IBM® DB2 Universal Database™

アプリケーション開発ガイド:
サーバー・アプリケーションのプログラミング

バージョン 8.2

SC88-9139-01
(英文原典：SC09-4827-01)

IBM® DB2 Universal Database™

アプリケーション開発ガイド:
サーバー・アプリケーションのプログラミング

バージョン 8.2

SC88-9139-01
(英文原典：SC09-4827-01)

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、*特記事項*に記載されている情報をお読みください。

本マニュアルに関するご意見やご感想は、次の URL からお送りください。今後の参考にさせていただきます。

http://www.ibm.com/jp/manuals/main/mail.html

なお、日本 IBM 発行のマニュアルはインターネット経由でもご購入いただけます。詳しくは

http://www.ibm.com/jp/manuals/ の「ご注文について」をご覧ください。

(URL は、変更になる場合があります)

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

原　典： SC09–4827–01
IBM® DB2 Universal Database™
Application Development Guide: Programming Server Applications
Version 8.2

発　行： 日本アイ・ビー・エム株式会社

担　当： ナショナル・ランゲージ・サポート

第1刷 2004.8

この文書では、平成明朝体™W3、平成明朝体™W9、平成角ゴシック体™W3、平成角ゴシック体™W5、および平成角ゴシック体™W7を使用しています。この(書体*)は、(財）日本規格協会と使用契約を締結し使用しているものです。フォントとして無断複製することは禁止されています。

注* 平成明朝体™W3、平成明朝体™W9、平成角ゴシック体™W3、
平成角ゴシック体™W5、平成角ゴシック体™W7

目次

本書について

アプリケーション開発ガイド は、DB2 アプリケーションのコーディング、デバッグ、ビルド、および実行に関して知っておくべきことを説明した 3 冊からなるガイドです。

- *アプリケーション開発ガイド クライアント・アプリケーションのプログラミング* では、DB2 クライアントで実行されるスタンドアロン DB2 クライアントをコーディングする際に知っておくべきことを説明しています。以下の情報を扱います。
  - DB2 でサポートされているプログラミング・インターフェース。DB2 Developer's Edition、サポートされているプログラミング・インターフェース、Web アプリケーションを作成するための機能、および DB2 が提供するルーチンやトリガーなどのプログラミング機能についてハイレベルな説明がなされています。
  - DB2 アプリケーションが従うべき一般的な構造。データベース内のデータ値やリレーションシップの推奨される保守方法や、許可に関する考慮事項が説明されており、アプリケーションのテストとデバッグ方法に関する情報もあります。
  - 動的組み込み SQL と静的組み込み SQL。組み込み SQL に関する一般的な考慮事項、および DB2 アプリケーションで静的 SQL と動的 SQL を使用する際の特有な考慮事項。
  - C/C++、COBOL、Perl、および REXX などのサポートされているホストおよびインタープリター言語とこれらの言語で書かれたアプリケーションでの組み込み SQL の使用方法。
  - DB2 .NET Data Provider、OLE DB .NET データ・プロバイダー、ODBC .NET データ・プロバイダー。
  - Java (JDBC と SQLJ) と、 WebSphere Application Server で使用する Java アプリケーションを構築する際の考慮事項。
  - IBM OLE DB Provider for DB2 Server。 IBM OLE DB Provider による OLE DB サービス、コンポーネント、およびプロパティーのサポートに関する一般情報。 ActiveX Data Objects (ADO) 用の OLE DB インターフェースを使用する Visual Basic および Visual C++ アプリケーション特有の情報があります。
  - 各国語サポートの問題。照合シーケンス、コード・ページとロケールから派生する問題、および文字変換などの一般トピックが説明されています。 DBCS コード・ページ、EUC 文字セット、および日本語と中国語 (繁体字) EUC および UCS-2 環境に適用される問題についても説明されます。
  - トランザクション管理。マルチサイト更新を実行するアプリケーション、および並行トランザクションを実行するアプリケーションに適用される問題が説明されています。
  - パーティション・データベース環境におけるアプリケーション。パーティション・データベース環境における指示 DSS、ローカル・バイパス、バッファー挿入、アプリケーションのトラブルシューティングについて説明します。

- 一般的に使用されるアプリケーション技法。生成列と ID 列、宣言済み一時表の使用方法、およびトランザクションを管理するためのセーブポイントの使用法について説明されます。
- 組み込み SQL アプリケーションの使用がサポートされている SQL ステートメント。
- ホストおよび iSeries 環境にアクセスするアプリケーション。ホストおよび iSeries 環境にアクセスする組み込み SQL アプリケーションに関する問題。
- EBCDIC バイナリー照合のシミュレーション。

• *アプリケーション開発ガイド サーバー・アプリケーションのプログラミング* では、ルーチン、ラージ・オブジェクト、ユーザー定義タイプ、トリガーなどのサーバー・サイドのオブジェクトを使用するプログラミングを行う上で知っておくべきことが説明されています。以下の情報を扱います。
  - ルーチン (ストアード・プロシージャー、ユーザー定義関数、およびメソッド)。以下のことも扱われています。
    - ルーチンのパフォーマンス、セキュリティー、ライブラリー管理上の考慮事項、および制限。
    - ルーチン (外部ルーチンも含む) の作成と、CREATE ステートメント。
    - プロシージャーのパラメーター・モードおよびパラメーターの取り扱い。
    - プロシージャーの結果セット。
    - デバッグおよび条件処理を含む SQL プロシージャー。
    - ユーザー定義のスカラー関数および表関数。
    - ユーザー定義のスカラー関数呼び出しおよび表関数呼び出し (FIRST 呼び出し、FINAL 呼び出し...) およびスクラッチパッド。
    - メソッド。
    - 許可、および外部ルーチンのバインディング。
    - C、Java、.NET 共通言語ランタイム、および OLE オートメーション・ルーチンの言語特有の考慮事項。
    - ルーチンの呼び出し。
    - 関数選択。
    - 関数に対する特殊タイプと LOB の引き渡し。
    - コード・ページとルーチン。
  - LOB の使用法とロケーター、参照変数、および CLOB データを含むラージ・オブジェクト。
  - ユーザー定義特殊タイプ (UDT) (以下のことの説明も含まれます)。強い型定義、UDT の定義とドロップ、構造化型による表の作成、特定のアプリケーション用の特殊タイプと型付き表の使用、複数の特殊タイプの取り扱いとそれらの間のキャスト、特殊タイプ間の比較と代入、特殊タイプ列における UNION 操作。
  - ユーザー定義構造化型 (以下のことも説明されています)。インスタンスの保管インスタンス生成、構造化型の階層、構造化型の動作の定義、メソッドの動的ディスパッチング、比較関数、cast 関数、コンストラクター関数、および構造化型用の mutator メソッドと observer メソッド。

- 型付き表 (以下のことも説明されています)。オブジェクトの作成・ドロップ・置換・保管、システム生成オブジェクト ID の定義、およびオブジェクト ID 列における制約。
- 参照タイプ (以下のことも説明されています)。型付き表のオブジェクト間のリレーションシップ、参照のあるセマンティック・リレーションシップ、および参照保全と有効範囲参照。
- 型付き表と型付きビュー (以下のことも説明されています)。列型としての構造化型、transform 関数と transform グループ、ホスト言語プログラムのマッピング、および構造化型ホスト変数。
- トリガー (以下のことも説明されています)。 INSERT/UPDATE/DELETE トリガー、参照制約との相互作用、作成に関するガイドライン、細分性、活動化時間、遷移変数と表、トリガー・アクション、多重トリガー、および複数のトリガーと制約とルーチン間の協同。

• *アプリケーション開発ガイド アプリケーションの構築および実行* では、DB2 でサポートされている以下のオペレーティング・システムにおいて DB2 アプリケーションをビルドして実行する上で知っておくべきことが説明されています。

- AIX
- HP-UX
- Linux
- Solaris
- Windows

以下の情報を扱います。

- DB2 がサポートするコンパイラーとインタープリターを含め、アプリケーションを作成するためにサポートされているサーバーとソフトウェア。
- DB2 サンプル・プログラム・ファイル、makefile、ビルド・ファイル、およびエラー・チェック・ユーティリティー・ファイル。
- アプリケーション開発環境のセットアップ方法 (Java 関数と WebSphere MQ 関数に関する具体的な説明も含みます)。
- サンプル・データベースのセットアップ方法。
- 以前のバージョンの DB2 からのアプリケーションの移行方法。
- Java アプレット、アプリケーション、およびルーチンのビルドと実行の仕方。
- SQL プロシージャーのビルドと実行の仕方。
- C/C++ アプリケーションとルーチンのビルドと実行の仕方。
- IBM COBOL および Micro Focus COBOL アプリケーションとルーチンのビルドと実行の仕方。
- AIX および Windows における REXX アプリケーションのビルドと実行の仕方。
- Windows 上での C# および Visual Basic .NET アプリケーションおよび CLR .NET ルーチンのビルドと実行の仕方。
- Windows における Visual Basic および Visual C++ を使用した ActiveX Data Object (ADO) のあるアプリケーションのビルドと実行の仕方。
- Windows における Visual C++ を使用したリモート・データ・オブジェクトのあるアプリケーションのビルドと実行の仕方。

第 1 部 ルーチン

第 1 章 ルーチンの概要

アプリケーション開発におけるルーチン

ルーチンとは、特定のタスクと関連したプログラミング・ロジックとデータベース・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。ルーチンには、プロシージャー、関数、メソッドという 3 つのタイプがあります。ロジックとデータベース操作を組み込むためのインターフェースは各ルーチン・タイプで異なりますが、それぞれのインターフェースを使用することで、 SQL ステートメントまたはクライアント・アプリケーションの機能を拡張できます。データベース・アプリケーションの開発時または更新時には、ルーチンを作成して使用することから得られる多くのメリットについて検討するべきです。

データベースと対話する新機能を開発するタスクに取り組む場合は、 2 種類の方法のいずれかを選択できます。つまり、クライアント・アプリケーションに新規のロジックを追加する方法と、ルーチンを開発してデータベース・サーバーに新規のロジックを作成する方法です。後者の方法を選択した場合には、いくつかの利点があります。

ルーチンを使用することの利点:

以下の利点を活用するには、アプリケーション・ロジックをルーチンに移動します。

アプリケーション・ロジックのカプセル化
多数のクライアント・コンピューターがそれぞれ多種多様なデータベース・アプリケーションを実行している環境では、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。たとえば、ルーチンを使用している環境内であれば、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ルーチンを変更するだけで済みます。そのような場合に、ルーチンを使用していなかったとしたら、各クライアント・アプリケーションでアプリケーション・ロジックを変更しなければなりません。

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。たとえば、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特定の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。

ネットワーク・トラフィックの削減
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サーバ

ー・コンピューターに送信され、結果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、多くのデータベース・アクティビティーを実行する処理があれば、その処理をサーバー上にインストールするというのは筋の通ったことです。この処理をサーバー上で実行すれば、クライアント・コンピューターとサーバー・コンピューターの間のネットワーク・トラフィックの量は削減されます。 DB2 ルーチンはデータベース・サーバー上でこの処理を行います。ルーチンを使用することによって、効率的にネットワーク・トラフィックを削減し、クライアント・アプリケーション全体のパフォーマンスを向上させることができます。

クライアントでの処理作業負担の緩和

クライアント・コンピューターのパフォーマンスが重視される環境の場合、ルーチンはクライアント・コンピューターへの依存度を低くするための具体的な手段になります。アプリケーションがルーチンを呼び出すと、ルーチンの処理がデータベース・サーバー上で行われるので、アプリケーションは、データベース・サーバーの機能を活用しながら、クライアント・コンピューターの処理作業負担を軽減できます。

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データベース・マネージャーとの関係が近いと言えます。ルーチンのタイプによっては、ルーチンの SQL ステートメントのほうが、クライアント・アプリケーションから実行する SQL ステートメントよりもパフォーマンスが優れています。たとえば、NOT FENCED ルーチンは、通信に共用メモリーを使用してデータベース・マネージャーと同じプロセスで処理を行います。したがって、この種のルーチンは、 TCP/IP プロトコルによって通信するクライアント・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。

ロジックのインプリメンテーションのインターオペラビリティー

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て、 DB2® のルーチンは、高いインターオペラビリティーを備えています。

- 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を呼び出すことができます。たとえば、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。たとえば、Java™ プロシージャー (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。

関連概念:

- 13 ページの『プロシージャー』
- 217 ページの『ルーチンの呼び出し』
- 21 ページの『サポートされているルーチン・プログラミング言語』
- 15 ページの『ユーザー定義のスカラー関数』
- 18 ページの『メソッド』
- 17 ページの『ユーザー定義のスカラー関数』

関連タスク:

- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『JDBC ルーチンの作成』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『JDBC ルーチンの作成』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『UNIX C ルーチンの構築』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『UNIX C++ ルーチンの構築』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『Building IBM COBOL routines on AIX』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『UNIX Micro Focus COBOL ルーチンの構築』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『Windows での C/C++ ルーチンの構築』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『Windows での IBM COBOL ルーチンの構築』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『Windows での Micro Focus COBOL ルーチンの構築』
- 37 ページの『ルーチンの作成』
- 36 ページの『データベースでのルーチンの作成』
- 43 ページの『ルーチンのデバッグ』

ルーチンのタイプ (プロシージャー、関数、メソッド)

ルーチンは、主に機能によってグループ分けしますが、インプリメンテーション別にグループ分けすることもできます。ルーチンの主な機能タイプは、プロシージャー (ストアード・プロシージャーともいう)、関数、メソッドの 3 つです。ルーチンをインプリメンテーション別に分けるとすれば、組み込み、ソース派生、SQL、外

部などに分類できます。ここでは、まずルーチンの機能タイプについて説明してから、可能なインプリメンテーションについて取り上げます。

ルーチンの機能タイプ:

プロシージャー
プロシージャー (ストアード・プロシージャーともいう) 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。プロシージャーを呼び出すには、そのプロシージャーへの参照を指定した CALL ステートメントを実行します。

関数 関数は、入力データ値のセットと結果値のセットとの間のリレーションシップです。関数は、SQL を拡張してカスタマイズする手段になります。関数の呼び出しは、選択リストや FROM 文節など、 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。

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 文節内の組などで行えます。この値のセットのことを引き数セットといいます。たとえば、以下の照会は、集約関数 SUM を使用して、在庫と注文を合わせた全種類のボルトの合計数量を計算します。

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

集約関数を外部関数としてインプリメントすることはできません。集約関数は、組み込み集約関数から派生したソース派生関数としてのみインプリメントできます。

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。たとえば、`length` や `substr` などはスカラー関数です。さらに、入力パラメーターに対して複雑な数値計算を行うスカラー関数を作成することも可能です。選択リストや FROM 文節など、 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。スカラー関数は、外部関数としてもソース派生関数としてもインプリメントできます。

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。この関数は、構造化型の属性を行内の組み込みデータ型にマップする transform 関数としてのみ使用することができます。行関数は、SQL 関数としてのみインプリメントできます。

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 文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など、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。表関数は、SQL でも外部プログラム言語でもインプリメントできます。

メソッド
: メソッドは、構造化型に対して振る舞いを提供するロジックをカプセル化したものです。構造化型は、1 つ以上の名前付き属性を含み、それぞれがデータ型を持っている、ユーザー定義データ型です。属性は、タイプのインスタンスを記述するプロパティーです。たとえば、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、基本的に構造化型の属性に対する操作を表すために、その構造化型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、SQL メソッドまたは外部メソッドとしてインプリメントできます。

以下の図に、ルーチンの種別階層を示します。

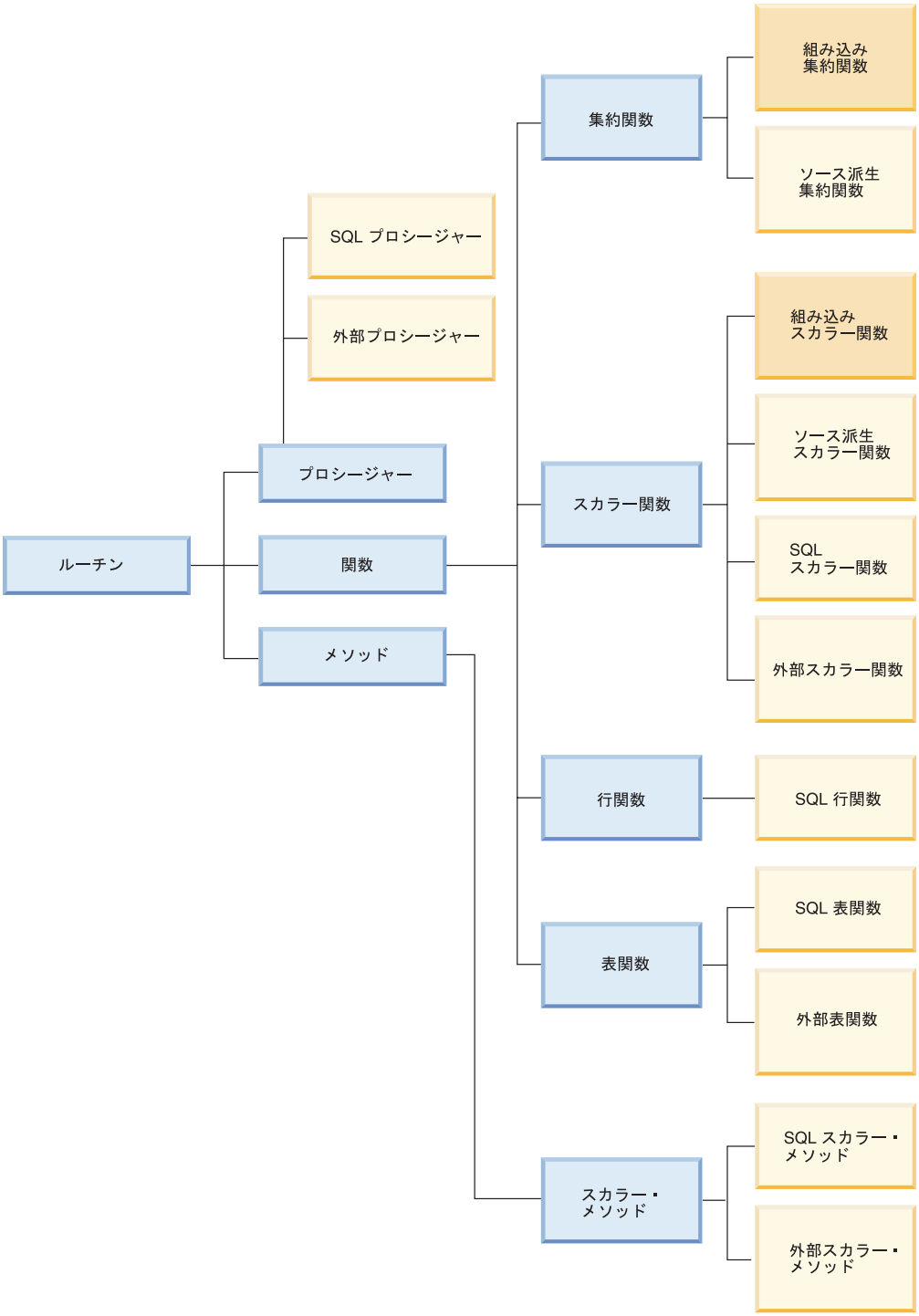


図 1. ルーチンの種別

ルーチンのインプリメンテーションのタイプ:

ルーチンをインプリメンテーション別に分けるとすれば、組み込み、ソース派生、SQL、外部などに分類できます。

組み込み

いくつかのルーチンが DB2 システム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強く型定義されており、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、頻繁に必要になる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。

それ以外のインプリメンテーションは、ユーザー主導のインプリメンテーションです。組み込み関数とは異なり、これらのインプリメンテーションの場合は、ユーザーが各ルーチン・タイプの CREATE ステートメントを使用してルーチンを明示的に作成する必要があります。ユーザーが作成した関数やプロシージャーは、SYSTOOLS スキーマの中に配置されます。

ソース派生

ソース派生関数は、別の関数 (ソース関数) のセマンティクスを複製した関数です。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

SQL ルーチンは、SQL ステートメントだけで成っています。そのステートメントは、データベース内にルーチンを作成するときに使用する CREATE ステートメント内に指定します。 SQL Procedural Language (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 (割り当てステートメント)、プロシージャー・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 のプロシージャー、関数、メソッドの作成には、 SQL と SQL PL (限定的な状況では SQL PL のサブセット) を使用できます。

外部 この種のルーチン自体は、各ルーチン・タイプ固有の CREATE ステートメントを使用してデータベース内に作成しますが、ルーチンのロジックは、外部のホスト・プログラム言語アプリケーションでインプリメントします。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、 CREATE ステートメントの EXTERNAL 文節で宣言します。外部ルーチンは、C、

C++、Java™、OLE と、 .NET 共通言語ランタイムがサポートするプログラム言語で作成できます。外部プロシージャーの場合は、COBOL で記述することもできます。

DB2 に用意されているルーチン:

DB2 の SYSPROC、SYSFUN、SYSTOOLS の各スキーマには、いくつかのプロシージャーと関数が用意されています。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。たとえば、SNAPSHOT_TABLE、HEALTH_DB_HI、 SNAPSHOT_FILEW、REBIND_ROUTINE_PACKAGE などがあります。 CURRENT PATH 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ジャーをすぐに使用できます。これらのスキーマは、デフォルトで CURRENT PATH に入っています。

関連概念:

- 3 ページの『アプリケーション開発におけるルーチン』
- 24 ページの『ルーチン開発に関するパフォーマンスの考慮』
- 27 ページの『ルーチンのセキュリティーに関する考慮事項』

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組み込みルーチンが用意されていますが、ユーザーが独自のロジックをカプセル化するために独自のルーチンを作成する余地も残されています。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、ユーザーが独自のプロシージャー、メソッド、関数を作成できるということです。ただし、ユーザー定義関数は一般に UDF と表記しますが、プロシージャー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。

ルーチンの CREATE ステートメント:

ユーザー定義のプロシージャー、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・クラスに該当する 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、 CREATE PROCEDURE、CREATE FUNCTION、CREATE METHOD があります。各 CREATE ステートメントに固有の文節では、ルーチンの特性を定義します。具体的には、ルーチン名、ルーチンの引き数の数とタイプ、ルーチン・ロジックの詳細などの特性があります。 DB2 は、それらの文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。

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、「SQL」、「外部」、「ソース派生」という 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。ここでは、それぞれのインプリメンテーションを比較します。各インプリメンテーションのメリットと用途を確認してください。

SQL

SQL ルーチンのロジックは、ルーチンを作成する CREATE ステートメントの本体に指定する SQL ですべて記述します。 SQL プロシージャーの本体、SQL 関数の本体、SQL メソッドの本体はいずれも、 SQL ステートメントと SQL PL ステートメントから成っています。

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、大半が SQL ステートメントであり、一部にあまり複雑でない SQL PL ロジックを使用したインプリメンテーションの場合は、 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 プロシージャーの場合は、使いやすいエラー処理サポートも用意されています。ただし、SQL ルーチンにも制限があります。システム呼び出しに直接影響を与えることはできませんし、データベースの外部のエンティティーに対して操作を実行することもできません。ルーチンの機能タイプによっては、サポートされていない SQL ステートメントもあります。

外部

外部ルーチンは、データベース・サーバーのファイル・システム (つまり、データベース自体の外部) に存在するユーザー作成のライブラリーまたはクラスにロジックをインプリメントしたルーチンです。そのライブラリーまたはクラスは、 C、C++、Java™、OLE、.NET 互換言語のうちのいずれかのホスト・プログラム言語で記述したソース・アプリケーションからコンパイルできます。外部プロシージャーは、 COBOL で記述することもできます。OLE ルーチン以外のすべての外部ルーチンには SQL を組み込めます。

外部ルーチンは、 SQL ルーチンよりもインプリメントの方法が少し複雑ですが、選択したインプリメンテーション・プログラム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。たとえば、VARCHAR データ型を操作する新しいストリング関数や、 DOUBLE データ型を操作する複雑な数学関数など、組み込みデータ型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、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。データベース・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プログラム言語によるプログラミングが苦にならないのであれば、外部ルーチンの簡単な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。

ソース派生

ソース派生というインプリメンテーション・スタイルは、関数だけに該当します。ソース派生関数のロジックは、既存のソース関数から取り込みます。

ソース関数を指定するには、特別な CREATE FUNCTION (ソース派生またはテンプレート) ステートメントの SOURCE 文節を使用します。このインプリメンテーションに特に関連付けられている言語はありません。ソース派生関数を使用するのは、ソース・タイプに該当する関数と演算子の一部を選択的に継承した特殊タイプを使用するため、というのが最も一般的です。ソース派生関数は簡単にインプリメントでき、特に既存の関数の名前を変更したい場合に便利です。

ユーザー定義ルーチンの開発の概要:

