出されます。これは、C UDF での最終呼び出しと似ています。表関数の場合、close() を呼び出すのは、UDF メソッドに対する CLOSE 呼び出しの後 (NO FINAL CALL がコーディングされているか、またはデフォルトとして設定されている場合)、または FINAL 呼び出しの後 (FINAL CALL がコーディングされている場合) です。Java UDF クラスがこの関数を実現しない場合、ノーオペレーション・スタブはこのイベントを処理し、無視します。

```
public int getCallType() throws Exception
```

表関数の UDF メソッドは、getCallType() を使用して特定の呼び出しの呼び出しタイプを検出します。これによって次のような値が戻されます (これらの値に対するシンボル定義は、COM.ibm.db2.app.UDF クラス定義で提供されています)。

- -2 FIRST 呼び出し
- -1 OPEN 呼び出し
- 0 FETCH 呼び出し
- 1 CLOSE 呼び出し
- 2 FINAL 呼び出し

```
public boolean isNull(int) throws Exception
```

この関数は、所定の索引の付いた入力引数が SQL NULL であるかどうかをテストします。

```
public boolean needToSet(int) throws Exception
```

この関数は、所定の索引の付いた出力引数を設定する必要があるかどうかをテストします。その列が UDF 呼び出し元によって使用されていない場合、DBINFO で宣言された表 UDF についてはこのことが当てはまらない可能性があります。

```
public void set(int, short) throws Exception
public void set(int, int) throws Exception
public void set(int, double) throws Exception
public void set(int, float) throws Exception
public void set(int, java.math.BigDecimal) throws Exception
public void set(int, String) throws Exception
public void set(int, COM.ibm.db2.app.Blob) throws Exception
public void set(int, COM.ibm.db2.app.Clob) throws Exception
```

この関数は、所定の索引の付いた出力引数を所定の値に設定します。この索引は有効な出力引数を参照し、データ・タイプは一致し、値は有効な長さや内容である必要があります。Unicode 文字のストリングは、データベース・コード・ページで表せるストリングでなければなりません。エラーがあると、例外が生じます。

```
public void setSQLstate(String) throws Exception
```

この関数は、この呼び出しから SQLSTATE を戻すよう設定するために、UDF から呼び出すことができます。表 UDF は、表の終了条件を通知するために、“02000” の付いたこの関数を呼び出す必要があります。ストリングが SQLSTATE の値として受け入れられないものである場合、例外がスローされます。

```
public void setSQLmessage(String) throws Exception
```

この関数は、setSQLstate 関数と似ています。これにより、SQL メッセージの結果が設定されます。ストリングが受け入れられない (例えば、70 文字を超えている) ものである場合、例外がスローされます。

```
public String getFunctionName() throws Exception
```

この関数は、実行中の UDF の名前を返します。

```
public String getSpecificName() throws Exception
```

この関数は、実行中の UDF の特定名を返します。

```
public byte[] getDBinfo() throws Exception
```

この関数は、実行中の UDF の未処理の DBINFO 構造をバイト配列で返します。まず、DBINFO 構造をバイト配列で返すことを DBINFO オプションで宣言しておく必要があります。

```
public String getDBname() throws Exception
public String getDBauthid() throws Exception
public String getDBtbschema() throws Exception
public String getDBtbname() throws Exception
public String getDBcolname() throws Exception
public String getDBver_rel() throws Exception
public String getDBplatform() throws Exception
public String getDBapplid() throws Exception
```

これらの関数は、実行中の UDF の DBINFO 構造から該当するフィールドの値を返します。

```
public int getDBprocid() throws Exception
```

この関数は、このルーチンを直接または間接に呼び出したプロシージャのルーチン ID を返します。そのルーチン ID は、呼び出し元プロシージャの名前を検索するのに使用できる SYSCAT.ROUTINES 内の ROUTINEID 列に一致します。実行中のルーチンがアプリケーションから呼び出されると、getDBprocid() は 0 を返します。

```
public int[] getDBcodepg() throws Exception
```

この関数は、DBINFO 構造から SBCS、DBCS、およびデータベースの複合コード・ページ番号を返します。戻された整数の配列には、最初の 3 つの元素に該当する番号が入れられます。

```
public byte[] getScratchpad() throws Exception
```

この関数は、現在実行中の UDF のスクラッチパッドのコピーを返します。まず SCRATCHPAD オプションで UDF を宣言する必要があります。

```
public void setScratchpad(byte[]) throws Exception
```

この関数は、所定のバイト配列の内容で、現在実行中の UDF のスクラッチパッドを上書きします。まず SCRATCHPAD オプションで UDF を宣言する必要があります。バイト配列のサイズは、getScratchpad() が返すサイズと同じでなければなりません。

DB2GENERAL Java クラス: COM.IBM.db2.app.Lob:

このクラスは、ルーチン内で計算を行うための、Blob または Clob 一時オブジェクトを作成するユーティリティー・ルーチンを提供します。

次のメソッドは、COM.ibm.db2.app.Lob クラスに関連付けられています。

```
public static Blob newBlob() throws Exception
```

この関数は、一時的な Blob を作成します。これは、可能であれば LOCATOR を使用して実現します。

```
public static Clob newClob() throws Exception
```

この関数は、一時的な Clob を作成します。これは、可能であれば LOCATOR を使用して実現します。

DB2GENERAL Java クラス: COM.IBM.db2.app.Blob:

このクラスのインスタンスは、ルーチン入力として BLOB を表すためにデータベースから渡されますが、出力として戻されることもあります。アプリケーションはインスタンスを作成することはできませんが、実行中のルーチンの目的に追従してのみ作成できます。追従しないやり方でオブジェクトを使用すると、例外がスローされます。

次のメソッドは、COM.ibm.db2.app.Blob クラスに関連付けられています。

```
public long size() throws Exception
```

この関数は、BLOB の長さ (バイト単位) を戻します。

```
public java.io.InputStream getInputStream() throws Exception
```

この関数は、BLOB の内容を読み取るために新しい InputStream を戻します。そのオブジェクト上で、有効なシーク/マーク操作を行えます。

```
public java.io.OutputStream getOutputStream() throws Exception
```

この関数は、BLOB に何バイトか追加するために新しい OutputStream を戻します。追加したバイトは、このオブジェクトの getInputStream() 呼び出しによって作成された既存のすべての InputStream インスタンス上にすぐに反映されます。

DB2GENERAL Java クラス: COM.IBM.db2.app.Clob:

このクラスのインスタンスは、ルーチン入力として CLOB または DBCLOB を表すためにデータベースから渡されますが、出力として戻されることもあります。アプリケーションはインスタンスを作成することはできませんが、実行中のルーチンの目的に追従してのみ作成できます。追従しないやり方でオブジェクトを使用すると、例外がスローされます。

Clob インスタンスは、文字をデータベース・コード・ページとして保管します。Unicode 文字によってはこのコード・ページ形式で表せないものもあるため、変換時に例外が出されることがあります。これは、追加操作時、あるいは UDF または StoredProc set() 呼び出し時に生じる可能性があります。このことは、Java プログラマーから CLOB と DBCLOB の違いを隠すために必要です。

次のメソッドは、COM.ibm.db2.app.Clob クラスに関連付けられています。

```
public long size() throws Exception
```

この関数は、CLOB の長さ (文字単位) を戻します。

```
public java.io.Reader getReader() throws Exception
```

この関数は、CLOB または DBCLOB の内容を読み取るために新しい Reader を戻します。そのオブジェクト上で、有効なシークマーク操作を行えます。

```
public java.io.Writer getWriter() throws Exception
```

この関数は、この CLOB または DBCLOB に何文字か追加するために新しい Writer を戻します。追加した文字は、このオブジェクトの GetReader() 呼び出しによって作成された既存のすべての Reader インスタンス上にすぐに反映されます。

Java ルーチンに対するデータ・タイプ ARRAY のパラメーターの引き渡し

DB2 9.5 では、データ・タイプ ARRAY のパラメーターを Java プロシージャとの間で受け渡す機能がサポートされています。

このタスクについて

アプリケーションと Java ストアド・プロシージャの間で以下の受け渡しを行う場合、配列を引き渡して使用するようインプリメントできます。

- 1 つのパラメーターを使用して、プロシージャに大量の同種データを引き渡す場合。
- 1 つのパラメーターだけを使用して、プロシージャに同じデータ・タイプの可変数の入力を引き渡す場合。

例えば、あるクラスの学生の名前すべてを、学生数を把握していなくても、1 つのパラメーターを使用してプロシージャに引き渡すことができます。ARRAY データ・タイプを使用しないでこれを行うとすると、1 つの学生名ごとに 1 つのパラメーターが必要となります。

手順

タイプ ARRAY のパラメーターを引き渡すには、以下のようにします。

1. ARRAY データ・タイプを事前に定義しておく必要があります。配列タイプを定義するには、CREATE TYPE ステートメントを実行しなければなりません。
2. プロシージャ定義には、定義されたタイプのパラメーターを含める必要があります。例えば、IntArray という名前の ARRAY データ・タイプを CREATE TYPE ステートメントを実行して作成する場合、このタイプのパラメーターをプロシージャに引き渡すには、以下のようにします。

例

```
CREATE PROCEDURE inArray (IN input IntArray)
LANGUAGE JAVA
PARAMETER STYLE JAVA
EXTERNAL NAME 'MyProcs:MyArrayProcs!inArray';
```

このプロシージャ定義の場合、配列パラメーターは java.sql.Array と入力します。このプロシージャでは、以下の例に示されているように、引数は getArray() メソッドを使用して Java 配列にマップされます。配列では、int (または他のプリミティブ型) ではなく Integer を使用することに注意してください。

```
static void inArray(java.sql.Array input)
{
    Integer[] inputArr = (Integer [])input.getArray();
```



```

int sum = 0;
for(int i=0, i < inputArr.length; i++)
{
sum += inputArr[i];
}
}

```

JDBC プロシージャからの結果セットの戻り

呼び出し元のルーチンまたはアプリケーションに結果セットを戻す JDBC プロシージャを開発することができます。JDBC プロシージャでは、結果セットの戻りは `ResultSet` オブジェクトで処理します。

JDBC プロシージャから結果セットを戻すには、次のようにします。

1. 戻される各結果セットごとに、プロシージャの宣言内に `ResultSet[]` タイプのパラメーターを組み込みます。例えば、以下の関数シグニチャーは `ResultSet` オブジェクトの配列を受け入れます。

```

public static void getHighSalaries(
    double inSalaryThreshold,      // double input
    int[] errorCode,                // SQLCODE output
    ResultSet[] rs)                // ResultSet output

```

2. 次のようにして、呼び出し側のデータベース接続をオープンします (接続オブジェクトを使用して)。

```

Connection con =
    DriverManager.getConnection("jdbc:default:connection");

```

3. 結果セットを生成する SQL ステートメントを準備します (`PreparedStatement` オブジェクトを使用して)。以下の例では、準備の後で、照会ステートメント内のパラメーター・マーカーの値に対して、入力変数 (`inSalaryThreshold` という名前。上記の関数シグニチャーの例を参照) が割り当てられます。パラメーター・マーカーは `?` か、またはコロンの後に名前が続く形 (`:name`) で表されます。

```

String query =
    "SELECT name, job, CAST(salary AS DOUBLE) FROM staff " +
    " WHERE salary > ? " +
    " ORDER BY salary";

```

```

PreparedStatement stmt = con.prepareStatement(query);
stmt.setDouble(1, inSalaryThreshold);

```

4. ステートメントを実行する。

```
rs[0] = stmt.executeQuery();
```
5. プロシージャの本体を終了します。

ストアード・プロシージャからの結果セットを受け入れるクライアント・アプリケーションまたは呼び出し元ルーチンをまだ開発していなければ、開発してください。

SQLJ プロシージャからの結果セットの戻り

呼び出し元のルーチンまたはアプリケーションに結果セットを戻す SQLJ プロシージャを開発することができます。SQLJ プロシージャでは、結果セットの戻りは `ResultSet` オブジェクトで処理します。

SQLJ プロシージャから結果セットを戻すには、次のようにします。

1. 照会データを処理するイテレーター・クラスを宣言します。以下に例を示します。

```
#sql iterator SpServerEmployees(String, String, double);
```

2. 戻される各結果セットごとに、プロシーチャーの宣言内に `ResultSet[]` タイプのパラメーターを組み込みます。例えば、以下の関数シグニチャーは `ResultSet` オブジェクトの配列を受け入れます。

```
public static void getHighSalaries(  
    double inSalaryThreshold, // double input  
    int[] errorCode,          // SQLCODE output  
    ResultSet[] rs)          // ResultSet output
```

3. イテレーター・オブジェクト・インスタンスを生成します。以下に例を示します。

```
SpServerEmployees c1;
```

4. 結果セットを生成する SQL ステートメントをイテレーターに割り当てます。以下の例では、ホスト変数 (`inSalaryThreshold` という名前。上記の関数シグニチャーの例を参照) が照会の `WHERE` 節内で使用されています。

```
#sql c1 = {SELECT name, job, CAST(salary AS DOUBLE)  
          FROM staff  
          WHERE salary > :inSalaryThreshold  
          ORDER BY salary};
```

5. 以下のように、ステートメントを実行して結果セットを入手します。

```
rs[0] = c1.getResultSet();
```

プロシーチャーからの結果セットを受け入れるクライアント・アプリケーションまたは呼び出し元ルーチンをまだ開発していなければ、開発してください。

JDBC アプリケーションおよびルーチンでのプロシーチャーの結果セットの受け取り

JDBC ルーチンまたはアプリケーションから呼び出したプロシーチャーから、結果セットを受け取ることができます。

JDBC ルーチンまたはアプリケーションからのプロシーチャー結果セットを受け入れるには、次のようにします。

1. 次のようにして、データベース接続をオープンします (接続オブジェクトを使用して)。

```
Connection con =  
    DriverManager.getConnection("jdbc:db2:sample", userid, passwd);
```

2. 結果セットを戻すプロシーチャーを呼び出す `CALL` ステートメントを準備します (`CallableStatement` オブジェクトを使用して)。以下の例では、`GET_HIGH_SALARIES` という名前のプロシーチャーが呼び出されます。準備が終わったら、次に前のステートメント内のパラメーター・マーカ―の値に対して、入力変数 (`inSalaryThreshold` という名前。これは、プロシーチャーに渡される数値) を割り当てます。パラメーター・マーカ―は `?` か、またはコロンの後に名前が続く形 (`:name`) で表されます。

```
String query = "CALL GET_HIGH_SALARIES(?)";
```

```
CallableStatement stmt = con.prepareCall(query);  
stmt.setDouble(1, inSalaryThreshold);
```

3. 次のようにしてプロシーチャーを呼び出します。

```
stmt.execute();
```

4. 次のように、CallableStatement オブジェクトの `getResultSet()` メソッドを使用して、プロシーチャーから最初の結果セットを受け入れて、`fetchAll()` メソッドを使用して結果セットから行を取り出します。

```
ResultSet rs = stmt.getResultSet();

// Result set rows are fetched and printed to screen.
while (rs.next())
{
    r++;
    System.out.print("Row: " + r + ": ");
    for (int i=1; i <= numOfColumns; i++)
    {
        System.out.print(rs.getString(i));
        if (i != numOfColumns)
        {
            System.out.print(", ");
        }
    }
    System.out.println();
}
```

5. 複数の結果セットの場合、CallableStatement オブジェクトの `getNextResultSet()` メソッドを使用して、次の結果セットを読み取れるようにします。次に、ResultSet オブジェクトが現行結果セットを受け入れた前のステップのプロセスを反復してから、結果セットの行を取り出します。以下に例を示します。

```
while (callStmt.getMoreResults())
{
    rs = callStmt.getResultSet()

    ResultSetMetaData stmtInfo = rs.getMetaData();
    int numOfColumns = stmtInfo.getColumnCount();
    int r = 0;

    // Result set rows are fetched and printed to screen.
    while (rs.next())
    {
        r++;
        System.out.print("Row: " + r + ": ");
        for (int i=1; i <= numOfColumns; i++)
        {
            System.out.print(rs.getString(i));
            if (i!= numOfColumns)
            {
                System.out.print(", ");
            }
        }
        System.out.println();
    }
}
```

6. 次のように `close()` メソッドを使用して、ResultSet オブジェクトをクローズします。

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

SQLJ アプリケーションおよびルーチンでのプロシージャの結果セットの受け取り

SQLJ ルーチンまたはアプリケーションから呼び出したプロシージャから、結果セットを受け取ることができます。

SQLJ ルーチンまたはアプリケーションからのプロシージャ結果セットを受け取るには、次のようにします。

1. 次のようにして、データベース接続をオープンします (接続オブジェクトを使用して)。

```
Connection con =
    DriverManager.getConnection("jdbc:db2:sample", userid, passwd);
```

2. 次のように、デフォルト・コンテキストを設定します (DefaultContext オブジェクトを使用します)。

```
DefaultContext ctx = new DefaultContext(con);
DefaultContext.setDefaultContext(ctx);
```

3. 次のように、実行コンテキストを設定します (ExecutionContext オブジェクトを使用します)。

```
ExecutionContext execCtx = ctx.getExecutionContext();
```

4. 結果セットを戻すプロシージャを呼び出します。以下の例では、GET_HIGH_SALARIES という名前のプロシージャが呼び出して、入力変数 (inSalaryThreshold という名前) を渡されます。

```
#sql {CALL GET_HIGH_SALARIES(:in inSalaryThreshold, :out outErrorCode)};
```

5. ResultSet オブジェクトを宣言してから、ExecutionContext オブジェクトの getNextResultSet() メソッドを使用して、プロシージャから結果セットを受け入れます。複数の結果セットの場合、getNextResultSet() 呼び出しをループ構造に入れます。プロシージャから戻された結果セットはすべて、ループ反復を作成します。そのループ内では、結果セット行メソッドを取り出してから、結果セット・オブジェクトをクローズする (ResultSet オブジェクトの close() メソッドを使用して) ことができます。以下に例を示します。

```
ResultSet rs = null;

while ((rs = execCtx.getNextResultSet()) != null)
{
    ResultSetMetaData stmtInfo = rs.getMetaData();
    int numOfColumns = stmtInfo.getColumnCount();
    int r = 0;

    // Result set rows are fetched and printed to screen.
    while (rs.next())
    {
        r++;
        System.out.print("Row: " + r + ": ");
        for (int i=1; i <= numOfColumns; i++)
        {
            System.out.print(rs.getString(i));
            if (i != numOfColumns)
            {
                System.out.print(", ");
            }
        }
        System.out.println();
    }
}
```

```

    }
    rs.close();
}

```

Java ルーチンに関する制約事項

Java ルーチンには以下の制約事項が適用されます。

- JAR ファイル内の Java ルーチン・コードをデータベース・サーバー・ファイル・システムにデプロイするのに使用するシステム定義プロシージャ `install_jar` は、DB2 Universal JDBC ドライバーの使用時には Java データベース・アプリケーションで呼び出すことはできません。このドライバーは、このプロシージャではサポートされていません。

推奨されている代替の方法は、DB2 コマンド行プロセッサを使用することです。

- PROGRAM TYPE MAIN 節は、指定されている PARAMETER STYLE 節値に関係なく、Java ルーチンの CREATE PROCEDURE ステートメントまたは CREATE FUNCTION ステートメントではサポートされません。
- 以下のフィーチャーは、パラメーター・スタイル JAVA ではサポートされていません。
 - 表関数
 - 関数内のスクラッチパッド
 - 関数内の DBINFO 構造へのアクセス
 - 関数内の FINAL CALL 呼び出し

こうしたフィーチャーが必要な場合に推奨される代替の方法は、パラメーター・スタイル DB2GENERAL を使用して Java 関数を作成するか、C または C++ プログラミング言語のどちらかを使用して関数を作成する方法です。

- Java ルーチンからの Java Native Interface (JNI) 呼び出しはサポートされていません。

Java ルーチンから C または C++ コードを呼び出す必要がある場合は、別個に定義された C または C++ ルーチンを呼び出さなければなりません。

- NOT FENCED Java ルーチンは現在サポートされていません。NOT FENCED として定義される Java ルーチンは、FENCED THREADSAFE として定義されているかのように呼び出されます。
- Java ストアード・プロシージャは、プロパティ・ファイルなどのシステム・リソース以外のリソースに依存するものであってはなりません。システム・リソース以外のリソースに依存する Java ストアード・プロシージャを呼び出した場合、それらのリソースはロードされず、エラーは戻されません。

Java の表関数実行モデル

Java で作成されて PARAMETER STYLE DB2GENERAL を使用する表関数の場合、DB2 での特定のステートメントの処理のさまざまな時点で何が起きるか分かっていることが大切です。以下の表は、通常の場合のそれに関する詳細を示しています。NO FINAL CALL の場合と FINAL CALL の場合の両方が取り上げられており、どちらも SCRATCHPAD が指定されていると想定しています。

スキヤンの時点	NO FINAL CALL LANGUAGE JAVA SCRATCHPAD	FINAL CALL LANGUAGE JAVA SCRATCHPAD
表関数に対する最初の OPEN の前	<ul style="list-style-type: none"> 呼び出しなし。 	<ul style="list-style-type: none"> クラス・コンストラクターが呼び出される (すなわち、新しいスクラッチパッド)。 FIRST 呼び出しで UDF メソッドが呼び出される。 コンストラクターが、クラスおよびスクラッチパッド変数を初期化する。メソッドが Web サーバーに接続する。
表関数に対する各 OPEN 時	<ul style="list-style-type: none"> クラス・コンストラクターが呼び出される (すなわち、新しいスクラッチパッド)。 OPEN 呼び出しで UDF メソッドが呼び出される。 コンストラクターが、クラスおよびスクラッチパッド変数を初期化する。メソッドが Web サーバーに接続し、Web データのスキヤンをオープンする。 	<ul style="list-style-type: none"> OPEN 呼び出しで UDF メソッドがオープンされる。 メソッドが、必要な Web データのスキヤンをオープンする。(スクラッチパッドに保管されるものに応じて、CLOSE 位置変更後の再オープンを避けることができる。)
新しい行の表関数データに対する各 FETCH 時	<ul style="list-style-type: none"> FETCH 呼び出しで UDF メソッドが呼び出される。 メソッドが、次の行のデータまたは EOT を取り出して戻す。 	<ul style="list-style-type: none"> FETCH 呼び出しで UDF メソッドが呼び出される。 メソッドが、新しい行のデータまたは EOT を取り出して戻す。
表関数に対する CLOSE 時	<ul style="list-style-type: none"> CLOSE 呼び出しで UDF メソッドが呼び出される。 close() メソッドがクラスに対してあれば、呼び出される。 メソッドがその Web スキヤンをクローズし、Web サーバーから切断する。close() は必要とされない。 	<ul style="list-style-type: none"> CLOSE 呼び出しで UDF メソッドが呼び出される。 メソッドはスキヤンの最上部に位置変更するか、スキヤンをクローズする。持続されるどんな状態でも、スクラッチパッドに保管できる。
表関数に対する最後の CLOSE の後	<ul style="list-style-type: none"> 呼び出しなし。 	<ul style="list-style-type: none"> FINAL 呼び出しで UDF メソッドが呼び出される。 close() メソッドがクラスに対してあれば、呼び出される。 メソッドは Web サーバーから切断する。close() メソッドは必要とされない。

注:

- 「UDF メソッド」とは、UDF をインプリメントした Java クラス・メソッドのことです。これは、CREATE FUNCTION ステートメントの EXTERNAL NAME 節で識別されるメソッドです。
- NO SCRATCHPAD が指定された表関数では、UDF メソッドの呼び出しはこの表で示されているとおりですが、ユーザーはスクラッチパッドによる連続性を求めないために、クラス・コンストラクターが DB2 によって呼び出され、各呼び

出しの前に新しいオブジェクトがインスタンス化されます。NO SCRATCHPADが指定された (したがって連続性がない) 表関数が役立つかどうかは不明ですが、それらはサポートされています。

Java ルーチンの作成

Java ルーチンの作成は、以下から構成されます。

- DB2 データベース・サーバーでルーチンを定義する CREATE ステートメントの実行
- ルーチン定義に対応するルーチン・インプリメンテーションの開発

Java ルーチンを作成する方法は、以下に従ってください。

- IBM Data Studioを使用する
- IBM Optim Development Studioを使用する
- IBM Rational Application Developer の DB2 ルーチン開発フィーチャーを使用する
- DB2 コマンド・ウィンドウを使用する

一般的に言って、IBM Data Studioを使用して Java ルーチンを作成するのが最も簡単ですが、IBM Rational Application Developer が提供する統合 Java 開発環境の中で Java ルーチンを作成する機能を好む開発者も多くいます。こうしたグラフィック・ツールが使用できない場合は、DB2 コマンド・ウィンドウがコマンド行インターフェースを介して同様のサポートを提供します。