ルーチンの開発では、 DB2 デベロップメント・センターを利用すると便利です。これには、簡単なインターフェースと一連のウィザードが装備されているので、開発タスクを容易に実行することができます。また、Microsoft® Visual Studio などの広く使用されているアプリケーション開発ツールに DB2 デベロップメント・センターを統合することもできます。その他の代替手段として、 DB2 コマンド行プロセッサーによってユーザー定義ルーチンを開発することも可能です。

ユーザー定義ルーチンの開発には、次のようなタスクがかかわってきます。

1. データベース内にルーチンを作成します。このタスク (ルーチンの定義または登録ともいう) は、基本的にルーチンの呼び出し前であればどの時点でも実行できますが、以下の例外的な状況があります。
   - 1 つ以上の外部 JAR ファイルを参照する Java ルーチンの場合は、外部のコードと JAR ファイルを記述し、コンパイルしてから、ルーチン・タイプ固有の CREATE ステートメントによって、データベース内にルーチンを作成する必要があります。
   - SQL ステートメントを実行し、自身を直接参照するルーチンの場合は、そのルーチンに関連付けられている外部コードをプリコンパイルし、バインドする前に、 CREATE ステートメントによってデータベース内にルーチンを作成する必要があります。これは、たとえばルーチン A がルーチン B を参照し、さらにルーチン B がルーチン A を参照する循環参照の場合にも当てはまります。
2. 外部ルーチンの場合は、ルーチン・ロジックを記述します。 SQL ルーチンのロジックは、 SQL ルーチンの CREATE ステートメントに組み込みます。
3. 外部ルーチンの場合は、ルーチンをビルドします。ビルドの中身は、コンパイルとリンクです (ただし、組み込み SQL を使用したルーチンの場合は、最初にプリコンパイルという段階が入ります)。 (オペレーティング・システムと言語に固有のビルドの詳細については、関連リンクを参照してください。)
4. ルーチンのデバッグとテストを実行します。
5. ルーチンの呼び出し元に、ルーチンの EXECUTE 権限を与えます。
6. ルーチンを呼び出します。

関連概念:

- 13 ページの『プロシージャー』
- 116 ページの『外部ルーチンでの SQL』
- 5 ページの『ルーチンのタイプ (プロシージャー、関数、メソッド)』
- 65 ページの『ユーザー定義表関数』
- 15 ページの『ユーザー定義のスカラー関数』

- 18 ページの『メソッド』
- 73 ページの『SQL ルーチンの SQL アクセス・レベル』
- 71 ページの『DB2 での SQL Procedural Language (SQL PL)』

プロシージャー、関数、メソッドの比較

開発できるルーチンには、プロシージャー、ユーザー定義関数 (UDF)、メソッドという 3 つのタイプがあります。それぞれのタイプのルーチンの作成とインプリメントにかかわる詳細は似ていますが、それぞれの目的は異なります。

以下の項では、比較しやすい形式で各ルーチン・タイプの機能を示します。 UDF の場合は、ユーザー定義スカラー関数とユーザー定義表関数という 2 つの項に分かれていることに注意してください。この 2 つにはかなりの違いがあるので、別々に取り上げています。

プロシージャー

プロシージャー (ストアード・プロシージャーともいう) は、ロジックと SQL ステートメントをカプセル化できるデータベース・オブジェクトであり、 CREATE PROCEDURE ステートメントの実行によって作成します。プロシージャーは、アプリケーションや、ロジックを含む他のデータベース・オブジェクトに対するサブルーチン拡張として使用します。

機能

- SQL ステートメント、関数呼び出し、再利用可能な特定のサブルーチン・モジュールを定式化したロジックの各エレメントをカプセル化します。
- プロシージャーは、クライアント・アプリケーション、他のルーチン、トリガー、動的コンパウンド・ステートメントから呼び出せます。 CALL ステートメントを使用して呼び出すことも可能です。
- プロシージャーは複数の結果セットを戻すことができます。
- プロシージャーには、単一パーティション・データベースと複数パーティション・データベースの両方の表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。
- プロシージャーを呼び出すと、その中の SQL とロジックがサーバー上で実行されます。クライアントとデータベース・サーバーの間のデータのやり取りは、プロシージャーの呼び出し時とプロシージャーの戻り時だけに発生します。 1 つのクライアント・アプリケーションで一連の SQL ステートメントを実行する場合、そのアプリケーションがステートメントとステートメントの間で他の処理を実行する必要がなければ、その一連のステートメントを 1 つのプロシージャーに組み込むと便利です。

  注: プロシージャーで 1 つの SQL ステートメントしか呼び出さない場合は、その呼び出しのセットアップにかかるオーバーヘッドのほうが、ネットワーク・トラフィックの削減という利点を上回ってしまうことがあります。

制限

- プロシージャーは、SQL 照会のエレメント内から呼び出すことを想定していません。プロシージャーを呼び出せるのは、CALL ステートメントだけです (サポートされている場合)。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。プロシージャーで結果セットを戻すことはできますが、 SQL 照会の FROM 文節内で表を戻すには表関数を使用できます。
- プロシージャー呼び出しの出力引き数を別の SQL ステートメントで直接使用することはできません。
- プロシージャーは、呼び出しと呼び出しの間で状態を保存できません。

一般的な使用法

- 特定のタスクと関連したデータベース・ロジックだけをカプセル化したアプリケーション・サブルーチンをインプリメントするために使用できます。たとえば、従業員情報を管理するビジネス・アプリケーションから、従業員の雇用に伴うデータベース操作をカプセル化したプロシージャーを呼び出す、といった使用法があります。

  この種のプロシージャーでは、従業員表、部門表、給付金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、出力パラメーターの 1 つ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ーでは、従業員表にデータの統計分析を入れ、分析結果を含んだ結果セットを戻すこともできます。このようにしてプロシージャーを使用すれば、アプリケーション内でデータベース・タスクと非データベース・タスクを効果的に切り分けることができます。
- アプリケーション・ロジックを標準化します。複数のアプリケーションからデータベースに同じようにアクセスして変更操作を実行する必要がある場合は、 1 つのプロシージャーによって、そのアクセスや変更のための統一的なインターフェースを提供できます。そのプロシージャーは、どのアプリケーションからでも使用できます。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しなければならない場合でも、その 1 つのプロシージャーを変更するだけで済みます。

サポートされている言語

- SQL
- C/C++
- Java™
- OLE
- COBOL
- .NET 共通言語ランタイム言語

注: SQL プロシージャーはネイティブにサポートされているので、コンパイラーをインストールする必要はありません。

関連概念:

- 3 ページの『アプリケーション開発におけるルーチン』
- 48 ページの『プロシージャーのパラメーター・モード』
- 48 ページの『プロシージャーの結果セット』

- 71 ページの『DB2 での SQL Procedural Language (SQL PL)』

関連タスク:

- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『アプリケーション開発環境のセットアップ』
- 77 ページの『コマンド行からの SQL プロシージャーの作成』
- 227 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ーの呼び出し』
- 226 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ーの呼び出し』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CALL ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE ステートメント』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『DB2 でサポートされる開発ソフトウェア』

ユーザー定義のスカラー関数

スカラーユーザー定義関数 (UDF) は、 SQL ステートメントを拡張してカスタマイズする手段になります。これは、DB2® 組み込み関数 (たとえば、 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。スカラー UDF は、入力引き数としてゼロ個以上の型付き値を受け入れて、呼び出しごとに 1 つの値を戻します。

SQL スカラー・ユーザー定義関数:

SQL スカラー UDF を使用すれば、基本的なデータベース・ロジックのインプリメントに使用できる SQL ステートメント、組み込み関数やその他のルーチン参照、SQL PL ステートメントのサブセットをまとめてカプセル化できます。 SQL スカラー関数では、SQL データの読み取りと変更を実行できます。 SQL 関数が最高のパフォーマンスを発揮するのは、組み込み関数を利用し、あまりにも複雑なロジックを組み込まない場合です。あまりにも複雑なロジックを組み込む場合は、外部スカラー UDF のインプリメントを検討してください。

外部スカラー・ユーザー定義関数:

外部スカラー UDF の場合は、外部のプログラム言語でロジックをインプリメントします。この関数のロジックでは、ファイル・システムへのアクセスや、システム呼び出しや、ネットワークへのアクセスを実行します。外部スカラー UDF ルーチンのロジックは、 SQL スカラー UDF の場合と同じくサーバー上で実行します。外部スカラー UDF 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 UDF 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。

SQL スカラー UDF と外部スカラー UDF の機能

- 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 UDF 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、関数を繰り返し呼び出すときに、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、述部で使用するとき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。
- 既存の組み込み関数からスカラー関数を構築するための優れた方法です。たとえば、他のロジックとの組み合わせの中で組み込みスカラー関数を再利用するだけで、複雑な数式を作成できます。

制限

- スカラー UDF 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 UDF 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 UDF 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 UDF は、1 度の呼び出しによる使用を想定していません。むしろ、UDF 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 UDF を 1 度ずつ呼び出し、そのたびに UDF 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 UDF 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。 1 度の呼び出しだけを必要とする機能には、SQL スカラー UDF 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 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。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 UDF に SQL ステートメントを含めることはできません。

  シリアル・データベースやパーティション・データベースでは、 SQL スカラー UDF にデータベース表からデータを読み取る 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。たとえば、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

- SQL
- 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。たとえば、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 UDF を集約関数として定義できます。
2. また、行を戻す UDF を作成することもできます。これは、行 UDF と呼ばれますが、構造化型用の transform 関数としてのみ使用することができます。行 UDF の出力は単一行です。

関連概念:

- 3 ページの『アプリケーション開発におけるルーチン』
- 60 ページの『UDF とメソッドのスクラッチパッド』

関連タスク:

- 238 ページの『スカラー関数またはメソッドの呼び出し』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION ステートメント』

ユーザー定義のスカラー関数

スカラー UDF と同様に表 UDF も、 SQL を拡張してカスタマイズする手段になりますが、その目的は表を生成することにあります。表 UDF は、SQL ステートメントの FROM 文節内からのみ呼び出すことができます。表 UDF は、入力引き数としてゼロ個以上の型付き値を受け入れて、表を戻します。

表関数は、ほとんどすべてのデータ・ソースを DB2® 表として戻すことができる強力な関数です。表関数の作成は簡単です。必要なデータを収集するプログラムを作成し、必要に応じていくつかの入力パラメーターをフィルターとしてデータを絞り込み、そのデータを DB2 に 1 行ずつ戻すだけのことです。

機能

- SQL ステートメントの FROM 文節の一部として参照できます。
- 外部表関数では、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。

- 結果は、表関数を参照する SQL ステートメントで直接処理できます。
- 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。 (このプロパティーがあるのは SQL 表関数だけです)
- 1 つの表関数参照によって表関数を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。
- 処理用の一連のデータを提供します。

制限

- トランザクション管理を行うことはできません。つまり、表関数内では COMMIT ステートメントや ROLLBACK ステートメントを実行できません。
- 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。
- FROM 文節内でしか使用することはできません。
- 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。 SQL データを変更するステートメントを含める場合は、 SQL 表関数を使用します。

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。たとえば、スプレッドシートを読み取って表を作成してから、それを DB2® 表に挿入することができます。

サポートされている言語

- SQL
- C/C++
- Java™
- OLE
- OLE DB
- .NET 共通言語ランタイム言語

関連概念:

- 3 ページの『アプリケーション開発におけるルーチン』
- 60 ページの『UDF とメソッドのスクラッチパッド』
- 66 ページの『表関数の処理モデル』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION ステートメント』

メソッド

メソッドは、構造化型の動作を定義する手段になります。これはスカラー UDF に似ていますが、構造化型に対してしか定義することはできません。メソッドは、以下の機能に加えて、スカラー UDF のすべての機能も持っています。

機能

- 構造化型に緊密に関連付けられています。
- サブジェクト・タイプの動的タイプに重点を置くことができます。

制限

- スカラー値しか戻すことはできません。
- 構造化型に対してしか使用できません。
- 型付き表を対象として呼び出すことはできません。

一般的な使用法

- 構造化型に対する操作の手段になります。
- 構造化型をカプセル化します。

サポートされている言語

- SQL
- C/C++
- Java™
- OLE

関連概念:

- 3 ページの『アプリケーション開発におけるルーチン』
- 60 ページの『UDF とメソッドのスクラッチパッド』

関連タスク:

- 277 ページの『構造化型の振る舞いの定義』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE TYPE (構造化) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE METHOD ステートメント』

第 2 章 ルーチンの開発

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ー。

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 Procedural Language (SQL PL) エレメントを組み込むこともできます。 SQL PL は、 SQL ルーチンを短時間でインプリメントすることを可能にする高水準の使いやすい言語です。
- 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ーが実現されます。

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、 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ーに関する考慮事項』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ー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ーに優れています。 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 オートメーション・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ー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ーにあまり優れていないことも意味します。

関連概念:

- 24 ページの『ルーチン開発に関するパフォーマンスの考慮』
- 27 ページの『ルーチンのセキュリティーに関する考慮事項』
- 170 ページの『C/C++ ルーチン』
- 189 ページの『Java ルーチン』
- 71 ページの『DB2 での SQL Procedural Language (SQL PL)』

関連タスク:

- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『JDBC ルーチンの作成』

- 「アプリケーション開発ガイド アプリケーションの構築および実行」の『JDBC ルーチンの作成』
- 「アプリケーション開発ガイド アプリケーションの構築および実行」の『SQL プロシージャーの作成』
- 「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『UNIX での CLI ルーチンの作成』
- 「アプリケーション開発ガイド アプリケーションの構築および実行」の『UNIX C ルーチンの構築』
- 「アプリケーション開発ガイド アプリケーションの構築および実行」の『UNIX C++ ルーチンの構築』
- 「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『Windows での CLI ルーチンの作成』
- 「アプリケーション開発ガイド アプリケーションの構築および実行」の『Windows での C/C++ ルーチンの構築』

ルーチンの開発の最も望ましい実践法

以下の項では、正常に機能する安全なルーチンを開発するためにお勧めする実践法を解説しています。

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パフォーマンスの向上は期待できますが、データベース制御構造がユーザー・コードによって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ーに関する考慮事項』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 文節を使用します。危険の潜むコードの作成および実行を制限するために、 DB2 では、NOT FENCED ルーチンを作成するユーザーの必須条件として、CREATE_NOT_FENCED_ROUTINE という特別な権限を用意しています。

ルーチンが NO SQL として登録されている場合に、NOT FENCED ルーチンの実行中に異常終了が起きたら、データベース・マネージャーは該当する

リカバリーを試みます。一方、NO SQL と定義されていないルーチンの場合は、データベース・マネージャーは失敗します。

NOT FENCED ルーチンが GRAPHIC または DBCLOB データを使用する場合、 WCHARTYPE NOCONVERT オプションを使用してこのルーチンをプリコンパイルしなければなりません。

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。具体的に言うと、非 Java ルーチンは 1 つのプロセスを共用するのに対して、 Java™ ルーチンは、他の言語で書かれたルーチンとは別個の別のプロセスを共用します。このような個別化によって、他の言語で書かれたエラーの可能性のより高いルーチンから Java ルーチンは保護されます。また、Java ルーチンのプロセスには JVM が入っていますが、これはメモリー・コストが高いので、他のルーチン・タイプでは使用されません。FENCED THREADSAFE ルーチンを複数回呼び出せば、リソースが共用されるので、呼び出しごとにおのおのが独自の専用プロセスで稼働する FENCED NOT THREADSAFE ルーチンよりもシステムのオーバーヘッドは小さくて済みます。

他のルーチンと同じプロセスで実行しても十分に安全と思われるルーチンの場合、登録時には THREADSAFE 文節を使用します。 NOT FENCED ルーチンと同様に、 C/C++ ルーチンを FENCED THREADSAFE として登録する場合のリスクの評価とその軽減に関する詳細は、『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。プロセス中の他のルーチンは、稼働を継続します。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。 1 つのスレッドが異常終了した後は、プロセスは、新規のルーチンの呼び出しには使用されません。すべてのアクティブ・ユーザーがそのプロセスでのジョブを完了したら、そのプロセスは終了されます。

Java ルーチンを登録すると、他に指示しない限り必ず THREADSAFE になります。他のどの LANGUAGE タイプも、デフォルトで NOT THREADSAFE になることはありません。 LANGUAGE OLE および OLE DB を使用するルーチンを THREADSAFE と指定することはできません。

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)` を見つけ出せば、 Java プロセスと C THREADSAFE プロセスを確認することができます。

FENCED NOT THREADSAFE モード

FENCED NOT THREADSAFE ルーチンはいずれも、独自の専用プロセスで稼働します。多数のルーチンを実行している場合は、それによってデータベース・システムのパフォーマンスが悪影響を受けることがあります。他のル

ーチンと同じプロセスで実行するには十分安全ではないルーチンの場合、登録時には NOT THREADSAFE 文節を使用します。

UNIX では NOT THREADSAFE プロセスは db2fmp (pid) (ただし 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) と表示されますが、プールされた NOT THREADSAFE db2fmp の場合は db2fmp (idle) と表示されます。

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 ルーチンが呼び出されると、DB2 は、このルーチンの実行に十分な大きさの Java ヒープをもったスレッド化された Java fenced モード・プロセスの選択を試みます。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 プロシージャーのほうが 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になります。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 文字にとって代わる VARCHAR パラメーター

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 データ型ではなく VARCHAR データ型を使用すると、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。

たとえば、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 で終了させてから、ネットワークを介して 200 文字のストリングと NULL 終止符全体をストアード・プロシージャーに送信する必要があります。

それに対して、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″ を渡せば、 DB2 はネットワークを介して 14 文字のストリングと NULL 終止符を渡すだけで済みます。

関連概念:

- 「*アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」の『C および C++ での WCHARTYPE プリコンパイラー・オプション』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『WCHARTYPE CONVERT プリコンパイル・オプション』
- 27 ページの『ルーチンのセキュリティーに関する考慮事項』
- 170 ページの『C/C++ ルーチン』
- 189 ページの『Java ルーチン』
- 33 ページの『ルーチンの使用に関する制約事項』
- 30 ページの『ライブラリーおよびクラスの管理に関する考慮事項』
- 87 ページの『SQL プロシージャーのパフォーマンスの改善』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CALL ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE TYPE (構造化) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE METHOD ステートメント』

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いてはなりません。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 以外のプログラム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いてはなりません。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、支障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な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てはなりません。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てはなりません (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。たとえば、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を呼び出すと、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ので、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弱点

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、保全性、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。

ルーチンの保全性を保護するには、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。

セキュリティー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、不測の副次効果を生じなかった場合に限り、 FENCED および NOT THREADSAFE で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。データベースの保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。

関連概念:

- 3 ページの『アプリケーション開発におけるルーチン』
- 24 ページの『ルーチン開発に関するパフォーマンスの考慮』
- 33 ページの『ルーチンの使用に関する制約事項』
- 30 ページの『ライブラリーおよびクラスの管理に関する考慮事項』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『GRANT (ルーチン特権) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『REVOKE (ルーチン特権) ステートメント』

ライブラリーおよびクラスの管理に関する考慮事項

DB2® のルーチンを開発するときには、 SQL、Java™、C、C++、.NET 互換言語など、多様な選択肢の中からプログラム言語を選択できます。 SQL 以外の言語でルーチンを開発する場合には、外部ルーチンとして認識されます。外部ルーチンのコンパイル済みソース・コードは、ルーチン本体と呼ばれます。

ルーチン本体の保護

外部ルーチンの本体は、データベース・サーバーに保管されるライブラリーおよびクラス内に置かれます。そのファイルは、DB2 ではどのような手段によってもバックアップまたは保護されることはありません。データベース内にルーチンを作成するための CREATE ステートメントは、データベース・カタログにルーチンの定義情報を追加します。この追加される情報には、ルーチンに関連した外部コード・ライブラリーが存在する場所についての情報も含まれます。その場所は、CREATE ステートメントの EXTERNAL 文節で指定します。 EXTERNAL 文節で指定するルーチンのライブラリー

またはクラスは、データベースに保管するのではなく、むしろサーバーのファイル・システムに配置します。外部ルーチンを正常に呼び出すには、EXTERNAL 文節で指定するロケーションにそのルーチンに関連したライブラリーを配置しておく必要があります。そのライブラリーを移動したり削除したりすることもあり得ます。その場合は、ルーチンを正常に呼び出せなくなります。

呼び出し元クライアントおよびルーチンに依存するルーチンの保全性を維持するには、ルーチン本体が、無意識または意識的に削除されたり置き換えられたりしないようにする必要があります。それは、ルーチンが置かれたディレクトリーへのアクセスを管理し、そしてルーチン本体そのものを保護することで実現することができます。

注: SQL ルーチンの本体は、データベースの一部と見なされるので、他のデータベース・オブジェクトと一緒にバックアップされることになります。ただし、外部ルーチンと同様に、その本体は変更されることがよくあるので、同じような保護を必要とします。

ルーチン本体の有効範囲

データベース内で使用されるルーチンの場合、その同一のデータベースでカタログする必要があります。 1 つのインスタンスに複数のデータベースが存在する場合には、別のデータベースで既に使用されているルーチン本体を使用して、データベースに外部ルーチンをカタログできます。ですから、ルーチン本体の有効範囲はインスタンス全体になります。これにより、コードの再利用が可能になります。コードが再利用されていないところでは、ライブラリーまたはクラス名が競合する可能性があります。

特に、ライブラリーまたはクラス名が明らかに競合するのは、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を使用する場合です。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。これは、ルーチン本体は sqllib/function ディレクトリーに通常保管され、このディレクトリーはインスタンスのすべてのデータベースが使用するからです。

非 Java ルーチンの場合、ライブラリー名の競合は以下のステップによって解決できます。

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. ルーチンを、EXTERNAL NAME 文節を使用して特定のライブラリーの絶対パスを指定してカタログする。

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しません。 CLASSPATH で最初に出現するクラスが、使用されるクラスです。ですから、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。

ルーチン本体の更新

もしルーチンの本体を変更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と同じファイル (たとえば、sqllib/function/foo.a) を使用して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名を使用して、ルーチンの新規の本体を作成します。
2. 新規のルーチン本体 (組み込み SQL が含まれる場合) をデータベースにバインドします。
3. ALTER ステートメントを使用して、更新後のルーチン本体を参照するようにルーチンの EXTERNAL NAME を変更します。

ALTER を使用してルーチンのカタログ項目を更新し終わったら、以後は更新後のルーチンをいつ呼び出しても、必ずその新規のルーチン本体が指し示されます。

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。

注: 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。

ライブラリー管理に関連するパフォーマンスに関する考慮事項

DB2 ライブラリー・マネージャーは、ワークロードに沿ってライブラリー・キャッシングを動的に調整します。パフォーマンスを最大化するために、以下の点を考慮してください。

- 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。 1 つのライブラリー内に複数のルーチンを収容する場合、それらが同じ時間フレーム内に呼び出されるかどうかに基づいたグループ分けになっていることを確認してください。いくつかのアプリケーションにおいて、プロシージャー ProcA の呼び出しの後にプロシージャー ProcB の呼び出しが続くというシナリオを考察してみます。この場合は、ProcA と ProcB を同じライブラリーに収容するのが適しています。ライブラリー・キャッシング・スキームを使用する場合は、少数の大きなライブラリーよりも多数の小さいライブラリーを用意したほうがよいと思われます。
- C プロセスに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さらに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ロードする必要がありません。

パーティション・データベースのルーチン本体

パーティション・データベースの外部ルーチンを使用する場合、ライブラリーまたはクラスはそのデータベースのすべてのパーティションで使用できなければなりません。

UNIX® では、sqllib ディレクトリーはデータベースのすべてのパーティション間でクロスマウントされているため、sqllib/function がルーチン本体の適切なロケーションとなります。

Windows® では、すべてのパーティションにアクセスできる共有ディレクトリーを作成し、そのディレクトリーにライブラリーとクラスを置くことが適切な方法です。

関連概念:

- 24 ページの『ルーチン開発に関するパフォーマンスの考慮』
- 27 ページの『ルーチンのセキュリティーに関する考慮事項』
- 33 ページの『ルーチンの使用に関する制約事項』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE METHOD ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『ALTER FUNCTION ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『ALTER METHOD ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『ALTER PROCEDURE ステートメント』

ルーチンの使用に関する制約事項

以下の項では、ルーチンを開発する際の制約事項を説明しています。

- DB2® のバージョン 8 より前のエディションでは、CALL はコンパイル済みステートメントではなく、データ型の一致は実施されていませんでした。ルーチンの登録に使用するデータ型は、ルーチンで使用するデータ型に一致していなければなりません。 Java™、C、OLE オートメーション、および OLE DB データ型に対する SQL タイプのマッピングを示した表を参照してください。
- UDF は結果セットを戻すことはできません。 SQL を使用して UDF によってオープンされたカーソルはすべて、 FINAL 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。
- ルーチンは新規のスレッドを作成してはなりません。
- UDF またはメソッドからは、どの接続レベルの API も発行することはできません。
- 画面およびキーボードに対する入出力をルーチンから行うことはできません。したがって、たとえば、 Java の `System.out.println()`、 C/C++ の `printf()`、COBOL での `display` の呼び出しといった、標準入出力ストリームを使用してはなりません。 DB2 の処理モデルでは、ルーチンはバックグラウンドで実行されるため画面に表示することはできません。ただし、ルーチンはファイルに書き込むことはできます。

UNIX® 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もっていなければなりません。

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、たと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。

- ルーチンでは、以下を含め、接続に関連するステートメントまたはコマンドを実行できない。
  - BACKUP
  - CONNECT
  - CONNECT TO
  - CONNECT RESET
  - CREATE DATABASE
  - DROP DATABASE
  - FORWARD RECOVERY
  - RESTORE
- 一般的に、DB2 はオペレーティング・システム関数の使用を制限していない。ただし、次のようないくつかの例外があります。
  1. **ルーチンが独自のシグナル・ハンドラーをインストールしないことが必須です。** この制約事項を守らないと、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。
  2. 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、システム・エラーかアプリケーション・エラーが発生する。

     DB2 の通常の操作とやりとりしている場合、他のシステム呼び出しによって問題が発生することがあります。たとえば、UDF が含まれるライブラリーを UDF がメモリーからアンロードしようとすると、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ルーチンに入れてはならない。ルーチンは、現行プロセスを終了させることなく常に DB2 に制御を戻さなければなりません。
- ネストされたストアード・プロシージャー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。
- データベースがアクティブな間にルーチンの本体を変更してはならない。 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、

別のライブラリー名を使用してルーチンの別の本体を作成します。次に ALTER ステートメントを使用して、その新規の本体を参照するようにルーチンの EXTERNAL NAME を変更することができます。

- 'DB2' で始まるすべての環境変数の値は、 db2start を使用してデータベース・マネージャーが開始された時点でキャプチャーされて、 FENCED または NOT FENCED のどちらのルーチンでも使用できる。ただし DB2CKPTR 環境変数だけは例外です。その他の環境変数には NOT FENCED ルーチンからアクセスすることはできますが、 FENCED ルーチン・プロセス (たとえば LIBPATH) からはアクセスできません。環境変数はキャプチャーされる ということに注意してください。 db2start が発行された後に環境変数に対して加えられた変更は、ルーチンでは使用できません。
- 保護されたリソース (ルーチン内で一度に 1 つのプロセスにしかアクセスを許容しないリソース) を使用する際は、ルーチン同士のデッドロックが起きないようにしなければならない。複数のルーチンのデッドロックが発生しても、DB2 はその条件を検出も解決もできないので、ルーチン・プロセスがハングすることになります。
- 動的メモリーをルーチン内で割り振る際は、DB2 に戻る前にそのメモリーを解放する。そうしないと、メモリー・リークが発生し、DB2 プロセスが肥大化し続けます。それは、究極的にメモリー不足条件を引き起こします。

  UDF およびメソッドの場合、スクラッチパッド機能を使用すれば、必要な動的メモリーを複数の呼び出しにまたがって確保することができます。スクラッチパッドをこのように使用する場合は、割り振られたメモリーがステートメントの終わりの処理時に解放されるように、 UDF またはメソッドの CREATE ステートメントに FINAL CALL 属性を指定します。
- データベース・サーバー上で、ルーチン内のどのパラメーター用のストレージも割り振ってはならない。データベース・マネージャーは、 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。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と、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。
- 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、ルーチンの次の呼び出しまで不変のままであるとは限りません。 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 引き数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ルーチンに提供されることを意味します。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もちません。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。

関連概念:

- 24 ページの『ルーチン開発に関するパフォーマンスの考慮』
- 27 ページの『ルーチンのセキュリティーに関する考慮事項』
- 177 ページの『C/C++ ルーチンでの SQL データ型処理』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CALL ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE METHOD ステートメント』
- 「*アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」の『C および C++ においてサポートされている SQL データ・タイプ』
- 「*アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」の『DB2 と OLE DB の間のデータ・タイプ・マッピング』
- 「*SQL リファレンス 第 2 巻*」の『ALTER FUNCTION ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『ALTER METHOD ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『ALTER PROCEDURE ステートメント』
- 214 ページの『OLE DB でサポートされている SQL データ型』
- 206 ページの『OLE オートメーションでサポートされている SQL データ型』

データベースでのルーチンの作成

ルーチンをデータベース内に作成するには、そのルーチンを作成するための CREATE ステートメントを実行します。外部ルーチンの場合は、 CREATE ステートメントの実行によって、そのルーチンの名前とプロパティーを定義できるだけでなく、 EXTERNAL 文節を組み込んで、そのルーチンのロジックを含んだ外部言語ライブラリーのロケーションを指定することもできます。ルーチンを呼び出すには、まずデータベース内にルーチンを作成しなければなりません。外部ルーチンを正常に呼び出すには、データベース内にまず外部ルーチンを作成し、そのルーチンに関連したライブラリーを EXTERNAL 文節で指定したロケーションに配置する必要があります。

ルーチンを正しく実行するには、そのルーチンの特性に該当する文節を使用してルーチンを作成することが不可欠です。ルーチンのタイプは異なっていても、ルーチンを登録するための文節の多くは共通しています。

前提条件:

データベース内にルーチンを作成するため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
- CREATE METHOD
- CREATE TYPE
- CREATE PROCEDURE

手順:

ルーチンを作成するには、処理しているルーチンのタイプに対応した適切な文節を使用して CREATE ステートメントを発行します。そのようなステートメントには、CREATE FUNCTION、CREATE METHOD、CREATE TYPE、および CREATE PROCEDURE があります。

メソッドを作成するには、 CREATE TYPE ステートメントを実行して構造化型を作成しておく必要があります。 CREATE TYPE ステートメントには、タイプと関連付けるメソッド宣言を指定するためにオプションで使用できる METHOD 文節を組み込みます。あるいは、ALTER TYPE ステートメントを使用して、既存の構造化型のメソッドを宣言することもできます。その後、CREATE METHOD ステートメントを実行して、メソッドを正式に作成します。 CREATE METHOD ステートメントでは、メソッドのシグニチャーに関連した属性のみを扱います。

ルーチンの作成が完了したら、そのルーチン・タイプのルーチン呼び出しをサポートするどのインターフェースからでもルーチンを呼び出すことができます。ルーチンのタイプによって異なりますが、クライアント・アプリケーション、他のルーチン、トリガー、 SQL ステートメントなどからの呼び出しが可能です。

関連概念:

- 3 ページの『アプリケーション開発におけるルーチン』
- 99 ページの『外部ルーチン用のパラメーター・スタイル』
- 24 ページの『ルーチン開発に関するパフォーマンスの考慮』
- 27 ページの『ルーチンのセキュリティーに関する考慮事項』

関連タスク:

- 37 ページの『ルーチンの作成』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE TYPE (構造化) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE METHOD ステートメント』

関連サンプル:

- 『spcreate.db2 -- How to catalog the stored procedures contained in spserver.sqc (C)』
- 『spserver.db2 -- To create a set of SQL procedures 』
- 『UDFsCreate.db2 -- How to catalog the UDFs contained in UDFsqlsv.java 』

ルーチンの作成

3 つのタイプのルーチン (プロシージャー、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。たとえば、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。たとえば、結果セットはストアード・プロシージャー独特のものであり、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。

前提条件:

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。 (ルーチンのタイプ (プロシージャー、関数、メソッド) を参照。)
- 作成に使用するプログラム言語。 (サポートされているルーチン・プログラミング言語 を参照。)
- 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。 (DB2 CLI または組み込み SQL をいつ使用するか を参照。)

『セキュリティー、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。

手順:

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。

   たとえば、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 プロシージャーでは、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 プロシージャー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・プロシージャー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・プロシージャーに備わっています。 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。

関連概念:

- 「*アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」の『DB2 CLI または組み込み SQL をいつ使用するか』
- 99 ページの『外部ルーチン用のパラメーター・スタイル』
- 24 ページの『ルーチン開発に関するパフォーマンスの考慮』
- 27 ページの『ルーチンのセキュリティーに関する考慮事項』
- 170 ページの『C/C++ ルーチン』
- 189 ページの『Java ルーチン』
- 33 ページの『ルーチンの使用に関する制約事項』
- 30 ページの『ライブラリーおよびクラスの管理に関する考慮事項』
- 203 ページの『OLE オートメーション・ルーチンの設計』
- 210 ページの『OLE DB ユーザー定義表関数』
- 21 ページの『サポートされているルーチン・プログラミング言語』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE TYPE (構造化) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE METHOD ステートメント』

関連サンプル:

- 『spserver.c -- Definition of various types of stored procedures』
- 『spserver.db2 -- To create a set of SQL procedures 』
- 『spserver.sqc -- Definition of various types of stored procedures (C)』
- 『spserver.sqC -- Definition of various types of stored procedures (C++)』
- 『SpServer.java -- Provide a variety of types of stored procedures to be called from (JDBC)』
- 『SpServer.sqlj -- Provide a variety of types of stored procedures to be called from (SQLj)』

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役割、それぞれの役割の判別、各役割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
- SYSADM または DBADM 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 特権を持つことになります。定義者がパッケージ所有者でない場合は、パッケージ所有者か、SYSADM または DBADM 権限を持つユーザーが、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。

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。

ルーチン定義者の役割は、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 を持つことになります。

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。プロシージャーは、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もあります)。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、 SYSADM 権限または DBADM 権限を持つユーザーです。

たとえば、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 特権の管理に関係するステップを示します。

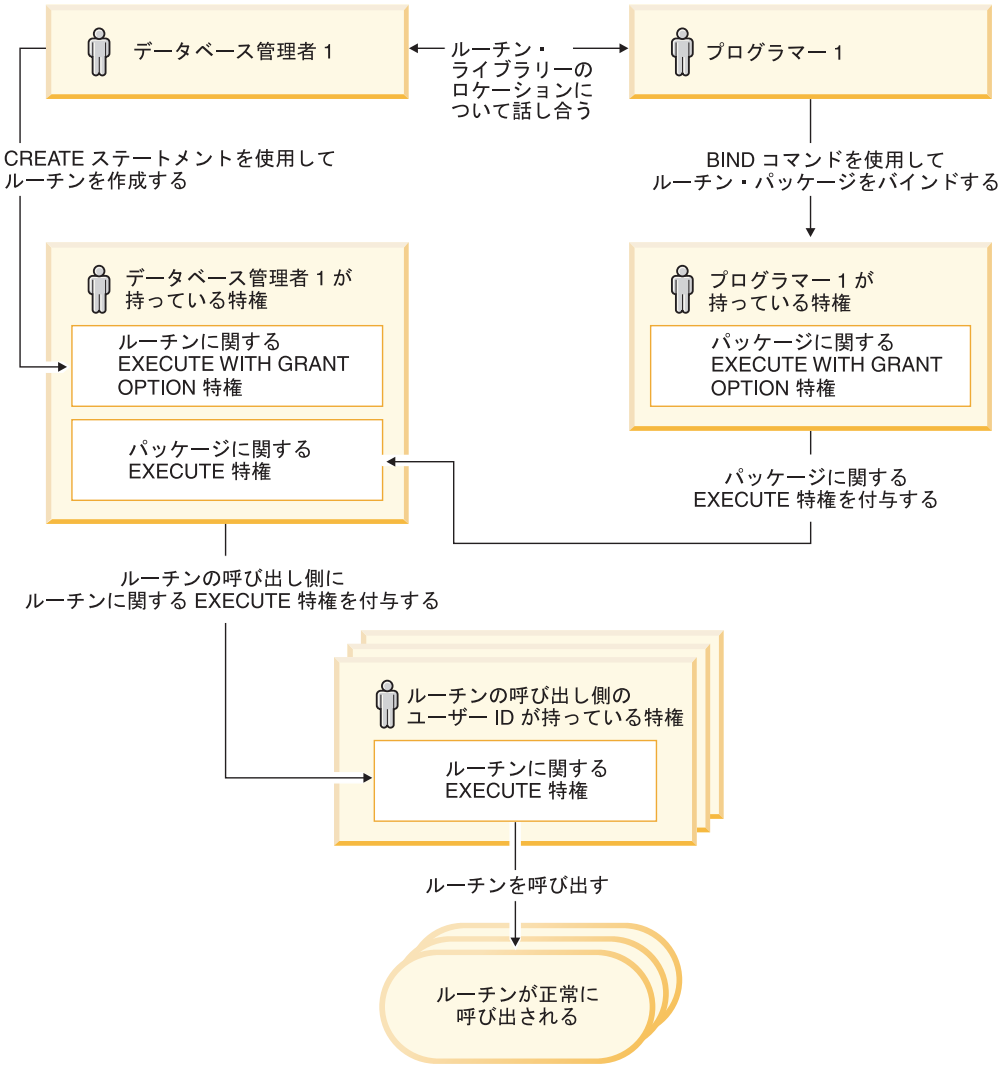


図 2. ルーチンに関する EXECUTE 特権の管理

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 文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 特権を持つことになりますが、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、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しかありません。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 特権を定義者がも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)。

関連概念:

- 「*管理ガイド: インプリメンテーション*」の『特権、権限レベル、およびデータベース権限』
- 119 ページの『動的 SQL における DYNAMICRULES BIND オプションの影響』
- 「*管理ガイド: インプリメンテーション*」の『ルーチン特権』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE ステートメント』
- 「*コマンド・リファレンス*」の『BIND コマンド』

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。

手順:

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き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き数のデータ型は、 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 用のルーチン引き数が存在する。
  - 外部ルーチンの場合、CREATE ステートメント内の EXTERNAL NAME 文節の値は、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・ポイント名に適用する。タイプ修飾名を EXTERNAL NAME 文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 プロシージャーの場合にはそれは、 `sqllib/function` (UNIX) または `sqllib¥function` (Windows) となります。 unfenced プロシージャーの場合にはそれは、 `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、 SYSADM、DBADM 権限を持つユーザーか、ルーチンの定義者です。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ーと Java プロシージャー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。

  プロシージャー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 文節を使用して、テストするプロシージャーを登録する必要があります。既存のプロシージャーでデータを 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、プロシージャーをいったんドロップしてから、 MODIFIES SQL DATA 文節を使用して (または使用しないで) 再登録しなければなりません。プロシージャー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。

関連概念:

- 39 ページの『SQL の入ったルーチンの許可およびバインド』
- 27 ページの『ルーチンのセキュリティーに関する考慮事項』

関連タスク:

- 198 ページの『Java ストアード・プロシージャーのデバッグ』
- 82 ページの『SQL プロシージャーからのエラー・メッセージの戻り』

関連資料:

- 「*SQL リファレンス 第 1 巻*」の『ID』
- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE TYPE (構造化) ステートメント』
- 「*コマンド・リファレンス*」の『BIND コマンド』
- 「*コマンド・リファレンス*」の『PRECOMPILE コマンド』
- 「*SQL リファレンス 第 2 巻*」の『CREATE METHOD ステートメント』

- 214 ページの『OLE DB でサポートされている SQL データ型』
- 101 ページの『C/C++、OLE、COBOL で書かれたルーチンに引き数を渡すときの構文』
- 206 ページの『OLE オートメーションでサポートされている SQL データ型』
- 175 ページの『C/C++ でサポートされている SQL データ型』

プロシージャー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

データベースの保全性を保つには、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。たとえば、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・プロシージャーから発行される任意の SQL ステートメントのことです。プロシージャー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。
- プロシージャーが、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。

たとえば、アプリケーションがストアード・プロシージャーを呼び出す場合、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・プロシージャーが 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・プロシージャーは、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。

関連概念:

• 3 ページの『アプリケーション開発におけるルーチン』

- 116 ページの『外部ルーチンでの SQL』

プロシージャーの機能

ストアード・プロシージャーには、呼び出し元のアプリケーションとルーチンとの間でデータをやり取りするための特別な機能があります。以下の項では、プロシージャーのパラメーター・モード、プロシージャーから結果セットを戻す機能、メインルーチンまたはサブルーチンのスタイルでパラメーターを受け入れるオプションについて説明しています。

プロシージャーのパラメーター・モード

クライアント・アプリケーションと呼び出しルーチンは、パラメーターと結果セットを介してプロシージャーと情報を交換します。ルーチンのパラメーターは、具体的なデータ型をもつものとして定義されます。他のルーチンとは違ってプロシージャーのパラメーターは、データの送信先によっても定義されます (パラメーター・モード)。

プロシージャーのパラメーター・タイプには、以下の 3 種類があります。

- IN パラメーター:プロシージャーに渡されるデータ。
- OUT パラメーター:プロシージャーから戻されるデータ。
- INOUT パラメーター:プロシージャーに渡されて、そのプロシージャーの実行中に、プロシージャーから戻される予定のデータに置き換えられるデータ。

パラメーターのモードとそのデータ型は、プロシージャーが CREATE PROCEDURE ステートメントに登録されるときに定義されます。

関連概念:

- 48 ページの『プロシージャーの結果セット』

関連タスク:

- 36 ページの『データベースでのルーチンの作成』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE ステートメント』

プロシージャーの結果セット

以下の項では、結果セットを戻すプロシージャーの機能を取り上げ、各種インターフェースを使用して結果セットを戻したり受け取ったりする方法を説明しています。

プロシージャーの結果セット

プロシージャーは、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と、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

ストアード・プロシージャーは、カーソルを介して呼び出し側に結果セットを渡します。プロシージャーの本体には、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ければなりません。プロシージャー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。プロシージャーを終了するときは、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 文節内に宛先を指定することを強くお勧めします (SQL プロシージャー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。中間のネストされたルーチンをすべてう回して、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ーを登録するときは、 DYNAMIC RESULT SETS 文節を使用して、そのプロシージャー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。プロシージャー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 ストアード・プロシージャーの場合、 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® UDB 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ーを宣言して、そのロケーターを、結果セットの返送元になる予定のプロシージャーに関連付けなければなりません。次に、結果セット・ロケーター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ーを呼び出す場合は、どの結果セットにもアクセスできません。

注: プロシージャー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・プロシージャーから 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。プロシージャーから 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。

関連概念:

- 13 ページの『プロシージャー』
- 「*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 アプリケーションのカーソル』
- 「*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 アプリケーションにおける結果セットの用語』
- 「*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 アプリケーションでの結果セットの配列への取り出し』

関連タスク:

- 「*アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」の『静的 SQL プログラムにおけるカーソルの宣言と使用』
- 「*アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」の『動的 SQL プログラムにおけるカーソルの宣言と使用』
- 51 ページの『SQL および組み込み SQL プロシージャーからの結果セットの戻り』
- 54 ページの『SQL ルーチンでのプロシージャーの結果セットの受け取り』
- 57 ページの『JDBC アプリケーションおよびルーチンでのプロシージャーの結果セットの受け取り』
- 53 ページの『JDBC プロシージャーからの結果セットの戻り』
- 55 ページの『SQLJ アプリケーションおよびルーチンでのプロシージャーの結果セットの受け取り』
- 52 ページの『SQLJ プロシージャーからの結果セットの戻り』
- 130 ページの『CLR プロシージャーからの結果セットの戻り』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『COMMIT ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『DESCRIBE ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『PREPARE ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『ROLLBACK ステートメント』
- 「*SQL リファレンス 第 1 巻*」の『SYSCAT.ROUTINES カタログ・ビュー』

SQL および組み込み SQL プロシージャー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プロシージャーを開発することができます。 SQL および組み込み SQL プロシージャーでは、結果セットの戻りは DECLARE CURSOR ステートメントで処理します。

手順:

SQL または組み込み SQL プロシージャーから結果セットを戻すには、次のようにします。

1. DECLARE CURSOR ステートメントを使用してカーソルを宣言します。カーソル宣言には、結果セットを構成する一連の行を生成する SELECT ステートメントが組み込まれます。カーソル宣言においては、 WITH RETURN TO 文節を使用して結果セットの宛先を指定することを強くお勧めします (これは、 SQL プロシージャーでは必須です)。
   - プロシージャーの呼び出し側に結果セットを戻すには、その呼び出し側がクライアント・アプリケーションまたは別のルーチンのどちらであっても、 WITH RETURN TO CALLER 文節を使用します。

     次の例では、 SQL プロシージャー “CALLER_SET” は、 WITH RETURN TO CALLER 文節を使用して結果セットを CALLER_SET の呼び出し側に戻します。

     ```
     CREATE PROCEDURE CALLER_SET()
       DYNAMIC RESULT SETS 1
       LANGUAGE SQL
       BEGIN
          DECLARE clientcur CURSOR WITH RETURN TO CALLER
              FOR SELECT name, dept, job
              FROM staff
              WHERE salary > 15000;
          OPEN clientcur;
       END
     ```
   - プロシージャーから発信元のクライアント・アプリケーションに結果セットを戻すには、 WITH RETURN TO CLIENT 文節を使用します。結果セット上に WITH RETURN TO CLIENT を指定すると、ネストされたプロシージャーはいずれも結果セットにアクセスできなくなります。

     次の例では、 SQL プロシージャー “CLIENT_SET” はネストされたルーチンとして呼び出された場合でも、 “CLIENT_SET” は DECLARE CURSOR ステートメント内で WITH RETURN TO CLIENT 文節を使用して、クライアント・アプリケーションに結果セットを戻します。

     ```
     CREATE PROCEDURE CLIENT_SET()
       DYNAMIC RESULT SETS 1
       LANGUAGE SQL
       BEGIN
          DECLARE clientcur CURSOR WITH RETURN TO CLIENT
              FOR SELECT name, dept, job
              FROM staff
              WHERE salary > 20000;
          OPEN clientcur;
       END
     ```
2. OPEN ステートメントを使用してカーソルをオープンします。プロシージャー内でカーソルのオープンが完了したら、そこから行を取り出すことができます。た

だし、アプリケーションまたは呼び出し元のルーチンに戻された結果セットには、取り出されなかった行だけが入っています。

3. カーソルをクローズせずにプロシージャーを終了します。

プロシージ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。

関連概念:

- 83 ページの『SQL プロシージャーの条件ハンドラー』
- 87 ページの『SQL プロシージャーでの SQLCODE および SQLSTATE 変数』
- 48 ページの『プロシージャーの結果セット』

関連タスク:

- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『SQL プロシージャーの作成』
- 230 ページの『コマンド行プロセッサー (CLP) からのプロシージャーの呼び出し』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『クライアント・アプリケーションによる SQL プロシージャーの呼び出し』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『Windows でのクライアント・アプリケーションによる SQL プロシージャーの呼び出し』
- 54 ページの『SQL ルーチンでのプロシージャーの結果セットの受け取り』
- 57 ページの『JDBC アプリケーションおよびルーチンでのプロシージャーの結果セットの受け取り』
- 55 ページの『SQLJ アプリケーションおよびルーチンでのプロシージャーの結果セットの受け取り』

関連資料:

- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『SQL プロシージャーのサンプル』

関連サンプル:

- 『spserver.sqc -- Definition of various types of stored procedures (C)』
- 『spserver.sqC -- Definition of various types of stored procedures (C++)』

SQLJ プロシージャー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 プロシージャーを開発することができます。 SQLJ プロシージャーでは、結果セットの戻りは ResultSet オブジェクトで処理します。

手順:

SQLJ プロシージャー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。たとえば、次のようになります。

```
#sql iterator SpServerEmployees(String, String, double);
```

2. 戻される各結果セットごとに、プロシージャーの宣言内に ResultSet[] タイプのパラメーターを組み込みます。たとえば、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。たとえば、次のようになります。

```
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 文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();
```

プロシージ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。

関連概念:

- 48 ページの『プロシージャーの結果セット』

関連タスク:

- 54 ページの『SQL ルーチンでのプロシージャーの結果セットの受け取り』
- 57 ページの『JDBC アプリケーションおよびルーチンでのプロシージャーの結果セットの受け取り』
- 55 ページの『SQLJ アプリケーションおよびルーチンでのプロシージャーの結果セットの受け取り』

関連サンプル:

- 『SpServer.sqlj -- Provide a variety of types of stored procedures to be called from (SQLj)』

JDBC プロシージャー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 プロシージャーを開発することができます。 JDBC プロシージャーでは、結果セットの戻りは ResultSet オブジェクトで処理します。

手順:

JDBC プロシージャー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、プロシージャーの宣言内に ResultSet[] タイプのパラメーターを組み込みます。たとえば、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 (パラメーター・マーカーは ? で示されます) 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。

```
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. プロシージャーの本体を終了します。

ストアード・プロシージ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。

関連概念:

- 48 ページの『プロシージャーの結果セット』

関連タスク:

- 54 ページの『SQL ルーチンでのプロシージャーの結果セットの受け取り』
- 57 ページの『JDBC アプリケーションおよびルーチンでのプロシージャーの結果セットの受け取り』
- 55 ページの『SQLJ アプリケーションおよびルーチンでのプロシージャーの結果セットの受け取り』

関連サンプル:

- 『SpServer.java -- Provide a variety of types of stored procedures to be called from (JDBC)』

SQL ルーチンでのプロシージャーの結果セットの受け取り

SQL を本体とするルーチン内から呼び出したプロシージャーから、結果セットを受け取ることができます。

前提条件:

呼び出したプロシージャーからいくつの結果セットが戻されるかを知っている必要があります。呼び出し元のルーチンが受け取る各結果セットごとに、結果セットを宣言しなければなりません。

手順:

SQL を本体とするルーチンからのプロシージャー結果セットを受け入れるには、次のようにします。

1. プロシージャーから戻される各結果セットごとに結果セット・ロケーターを宣言します。たとえば、次のようにします。

```
DECLARE result1 RESULT_SET_LOCATOR VARYING;
DECLARE result2 RESULT_SET_LOCATOR VARYING;
DECLARE result3 RESULT_SET_LOCATOR VARYING;
```

2. プロシージャーを呼び出します。たとえば、次のようにします。

```
CALL targetProcedure();
```

3. 結果セット・ロケーター変数 (上記で定義済み) を呼び出し元のプロシージャーに関連付けます。たとえば、次のようにします。

```
ASSOCIATE RESULT SET LOCATORS(result1, result2, result3)
   WITH PROCEDURE targetProcedure;
```

4. 呼び出し先のプロシージャーから渡される結果セット・カーソルをその結果セット・ロケーターに割り振ります。たとえば、次のようにします。

```
ALLOCATE rsCur CURSOR FOR RESULT SET result1;
```

5. 結果セットから行を取り出します。たとえば、次のようにします。

```
FETCH rsCur INTO ...
```

関連概念:

- 48 ページの『プロシージャーの結果セット』

関連タスク:

- 51 ページの『SQL および組み込み SQL プロシージャーからの結果セットの戻り』
- 53 ページの『JDBC プロシージャーからの結果セットの戻り』
- 52 ページの『SQLJ プロシージャーからの結果セットの戻り』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CALL ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『DECLARE CURSOR ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『FETCH ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『ALLOCATE CURSOR ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『ASSOCIATE LOCATORS ステートメント』

SQLJ アプリケーションおよびルーチンでのプロシージャー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ーから、結果セットを受け取ることができます。

手順:

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ー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ーを呼び出します。以下の例では、GET_HIGH_SALARIES という名前のプロシージャーが呼び出して、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、プロシージャー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。プロシージャー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。たとえば、次のようになります。

```
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();
}
```

関連概念:

- 48 ページの『プロシージャーの結果セット』

関連タスク:

- 51 ページの『SQL および組み込み SQL プロシージャーからの結果セットの戻り』
- 53 ページの『JDBC プロシージャーからの結果セットの戻り』
- 52 ページの『SQLJ プロシージャーからの結果セットの戻り』

関連サンプル:

- 『SpClient.sqlj -- Call a variety of types of stored procedures from SpServer.sqlj (SQLj)』

JDBC アプリケーションおよびルーチンでのプロシージ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ジャーから、結果セットを受け取ることができます。

手順:

JDBC ルーチンまたはアプリケーションからのプロシージ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ジャーを呼び出す 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ジ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・マーカーの値に対して、入力変数 (inSalaryThreshold という名前。これは、プロシージャーに渡される数値) を割り当てます。 (パラメーター・マーカーは ? で示されます.)