前提条件

- 283 ページの『第 7 章 Java ルーチン』を確認します。
- DB2 バージョン 9 サーバー (インスタンスおよびデータベースを含む) にアクセスしていることを確認します。
- オペレーティング・システムのバージョン・レベルが、DB2 データベース製品でサポートされているものであることを確認します。
- 284 ページの『サポートされる Java ルーチン開発ソフトウェア』のバージョン・レベルが、Java ルーチン開発にサポートされるものであることを確認します。
- 286 ページの『Java ルーチン用のドライバーの指定』開発が有効であることを確認します。
- CREATE PROCEDURE または CREATE FUNCTION ステートメントを実行する権限。

Java ルーチンに関連した制約事項のリストについては、以下を参照してください。

- 307 ページの『Java ルーチンに関する制約事項』

以下のいずれかの手順を使用して、Java ルーチンを作成します。

- IBM Data StudioJava ルーチンを作成する
- Rational Application Developer を使用して Java ルーチンを作成する
- 310 ページの『コマンド行からの Java ルーチンの作成』

コマンド行からの Java ルーチンの作成

Java クラスを参照するプロシージャまたは関数は、他のインプリメンテーションを持つ外部ルーチンと類似の方法で作成されます。このタスクはいくつかのステップで構成されますが、それにはルーチンの CREATE ステートメントの公式化、ルーチンのインプリメンテーションのコーディングおよびコンパイル (変換)、DB2 データベース・サーバーへの Java クラスのデプロイメントが含まれます。

Java ルーチンをインプリメントするのは、以下のような場合です。

- データベースにアクセスするルーチンや、データベースの外部でアクションを実行するルーチンに複雑なロジックをカプセル化したい場合。
- 複数のアプリケーション、CLP、他のルーチン (プロシージャ、関数 (UDF)、メソッド)、トリガーのいずれかから、カプセル化されたロジックを呼び出す必要がある場合。
- そのロジックのコーディングに Java および JDBC または SQLJ アプリケーション・プログラミング・インターフェースのいずれかが最も使いやすいと感じる場合。

前提条件

- 283 ページの『第 7 章 Java ルーチン』を確認します。
- DB2 バージョン 9 データベース・サーバー (インスタンスおよびデータベースを含む) にアクセスしていることを確認します。
- オペレーティング・システムのバージョン・レベルが、DB2 データベース製品でサポートされているものであることを確認します。
- 284 ページの『サポートされる Java ルーチン開発ソフトウェア』のバージョン・レベルが、Java ルーチン開発にサポートされるものであることを確認します。
- 286 ページの『Java ルーチン用のドライバーの指定』開発が有効であることを確認します。
- CREATE PROCEDURE または CREATE FUNCTION ステートメントを実行する権限。

手順

1. ルーチン・ロジックを Java でコーディングします。
 - Java ルーチンおよび Java ルーチン・フィーチャーの一般情報については、『前提条件』のセクションで参照されているトピックを参照してください。
 - ルーチン・パラメーター・シグニチャーが、以下のいずれかのサポートされるパラメーター・スタイルを使用してインプリメントされている必要があります。すべての Java ルーチンにパラメーター・スタイル JAVA を使用することを強くお勧めします。パラメーター・シグニチャーおよびパラメーター・インプリメンテーションの詳細については、以下を参照してください。
 - 290 ページの『Java ルーチンのパラメーター』
 - 291 ページの『パラメーター・スタイル JAVA プロシージャ』
 - 292 ページの『PARAMETER STYLE JAVA の Java 関数およびメソッド』

- 変数を Java データベース・アプリケーションに対するのと同じ方法で宣言します。DB2 SQL データ・タイプにマップするデータ・タイプを正しく使用するように注意してください。DB2 と Java データ・タイプとの間のデータ・タイプ・マッピングの詳細については、以下を参照してください。
 - 「Java アプリケーションの開発」内の『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』
 - ルーチン・ロジックを組み込みます。ルーチン・ロジックは、Java プログラミング言語でサポートされる任意のコードで構成できます。これには、Java データベース・アプリケーションの場合と同じ方法で SQL ステートメントの実行を含めることもできます。Java コードでの SQL ステートメントの実行の詳細については、以下を参照してください。
 - 「Java アプリケーションの開発」内の『SQL を実行するための JDBC インターフェース』
 - 「Java アプリケーションの開発」内の『SQLJ アプリケーションでの SQL ステートメントの実行』
 - ルーチンがプロシージャで、ルーチンの呼び出し元に結果セットを戻す場合、結果セット用のパラメーターは必要ありません。Java ルーチンから結果セットを戻す方法の詳細については、以下を参照してください。
 - 303 ページの『JDBC プロシージャからの結果セットの戻り』
 - 303 ページの『SQLJ プロシージャからの結果セットの戻り』
 - ルーチンの末尾にルーチンの戻り値を設定します。
2. Java クラス・ファイルまたは Java クラス・ファイルのコレクションを含む JAR ファイルを生成するためのコードを作成します。Java ルーチン・コードの構築方法については、以下を参照してください。
- 「Java アプリケーションの開発」内の『JDBC ルーチンの構築』
 - 「Java アプリケーションの開発」内の『SQL ルーチンの構築』
3. クラス・ファイルを DB2 データベース・サーバーにコピーするか、または JAR ファイルを DB2 データベース・サーバーにインストールします。これを行う方法については、以下を参照してください。
- 316 ページの『Java ルーチン・クラス・ファイルの DB2 データベース・サーバーへのデプロイ』
 - 318 ページの『データベース・サーバーでの JAR ファイル管理』

DB2 ルーチンに関連したクラス・ファイルは、*function* ディレクトリーに保管することをお勧めします。*function* ディレクトリーの詳細については、CREATE PROCEDURE ステートメントまたは CREATE FUNCTION ステートメントのいずれかの EXTERNAL 節に関連する情報を参照してください。

ライブラリーをサーバー上の別のディレクトリーにコピーすることもできますが、ルーチンを正常に呼び出すには、ライブラリーの完全修飾パス名をメモしておく必要があります。次のステップでこれが必要になるからです。

4. 該当するルーチン・タイプの CREATE ステートメント (CREATE PROCEDURE または CREATE FUNCTION) を動的または静的に実行します。
- LANGUAGE 節に JAVA を指定します。

- PARAMETER STYLE 節に、ルーチン・コードでインプリメントした有効なパラメーター・スタイルの名前を指定します。必要とするフィーチャーが PARAMETER STYLE DB2GENERAL が使用される場合にのみサポートされるのではない限り、PARAMETER STYLE JAVA を使用することを強くお勧めします。
- EXTERNAL 節に、ルーチンと関連した JAR ファイルまたは Java クラスの名前を指定します。そのためには、以下のいずれかの値を使用します。
 - Java クラス・ファイルの完全修飾パス名。
 - function ディレクトリーを基準にしたルーチン Java クラス・ファイルの相対パス名。
 - データベース・サーバー上の、Java クラスを含む JAR ファイルの JAR ファイル ID。

EXTERNAL 節に JAR ファイル ID およびクラス、完全修飾パス名、または相対パス名を指定しない場合、DB2 はデフォルトで、function ディレクトリー内でライブラリーを探します。

- ルーチンがプロシージャーの場合に、DYNAMIC RESULT SETS に数値を指定すると、呼び出し元に 1 つ以上の結果セットが戻されます。
- ルーチンの特徴を指定するために使用する CREATE ステートメントで、その他のデフォルト以外の節値を指定する。

Java ルーチン呼び出しには、337 ページの『第 10 章 ルーチンの呼び出し』を参照してください。

Java ルーチン・コードのビルド

Java ルーチンのインプリメンテーション・コードを作成したなら、ルーチン・アセンブリーをデプロイしてルーチン呼び出しできるようにするには、その前にインプリメンテーション・コードをビルドする必要があります。Java ルーチンのビルドに必要なステップは、外部ルーチンのビルドに必要なステップに似ていますが、いくつかの点で異なります。

Java ルーチンをビルドするには、次のいくつかの方法があります。

- IBM Data Studioに付属するグラフィック・ツールを使用する
- IBM Optim Development Studioに組み込まれているグラフィック・ツールを使用する
- IBM Rational Application Developer に組み込まれているグラフィック・ツールを使用する
- DB2 サンプル・ビルド・スクリプトを使用する
- DB2 コマンド・ウィンドウからコマンドを入力する

グラフィック・ツールおよび DB2 は、さまざまなオペレーティング・システムに合った Java ルーチンをさまざまな設定で構築するためにカスタマイズできます。ルーチン用のサンプル・ビルド・スクリプトおよびバッチ・ファイルは、サポートされるデフォルト開発ソフトウェアを使用することにより、特定のオペレーティング・システムのための DB2 サンプル・ルーチン (プロシージャーおよびユーザー定義関数) を作成することを目的として設計されています。

JDBC および SQLJ で作成された Java ルーチン用の DB2 サンプル・ビルド・スクリプトおよびバッチ・ファイルには個別セットがあります。一般に、Java ルーチンをビルドするには、グラフィック・ツールまたはビルド・スクリプト (必要に応じて容易に変更できる) を使用するのが最も簡単な方法ですが、DB2 コマンド・ウィンドウからルーチンをビルドする方法も知っておくと、いろいろと役に立ちます。

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 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. 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します。
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。

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・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・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。

Java (SQLJ) ルーチンのコンパイルとリンクのオプション

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

blsqljs.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。blsqljs.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 blsqljs.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 blsqljs.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ライズド・プロファイルを指定します。

Java ルーチン・クラス・ファイルの DB2 データベース・サーバーへのデプロイ

呼び出し時に、ルーチンが配置されていて、ロードと実行ができるようにするためには、DB2 データベース・サーバー・ファイル・システムに Java ルーチン・インプリメンテーションがデプロイされている必要があります。

個々の Java クラス・ファイルには、1 つ以上の Java ルーチン・インプリメンテーションを含めることができます。複数の Java ルーチン・インプリメンテーションが含まれる Java クラス・ファイルを JAR ファイルと一緒に集めることができます。ルーチンをインプリメントするのに使用する Java クラス・ファイルは、DB2 データベースにインストールした JAR ファイルに置かなければなりません。

前提条件:

- ルーチン・クラスのデプロイ先の DB2 データ・サーバーを識別します。
- UNIX オペレーティング・システムで、DB2 インスタンス所有者のユーザー ID を識別します。不明の場合は、データベース管理者に問い合わせてください。

以下のようにして、それぞれの Java ルーチン・クラス・ファイルをデプロイします。

- DB2 function ディレクトリーに Java ルーチン・クラスをコピーします。

UNIX オペレーティング・システムでは、function ディレクトリーは *install_path/function* として定義されています (*install_path* は DB2 データベース・マネージャーのインストール・パス)。例えば、*\$HOME/sqlllib/function* では、*\$HOME* はインスタンス所有者のホーム・ディレクトリーです。

Windows オペレーティング・システムでは、function ディレクトリーは *instance_profile_path\function* として定義されています (*instance_profile_path* は *db2icrt* (インスタンスの作成) コマンドで指定されたパス)。インスタンス・プロファイル・パス名は、以下のように *db2set* コマンドを発行することによって検出できます。

```
db2set DB2INSTPROF
```

例: C:\Documents and Settings\All Users\Application Data\IBM\DB2\db2copy1\function

クラスを Java パッケージの一部として宣言する場合は、function ディレクトリーの中にクラスの完全修飾名と一致するサブディレクトリーを作成し、それぞれのサブディレクトリーに対応するクラス・ファイルを配置します。例えば、Linux オペレーティング・システムで *ibm.tests.test1* というクラスを作成する場合は、対応する Java バイトコード・ファイル (*test1.class*) を *\$HOME/sqlllib/function/ibm/tests* (*\$HOME* はインスタンス所有者のホーム・ディレクトリー) に保管します。

以下のようにして、Java ルーチン・クラス・ファイルが含まれている JAR ファイルをデプロイします。

- Java ルーチン・クラス・ファイルが含まれている JAR ファイルを DB2 データベース・サーバー・ファイル・システムにインストールする必要があります。

Java ルーチン・クラス・ファイルをデプロイしてから、**CREATE** ステートメントを実行してデータベースにルーチンを定義すると、ルーチンを呼び出すことができます。

従属クラスを持つ Java ルーチン・クラス・ファイルの DB2 データベース・サーバーへのデプロイ

Java ルーチン・クラス・ファイルに、標準 Java クラスまたは DB2 クラスの一部ではないクラスへの従属性がある場合、上記のセクションで示したステップを各従属クラスに対して繰り返します。

別の方法として、従属クラスを検出するために **CLASSPATH** 環境変数内のディレクトリーを検索するよう DB2 データベースを構成することもできます。Windows オペレーティング・システムの場合、データベース・サーバーは **CLASSPATH** システム環境変数内で、指定されたディレクトリーを自動的に検索します。UNIX オペレーティング・システムでは、**DB2ENVLIST** 環境変数の一部としてテキスト "CLASSPATH" が指定されている場合は、データベース・サーバーはインスタンス

所有者の **CLASSPATH** 環境変数を検索します。 **CLASSPATH** 環境変数に頼るのではなく、従属クラスをインストールすることを強くお勧めします。

データベース・サーバーでの JAR ファイル管理

Java ルーチンのクラス・ファイルを含む JAR ファイルをデプロイするには、JAR ファイルを DB2 データベース・サーバーにインストールする必要があります。これは、DB2 データベース・サーバー上で JAR ファイルのインストール、置き換え、または除去を行うシステム定義ルーチンを使用して、IBM Data Server Client から行われます。

DB2 インスタンス内の JAR ファイルをインストール、置換、または除去するには、次のように DB2 に付属しているストアード・プロシージャを使用します。

インストール

```
sqlj.install_jar( jar-url, jar-id )
```

注: 呼び出し元 `sqlj.install_jar` の許可 ID によって保持される特権には、少なくとも以下のいずれか 1 つが含まれていなければなりません。

- 明示的または暗黙的に指定されたスキーマの **CREATEIN** 特権
- **DBADM** 権限

置換

```
sqlj.replace_jar( jar-url, jar-id )
```

除去

```
sqlj.remove_jar( jar-id )
```

- *jar-url*: インストールまたは置換の対象の JAR ファイルが置かれている URL を指定します。サポートされる URL 体系は、'file:' だけです。
- *jar-id*: 128 バイトまでの長さのユニーク・ストリング ID。これは、*jar-url* ファイルに関連したデータベース内の JAR ID を指定します。

注: ストアード・プロシージャ `sqlj.install_jar` および `sqlj.remove_jar` をアプリケーションから呼び出すときは、さらに別のパラメーターが付きます。それは、指定の JAR ファイル内でのデプロイメント記述子の使用を指示する整数値です。現在、デプロイメント・パラメーターはサポートされていないので、非ゼロ値を指定して呼び出してもすべてリジェクトされてしまいます。

以下に、前記の JAR ファイル管理ストアード・プロシージャの使用方法の例をいくつか示します。

パス `/home/bob/bobsjar.jar` に置かれている JAR を **MYJAR** としてデータベースに登録するには、次のようにします。

```
CALL sqlj.install_jar( 'file:/home/bob/bobsjar.jar', 'MYJAR' )
```

以降、SQL コマンドが `bobsjar.jar` ファイルを使用する際には、**MYJAR** という名前で参照します。

MYJAR を、特定の更新済みクラスをもった別の JAR に置き換えるには、次のようにします。

```
CALL sqlj.replace_jar( 'file:/home/bob/bobsnewjar.jar', 'MYJAR' )
```


データベース・カタログから MYJAR を除去するには、次のようにします。

```
CALL sqlj.remove_jar( 'MYJAR' )
```

注: Windows オペレーティング・システムでは、DB2 は *DB2INSTPROF* インスタンス固有のレジストリー設定によって指定されたパスに JAR ファイルを保管します。JAR ファイルをインスタンス内のユニーク・ファイルにするには、そのインスタンスの *DB2INSTPROF* での固有値を指定する必要があります。

Java ルーチン・クラスの更新

Java ルーチンのロジックを変更する場合、ルーチン・ソース・コードを更新し、コードをコンパイル (変換) してから、DB2 データベース・サーバーにデプロイされる Java クラスまたは JAR ファイルのバージョンを更新する必要があります。DB2 データベース・マネージャーが Java ルーチンの新規バージョンを必ず使用するようにするには、Java クラスの新規バージョンをメモリーにロードするシステム定義プロシージャを実行する必要があります。

Java ルーチン・クラスを更新するには、以下のようになります。

1. 新規の Java クラスまたは JAR ファイルを DB2 データベース・サーバーにデプロイする。
2. fenced ルーチンに対して以下のシステム定義プロシージャを実行する。

```
CALL SQLJ.REFRESH_CLASSES()
```

これで、DB2 データベース・マネージャーは次のコミットまたはロールバック操作で新しいクラスをメモリーにロードします。

unfenced ルーチンの場合、このステップは通用しません。unfenced ルーチンの場合、Java ルーチン・クラスの新しいバージョンがロードされて使用されるようにするには、DB2 データベース・マネージャーを明示的に停止してから再始動する必要があります。

Java ルーチン・クラスを更新した後に上述のステップを実行しないと、DB2 データベース・マネージャーは以前のバージョンのクラスを使用し続けます。

Java (JDBC) ルーチンの例

JDBC アプリケーション・プログラミング・インターフェースを使用する Java ルーチンを開発する際に、CREATE ステートメントと Java ルーチン・コードをどのようにすればよいかという感覚をつかむと役に立ちます。以下のトピックには、Java プロシージャおよび関数の例が含まれています。

- Java (JDBC) プロシージャの例
- XML フィーチャーを持つ Java (JDBC) プロシージャの例
- Java (JDBC) 関数の例

例: Java (JDBC) プロシージャでの配列データ・タイプ

配列データ・タイプを使った Java ルーチンの例。

以下の例では、配列データ・タイプの IN および OUT パラメーターを伴う Java ルーチンのスケルトンを示しています。

```
CREATE TYPE phonenumbers AS VARCHAR(20) ARRAY[10] %
CREATE PROCEDURE javaproц( IN in1 phonenumbers,
                           OUT out1 phonenumbers)
    LANGUAGE java
    PARAMETER STYLE java
    FENCED THREADSAFE
    EXTERNAL NAME 'myjar:stpclass.javastp' %
import java.sql.Array;

public static void javaproц(Array input, Array[] output)
{
    output[0] = input;
}
```

例: Java (JDBC) プロシージャでの XML および XQuery サポート

Java プロシージャの基本、JDBC アプリケーション・プログラミング・インターフェース (API) を使用した Java でのプログラミング、および XQuery を理解したなら、XML データを照会する Java プロシージャの作成および使用を始めることができます。

ここでの Java プロシージャの例では、以下について示します。

- パラメーター・スタイル JAVA プロシージャの CREATE PROCEDURE ステートメント
- パラメーター・スタイル JAVA プロシージャのソース・コード
- データ・タイプ XML の入出力パラメーター
- 照会での XML 入力パラメーターの使用
- XQuery の結果、XML 値の出力パラメーターへの割り当て
- SQL ステートメントの結果、XML 値の出力パラメーターへの割り当て

前提条件

Java プロシージャの例を使用した作業を開始する前に、以下のトピックを参照することもできます。

- Java ルーチン
- ルーチン
- Java ルーチン・コードのビルド

この下の例では、xmlDataTable という名前の表を使用します。その定義および含まれているデータは以下のとおりです。

```
CREATE TABLE xmlDataTable
(
    num INTEGER,
    xdata XML
)@

INSERT INTO xmlDataTable VALUES
(1, XMLPARSE(DOCUMENT '<doc>
                    <type>car</type>
                    <make>Pontiac</make>
                    <model>Sunfire</model>
                    </doc>' PRESERVE WHITESPACE)),
```

```

(2, XMLPARSE(DOCUMENT '
    <type>car</type>
    <make>Mazda</make>
    <model>Miata</model>
  </doc>' PRESERVE WHITESPACE)),
(3, XMLPARSE(DOCUMENT '
    <type>person</type>
    <name>Mary</name>
    <town>Vancouver</town>
    <street>Waterside</street>
  </doc>' PRESERVE WHITESPACE)),
(4, XMLPARSE(DOCUMENT '
    <type>person</type>
    <name>Mark</name>
    <town>Edmonton</town>
    <street>Oak</street>
  </doc>' PRESERVE WHITESPACE)),
(5, XMLPARSE(DOCUMENT '
    <type>animal</type>
    <name>dog</name>
  </doc>' PRESERVE WHITESPACE)),
(6, NULL),
(7, XMLPARSE(DOCUMENT '
    <type>car</type>
    <make>Ford</make>
    <model>Taurus</model>
  </doc>' PRESERVE WHITESPACE)),
(8, XMLPARSE(DOCUMENT '
    <type>person</type>
    <name>Kim</name>
    <town>Toronto</town>
    <street>Elm</street>
  </doc>' PRESERVE WHITESPACE)),
(9, XMLPARSE(DOCUMENT '
    <type>person</type>
    <name>Bob</name>
    <town>Toronto</town>
    <street>Oak</street>
  </doc>' PRESERVE WHITESPACE)),
(10, XMLPARSE(DOCUMENT '
    <type>animal</type>
    <name>bird</name>
  </doc>' PRESERVE WHITESPACE)))@

```

手順 独自の Java プロシージャを作成するときには、以下の例を参考にしてください。

- 322 ページの表 45
- 322 ページの表 46

Java 外部コード・ファイル

例では、Java プロシージャ・インプリメンテーションを示します。例は、CREATE PROCEDURE ステートメントと、関連 Java クラスのビルド元プロシージャの外部 Java コード・インプリメンテーションという 2 つの部分から成っています。

以下の例のプロシージャ・インプリメンテーションに含まれる Java ソース・ファイルは、stpclass.java という名前で、myJAR という名前の JAR ファイルに組み込まれています。ファイルの形式は以下のとおりです。

表 45. Java 外部コード・ファイルの形式

```
using System;
import java.lang.*;
import java.io.*;
import java.sql.*;
import java.util.*;
import com.ibm.db2.jcc.DB2Xml;

public class stpclass
{
    ...
    // Java procedure implementations
    ...
}
```

ファイルに先頭では、Java クラス・ファイルの `import` が示されています。ファイル内の、タイプ `XML` のパラメーターまたは変数を含むプロシージャが使用される場合は、`com.ibm.db2.jcc.DB2Xml import` が必要です。

クラス・ファイルの名前、および特定のプロシージャ・インプリメンテーションを含む JAR 名をメモしておくことは重要です。各プロシージャの `CREATE PROCEDURE` ステートメントの `EXTERNAL` 節でその情報を指定して、DB2 が実行時にそのクラスを見つけられるようにする必要があります。

例 1: XML パラメーターを使用するパラメーター・スタイル JAVA プロシージャ

この例では、以下について説明します。

- パラメーター・スタイル JAVA のプロシージャの `CREATE PROCEDURE` ステートメント
- XML パラメーターを使用するパラメーター・スタイル JAVA プロシージャの Java コード

このプロシージャは入力パラメーター、`inXML` を取り、その値を含む行を表に挿入し、SQL ステートメントと XQuery 式の両方を使用して XML データを照会して、2 つの出力パラメーター、`outXML1` と `outXML2` を設定します。

表 46. XML パラメーターを使用するパラメーター・スタイル JAVA プロシージャを作成するためのコード

```
CREATE PROCEDURE xmlProc1 ( IN inNUM INTEGER,
                           IN inXML XML as CLOB (1K),
                           OUT out1XML XML as CLOB (1K),
                           OUT out2XML XML as CLOB (1K)
                           )

DYNAMIC RESULT SETS 0
DETERMINISTIC
LANGUAGE JAVA
PARAMETER STYLE JAVA
MODIFIES SQL DATA
FENCED
THREADSAFE
DYNAMIC RESULT SETS 0
PROGRAM TYPE SUB
NO DBINFO
EXTERNAL NAME 'myJar:stpclass.xmlProc1'
```