```
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ジ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、プロシージ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。たとえば、次のようになります。

```
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();
```

関連概念:

- 48 ページの『プロシージャーの結果セット』

関連タスク:

- 51 ページの『SQL および組み込み SQL プロシージャーからの結果セットの戻り』
- 53 ページの『JDBC プロシージャーからの結果セットの戻り』
- 52 ページの『SQLJ プロシージャーからの結果セットの戻り』

関連サンプル:

- 『SpClient.java -- Call a variety of types of stored procedures from SpServer.java (JDBC)』

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き数を受け入れます。パラメーターをポインターとして受け渡します。たとえば、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き数を受け入れません。 IN パラメーターは単純な引き数として渡します。 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 関数のような引き数を受け入れるように C プロシージャーを作成するには、 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 のプロシージャーを作成する場合は、以下の仕様に準拠していなければなりません。

- プロシージャーは以下の 2 つの引き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。たとえば、*argc*。
  - パラメーターを指すポインターの配列。たとえば、 *char **argv*。
- プロシージャー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 プロシージャーでは、 DB2® は *argv* 配列の最初のエレメントの値 (*argv[0]*) をプロシージャーの名前に設定します。 *argv* 配列の残りのエレメントは、プロシージャーの 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。たとえば、次のような組み込み型の C プロシージャーは、 *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ーのコード例では、 *argv[1]* の値を CHAR(8) のホスト変数 *injob* にコピーしてから、 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 */
```

関連概念:

- 13 ページの『プロシージャー』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE ステートメント』

関連サンプル:

- 『spcreate.db2 -- How to catalog the stored procedures contained in spserver.sqc (C)』
- 『spserver.sqc -- Definition of various types of stored procedures (C)』

UDF とメソッドの機能

ストアード・プロシージャーとは異なり、 UDF とメソッドは、 SQL ステートメント内から呼び出します。ストアード・プロシージャーの場合は、 1 回呼び出すだけですが、関数やメソッドの場合は、 SQL ステートメント内の 1 つの参照で複数回呼び出すことができます。このようなインプリメンテーションの違いに対応するには、特別な機能が必要です。以下の項では、複数の呼び出しにまたがって状態情報を保存するために使用できるスクラッチパッドと、 FINAL CALL オプションで UDF とメソッドを登録するための処理モデルについて説明しています。

UDF 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。たとえば、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。

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' を戻し、 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 引き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、最初の引き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 番目の引き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、入力される引き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複数パーティション・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になります。複数パーティション・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になります。

関数が正しく実行されるかどうかが、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があり、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。たとえば上記の *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き数を使用して 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。

関連概念:

- 63 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』
- 64 ページの『メソッドおよびスカラー関数の処理モデル』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE TYPE (構造化) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE METHOD ステートメント』

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。

スクラッチパッドは、以下のような構造を持った LOB の形式で渡されます。

```
struct lob
{
   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 LOB 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。たとえば、次のようにします。

```
struct spadlob
{
  sqlint32 lob_length;
  sqlint32 int_var;
  sqlint64 bigint_var;
};
void SQL_API_FN routine( ..., struct spadlob* scratchpad, ... )
{
  /* Use scratchpad */
}
```

2. スクラッチパッド LOB 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 lob* lob_spad, ... )
{
  struct spaddata* scratchpad = (struct spaddata*)lob_spad-->data;
  /* Use scratchpad */
}
```

アプリケーションは、スクラッチパッド LOB 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。

関連概念:

- 60 ページの『UDF とメソッドのスクラッチパッド』
- 15 ページの『ユーザー定義のスカラー関数』
- 17 ページの『ユーザー定義のスカラー関数』

関連タスク:

- 222 ページの『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』

メソッドおよびスカラー関数の処理モデル

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。引き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いため、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、引き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 呼び出しでは引き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。たとえば、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。

関連概念:

- 15 ページの『ユーザー定義のスカラー関数』
- 18 ページの『メソッド』

ユーザー定義表関数

スカラー値を戻す UDF 以外に、表を戻す UDF も開発できます。以下の項では、ユーザー定義表関数を取り上げ、 FINAL CALL オプションで表 UDF を登録するための処理モデルについて説明しています。

ユーザー定義表関数

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 文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、スカラー関数で使用されるのと同じ*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 文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* 引き数によって、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、DB2 オプティマイザーは結果の適切なサイズが分かり、関数が参照されるときによりよい決定を下せます。

  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うまで、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 文節で使用すると、無限処理ループが生じることがあります。

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION ステートメント』
- 101 ページの『C/C++、OLE、COBOL で書かれたルーチンに引き数を渡すときの構文』

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。引き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。引き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になります。引き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、この引き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。表関数の次回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。どの引き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。たとえば、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 (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。

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。スキャンの終了時点ごとにさらに別のスキャ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ャ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ャンを対象とすることが主な原因ですが、スキャ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。

関連概念:

- 68 ページの『Java の表関数実行モデル』
- 17 ページの『ユーザー定義のスカラー関数』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION (OLE DB 外部表) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION (SQL スカラー、表、または行) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION (外部表) ステートメント』

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、通常の表関数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。

スキャンの時点	NO FINAL CALL LANGUAGE JAVA SCRATCHPAD	FINAL CALL LANGUAGE JAVA SCRATCHPAD
表関数に対する最初の OPEN の前	呼び出しなし。	• 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。

スキャンの時点	NO FINAL CALL LANGUAGE JAVA SCRATCHPAD	FINAL CALL LANGUAGE JAVA SCRATCHPAD
表関数に対する各 OPEN 時	• 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 データのスキャンをオープンする。	• 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 データのスキャンをオープンする。 (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	• 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。	• 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。
表関数に対する CLOSE 時	• 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 スキャンをクローズし、Web サーバーから切断する。 `close()` は必要とされない。	• CLOSE 呼び出しで UDF メソッドが呼び出される。 • メソッドはスキャンの最上部に位置変更するか、スキャンをクローズする。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	呼び出しなし。	• 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:

1. 「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 文節で識別されるメソッドです。
2. 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。

関連概念:

- 363 ページの『DB2GENERAL ルーチン』
- 189 ページの『Java ルーチン』
- 66 ページの『表関数の処理モデル』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION (外部表) ステートメント』

第 3 章 SQL ルーチン

SQL ルーチンを作成するには、そのルーチン・タイプに該当する CREATE ステートメントを実行します。そのステートメントにはルーチン本体も指定します。ルーチン本体は、SQL ルーチンの場合、 SQL ステートメントまたは SQL PL ステートメントだけで記述する必要があります。 SQL プロシージャーの作成、デバッグ、実行には、 IBM DB2 デベロップメント・センターを使用すると便利です。DB2 コマンド行プロセッサーを使用して、 SQL プロシージャーを作成することもできます。

DB2 での SQL Procedural Language (SQL PL)

プロシージャーや関数で SQL PL を使用する方法について説明する前に、 DB2 のプロシージャー型 SQL に関連した基本的な用語や概念を確認しておくのは重要です。スカラー変数、IF ステートメント、 WHILE ループなどのプロシージャー型 SQL の構造体は、 DB2 バージョン 7 のリリース時に DB2 に導入されました。これらの構造体が発展して、現在 SQL Procedural Langauge (SQL PL) と呼んでいる SQL ステートメントのセットになりました。

SQL PL:

SQL PL とは、実際にはプロシージャー型の構造体を提供する SQL のサブセットであり、従来の SQL ステートメントの周辺のロジックをインプリメントするときに使用できます。 SQL PL は、構文が単純な高水準のプログラム言語です。基本的なプログラミング制御ステートメントとしては、
IF、ELSE、WHILE、FOR、ITERATE、GOTO の各ステートメントや他のステートメントがあります。

SQL PL および SQL プロシージャー:

SQL PL プロシージャーには、パラメーター、変数、代入ステートメント、SQL PL 制御ステートメント、コンパウンド SQL ステートメントを組み込めます。 SQL PL プロシージャーは、条件処理とエラー処理の強力なメカニズム、呼び出しのネスト処理と再帰処理、呼び出し側またはクライアント・アプリケーションに複数の結

果セットを戻す機能などもサポートしています。 SQL PL プロシージャーでサポートされている言語エレメントの完全セットについては、 SQL リファレンスの CREATE PROCEDURE (SQL) ステートメントの項目を参照してください。

インライン SQL PL および SQL 関数、トリガー、動的コンパウンド・ステートメント:

DB2 バージョン 7.2 以降、 SQL 関数とトリガーの本体で SQL PL のサブセットをサポートできるようになりました。この SQL PL のサブセットのことをインライン SQL PL といいます。このインラインという語は、インライン SQL PL とフルセットの SQL PL 言語の重要な違いを強調しています。 SQL PL プロシージャーは、個々の SQL 照会をパッケージ内の各セクションに静的にコンパイルすることによってインプリメントするのに対し、インライン SQL PL 関数は、その名が示すとおり、関数の本体を、関数を使用する照会の中にインライン化することによってインプリメントします。この違いによってパフォーマンスに関する考慮事項がいくつか発生するので、プロシージャー・ロジックをプロシージャー内の SQL PL でインプリメントするのか、インライン SQL PL によってインプリメントするのかを計画するときには、それらの考慮事項について検討する必要があります。

動的コンパウンド・ステートメントとは、複数の SQL ステートメントを実際に 1 つの小さなロジックのアトミック・ブロックにまとめるためのステートメントであり、この動的コンパウンド・ステートメントの中では、変数や条件処理エレメントを宣言できます。この種のステートメントは、 DB2 によって 1 つの SQL ステートメントとしてコンパイルされます。また、SQL PL のエレメントを組み込むことも可能です。動的コンパウンド・ステートメントには、インライン SQL PL と呼ばれる SQL PL のサブセットと、少数の基本的な SQL ステートメントだけを組み込めます。かなりのデータ・フローを伴うものの、最小限の制御フローで小さなロジック作業単位を実行する小さなスクリプトを作成する場合は、動的コンパウンド・ステートメントが便利です。一方、パラメーターや、結果セットの受け渡しや、さらに高度な他のプロシージャー型のエレメントを必要とする複雑なロジックの場合は、 SQL のプロシージャーと関数のほうが適していると言えます。

SQL PL プロシージャー、SQL 関数、トリガーなどの動的コンパウンド・ステートメントでサポートされている SQL PL ステートメントの完全リストについては、SQL リファレンスの各ルーチン・タイプの CREATE ステートメントの項目を参照してください。

関連概念:

- 3 ページの『アプリケーション開発におけるルーチン』
- 10 ページの『ユーザー定義ルーチン』
- 73 ページの『SQL ルーチンの CREATE ステートメント』

関連タスク:

- 77 ページの『コマンド行からの SQL プロシージャーの作成』

SQL ルーチンの CREATE ステートメント

SQL ルーチンを作成するには、そのルーチン・タイプに該当する CREATE ステートメントを実行します。そのステートメントにはルーチン本体も指定します。ルーチン本体は、SQL ルーチンの場合、 SQL ステートメントまたは SQL PL ステートメントだけで記述する必要があります。 SQL プロシージャーの作成、デバッグ、実行には、 IBM DB2 デベロップメント・センターを使用すると便利です。 SQL のプロシージャー、関数、メソッドは、 DB2 コマンド行プロセッサーで作成することも可能です。

SQL のプロシージャー、関数、メソッドには、それぞれの CREATE ステートメントがあります。ステートメントの構文はそれぞれ違いますが、共通のエレメントもいくつかあります。各ステートメントでは、ルーチン名を指定しなければなりません。パラメーターが必要であれば、パラメーターも指定しなければなりません。戻りタイプの指定も必要です。ルーチンに組み込むロジックに関する情報を DB2 に渡すための追加のキーワードも指定できます。 DB2 は、そのルーチン・プロトタイプと追加のキーワードを使用して、呼び出し時にルーチンを識別し、必要な機能サポートと最適なパフォーマンスでルーチンを実行します。

DB2 デベロップメント・センターまたはコマンド行プロセッサーで SQL プロシージャーを作成するための具体的な情報や、関数やメソッドを作成するための具体的な情報については、以下の関連トピックを参照してください。

SQL ルーチンの SQL アクセス・レベル

SQL アクセス・レベルとは、ルーチンで使用する SQL アクセス・レベルを示すために、ルーチンの CREATE ステートメントで指定する文節です。この文節を使用して、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。

SQL プロシージャーは、デフォルトでは、MODIFIES SQL DATA という SQL アクセス・レベルで作成されます。プロシージャー内の SQL ステートメントによってどのデータも変更しない場合は、これをより低いアクセス・レベル (READS SQL DATA や CONTAINS SQL など) に変更できます。この変更は、プロシージャーの CREATE ステートメント内に、該当する SQL アクセス・レベルの文節を指定することによって行えます。ルーチンが最高のパフォーマンスを発揮するのは、CREATE ステートメントに最も限定的な (有効な) SQL アクセス文節を指定した場合です。

すべての UDF (表関数、スカラー関数、メソッド) は、デフォルトでは、READS SQL DATA という SQL アクセス・レベルで作成されます。 UDF でデータを読み取らない場合は、 SQL を本体として持つ UDF の SQL アクセス・レベルを CONTAINS SQL に定義または変更できます。 SQL を本体として持つ表関数の場合は、表を変更する SQL ステートメントを本体で使用できるので、 SQL アクセス・レベルを MODIFIES SQL DATA に定義または変更できます。

関連概念:

• 93 ページの『SQL データを変更する SQL 表関数』

関連資料:

- 「*アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」の『サポートされる SQL ステートメント』

SQL ルーチンでの動的 SQL

外部ルーチンと同様、 SQL ルーチンは動的 SQL ステートメントを発行できます。動的 SQL ステートメントにパラメーター・マーカーが含まれておらず、それを実行するのが一度のみである場合には、EXECUTE IMMEDIATE ステートメントを使用します。

動的 SQL ステートメントにパラメーター・マーカーが含まれている場合には、PREPARE および EXECUTE ステートメントを使用する必要があります。動的 SQL ステートメントを複数回実行する場合には、単一の PREPARE ステートメントを発行してから EXECUTE ステートメントを複数回発行するほうが、 EXECUTE IMMEDIATE ステートメントをその度に発行するよりも効率的です。

SQL ルーチンで動的 SQL を発行するために PREPARE および EXECUTE ステートメントを使用するには、 SQL ルーチン本体で以下のようなステートメントを含める必要があります。

1. DECLARE ステートメントを使用して、動的 SQL ステートメントを入れるために十分な大きさの VARCHAR タイプの変数を宣言します。
2. SET ステートメントを使用して、ステートメント・ストリングを変数に割り当てます。変数はステートメント・ストリングに直接含めることはできません。その代わりに、疑問符 ('?') シンボルを、ステートメントで使用される変数のパラメーター・マーカーとして使用する必要があります。
3. PREPARE ステートメントを使用して、ステートメント・ストリングから準備済みステートメントを作成します。
4. EXECUTE ステートメントを使用して準備済みステートメントを実行します。ステートメント・ストリングにパラメーター・マーカーが組み込まれている場合、USING 文節を使用して変数の値に置き換えます。出力パラメーター・マーカーがステートメントに組み込まれている場合は、 INTO 文節を使用して、出力を受け取る変数を指定します。

注: SQL ルーチンの PREPARE ステートメントで定義されているステートメント名は、範囲付き変数として扱われます。 SQL ルーチンがそのステートメント名を定義した有効範囲を出ると、 DB2® はステートメント名をアクセスできなくなります。コンパウンド・ステートメント内では、同一のステートメント名を使用する PREPARE ステートメントを 2 つ発行することはできません。

動的 SQL ステートメントを含む SQL プロシージャーを以下の例に示します。

この SQL プロシージャーは、部門番号 (*deptNumber*) を入力パラメーターとして受け取ります。 SQL プロシージャー内では、 3 つのステートメント・ストリングが作成、準備、および実行されます。最初のステートメント・ストリングでは、DROP ステートメントが実行されて、作成される表が存在していないことが確認されます。この表には、 DEPT_*deptno* _T という名前が付けられます。ここで、*deptno* は入力パラメーター *deptNumber* の値です。 CONTINUE HANDLER は、

DROP ステートメントを実行する際に表が存在しないときに、 DB2 によって戻される SQLSTATE 42704 (“未定義のオブジェクト名です”) が検出されても SQL プロシージャーが継続するようにします。 2 番目のステートメント・ストリングは CREATE ステートメントを発行して DEPT_*deptno* _T を作成します。 3 番目のステートメント・ストリングは、部署 *deptno* の社員行を *deptno* _T に挿入します。3 番目のステートメント・ストリングには、 *deptNumber* を表すパラメーター・マーカーが入っています。準備済みステートメントが実行されると、パラメーター・マーカーが *deptNumber* パラメーターに置換されます。

```
CREATE PROCEDURE create_dept_table
(IN deptNumber VARCHAR(3), OUT table_name VARCHAR(30))
LANGUAGE SQL
  BEGIN
    DECLARE stmt VARCHAR(1000);

    -- continue if sqlstate 42704 ('undefined object name')
    DECLARE CONTINUE HANDLER FOR SQLSTATE '42704'
      SET stmt = '';
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
      SET table_name = 'PROCEDURE_FAILED';

    SET table_name = 'DEPT_'||deptNumber||'_T';
    SET stmt = 'DROP TABLE '||table_name;
    PREPARE s1 FROM stmt;
    EXECUTE s1;
    SET stmt = 'CREATE TABLE '||table_name||
     '( empno CHAR(6) NOT NULL, '||
     'firstnme VARCHAR(12) NOT NULL, '||
     'midinit CHAR(1) NOT NULL, '||
     'lastname VARCHAR(15) NOT NULL, '||
     'salary DECIMAL(9,2))';
    PREPARE s2 FROM STMT;
    EXECUTE s2;
    SET stmt = 'INSERT INTO '||table_name || ' ' ||
     'SELECT empno, firstnme, midinit, lastname, salary '||
     'FROM employee '||
     'WHERE workdept = ?';
   PREPARE s3 FROM stmt;
   EXECUTE s3 USING deptNumber;
END
```

関連概念:

- 「アプリケーション開発ガイド クライアント・アプリケーションのプログラミング」の『動的 SQL サポート・ステートメント』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『EXECUTE ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『PREPARE ステートメント』

SQL/ SQL PL プロシージャー

SQL プロシージャーの設計上の考慮事項

SQL プロシージャーを設計するときには、以下のような考慮事項があります。

本当にプロシージャーが必要なのか

各ルーチン・タイプを比較している箇所を参照して、プロシージャーと他のルーチン・タイプの使用方法の違いを確認し、各タイプの機能と制限を比較してください。

実行する必要のある SQL ステートメントが SQL プロシージャー内でサポートされているか

この点を確認するために、「SQL リファレンス」を参照してください。必要なステートメントが SQL プロシージャー内でサポートされていない場合は、必要なロジックを外部プロシージャーや SQL 関数やアプリケーションの中でインプリメントすることを検討してください。

単純な動的コンパウンド・ステートメントでニーズに対応できないか

非常に小さく単純なプロシージャー・ロジックの場合は、コンパウンド SQL ステートメントで十分に対応できることがあります。コンパウンド・ステートメントを使用すれば、複数のステートメントを 1 つの実行単位にまとめることができます。コンパウンド・ステートメントには、 SQL PL 言語の一部のエレメントを組み込むことも可能です。パラメーターや大量のプロシージャー・ロジックを必要とせず、主にデータ・フローのための最小限のプロシージャー・ロジックだけを必要とするような場合には、コンパウンド・ステートメントが最適です。コンパウンド・ステートメントの詳細については、以下の資料を参照してください。

- 71 ページの『DB2 での SQL Procedural Language (SQL PL)』
- 「動的コンパウンド SQL ステートメント」

SQL プロシージャーよりも SQL 関数のほうが適していないか

プロシージャーに組み込む SQL を複数の SQL ステートメントではなく 1 つの式として記述できる場合は、ロジックを 1 つの関数としてインプリメントするほうが望ましいと言えます。 SQL PL ステートメントと SQL ステートメントを組み合わせた場合よりも、 SQL 式のほうが効率的であり、パフォーマンスが優れているからです。たとえば、CASE 式は、 IF ELSE ステートメントや CASE ステートメントに他の SQL ステートメントを組み込む場合よりも、パフォーマンスが優れています。 SQL 関数に書き換えたほうが効率的な SQL プロシージャーの例については、以下の資料を参照してください。

- 87 ページの『SQL プロシージャーのパフォーマンスの改善』

* このプロシージャーを OLTP アクティビティーに使用するか

* 作成する SQL プロシージャーを OLTP アプリケーションで使用する場合は、アプリケーションのパフォーマンスを最大化するために役立つ他の DB2 機能について調べてください。調べるべき他の DB2 機能としては、以下のような機能があります。

  - グローバル一時表: 中間結果を保管するときに便利であり、基本表よりもアクセスが高速です。
  - 表内の生成列: 行の列値を自動生成するときに使用します。
  - RUNSTATS コマンド: 照会のパフォーマンスを改善するために表の統計を収集するときに使用します。
  - 87 ページの『SQL プロシージャーのパフォーマンスの改善』

SQL PL プロシージャーのパフォーマンスに関する考慮事項

パフォーマンスは、ほとんどの場合に重要な考慮事項になります。複雑な数学アルゴリズムなどの込み入ったロジックを実行する SQL プロシージャーをインプリメントしようとしている場合、そのプロシージャーが大量の SQL PL を必要とし、データベースの照会や変更をほとんど必要としないのであれば、 SQL プロシージャーの代わりに外部プロシージャーをインプリメントすることを検討してください。 SQL PL プロシージャーをインプリメントすることにした場合は、パフォーマンスの優れた SQL PL プロシージャーを作成するためのヒントを確認するために、以下の資料を参照してください。

- 87 ページの『SQL プロシージャーのパフォーマンスの改善』

関連概念:

- 3 ページの『アプリケーション開発におけるルーチン』
- 5 ページの『ルーチンのタイプ (プロシージャー、関数、メソッド)』
- 73 ページの『SQL ルーチンの SQL アクセス・レベル』
- 71 ページの『DB2 での SQL Procedural Language (SQL PL)』
- 81 ページの『SQL プロシージャー内のコンパウンド・ブロックと変数の有効範囲』

コマンド行からの SQL プロシージャーの作成

前提条件:

- ユーザーには、 SQL プロシージャーの CREATE PROCEDURE ステートメントを実行するための特権が必要です。
- プロシージャーの SQL プロシージャー本体に組み込まれているすべての SQL ステートメントを実行するための特権が必要です。
- SQL プロシージャーの CREATE PROCEDURE ステートメント内で参照されているデータベース・オブジェクトは、そのステートメントの実行前に存在している必要があります。

手順:

- 次のステップでスクリプトを作成するときに使用する終了文字として、コマンド行プロセッサー (DB2 CLP) のデフォルトの終了文字、つまりセミコロン (';'), 以外の代替の終了文字を選択します。

  これが必要なのは、ルーチンの CREATE ステートメントの本体に組み込まれている SQL ステートメントの終了と、 CREATE PROCEDURE ステートメントそのものの終了を CLP が区別するためです。 SQL ルーチン本体の中に組み込む SQL ステートメントの終了を示すためにはセミコロンを使用し、 CREATE ステートメントそのものの終了を示すためにはその選択した終了文字を使用する必要があります。また、 CLP スクリプトの中にさらに他の SQL ステートメントを組み込む場合は、その選択した終了文字でそれらのステートメントの終了を示します。

たとえば、以下の CREATE PROCEDURE ステートメントでは、myCLPscript.db2 という名前の DB2 CLP スクリプトの終了文字として、アットマーク ('@') を使用しています。

```
CREATE PROCEDURE UPDATE_SALARY_IF
(IN employee_number CHAR(6), IN rating SMALLINT)
LANGUAGE SQL
BEGIN
  DECLARE not_found CONDITION FOR SQLSTATE '02000';
  DECLARE EXIT HANDLER FOR not_found
     SIGNAL SQLSTATE '20000' SET MESSAGE_TEXT = 'Employee not found';

  IF (rating = 1)
    THEN UPDATE employee
      SET salary = salary * 1.10, bonus = 1000
      WHERE empno = employee_number;
  ELSEIF (rating = 2)
    THEN UPDATE employee
      SET salary = salary * 1.05, bonus = 500
      WHERE empno = employee_number;
  ELSE UPDATE employee
      SET salary = salary * 1.03, bonus = 0
      WHERE empno = employee_number;
  END IF;
END
@
```

- コマンド行から以下の CLP コマンドを使用して、プロシージャーの CREATE PROCEDURE ステートメントを含んだ DB2 CLP スクリプトを実行します。

  ```
  db2 -td <terminating-character> -vf <CLP-script-name>
  ```

  <*terminating-character*> は、実行する CLP スクリプト・ファイル *CLP-script-name* で使用している終了文字です。

  DB2 CLP オプションの -td は、 CLP 終止符のデフォルトを *terminating character* にリセットするという指定です。 *-vf* は、 CLP の任意指定の冗長 (*-v*) オプションを使用するという指定です。このオプションを指定した場合、スクリプト内の各 SQL ステートメントやコマンドがそれぞれの実行時に画面に表示され、実行結果に関する出力も表示されることになります。 *-f* オプションは、コマンドのターゲットがファイルであるという指定です。

  最初のステップで示したスクリプトを実行するには、システム・コマンド・プロンプトから以下のコマンドを実行します。

  ```
  db2 -td@ -vf myCLPscript.db2
  ```

関連概念:

- 3 ページの『アプリケーション開発におけるルーチン』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE TYPE (構造化) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION (SQL スカラー、表、または行) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE (SQL) ステートメント』

SQL プロシージャーのパラメーター

DB2 では、 SQL プロシージャーでの入力パラメーター、出力パラメーター、入出力パラメーターの使用をサポートしています。パラメーターのモードまたは用途を指定するには、 CREATE PROCEDURE ステートメントで IN、OUT、INOUT のいずれかのキーワードを使用します。 IN パラメーターと OUT パラメーターは値による受け渡し、 INOUT パラメーターは参照による受け渡しになります。

パラメーターの使用は任意です。パラメーターのない SQL プロシージャーを作成することもできます。ただし、複数のパラメーターを指定する場合、それらのパラメーターはプロシージャー内でユニークでなければなりません。また、プロシージャー内でパラメーターと同じ名前の変数を宣言する場合は、プロシージャー内でネストしたラベル付きのアトミック・ブロックの中に宣言する必要があります。そうしない場合、 DB2 は、あいまい参照を検出してしまいます。

パラメーターのデータ型にかかわりなく、 SQL プロシージャーのパラメーターを SQLSTATE または SQLCODE という名前にすることはできません。 SQL プロシージャー内のパラメーターのモードと制限の詳細については、 CREATE PROCEDURE ステートメントを参照してください。

以下の SQL プロシージャー (*myparams*) は、 IN、INOUT、OUT の各パラメーター・モードの使用法を示した例です。 *myfile.db2* という CLP ファイルに SQL プロシージャーを定義し、コマンド行を使用することが前提になります。

```
CREATE PROCEDURE myparams (IN p1 INT, INOUT p2 INT, OUT p3 INT)
LANGUAGE SQL
BEGIN
 SET p2 = p2 + 1;
 SET p3 = 2 * p1;
END@
```

ストアード・プロシージャーを作成するには、コマンド行に以下のコマンドを入力します。

```
db2 -td@ -vf myfile.db2
```

次に、このプロシージャーを呼び出すには、コマンド行に以下のコマンドを入力します。

```
db2 "CALL myParms(1, 3, ?)"
```

'?' は、出力パラメーターのパラメーター・マーカーです。 INOUT パラメーターがあれば、プロシージャー内でそのパラメーターを参照していない場合でも、入力値を指定する必要があります。以下の出力が戻されます。

```
 Value of output parameters
--------------------------
Parameter Name  : P2
Parameter Value : 4

Parameter Name  : P3
Parameter Value : 2

Return Status = 0
```

関連概念:

- 99 ページの『外部ルーチン用のパラメーター・スタイル』
- 48 ページの『プロシージャーのパラメーター・モード』
- 71 ページの『DB2 での SQL Procedural Language (SQL PL)』
- 73 ページの『SQL ルーチンの CREATE ステートメント』
- 80 ページの『SQL プロシージャーの変数 (DECLARE、DEFAULT、SET ステートメント)』

関連タスク:

- 77 ページの『コマンド行からの SQL プロシージャーの作成』

SQL プロシージャーの変数 (DECLARE、DEFAULT、SET ステートメント)

プロシージャーの変数は、コンパウンド・ステートメントの一部としてサポートされています。変数を宣言するときには、キーワード DECLARE を使用します。変数宣言は、 SQL プロシージャー本体の先頭、つまり、条件、条件ハンドラー、カーソル、SQL ステートメントの宣言の前に置く必要があります。

DEFAULT 文節を使用して、変数にデフォルト値を宣言することもできます。この場合のデフォルト値は、定数、特殊レジスター値、式のいずれかになります。以下に例を示します。

```
CREATE PROCEDURE P2(INOUT a VARCHAR(8),
                    OUT  b INTEGER)
LANGUAGE SQL
BEGIN
  DECLARE var1 INTEGER DEFAULT 0;
  DECLARE var2 VARCHAR(5) DEFAULT  a || 'bc';

   -- other SQL statements --

 END@
```

SQL プロシージャーの変数宣言の後に、以下のように割り当てステートメントを使用して、変数やパラメーター (入力パラメーターも含む) に値を割り当てることができます。

```
CREATE PROCEDURE P2(INOUT a VARCHAR(8),
                    OUT  b INTEGER)
LANGUAGE SQL
BEGIN
  DECLARE var1 INTEGER DEFAULT 0;
  DECLARE var2 VARCHAR(5) DEFAULT  a || 'bc';

   SET var1 = 0;
   SET var1 = var1 + 1;
   SET var2  = var2 || 'def';

   SET a = var1;
   SET b = var2;
 END@
```

関連概念:

- 71 ページの『DB2 での SQL Procedural Language (SQL PL)』
- 73 ページの『SQL ルーチンの CREATE ステートメント』
- 79 ページの『SQL プロシージャーのパラメーター』

- 87 ページの『SQL プロシージャーのパフォーマンスの改善』

関連タスク:

- 77 ページの『コマンド行からの SQL プロシージャーの作成』

SQL プロシージャー内のコンパウンド・ブロックと変数の有効範囲

1 つの SQL プロシージャー内に 1 つ以上のコンパウンド・ステートメントを記述できます。コンパウンド・ステートメントは、 DB2 で 1 つのステートメントとしてコンパイルされ、実行される SQL ステートメント・ブロックの開始を示します。コンパウンド・ステートメントは、BEGIN キーワードで始まり、 END キーワードで終わるので、簡単に見分けることができます。また、コード・ブロックを識別するためにラベルを付けることもできます。

変数の有効範囲というコンテキストでは、ラベルを付けることが重要になります。ラベルによって変数名を修飾できるからです。別のコンパウンド・ステートメントや、ネストしたコンパウンド・ステートメントで変数を識別して参照するときに、ラベルによる変数名の修飾は重要な意味を持ちます。

以下の例には、変数 *a* の 2 つの宣言があります。 1 つのインスタンスは、 *lab1* というラベルの付いた外側のコンパウンド・ステートメント内の宣言であり、もう 1 つのインスタンスは、 *lab2* というラベルの付いた内側のコンパウンド・ステートメント内の宣言です。この記述のとおり、代入ステートメント内の *a* の参照先は、*lab2* というラベルの付いたコンパウンド・ブロックのローカル有効範囲内の宣言であると DB2 は判断します。しかし、変数 *a* によって、 lab1 というラベルの付いたコンパウンド・ステートメント・ブロック内の宣言を参照する場合は、そのコンパウンド・ブロック内の宣言を正しく参照するために、変数をそのブロックのラベルで修飾する必要があります。つまり、*lab1.a* というふうに修飾するということです。

```
CREATE PROCEDURE P1 ()
LANGUAGE SQL
  lab1: BEGIN
    DECLARE a INT DEFAULT 100;
    lab2: BEGIN
      DECLARE a INT DEFAULT NULL;

      SET a = a + lab1.a;

      UPDATE T1
       SET T1.b = 5
        WHERE T1.b = a;  <-- Variable a refers to lab2.a
                             unless qualified otherwise

    lab2: END;
 END lab1@
```

SQL プロシージャー内の外側のコンパウンド・ステートメントをアトミックとして宣言できます。そのためには、BEGIN キーワードの後に ATOMIC キーワードを追加します。アトミック・コンパウンド・ステートメント内のステートメントの実行時にエラーが発生した場合は、そのコンパウンド・ステートメント全体がロールバックされます。

関連概念:

- 71 ページの『DB2 での SQL Procedural Language (SQL PL)』
- 79 ページの『SQL プロシージャーのパラメーター』
- 80 ページの『SQL プロシージャーの変数 (DECLARE、DEFAULT、SET ステートメント)』

関連タスク:

- 77 ページの『コマンド行からの SQL プロシージャーの作成』

SQL プロシージャーからのエラー・メッセージの戻り

SQL プロシージャーの CREATE PROCEDURE ステートメントを発行した場合、DB2 は SQL プロシージャー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 プロシージャー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・メッセージを載せたログ・ファイルを作成します。

SQL プロシージャーの 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 プロシージャー・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
```

関連タスク:

- 43 ページの『ルーチンのデバッグ』

関連資料:

- 「*SQL リファレンス 第 1 巻*」の『CURRENT SCHEMA 特殊レジスター』

SQL プロシージャーの条件ハンドラー

以下の項では、条件ハンドラーを取り上げ、 SQL プロシージャーの条件ハンドラーで各種データベース条件に対応する方法について説明しています。

SQL プロシージャーの条件ハンドラー

条件ハンドラー は、ある条件が発生したときの SQL プロシージャーの振る舞いを決定します。一般的な条件、名前付き条件、または特定の SQLSTATE 値に対して 1 つまたは複数の条件ハンドラーを SQL プロシージャーで宣言することができます。

SQL プロシージャー内のステートメントによって SQLWARNING または NOT FOUND 条件が生じた場合、もしそれぞれの条件に対してハンドラーを宣言していたときは、それに対応するハンドラーに DB2® から制御が渡されます。そのような条件に対してハンドラーを宣言していなかった場合、 SQL プロシージャー本体の次のステートメントに DB2 から制御が渡されます。 SQLCODE および SQLSTATE 変数を宣言していた場合、それに対応する条件値がその変数内に入ります。

特定の SQLSTATE または SQLEXCEPTION 条件に対してハンドラーを宣言していた場合に、 SQL プロシージャー内のステートメントが SQLEXCEPTION 条件を生じたときは、そのハンドラーに DB2 から制御が渡されます。 SQLSTATE および SQLCODE 変数を宣言していた場合、ハンドラーの実行が正常に完了した後のその変数の値はそれぞれ ‘00000’ と 0 になります。

特定の SQLSTATE または SQLEXCEPTION 条件に対してハンドラーを宣言していなかった場合に、 SQL プロシージャーのステートメントが SQLEXCEPTION 条件を生じたときは、 DB2 は SQL プロシージャーを終了してから呼び出し元に戻ります。

関連概念:

- 86 ページの『条件ハンドラーでの SIGNAL および RESIGNAL ステートメント』
- 83 ページの『条件ハンドラーの宣言』
- 87 ページの『SQL プロシージャーでの SQLCODE および SQLSTATE 変数』

関連タスク:

- 82 ページの『SQL プロシージャーからのエラー・メッセージの戻り』

条件ハンドラーの宣言

特定の条件が発生したときの SQL プロシージャーの動作を定義するには、条件ハンドラーを宣言する必要があります。ハンドラー宣言の一般的な形式は、以下のようなものです。

```
  DECLARE handler-type HANDLER FOR  condition
SQL-procedure-statement
```

DB2® で *condition* に合致した条件が発生した場合、 DB2 は制御をその条件ハンドラーに渡します。そして、条件ハンドラーは *handler-type* によって示されているアクションを実行してから、 *SQL-procedure-statement* を実行します。

Handler-types

CONTINUE
SQL-procedure-statement が完了した後に、エラーが起きた後のステートメントで実行が継続されることを指定します。

EXIT *SQL-procedure-statement* が完了した後に、ハンドラーが含まれるコンパウンド・ステートメントの後から実行が継続されることを指定します。

UNDO *SQL-procedure-statement* が実行される前に、 DB2 によってハンドラーを含むコンパウンド・ステートメントの SQL 操作がロールバックされることを指定します。 *SQL-procedure-statement* が完了したら、ハンドラーをもったコンパウンド・ステートメントの終了時点から実行が継続されます。

注: UNDO ハンドラーは ATOMIC コンパウンド・ステートメントのみで宣言できます。

条件
DB2 には、以下のような 3 つの一般的条件があります。

NOT FOUND
SQLCODE が +100 になるか、または '02' で始まる SQLSTATE になるすべての条件を識別します。

SQLEXCEPTION
SQLCODE が負の値になる条件を識別します。

SQLWARNING
警告条件 (SQLWARN0 が 'W') になる条件、または +100 以外の正の数の SQL 戻りコードになる条件を識別します。それに対応する SQLSTATE 値は、文字 '01' で始まります。

さらに、DECLARE ステートメントを使用して特定の SQLSTATE に対して独自の条件を定義できます。

SQL-procedure-statement
単一 SQL プロシージャー・ステートメントを使用して、条件ハンドラーの振る舞いを定義することができます。 DB2 は、 BEGIN...END ブロックによって区切られたコンパウンド・ステートメントを、単一 SQL プロシージャー・ステートメントとして受け入れます。コンパウンド・ステートメントを使用して条件ハンドラーの振る舞いを定義し、その際にハンドラーで SQLSTATE または SQLCODE 変数の値を保存したい場合には、その変数の値をローカル変数か、コンパウンド・ブロックの最初のステートメントのパラメーターに割り当てる必要があります。コンパウンド・ブロックの最初のステートメントによって SQLSTATE または SQLCODE の値がローカル変数またはパラメーターに割り当てられない場合には、 DB2 が条件ハンドラーを呼び出す原因となった値を SQLSTATE および SQLCODE は保持できません。

次の例は、単純な条件ハンドラーを表したものです。

CONTINUE ハンドラー
DB2 で NOT FOUND 条件が生じると、このハンドラーは、ローカル変数 *at_end* に 1 の値を割り当てます。それから、DB2 は制御を、 NOT FOUND 条件を起こしたステートメントの次のステートメントに渡します。

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET at_end = 1;
```

EXIT ハンドラー

ここの例では、終了ハンドラーの有効範囲は、 A というラベルの付いたコンパウンド・ステートメントに限定されます。表 JAVELIN が存在しないと、 DROP ステートメントによって NO_TABLE 条件が生じます。終了ハンドラーが活動化され、OUT_BUFFER がストリング Table does not exist に設定され、実行は INSERT ステートメントを使用して C から続行されますが、コンパウンド・ステートメント内のステートメントはそれ以上探索されません。DROP ステートメントの実行が正常に完了した場合、ハンドラーは活動化されず、実行は SET ステートメントを使用して B から続行されます。

```
CREATE PROCEDURE EXIT_TEST ()
 LANGUAGE SQL
 BEGIN
    DECLARE OUT_BUFFER VARCHAR(80);
    DECLARE NO_TABLE CONDITION FOR SQLSTATE '42704';

    A: BEGIN
          DECLARE EXIT HANDLER FOR NO_TABLE
          BEGIN
             SET OUT_BUFFER='Table does not exist';
          END;

          -- Drop potentially nonexistent table:
          DROP TABLE JAVELIN;

       B: SET OUT_BUFFER='Table dropped successfully';
    END;

    -- Copy OUT_BUFFER to some message table:
    C: INSERT INTO MESSAGES VALUES OUT_BUFFER;
 END
```

UNDO ハンドラー

ここの例では、取り消しハンドラーの有効範囲は、 A というラベルの付いたコンパウンド・ステートメントに限定されます。表 JAVELIN が存在しないと、 DROP ステートメントによって NO_TABLE 条件が生じます。取り消しハンドラーが活動化され、DROP より前の INSERT がロールバックされ、OUT_BUFFER がストリング Table does not exist に設定され、実行は INSERT ステートメントを使用して C から続行されますが、コンパウンド・ステートメント A 内のステートメントはそれ以上探索されません。DROP ステートメントの実行が正常に完了した場合、ハンドラーは活動化されず、実行は SET ステートメントを使用して B から続行されます。

```
CREATE PROCEDURE UNDO_TEST ()
 LANGUAGE SQL
 BEGIN
    DECLARE OUT_BUFFER VARCHAR(80);
    DECLARE NO_TABLE CONDITION FOR SQLSTATE '42704';

    A: BEGIN ATOMIC
          DECLARE UNDO HANDLER FOR NO_TABLE
          BEGIN
             SET OUT_BUFFER='Table does not exist';
          END;

          INSERT INTO MESSAGES VALUES
             'This message will be removed by a rollback.';

          -- Drop potentially nonexistent table:
          DROP TABLE JAVELIN;
```

```
    B: SET OUT_BUFFER='Table dropped successfully';
  END;

  -- Copy OUT_BUFFER to some message table:
  C: INSERT INTO MESSAGES VALUES OUT_BUFFER;
END
```

注: UNDO ハンドラーは ATOMIC コンパウンド・ステートメントのみで宣言できます。

関連概念:

- 83 ページの『SQL プロシージャーの条件ハンドラー』
- 86 ページの『条件ハンドラーでの SIGNAL および RESIGNAL ステートメント』
- 87 ページの『SQL プロシージャーでの SQLCODE および SQLSTATE 変数』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『コンパウンド SQL (組み込み) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『コンパウンド SQL (動的) ステートメント』

条件ハンドラーでの SIGNAL および RESIGNAL ステートメント

SIGNAL および RESIGNAL ステートメントを使用して、特定の SQLSTATE を明示的に起こすことができます。 SIGNAL および RESIGNAL ステートメントの SET MESSAGE_TEXT 文節を使用して、発生した SQLSTATE とともに DB2® が表示するテキストを定義します。

次の例では、 SQL プロシージャー本体はカスタム SQLSTATE 72822 の条件ハンドラーを宣言します。 SQLSTATE 72822 を生じる SIGNAL ステートメントが SQL プロシージャーによって実行されると、 DB2 は条件ハンドラーを呼び出します。条件ハンドラーは、 SQL 変数 *var* の値を IF ステートメントでテストします。 *var* が OK である場合には、ハンドラーは SQLSTATE の値を 72623 と再定義して、 SQLSTATE 72623 に関連したテキストにストリング・リテラルを割り当てます。 *var* が OK でない場合には、ハンドラーによって SQLSTATE 値が 72319 と再定義されて、その SQLSTATE に関連したテキストに *var* 値が割り当てられます。

```
DECLARE EXIT HANDLER FOR SQLSTATE '72822'
BEGIN
  IF ( var = 'OK' )
    RESIGNAL SQLSTATE '72623' SET MESSAGE_TEXT = 'Got SQLSTATE 72822';
  ELSE
    RESIGNAL SQLSTATE '72319' SET MESSAGE_TEXT = var;
END;

SIGNAL SQLSTATE '72822';
```

関連概念:

- 83 ページの『SQL プロシージャーの条件ハンドラー』
- 83 ページの『条件ハンドラーの宣言』

- 87 ページの『SQL プロシージャーでの SQLCODE および SQLSTATE 変数』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『SIGNAL ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『RESIGNAL ステートメント』

SQL プロシージャーでの SQLCODE および SQLSTATE 変数

SQL プロシージャーのデバッグを容易にするには、 SQL プロシージャーの様々な時点で SQLCODE および SQLSTATE の値を表に挿入したり、診断ストリングの OUT パラメーターとして SQLCODE および SQLSTATE の値を戻すのが役に立つかもしれません。 SQLCODE および SQLSTATE 値を使用するには、 SQL プロシージャー本体で以下のような SQL 変数を宣言する必要があります。

```
DECLARE SQLCODE INTEGER DEFAULT 0;
DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
```

ステートメントが実行されると DB2® は常にこれらの変数を暗黙で設定します。ハンドラーを用意された条件がステートメントによって生じた場合、ハンドラーの実行の開始時点で SQLSTATE および SQLCODE 変数の値を利用することができます。ただしこの変数は、ハンドラー内の最初のステートメントが実行されるとただちにリセットされます。そのため、ハンドラーの最初のステートメント内のローカル変数に SQLSTATE および SQLCODE の値をコピーしておくのが一般的な措置です。以下の例では、すべての条件で CONTINUE ハンドラーが使用されて、retcode という別の変数に SQLCODE 変数がコピーされます。次に実行可能ステートメント内で変数 retcode を使用して、プロシージャー・ロジックを制御したり、出力パラメーターとして値を返したりすることができます。

```
BEGIN
  DECLARE SQLCODE INTEGER DEFAULT 0;
  DECLARE retcode INTEGER DEFAULT 0;