表 46. XML パラメーターを使用するパラメーター・スタイル JAVA プロシージャを作成するためのコード (続き)

```

//*****
// Stored Procedure: XMLPROC1
//
// Purpose:  Inserts XML data into XML column; queries and returns XML data
//
// Parameters:
//
// IN:      inNum -- the sequence of XML data to be insert in xmldata table
//          inXML -- XML data to be inserted
// OUT:    out1XML -- XML data to be returned
//          out2XML -- XML data to be returned
//
//*****

public void xmlProc1(int inNum,
                    DB2Xml inXML ,
                    DB2Xml [] out1XML,
                    DB2Xml [] out2XML
                    )
throws Exception
{
    Connection con = DriverManager.getConnection("jdbc:default:connection");

    // Insert data including the XML parameter value into a table
    String query = "INSERT INTO xmlDataTable (num, inXML ) VALUES ( ?, ? )" ;
    String xmlString = inXML.getDB2String() ;

    stmt = con.prepareStatement(query);
    stmt.setInt(1, inNum);
    stmt.setString (2, xmlString );
    stmt.executeUpdate();
    stmt.close();

    // Query and retrieve a single XML value from a table using SQL
    query = "SELECT xdata from xmlDataTable WHERE num = ? " ;

    stmt = con.prepareStatement(query);
    stmt.setInt(1, inNum);
    ResultSet rs = stmt.executeQuery();

    if ( rs.next() )
    { out1Xml[0] = (DB2Xml) rs.getObject(1); }

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

    // Query and retrieve a single XML value from a table using XQuery
    query = "XQUERY for $x in db2-fn:xmlcolumn(¥'xmlDataTable.xdata¥')/doc
            where $x/make = ¥'Mazda¥'
            return <carInfo>{$x/make}{¥$x/model}</carInfo>";

    stmt = con.createStatement();

    rs = stmt.executeQuery( query );

    if ( rs.next() )
    { out2Xml[0] = (DB2Xml) rs.getObject(1) ; }

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

    return ;
}

```

第 8 章 OLE オートメーション・ルーチンの設計

オブジェクトのリンクと埋め込み (OLE) オートメーションは、Microsoft Corporation による OLE 2.0 アーキテクチャーの一部です。OLE オートメーションがあれば、ユーザー・アプリケーションは、作成に使用する言語に関係なく、OLE オートメーション・オブジェクト内でそのプロパティとメソッドを公開できます。Lotus Notes[®] や Microsoft Exchange のような他のアプリケーションは、OLE オートメーションによるこれらのプロパティとメソッドを利用して、これらのオブジェクトを統合することができます。

これらのプロパティとメソッドを公開するアプリケーションを OLE オートメーション・サーバーまたはオブジェクトと呼び、それらにアクセスするアプリケーションを OLE オートメーション・コントローラーと呼びます。OLE オートメーション・サーバーは、OLE IDispatch インターフェースをインプリメントする COM コンポーネント (オブジェクト) です。OLE オートメーション・コントローラーは、サーバーの IDispatch インターフェースを介してオートメーション・サーバーと通信する COM クライアントです。COM は、OLE の土台をなすものです。OLE オートメーション・ルーチンの場合、DB2 は OLE オートメーション・コントローラーとして動作します。DB2 は、この機構を介して、OLE オートメーション・オブジェクトの方式を外部ルーチンとして呼び出すことができます。

OLE オートメーションの解説ではすべて、OLE オートメーションの用語や概念をよく知っていることを前提としています。OLE オートメーションの概説については、「*Microsoft : The Component Object Model Specification* (1995 年 10 月)」を参照してください。OLE オートメーションの詳細については、「*OLE Automation Programmer's Reference* (Microsoft Press、1996 年、ISBN 1-55615-851-3)」を参照してください。

OLE オートメーション・ルーチンの作成

OLE オートメーション・ルーチンは、OLE オートメーション・オブジェクトのパブリック・メソッドとしてインプリメントされています。OLE オートメーション・オブジェクトは、OLE オートメーション・コントローラー (この場合は DB2) によって外部で作成可能でなければならず、遅延バインド (IDispatch ベースのバインドとも呼ばれる) をサポートしなければなりません。OLE オートメーション・オブジェクトは、クラス ID (CLSID) および任意で OLE プログラム ID (progID) を指定して、Windows レジストリーに登録し、オートメーション・オブジェクトを識別するようにします。progID は、プロセス内 (.DLL) またはローカル (.EXE) OLE オートメーション・サーバー、または DCOM (分散 COM) を介してリモート・サーバーを識別できます。

OLE オートメーション・ルーチンを登録するには、次のようにします。

OLE オートメーション・オブジェクトのコーディングが終わったら、CREATE ステートメントを使用して、そのオブジェクトのメソッドをルーチンとして作成する

必要があります。OLE オートメーション・ルーチンの作成は、C または C++ のルーチンの登録とよく似ていますが、以下のオプションを使用する必要があります。

- LANGUAGE OLE
- FENCED NOT THREADSAFE。OLE オートメーション・ルーチンは FENCED モードで実行する必要がありますが、THREADSAFE で実行することはできないからです。

外部名は、OLE 自動化オブジェクトを識別する OLE progID とメソッド名を ! (感嘆符) で区切った形になります。

```
CREATE FUNCTION bcounter () RETURNS INTEGER
EXTERNAL NAME 'bert.bcounter!increment'
LANGUAGE OLE
FENCED
NOT THREADSAFE
SCRATCHPAD
FINAL CALL
NOT DETERMINISTIC
NULL CALL
PARAMETER STYLE DB2SQL
NO SQL
NO EXTERNAL ACTION
DISALLOW PARALLEL;
```

OLE メソッド・インプリメンテーションの呼び出し規則は、C や C++ で作成されたルーチンの呼び出し規則と同一です。上記のメソッドを BASIC 言語でインプリメントすると、次のようになります (BASIC では、パラメーターはデフォルト設定で参照呼び出しとして定義されることに注意してください)。

```
Public Sub increment(output As Long, _
                    indicator As Integer, _
                    sqlstate As String, _
                    fname As String, _
                    fspecname As String, _
                    sqlmsg As String, _
                    scratchpad() As Byte, _
                    calltype As Long)
```

OLE ルーチン・オブジェクト・インスタンスおよびスクラッチパッドに関する考慮事項

OLE オートメーション UDF およびメソッド (OLE オートメーション・オブジェクトの方式) は、OLE オートメーション・オブジェクトのインスタンス上で適用されます。DB2 は、SQL ステートメント内で UDF またはメソッドを照会するたびにオブジェクト・インスタンスを作成します。オブジェクト・インスタンスは SQL ステートメント内でそれ以後の UDF またはメソッド参照の方式呼び出しに再使用されます。つまり、方式呼び出しの後にインスタンスは解放され、それ以後の方式呼び出しのたびに新しいオブジェクトが作成されます。CREATE ステートメントの SCRATCHPAD オプションによって、適切な振る舞いを指定できます。

LANGUAGE OLE 節の場合、SCRATCHPAD オプションには C や C++ の場合よりも多くのセマンティックがあり、1 つのオブジェクト・インスタンスが 1 つの照会を通じて使用するために作成および再利用されますが、NO SCRATCHPAD が指定されている場合には、メソッドが呼び出されるたびに新しいオブジェクト・インスタンスが作成されます。

スクラッチパッドを使用すると、メソッドは複数の関数またはメソッド呼び出しにわたって状態情報をオブジェクトのインスタンス変数内に保持できます。また、オブジェクト・インスタンスは一度だけ作成され、後の呼び出しで再利用されるので、パフォーマンスも向上します。

OLE オートメーションでサポートされている SQL データ・タイプ

DB2 は、SQL タイプと OLE オートメーション・タイプの間でタイプ変換を処理します。次の表では、サポートされるデータ・タイプと、それらがどのようにマップされるかを要約しています。

表 47. SQL と OLE オートメーション・データ・タイプのマッピング

SQL タイプ	OLE オートメーション・タイプ	OLE オートメーション・タイプの説明
SMALLINT	short	16 ビットの符号付き整数
INTEGER	long	32 ビットの符号付き整数
REAL	float	32 ビットの IEEE 浮動小数点数
FLOAT または DOUBLE	double	64 ビットの IEEE 浮動小数点数
DATE	DATE	1899 年 12 月 30 日からの日数を表す、64 ビットの浮動小数点分数
TIME	DATE	
TIMESTAMP	DATE	
CHAR(<i>n</i>)	BSTR	「OLE Automation Programmer's Reference」で説明されている、長さフィールド付きストリング
VARCHAR(<i>n</i>)	BSTR	
LONG VARCHAR	BSTR	
CLOB(<i>n</i>)	BSTR	
GRAPHIC(<i>n</i>)	BSTR	「OLE Automation Programmer's Reference」で説明されている、長さフィールド付きストリング
VARGRAPHIC(<i>n</i>)	BSTR	
LONG GRAPHIC	BSTR	
DBCLOB(<i>n</i>)	BSTR	
CHAR(<i>n</i>)	SAFEARRAY[unsigned char]	8 バイト符号なしデータ項目の 1 デイメンション Byte() 配列。(SAFEARRAY については、OLE Automation Programmer's Reference で解説されています。)
VARCHAR(<i>n</i>)	SAFEARRAY[unsigned char]	
LONG VARCHAR	SAFEARRAY[unsigned char]	
CHAR(<i>n</i>) FOR BIT DATA	SAFEARRAY[unsigned char]	
VARCHAR(<i>n</i>) FOR BIT DATA	SAFEARRAY[unsigned char]	
LONG VARCHAR FOR BIT DATA	SAFEARRAY[unsigned char]	
BLOB(<i>n</i>)	SAFEARRAY[unsigned char]	

DB2 と OLE オートメーション・ルーチンがやりとりするデータは、参照呼び出しとして渡されます。表に載っていない、BIGINT、DECIMAL、LOCATORS などの SQL タイプ、Boolean や CURRENCY などの OLE オートメーション・タイプは、サポートされません。BSTR にマップされる文字と GRAPHIC データは、データベース・コード・ページから UCS-2 スキーマに変換されます。(UCS-2 は Unicode とも呼ばれます。これは IBM コード・ページ 13488 です。) 戻されるとただちにデータは UCS-2 からデータベース・コード・ページに変換し直されます。これらの

変換は、データベース・コード・ページに関係なく起こります。このようなコード・ページ変換表をインストールしていないと、SQLCODE -332 が出されます (SQLSTATE 57017)。

BASIC および C++ での OLE オートメーション・ルーチン

OLE オートメーション・ルーチンは、どの言語でもインプリメントできます。この項では、2 つの言語 BASIC と C++ を例として取り上げ、OLE オートメーション・ルーチンをインプリメントする方法を示します。以下の表は、OLE オートメーション・タイプから BASIC と C++ のデータ・タイプへのマッピングを示しています。

表 48. SQL および OLE データ・タイプから BASIC および C++ データ・タイプへのマッピング

SQL タイプ	OLE オートメーション・タイプ	BASIC タイプ	C++ タイプ
SMALLINT	short	Integer	short
INTEGER	long	Long	long
REAL	float	Single	float
FLOAT または DOUBLE	double	Double	double
DATE、TIME、TIMESTAMP	DATE	Date	DATE
CHAR(<i>n</i>)	BSTR	String	BSTR
CHAR(<i>n</i>) FOR BIT DATA	SAFEARRAY[unsigned char]	Byte()	SAFEARRAY
VARCHAR(<i>n</i>)	BSTR	String	BSTR
VARCHAR(<i>n</i>) FOR BIT DATA	SAFEARRAY[unsigned char]	Byte()	SAFEARRAY
LONG VARCHAR	BSTR	String	BSTR
LONG VARCHAR FOR BIT DATA	SAFEARRAY[unsigned char]	Byte()	SAFEARRAY
BLOB(<i>n</i>)	BSTR	String	BSTR
BLOB(<i>n</i>) FOR BIT DATA	SAFEARRAY[unsigned char]	Byte()	SAFEARRAY
GRAPHIC(<i>n</i>)、VARGRAPHIC(<i>n</i>)、LONG GRAPHIC、DBCLOB(<i>n</i>)	BSTR	String	BSTR

BASIC での OLE オートメーション

BASIC で OLE オートメーション・ルーチンをインプリメントするには、OLE オートメーション・タイプにマップされた SQL データ・タイプに対応する BASIC データ・タイプを使用する必要があります。

OLE オートメーション UDF を BASIC で宣言すると、bcounter は次のようになります。

```
Public Sub increment(output As Long, _
                    indicator As Integer, _
                    sqlstate As String, _
                    fname As String, _
                    fspecname As String, _
                    sqlmsg As String, _
                    scratchpad() As Byte, _
                    calltype As Long)
```

C++ での OLE オートメーション

OLE オートメーション UDF を C++ で宣言すると、increment は次のようになります。

```
STDMETHODIMP Ccounter::increment (long *output,
                                   short *indicator,
                                   BSTR *sqlstate,
                                   BSTR *fname,
                                   BSTR *fspecname,
                                   BSTR *sqlmsg,
                                   SAFEARRAY **scratchpad,
                                   long *calltype );
```

OLE は、OLE オートメーション・オブジェクトのプロパティとメソッドを記述するタイプ・ライブラリーをサポートします。公開されるオブジェクト、プロパティ、およびメソッドは、オブジェクト記述言語 (ODL) で記述されます。上記の C++ メソッドを ODL で記述すると、次のようになります。

```
HRESULT increment ([out] long *output,
                   [out] short *indicator,
                   [out] BSTR *sqlstate,
                   [in] BSTR *fname,
                   [in] BSTR *fspecname,
                   [out] BSTR *sqlmsg,
                   [in,out] SAFEARRAY (unsigned char) *scratchpad,
                   [in] long *calltype);
```

ODL の記述では、パラメーターを入力 (in)、出力 (out)、入出力 (in,out) パラメーターのどれにするかを指定できます。OLE オートメーション・ルーチンの場合、ルーチンの入力パラメーターとその入力標識は [in] パラメーターとして指定され、ルーチンの出力パラメーターとその出力標識は [out] パラメーターとして指定されます。ルーチンの末尾引数の場合、sqlstate は [out] パラメーター、fname と fspecname は [in] パラメーター、scratchpad は [in,out] パラメーター、および calltype は [in] パラメーターです。

OLE オートメーションは、文字列を処理する BSTR データ・タイプを定義します。BSTR は、OLECHAR: typedef OLECHAR *BSTR へのポインターとして定義されます。BSTR の割り振りおよび解放に関しては、呼び出される側のルーチンは参照呼び出しパラメーターとして渡した BSTR を解放してから、参照呼び出しパラメーターに新しい値を割り当てる、という規則が OLE では適用されます。呼び出される側のルーチンが SAFEARRAY** として受け取る 1 ディメンションのバイト配列にも、同じ規則が適用されます。この規則は、DB2 と OLE オートメーション・ルーチンにとっては、次のような意味があります。

- [in] パラメーター: DB2 は [in] パラメーターの割り振りと解放を行います。
- [out] パラメーター: DB2 は NULL へのポインターを渡します。[out] パラメーターは、呼び出される側のルーチンによって割り振られ、DB2 によって解放されなければなりません。
- [in,out] パラメーター: DB2 は最初に [in,out] パラメーターを割り当てます。これらのパラメーターは、呼び出される側のルーチンによって解放および再割り振りすることができます。[out] パラメーターの場合のように、最後に戻されたパラメーターは DB2 が解放します。

他のすべてのパラメーターは、ポインターとして渡されます。DB2 は、参照されるメモリーを割り振りおよび管理します。

OLE オートメーションには、BSTR と SAFEARRAY を扱うための一そろいのデータ操作関数が備わっています。データ操作関数については、「*OLE Automation Programmer's Reference*」で説明されています。

次の C++ ルーチンは、CLOB 入力パラメーターの最初の 5 文字を戻します。

```
// UDF DDL: CREATE FUNCTION crunch (CLOB(5k)) RETURNS CHAR(5)

STDMETHODIMP Cobj::crunch (BSTR *in,          // CLOB(5K)
                           BSTR *out,        // CHAR(5)
                           short *indicator1, // input indicator
                           short *indicator2, // output indicator
                           BSTR *sqlstate,    // pointer to NULL
                           BSTR *fname,      // pointer to function name
                           BSTR *fspecname,   // pointer to specific name
                           BSTR *msgtext)    // pointer to NULL
{
    // Allocate BSTR of 5 characters
    // and copy 5 characters of input parameter

    // out is an [out] parameter of type BSTR, that is,
    // it is a pointer to NULL and the memory does not have to be freed.
    // DB2 will free the allocated BSTR.

    *out = SysAllocStringLen (*in, 5);
    return NOERROR;
};
```

OLE オートメーション・サーバーは、作成可能単独使用 か、作成可能複数使用 としてインプリメントできます。作成可能単独使用の場合、CoGetClassObject で OLE オートメーション・オブジェクトに接続している各クライアント (つまり、DB2 fenced プロセス) は、クラス・ファクトリーの独自インスタンスを使用し、必要に応じて OLE オートメーション・サーバーのコピーを新規に実行します。作成可能複数使用の場合、多数のクライアントが同じクラス・ファクトリーに接続します。つまり、クラス・ファクトリーの各インスタンスは、すでに実行されている OLE サーバー (存在する場合) によって提供されます。実行中の OLE サーバー・コピーがない場合は、自動的に 1 つが起動され、クラス・オブジェクトを提供します。単独使用と複数使用 OLE オートメーションの選択は、オートメーション・サーバーをインプリメントするときにユーザーが行います。パフォーマンスを重視する場合は単独使用サーバーをお勧めします。

第 9 章 OLE DB ユーザー定義表関数

Microsoft OLE DB は、アプリケーションが、さまざまな情報ソースに保管されているデータに同じ方法でアクセスできるようにする、OLE/COM インターフェースのセットです。OLE DB コンポーネントの DBMS アーキテクチャーでは、OLE DB Consumer と OLE DB Provider を定義しています。OLE DB Consumer は、OLE DB インターフェースを使用するシステムまたはアプリケーションで、OLE DB Provider は、OLE DB インターフェースを公開するコンポーネントです。OLE DB Providers には以下の 2 つのクラスがあります。1 つは *OLE DB データ提供者* で、データを所有し、そのデータを行セットのような表形式で公開します。もう 1 つは *OLE DB サービス提供者* で、それ自身のデータを所有しませんが、OLE DB インターフェースによってデータを作成および使用して、サービスをカプセル化します。

DB2 データベース・システムは、OLE DB データ・ソースにアクセスする表関数を定義できるようにすることにより、OLE DB アプリケーションの作成を単純化しています。DB2 は、任意の OLE DB データまたはサービス提供者にアクセスすることができる OLE DB Consumer になります。OLE DB インターフェースを介してデータを公開するデータ・ソース上で、GROUP BY、JOIN、および UNION を含む操作を実行することができます。例えば、OLE DB 表関数を定義して、Microsoft Access データベースまたは Microsoft Exchange のアドレス帳から表を戻し、それからこの OLE DB 表関数からのデータと DB2 データベース中のデータとをシームレスに結合したレポートを作成することができます。

OLE DB 表関数を使用すると、OLE DB Provider への組み込みアクセスが提供されるため、アプリケーション開発に費やす労力が軽減されます。C、Java、および OLE オートメーション表関数では、開発者は表関数をインプリメントする必要がありますが、OLE DB 表関数では、汎用組み込み OLE DB Consumer が、データを検索する OLE DB Provider とやり取りをします。それには、表関数を LANGUAGE OLEDB として登録し、OLE DB Provider とそれに関連した行セットをデータ・ソースとして参照する必要があるだけです。OLE DB 表関数を使用するために、なんらかの UDF プログラミングを行う必要はありません。

OLE DB 表関数を DB2 データベースで使用するには、OLE DB 2.0 またはそれ以降をインストールする必要があります (これは、Microsoft 社の <http://www.microsoft.com> から入手できます)。OLE DB をインストールしていないまま OLE DB 表関数を呼び出そうとすると、DB2 は SQLCODE -465、SQLSTATE 58032、理由コード 35 を発行します。システム要件と、ご使用のデータ・ソースで使用可能な OLE DB Providers については、データ・ソース資料を参照してください。OLE DB の仕様については、*Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK* (Microsoft Press から入手可能) を参照してください。

OLE DB 表関数の使用に関する制約事項: OLE DB 表関数は、DB2 データベースに接続できません。

OLE DB 表 UDF の作成

OLE DB 表関数を単一の CREATE FUNCTION ステートメントで定義するには、以下のようにする必要があります。

- OLE DB Provider が戻す表を定義する。
- LANGUAGE OLEDB を指定する。
- OLE DB 行セットを識別し、OLE DB Provider 接続ストリングを EXTERNAL NAME 節に指定する。

OLE DB データ・ソースは、そのデータを行セット と呼ばれる表形式で公開します。行セットとは、それぞれの行が列セットを持つ行のセットです。RETURNS TABLE 節には、ユーザーと関係がある列だけが含まれます。OLE DB データ・ソースでの表関数列と行セットの列とのバインドは、列名に基づいて行われます。OLE DB Provider が大文字小文字の区別をする場合、例えば、"UPPERcase" のように、列名を引用符の間に置いてください。

EXTERNAL NAME 節は、以下のいずれかの形式をとることができます。

```
'server!rowset'  
または  
!'rowset!connectstring'
```

詳細は次のとおりです。

server CREATE SERVER ステートメントで登録されたサーバー。

rowset OLE DB Provider によって公開された行セットまたは表を識別します。コマンド・テキストから OLE DB Provider に渡される入力パラメーターがその表にある場合、この値は空になります。

connectstring

OLE DB Provider に接続するために必要な初期化特性が含まれます。接続ストリングの完全な構文とセマンティクスについては、「*Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*」(Microsoft Press、1998)の『Data Link API of the OLE DB Core Components』を参照してください。

CREATE FUNCTION ステートメントの EXTERNAL NAME 節で接続ストリングを使用するか、または CREATE SERVER ステートメントで *CONNECTSTRING* オプションを指定することができます。

例えば、以下の CREATE FUNCTION および SELECT ステートメントを使用して、OLE DB 表関数を定義して Microsoft Access データベースからの表を戻すことができます。

```
CREATE FUNCTION orders ()  
  RETURNS TABLE (orderid INTEGER, ...)  
  LANGUAGE OLEDB  
  EXTERNAL NAME '!orders!Provider=Microsoft.Jet.OLEDB.3.51;  
                Data Source=c:\msdasdk\bin\oledb\wind.mdb';  
  
SELECT orderid, DATE(orderdate) AS orderdate,  
       DATE(shippeddate) AS shippeddate  
FROM TABLE(orders()) AS t  
WHERE orderid = 10248;
```

EXTERNAL NAME 節に接続ストリングを入れる代わりに、サーバー名を作成および使用することができます。例えば、サーバー Nwind を定義してあるとすると、次の CREATE FUNCTION ステートメントを使用することができます。

```
CREATE FUNCTION orders ()
  RETURNS TABLE (orderid INTEGER, ...)
  LANGUAGE OLEDB
  EXTERNAL NAME 'Nwind!orders';
```

OLE DB 表関数により、任意の文字ストリング・データ・タイプの入力パラメーターを 1 つ指定することもできます。入力パラメーターを使用して、OLE DB Provider にコマンド・テキストを直接渡します。入力パラメーターを定義する場合、EXTERNAL NAME 節に行セット名を指定しないでください。DB2 は、実行するコマンド・テキストを OLE DB Provider に渡し、OLE DB Provider は DB2 に行セットを戻します。結果として戻される行セットの列名とデータ・タイプは、CREATE FUNCTION ステートメントの RETURNS TABLE 定義と互換性がある必要があります。行セットの列名とのバインドは、一致する列名に基づいているため、列を正しく命名していることを確かめる必要があります。

以下の例では、Microsoft SQL Server 7.0 データベースから保管情報を検索する OLE DB 表関数を登録します。EXTERNAL NAME 節に接続ストリングを指定します。表関数に、コマンド・テキストから OLE DB Provider に渡される入力パラメーターがあるため、EXTERNAL NAME 文節では行セット名は指定されません。SQL コマンド・テキストで照会の例が渡され、SQL サーバー・データベースからの上位 3 つの保管についての情報が検索されます。

```
CREATE FUNCTION favorites (VARCHAR(600))
  RETURNS TABLE (store_id CHAR (4), name VARCHAR (41), sales INTEGER)
  SPECIFIC favorites
  LANGUAGE OLEDB
  EXTERNAL NAME '!!Provider=SQLOLEDB.1;Persist Security Info=False;
  User ID=sa;Initial Catalog=pubs;Data Source=WALTZ;
  Locale Identifier=1033;Use Procedure for Prepare=1;
  Auto Translate=False;Packet Size=4096;Workstation ID=WALTZ;
  OLE DB Services=CLIENTCURSOR;';

SELECT *
FROM TABLE (favorites (' select top 3 sales.stor_id as store_id, ' |
  ' stores.stor_name as name, ' |
  ' sum(sales.qty) as sales ' |
  ' from sales, stores ' |
  ' where sales.stor_id = stores.stor_id ' |
  ' group by sales.stor_id, stores.stor_name ' |
  ' order by sum(sales.qty) desc')) as f;
```

完全修飾行セット名

EXTERNAL NAME 節の一部の行セットは、完全修飾名 で識別される必要があります。完全修飾名には、以下のいずれかまたはその両方が組み込まれます。

- 関連するカタログ名。以下の情報が必要です。
 - プロバイダーがカタログ名をサポートしているかどうか
 - カatalog名を完全修飾名のどこに入れるか
 - 使用するカタログ名区切り記号
- 関連するスキーマ名。以下の情報が必要です。
 - プロバイダーがスキーマ名をサポートしているかどうか

- 使用するスキーマ名区切り記号

OLE DB Provider によって提供されるカタログとスキーマ名のサポートについての情報は、OLE DB Provider のリテラル情報の資料を参照してください。

ご使用のプロバイダーのリテラル情報で DBLITERAL_CATALOG_NAME が NULL でない場合、カタログ名と DBLITERAL_CATALOG_SEPARATOR の値を区切り記号として使用してください。完全修飾名の先頭または終わりのどちらかにカタログ名を置くかを判別するには、OLE DB Provider のプロパティ・セット DBPROPSSET_DATASOURCEINFO にある DBPROP_CATALOGLOCATION の値を参照してください。

ご使用のプロバイダーのリテラル情報で DBLITERAL_SCHEMA_NAME が NULL でない場合、スキーマ名と DBLITERAL_SCHEMA_SEPARATOR の値を区切り記号として使用してください。

名前に特殊文字や突き合わせキーワードが含まれている場合、OLE DB Provider に指定された引用符文字で名前を囲んでください。引用符文字は、OLE DB Provider のリテラル情報で、DBLITERAL_QUOTE_PREFIX および DBLITERAL_QUOTE_SUFFIX として定義されます。例えば、以下の EXTERNAL NAME では、指定された行セットには、authors と呼ばれる行セットのカタログ名 pubs とスキーマ名 dbo が含まれていて、その名前を囲むために引用符文字 " が使用されています。

```
EXTERNAL NAME '!"pubs"."dbo"."authors"!Provider=SQLOLEDB.1;...!';
```

完全修飾名の構成の詳細については、「*Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*」(Microsoft Press、1998) と、OLE DB Provider の資料を参照してください。

OLE DB でサポートされている SQL データ・タイプ

以下の表では、DB2 データ・タイプを OLE DB データ・タイプにマップする方法が示されています。これについては「*Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*」(Microsoft Press、1998) に記載されています。マッピング表を使用して、OLE DB 表関数に適切な RETURNS TABLE 列を定義します。例えば、データ・タイプ INTEGER の列を使用して OLE DB 表関数を定義する場合、DB2 は OLE DB Provider からのデータを DBTYPE_I4 として要求します。

OLE DB Provider ソース・データ・タイプから OLE DB データ・タイプへのマッピングについては、OLE DB Provider の資料を参照してください。ANSI SQL、Microsoft Access、および Microsoft SQL Server プロバイダーが、それぞれのデータ・タイプを OLE DB データ・タイプにマップする方法の例については、

「*Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*」(Microsoft Press、1998) を参照してください。

表 49. DB2 データ・タイプの OLE DB へのマッピング

DB2 データ・タイプ	OLE DB データ・タイプ
SMALLINT	DBTYPE_I2
INTEGER	DBTYPE_I4
BIGINT	DBTYPE_I8
REAL	DBTYPE_R4

表 49. DB2 データ・タイプの OLE DB へのマッピング (続き)

DB2 データ・タイプ	OLE DB データ・タイプ
FLOAT/DOUBLE	DBTYPE_R8
DEC (p, s)	DBTYPE_NUMERIC (p, s)
DATE	DBTYPE_DBDATE
TIME	DBTYPE_DBTIME
TIMESTAMP	DBTYPE_DBTIMESTAMP
CHAR(N)	DBTYPE_STR
VARCHAR(N)	DBTYPE_STR
LONG VARCHAR	DBTYPE_STR
CLOB(N)	DBTYPE_STR
CHAR(N) FOR BIT DATA	DBTYPE_BYTES
VARCHAR(N) FOR BIT DATA	DBTYPE_BYTES
LONG VARCHAR FOR BIT DATA	DBTYPE_BYTES
BLOB(N)	DBTYPE_BYTES
GRAPHIC(N)	DBTYPE_WSTR
VARGRAPHIC(N)	DBTYPE_WSTR
LONG GRAPHIC	DBTYPE_WSTR
DBCLOB(N)	DBTYPE_WSTR

注: OLE DB データ・タイプの変換規則は、「*Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998」で定義されています。以下に例を示します。

- OLE DB データ・タイプ DBTYPE_CY を検索するには、データを OLE DB データ・タイプ DBTYPE_NUMERIC(19,4) に変換することができ、これは DB2 データ・タイプ DEC(19,4) にマップされます。
- OLE DB データ・タイプ DBTYPE_I1 を検索するには、データを OLE DB データ・タイプ DBTYPE_I2 に変換することができ、これは DB2 データ・タイプ SMALLINT にマップされます。
- OLE DB データ・タイプ DBTYPE_GUID を検索するには、データを OLE DB データ・タイプ DBTYPE_BYTES に変換することができ、これは DB2 データ・タイプ CHAR(12) FOR BIT DATA にマップされます。

第 10 章 ルーチンの呼び出し

ルーチンを開発し、CREATE ステートメントの実行によってルーチンをデータベース内に作成した場合、そのルーチンを呼び出すことができます (ただし、ルーチンの定義元また呼び出し元に適切なルーチン特権が付与されている必要があります)。

ルーチンの目的と使用法は、それぞれのタイプによって異なります。ルーチンを呼び出すための前提条件は共通ですが、呼び出しのインプリメンテーションがそれぞれ異なります。

ルーチン呼び出しの前提条件

- CREATE ステートメントを使用して、データベース内にルーチンを作成しておく必要があります。
- 外部ルーチンの場合は、CREATE ステートメントの EXTERNAL 節で指定するロケーションにライブラリー・ファイルまたはクラス・ファイルをインストールする必要があります。これを行わないと、エラー (SQLCODE SQL0444、SQLSTATE 42724) になります。
- ルーチンの呼び出し側には、そのルーチンに関する EXECUTE 特権が必要です。呼び出し側にルーチン実行の許可が与えられていない場合は、エラー (SQLSTATE 42501) になります。

プロシージャの呼び出し

プロシージャを呼び出すには、そのプロシージャへの参照を指定した CALL ステートメントを実行します。

CALL ステートメントは、プロシージャを呼び出し、プロシージャにパラメーターを渡し、プロシージャから戻されるパラメーターを受け取ることを可能にするためのステートメントです。プロシージャから戻されるアクセス可能な結果セットの処理はすべて、プロシージャが正常に戻された後に行えます。

プロシージャは、CALL ステートメントがサポートされているところであればどこからでも呼び出せます。例えば、以下から呼び出すことができます。

- クライアント・アプリケーション
- 外部ルーチン (プロシージャ、UDF、またはメソッド)
- SQL ルーチン (プロシージャ、UDF、またはメソッド)
- トリガー (BEFORE トリガー、AFTER トリガー、または INSTEAD OF トリガー)
- 動的コンパウンド・ステートメント
- コマンド行プロセッサ (CLP)

クライアント・アプリケーションまたは外部ルーチンからプロシージャを呼び出す場合、そのクライアント・アプリケーションまたは外部ルーチンは、プロシージャの言語以外の言語で作成してもかまいません。例えば、C++ で作成したクライアント・アプリケーションから、CALL ステートメントを使用して、Java で作成したプロシージャを呼び出すことができます。したがって、プログラマーにとっ

では、自分の好みの言語でプログラミングを行い、さまざまな言語で作成したコードの断片を統合する、という柔軟な作業が可能になります。

さらに、プロシージャが置かれているものとは異なるオペレーティング・システム上で、プロシージャを呼び出すクライアント・アプリケーションを実行してもかまいません。例えば、Windows オペレーティング・システム上で実行しているクライアント・アプリケーションから、CALL ステートメントを使用して、Linux データベース・サーバー上にあるプロシージャを呼び出すことができます。

プロシージャの呼び出し側によっては、追加の考慮事項があります。

関数の呼び出し

関数は、SQL ステートメント内での参照によって呼び出します。

組み込み関数、ソース派生集約関数、スカラー・ユーザー定義関数への参照は、SQL ステートメント内で式を使用できる場所であれば、どこにでも記述できます。例えば、照会の選択リスト内や INSERT ステートメントの VALUES 節内などの場所があります。表関数は FROM 節内でしか参照できません。例えば、照会やデータ変更ステートメントの FROM 節内です。

メソッドの呼び出し

メソッドは、構造化タイプに対して振る舞いを指定するという点を除けば、スカラー関数と似ています。メソッドの呼び出しは、メソッドのパラメーターの 1 つがメソッドの操作対象の構造化タイプでなければならないという点を除けば、スカラー・ユーザー定義関数呼び出しと同じです。

ルーチン呼び出しの関連タスク

特定のタイプのルーチンを呼び出すには、以下の項を参照してください。

- 347 ページの『アプリケーションまたは外部ルーチンからのプロシージャの呼び出し』
- 348 ページの『トリガーまたは SQL ルーチンからのプロシージャの呼び出し』
- 詳しくは、「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」の『CLI アプリケーションからのストアード・プロシージャの呼び出し』を参照してください。
- 351 ページの『コマンド行プロセッサ (CLP) からのプロシージャの呼び出し』
- 364 ページの『スカラー関数またはメソッドの呼び出し』
- 364 ページの『ユーザー定義の表関数の呼び出し』

SQL の入ったルーチンの許可およびバインド

ルーチン・レベルの許可について論じるときには、まずルーチンに関連したいくつかのロール、それぞれのロールの判別、各ロールに関連した特権について説明することが大切です。

パッケージ所有者

ルーチンのインプリメンテーションにかかわっているパッケージの所有者です。パッケージの所有者とは、パッケージをデータベースにバインドするために BIND コマンドを実行したユーザーです (ただし、プリコンパイルバインドの OWNER オプションを使用してパッケージの所有権をオーバーライドし、別のユーザーにその所有権を設定した場合は別です)。BIND コマンドを実行すると、パッケージに関する EXECUTE WITH GRANT 特権がパッケージ所有者に付与されます。ルーチンのライブラリーまたは実行可能ファイルは、複数のパッケージで構成されることがあるので、そのような場合は、複数のパッケージ所有者が関連付けられます。

ルーチン定義者

ルーチンを登録するために CREATE ステートメントを発行した ID です。基本的には DBA がルーチン定義者になりますが、ルーチンのパッケージ所有者がルーチン定義者になることもあります。ルーチンを呼び出すと、パッケージのロード時に、そのルーチンを実行する許可が、そのルーチンに関連した 1 つ以上のパッケージを実行する定義者の許可 (ルーチンの呼び出し側の許可ではない) に照らしてチェックされます。ルーチンを正常に呼び出すには、ルーチン定義者に以下の特権のいずれかが必要です。

- ルーチンの 1 つ以上のパッケージに関する EXECUTE 特権とルーチンに関する EXECUTE 特権
- DATAACCESS 権限

ルーチン定義者とルーチンのパッケージ所有者が同じユーザーの場合、ルーチン定義者はパッケージに関するその必要な EXECUTE 特権を持つこととなります。定義者がパッケージ所有者でない場合は、ACCESSCTRL または SECADM 権限を持つユーザーか、パッケージに関する CONTROL または EXECUTE WITH GRANT OPTION 特権が、パッケージに関する EXECUTE 特権を定義者に明示的に付与する必要があります。(パッケージの作成者には、パッケージに対する CONTROL および EXECUTE WITH GRANT OPTION 特権が自動的に付与されます。)

ルーチンを登録する CREATE ステートメントを実行すると、ルーチンに関する EXECUTE WITH GRANT OPTION 特権が定義者に暗黙的に GRANT されます。

ルーチン定義者のロールは、ルーチンに関連付けられているパッケージを実行する特権と、PUBLIC またはルーチンを呼び出す必要のある特定のユーザーに対して、ルーチンに関する EXECUTE 特権を付与する特権を 1 つの許可 ID にカプセル化することです。

注: SQL ルーチンの場合、ルーチン定義者は暗黙的にパッケージ所有者にもなります。したがって定義者は、ルーチンの CREATE ステートメントの実行時に、ルーチンとルーチン・パッケージの両方に関する EXECUTE WITH GRANT OPTION を持つこととなります。

ルーチンの呼び出し側

ルーチンを呼び出す ID です。どのユーザーをルーチンの呼び出し側とするかを決定するには、どのようにルーチンが呼び出されるかを考慮することが必要です。ルーチンは、コマンド・ウィンドウか、組み込み SQL アプリケーション内から呼び出せます。メソッドと UDF の場合は、ルーチン参照を

別の SQL ステートメントに組み込みます。プロシージャは、CALL ステートメントによって呼び出します。アプリケーション内の動的 SQL の場合、呼び出し側は、すぐ上のレベルのルーチンのランタイム許可 ID か、そのルーチン呼び出しが入ったアプリケーションのランタイム許可 ID です (ただし、この ID は、そのルーチンまたはアプリケーションをバインドしたときに使用した DYNAMICRULES オプションに依存する場合があります)。静的 SQL の場合、呼び出し側は、ルーチンの参照を含むパッケージの OWNER プリコンパイル/BIND オプションの値です。正常にルーチンを呼び出すには、これらのユーザーにルーチンに関する EXECUTE 特権が必要です。GRANT ステートメントを明示的に実行することによってこの特権をだれかに GRANT できるのは、ルーチンに関する EXECUTE WITH GRANT OPTION 特権を持つユーザー (この特権が明示的に取り消されていないルーチン定義者も含まれる) と、ACCESSCTRL 権限または SECADM 権限を持つユーザーです。

例えば、動的 SQL を含むアプリケーションに関連したパッケージを DYNAMICRULES BIND でバインドした場合は、そのパッケージを呼び出した人物ではなく、そのアプリケーションの実行時許可 ID がパッケージ所有者になります。また、パッケージ所有者は、実際のバインド・プログラムになるか、プリコンパイルバインドの OWNER オプションの値になります。その場合、ルーチンの呼び出し側は、アプリケーションを実行しているユーザーの ID ではなく、そのオプション値をとります。

注:

1. ルーチン内の静的 SQL の場合、パッケージ所有者の特権は、ルーチン本体内の SQL ステートメントの実行に関して十分なものでなければなりません。ルーチンに対するネストされた参照がある場合、これらの SQL ステートメントで表へのアクセス権または EXECUTE 特権が必要になることもあります。
2. ルーチン内の動的 SQL の場合、特権を検査されるユーザー ID は、ルーチン本体の BIND の DYNAMICRULES オプションによって規制されます。
3. ルーチン・パッケージ所有者は、ルーチンの定義者に対してパッケージでの実行許可を与える必要があります。許可を与えるのは、ルーチンの登録の前でも後でもかまいませんが、いずれにしてもルーチンの呼び出し前でなければなりません。そうでない場合は、エラー (SQLSTATE 42051) が戻されます。

以下の図とテキストに、ルーチンに関する EXECUTE 特権の管理に関するステップを示します。

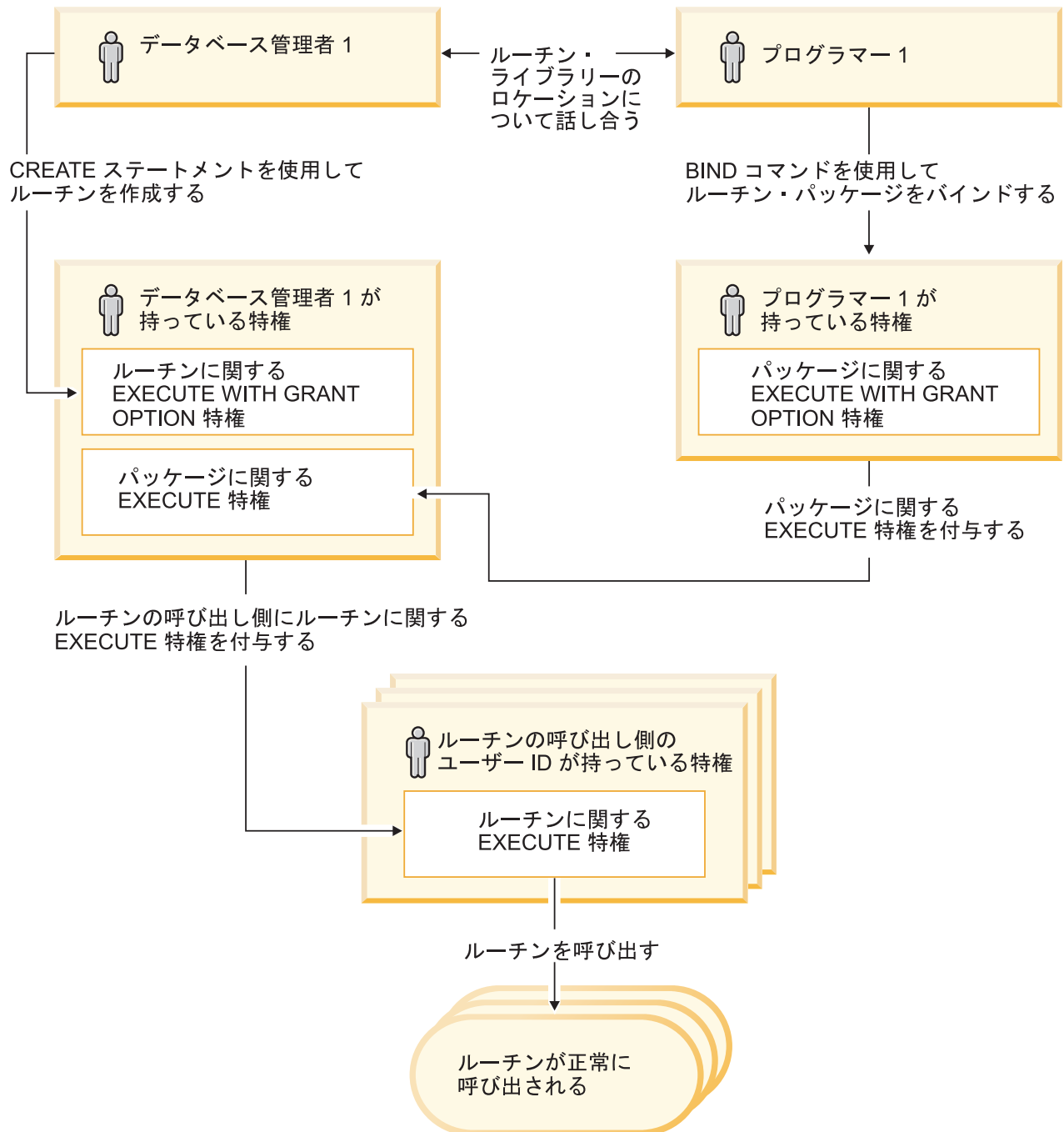


図 3. ルーチンに関する EXECUTE 特権の管理

1. 定義者は、該当する CREATE ステートメントを実行してルーチンを登録します。このステートメントでは、所定のレベルの SQL アクセス権の指定のもとにルーチンを DB2 に登録し、ルーチンのシグニチャーを確立し、ルーチンの実行可能プログラムの位置を指定します。定義者がパッケージ所有者も兼ねていない場合は、パッケージ所有者およびルーチン・プログラムの作成者と連絡を取り合って、ルーチン・ライブラリーが存在する場所を明確にする必要があります。そうしないと、CREATE ステートメントの EXTERNAL 節でその場所を正確に指定できないからです。CREATE ステートメントを正常に実行すると、定義者は

そのルーチンに関する EXECUTE WITH GRANT 特権を持つこととなりますが、ルーチンのパッケージに関する EXECUTE 特権を持つことにはなりません。

2. 定義者は、ルーチンの使用を許可される予定のすべてのユーザーに対してルーチンに関する EXECUTE 特権を付与する必要があります。(そのルーチンのパッケージがルーチンを再帰的に呼び出す場合、次のステップの前に必ずこのステップを実行する必要があります。)
3. パッケージ所有者は、ルーチン・プログラムをプリコンパイルおよびバインドするか、または他でそれを代行させます。プリコンパイルとバインドが正常に完了すると、パッケージの所有者にはそれぞれのパッケージに関する EXECUTE WITH GRANT OPTION 特権が暗黙的に GRANT されます。このリストのステップ 1 の後にこのステップが続いているのは、ルーチン内で SQL を繰り返す場合に備えるためではありません。どのような場合でもそのような繰り返しが存在しない場合は、ルーチンの CREATE ステートメントを実行する前にプリコンパイルバインドを実行してかまいません。
4. 各パッケージ所有者は、それぞれのルーチン・パッケージに関する EXECUTE 特権をルーチンの定義者に対して明示的に付与する必要があります。このステップは、前のステップの少し後で実行しなければなりません。パッケージ所有者がルーチンの定義者も兼ねている場合は、このステップを省略できます。
5. ルーチンの静的使用: ルーチンを参照するパッケージのバインド所有者は、ルーチンに対する EXECUTE 特権をすでに与えられていなければなりません。したがって、この時点で前のステップはすでに完了していなければなりません。ルーチンの実行時に DB2 は、すべてのパッケージに対する必要な EXECUTE 特権を定義者が持っているかどうかを検査します。したがってこの時点で、該当する各パッケージを対象としたステップ 3 は完了していなければなりません。
6. ルーチンの動的使用: DYNAMICRULES オプションによって制御される呼び出し元アプリケーションの許可 ID には、ルーチンに対する EXECUTE 特権がなければならず (ステップ 4)、ルーチンの定義者にはパッケージに対する EXECUTE 特権がなければなりません (ステップ 3)。

ルーチン名およびパス

ストアード・プロシージャまたは UDF の修飾名は `schema-name.routine-name` です。ストアード・プロシージャまたは UDF を参照するどの場所でもこの修飾名を使用できます。以下に例を示します。

```
SANDRA.BOAT_COMPARE    SMITH.FOO    SYSIBM.SUBSTR    SYSFUN.FLOOR
```

`schema-name.` を省略することもできますが、省略すると、DB2 は参照されているストアード・プロシージャまたは UDF の識別を試みます。以下に例を示します。

```
BOAT_COMPARE    FOO    SUBSTR    FLOOR
```

メソッドの修飾名は `schema-name.type.method-name` です。

SQL パスの概念は、`schema-name` を使用しない場合に非修飾参照を DB2 が解決するための中心となるものです。SQL パスとは、スキーマ名の順序付けられたリストです。これには、ストアード・プロシージャ、UDF、およびタイプへの非修飾参照を解決するためのスキーマが用意されています。パスの複数のスキーマ内の

ストアード・プロシージャー、タイプ、または UDF と参照が一致する場合は、パス内のスキーマの順序を使用してこの一致が解決されます。SQL パスは、静的 SQL の場合はプリコンパイルおよびバインド・コマンド上の FUNCSPATH オプションによって設定されます。SQL パスは、動的 SQL の SET PATH ステートメントによって設定されます。SQL パスのデフォルト値は、次のようになります。

```
"SYSIBM", "SYSFUN", "SYSPROC", "ID"
```

これは静的 SQL と動的 SQL の両方の場合に当てはまります。この場合の ID は、現行ステートメントの許可 ID を表します。

ルーチン名は多重定義 することができます。つまり、1 つのスキーマ内で複数のルーチンに同じ名前を付けられるということです。同一名の付いた複数の関数またはメソッドは、それぞれのデータ・タイプが異なってさえいれば、同一数のパラメーターを持つことができます。これはストアード・プロシージャーには当てはまりません。すなわち、複数のストアード・プロシージャーに同一名が付いていれば、それぞれ異なる数のパラメーターをもっていなければならないということです。ルーチン・タイプがそれぞれ異なるインスタンスは互いに多重定義しあうことはありません。ただし、関数を多重定義できるメソッドは例外です。メソッドで関数を多重定義するには、WITH FUNCTION ACCESS 節を使用してそのメソッドを登録する必要があります。

関数、ストアード・プロシージャー、およびメソッドは、互いの多重定義なしで同一のシグニチャー をもつことができ、同一のスキーマ内に存在することができます。ルーチンのコンテキストに関するかぎり、シグニチャーとは、すべてのパラメーターの定義済みでしかも定義順に並んだデータ・タイプに連結された修飾ルーチン名のことで、

メソッドは、関連した構造化タイプのインスタンスに対して呼び出されます。サブタイプが作成されるときにそれが継承する属性の中には、スーパータイプ用に定義されたメソッドがあります。したがって、スーパータイプのメソッドを、そのサブタイプのインスタンスに対して実行することもできます。サブタイプを定義するときには、スーパータイプのメソッドをオーバーライドする ことができます。メソッドをオーバーライドするということは、そのメソッドを特定のサブタイプ用に特別に再インプリメントすることを意味します。これにより、メソッドの動的ディスパッチング (ポリモアフィズムとも言います) が容易になります。動的ディスパッチングでは、構造化タイプ・インスタンスのタイプに従って最も特定されたメソッドがアプリケーションによって実行されます (例えば、構造化タイプ階層にあるもの)。

どのルーチン・タイプにも独自の選択アルゴリズムがありますが、ここでは、多重定義 (メソッドの場合とオーバーライド) と SQL パスへの対策が講じられていて、ルーチンの各参照ごとに最も一致するものが選択されます。