  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION, SQLWARNING, NOT FOUND
     SET retcode = SQLCODE;

  executable-statements
END
```

注: SQL プロシージャーで SQLCODE または SQLSTATE 変数をアクセスする場合には、 DB2 によって、後続するステートメントの SQLCODE 値は 0、また SQLSTATE 値は ‘00000’ に設定されます。

関連概念:

- 83 ページの『SQL プロシージャーの条件ハンドラー』
- 86 ページの『条件ハンドラーでの SIGNAL および RESIGNAL ステートメント』
- 83 ページの『条件ハンドラーの宣言』

SQL プロシージャーのパフォーマンスの改善

DB2 による SQL PL とインライン SQL PL のコンパイルの概要:

SQL プロシージャーのパフォーマンスを改善する方法を説明する前に、 CREATE PROCEDURE ステートメントの実行時に DB2 が SQL プロシージャーをコンパイルする方法について取り上げる必要があります。

SQL プロシージャーの作成時に、 DB2 は、プロシージャー本体の中にある SQL 照会とプロシージャー・ロジックを分離します。 SQL 照会については、パフォーマンスの最大化のために、パッケージ内のセクションに静的にコンパイルします。静的にコンパイルした照会のセクションの主な中身は、 DB2 オプティマイザーがその照会のために選択したアクセス・プランです。パッケージとは、そのようなセクションの集合です。パッケージとセクションの詳細については、「DB2 SQL 解説書」を参照してください。一方、プロシージャー・ロジックは、ダイナミック・リンク・ライブラリーにコンパイルします。

プロシージャーの実行時に、プロシージャー・ロジックから SQL ステートメントに制御が移るたびに、 DLL と DB2 エンジンとの間で「コンテキストの切り替え」が発生します。 DB2 バージョン 8.1 以降、 SQL プロシージャーは「unfenced モード」で実行されます。つまり、 DB2 エンジンと同じアドレッシング・スペースで実行されるということです。したがって、ここで言う「コンテキストの切り替え」とは、オペレーティング・システム・レベルで発生する完全な「コンテキストの切り替え」ではなく、むしろ DB2 内の層の切り替えです。頻繁に呼び出されるプロシージャー (OLTP アプリケーション内のプロシージャーなど) や、多数の行を処理するプロシージャー (データ・クレンジングを実行するプロシージャーなど) でコンテキストの切り替えの数を減らせば、パフォーマンスにかなりの影響を与えることができます。

SQL PL を含んだ SQL プロシージャーは、個々の SQL 照会をパッケージ内の各セクションに静的にコンパイルすることによってインプリメントするのに対し、インライン SQL PL 関数は、その名が示すとおり、関数の本体を、関数を使用する照会の中にインライン化することによってインプリメントします。 SQL 関数内の各照会は、あたかも関数本体が 1 つの照会であるかのように一緒にコンパイルされます。このコンパイルは、その関数を使用するステートメントのコンパイルが行われるたびに発生します。ただし、SQL プロシージャーの場合とは異なり、 SQL 関数内のプロシージャー・ステートメントは、データ・フロー・ステートメントとは別の層で実行されるわけではありません。したがって、プロシージャー・ステートメントとデータ・フロー・ステートメントの間で制御が移るたびに、コンテキストの切り替えが発生するわけではないということです。

ロジック内に副作用がなければ SQL 関数を使用する:

このように、プロシージャー内の SQL PL と関数内のインライン SQL PL とではコンパイルの方法が違うので、プロシージャー・コードが SQL データを照会するだけでデータを変更しない限り、つまり、データベース内外のデータに関する副作用がない限り、プロシージャー・コードは、プロシージャー内よりも関数内にあったほうが実行速度が上がると考えられます。

ただし、このようなメリットを生かせるのは、実行する必要のあるすべてのステートメントが SQL 関数内でサポートされている場合に限られます。 SQL 関数には、データベースを変更する SQL ステートメントを組み込めません。また、関数のイ

ンライン SQL PL として使用できるのは、 SQL PL のサブセットにすぎません。たとえば、 CALL ステートメントの実行、カーソルの宣言、 SQL 関数による結果セットの生成などは実行できません。

以下に示すのは、パフォーマンスを最大化する目的で SQL 関数に変換するのに適している SQL PL を含んだ SQL プロシージャーの一例です。

```
  CREATE PROCEDURE GetPrice (IN Vendor CHAR&(20&),
                             IN Pid INT, OUT price DECIMAL(10,3))
LANGUAGE SQL
BEGIN
    IF Vendor eq; ssq;Vendor 1ssq;
      THEN SET price eq; (SELECT ProdPrice
                            FROM V1Table
                            WHERE Id = Pid);
    ELSE IF Vendor eq; ssq;Vendor 2ssq;
      THEN SET price eq; (SELECT Price FROM V2Table
                            WHERE Pid eq; GetPrice.Pid);
    END IF;
  END
```

これを SQL 関数として記述すると、以下のようになります。

```
  CREATE FUNCTION GetPrice (Vendor CHAR(20), PId INT)
  RETURNS  DECIMAL(10,3)
LANGUAGE SQL
BEGIN
    DECLARE price DECIMAL(10,3);
    IF Vendor = 'Vendor 1'
      THEN SET price = (SELECT ProdPrice
                            FROM V1Table
                            WHERE Id = Pid);
    ELSE IF Vendor = 'Vendor 2'
      THEN SET price = (SELECT Price FROM V2Table
                  WHERE Pid = GetPrice.Pid);
    END IF;
    RETURN price;
  END
```

関数の呼び出しは、プロシージャーの呼び出しとは異なることも覚えておく必要があります。関数を呼び出すには、 VALUES ステートメントを使用するか、SELECT ステートメントや SET ステートメントなどの中で式が有効な場所に関数を記述して呼び出します。以下はいずれも、この新しい関数を呼び出す方法として有効です。

```
  VALUES (GetPrice('IBM', 324))

  SELECT VName FROM Vendors WHERE GetPrice(Vname, Pid) < 10

  SET price = GetPrice(Vname, Pid)
```

SQL PL プロシージャー内で 1 つのステートメントを使用すれば十分な場合に複数のステートメントを使用しない:

基本的に SQL は簡潔に記述するほうが良いのですが、実際には簡潔でない SQL を記述してしまうこともよくあります。たとえば、次のような SQL ステートメントがあるとしましょう。

```
  INSERT INTO tab_comp VALUES (item1, price1, qty1);
  INSERT INTO tab_comp VALUES (item2, price2, qty2);
  INSERT INTO tab_comp VALUES (item3, price3, qty3);
```

これは、以下の 1 つのステートメントとして記述できます。

```
INSERT INTO tab_comp VALUES (item1, price1, qty1),
                            (item2, price2, qty2),
                            (item3, price3, qty3);
```

この複数行の挿入ステートメントの実行にかかる時間は、元の 3 つのステートメントの実行にかかる時間のほぼ 3 分の 1 です。これだけを取り出したコードであれば、パフォーマンスの改善はごくわずかでしょうが、ループやトリガー本体などの中でこのコード断片を繰り返し実行する場合は、かなりの改善が期待できます。

同じように、以下のような一連の SET ステートメントがあるとしましょう。

```
SET A = expr1;
SET B = expr2;
SET C = expr3;
```

これは、以下の 1 つの VALUES ステートメントとして記述できます。

```
VALUES expr1, expr2, expr3 INTO A, B, C;
```

この書き換えでは、元の一連のステートメントのセマンティクスをそのまま保持しています。ただし、元のいずれか 2 つのステートメントの間に依存関係が存在する場合は別です。この点を示す以下の例について考えてみましょう。

```
SET A = monthly_avg * 12;
SET B = (A / 2) * correction_factor;
```

この 2 つのステートメントを以下のように書き換えるとしましょう。

```
VALUES (monthly_avg * 12, (A / 2) * correction_factor) INTO A, B;
```

この場合は、元のセマンティクスがそのまま保持されていません。 INTO キーワードの前の両方の式は「並列的に」評価されるわけではないからです。つまり、*B* に代入される値は *A* に代入される値に基づくというのが、元のステートメントで意図されているセマンティクスですが、書き換え後のコードにはそれが反映されていないということです。

複数の SQL ステートメントを 1 つの SQL 式にまとめる:

SQL 言語には、他のプログラム言語と同じように、 2 種類の条件構造体が用意されています。つまり、プロシージャー型の構造体 (IF ステートメント、CASE ステートメント) と関数型の構造体 (CASE 式) です。 1 つの計算処理を表すためにどちらのタイプの構造体でも使用できる状況では、ほとんどの場合、どちらを使用するかは好みの問題です。ただし、 CASE 式によって記述したロジックは、 CASE ステートメントや IF ステートメントによって記述したロジックよりもコンパクトであり、効率的でもあります。

以下の SQL PL コード断片について考えてみましょう。

```
IF (Price <= MaxPrice) THEN
  INSERT INTO tab_comp(Id, Val) VALUES(Oid, Price)semi;
ELSE
  INSERT INTO tab_comp(Id, Val) VALUES(Oid, MaxPrice)semi;
  END IF;
```

この IF 文節の条件は、 tab_comp.Val 列に挿入する値を決定するという目的のためだけに使用しています。プロシージャー層とデータ・フロー層の間のコンテキストの切り替えを避けるために、この同じロジックを CASE 式付きの 1 つの INSERT で記述すれば、以下のようになります。

```
INSERT INTO tab_comp(Id, Val)
       VALUES(Oid,
            CASE
               WHEN (Price <= MaxPrice) THEN Price
               ELSE MaxPrice
                END);
```

CASE 式は、スカラー値が有効な場所であればどんなコンテキストでも使用できるというのは注目に値します。特に便利なのは、代入の右辺で使用できるということです。以下に例を示します。

```
IF (Name IS NOT NULL) THEN
  SET ProdName = Name;
ELSEIF (NameStr IS NOT NULL) THEN
  SET ProdName = NameStr;
ELSE
  SET ProdName = DefaultName;
  END IF;
```

これは、以下のように記述できます。

```
SET ProdName = (CASE
                  WHEN (Name IS NOT NULL) THEN Name
                  WHEN (NameStr IS NOT NULL) THEN NameStr
                  ELSE  DefaultName
                END);
```

実際に、この例の場合はさらに優れた解決策があります。

```
SET ProdName = COALESCE(Name, NameStr, DefaultName);
```

SQL を分析して、必要に応じて書き換える作業には時間がかかりますが、その時間をかけることから得られるメリットを過小評価しないでください。パフォーマンス上のメリットは、プロシージャーの分析と書き換えにかけた時間の何倍もの価値があるはずです。

SQL の一括設定のセマンティクスを活用する:

ループ、代入、カーソルなどのプロシージャー型の構造体を使用すれば、 SQL DML ステートメントだけでは記述できない計算処理を記述できます。その一方で、プロシージャー・ステートメントが手元にあると、実際には SQL DML ステートメントだけで計算処理を記述できる場合でも、プロシージャー・ステートメントに頼ってしまう危険があります。すでに見たとおり、プロシージャーによる計算処理は、 DML ステートメントによって記述した等価の計算処理よりもパフォーマンスが桁違いに落ちることがあります。以下のコード断片について考えてみましょう。

```
DECLARE cur1 CURSOR FOR SELECT col1, col2 FROM tab_comp;
OPEN cur1;
  FETCH cur1 INTO v1, v2;
WHILE SQLCODE ≠ 100 DO
  IF (v1 > 20) THEN
    INSERT INTO tab_sel VALUES (20, v2);
ELSE
```

```
    INSERT INTO tab_sel VALUES (v1, v2);
  END IF;
  FETCH cur1 INTO v1, v2;
END WHILE;
```

まずループ本体は、『複数の SQL ステートメントを 1 つの SQL 式にまとめる』の項で取り上げた書き換えを適用することによって改善できます。

```
DECLARE cur1 CURSOR FOR SELECT col1, col2 FROM tab_comp;
OPEN cur1;
  FETCH cur1 INTO v1, v2;
WHILE SQLCODE ≠ 100 DO
  INSERT INTO tab_sel VALUES (CASE
                                WHEN v1 > 20 THEN 20
                                ELSE v1
                              END, v2);
  FETCH cur1 INTO v1, v2;
END WHILE;
```

しかし、よく見ると、このコード・ブロック全体は、サブ SELECT 付きの 1 つの INSERT として記述できます。

```
INSERT INTO tab_sel (SELECT (CASE
                               WHEN col1 > 20 THEN 20
                               ELSE col1
                             END),
                             col2
                      FROM tab_comp);
```

元のコードでは、 SELECT ステートメントの各行で、プロシージャー層とデータ・フロー層の間のコンテキストの切り替えが発生します。一方、書き換えた後のコードでは、コンテキストの切り替えがまったく発生しないので、オプティマイザーは計算処理全体をグローバルに最適化できます。

ただし、以下のように各 INSERT ステートメントの対象になっている表がそれぞれ異なる場合、これほど劇的な単純化は不可能です。

```
DECLARE cur1 CURSOR FOR SELECT col1, col2 FROM tab_comp;
OPEN cur1;
  FETCH cur1 INTO v1, v2;
WHILE SQLCODE ≠ 100 DO
  IF (v1 > 20) THEN
    INSERT INTO tab_default VALUES (20, v2);
ELSE
    INSERT INTO tab_sel VALUES (v1, v2);
  END IF;
  FETCH cur1 INTO v1, v2;
END WHILE;
```

それでも、以下のようにすれば、SQL の一括設定の機能を活用できます。

```
INSERT INTO tab_sel (SELECT col1, col2
                       FROM tab_comp
                      WHERE col1 <= 20);
INSERT INTO tab_default (SELECT col1, col2
                         FROM tab_comp
                         WHERE col1 > 20);
```

このようにカーソル・ループを除去するには時間がかかりますが、既存のプロシージャー・ロジックのパフォーマンスを改善できることを考えれば、そのための価値は十分にあると言えます。

DB2 オプティマイザーに常に最新の情報を提供する:

プロシージャーの作成時に、個々の SQL 照会は、パッケージ内の各セクションにコンパイルされます。 DB2 オプティマイザーが照会の実行プランを選択するための基礎になるのは、特に表の統計 (表のサイズや、列内のデータ値の相対度数など) と、照会のコンパイルの時点で使用可能な索引です。表にかなりの変更があった場合は、その表に関する統計を DB2 で収集するべきです。また、統計を更新した場合や、新しい索引を作成した場合は、その表を使用する SQL プロシージャーに関連するパッケージを再バインドして、最新の統計と索引に基づくプランを DB2 で作成するようにしてください。

表の統計を更新するには、RUNSTATS コマンドを使用します。 SQL プロシージャーに関連するパッケージを再バインドするには、 DB2 バージョン 8.1 に用意されている REBIND_ROUTINE_PACKAGE 組み込みプロシージャーを使用します。たとえば、プロシージャー MYSCHEMA.MYPROC のパッケージを再バインドするには、以下のコマンドを使用できます。

```
CALL SYSPROC.REBIND_ROUTINE_PACKAGE('P', 'MYSCHEMA.MYPROC', 'ANY')
```

'P' は、このパッケージがプロシージャーに対応していることを示し、 'ANY' は、関数とタイプの解決時に SQL パス内のすべての関数とタイプを対象にすることを示します。詳細については、「コマンド解説書」の『REBIND コマンド』の項目を参照してください。

関連概念:

- 5 ページの『ルーチンのタイプ (プロシージャー、関数、メソッド)』
- 71 ページの『DB2 での SQL Procedural Language (SQL PL)』
- 75 ページの『SQL プロシージャーの設計上の考慮事項』

関連タスク:

- 77 ページの『コマンド行からの SQL プロシージャーの作成』

SQL 表関数

SQL データを変更する SQL 表関数

MODIFIES SQL DATA 文節が SQL 表関数の CREATE FUNCTION ステートメントで指定されている場合、 SQL 表関数の本体には表データを変更する SQL ステートメントを含めることができます。動的コンパウンド・ステートメントでサポートされているすべてのステートメントは、以下のものを含む SQL 表関数の本体でサポートされています。

- INSERT
- UPDATE
- DELETE
- MERGE
- SELECT (FROM 文節がデータ変更ステートメントを参照する場所を select)

SQL データを変更する SQL 表関数は、表データを変更する作業をカプセル化し、結果セットを戻すのに有用な方法です。結果セットは表関数内でアクセスされた行または変更された行を戻すために使用できます。複数の表の変更された行を単一の

結果セットに戻すことができます。 SQL データを変更する SQL 表関数は、表データにアクセスするか、または表データを変更するトランザクションを監査するために使用できます。

MODIFIES SQL DATA 文節のサポートは、SQL プロシージャーおよび関数に限定されています。 MODIFIES SQL DATA を指定する外部表関数は作成できません。

SQL データを変更する SQL 表関数は、以下のようになります。

- SELECT、SELECT INTO、SET、または RETURN ステートメントに組み込まれた全選択の最も外側の FROM 文節でのみ参照できる。
- SELECT ステートメントの FROM 文節内で SQL データを変更する唯一の SQL 表関数でなければならない。
- 複数の表参照がある場合、FROM 文節で最後の表参照として表示されなければならない。
- FROM 文節の他のすべての表参照と相関関係がなければならない。
- ビュー定義の本体で参照できない。
- SQL データを変更するルーチン内、または AFTER トリガー内でネストできる。

すべてのルーチンと同様に、表関数の定義者が関数本体にあるすべての SQL ステートメントの実行を許可する場合にのみ、表関数を正常に呼び出すことができます。

これらの制限事項は、ステートメントにおける表関数および表の決定的な評価を確実にします。 SQL 表関数に先行する表参照は、SQL 表関数が実行される前に完全に評価されます。 SELECT ステートメントの SELECT リストまたは WHERE 文節における表参照は、 SQL 表関数の実行が完了した後に評価されます。

SQL データを変更する SQL 表関数の例:

注: 以下の例の前提条件になっている完全な SQL を参照する場合や、関連した SQL サンプルを実行する場合は、サンプル tbfnuse.db2 および前提条件スクリプト tbfn.db2 を参照してください。

例 1: SQL データを変更する SQL 表関数:

この表関数は、在庫表のアイテムの数量を更新します。 Inventory 表の itemNoによって指定されたアイテムの数量を、amountによって指定された量で更新するために、 UPDATE ステートメントが使用されます。製品名とアイテムの新規数量を含む結果セットが戻されます。関数が表データを更新するので、 MODIFIES SQL DATA 文節が使用されていることに注意してください。

```
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
```

例 2: SQL データを変更する SQL 表関数の呼び出し:

例 2 の SQL 表関数は SELECT ステートメントから呼び出されます。アイテム番号「ISBN-0-8021-3424-6」で識別されるアイテムの数量は、5 個増やされます。製品名、および更新されたアイテムの数量が戻されます。

```
SELECT productName, quantity
  FROM TABLE(updateInv('ISBN-0-8021-3424-6', 5)) AS T


PRODUCTNAME          QUANTITY
-------------------- -----------
Feng Shui at Home             15
```

例 3: 他の表参照と相関関係のある SQL データを変更する SQL 表関数の呼び出し:

この例では、在庫表「Inventory」内の既存の複数のアイテムの数量を更新します。VALUES 文節を使用して、更新対象のアイテムの行を含んだ「newItem」表参照を生成します。「newItem」の少なくとも 1 つの列は表関数「updateInv」の引き数として示されるので、表関数「updateInv」は表参照「newItem」と相関関係にあります。表関数は FROM 文節の最後の表参照であることに注意してください。

```
SELECT newItem.id, TF.productName, TF.quantity
     FROM (VALUES ('ISBN-0-8021-3424-6', 2),
                  ('ISBN-0-8021-4612-1', 5)) AS newItem(id, quantity),
          TABLE(updateInv(newItem.id, newItem.quantity)) AS TF


ID                 PRODUCTNAME          QUANTITY
------------------ -------------------- -----------
ISBN-0-8021-3424-6 Feng Shui at Home             12
ISBN-0-8021-4612-1 Baseball Heroes               15
```

副選択における SQL 表関数を参照する、または FROM 文節において複数の表関数を要求する、さらに複雑な照会を示すには、共通表式を使用できます。共通表式の使用は、最外部選択において SQL 表関数を分離したり、 SQL データを変更する表関数が FROM 文節における最後の表参照であることを確認するのに実用的な方法です。

例 4: 他の表参照と相関関係のある SQL データおよび共通表式にある SQL データを変更する SQL 表関数の呼び出し:

この例は例 3 を拡張したもので、更新された在庫アイテムの単価および合計在庫値を戻します。合計在庫値は、これらのアイテムの新規数量に価格リスト表 priceList からの価格を掛けて計算されます。

```
WITH newInv(itemNo, quantity) AS
  (SELECT id, TF.quantity
     FROM (VALUES ('ISBN-0-8021-3424-6', 5),
                  ('ISBN-0-8021-4612-1', 10)) AS newItem(id, q),
          TABLE(updateInv(newItem.id, newItem.q)) AS TF)
SELECT itemNo, quantity, unitPrice, (quantity * unitPrice) as TotalInvValue
  FROM newInv, priceList
    WHERE itemNo = priceList.itemID
```

```
ITEMNO             QUANTITY    UNITPRICE TOTALINVVALUE
------------------ ----------- --------- -----------------
ISBN-0-8021-3424-6          12     10.00            120.00
ISBN-0-8021-4612-1          15     20.00            300.70
```

関連タスク:

- 96 ページの『SQL 表関数を使用した監査』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION (SQL スカラー、表、または行) ステートメント』
- 「*SQL リファレンス 第 1 巻*」の『ルーチンで使用可能な SQL ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『サポートされる SQL ステートメント』

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、戻された結果セット行を別の表に挿入したり、保管したりできます。

前提条件:

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、入力引き数 `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 文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                                TIME
-------- ---------- ------------------------------------  --------------------------
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 文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 文節内にある 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
```

関連概念:

- 93 ページの『SQL データを変更する SQL 表関数』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION (SQL スカラー、表、または行) ステートメント』

第 4 章 外部ルーチン

外部ルーチンは、C、C++、Java、OLE のいずれかのプログラム言語で作成できます。ストアード・プロシージャーは、これらの言語以外に、COBOL でも作成できます。

外部ルーチンを構築するには、ルーチンの言語に対応するコンパイラーや開発者キットをデータベース・サーバーにインストールして構成する必要があります。外部ルーチンを呼び出すには、まずビルドと登録が必要です。

外部ルーチン用のパラメーター・スタイル

どのルーチンも、パラメーターの交換に関する個々の規則に準じ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。 PARAMETER STYLE 文節への登録時に特定のパラメーター・スタイルをルーチンに割り当てます。以下に、指定可能なパラメーター・スタイルとその属性を示します。

表 1. 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]	• C/C++ • OLE • .NET 共通言語ランタイム言語 • COBOL [2]	• UDF • ストアード・プロシージャー • メソッド	呼び出し時に渡されるパラメーターに加えて、以下の引き数が以下に示されている順序でルーチンに渡されます。 • 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。 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、以下の引き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。 • 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)。
DB2SQL [1]	• C/C++ • OLE • .NET 共通言語ランタイム言語 • COBOL	• ストアード・プロシージャー	呼び出し時に渡されるパラメーターに加えて、以下の引き数が以下に示されている順序でストアード・プロシージャーに渡されます。 • 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。 • DB2 に戻される SQLSTATE。 • ストアード・プロシージャーの修飾名。 • 個々のストアード・プロシージャー名。 • DB2 に戻される SQL 診断ストリング。 CREATE PROCEDURE ステートメント内で DBINFO 文節を指定すると、dbinfo 構造 (データベースに関する情報が入っています) がストアード・プロシージャーに渡されます。
JAVA	• Java™	• UDF • ストアード・プロシージャー	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。 ストアード・プロシージャーの場合には 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。ストアード・プロシージャー用の 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 文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。 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き数は渡されません。
DB2GENERAL	• Java	• UDF • ストアード・プロシージャー • メソッド	このタイプのルーチンは、 Java メソッドで使用するように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。 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き数は渡されません。

表 1. 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	• C/C++ • .NET 共通言語ランタイム言語 • COBOL	• ストアード・プロシージャー	PARAMETER STYLE GENERAL ストアード・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 文節を指定すると、 dbinfo 構造 (データベースに関する情報が入っています) がストアード・プロシージャーに渡されます。 GENERAL は、 DB2 Universal Database for z/OS and OS/390 の SIMPLE ストアード・プロシージャーと同等です。
GENERAL WITH NULLS	• C/C++ • .NET 共通言語ランタイム言語 • COBOL	• ストアード・プロシージャー	PARAMETER STYLE GENERAL WITH NULLS ストアード・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 文節を指定すると、 dbinfo 構造 (データベースに関する情報が入っています) がストアード・プロシージャーに渡されます。 GENERAL WITH NULLS は、 DB2 Universal Database for z/OS and OS/390 の SIMPLE WITH NULLS ストアード・プロシージャーと同等です。

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、ストアード・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。

関連概念:

- 363 ページの『DB2GENERAL ルーチン』
- 189 ページの『Java ルーチン』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE TYPE (構造化) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE METHOD ステートメント』
- 101 ページの『C/C++、OLE、COBOL で書かれたルーチンに引き数を渡すときの構文』

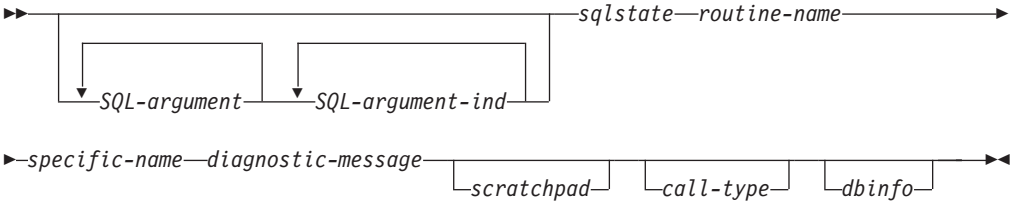
C/C++、OLE、COBOL で書かれたルーチンに引き数を渡すときの構文

ルーチンの DML 参照で指定された SQL 引き数に加えて、 DB2 は追加の引き数を外部ルーチン本体に渡します。そのような引き数の特性と順序は、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き数を受け入れることを確