ネストされたルーチンの呼び出し

ルーチンのコンテキストに関するかぎりでは、ネスティングとはルーチンが他のルーチンを呼び出す状況のことを言います。つまり、あるルーチンによって発行された SQL は別のルーチンを参照することができ、さらに後のルーチンはまた別のルーチンを参照する SQL を発行できる、というわけです。参照先の一連のルーチンの中に、以前に参照されたルーチンが含まれている場合、これを再帰的なネスティングの状況といます。

ネスティングおよび再帰は、次のような制約事項のもとに DB2 ルーチン内で使用することができます。

64 レベルのネスト

ルーチンの呼び出しは、64 段階の深さのレベルまでネストすることができます。ルーチン A がルーチン B を呼び出し、ルーチン B がルーチン C を呼び出すシナリオを考察してみます。この例では、ルーチン C の実行はネスト・レベル 3 で行われます。さらに 61 段階のネスト・レベルを指定することができます。

その他の制限

ルーチンは、自分より高い SQL データ・アクセス・レベルでカタログされたターゲット・ルーチンを呼び出すことはできません。例えば、CONTAINS SQL 節で作成された UDF は CONTAINS SQL 節または NO SQL 節のいずれかで作成されたストアード・プロシージャを呼び出すことはできません。しかしこのルーチンは、READS SQL DATA 節または MODIFIES SQL DATA 節を使用して作成されたストアード・プロシージャを呼び出すことはできません (SQLCODE -577、SQLSTATE 38002)。その理由は、呼び出し側の SQL レベルでは、行おうとしている操作の読み取りも変更も許可されないからです (これは、呼び出されるルーチンへと継承されます)。

ルーチンのネスティングでは、読み取りおよび書き込みの操作でルーチン同士に競合が生じないようにするために表へのアクセスが制限されるというまた別の制限事項もあります。

64 ビット・データベース・サーバーでの 32 ビット・ルーチンの呼び出し

64 ビット DB2 インスタンスでは、32 ビット外部ルーチン・ライブラリーを参照する C および COBOL ルーチンを呼び出すことができます。ただしこれらのルーチンは、スレッド・セーフではなく fenced として実行するように指定する必要があります。これは、新規ルーチンの作成時にルーチンの CREATE ステートメントに FENCED 節および NOT THREADSAFE 節の両方を組み込むことで実行できます。64 ビット・インスタンスですでに作成済みのルーチンの場合、ALTER FUNCTION または ALTER PROCEDURE ステートメントは、ルーチン定義を変更するために使用できます。そのような 32 ビット・ルーチンを 64 ビット環境で初めて呼び出したときは、パフォーマンスが低下します。その後の 32 ビット・ストアード・プロシージャの呼び出しでは、64 ビット・ルーチンと同等のパフォーマンスが示されます。32 ビット・ルーチンを 64 ビットの DB2 インスタンスで使用することはお勧めしません。

64 ビット・データベース・サーバー上で 64 ビットの DB2 インスタンス内の Java プロシージャを正常に呼び出すには、64 ビット Java 仮想マシン (JVM) が必要です。32 ビットの JVM は、64 ビットの DB2 インスタンスでのルーチンの実行にはサポートされていません。Java クラスはプラットフォームに依存しないので、32 ビットのソフトウェア開発キットでコンパイルされた Java クラスは、64 ビットの JVM で正常に実行できます。これを実行しても、ルーチンのパフォーマンスには影響はありません。

既存の 32 ビット・ルーチンを 64 ビット・サーバー上で呼び出すには、次のようにします。

1. 次のようにして、ルーチンのクラスまたはライブラリーをデータベースのルーチン・ディレクトリーにコピーします。
 - UNIX: `sqlib/function`
 - Windows: `sqlib¥function`
2. ストアド・プロシージャを `CREATE PROCEDURE` ステートメントで登録します。
3. `CALL` ステートメントでストアド・プロシージャを呼び出す。

プロシージャの参照

修飾名 (スキーマおよびストアド・プロシージャ名) と、その後続く括弧で囲まれた引数リストでストアド・プロシージャを参照している `CALL` ステートメントによってストアド・プロシージャは呼び出されます。また、スキーマ名を使用しないでストアド・プロシージャを呼び出すこともできます。その場合、同数のパラメーターをもった別のスキーマ内の選択可能なストアド・プロシージャを選ぶことになります。

ストアド・プロシージャに渡されるどのパラメーターも、ホスト変数、パラメーター・マーカ、または `NULL` で構成することができます。以下に、ストアド・プロシージャのパラメーターに関する制約事項を示します。

- `OUT` および `INOUT` パラメーターはホスト変数でなければなりません。
- `SQL` データ・タイプが Java クラス・タイプにマップされていない限り、`NULL` を Java ストアド・プロシージャに渡すことはできません。
- `PARAMETER STYLE GENERAL` ストアド・プロシージャに `NULL` を渡すことはできません。

引数の位置は重要で、それをセマンティクスするストアド・プロシージャの定義に正確に従っていなければなりません。引数の位置と、ストアド・プロシージャの定義の両方がストアド・プロシージャ本体に従っていなければなりません。DB2 は、引数がストアド・プロシージャの定義とうまく一致するように、引数を入れ替えたりはしません。また、DB2 はストアド・プロシージャのそれぞれのパラメーターのセマンティクスを理解していません。

プロシージャの呼び出し

プロシージャ (ストアド・プロシージャともいう) を作成するために必要なアクティビティーが完了したなら、`CALL` ステートメントを使用して、プロシージャ

ーを呼び出せます。CALL ステートメントは、プロシージャを呼び出し、プロシージャにパラメーターを渡し、プロシージャから戻されるパラメーターを受け取ることを可能にするための SQL ステートメントです。プロシージャから戻されるアクセス可能な結果セットの処理はすべて、プロシージャが正常に戻された後に行えます。プロシージャは、CALL ステートメントがサポートされているところであればどこからでも呼び出せます。例えば、以下から呼び出すことができます。

- 組み込み SQL クライアント・アプリケーション
- 外部ルーチン (プロシージャ、UDF、またはメソッド)
- SQL ルーチン (プロシージャ、UDF、またはメソッド)
- SQL トリガー (BEFORE トリガー、AFTER トリガー、または INSTEAD OF トリガー)
- SQL 動的コンパウンド・ステートメント
- コマンド行プロセッサ (CLP) から

クライアント・アプリケーションまたは外部ルーチンからプロシージャを呼び出す場合、そのクライアント・アプリケーションまたは外部ルーチンは、プロシージャの言語以外の言語で作成してもかまいません。例えば、C++ で作成したクライアント・アプリケーションから、CALL ステートメントを使用して、Java で作成したプロシージャを呼び出すことができます。したがって、プログラマーにとっては、自分の好みの言語でプログラミングを行い、さまざまな言語で作成したコードの断片を統合する、という柔軟な作業が可能になります。

さらに、プロシージャが置かれているものとは異なるプラットフォーム上で、プロシージャを呼び出すクライアント・アプリケーションを実行してもかまいません。例えば、Windows プラットフォーム上で実行しているクライアント・アプリケーションから、CALL ステートメントを使用して、Linux データベース・サーバー上にあるプロシージャを呼び出すことができます。

自律型プロシージャとは、呼び出されたときに、元のトランザクションとは独立して、新しいトランザクションの内部で実行されるプロシージャのことです。自律型プロシージャは、正常に完了すると、プロシージャ内で実行された作業をコミットしますが、正常に完了しなかった場合は、実行した作業があってもそれをロールバックします。自律型プロシージャの結果がどのようなものであっても、自律型プロシージャを呼び出した側のトランザクションは影響を受けません。プロシージャを自律型として指定するには、CREATE PROCEDURE ステートメントで AUTONOMOUS キーワードを指定します。

プロシージャを呼び出すとき、プロシージャを的確に選択するためにいくつかの規則が適用されます。スキーマ名を指定してプロシージャを修飾するかどうかによって、プロシージャの選択が異なってきます。プロシージャの呼び出しで指定するパラメーターの数や種類に基づいて DB2 データベース・マネージャーもチェックを行います。プロシージャの選択について詳しくは、CALL ステートメントに関する情報を参照してください。

アプリケーションまたは外部ルーチンからのプロシージャの呼び出し

クライアント・アプリケーションまたは外部ルーチンと関連したアプリケーションからのロジックをカプセル化したプロシージャ (ストアド・プロシージャともいう) の呼び出しは、アプリケーション内の単純なセットアップ作業と、CALL ステートメントの使用によって簡単に行えます。

CREATE PROCEDURE ステートメントを実行して、データベース内にプロシージャを作成しておく必要があります。

外部プロシージャの場合は、CREATE PROCEDURE ステートメントの EXTERNAL 節で指定するロケーションに、ライブラリー・ファイルまたはクラス・ファイルを配置する必要があります。

プロシージャの呼び出し側には、CALL ステートメントの実行に必要な特権が必要です。この場合のプロシージャの呼び出し側は、アプリケーションを実行するユーザー ID ですが、そのアプリケーションで DYNAMICRULES BIND オプションを使用する場合は、特別な規則が適用されます。

アプリケーションからプロシージャを呼び出す場合は、そのアプリケーションに特定の要素を組み込む必要があります。アプリケーションの作成時に、以下の作業を行ってください。

1. オプションのデータ構造および CALL ステートメントに必要なホスト変数またはパラメーター・マーカのストレージを宣言し、割り振り、初期化します。そのためには、次のようにします。
 - プロシージャの各パラメーターで使用するホスト変数またはパラメーター・マーカを割り当てます。
 - IN または INOUT パラメーターに対応するホスト変数またはパラメーター・マーカを初期化します。
2. データベース接続を確立します。そのためには、組み込み SQL 言語 CONNECT TO ステートメントを実行するか、暗黙的なデータベース接続をコーディングします。
3. プロシージャの呼び出しをコーディングします。プロシージャ呼び出しのコーディングは、データベース接続のコーディングの後に行います。そのためには、SQL 言語 CALL ステートメントを実行します。プロシージャが予想する IN、INOUT、OUT の各パラメーターに、ホスト変数、定数、パラメーター・マーカのいずれかを確実に指定してください。
4. OUT パラメーターと INOUT パラメーターと結果セットを処理するコードを追加します。このコードは、CALL ステートメントの実行の後に記述する必要があります。
5. データベースの COMMIT または ROLLBACK をコーディングします。CALL ステートメントの実行と、プロシージャによって戻される出力パラメーター値またはデータの評価の後に、アプリケーションからトランザクションのコミットまたはロールバックを実行することもできます。そのためには、COMMIT ステートメントまたは ROLLBACK ステートメントを組み込みます。プロシージャ

ーに COMMIT ステートメントまたは ROLLBACK ステートメントを組み込むこともできますが、トランザクションの管理は、クライアント・アプリケーション内で行うことをお勧めします。

注: データベースへのタイプ 2 接続を確立したアプリケーションから呼び出すプロシージャでは、COMMIT ステートメントまたは ROLLBACK ステートメントを発行できません。

6. データベースへの接続を切断します。
7. アプリケーションの準備、コンパイル、リンク、バインドを実行します。アプリケーションが外部ルーチン用の場合は、CREATE ステートメントを発行してルーチンを作成し、オペレーティング・システムの適切な関数パスに外部コード・ライブラリーを配置して、データベース・マネージャーがそのライブラリーを見つけられるようにします。
8. アプリケーションを実行するか、外部ルーチンを呼び出します。アプリケーションに組み込んだ CALL ステートメントが呼び出されます。

注: SQL ステートメントとルーチン・ロジックは、ステップ 2 から 5 までの任意の時点でコーディングできます。

トリガーまたは SQL ルーチンからのプロシージャの呼び出し

プロシージャの呼び出しは、SQL ルーチン、トリガー、動的コンパウンド・ステートメントのどれから行う場合も基本的に同じです。この呼び出しのインプリメントには、どの場合も同じステップを使用します。このトピックでは、トリガーのシナリオを使いながら、そのステップを説明します。ルーチンまたは動的コンパウンド・ステートメントからのプロシージャの呼び出しに固有の前提条件やステップについても取り上げます。

前提条件

- CREATE PROCEDURE ステートメントを実行して、データベース内にプロシージャを作成しておく必要があります。
- 外部プロシージャの場合は、CREATE PROCEDURE ステートメントの EXTERNAL 節で指定するロケーションに、ライブラリー・ファイルまたはクラス・ファイルを配置する必要があります。
- CALL ステートメントを含むトリガーの作成者には、CALL ステートメントの実行特権が必要です。実行時にトリガーが活動化される時点で、トリガーの作成者の許可に関して、CALL ステートメントの実行特権があるかどうかのチェックが行われます。CALL ステートメントを含む動的コンパウンド・ステートメントを実行するユーザーには、そのプロシージャの CALL ステートメントの実行特権が必要です。
- トリガーを呼び出すユーザーには、そのトリガー・イベントに関連したデータ変更ステートメントの実行特権が必要です。同様に、SQL ルーチンまたは動的コンパウンド・ステートメントを正常に呼び出すには、そのルーチンに関する EXECUTE 特権が必要です。

SQL トリガー、SQL ルーチン、動的コンパウンド・ステートメントのいずれかからプロシージャを呼び出すときには、以下の制約事項が適用されます。

- パーティション・データベース環境では、トリガーまたは SQL UDF からプロシージャを呼び出せません。

- 対称マルチプロセッサ (SMP) マシンでは、トリガーからのプロシージャー呼び出しが 1 つのプロセッサで実行されます。
- トリガーから呼び出すプロシージャーには、COMMIT ステートメント、または作業単位のロールバックを試行する ROLLBACK ステートメントを組み込みません。ROLLBACK TO SAVEPOINT ステートメントはプロシージャー内で使用できますが、指定のセーブポイントがプロシージャー内に存在する必要があります。
- トリガーから CALL ステートメントをロールバックしても、ファイル・システムへの書き込みなど、プロシージャーの結果として生じた外部アクションはロールバックできません。
- プロシージャーによってフェデレーテッド表を変更してはなりません。つまり、ニックネームの検索 UPDATE、ニックネームの検索 DELETE、ニックネームの検索 INSERT をプロシージャーに含めてはなりません。
- プロシージャーに指定した結果セットにはアクセスできません。

アクセス・レベル MODIFIES SQL DATA で作成したプロシージャーを参照する CALL ステートメントを含んだ BEFORE トリガーは作成できません。そのようなトリガーの CREATE TRIGGER ステートメントを実行すると、エラー (SQLSTATE 42987) になります。ルーチンの SQL アクセス・レベルの詳細については、以下を参照してください。

- 45 ページの『ルーチンの SQL アクセス・レベル』
- 39 ページの『ルーチン内で実行できる SQL ステートメント』

この手順のセクションでは、CALL ステートメントを含んだトリガーを作成する方法と呼び出す方法を説明します。プロシージャーをトリガーから呼び出すのに必要な SQL は、プロシージャーを SQL ルーチンや動的コンパウンド・ステートメントから呼び出すのに必要な SQL と同じです。

1. 必要なトリガー属性を指定した基本 CREATE TRIGGER ステートメントを作成します。CREATE TRIGGER ステートメントを参照してください。
2. トリガーのトリガー・アクションの部分で、プロシージャーが指定する IN、INOUT、OUT の各パラメーター用の SQL 変数を宣言できます。DECLARE ステートメントを参照してください。これらの変数の初期化や設定の方法については、代入文を参照してください。プロシージャーのパラメーターとして、トリガーの遷移変数を使用することもできます。
3. トリガーのトリガー・アクションの部分に、プロシージャーの CALL ステートメントを追加します。プロシージャーの IN、INOUT、OUT の各パラメーター用の値または式を指定します。
4. SQL プロシージャーの場合、オプションで GET DIAGNOSTICS ステートメントを使用して、プロシージャーの戻り状況を収集することもできます。そのためには、戻り状況を保持するための整数タイプの変数を使用する必要があります。GET DIAGNOSTICS ステートメントは、CALL ステートメントの直後にそのまま記述します。このステートメントは、ローカル・トリガー戻り状況変数に RETURN_STATUS を割り当てます。

5. CREATE TRIGGER ステートメントの作成が完了したなら、そのステートメントを静的に (アプリケーション内から) または動的に (CLP またはコントロール・センターから) 実行することによって、データベース内にトリガーを作成します。
6. トリガーを呼び出します。そのためには、トリガー・イベントに対応した適切なデータ変更ステートメントを実行します。
7. 表に関するデータ変更ステートメントを実行すると、その表に対して定義されている適切なトリガーが起動します。トリガー・アクションが実行されると、その中に含まれている SQL ステートメント (CALL ステートメントも含む) が実行されます。

ランタイム・エラー

トリガーが読み取り/書き込みを行う表に対してプロシージャも読み取り/書き込みを行おうとすると、読み取り/書き込みの競合が検出された場合にエラーが発生します。トリガーが変更する表 (トリガーの対象として定義した表を含む) のセットは、プロシージャによって変更される表とは区別しなければなりません。

例: トリガーからの SQL プロシージャの呼び出し

この例では、CALL ステートメントを組み込んでトリガー内のプロシージャを呼び出す方法と、GET DIAGNOSTICS ステートメントを使用してプロシージャ呼び出しの戻り状況を収集する方法を示します。以下の SQL によって、必要な表、SQL PL 言語プロシージャ、AFTER トリガーが作成されます。

```
CREATE TABLE T1 (c1 INT, c2 CHAR(2))@
CREATE TABLE T2 (c1 INT, c2 CHAR(2))@

CREATE PROCEDURE proc(IN val INT, IN name CHAR(2))
LANGUAGE SQL
DYNAMIC RESULTSETS 0
MODIFIES SQL DATA
BEGIN
  DECLARE rc INT DEFAULT 0;
  INSERT INTO TABLE T2 VALUES (val, name);
  GET DIAGNOSTICS rc = ROW_COUNT;
  IF ( rc > 0 ) THEN
    RETURN 0;
  ELSE
    RETURN -200;
  END IF;
END@

CREATE TRIGGER trig1 AFTER UPDATE ON t1
REFERENCING NEW AS n
FOR EACH ROW
WHEN (n.c1 > 100);
BEGIN ATOMIC
  DECLARE rs INTEGER DEFAULT 0;
  CALL proc(n.c1, n.c2);
  GET DIAGNOSTICS rs = RETURN_STATUS;
  VALUES(CASE WHEN rc < 0 THEN RAISE_ERROR('70001', 'PROC CALL failed'));
END@
```

以下の SQL ステートメントを実行すると、トリガーが起動し、プロシージャが呼び出されます。

```
UPDATE T1 SET c1 = c1+1 WHERE c2 = 'CA'@
```


コマンド行プロセッサ (CLP) からのプロシージャの呼び出し

DB2 コマンド行プロセッサ・インターフェースから、CALL ステートメントを使用してストアード・プロシージャを呼び出せます。呼び出すストアード・プロシージャは、DB2 システム・カタログ表で定義されていなければなりません。

ストアード・プロシージャを呼び出すには、まずデータベースに接続します。

```
db2 connect to sample user userid using password
```

ここで、*userid* と *password* は、sample データベースが置かれているインスタンスのユーザー ID とパスワードを表します。

CALL ステートメントを使用するには、ストアード・プロシージャ名、IN または INOUT パラメーター値、および各 OUT パラメーター値のプレースホルダーとしての '?' を入力します。

ストアード・プロシージャのパラメーターは、プログラム・ソース・ファイル内のストアード・プロシージャの CREATE PROCEDURE ステートメントで指定します。

SQL プロシージャの例

例 1。

whiles.db2 ファイルにある、DEPT_MEDIAN プロシージャ・シグニチャーの CREATE PROCEDURE ステートメントは、次のとおりです。

```
CREATE PROCEDURE DEPT_MEDIAN  
(IN deptNumber SMALLINT, OUT medianSalary DOUBLE)
```

このプロシージャを呼び出すには、CALL ステートメントを使用し、そこでプロシージャ名と適切なパラメーター引数を指定する必要があります。この場合は IN パラメーターの値と、OUT パラメーターの値の疑問符「?」を指定します。プロシージャの SELECT ステートメントは、STAFF 表の DEPT 列の値 deptNumber を使用するため、意味のある出力を得るには IN パラメーターが DEPT 列からの有効な値でなければなりません。例えば、値 "51" の場合は、次のようにします。

```
db2 call dept_median (51, ?)
```

注: UNIX プラットフォームでは、括弧はコマンド・シェルに対して特別な意味を持つので、括弧の前に " " 文字を入れるか、次のように引用符で囲む必要があります。

```
db2 "call dept_median (51, ?)"
```

コマンド行プロセッサの対話モードを使用している場合は、引用符を使用しません。

上記のコマンドを実行すると、次の結果を受け取ります。

```
Value of output parameters  
-----  
Parameter Name : MEDIANSALARY  
Parameter Value : +1.76545000000000E+004  
  
Return Status = 0
```

例 2。

この例は、配列パラメーターを使用したプロシージャの呼び出し方法を示しています。タイプ `phonenumbers` は、以下のように定義されます。

```
CREATE TYPE phonenumbers AS VARCHAR(12) ARRAY[1000]
```

以下に定義されているプロシージャ `find_customers` には、タイプ `phonenumbers` の `IN` および `OUT` パラメーターがあります。このプロシージャは、指定の `area_code` で始める `numbers_in` 内の番号を検索して、それを `numbers_out` に報告します。

```
CREATE PROCEDURE find_customers(  
  IN numbers_in phonenumbers,  
  IN area_code CHAR(3),  
  OUT numbers_out phonenumbers)  
BEGIN  
  DECLARE i, j, max INTEGER;  
  
  SET i = 1;  
  SET j = 1;  
  SET numbers_out = NULL;  
  SET max = CARDINALITY(numbers_in);  
  
  WHILE i <= max DO  
    IF substr(numbers_in[i], 1, 3) = area_code THEN  
      SET numbers_out[j] = numbers_in[i];  
      SET j = j + 1;  
    END IF;  
    SET i = i + 1;  
  END WHILE;  
END
```

このプロシージャを呼び出すには、以下の `CALL` ステートメントを使用できます。

```
db2 CALL find_customers(ARRAY['416-305-3745',  
                              '905-414-4565',  
                              '416-305-3746'],  
                        '416',  
                        ?)
```

`CALL` ステートメントに示されているように、プロシージャに配列データ・タイプの入力パラメーターがある場合、リテラル値のリストが含まれる配列コンストラクターを使用して入力引数を指定できます。

このコマンドを実行すると、次のような結果を受け取ります。

```
Value of output parameters  
-----  
Parameter Name : OUT_PHONENUMBERS  
Parameter Value : ['416-305-3745',  
                  '416-305-3746']  
  
Return Status = 0
```

C ストアード・プロシージャの例

コマンド行プロセッサから、サポートされているホスト言語で作成したストアード・プロシージャを呼び出すこともできます。DB2 では、UNIX の `samples/c` ディレクトリーと、Windows の `samples%c` ディレクトリーに、ストアード・プロシージャを作成するためのファイルを用意していません。spserver 共用ライブラリーには、ソース・ファイル `spserver.sqc` か

ら作成できる、いくつかのストアド・プロシージャが入っています。ストアド・プロシージャをカタログに登録するには、 `screate.db2` ファイルを使用します。

`screate.db2` ファイル内の `MAIN_EXAMPLE` プロシージャの `CREATE PROCEDURE` ステートメントは次のように始まります。

```
CREATE PROCEDURE MAIN_EXAMPLE (IN job CHAR(8),
                                OUT salary DOUBLE,
                                OUT errorcode INTEGER)
```

このストアド・プロシージャを呼び出すには、 `IN` パラメーター `job` に `CHAR` 値を、また各 `OUT` パラメーターに疑問符「?」を入力する必要があります。プロシージャの `SELECT` ステートメントは、 `EMPLOYEE` 表の `JOB` 列の値 `job` を使用するため、意味のある出力を得るには、 `IN` パラメーターが `JOB` 列からの有効な値でなければなりません。 `C` サンプル・プログラム `spclient` は、ストアド・プロシージャを呼び出して、 `JOB` 値に `'DESIGNER'` を使用します。次のように指定して、同じ内容を実行できます。

```
db2 "call MAIN_EXAMPLE ('DESIGNER', ?, ?)"
```

上記のコマンドを実行すると、次の結果を受け取ります。

```
Value of output parameters
-----
Parameter Name : SALARY
Parameter Value : +2.37312500000000E+004

Parameter Name : ERRORCODE
Parameter Value : 0

Return Status = 0
```

`ERRORCODE` がゼロであることは、成功を意味します。

CLI アプリケーションからのストアド・プロシージャの呼び出し

CLI アプリケーションは、 `CALL` プロシージャ `SQL` ステートメントを実行することにより、ストアド・プロシージャを呼び出します。このトピックでは、CLI アプリケーションからストアド・プロシージャを呼び出す方法を説明します。

ストアド・プロシージャを呼び出す前に、CLI アプリケーションを初期設定しておくようにします。

呼び出されるストアド・プロシージャがカタログされていない場合、CLI スキーマ関数のいずれも呼び出さないことを確認してください。カタログされていないストアド・プロシージャからの CLI スキーマ関数の呼び出しはサポートされていません。

CLI スキーマ関数は、以下のとおりです。 `SQLColumns()`、`SQLColumnPrivileges()`、`SQLForeignKeys()`、`SQLPrimaryKeys()`、`SQLProcedureColumns()`、`SQLProcedures()`、`SQLSpecialColumns()`、`SQLStatistics()`、`SQLTables()`、および `SQLTablePrivileges()`。

ストアド・プロシージャを呼び出すには、以下のようになります。

1. ストアド・プロシージャの IN、INOUT、および OUT パラメーターにそれぞれ対応するアプリケーション・ホスト変数を宣言します。アプリケーションの変数データのタイプと長さが、ストアド・プロシージャのシグニチャーのデータ・タイプと引数の長さに一致することを確認します。DB2 CLI は、すべての SQL タイプをパラメーター・マーカースとして使用して、ストアド・プロシージャを呼び出すことをサポートしています。
2. IN、INOUT、および OUT パラメーターのアプリケーション変数を初期設定しません。
3. CALL SQL ステートメントを発行します。以下に例を示します。

```
SQLCHAR *stmt = (SQLCHAR *)"CALL OUT_LANGUAGE (?);"
```

または

```
SQLCHAR *stmt = (SQLCHAR *)"CALL OUT_LANGUAGE (:language);"
```

要確認: 名前付きパラメーター・マーカ (`:language` など) を使用するには、`EnableNamedParameterSupport` 構成キーワードを `TRUE` に設定することによって、名前付きパラメーターの処理を明示的に有効にする必要があります。

パフォーマンスを最高にするために、アプリケーションでは、CALL プロシージャ・ストリングの中でストアド・プロシージャ引数のパラメーター・マーカを使用してから、ホスト変数をこれらのパラメーター・マーカにバインドする必要があります。ただし、インバウンド・ストアド・プロシージャ引数を、パラメーター・マーカではなく、ストリング・リテラルとして指定しなければならない場合、CALL プロシージャ・ステートメントに、ODBC 呼び出しエスケープ節の区切り文字 `{ }` を含めます。以下に例を示します。

```
SQLCHAR *stmt = (SQLCHAR *)"{CALL IN_PARAM (123, 'Hello World!')}";
```

CALL プロシージャ・ステートメントでストリング・リテラルおよび ODBC エスケープ節が使用される場合、IN モード・ストアド・プロシージャ引数として、ストリング・リテラルだけを指定できます。INOUT および OUT モード・ストアド・プロシージャ引数は、引き続きパラメーター・マーカを使用して指定する必要があります。

4. オプション: `SQLPrepare()` を呼び出して CALL ステートメントを準備します。
5. `SQLBindParameter()` を呼び出して、CALL プロシージャ・ステートメントの各パラメーターをバインドします。

注: 各パラメーターが (`SQL_PARAM_INPUT`、`SQL_PARAM_OUTPUT`、または `SQL_PARAM_INPUT_OUTPUT` に対して) 正しくバインドされたことを確認します。正しくバインドされていないと、CALL プロシージャ・ステートメントが実行されるときに、予期しない結果が生じる可能性があります。たとえば、入力パラメーターが、`SQL_PARAM_OUTPUT` の `InputOutputType` を使用して、不正確にバインドされる場合に、このことが生じます。

6. `SQLExecDirect()` を使用して CALL プロシージャ・ステートメントを実行するか、ステップ 4 で CALL プロシージャ・ステートメントを準備済みの場合には、`SQLExecute()` を使用して実行します。

注: ストアド・プロシージャを呼び出したアプリケーションかスレッドが、そのストアド・プロシージャの完了前に終了する場合、ストアド・プロシ

ージャの実行も終了します。ストアード・プロシージャが早めに終了してしまう場合にも、データベースは一貫した状態と望ましい状態を保つようなロジックを、そのストアード・プロシージャに含めることは大切です。

7. 関数が戻されるときに `SQLExecDirect()` または `SQLExecute()` の戻りコードを調べ、`CALL` プロシージャ・ステートメントまたはストアード・プロシージャのいずれかの実行時に、何らかのエラーが発生していないかを判別します。戻りコードが `SQL_SUCCESS_WITH_INFO` か `SQL_ERROR` である場合、CLI 診断関数 `SQLGetDiagRec()` および `SQLGetDiagField()` を使用して、エラーが発生した理由を判別します。

ストアード・プロシージャを正常に実行した場合、`OUT` パラメーターとしてバインドされた変数には、そのストアード・プロシージャが CLI アプリケーションに戻したデータが含まれる可能性があります。該当する場合には、ストアード・プロシージャは、スクロール不可カーソルを使用して、1 つ以上の結果セットを戻す場合もあります。CLI アプリケーションでは、`SELECT` ステートメントの実行によって生成された結果セットを処理するときに、ストアード・プロシージャの結果セットを処理する必要があります。

注: CLI アプリケーションが、ストアード・プロシージャによって戻された結果セットに示された、パラメーターの番号またはタイプが分からない場合、その結果セットに対して、`SQLNumResultCols()`、`SQLDescribeCol()`、および `SQLColAttribute()` 関数を (この順序で) 呼び出して、この情報を判別することができます。

`CALL` ステートメントを実行したら、該当する場合には、ストアード・プロシージャから結果セットを検索できます。

注:

値が ISO 形式で戻されない場合、DB2 CLI アプリケーションに戻されるプロシージャ結果セットの中で、`DATETIME` データ・タイプ値の数値の月日の部分になります。たとえば、ローカル形式が代わりに使用される場合に、これが発生する可能性があります。`DATETIME` データ・タイプ値の情報がクライアント・アプリケーションによって確実に正しく解釈されるようにするには、ローカルに依存しない `DATETIME` 形式 (たとえば ISO) を使用するデータベースにプロシージャをバインドする必要があります。以下に例を示します。

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

注:

データベースの作成またはアップグレード時に、DB2 CLI パッケージは、自動的にデータベースにバインドされます。

プロシージャの結果セット

プロシージャは、パラメーターの交換に加えて、結果セットを返送して呼び出し側に情報を渡すことができます。結果セットは、SQL 本体をもったルーチンによって、以下のインターフェースでプログラムされたルーチンとアプリケーションによって受け入れられます。

- CLI

- JDBC
- SQLJ
- ODBC

ストアド・プロシージャは、カーソルを介して呼び出し側に結果セットを渡します。プロシージャの本体には、返送の必要な各結果セットのそれぞれのカーソルが入っていないければなりません。プロシージャ内の結果セットのカーソルから行を取り出すことができる一方で、取り出されなかった行だけが、結果セットとして呼び出し側に渡されます。プロシージャを終了するときは、結果セットに対応するカーソルはオープンしたままにします。結果セットが複数個ある場合、それぞれに対応するカーソルがオープンされた順序で戻されます。

結果セットのカーソルを宣言する場合、`DECLARE CURSOR` ステートメントの `WITH RETURN TO` 節内に宛先を指定することを強くお勧めします (SQL プロシージャの場合はこれは必須です)。結果セットを呼び出し側に戻す場合は、その呼び出し側がアプリケーションまたはルーチンのどちらであっても、`WITH RETURN TO CALLER` を指定します。中間のネスト・ルーチンをすべて迂回して、結果セットをアプリケーションに直接戻す場合は、`WITH RETURN TO CLIENT` を指定します。外部ルーチンではカーソルは、`WITH RETURN TO CLIENT` と明示的に定義されない限り、デフォルトによって `WITH RETURN TO CALLER` と定義されます。

`CREATE PROCEDURE` ステートメントを使用してプロシージャを登録するときは、`DYNAMIC RESULT SETS` 節を使用して、そのプロシージャから戻す結果セットの数を指定します。その値は、`SYSCAT.ROUTINES` ビューの `RESULT_SETS` 列に保管されます。プロシージャから戻された結果セットの数が、`CREATE PROCEDURE` ステートメントに指定された数と異なると、警告が出されます (SQLCODE +464, SQLSTATE 0100E)。`PARAMETER STYLE JAVA` ストアド・プロシージャの場合、`CREATE PROCEDURE` ステートメントに指定する結果セットの数は、Java メソッド・シグニチャー内の `ResultSet[]` パラメーターの数に一致していなければなりません。

呼び出し側は結果セットを記述することができます。1つのカーソルが複数のネスト・レベルにまたがってオープンされる場合、DB2 Universal Database バージョン 7 クライアント上で稼働しているアプリケーションは、オープンしている最初の結果セットしか記述できないことに注意してください。

結果セットは、呼び出し側によってシリアル方式で処理される必要があります (呼び出し側が SQL を本体とするルーチンでない場合)。カーソルは、最初の結果セットに対して自動的にオープンされますが、ある結果セット上のカーソルをクローズし、次の結果セット上でそれをオープンするための特別な呼び出し (DB2 CLI の場合は `SQLMoreResults`、JDBC の場合は `getMoreResults`、SQLJ の場合は `getNextResultSet`) が用意されています。

SQL を本体とするルーチン内で結果セットを受け取るには、結果セットのロケータを宣言して、そのロケータを、結果セットの返送元になる予定のプロシージャに関連付けなければなりません。次に、結果セット・ロケータに戻されることになる各カーソルを割り振る必要があります。それが完了したなら、結果セットから行を取り出すことができます。

トリガー、動的コンパウンド・ステートメント、SQL 関数、SQL メソッドのいずれかの中からプロシージャを呼び出す場合は、どの結果セットにもアクセスできません。

注: プロシージャまたはアプリケーションから COMMIT を発行すると、WITH HOLD カーソルの対象ではない結果セットはすべてクローズされます。アプリケーションまたはストアード・プロシージャから ROLLBACK を発行すると、すべての結果セット・カーソルがクローズされます。プロシージャから COMMIT または ROLLBACK を発行し終わったら、カーソルをオープンすることも、または結果セットとして戻すこともできます。

SQL データ変更ステートメントからの結果セットの検索: INSERT、UPDATE、または DELETE ステートメントによって表を変更するアプリケーションは、変更行に追加の処理を必要とする場合があります。この処理を簡単にするために、SELECT および SELECT INTO ステートメントの FROM 節に、SQL データ変更操作を組み込むことができます。単一作業単位内で、アプリケーションは、SQL データ変更操作によって変更された表またはビューからの変更行を含む結果セットを検索できます。

例えば、次のステートメントは、SAMPLE データベースにある EMPLOYEE 表の給料のすべての記録を更新し、次いですべての更新済みの行に従業員番号および新規給料を戻します。

```
SELECT empno, salary FROM FINAL TABLE
(UPDATE employee SET salary = salary * 1.10 WHERE job = 'CLERK')
```

正常にデータを戻すために、FROM SQL データ変更操作の結果セットを検索する SELECT ステートメントは、正常に実行するために SQL データ変更操作を必要とします。正常な SQL データ変更操作では、該当する場合、すべての制約およびトリガーの処理を組み込みます。

例えば、EMPLOYEE 表に対して SELECT 特権はあるが INSERT 特権はないユーザーが、EMPLOYEE 表に対して SELECT FROM INSERT ステートメントを試行すると想定します。INSERT 操作は特権がないために失敗し、結果として SELECT ステートメント全体が失敗します。

次の照会を考慮してください。EMPLOYEE 表からの記録が選択され、EMP という名前の別の表に挿入されます。この SELECT ステートメントは失敗します。

```
SELECT empno FROM FINAL TABLE
(INsert INTO emp(name, salary)
SELECT firstnme || midinit || lastname, salary
FROM employee)
```

EMPLOYEE 表に行が 100 あり、90 行目に \$9,999,000.00 の SALARY 値がある場合、\$10,000.00 の追加は 10 進数のオーバーフローを引き起こす原因となります。オーバーフローが発生すると、データベース・マネージャーは EMP 表への追加をロールバックすることになります。

中間結果表

SELECT ステートメントの FROM 節で SQL データ変更操作によってターゲットにされた表またはビューの変更行は、中間結果表を構成します。中間結果表は、SQL データ変更操作で定義された列を組み込むことに加えて、ターゲット表または

ビューのすべての列を組み込みます。選択リスト、ORDER BY 節、または WHERE 節内の名前によって中間結果表のすべての列を参照できます。

中間結果表の内容は、FROM 節で指定された修飾子に依存しています。中間結果表として結果セットを検索する SELECT ステートメント内の、次の FROM 節修飾子の 1 つを組み込む必要があります。

OLD TABLE

中間結果表の行は、BEFORE トリガーおよび SQL データ変更操作の実行の直前のターゲット表の行の値を含みます。OLD TABLE 修飾子は UPDATE および DELETE 操作を適用します。

NEW TABLE

中間結果表の行には、SQL データ変更ステートメントが実行された直後、および参照整合性評価といずれかの AFTER トリガーの実行前のターゲット表の行の値が入ります。NEW TABLE 修飾子は UPDATE および INSERT 操作を適用します。

FINAL TABLE

この修飾子は、NEW TABLE と同じ中間結果表に戻ります。さらに、AFTER トリガーまたは参照整合性制約がない FINAL TABLE を使用すると、UPDATE または INSERT 操作のターゲットがさらに変更されることが保証されます。FINAL TABLE 修飾子は UPDATE および INSERT 操作を適用します。

FROM 節の修飾子は、中間結果表にどのバージョンのターゲット・データがあるかを判別します。これらの修飾子は、ターゲット表の行の追加、削除、または更新には影響しません。

ターゲット表とビュー

FROM SQL データ変更操作の結果セットを選択する場合、ターゲットは表またはビューにすることができます。

ビューに対する SQL データ変更操作で、結果表には、NEW TABLE と FINAL TABLE のビュー定義を満たさなくなった行を組み込むことはできません。SELECT ステートメントのビューを参照する INSERT または UPDATE ステートメントを組み込む場合、ビューは WITH CASCADED CHECK OPTION として定義される必要があります。代替として、ビューは、WITH CASCADED CHECK OPTION としてのビューの定義を許可する制限を満たす必要があります。

SELECT ステートメントの FROM 節に組み込まれた SQL データ変更操作のターゲットが全選択である場合、結果表には、全選択に該当しない行が組み込まれることがあります。これは、WHERE 節の述部が更新済みの値に対して再評価されないためです。

INPUT SEQUENCE に基づく結果セットのソート

ターゲット表またはビューに挿入されたのと同じ ORDER BY で行を SELECT するには、ORDER BY 節で INPUT SEQUENCE キーワードを使用します。INPUT SEQUENCE キーワードの使用によって、行がその提供と同じ順序で挿入されることはありません。

以下に示すのは、INSERT 操作の結果セットをソートするため、ORDER BY 節で INPUT SEQUENCE キーワードを使用する例です。

```
CREATE TABLE orders (purchase_date DATE,
                      sales_person VARCHAR(16),
                      region VARCHAR(10),
                      quantity SMALLINT,
                      order_num INTEGER NOT NULL
                      GENERATED ALWAYS AS IDENTITY (START WITH 100,
                                                    INCREMENT BY 1))

SELECT * FROM FINAL TABLE
  (INSERT INTO orders
   (purchase_date, sales_person, region, quantity)
  VALUES (CURRENT DATE,'Judith','Beijing',6),
          (CURRENT DATE,'Marieke','Medway',5),
          (CURRENT DATE,'Hanneke','Halifax',5))
 ORDER BY INPUT SEQUENCE
```

PURCHASE_DATE	SALES_PERSON	REGION	QUANTITY	ORDER_NUM
07/18/2003	Judith	Beijing	6	100
07/18/2003	Marieke	Medway	5	101
07/18/2003	Hanneke	Halifax	5	102

列の組み込みを使用して結果セットの行をソートすることもできます。

カーソルを使用した SQL データ変更ステートメントからの結果セットの検索:

SQL データ変更操作から結果セットを検索する照会に対して、カーソルを宣言することができます。以下に例を示します。

```
DECLARE C1 CURSOR FOR SELECT salary FROM FINAL TABLE
  (INSERT INTO employee (name, salary, level)
   SELECT name, income, band FROM old_employee)
```

定義に SQL データ変更操作を含むカーソルからの取り出しでエラーが生じて、変更された行のロールバックは実行されません。このエラーによってカーソルが閉じたとしても、アプリケーションがカーソルを開くときには、それらは完了しているので、行は変更された状態のままになります。

そのようにカーソルが開く時に、データベース・マネージャーは完全に SQL データ変更操作を実行し、結果セットは一時表に保管されます。カーソルが開いている間にエラーが生じた場合、SQL データ変更操作によって加えられた変更はロールバックされます。ターゲット表またはビューのそれ以降の更新は、SQL データ変更操作からの結果セットを検索するカーソルの結果表の行には表示されません。例えば、アプリケーションはカーソルを宣言し、カーソルを開き、取り出しを実行し、表を更新し、追加の行を取り出します。UPDATE ステートメント後の取り出しは、UPDATE ステートメント前のオープン・カーソル処理中に決定されたこれらの値を戻します。

SQL データ変更操作からの結果セットを検索する照会に対して、両方向スクロール・カーソルを宣言することができます。カーソルを OPEN にする場合には結果表が生成されるため、データの変更はすでにターゲット表またはビューに書き込まれています。SQL データ変更操作から行を選択する照会を持つカーソルは、INSENSITIVE または ASENSITIVE として定義される必要があります。

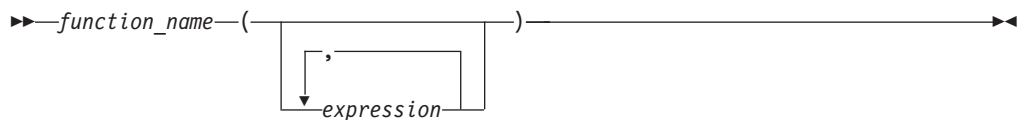
注: 両方向スクロール・カーソルは、CLI、JDBC、および SQLJ アプリケーションによってのみサポートされます。

WITH HOLD オプションと、COMMIT を実行するアプリケーションがあるカーソルを宣言する場合、すべてのデータ変更はコミットされます。WITH HOLD として宣言しないカーソルは、同じ方法で振る舞います。すべてのカーソルに対して、照会に組み込まれた SQL データ変更操作は、どの行が取り出されるよりも前に完全に評価されます。

OPEN CURSOR ステートメントに対して明示的なロールバックを実行する場合、または OPEN CURSOR ステートメントより前のセーブポイントまでロールバックする場合、カーソルに対するすべてのデータ変更は取り消されます。SQL データ変更操作からの結果セットを検索する照会付きのカーソルの場合、すべてのデータ変更はロールバック後に取り消されますが、カーソルは保持され、事前に挿入されていた行は引き続き取り出すことができます。

関数の参照

各関数の参照には、UDF または組み込み関数のいずれの場合も次の構文が含まれています。



前記の構文図の `function_name` は、非修飾または修飾のどちらの関数名でもかまいません。引数の数は、0 個から 90 個まで可能であり、式として指定します。式を構成できるいくつかのコンポーネントの例を以下に示します。

- 修飾または非修飾の列名
- 定数
- ホスト変数
- 特殊レジスター
- パラメーター・マーカ

引数の位置は重要で、それをセマンティクスする関数の定義に正確に従ってなければなりません。引数の位置と、関数の定義の両方が関数本体に従ってなければなりません。DB2 は、引数が関数の定義とうまく一致するように、引数を入れ替えたりはしません。また、DB2 はそれぞれの関数パラメーターのセマンティクスを認識しません。

UDF 引数式で列名を使用するには、その列の入った表参照が適切な有効範囲を持っている必要があります。結合で参照される表関数が、別の表や表関数にある列に関連した引数を使用している場合、FROM 節内の参照をもった表参照の前にその表または表関数がなければなりません。

関数内でパラメーター・マーカを使用するには、単に次のようにコーディングするだけでは済みません。

```
BLOOP(?)
```

引数がどのデータ・タイプになるかは、関数選択のロジックには分からないので、参照を解決できないからです。CAST 指定を使用すれば、パラメーター・マーカ

にタイプを指定することができます。例えば次のように INTEGER と指定すると、関数選択ロジックは処理を進めることができます。

```
BLOOP(CAST(? AS INTEGER))
```

以下に、関数呼び出しの正しい例を示します。

```
AVG(FLOAT_COLUMN)
BLOOP(COLUMN1)
BLOOP(FLOAT_COLUMN + CAST(? AS INTEGER))
BLOOP(:hostvar :indicvar)
BRIAN.PARSE(CHAR_COLUMN CONCAT USER, 1, 0, 0, 1)
CTR()
FLOOR(FLOAT_COLUMN)
PABLO.BLOOP(A+B)
PABLO.BLOOP(:hostvar)
"search_schema"(CURRENT FUNCTION PATH, 'GENE')
SUBSTR(COLUMN2,8,3)
SYSFUN.FLOOR(AVG(EMP.SALARY))
SYSFUN.AVG(SYSFUN.FLOOR(EMP.SALARY))
SYSIBM.SUBSTR(COLUMN2,11,LENGTH(COLUMN3))
SQRT((SELECT SUM(length*length)
      FROM triangles
      WHERE id= 'J522'
      AND legtype <> 'HYP'))
```

上記の関数のいずれかが表関数である場合、それらの関数を参照する構文は、前述の関数とはわずかに異なっています。例えば PABLO.BLOOP が表関数であれば、次のようにしてこの関数を正しく参照します。

```
TABLE(PABLO.BLOOP(A+B)) AS Q
```

関数選択

修飾および非修飾のどちらの関数の参照の場合でも、関数選択のアルゴリズムは、以下を保有する組み込みおよびユーザー定義の両方の該当する 関数を探索します。

- 所定の名前
- 関数の参照の引数と同じ数の定義済みパラメーター
- 対応する引数のタイプに一致する、またはその引数からプロモートできる各パラメーター

該当する関数とは、修飾参照の場合は名前の付いたスキーマ内の関数、非修飾参照の場合は SQL パスのスキーマ内の関数のことです。アルゴリズムでは正確に一致するものが検索されますが、一致するものが見つからなかった場合は、これらの関数のうちで最適なものが検索されます。非修飾参照の場合のみ、異なるスキーマでまったく同じものが 2 つ検出されると、判別要素として SQL パスが使用されません。

例外: RID という名前の関数への非修飾参照があり、副選択の FROM 節内の表参照と一致する 1 つの引数によってその関数が呼び出される場合、スキーマは SYSIBM となり、組み込み RID 関数が呼び出されます。

関数の参照をネストすることができます。それは、同一の関数の参照でもかまいません。このことは通常、UDF の他に、組み込み関数についても言えますが、列関数がかかわる場合にはいくつかの制限があります。

以下に例を示します。

```
CREATE FUNCTION BLOOP (INTEGER) RETURNS INTEGER ...
CREATE FUNCTION BLOOP (DOUBLE) RETURNS INTEGER ...
```

ここで次の DML ステートメントについて考えます。

```
SELECT BLOOP( BLOOP(COLUMN1)) FROM T
```

COLUMN1 が DECIMAL または DOUBLE 列である場合は、内部の BLOOP 参照は、上で定義されている 2 番目の BLOOP に変換されます。この BLOOP は INTEGER を戻すので、外部の BLOOP は最初の BLOOP に変換されます。

また、COLUMN1 が SMALLINT または INTEGER 列である場合は、内部の BLOOP 参照は、上で定義されている最初の BLOOP に変換されます。この BLOOP は INTEGER を戻すので、外部の BLOOP も最初の BLOOP に変換されます。この場合、同じ関数に対してネストされた参照を見ていることになります。

SQL 演算子名のうちいずれか 1 つを使用して関数を定義すると、インフィックス表記を使用して実際に UDF を呼び出すことができる。例えば、BOAT という特殊タイプを持つ値に対して "+" 演算子に何らかの意味を持たせることができるでしょう。その場合、次の UDF を定義できます。

```
CREATE FUNCTION "+" (BOAT, BOAT) RETURNS ...
```

さらに、次の有効な SQL ステートメントを作成できます。

```
SELECT BOAT_COL1 + BOAT_COL2
FROM BIG_BOATS
WHERE BOAT_OWNER = 'Nelson Mattos'
```

一方で、同様に有効な以下のステートメントも作成できます。

```
SELECT "+"(BOAT_COL1, BOAT_COL2)
FROM BIG_BOATS
WHERE BOAT_OWNER = 'Nelson Mattos'
```

このようにして、>、=、LIKE、IN などの組み込み条件演算子を多重定義することは許可されていません。

UDF またはメソッドのパラメーターとしての特殊タイプ

パラメーターまたは結果として特殊タイプを使用して UDF およびメソッドを定義することができます。DB2 は、特殊タイプのソース・データ・タイプのフォーマットの値を UDF またはメソッドに渡します。