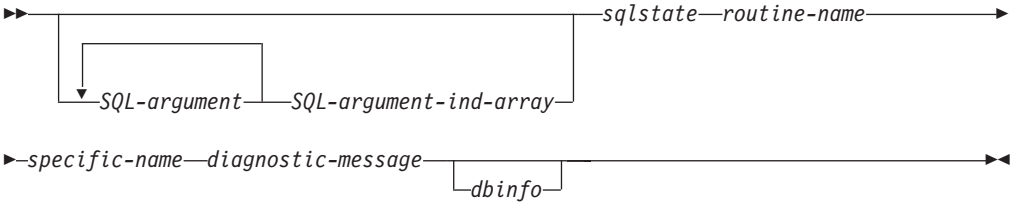
認しなければなりません。 sqludf 組み込みファイルを使用すれば、そのような引き数を簡単に処理して使用することができます。

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

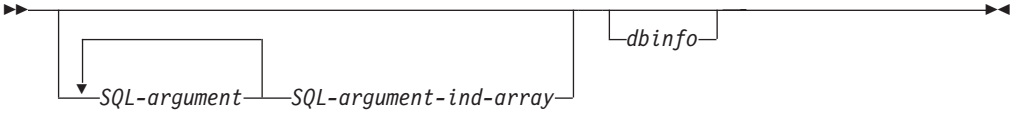


PARAMETER STYLE DB2SQL プロシージャー



PARAMETER STYLE GENERAL プロシージャー

PARAMETER STYLE GENERAL WITH NULLS プロシージャー



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・スタイルの引き数について、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。引き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 つの引き数の後に、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 つの引き数の後に、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 つの引き数の後に、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・プロシージャーの場合、ストアード・プロシージャー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・プロシージャーの IN パラメーター。

  この引き数は、ルーチンを呼び出す前に DB2 によって設定されます。これらの各引き数の値は、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型で表されます。

- 関数またはメソッドの結果、またはストアード・プロシージャーの OUT パラメーター。

  この引き数は、DB2 に戻る前にルーチンによって設定されます。 DB2 はバッファーを割り振り、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ーに入れます。 DB2 は、データ型で示される値を収容するのに十分なバッファー・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、結果データ型は、CAST FROM 文節があればそこで定義され、 CAST FROM 文節がなければ、RETURNS 文節で定義されます。

  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。この機能を利用するように 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。たとえば、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 引き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、結果のデータ型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・プロシージャーの INOUT パラメーター。

  この引き数は、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、ストアード・プロシージャーを呼び出す前に引き数を設定します。引き数用に DB2 で割り振られるバッファーは、 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型の最大サイズを収容するのに十分な大きさです。たとえば、CHAR タイプの INOUT パラメーターは、ストアード・プロシージャーに送られる 10 バイトの VARCHAR をもっていたり、ストアード・プロシージャーから送出される 100 バイトの VARCHAR をもっ

ていたりすることがあります。バッファーは、DB2 に戻る前にストアード・プロシージャーによって設定されます。

DB2 は、データ型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* が値をもっているかまたは NULL であるかを示します。

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、またはストアード・プロシージャーの IN パラメーター。

  この引き数は、ルーチンを呼び出す前に DB2 によって設定されます。この引き数には以下の値のうちの 1 つが入ります。

  0 NULL 以外の引き数があります。
  -1 NULL の引き数があります。

  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 で定義されると引き数はいずれも 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、またはストアード・プロシージャーの OUT パラメーター。

  この引き数は、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、この引き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。たとえば、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
  – 入力引き数の 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。

- ストアード・プロシージャーの INOUT パラメーター。

  この引き数は、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、ストアード・プロシージャーを呼び出す前に引き数を設定します。 *SQL-argument-ind* は、DB2 に戻る前にストアード・プロシージャー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、データ型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
ストアード・プロシージャーに渡される各 SQL 引き数ごとに、*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。

- ストアード・プロシージャーの 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。この引き数には以下の値のうちの 1 つが入ります。

  0 NULL 以外の引き数があります。
  -1 NULL の引き数があります。

  ストアード・プロシージャーを RETURNS NULL ON NULL INPUT で定義すると、ストアード・プロシージャー本体は 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 で定義されると引き数はいずれも NULL になる可能性があるため、ストアード・プロシージャーは *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。

- ストアード・プロシージャーの 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、この引き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。

- ストアード・プロシージャーの 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、ストアード・プロシージャーを呼び出す前に引き数を設定します。 *SQL-argument-ind-array* のエレメントは、 DB2 に戻る前にストアード・プロシージャー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、データ型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

この引き数は、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、この引き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き数には値 '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、データ型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

この引き数は、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。たとえば、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、データ型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

この引き数は、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 引き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、データ型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

この引き数は、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、この引き数を用いて 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* 引き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。 DB2 は、戻り時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。 X'FF' という文字は、SQLCA のトークンを区切るために使用するので、テキスト内では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、データ型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

この引き数は、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。この引き数は、以下のエレメントを持ち、任意の LOB データ型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。 NO FINAL CALL が指定されなかったか、または表関数のデフォルトが使用された場合、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次回の 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、渡される引き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・バッファー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、異常終了が起きたりして、 DB2 上の軽い障害につながることがあります。

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型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

この引き数 (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。この引き数が存在するのは、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、これらの引き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取ります。
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。 *scrachpad* (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。すべての引き数値が渡され、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* 引き数を使用しません。その代わり、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と、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) たとえば、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、たと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。引き数値は、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 呼び出しではなく、この呼び出しの後に OPEN 呼び出しが続きます。 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 引き数値は、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 (たとえば、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) たとえば、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 オプティマイザー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、データ型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 この引き数は、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。引き数は、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 を備えた変数は、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。このフィールドは無符号短整数です。

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。このフィールドは無符号短整数です。

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 Universal Database 製品に共通の構造 (cdpg_db2)、旧バージョンの DB2 Universal Database 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 UDB for z/OS および OS/390 用の構造 (cdpg_mvs) という 3 つの 48 バイト構造の集合体です。移植性を考えて、すべてのルーチンで共通構造 cdpg_db2 を使用することをお勧めします。

   cdgp_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 Universal Database の旧バージョンとの互換性のために、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、 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) の配列指標を示します。

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、無符号長整数です。
- db2_dbcs。 2 バイトのコード・ページで、無符号長整数です。
- db2_mixed。複合コード・ページ (混合コード・ページともいう) で、無符号長整数です。

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。このフィールドは無符号短整数です。

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。このフィールドは無符号短整数です。

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 文節の右側にあるか、 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。このフィールドは無符号短整数です。

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、 *pppvvrrm* 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 Universal Database for z/OS および OS/390
  - **ARI** SQL/DS または DB2 for VM and VSE
  - **QSQ** DB2 Universal Database for iSeries
  - **SQL** DB2 Universal Database
- *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 NT、Windows 2000、Windows XP
    SQLUDF_PLATFORM_SUN Solaris オペレーティング環境版
    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 (numtfcol)

    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. 現在のルーチンを呼び出したストアード・プロシージャーのルーチン ID (procid)。

    ストアード・プロシージャーのルーチン ID は、呼び出し元ストアード・プロシージャー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 (tfcolumn)

    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)、 numtfcol のみが関係します。 *n* は 0 のこともありますが、いずれにしても、CREATE FUNCTION ステートメントの RETURNS TABLE(...) 文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 numtfcol 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で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、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、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。

関連概念:

- 99 ページの『外部ルーチン用のパラメーター・スタイル』
- 174 ページの『C/C++ ルーチン用の組み込みファイル (sqludf.h)』
- 170 ページの『C/C++ ルーチン』

関連タスク:

- 37 ページの『ルーチンの作成』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE TYPE (構造化) ステートメント』
- 「*システム・モニター ガイドおよびリファレンス*」の『appl_id アプリケーション ID : モニター・エレメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE METHOD ステートメント』

外部ルーチンでの 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 (ストアード・プロシージャー、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・レベルを定義する文節を組み込みます。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文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 (たとえば、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。この文節を使用できるのは、ストアード・プロシージャーと 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。たとえば、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。たとえば、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 文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ード・プロシージャーだけが、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ード・プロシージャー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ード・プロシージャーが呼び出された場合、そのストアード・プロシージャーは 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ード・プロシージャーだけが、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。ストアード・プロシージャー内に作成されたセーブポイントは、そのストアード・プロシージャー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、ストアード・プロシージャーも、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る時に暗黙で解放します。

ルーチンは、DB2 から渡された sqlstate 引き数に 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。

ルーチンによる 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わ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。たとえば、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。たとえば、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・プロシージャーのうちの、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返されますが、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元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・プロシージャー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・プロシージャーは 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では異なります。

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。たとえば、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・プロシージャー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き数が、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るということです。

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。

関連タスク:

- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『SQL プロシージャーのプリコンパイル・オプションと BIND オプションのカスタマイズ』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE ステートメント』
- 「コマンド・リファレンス」の『BIND コマンド』
- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION (OLE DB 外部表) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION (SQL スカラー、表、または行) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION (外部スカラー) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION (外部表) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION (ソースまたはテンプレート) ステートメント』
- 「*SQL リファレンス 第 1 巻*」の『ルーチンで使用可能な SQL ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE (外部) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE (SQL) ステートメント』
- 「*SQL リファレンス 第 1 巻*」の『特殊レジスター』

動的 SQL における DYNAMICRULES BIND オプションの影響

PRECOMPILE および BIND の DYNAMICRULES オプションにより、次の動的 SQL 属性にランタイムに適用される値を決定できます。

- 許可検査中に使用される許可 ID。
- 非修飾オブジェクトの修飾に使用される修飾子。
- GRANT、REVOKE、ALTER、CREATE、DROP、COMMENT ON、RENAME、SET INTEGRITY および SET EVENT MONITOR STATE ステートメントを動的に準備するためにパッケージを使用できるかどうか。

DYNAMICRULES 値に加えてパッケージのランタイム環境が、動的 SQL ステートメントがランタイムにどのように振る舞うかを制御します。次の 2 つのランタイム環境が考えられます。

- パッケージはスタンドアロン・プログラムの一部として実行される。
- パッケージはルーチン・コンテキスト内で実行される。

DYNAMICRULES 値とランタイム環境の組み合わせで動的 SQL 属性の値が決まります。その属性値の集合が、動的 SQL ステートメントの振る舞いと呼ばれます。次のような 4 つの振る舞いがあります。

実行振る舞い 動的 SQL ステートメントの許可検査に使用する値および動的 SQL ステートメント内の非修飾オブジェクト参照の暗黙修飾に使用される初期値として、DB2® はパッケージを実行しているユーザーの許可 ID (最初に DB2 へ接続した ID) を使用します。

バインド振る舞い

ランタイムに DB2 は、静的 SQL に適用されるすべての規則を許可と修飾に使用します。つまり、パッケージ所有者の許可 ID を動的 SQL ステートメントの許可検査で使用される値とし、動的 SQL ステートメント内の非修飾オブジェクト参照の暗黙修飾に使用されるパッケージ・デフォルト修飾子にするということです。

定義振る舞い 定義振る舞いは、動的 SQL ステートメントがルーチン・コンテキスト内で実行されるパッケージに存在し、DYNAMICRULES DEFINEBIND または DYNAMICRULES DEFINERUN でバインドされている場合にのみ適用されます。 DB2 は、 (ルーチンのパッケージ・バインド・プログラムではなく) ルーチンの定義者の許可 ID を、動的 SQL ステートメントの許可検査で使用される値として使用し、動的 SQL ステートメント内の非修飾オブジェクト参照の暗黙修飾に使用されるパッケージ・デフォルト修飾として使用します。

呼び出し振る舞い

呼び出し振る舞いは、動的 SQL ステートメントがルーチン・コンテキスト内で実行されるパッケージに存在し、DYNAMICRULES INVOKEBIND または DYNAMICRULES INVOKERUN でバインドされている場合にのみ適用されます。 DB2 は、ルーチンが呼び出されたときに有効であった現行ステートメント許可 ID を、動的 SQL の許可検査で使用される値として使用し、そのルーチン内の動的 SQL 内で非修飾オブジェクト参照の暗黙修飾として使用します。これらのことを次の表にまとめます。

呼び出し環境	使用される ID
任意の静的 SQL	ルーチンを呼び出した SQL の入っているパッケージの OWNER の暗黙または明示的な値。
ビューまたはトリガー定義で使用	ビューまたはトリガーの定義者。
実行振る舞いパッケージからの動的 SQL	DB2 への初期接続を行うために使用された ID。
定義振る舞いパッケージからの動的 SQL	ルーチンを呼び出した SQL の入っているパッケージを使用しているルーチンの定義者。
呼び出し振る舞いパッケージからの動的 SQL	ルーチンを呼び出すカレント許可 ID。

次の表は、それぞれの動的 SQL 振る舞いを決定する DYNAMICRULES 値とランタイム環境の組み合わせを示します。

表 2. 動的 SQL ステートメントの振る舞いを決定する DYNAMICRULES とランタイム環境

DYNAMICRULES 値	スタンドアロン・プログラム環境における動的 SQL ステートメントの振る舞い	ルーチン環境における動的 SQL ステートメントの振る舞い
BIND	バインド振る舞い	バインド振る舞い
RUN	実行振る舞い	実行振る舞い
DEFINEBIND	バインド振る舞い	定義振る舞い
DEFINERUN	実行振る舞い	定義振る舞い
INVOKEBIND	バインド振る舞い	呼び出し振る舞い
INVOKERUN	実行振る舞い	呼び出し振る舞い

次の表に、動的 SQL 振る舞いの各タイプ用の動的 SQL 属性値を示します。

表 3. 動的 SQL ステートメント振る舞いの定義

動的 SQL 属性	バインド振る舞いの動的 SQL 属性の設定	実行振る舞いの動的 SQL 属性の設定	定義振る舞いの動的 SQL 属性の設定	呼び出し振る舞いの動的 SQL 属性の設定
許可 ID	OWNER BIND オプションの暗黙または明示的な値	パッケージを実行するユーザーの ID	ルーチン定義者 (ルーチンのパッケージ所有者ではない)	ルーチンが呼び出されたときの現行ステートメント許可 ID
非修飾オブジェクトのデフォルト修飾子	QUALIFIER BIND オプションの暗黙または明示的な値	CURRENT SCHEMA 特殊レジスター	ルーチン定義者 (ルーチンのパッケージ所有者ではない)	ルーチンが呼び出されたときの現行ステートメント許可 ID
GRANT、REVOKE、ALTER、CREATE、DROP、COMMENT ON、RENAME、 SET INTEGRITY および SET EVENT MONITOR STATE の実行	不可	可	不可	不可

関連概念:

- 「*アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」の『動的 SQL における許可に関する考慮事項』
- 39 ページの『SQL の入ったルーチンの許可およびバインド』

.NET 共通言語ランタイム・ルーチン

以下の項では、 .NET Framework 共通言語ランタイムで実行する .NET ルーチンを作成する方法を説明しています。

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。たとえば、Managed C++、C#、Visual Basic、J# などがあります。

CLR 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機能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『アプリケーション開発におけるルーチン』
- 126 ページの『DB2 .NET Data Provider でサポートされている SQL データ型』
- 127 ページの『CLR ルーチンのパラメーター』
- 130 ページの『CLR プロシージャーからの結果セットの戻り』
- 132 ページの『CLR ルーチンに関する制約事項』
- 133 ページの『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。

- 123 ページの『CLR ルーチンの作成』
- 136 ページの『C# の CLR プロシージャーの例』
- 158 ページの『C# の CLR ユーザー定義関数の例』

関連概念:

- 3 ページの『アプリケーション開発におけるルーチン』
- 99 ページの『外部ルーチン用のパラメーター・スタイル』
- 116 ページの『外部ルーチンでの SQL』

- 5 ページの『ルーチンのタイプ (プロシージャー、関数、メソッド)』
- 39 ページの『SQL の入ったルーチンの許可およびバインド』

関連タスク:

- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『Common Language Runtime (CLR) .NET ルーチンの構築』

関連サンプル:

- 『SpCreate.db2 -- Creates the external procedures implemented in spserver.cs』
- 『SpServer.cs -- C# external code implementation of procedures created in spcat.db2』
- 『SpCreate.db2 -- Creates the external procedures implemented in spserver.vb』
- 『SpServer.vb -- VB.NET implementation of procedures created in SpCat.db2』

CLR ルーチンの作成

IL アセンブリーを参照するプロシージャー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 (プロシージャー、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のある場合。
- 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 の機能については、以下を参照してください。
  - 122 ページの『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。
- .NET Framework バージョン 1.1 がサーバーにインストールされていること。 .NET Framework は単体で入手することも、 Microsoft .NET Framework 1.1 Software Development Kit の一部としても入手することも可能です。
- DB2 の以下のバージョンがインストールされていること。
  - サーバー: DB2 8.2 以降のリリース。
  - クライアント: DB2 8.2 インスタンスにアタッチできるクライアントであれば、 CLR ルーチンを呼び出せます。クライアントに DB2 バージョン 7.2 以降のリリースをインストールすることをお勧めします。
- 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、該当するステートメントの詳細を参照してください。

制約事項:

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。

- 132 ページの『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 ルーチンの機能については、「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` を使用するか、インポートします。
   - DB2 SQL データ型にマップするデータ型を使用して、ホスト変数とパラメーターを正しく宣言します。 DB2 データ型と .NET データ型の間のマッピングについては、以下を参照してください。
     - 126 ページの『DB2 .NET Data Provider でサポートされている SQL データ型』
   - 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。
     - 127 ページの『CLR ルーチンのパラメーター』
   - ルーチンがプロシージャーで、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。
     - 130 ページの『CLR プロシージャー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 プロシージャー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 ルーチンのビルド方法については、以下の関連リンクを参照してください。
   - 共通言語ランタイム・ルーチンのビルド (Building common language runtime routines)
3. そのアセンブリーをデータ・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 文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` 文節に、`CLR` という値を指定します。

- PARAMETER STYLE 文節に、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 文節に、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 文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 文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。
- ルーチンがプロシージャーの場合に、 DYNAMIC RESULT SETS に値 1 を指定すると、呼び出し元に結果セットが戻されます。
- CLR プロシージャーに NOT FENCED 文節を指定することはできません。CLR プロシージャーはデフォルトで、FENCED プロシージャーとして実行されます。

CLR ルーチンを呼び出すには、 217 ページの『ルーチンの呼び出し』を参照してください。

関連概念:

- 122 ページの『共通言語ランタイム (CLR) ルーチン』
- 127 ページの『CLR ルーチンのパラメーター』
- 217 ページの『ルーチンの呼び出し』
- 99 ページの『外部ルーチン用のパラメーター・スタイル』
- 60 ページの『UDF とメソッドのスクラッチパッド』
- 116 ページの『外部ルーチンでの SQL』
- 5 ページの『ルーチンのタイプ (プロシージャー、関数、メソッド)』
- 48 ページの『プロシージャーの結果セット』

関連タスク:

- 130 ページの『CLR プロシージャーからの結果セットの戻り』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『Common Language Runtime (CLR) .NET ルーチンの構築』
- 36 ページの『データベースでのルーチンの作成』
- 43 ページの『ルーチンのデバッグ』

関連資料:

- 132 ページの『CLR ルーチンに関する制約事項』
- 126 ページの『DB2 .NET Data Provider でサポートされている SQL データ型』

関連サンプル:

- 『SpCreate.db2 -- Creates the external procedures implemented in spserver.cs』

- 『SpServer.cs -- C# external code implementation of procedures created in spcat.db2』
- 『SpCreate.db2 -- Creates the external procedures implemented in spserver.vb』
- 『SpServer.vb -- VB.NET implementation of procedures created in SpCat.db2』

DB2 .NET Data Provider でサポートされている SQL データ型

DB2 .NET Data Provider の DB2Type データ型、 DB2 データ型、対応する .NET Framework データ型の間のマッピングを以下の表にまとめます。

表 4. DB2 データ型から .NET データ型へのマッピング

DB2Type Enum	DB2 データ型	.NET データ型
SmallInt	SMALLINT	Int16
Integer	INTEGER	Int32
BigInt	BIGINT	Int64
Real	REAL	Single
Double	DOUBLE PRECISION	Double
Float	FLOAT	Double
Decimal	DECIMAL	Decimal
Numeric	DECIMAL	Decimal
Date	DATE	DateTime
Time	TIME	TimeSpan
Timestamp	TIMESTAMP	DateTime
Char	CHAR	String
VarChar	VARCHAR	String
LongVarChar(1)	LONG VARCHAR	String
Binary	CHAR FOR BIT DATA	Byte[]
VarBinary	VARCHAR FOR BIT DATA	Byte[]
LongVarBinary(1)	LONG VARCHAR FOR BIT DATA	Byte[]
Graphic	GRAPHIC	String
VarGraphic	VARGRAPHIC	String
LongVarGraphic(1)	LONG GRAPHIC	String
Clob	CLOB	String
Blob	BLOB	Byte[]
DbClob	DBCLOB(N)	String

注:

1. これらのデータ型は、 DB2 .NET 共通言語ランタイム・ルーチンではサポートされていません。クライアント・アプリケーションでのみサポートされています。

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 データ型の詳細については、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

関連概念:

- 99 ページの『外部ルーチン用のパラメーター・スタイル』
- 122 ページの『共通言語ランタイム (CLR) ルーチン』
- 127 ページの『CLR ルーチンのパラメーター』

関連タスク:

- 321 ページの『構造化型パラメーターを外部ルーチンに渡す』
- 123 ページの『CLR ルーチンの作成』
- 158 ページの『C# の CLR ユーザー定義関数の例』
- 136 ページの『C# の CLR プロシージャーの例』

関連サンプル:

- 『SpCreate.db2 -- Creates the external procedures implemented in spserver.cs』
- 『SpServer.cs -- C# external code implementation of procedures created in spcat.db2』
- 『SpCreate.db2 -- Creates the external procedures implemented in spserver.vb』
- 『SpServer.vb -- VB.NET implementation of procedures created in SpCat.db2』

CLR ルーチンのパラメーター

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 .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 CREATE ステートメントの EXTERNAL 文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` (プロシージャーと関数に対応)
- `GENERAL` (プロシージャーにのみ対応)
- `GENERAL WITH NULLS` (プロシージャーにのみ対応)
- `DB2SQL` (プロシージャーと関数に対応)

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。

- 99 ページの『外部ルーチン用のパラメーター・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。引き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。

- 99 ページの『外部ルーチン用のパラメーター・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ー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。たとえば、 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 のプロシージャーのプロトタイプは、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,
                             ref 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。

プロシージャーの結果セットのためのパラメーター・マーカーは不要:

プロシージャーのプロシージャー宣言内で、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・プロシージャー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。

- 130 ページの『CLR プロシージャー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プロパティーによって検出できます。

dbinfo クラスにアクセスするには、対象のルーチンを内容とするファイルに IBM®.Data.DB2 アセンブリーを組み込み、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特定の場所 (パラメーター・スタイルによって決まっている位置) 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 データ型としてルーチンに渡します。

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

.NET CLR プロシージャーでは、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ーを定義する場合は、以下のシグニチャーが必要です。

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

関連概念:

- 99 ページの『外部ルーチン用のパラメーター・スタイル』
- 48 ページの『プロシージャーのパラメーター・モード』
- 60 ページの『UDF とメソッドのスクラッチパッド』
- 48 ページの『プロシージャーの結果セット』
- 59 ページの『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』

関連タスク:

- 321 ページの『構造化型パラメーターを外部ルーチンに渡す』
- 158 ページの『C# の CLR ユーザー定義関数の例』
- 136 ページの『C# の CLR プロシージャーの例』
- 130 ページの『CLR プロシージャーからの結果セットの戻り』

関連資料:

- 126 ページの『DB2 .NET Data Provider でサポートされている SQL データ型』

CLR プロシージャー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 プロシージャーを開発できます。 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、プロシージャーの戻りの前に 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。

前提条件:

CLR ルーチンの作成方法の概略を理解しておくと、 CLR プロシージャー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。

123 ページの『CLR ルーチンの作成』

手順:

CLR プロシージャー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、さまざまな文節を指定する中で、特に DYNAMIC RESULT SETS 文節に、プロシージャー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. プロシージャー宣言内で、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 プロパティーを初期化します。
5. 戻したい結果セットを定義する DB2Command オブジェクトの CommandText プロパティー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 の Close() メソッドは、プロシージャー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。

   プロシージャーの戻り時に複数の 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 言語プロシージャーをコンパイルし、 CREATE PROCEDURE ステートメントの EXTERNAL 文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 プロシージャーの 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 プロシージャー・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ら、 CALL ステートメントでプロシージャーを呼び出し、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。