ホスト変数から発生し、しかも特殊タイプと定義された対応したパラメーターを持つ UDF に対する引数として使用される特殊タイプの値は、**ユーザーによって特殊タイプに明示的にキャストされなければなりません**。特殊タイプ用のホスト言語タイプはありません。DB2 の厳密な型判定ではそれが義務付けられています。そうしないと、結果があいまいになるからです。BLOB を介して定義される BOAT 特殊タイプについてと、および次のように定義された BOAT_COST UDF について考えてみます。

```
CREATE FUNCTION BOAT_COST (BOAT)
RETURNS INTEGER
...
```

以下の C 言語アプリケーションの一部では、`:ship` というホスト変数により、`BOAT_COST` 関数に渡されるべき値 `BLOB` が保留されます。

```
EXEC SQL BEGIN DECLARE SECTION;
SQL TYPE IS BLOB(150K) ship;
EXEC SQL END DECLARE SECTION;
```

以下のステートメントはどちらも、タイプ `BOAT` に `:ship` ホスト変数をキャストするので、`BOAT_COST` 関数に正しく変換されます。

```
... SELECT BOAT_COST (BOAT(:ship)) FROM ...
... SELECT BOAT_COST (CAST(:ship AS BOAT)) FROM ...
```

データベース中に複数の `BOAT` 特殊タイプがあるか、あるいは、別のスキーマに `BOAT UDF` がある場合は、SQL パスに注意しなければなりません。そうしないと、結果があいまいになることがあります。

UDF パラメーターとしての LOB 値

UDF は、`BLOB`、`CLOB`、または `DBCLOB` などの `LOB` タイプのパラメーターまたは結果を使用して定義できます。DB2 は、`LOB` 値のソースが `LOB` ロケーターのホスト変数である場合にも、そのような関数を呼び出す前にストレージ中のすべての `LOB` 値をマテリアライズします。例えば、以下のような C 言語アプリケーションの一部を例に考えてみます。

```
EXEC SQL BEGIN DECLARE SECTION;
SQL TYPE IS CLOB(150K) clob150K ; /* LOB host var */
SQL TYPE IS CLOB_LOCATOR clob_locator1; /* LOB locator host var */
char string[40]; /* string host var */
EXEC SQL END DECLARE SECTION;
```

対応するパラメーターが `CLOB(500K)` として定義される関数の引数として有効なのは、`:clob150K` または `:clob_locator1` のいずれかのホスト変数です。例えば、UDF を次のように登録するとします。

```
CREATE FUNCTION FINDSTRING (CLOB(500K, VARCHAR(200)))
...
```

以下の `FINDSTRING` の呼び出しは、プログラムではどちらも有効です。

```
... SELECT FINDSTRING (:clob150K, :string) FROM ...
... SELECT FINDSTRING (:clob_locator1, :string) FROM ...
```

`LOB` タイプのいずれかをとる、UDF のパラメーターや結果は、`AS LOCATOR` 修飾子によって作成できます。この場合、呼び出し前に `LOB` 値全体がマテリアライズされることはありません。代わりに、`LOB LOCATOR` が UDF に渡されると、UDF は SQL を使用して、`LOB` 値の実際のバイトを操作することができます。この機能は、`LOB` に基づく特殊タイプを持つ UDF のパラメーターや結果にも使用できます。この関数の引数には、定義されたタイプの `LOB` 値を取ることができます。引数が `LOCATOR` タイプの 1 つとして定義されたホスト変数である必要はありません。UDF のパラメーターおよび結果の定義で `AS LOCATOR` を使用する場合は、普通はホスト変数ロケーターを引数として使用します。

スカラー関数またはメソッドの呼び出し

組み込みスカラー関数、ユーザー定義スカラー関数、メソッドについては、呼び出しの方法がよく似ています。スカラー関数とメソッドは、SQL ステートメント内で式がサポートされている場所でのみ呼び出すことができます。

前提条件

- 組み込み関数の場合は、CURRENT PATH 特殊レジスターに SYSIBM が入っている必要があります。SYSIBM はデフォルトで CURRENT PATH に入っています。
- ユーザー定義スカラー関数の場合は、CREATE FUNCTION か CREATE METHOD のいずれかのステートメントを使用して、データベース内に関数を作成しておく必要があります。
- 外部ユーザー定義スカラー関数の場合は、CREATE FUNCTION か CREATE METHOD のいずれかのステートメントの EXTERNAL 節で指定するロケーションに、関数に関連したライブラリー・ファイルまたはクラス・ファイルを配置する必要があります。
- ユーザー定義関数またはメソッドを呼び出すユーザーには、その関数またはメソッドに関する EXECUTE 特権が必要です。その関数またはメソッドをすべてのユーザーが使用する場合は、その関数またはメソッドに関する EXECUTE 特権を PUBLIC に付与します。特権の詳細については、各 CREATE ステートメントのリファレンスを参照してください。

スカラー UDF またはメソッドを呼び出すには、次のようにします。

SQL ステートメントに含まれる式の中にスカラー・ユーザー定義関数またはメソッドへの参照を組み込みます (その式の中で関数またはメソッドが 1 つ以上の入力値を処理します)。式が有効な場所であれば、どこからでも関数やメソッドを呼び出せます。例えば、照会の選択リストや VALUES 節の中で、スカラー UDF やメソッドの参照を記述できます。

例えば、EMPLOYEE 表の各社員行の基本給とボーナスを合算する TOTAL_SAL というユーザー定義スカラー関数を作成したとします。

```
CREATE FUNCTION TOTAL_SAL
(SALARY DECIMAL(9,2), BONUS DECIMAL(9,2))
RETURNS DECIMAL(9,2)
LANGUAGE SQL
CONTAINS SQL
NO EXTERNAL ACTION
DETERMINISTIC
RETURN SALARY+BONUS
```

以下は、TOTAL_SAL を活用した SELECT ステートメントです。

```
SELECT LASTNAME, TOTAL_SAL(SALARY, BONUS) AS TOTAL
FROM EMPLOYEE
```

ユーザー定義の表関数の呼び出し

ユーザー定義の表関数を作成してデータベースに登録し終わったら、SELECT ステートメントの FROM 節を使用して呼び出すことができます。

始める前に

- CREATE FUNCTION を実行して、データベース内に表関数を作成しておく必要があります。
- 外部ユーザー定義表関数の場合は、CREATE FUNCTION の EXTERNAL 節で指定するロケーションに、関数に関連したライブラリー・ファイルまたはクラス・ファイルを配置する必要があります。
- ユーザー定義表関数を呼び出すユーザーには、その関数に関する EXECUTE 特権が必要です。特権の詳細については、CREATE FUNCTION のリファレンスを参照してください。

手順

ユーザー定義表関数を呼び出すには、SQL ステートメントの FROM 節でその関数を参照します (SQL ステートメントは、その場所で一連の入力値を処理します)。表関数の参照の前に TABLE 節を指定する必要があり、参照は大括弧で囲む必要があります。

例えば以下の CREATE FUNCTION ステートメントは、指定した部門番号の社員名を戻す表関数を定義しています。

```
CREATE FUNCTION DEPTEMPLOYEES (DEPTNO VARCHAR(3))
  RETURNS TABLE (EMPNO CHAR(6),
                 LASTNAME VARCHAR(15),
                 FIRSTNAME VARCHAR(12))
  LANGUAGE SQL
  READS SQL DATA
  NO EXTERNAL ACTION
  DETERMINISTIC
  RETURN
  SELECT EMPNO, LASTNAME, FIRSTNAME FROM EMPLOYEE
  WHERE EMPLOYEE.WORKDEPT = DEPTEMPLOYEES.DEPTNO
```

以下は、DEPTEMPLOYEES を活用した SELECT ステートメントです。

```
SELECT EMPNO, LASTNAME, FIRSTNAME FROM TABLE(DEPTEMPLOYEES('A00')) AS D
```

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 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® 資料などのその他の DB2 技術情報には、オンライン (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 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 にある IBM Publications Center にアクセスしてください。英語および翻訳された DB2 バージョン 9.7 のマニュアル (PDF 形式) は、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。

資料番号は、資料が更新される度に大きくなります。資料を参照する際は、以下にリストされている最新版であることを確認してください。

注: DB2 インフォメーション・センターは、PDF やハードコピー資料よりも頻繁に更新されます。

表 50. DB2 の技術情報

資料名	資料番号	印刷資料が入手可能かどうか	最終更新
管理 API リファレンス	SC88-5883-01	入手可能	2009 年 11 月
管理ルーチンおよびビュー	SC88-5880-01	入手不可	2009 年 11 月
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	SC88-5885-01	入手可能	2009 年 11 月
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	SC88-5886-01	入手可能	2009 年 11 月
コマンド・リファレン ス	SC88-5884-01	入手可能	2009 年 11 月
データ移動ユーティリ ティー ガイドおよびリ ファレンス	SC88-5903-00	入手可能	2009 年 8 月
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	SC88-5904-01	入手可能	2009 年 11 月
データベース: 管理の 概念および構成リファ レンス	SC88-5870-01	入手可能	2009 年 11 月
データベースのモニタ リング ガイドおよびリ ファレンス	SC88-5872-01	入手可能	2009 年 11 月
データベース・セキュ リティ・ガイド	SC88-5905-01	入手可能	2009 年 11 月

表 50. DB2 の技術情報 (続き)

資料名	資料番号	印刷資料が入手可能 かどうか	最終更新
DB2 Text Search ガイド	SC88-5902-01	入手可能	2009 年 11 月
ADO.NET および OLE DB アプリケーションの開発	SC88-5874-01	入手可能	2009 年 11 月
組み込み SQL アプリケーションの開発	SC88-5875-01	入手可能	2009 年 11 月
Java アプリケーションの開発	SC88-5878-01	入手可能	2009 年 11 月
Perl、PHP、Python および Ruby on Rails アプリケーションの開発	SC88-5879-00	入手不可	2009 年 8 月
SQL および外部ルーチンの開発	SC88-5876-01	入手可能	2009 年 11 月
データベース・アプリケーション開発の基礎	GI88-4201-01	入手可能	2009 年 11 月
DB2 インストールおよび管理 概説 (Linux および Windows 版)	GI88-4202-00	入手可能	2009 年 8 月
グローバリゼーション・ガイド	SC88-5906-00	入手可能	2009 年 8 月
DB2 サーバー機能 インストール	GC88-5888-01	入手可能	2009 年 11 月
IBM データ・サーバー・クライアント機能 インストール	GC88-5889-00	入手不可	2009 年 8 月
メッセージ・リファレンス 第 1 巻	SC88-5897-00	入手不可	2009 年 8 月
メッセージ・リファレンス 第 2 巻	SC88-5898-00	入手不可	2009 年 8 月
Net Search Extender 管理およびユーザズ・ガイド	SC88-5901-01	入手不可	2009 年 11 月
パーティションおよびクラスタリングのガイド	SC88-5907-01	入手可能	2009 年 11 月
pureXML ガイド	SC88-5895-01	入手可能	2009 年 11 月
Query Patroller 管理およびユーザズ・ガイド	SC88-5908-00	入手不可	2009 年 8 月

表 50. DB2 の技術情報 (続き)

資料名	資料番号	印刷資料が入手可能 かどうか	最終更新
<i>Spatial Extender</i> および <i>Geodetic Data</i> <i>Management Feature</i> ユ ーザーズ・ガイドおよ びリファレンス	SC88-5900-00	入手不可	2009 年 8 月
<i>SQL</i> プロシージャ言 語: アプリケーション のイネーブルメントお よびサポート	SC88-5877-01	入手可能	2009 年 11 月
<i>SQL</i> リファレンス 第 1 巻	SC88-5881-01	入手可能	2009 年 11 月
<i>SQL</i> リファレンス 第 2 巻	SC88-5882-01	入手可能	2009 年 11 月
問題判別およびデータ ベース・パフォーマンス のチューニング	SC88-5871-01	入手可能	2009 年 11 月
<i>DB2</i> バージョン 9.7 へ のアップグレード	SC88-5887-01	入手可能	2009 年 11 月
<i>Visual Explain</i> チュー リアル	SC88-5899-00	入手不可	2009 年 8 月
<i>DB2</i> バージョン 9.7 の 新機能	SC88-5893-01	入手可能	2009 年 11 月
ワークロード・マネー ジャー ガイドおよびリ ファレンス	SC88-5894-01	入手可能	2009 年 11 月
<i>XQuery</i> リファレンス	SC88-5896-01	入手不可	2009 年 11 月

表 51. DB2 Connect 固有の技術情報

資料名	資料番号	印刷資料が入手可能 かどうか	最終更新
<i>DB2 Connect Personal</i> <i>Edition</i> インストールお よび構成	SC88-5891-01	入手可能	2009 年 11 月
<i>DB2 Connect</i> サーバー 機能 インストールおよ び構成	SC88-5892-01	入手可能	2009 年 11 月
<i>DB2 Connect</i> ユーザー ズ・ガイド	SC88-5890-01	入手可能	2009 年 11 月

表 52. Information Integration の技術情報

資料名	資料番号	印刷資料が入手可能かどうか	最終更新
Information Integration: フェデレーテッド・システム管理ガイド	SC88-4166-02	入手可能	2009 年 8 月
Information Integration: レプリケーションおよびイベント・パブリッシングのための ASNCLP プログラム・リファレンス	SC88-4167-04	入手可能	2009 年 8 月
Information Integration: フェデレーテッド・データ・ソース構成ガイド	SC88-4185-02	入手不可	2009 年 8 月
Information Integration: SQL レプリケーションガイドとリファレンス	SC88-4168-02	入手可能	2009 年 8 月
Information Integration: レプリケーションとイベント・パブリッシング 概説	GC88-4187-02	入手可能	2009 年 8 月

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/v9r7>) 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> をご覧ください。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)
 - 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、368 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 製品は、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 状態ヘルプを開始するには、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 9.7 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/v9r7/> です。

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 9.1 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、DB2 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・センターでの希望する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. ページを最新表示します。希望する言語で DB2 インフォメーション・センターが表示されます。
- Firefox または Mozilla Web ブラウザーの場合に、希望する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. ページを最新表示します。希望する言語で DB2 インフォメーション・センターが表示されます。

ブラウザー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りません。

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

ローカルにインストールされた DB2 インフォメーション・センターは、定期的に更新する必要があります。

始める前に

DB2 バージョン 9.7 インフォメーション・センターが既にインストールされている必要があります。詳しくは、「DB2 サーバー機能 インストール」の『DB2 セットアップ・ウィザードによる DB2 インフォメーション・センターのインストール』

のトピックを参照してください。インフォメーション・センターのインストールに適用されるすべての前提条件と制約事項は、インフォメーション・センターの更新にも適用されます。

このタスクについて

既存の DB2 インフォメーション・センターは、自動で更新することも。手動で更新することもできます。

- 自動更新 - 既存のインフォメーション・センターのフィーチャーと言語を更新します。自動更新を使用すると、更新中にインフォメーション・センターが使用できなくなる時間が最小限で済むというメリットもあります。さらに、自動更新は、定期的に行う他のバッチ・ジョブの一部として実行されるように設定することができます。
- 手動更新 - 更新処理中にフィーチャーまたは言語を追加する場合に使用する必要があります。例えば、ローカルのインフォメーション・センターが最初は英語とフランス語でインストールされており、その後ドイツ語もインストールすることにした場合、手動更新でドイツ語をインストールし、同時に、既存のインフォメーション・センターのフィーチャーおよび言語を更新できます。しかし、手動更新ではインフォメーション・センターを手動で停止、更新、再始動する必要があります。更新処理の間はずっと、インフォメーション・センターは使用できなくなります。

手順

このトピックでは、自動更新のプロセスを詳しく説明しています。手動更新の手順については、『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・センターの手動更新』のトピックを参照してください。

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ている DB2 インフォメーション・センターを自動で更新するには、次のようにします。

1. Linux オペレーティング・システムの場合、次のようにします。
 - 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/V9.7` 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・ディレクトリーから `doc/bin` ディレクトリーにナビゲートします。
 - c. 次のように `ic-update` スクリプトを実行します。

```
ic-update
```
2. Windows オペレーティング・システムの場合、次のようにします。
 - 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 9.7` 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・ディレクトリーから doc¥bin ディレクトリーにナビゲートします。
- d. 次のように ic-update.bat ファイルを実行します。

```
ic-update.bat
```

結果

DB2 インフォメーション・センターが自動的に再始動します。更新が入手可能な場合、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。インフォメーション・センターの更新が入手可能でなかった場合、メッセージがログに追加されます。ログ・ファイルは、doc¥eclipse¥configuration ディレクトリーにあります。ログ・ファイル名はランダムに生成された名前です。例えば、1239053440785.log のようになります。

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・センターの手動更新

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 から資料の更新を入手してインストールすることができます。

このタスクについて

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 を手動で更新するには、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、更新を適用できるようになります。DB2 インフォメーション・センターのワークステーション・バージョンは、常にスタンドアロン・モードで実行されます。を参照してください。
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。インストールしなければならない更新がある場合は、「更新」機能を使用してそれを入手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、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してください。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入手します。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 2008、Windows Vista (およびそれ以上) では、このセクションの後の部分でリストされている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・ツールを開くには、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。

手順

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/db2icdv97 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 9.7` 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・ディレクトリーから `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/V9.7` 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・ディレクトリーから `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 ブラウザーが開き、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。(ブラウザーで JavaScript™ が有効になっている必要があります。) **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. インストール・プロセスを開始するには、インストールする更新をチェックして選択し、「**更新のインストール**」をクリックします。
5. 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. 次のようにして、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停止するのに必要なコマンドが含まれています。`help_start.bat` は、Ctrl-C や他の方法を使用して停止しないでください。

- 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停止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` スクリプトを停止しないでください。

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/db2icdv97 start
```

結果

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 データベース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『データベースの基本』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 データベース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/support/db2_9/) 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。IBM 以外の製品に関する情報は、本書の最初の発行時点で入手可能な情報に基づいており、変更される場合があります。

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。

〒242-8502
神奈川県大和市下鶴間1623番14号
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。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

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。

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せんが、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ます、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。サンプル・プログラムは、現存するままの状態を提供されるものであり、いかなる種類の保証も提供されません。IBM は、これらのサンプル・プログラムの使用から生ずるいかなる損害に対しても責任を負いません。

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

IBM、IBM ロゴおよび [ibm.com](http://www.ibm.com)[®] は、世界の多くの国で登録された International Business Machines Corp. の商標です。他の製品名およびサービス名等は、それぞれ IBM または各社の商標である場合があります。現時点での IBM の商標リストについては、<http://www.ibm.com/legal/copytrade.shtml> をご覧ください。

以下は、それぞれ各社の商標または登録商標です。

- Linux は、Linus Torvalds 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。
- Intel、Intel ロゴ、Intel Inside[®]、Intel Inside ロゴ、Intel[®] Centrino[®]、Intel Centrino ロゴ、Celeron[®]、Intel[®] Xeon[®]、Intel SpeedStep[®]、Itanium、Pentium[®] は、Intel Corporation または子会社の米国およびその他の国における商標または登録商標です。
- Microsoft、Windows、Windows NT[®]、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。

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]

アプリケーション開発

ルーチン 3

移植性

ルーチン 47

インフィックス表記 361

エラー

.NET CLR ルーチン 139

エラー・メッセージ

SQL プロシージャ 112

オブジェクト・インスタンス

OLE オートメーション・ルーチン 326

[カ行]

カーソル

結果セットの取得 359

ルーチン 79

外部プロシージャ

COBOL 267

外部ルーチン

インプリメンテーション 24

概要 67

クラス・ファイル

セキュリティ 114

デプロイ 113

バックアップ 116

変更 115

リストア 116

作成 107

パフォーマンス 116

パラメーター・スタイル 82

フィーチャー 67

プログラミング言語 24, 25, 88

命名の競合 114

ライブラリー

管理 116

セキュリティ 114

デプロイ 113

バックアップ 116

パフォーマンス 116

変更 115

リストア 116

例 319

32 ビットのサポート 102

外部ルーチン (続き)

64 ビットのサポート 102

API 24, 25, 88

外部ルーチン GENERAL WITH NULLS のパラメーター・スタイル 82

外部ルーチンの DB2SQL パラメーター・スタイル 82

外部ルーチンの GENERAL パラメーター・スタイル 82

監査

トランザクション 35

関数

外部

概要 68

概要 13

参照 360

集約

概要 13

スカラー

概要 13

詳細 15

選択 361

パラメーター 194

表

概要 13

詳細 16

呼び出し 360

ルーチンの概要 34

ルーチンの他の機能タイプとの比較 18

Java 292

row 13, 16

行関数

詳細 16

行セット

OLE DB

完全修飾名 333

共通言語ランタイム (CLR)

関数

例 154, 178

プロシージャ

結果セットの戻り 126

例 143, 159

ルーチン

エラー 139

開発 120

概要 119

構築 132, 133, 135

コンパイラー・オプション 137

作成 129, 130

制約事項 128

セキュリティ 127

設計 120

パラメーター 123

- 共通言語ランタイム (CLR) (続き)
 - ルーチン (続き)
 - リンク・オプション 137
 - 例 142
 - Dbinfo 構造の使用法 123
 - scratchpad 123
 - SQL データ・タイプ 121
 - XML サポート 170
 - XQuery サポート 170
 - .NET 138
- 共用ライブラリー
 - ルーチンの再構築 262
- 許可
 - 外部ルーチン 59, 338
- 組み込みルーチン
 - 概要 23
- グラフィック・データ
 - ホスト変数
 - C/C++ ルーチン 229
- 結果セット
 - 受け取り
 - JDBC アプリケーションおよびルーチン 304
 - SQLJ アプリケーションおよびルーチン 306
 - ストアード・プロシージャ 355
 - 戻り
 - JDBC ストアード・プロシージャ 303
 - SQLJ ストアード・プロシージャ 303
 - .NET CLR プロシージャ 126
- 結果セットの戻り
 - JDBC ストアード・プロシージャ 303
 - SQLJ ストアード・プロシージャ 303
- コード・ページ
 - 変換
 - ルーチン 99
- コール・レベル・インターフェース (CLI)
 - ストアード・プロシージャ
 - 呼び出し 353
- 更新
 - DB2 インフォメーション・センター 373, 375
- 構成パラメーター
 - javaheapsz 316
 - jdk11path 316
- 構造化タイプ
 - 属性
 - 使用メソッドへのアクセス 17
 - メソッド
 - 概要 17
- コマンド行プロセッサ (CLP)
 - ルーチン作成 38
- ご利用条件
 - 資料 378
- コンテキスト
 - マルチスレッド DB2 アプリケーションでの設定
 - SQLJ ルーチン 289

[サ行]

- システム定義ルーチン
 - 概要 7
- 資料
 - 印刷 368
 - 概要 367
 - 使用に関するご利用条件 378
 - 注文 371
 - PDF ファイル 368
- スカラー関数
 - 詳細 68
 - 処理モデル 70
 - ルーチン 15
- スクラッチパッド
 - 概要 93
 - 詳細 74
 - 32 ビット・プラットフォーム 78
 - 64 ビット・プラットフォーム 78
 - Java UDF 293
- ストアード・プロシージャ
 - 概要 12
 - 結果セットの取り出し
 - 概要 355
 - 参照 345
 - 呼び出し
 - 一般的なアプローチ 345
 - CLI アプリケーション 353
 - AIX C 構成ファイル 259
 - CALL ステートメント 351
 - COBOL 265
- セーブポイント
 - プロシージャ 79
- セキュリティ
 - ルーチン 56, 58
- ソース派生ルーチン 23

[タ行]

- タイプ装飾
 - C++ ルーチン本体 230
- タイプ・マッピング
 - OLE オートメーション BASIC タイプ 328
- チュートリアル
 - トラブルシューティング 378
 - 問題判別 378
 - リスト 377
 - Visual Explain 377
- 中間結果表 357
- ツール
 - ルーチン開発 38
- データ・タイプ
 - 変換
 - COBOL 267
 - OLE オートメーション・タイプ 327
 - ARRAY 302

データ・タイプ (続き)
 COBOL 267
 Java 288
デバッグ
 ルーチン
 一般的な問題 110
 技法 110
 .NET CLR 138
特殊タイプ
 ルーチンへの引き渡し 362
特記事項 381
トラブルシューティング
 オンライン情報 378
 チュートリアル 378

[ハ行]

バイナリー・ラージ・オブジェクト (BLOB)
 ユーザー定義関数 (UDF) 205
 COBOL 267
 Java 288, 296
 OLE DB 表関数 334
バインド
 ルーチン 59, 338
バックアップ
 外部ルーチン・ライブラリー 116
パフォーマンス
 外部ルーチン 116
 ルーチン
 推奨事項 48, 93
 利点 3
パラメーター
 C/C++ ルーチン 189
パラメーター・スタイル
 概要 82
 PARAMETER STYLE DB2GENERAL 290
 PARAMETER STYLE JAVA 290
表
 関数 16
 読み取りおよび書き込みの競合 63
表関数
 概要 16
 ユーザー定義表関数 71
 Java 実行モデル 73, 307
表ユーザー定義関数 (UDF)
 処理モデル 71
浮動小数点パラメーター 205
プロシージャ
 概要 12
 関数の比較 18
 共通言語ランタイム (CLR) 例 143
 結果セット
 .NET CLR (C# の例) 143
 .NET CLR (プロシージャ) 126
 参照 345

プロシージャ (続き)
 パラメーター
 処理 84
 PARAMETER STYLE JAVA 節 291
 PARAMETER STYLE SQL 節 191
 メソッドの比較 18
 呼び出し
 アプリケーション 347
 外部ルーチン 347
 概要 345
 トリガー 348
 SQL ルーチン 348
ADMIN_CMD
 概要 34
C/C++ 結果セット 232
Java
 PARAMETER STYLE JAVA 節 291
分離レベル
 ルーチン 79
ヘルプ
 言語の構成 372
 SQL ステートメント 372
ホスト変数
 COBOL アプリケーション 267
本書の対象読者 vii

[マ行]

マルチスレッド・アプリケーション
 SQLJ ルーチン 289
メソッド
 外部 68
 概要 17
 パラメーターとしての特種タイプ 362
 ルーチンの他の機能タイプとの比較 18
 Java PARAMETER STYLE 節 292
問題判別
 チュートリアル 378
 利用できる情報 378

[ヤ行]

ユーザー定義関数 (UDF)
 インフィックス表記 361
 再入可能 74
 作成 35
 状態の保存 74
 スカラー
 FINAL CALL 70
 データの戻り 205
 パラメーターとしての LOB 値 363
 パラメーターとしての特種タイプ 362
 日付パラメーター 205
表
 概要 71

ユーザー定義関数 (UDF) (続き)

表 (続き)

処理モデル 71

呼び出し 364

FINAL CALL 71

NO FINAL CALL 71

SQL-result 引数 71

SQL-result-ind 引数 71

呼び出し 364

32 ビット・プラットフォームと 64 ビット・プラットフォームのスクラッチパッドの移植性 78

AIX C 構成ファイル 261

C# における共通言語ランタイム UDF 178

C/C++ 205

DETERMINISTIC 74

FOR BIT DATA 修飾子 205

Java 293

NOT DETERMINISTIC 74

OLE DB 表関数 331

SCRATCHPAD オプション 74

ユーザー定義ルーチン

概要 8

[ラ行]

ラージ・オブジェクト (LOB)

ルーチンへの引き渡し 363

ライブラリー

共用

ルーチンの再構築 262

リストア

外部ルーチン・ライブラリー 116

ルーチン

移植性

概要 47

32 ビット・プラットフォームと 64 ビット・プラットフォームの間 78

インターオペラビリティ 47

インプリメンテーション

概要 22

組み込み 23

ソース派生 23

比較 33

SQL 23

カーソル 79

開発ツール 38

外部

インプリメンテーション 24

概要 3, 67

共通言語ランタイム 119, 129, 130, 132, 133, 135

許可 59, 338

禁止ステートメント 104

クラス (デプロイ) 113

クラス (バックアップ) 116

クラス (変更) 115

クラス (リストア) 116

ルーチン (続き)

外部 (続き)

作成 107

サポートされている API 24, 25, 88

サポートされているプログラミング言語 24, 25, 88

制約事項 68, 104

セキュリティ 114

その他のタイプとの比較 30

パフォーマンス 116

パラメーター・スタイル 82

判別の必要 33

フィーチャー 67, 68

命名の競合 114

ライブラリー (デプロイ) 113

ライブラリー (バックアップ) 116

ライブラリー (変更) 115

ライブラリー (リストア) 116

ライブラリーおよびクラスのデプロイメント 113

ライブラリーおよびクラスのバックアップ 116

ライブラリーおよびクラスの変更 115

ライブラリーおよびクラスのリストア 116

ライブラリー管理 116

32 ビットのサポート 102

64 ビットのサポート 102

C/C++ 185, 187, 236, 243

Java 312

Java ルーチンの更新 319

SQL ステートメント・サポート 39, 79

XML データ・タイプのサポート 103

概要 2, 3, 4

書き込みの競合 63

関数

概要 13

使用するタイプの決定 21

スカラー 15

表 16

row 16

関数パス 342

機能タイプ 11

共通言語ランタイム

エラー 139

開発サポート 120

開発ツール 120

結果セットの戻り 126

構築 132, 133, 135

作成 129, 130

サポートされる SQL データ・タイプ 121

詳細 119

スクラッチパッドの使用法 123

制約事項 128

セキュリティ 127

設計 120

例 142

CLR 関数 (UDF) の例 178

C# の CLR プロシーチャーの例 143

EXECUTION CONTROL 節 127

ルーチン (続き)

共通言語ランタイム (続き)

- Visual Basic .NET CLR 関数の例 154
- Visual Basic .NET CLR プロシージャの例 159
- XML データ・タイプのサポート 103

共用ライブラリーの再構築 262

禁止ステートメント 104

組み込み 23, 30

クラス 113

グラフィック・ホスト変数 229

コード・ページ変換 99

再帰的な 344

作成 109

外部 107

セキュリティ 56

プロシージャ 1

Data Studio 38

サポートされているプログラミング言語 86

システム定義 22

いつ使用するか 10

概要 4, 7

詳細 7

データベース管理 34

ユーザー定義との比較 9

使用 34, 35

スカラー UDF 68

スクラッチパッド構造 78

制約事項 104

セキュリティ 56, 58, 96

ソース派生 30, 33

タイプ

概要 4

機能 11

機能タイプの比較 18

サポートされる SQL ステートメント 39

使用する機能タイプの決定 21

ルーチンのインプリメンテーションの比較 33

多重定義 342

データベース管理 34

デバッグ 110

特殊タイプの引き渡し 362

名前 342

ネストされた 344

パフォーマンス 48, 93

比較

機能タイプ 18

システム定義およびユーザー定義 9, 10

引数の引き渡し 215

プロシージャ

いつ使用するか 21

作成 109

詳細 12

分離レベル 79

変更 113

メソッド

いつ使用するか 21

ルーチン (続き)

メソッド (続き)

作成 109

詳細 17

ユーザー定義 22

いつ使用するか 10

概要 4, 7, 35

作成 1, 8, 109

システム定義との比較 9

事前インストール 7

詳細 8

使用するインプリメンテーションの判別 33

呼び出し

関数 337

セキュリティ 56

前提条件 337

プロシージャ 337

別のルーチンから 47

メソッド 337

64 ビット・データベース・サーバーでの 32 ビット・ルーチン 344

読み取りの競合 63

ライブラリー 113

利点 3

CLR

エラー 139

COBOL

XML データ・タイプのサポート 103

C/C++

開発サポート 186

開発ツール 186

関数パラメーターとしてのスクラッチパッド 199

組み込みファイル 187

グラフィック・ホスト変数 229

結果セット 197, 232

構築 236, 243

作成 233

サポートされる SQL データ・タイプ 202

詳細 185

設計 187

パフォーマンス 103

パラメーター 189, 197

パラメーターの受け渡し 196

パラメーター・スタイル 190

64 ビット・データベース・サーバーでの 32 ビット・ルーチン 103

NULL 標識パラメーター 190

PROGRAM TYPE 節 200

sqludf_scrat 構造 199

XML データ・タイプのサポート 103

DB2GENERAL

詳細 293

COM.ibm.db2.app.Blob 301

COM.ibm.db2.app.Clob 301

COM.ibm.db2.app.Lob 300

Java クラス 297

ルーチン (続き)

EXECUTE 特権 59, 338

Java

概要 283

作成 309, 310

制約事項 307

設計 287

JAR ファイル 318

JDBC 284, 312

SQLJ 284

XML データ・タイプのサポート 103

LOB の引き渡し 363

NOT FENCED

セキュリティ 56, 58, 96

パフォーマンス 93

OLE オートメーション

定義 325

SQL

概要 23

その他のタイプとの比較 30, 33

SQL ステートメント・サポート 39, 46

THREADSAFE

セキュリティ 96

パフォーマンス 93

WCHARTYPE プリコンパイラ・オプション 229

ルーチン名の多重定義 342

列

データ・タイプ

作成 (COBOL) 267

ログ

SQL プロシージャからのメッセージ 112

[数字]

32 ビット外部ルーチン

概要 102

32 ビット・アプリケーション

概要 100

32 ビット・ルーチン

概要 100

64 ビット外部ルーチン

概要 102

64 ビット・アプリケーション

概要 100

64 ビット・ルーチン

概要 100

A

ADMIN_CMD プロシージャ

概要 34

AIX

C スタード・プロシージャ

構成ファイルでの構築 259

AIX (続き)

C ユーザー定義関数

構成ファイルでの構築 261

C ルーチン

コンパイラおよびリンクのオプション 245

C++ ルーチン

コンパイラおよびリンクのオプション 246

IBM COBOL ルーチン

構築 276

コンパイラおよびリンクのオプション 271

Micro Focus COBOL ルーチン

コンパイラおよびリンクのオプション 272

B

BASIC

言語 325

データ・タイプ 328

BigDecimal データ・タイプ 288

BIGINT データ・タイプ

ユーザー定義関数 (UDF) 205

COBOL 267

Java 288, 296

OLE DB 表関数 334

BLOB データ・タイプ

ユーザー定義関数 (UDF) 205

COBOL 267

Java 288, 296

OLE DB 表関数 334

BLOB-FILE COBOL タイプ 267

BLOB-LOCATOR COBOL タイプ 267

C

C 言語

関数 194

パラメーター処理、スタード・プロシージャでの
プロシージャ

結果セット 232

パラメーター・スタイル 191

XML 例 174

XQuery 例 174

ルーチン

開発サポート 186

開発ツール 186

概要 185

関数パラメーターとしてのスクラッチパッド 199

組み込みファイル 187

結果セット 197

構築 236, 243

構築 (UNIX) 238

構築 (Windows) 240

コンパイラ・オプション (AIX) 245

コンパイラ・オプション (HP-UX) 247

コンパイラ・オプション (Linux) 251

C 言語 (続き)

ルーチン (続き)

- コンパイラー・オプション (Solaris) 255
- コンパイラー・オプション (Windows) 258
- 作成 233
- 設計 187
- パフォーマンス 93
- パラメーター 189
- パラメーターとしての dbinfo 構造 197
- パラメーターの受け渡し 196
- パラメーター・スタイル 190
- 引数の引き渡し 215
- 64 ビット・データベース・サーバーでの 32 ビット・ルーチン 103
- NULL 標識パラメーター 190
- PROGRAM TYPE 節 200
- SQL データ・タイプ 202

CALL ステートメント

- アプリケーション 347
- 外部ルーチン 347
- 概要 345
- トリガー 348
- CLP 351
- SQL ルーチン 348

CHAR FOR BIT DATA データ・タイプ 296

CHAR データ・タイプ

- ユーザー定義関数 (UDF) 205
- COBOL 267
- Java 288, 296
- OLE DB 表関数 334

CLASSPATH 環境変数 316

CLOB データ・タイプ

- ユーザー定義関数 (UDF) 205
- COBOL 267
- Java 288, 296
- OLE DB 表関数 334

CLOB-FILE COBOL タイプ 267

CLOB-LOCATOR COBOL タイプ 267

COBOL 言語

- 外部プロシージャー用の開発ソフトウェア 267
- ストアド・プロシージャー 265
- データ・タイプ
 - COBOL 組み込み SQL アプリケーションでサポートされている SQL データ・タイプ 267

IBM COBOL ルーチン

- 構築 (AIX) 276
- 構築 (Windows) 278
- コンパイラー・オプション (AIX) 271
- コンパイラー・オプション (Windows) 275

Micro Focus ルーチン

- 構築 (UNIX) 277
- 構築 (Windows) 280
- コンパイラー・オプション (AIX) 272
- コンパイラー・オプション (HP-UX) 272
- コンパイラー・オプション (Linux) 274
- コンパイラー・オプション (Solaris) 273

COBOL 言語 (続き)

Micro Focus ルーチン (続き)

- コンパイラー・オプション (Windows) 276

COBOL タイプの PICTURE (PIC) 節 267

COBOL タイプの USAGE 節 267

COMP-1 データ・タイプ 267

COMP-3 データ・タイプ 267

COMP-5 データ・タイプ 267

COM.ibm.db2.app.Blob クラス

- 概要 301
- データ・タイプ 296

COM.ibm.db2.app.Clob クラス

- 概要 301
- データ・タイプ 296

COM.ibm.db2.app.Lob クラス

- 概要 300

COM.ibm.db2.app.StoredProc クラス

- 概要 297

COM.ibm.db2.app.UDF

- DB2GENERAL UDF 293

COM.ibm.db2.app.UDF クラス

- 概要 298

CONTAINS SQL 節 79

CREATE FUNCTION ステートメント

- CAST FROM 節 205
- LANGUAGE OLE 節 325
- OLE オートメーション・ルーチン 325
- PARAMETER STYLE 節 194, 292
- RETURNS 節 205

CREATE METHOD ステートメント

- PARAMETER STYLE 節 292

CREATE PROCEDURE ステートメント

- PARAMETER STYLE 節 191, 291
- PROGRAM TYPE 節 200

CREATE ROUTINE ステートメント

- PARAMETER STYLE 節 190

C# .NET

- XML 例 170

C/C++ 言語

- 関数 194
- データ・タイプ
 - OLE オートメーション 328
- プロシージャー
 - 結果セット 232
 - パラメーター・スタイル 191

ルーチン

- 開発サポート 186
- 開発ツール 186
- 概要 185
- 関数パラメーターとしてのスクラッチパッド 199
- 組み込みファイル 187
- 結果セット 197
- 構築 236, 243
- 構築 (Windows) 240
- コンパイラー・オプション (AIX) 246
- コンパイラー・オプション (HP-UX) 249

C/C++ 言語 (続き)

ルーチン (続き)

コンパイラー・オプション (Linux) 253

コンパイラー・オプション (Solaris) 257

コンパイラー・オプション (Windows) 258

作成 233

設計 187

パラメーター 189

パラメーターとしての dbinfo 構造 197

パラメーターの受け渡し 196

パラメーター・スタイル 190

64 ビット・データベース・サーバーでの 32 ビット・ルーチン 103

NULL 標識パラメーター 190

PROGRAM TYPE 節 200

SQL データ・タイプ 202

ルーチン本体のタイプ装飾 230

D

Data Studio

概要 38

DATE データ・タイプ

COBOL 267

Java 288, 296

OLE DB 表関数 334

DB2 インフォメーション・センター

言語 372

更新 373, 375

バージョン 372

DB2 資料の印刷方法 371

DB2GENERAL パラメーター・スタイル 82

DB2GENERAL ルーチン

概要 293

ストアド・プロシージャ 297

ユーザー定義関数 293, 298

Java クラス

概要 297

COM.ibm.db2.app.Blob 301

COM.ibm.db2.app.Clob 301

COM.ibm.db2.app.Lob 300

COM.ibm.db2.app.StoredProc 297

COM.ibm.db2.app.UDF 298

DB2_USE_DB2JCCT2_JROUTINE 変数

ドライバの指定 286

DBCLOB データ・タイプ

ユーザー定義関数 (UDF) 205

ルーチン 296

COBOL 267

Java 288

OLE DB 表関数 334

DBCLOB-FILE COBOL データ・タイプ 267

DBCLOB-LOCATOR COBOL データ・タイプ 267

DECIMAL データ・タイプ

変換

COBOL 267

DECIMAL データ・タイプ (続き)

変換 (続き)

Java 288

ユーザー定義関数 (UDF) 205

DB2GENERAL ルーチン 296

OLE DB 表関数 334

DOUBLE データ・タイプ

ユーザー定義関数 (UDF) 205

Java 288

E

EXECUTE 特権

ルーチン 59, 338

F

FLOAT データ・タイプ

ユーザー定義関数 (UDF) 205

COBOL 267

Java 288, 296

OLE DB 表関数 334

G

GRAPHIC データ・タイプ

COBOL 267

Java 288, 296

OLE DB 表関数 334

GRAPHIC パラメーター 205

H

HP-UX

コンパイラー・オプション

C ルーチン 247

C++ ルーチン 249

Micro Focus COBOL ルーチン 272

リンク・オプション

C ルーチン 247

C++ ルーチン 249

Micro Focus COBOL ルーチン 272

I

IBM Software Development Kit (SDK)

外部 Java ルーチンの開発 284

Int Java データ・タイプ 288

INTEGER データ・タイプ

ユーザー定義関数 (UDF) 205

COBOL 267

Java 288, 296

OLE DB 表関数 334

J

Java

- 関数 292
- クラス 319
- クラス・ファイル 316
- ストアド・プロシージャー
 - 概要 283
 - パラメーター処理 84
- JAR ファイル 318
- データ・タイプ 288
- 表関数の実行モデル 73, 307
- プロシージャー 291
- メソッド
 - COM.ibm.db2.app.Blob 301
 - COM.ibm.db2.app.Clob 301
 - COM.ibm.db2.app.Lob 300
 - COM.ibm.db2.app.StoredProc 297
 - COM.ibm.db2.app.UDF 293, 298
 - PARAMETER STYLE JAVA 292
- ユーザー定義関数 (UDF)
 - DB2GENERAL 293
 - JAR ファイルの CALL ステートメント 318
- ルーチン
 - 開発ソフトウェア 284
 - 開発ツール 286
 - 概要 283
 - 構築 (概要) 312
 - 構築 (JDBC) 313
 - 構築 (SQLJ) 314
 - 制約事項 307
 - 設計 287
 - ドライバ 286
 - パフォーマンス 93
 - パラメーター・スタイル 82, 290
 - PARAMETER STYLE DB2GENERAL 293
 - UNIX 285
- CLASSPATH 環境変数 316
- JAR ファイル 318
- PARAMETER STYLE DB2GENERAL 293
- PARAMETER STYLE JAVA 291, 292
- java.math.BigDecimal Java データ・タイプ 288

JDBC

- ストアド・プロシージャー 303
- ルーチン
 - 開発ツール 286
 - 構築 (概要) 312
 - 構築 (プロシージャー) 313
 - 作成 310
 - ドライバ 284
 - 例 (配列データ・タイプ) 320
 - 例 (要約) 319
 - 例 (XML および XQuery サポート) 320
 - API 284
 - ARRAY データ・タイプ 302, 320

JDBC (続き)

XML

- 例 320

- jdk_path 構成パラメーター
 - アプリケーション開発 262
- ルーチン
 - 構築 (UNIX) 285
 - 実行 (UNIX) 285

K

- keepfenced 構成パラメーター
 - 更新 262

L

Linux

C

- ルーチン 251

C++

- ルーチン 253

Micro Focus COBOL

- ルーチン 274

LONG VARCHAR FOR BIT DATA データ・タイプ

- Java 296

LONG VARCHAR データ・タイプ

- COBOL 267

- C/C++ 205

- Java 288, 296

- OLE DB 表関数 334

LONG VARGRAPHIC データ・タイプ

- COBOL 267

- Java 288, 296

- OLE DB 表関数 334

- UDF へのパラメーター 205

M

MODIFIES SQL DATA 節

- 外部ルーチン 79

- SQL ルーチンの SQL アクセス・レベル 45

N

- NO SQL 節 79

- NOT FENCED ルーチン 96

NUMERIC データ・タイプ

- COBOL 267

- Java 288, 296

- OLE DB 表関数 334

- NUMERIC パラメーター 205

O

OLE DB

- 完全修飾行セット名 333
- データ・タイプ
 - SQL データ・タイプへの変換 334
- 表ユーザー定義関数 331, 332
- ルーチン 215

OLE オートメーション

- クラス ID (CLSID) 325
- コントローラー 325
- サーバー 325
- ストリング・データ・タイプ 328
- プログラム ID (progID) 325
- メソッド 325
- ルーチン
 - オブジェクト・インスタンス 326
 - 設計 325
 - 定義 325
 - メソッドの呼び出し 326
- SCRATCHPAD オプション 326
- BSTR データ・タイプ 328
- OLECHAR データ・タイプ 328

P

PROGRAM TYPE

- PROGRAM TYPE MAIN 節 84
- PROGRAM TYPE SUB 節 84

R

READS SQL DATA 節 79

REAL SQL データ・タイプ

- 変換
 - C および C++ ルーチン 205
 - Java (DB2GENERAL) ルーチン 296
- COBOL 267
- Java 288
- OLE DB 表関数 334

S

SCRATCHPAD オプション

- 状態の保存 74
- ユーザー定義関数 (UDF) 74
- OLE オートメーション・ルーチン 326

SDK

- UNIX 285

SELECT ステートメント

- データ変更ステートメントからの選択 357

short データ・タイプ

- Java 288

SMALLINT データ・タイプ

- ユーザー定義関数 (UDF) 205

SMALLINT データ・タイプ (続き)

- ルーチン 296

COBOL 267

- Java 288

OLE DB 表関数 334

Solaris オペレーティング・システム

- C ルーチン 255

- C++ ルーチン 257

- Micro Focus COBOL ルーチン 273

SQL

- 外部ルーチン 79

- 外部ルーチンのパラメーター・スタイル 82

メソッド

- SQL ステートメント・サポート 39

ルーチン

- パフォーマンス 93

- SQL-bodied ルーチン (SQL を本体として持つルーチン)

- の SQL アクセス・レベル 45

SQL 関数

- SQL ステートメント・サポート 39

- SQL データを変更する表関数 35

SQL ステートメント

- データ変更 359

ヘルプ

- 表示 372

- ルーチンで許可 39, 46

CREATE FUNCTION

- 機能の開発 1

- OLE オートメーション・ルーチンの開発 325

CREATE METHOD 1

CREATE PROCEDURE

- プロシーチャーの開発 1

SQL データ・タイプ

- 組み込み SQL アプリケーション

- COBOL 267

- ユーザー定義関数 (UDF)

- C/C++ 205

ルーチン

- Java 288

- Java (DB2GENERAL) 296

- OLE DB データ・タイプへの変換 334

- OLE オートメーションでサポートされている 327

SQL プロシーチャー

- エラー・メッセージ 112

- CALL ステートメント 351

- SQL ステートメント・サポート 39

SQL ルーチン

- インプリメンテーション 23

sqlbchar データ・タイプ

- C/C++ ルーチン 205

SQLJ

- ストアード・プロシーチャー 303

ルーチン

- 開発ツール 286

- コンパイラー・オプション (UNIX) 315

- コンパイラー・オプション (Windows) 316

SQLJ (続き)

- ルーチン (続き)
 - 作成 309
 - 接続コンテキスト 289
 - ルーチンの構築 284, 314

SQLUDF 組み込みファイル

- C/C++ ルーチン 187

SQL-result 引数 71

SQL-result-ind 引数 71

String Java データ・タイプ 288

T

THREADSAFE ルーチン 96

TIME データ・タイプ

- COBOL 267
- Java 288, 296
- OLE DB 表関数 334

TIME パラメーター 205

TIMESTAMP データ・タイプ

- COBOL 267
- Java 288, 296
- OLE DB 表関数 334

TIMESTAMP パラメーター 205

U

UNIX

- C ルーチン 238
- Micro Focus COBOL ルーチン 277
- SQLJ ルーチン 315

V

VARCHAR FOR BIT DATA データ・タイプ

- C/C++ ユーザー定義関数 205
- Java (DB2GENERAL) ルーチン 296

VARCHAR データ・タイプ

- COBOL 267
- Java (DB2GENERAL) ルーチン 296
- Java ルーチン 288
- OLE DB 表関数 334

VARGRAPHIC データ・タイプ

- COBOL 267
- C/C++ ユーザー定義関数 205
- Java (DB2GENERAL) ルーチン 296
- Java ルーチン 288
- OLE DB 表関数 334

W

WCHARTYPE NOCONVERT プリコンパイラー・オプション 229

wchar_t データ・タイプ

- C/C++ ルーチン 205

Windows

COBOL ルーチン

- 構築 278
- コンパイラー・オプション 275
- リンク・オプション 275

C/C++ ルーチン

- 構築 240
- コンパイラー・オプション 258
- リンク・オプション 258

Micro Focus COBOL ルーチン

- 構築 280
- コンパイラー・オプション 276
- リンク・オプション 276

SQLJ ルーチン 316

X

XML データ・タイプ

- 外部ルーチン 103

[特殊文字]

.NET

共通言語ランタイム (CLR) ルーチン

- 開発ツール 120
- 外部 120
- 概要 119
- 構築 132, 133, 135
- デバッグ 138
- 例 170

ルーチン

- コンパイルとリンクのオプション 137



Printed in Japan

SC88-5876-01



日本アイ・ビー・エム株式会社
〒103-8510 東京都中央区日本橋箱崎町19-21

Spine information:

IBM DB2 9.7 for Linux, UNIX, and Windows バージョン 9 リリース 7

SQL および外部ルーチンの開発