プロシージャー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。

- 217 ページの『ルーチンの呼び出し』

DYNAMIC RESULT SETS に 1 より大きい値を指定した場合:

現時点では、CLR プロシージャーから戻される動的結果セットは 1 つだけです。この制限の詳細については、以下を参照してください。

- 『CLR ルーチンに関する制約事項』

関連概念:

- 217 ページの『ルーチンの呼び出し』
- 48 ページの『プロシージャーのパラメーター・モード』
- 48 ページの『プロシージャーの結果セット』
- 122 ページの『共通言語ランタイム (CLR) ルーチン』

関連タスク:

- 123 ページの『CLR ルーチンの作成』

関連資料:

- 132 ページの『CLR ルーチンに関する制約事項』

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 (プロシージャーや 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 文節付きの 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 構造化型の外部メソッドは作成できません。LANGUAGE 文節に 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 プロシージャーは NOT FENCED プロシージャーとしてインプリメントできない:

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

現時点で CLR プロシージャーから戻される結果セットは最大 1 個:

CLR プロシージャーから戻される結果セットの最大数は、データ・プロバイダー (`IBM.Data.DB2`) が 1 つの接続内で同時に開いておける `DB2DataReader` オブジェクトの最大数に制限されています。現時点で、開いておける最大数は 1 です。したがって、 CLR プロシージャーから戻される結果セットは 1 個だけになります。

CLR プロシージャーを作成する CREATE PROCEDURE ステートメントの DYNAMIC RESULT SETS 文節に 1 よりも大きな値を指定したとしても、そのステートメントの実行時にエラーが発生するわけではありません。

ただし、実行時にプロシージャーが戻るときに開いた状態になっている DB2DataReader は 1 個だけです。したがって、プロシージャーが戻るときには、その開いている 1 個の DB2DataReader と関連付けられている 1 個の結果セットだけが呼び出し元に戻されます。

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 10 進数データ型は、 31 桁の精度と 28 桁のスケールで表現します。.NET プログラム言語の 10 進数データ型は、 29 桁の精度と 28 桁の精度で表現します。したがって、 DB2 の外部 CLR ルーチンでは、データの切り捨てを避けるために、 29 桁の精度と 28 桁のスケールを超える 10 進数値を指定しないでください。

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 (C や 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型:

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

- LONG VARCHAR
- LONG VARCHAR FOR BIT DATA
- LONG GRAPHIC
- DATALINK
- 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 ビットのオペレーティング・システムには .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。

関連概念:

- 122 ページの『共通言語ランタイム (CLR) ルーチン』
- 127 ページの『CLR ルーチンのパラメーター』

関連タスク:

- 123 ページの『CLR ルーチンの作成』
- 130 ページの『CLR プロシージャーからの結果セットの戻り』

CLR ルーチンに関連したエラー

すべての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、その

種のエラーのうち最もよく発生するエラーを 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りなく、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.log 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 文節に 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型の DB2 外部メソッドは作成できません。 LANGUAGE 文節を変更して、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 文節に、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 文節の値は、 '<a>:<b>!<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。
- <b> 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 (たとえば、' <a> ! <b> ' 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している必要があります。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 に存在している必要があります。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 文節で指定した外部アセンブリーが見つかりません。

- EXTERNAL 文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 文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 文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 文節で指定したクラスが存在しません。

- EXTERNAL 文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 文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 文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、 CREATE ステートメントで指定したルーチンのシグニチャーと一致しません。

- EXTERNAL 文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 文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型が、 CLR .NET でサポートされているデータ型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 mscoree.dll が存在している必要があります)。 db2clr.dll が sqllib/bin ディレクトリー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、未処理の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。

関連概念:

- 116 ページの『外部ルーチンでの SQL』
- 39 ページの『SQL の入ったルーチンの許可およびバインド』
- 27 ページの『ルーチンのセキュリティーに関する考慮事項』
- 30 ページの『ライブラリーおよびクラスの管理に関する考慮事項』
- 122 ページの『共通言語ランタイム (CLR) ルーチン』

関連タスク:

- 123 ページの『CLR ルーチンの作成』

C# の CLR プロシージャーの例

プロシージャー (ストアード・プロシージャーともいう) 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 プロシージャー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 プロシージャー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。

- 158 ページの『C# の CLR ユーザー定義関数の例』

前提条件:

CLR プロシージャー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。

- 122 ページの『共通言語ランタイム (CLR) ルーチン』
- 123 ページの『CLR ルーチンの作成』
- 3 ページの『アプリケーション開発におけるルーチン』
- 共通言語ランタイム (CLR) .NET ルーチンのビルド (Building common language runtime (CLR) .NET routines)

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。

手順:

独自の C# CLR プロシージャー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 のプロシージャー』
- 139 ページの『例 2: C# のパラメーター・スタイル GENERAL WITH NULLS のプロシージャー』
- 140 ページの『例 3: C# のパラメーター・スタイル SQL のプロシージャー』
- 143 ページの『例 4: 結果セットを戻す C# のプロシージャー』
- 143 ページの『例 5: dbinfo 構造にアクセスする C# のプロシージャー』
- 144 ページの『例 6: PROGRAM TYPE MAIN スタイルの C# プロシージャー』

C# 外部コード・ファイル:

以下の例では、C# プロシージャー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ーの外部 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ー・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 5. 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ーのいずれかに 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、プロシージャー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 文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ー・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。各プロシージャーの CREATE PROCEDURE ステートメントの EXTERNAL 文節でその情報を指定して、 DB2 がアセンブリーと CLR プロシージャーのクラスを見つけられるようにする必要があるからです。

例 1: C# のパラメーター・スタイル GENERAL 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 GENERAL 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 GENERAL 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、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 6. 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

```
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
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenProc.dll:bizLogic.empOps!SetEmpBonusGEN' ;
```

```
 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 GENERAL WITH NULLS 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 GENERAL WITH NULLS 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。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 7. 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
FENCED
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenProc.dll:bizLogic.empOps!SetEmpBonusGENNULL'
;
```

表 7. 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 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;
              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 8. 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
FENCED
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenProc.dll:bizLogic.empOps!SetEmpBonusSQL' ;
```

表 8. 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 *(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 のプロシージャー:

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# プロシージャーの 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 のプロシージャーの C# コード

このプロシージャーは、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、プロシージャーの戻り時に特定の照会結果セットの DB2DataReader をオープンにしておきます。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 9. 結果セットを戻す C# プロシージャー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 のプロシージャー:

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ーの 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 のプロシージャーの 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 文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。このプロシージャーは、 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 10. dbinfo 構造にアクセスする C# プロシージャー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
DBINFO
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenProc.dll:bizLogic.empOps!ReturnDbName'
;
```

```
public static void ReturnDbName(out string dbName,
                                 out Int16  dbNameNullInd,
                                 ref string sqlStateate,
                                     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
   // sqlStateate and an error message string text.
   // For example:
   //
   //   sqlStateate = "ABCDE";
   //   sqlMessageText = "A user-defined error has occured"
   //
   //  DB2 returns the above values to the client in the
   //  SQLCA structure.  The values are used to generate a
   //  standard DB2 sqlStateate error.
}
```

例 6: PROGRAM TYPE MAIN スタイルの C# プロシージャー:

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ーの 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 文節に 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 11. メインプログラム・スタイルで C# プロシージャー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 'gwenProc.dll:bizLogic.empOps!main' ;
```

表 11. メインプログラム・スタイルで C# プロシージ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 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
       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();
   }
}
```

関連概念:

- 122 ページの『共通言語ランタイム (CLR) ルーチン』

- 3 ページの『アプリケーション開発におけるルーチン』

関連タスク:

- 158 ページの『C# の CLR ユーザー定義関数の例』
- 123 ページの『CLR ルーチンの作成』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『Common Language Runtime (CLR) .NET ルーチンの構築』

関連サンプル:

- 『SpCreate.db2 -- Creates the external procedures implemented in spserver.cs』
- 『SpServer.cs -- C# external code implementation of procedures created in spcat.db2』
- 『SpCreate.db2 -- Creates the external procedures implemented in spserver.vb』
- 『SpServer.vb -- VB.NET implementation of procedures created in SpCat.db2』

Visual Basic の CLR プロシージャーの例

プロシージャー (ストアード・プロシージャーともいう) 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 プロシージャー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 プロシージャー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。 `Visual Basic` の CLR UDF の例については、以下を参照してください。

- 164 ページの『Visual Basic の CLR ユーザー定義関数の例』

前提条件:

CLR プロシージャー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。

- 122 ページの『共通言語ランタイム (CLR) ルーチン』
- 123 ページの『CLR ルーチンの作成』
- 3 ページの『アプリケーション開発におけるルーチン』
- 共通言語ランタイム (CLR) .NET ルーチンのビルド (Building common language runtime (CLR) .NET routines)

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。

手順:

独自の Visual Basic CLR プロシージャーを作成するときには、以下の例を参考にしてください。

- 148 ページの『Visual Basic 外部コード・ファイル』
- 148 ページの『例 1: Visual Basic のパラメーター・スタイル GENERAL のプロシージャー』

- 150 ページの『例 2: Visual Basic のパラメーター・スタイル GENERAL WITH NULLS のプロシージャー』
- 151 ページの『例 3: Visual Basic のパラメーター・スタイル SQL のプロシージャー』
- 153 ページの『例 4: 結果セットを戻す Visual Basic のプロシージャー』
- 154 ページの『例 5: dbinfo 構造にアクセスする Visual Basic のプロシージャー』
- 155 ページの『例 6: PROGRAM TYPE MAIN スタイルの Visual Basic プロシージャー』

Visual Basic 外部コード・ファイル:

以下の例では、Visual Basic プロシージャー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ーの外部 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ー・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 12. 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ーのいずれかに 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、プロシージャー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 文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ー・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。各プロシージャーの CREATE PROCEDURE ステートメントの EXTERNAL 文節でその情報を指定して、 DB2 がアセンブリーと CLR プロシージャーのクラスを見つけられるようにする必要があるからです。

例 1: Visual Basic のパラメーター・スタイル GENERAL 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 GENERAL 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 GENERAL のプロシージャーの Visual Basic 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、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 13. Visual Basic 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

```
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'
```

```
  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 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)

           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 GENERAL WITH NULLS 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 GENERAL WITH NULLS のプロシージャーの Visual Basic 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。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 14. 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'
```

表 14. 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 (続き)

```
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 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)

        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 Visual Basic 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 15. 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
FENCED
PROGRAM TYPE SUB
EXTERNAL NAME 'gwenVbProc.dll:bizLogic.empOps!SetEmpBonusSQL'
```

表 15. 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 プロシージャー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 のプロシージャー:

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 プロシージャーの 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 のプロシージャーの Visual Basic コード

このプロシージャーは、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、プロシージャーの戻り時に特定の照会結果セットの DB2DataReader をオープンにしておきます。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 16. 結果セットを戻す Visual Basic プロシージャー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 のプロシージャー:

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ーの 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 のプロシージャーの 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 文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。このプロシージャーは、 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 17. dbinfo 構造にアクセスする Visual Basic プロシージャー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 occured"
    '
    ' 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 プロシージャー:

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ーの 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 文節に 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 18. メインプログラム・スタイルで *Visual Basic* プロシージャー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 18. メインプログラム・スタイルで Visual Basic プロシージャー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
```

関連概念:

- 122 ページの『共通言語ランタイム (CLR) ルーチン』
- 3 ページの『アプリケーション開発におけるルーチン』

関連タスク:

- 164 ページの『Visual Basic の CLR ユーザー定義関数の例』
- 123 ページの『CLR ルーチンの作成』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『Common Language Runtime (CLR) .NET ルーチンの構築』

C# の CLR ユーザー定義関数の例

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 プロシージャーの例については、以下を参照してください。

- 136 ページの『C# の CLR プロシージャーの例』

前提条件:

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。

- 122 ページの『共通言語ランタイム (CLR) ルーチン』
- 123 ページの『CLR ルーチンの作成』
- 15 ページの『ユーザー定義のスカラー関数』
- 17 ページの『ユーザー定義のスカラー関数』
- 共通言語ランタイム (CLR) .NET ルーチンのビルド (Building common language runtime (CLR) .NET routines)

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。

手順:

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』
- 159 ページの『例 1: C# のパラメーター・スタイル SQL の表関数』
- 162 ページの『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 19. 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 文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ます。

表 20. 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
SCRATCHPAD 10
FINAL CALL
DISALLOW PARALLEL
NO DBINFO
```

表 20. 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 20. 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 つのカウント値を戻します。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を維持できるようになっています。

たとえば、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戻す代わりに、データの形式を既存の列に合わせることもできます。たとえば、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 21. 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
FENCED
SCRATCHPAD 10
FINAL CALL
VARIANT
NO SQL
EXTERNAL NAME 'gwenUDF.dll:bizLogic.empOps!CountUp' ;
```

表 21. 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;
      }
   }
}
```

関連概念:

- 122 ページの『共通言語ランタイム (CLR) ルーチン』
- 15 ページの『ユーザー定義のスカラー関数』
- 17 ページの『ユーザー定義のスカラー関数』

関連タスク:

- 136 ページの『C# の CLR プロシージャーの例』
- 123 ページの『CLR ルーチンの作成』

- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『Common Language Runtime (CLR) .NET ルーチンの構築』

関連サンプル:

- 『SpCreate.db2 -- Creates the external procedures implemented in spserver.cs』
- 『SpServer.cs -- C# external code implementation of procedures created in spcat.db2』
- 『SpCreate.db2 -- Creates the external procedures implemented in spserver.vb』
- 『SpServer.vb -- VB.NET implementation of procedures created in SpCat.db2』

Visual Basic の CLR ユーザー定義関数の例

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 プロシージャーの例については、以下を参照してください。

- 147 ページの『Visual Basic の CLR プロシージャーの例』

前提条件:

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。

- 122 ページの『共通言語ランタイム (CLR) ルーチン』
- 123 ページの『CLR ルーチンの作成』
- 15 ページの『ユーザー定義のスカラー関数』
- 17 ページの『ユーザー定義のスカラー関数』
- 共通言語ランタイム (CLR) .NET ルーチンのビルド (Building common language runtime (CLR) .NET routines)

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。

手順:

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』
- 165 ページの『例 1: Visual Basic のパラメーター・スタイル SQL の表関数』
- 168 ページの『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 22. 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 文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ます。

表 23. *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
```

```
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 23. 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 つのカウント値を戻します。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を維持できるようになっています。

たとえば、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戻す代わりに、データの形式を既存の列に合わせることもできます。たとえば、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 24. 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
FENCED
SCRATCHPAD 10
FINAL CALL
VARIANT
NO SQL
EXTERNAL NAME 'gwenUDF.dll:bizLogic.empOps!CountUp';
```

表 24. 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
```

関連概念:

- 122 ページの『共通言語ランタイム (CLR) ルーチン』
- 15 ページの『ユーザー定義のスカラー関数』
- 17 ページの『ユーザー定義のスカラー関数』

関連タスク:

- 147 ページの『Visual Basic の CLR プロシージャーの例』
- 123 ページの『CLR ルーチンの作成』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『Common Language Runtime (CLR) .NET ルーチンの構築』

C/C++ ルーチン

以下の項では、C または C++ のルーチンを作成する方法を説明しています。

C/C++ ルーチン

ルーチンを C または C++ で開発する場合、 CREATE ステートメント内で PARAMETER STYLE SQL 文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。また、sqludf.h 組み込みファイルを使用することもお勧めします。これには、UDF とストアード・プロシージャーを作成するのに役に立つ構造、定義、および値が用意されています。

C/C++ UDF およびメソッド:

PARAMETER STYLE SQL の UDF とメソッドの C/C++ シグニチャーは以下のフォーマットに準拠します。

```
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/C++ 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。これらは、sqlsystm.h 内で宣言されます。このマクロは、 C/C++ ルーチンを作成するときに必要です。

function-name
: C/C++ 関数の名前。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 文節内のライブラリー名を使用して指定されます。 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 文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。

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。

SQL-argument-inds
: 各 SQL-argument ごとに標識変数が 1 つずつあります。その各標識を、sqludf.h の SQLUDF_NULLIND タイプ定義を使用して定義します。

SQLUDF_TRAIL_ARGS
: sqludf.h 内で定義されていて、ルーチンの末尾引き数を定義するマクロ。これは、SQLSTATE を指すポインター、完全修飾関数名、関数の固有名、およびメッセージ・テキストなどで構成されます。 SCRATCHPAD および FINAL CALL を使用して登録された UDF の場合は、SQLUDF_TAIL_ARGS_ALL マクロを使用します。これには、SQLUDF_TRAIL_ARGS に組み込まれている引き数に加えて、スクラッチパッドを指すポインターと呼び出しタイプが入っています。

以下に、2 つの入力引き数の積を戻す C/C++ の 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 )
{
  *outProduct = (*in1) * (*in2);

  return (0);
}
```

この UDF の対応する CREATE FUNCTION ステートメントは次のとおりです。

```
CREATE FUNCTION product( DOUBLE in1, DOUBLE in2 )
  RETURNS DOUBLE
  LANGUAGE c
  PARAMETER STYLE sql
  NO SQL
  FENCED THREADSAFE
  DETERMINISTIC
  RETURNS NULL ON NULL INPUT
  NO EXTERNAL ACTION
  EXTERNAL NAME 'c_rtns!product'
```

上記のステートメントでは、 C/C++ 関数は c_rtns というライブラリー内にあることが前提になっています。

C/C++ ストアード・プロシージャー:

PARAMETER STYLE SQL のストアード・プロシージャーの C/C++ シグニチャーは以下のフォーマットに準拠します。

```
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/C++ 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。これらは、sqlsystm.h 内で宣言されます。このマクロは、 C/C++ ルーチンを作成するときに必要です。

function-name
: C/C++ 関数の名前。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 文節内のライブラリー名を使用して指定されます。 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 文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・ポイントを外部 ″C″ と定義しなければなりません。

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。

sqlstate
これをルーチンで使用して、警告またはエラー条件を出すことができます。

routine-name
修飾関数名。この値は DB2® によって生成されて、 schema.routine の形式でルーチンに渡されます。この値は、 SYSCAT.ROUTINES ビューの ROUTINESCHEMA および ROUTINENAME 列に対応します。

specific-name
特定の関数名。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 SPECIFICNAME 列に対応します。

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で使用されます。

注: 『C/C++ の UDF およびメソッド』の項に述べられている関数シグニチャーと違って、 C/C++ ストアード・プロシージャーの関数シグニチャーは、 sqludf.h 内で宣言されたマクロを使用しません。ただし、 sqludf.h マクロを使用して C/C++ ストアード・プロシージャーを作成することは可能です。それとは逆に、sqludf.h マクロを使用しないで C/C++ の UDF およびメソッドを作成することも可能です。

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 C/C++ のストアード・プロシージャーの例を示します。

```
SQL_API_RC SQL_API_FN cstp ( sqlint16 *inParm,
                             double *outParm,
                             sqlint16 *inParmNullInd,
                             sqlint16 *outParmNullInd,
                             char sqlst[6],
                             char qualname[28],
                             char specname[19],
                             char diagmsg[71] )
{
  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;

  EXEC SQL OPEN cur1;

  return (0);
}
```

このストアード・プロシージャーの対応する 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_rtns!cstp'
```

上記のステートメントでは、 C/C++ 関数は c_rtns というライブラリー内にあることが前提になっています。

注: C または C++ ルーチンを 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 文節でルーチン本体を指示するときは以下のように気を付けてください。ルーチン本体を指示するのに絶対パス ID を使用する場合、 .dll 拡張子を付加する必要があります。たとえば、次のようになります。

```
CREATE PROCEDURE getSalary( IN inParm INT, OUT outParm INT )
  LANGUAGE c
  PARAMETER STYLE sql
  DYNAMIC RESULT SETS 1
  FENCED THREADSAFE
  RETURNS NULL ON NULL INPUT
  EXTERNAL NAME 'd:¥mylib¥myfunc.dll'
```

関連概念:

- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『データベース・マネージャー・インスタンス』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『ルーチン用の AIX エクスポート・ファイル』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『AIX ルーチンと CREATE ステートメント』
- 174 ページの『C/C++ ルーチン用の組み込みファイル (sqludf.h)』
- 177 ページの『C/C++ ルーチンでの SQL データ型処理』

関連タスク:

- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『UNIX C ルーチンの構築』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『UNIX C++ ルーチンの構築』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『Windows での C/C++ ルーチンの構築』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE TYPE (構造化) ステートメント』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『C サンプル』
- 101 ページの『C/C++、OLE、COBOL で書かれたルーチンに引き数を渡すときの構文』

関連サンプル:

- 『spserver.c -- Definition of various types of stored procedures』
- 『udfcli.c -- How to work with different types of user-defined functions (UDFs)』
- 『spserver.sqC -- Definition of various types of stored procedures (C++)』
- 『udfemsrv.sqC -- Call a variety of types of embedded SQL user-defined functions. (C++)』
- 『udfemsrv.sqc -- Call a variety of types of embedded SQL user-defined functions. (C)』

C/C++ ルーチン用の組み込みファイル (sqludf.h)

sqludf.h 組み込みファイルには、ルーチンを作成するのに役立つ構造、定義、および値が入っています。このファイルの名前には udf が入っていますが、(履歴上の理由で) それはストアード・プロシージャーやメソッドにも便利です。ルーチンのコンパイル時には、このファイルがあるディレクトリーを参照する必要があります。そのディレクトリーは sqllib/include です。

sqludf.h 組み込みファイルは自己記述性です。以下にその内容について簡単に要約します。

1. 受け渡された引き数に対する構造定義。引き数の構造は次のとおりです。
   - VARCHAR FOR BIT DATA 引き数とその結果
   - LONG VARCHAR (FOR BIT DATA を持つ、または持たない) 引き数とその結果
   - LONG VARGRAPHIC 引き数とその結果
   - すべての LOB タイプ、SQL 引き数とその結果
   - スクラッチパッド
   - dbinfo 構造
2. すべての SQL データ型に対する C 言語型定義。 SQL 引き数に対応するルーチン引き数と、データ型を持つ結果を定義するために使用されます。データ型は、SQLUDF_x および SQLUDF_x_FBD という名前で定義されます。この場合の x とは SQL のデータ型名であり、FBD は For Bit Data を表します。

   また、AS LOCATOR 文節を使用して定義される引き数や結果の 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。
3. *スクラッチパッド* および*呼び出しタイプ* 引き数に対する C 言語型定義。*呼び出しタイプ* 引き数の enum 型定義を使用します。
4. 標準*後書き* 引き数を定義するマクロ。*スクラッチパッド* と*呼び出しタイプ* 引き数を含むものと含まないものがあります。これは、関数定義の中の SCRATCHPAD と FINAL CALL キーワードの有無と一致します。これらは、*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 呼び出し引き数です。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。
5. SQL 引き数が NULL であるかどうかをテストするマクロ。

それに対応する sqludf.cbl という COBOL 用の組み込みファイルが存在します。このファイルには、スクラッチパッドと dbinfo 構造の定義だけが組み込まれています。

関連概念:

- 177 ページの『C/C++ ルーチンでの SQL データ型処理』
- 170 ページの『C/C++ ルーチン』

関連資料:

- 101 ページの『C/C++、OLE、COBOL で書かれたルーチンに引き数を渡すときの構文』
- 175 ページの『C/C++ でサポートされている SQL データ型』

C/C++ でサポートされている SQL データ型

次の表は、ルーチンの SQL データ型および C データ型同士の間でサポートされるマッピングをリストしています。各 C/C++ データ型には、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 25. C/C++ 宣言にマップされた SQL データ型

SQL 列名	C/C++ データ型	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(*n*)。ただし 1<=n<=24。	float SQLUDF_REAL	単精度浮動小数点
DOUBLE FLOAT FLOAT(*n*)。ただし 25<=n<=53。	double SQLUDF_DOUBLE	倍精度浮動小数点
DECIMAL(*p*, *s*)	サポートされていません。	10 進数を渡すには、パラメーターを DECIMAL からキャスト可能にデータ型 (たとえば CHAR または DOUBLE) および明示的に引き数をこのタイプにキャストするように定義します。
CHAR(*n*)	char[*n+1*] (n はデータを保持するだけの十分な大きさ) 1<=*n*<=254 SQLUDF_CHAR	固定長、NULL 終了文字ストリング
CHAR(*n*) FOR BIT DATA	char[*n+1*] (n はデータを保持するだけの十分な大きさ) 1<=*n*<=254 SQLUDF_CHAR	固定長、NULL 終了文字ストリング

表 25. C/C++ 宣言にマップされた SQL データ型 (続き)

SQL 列名	C/C++ データ型	SQL 列タイプ記述
VARCHAR(*n*)	char[*n+1*] (n はデータを保持するだけの十分な大きさ) 1<=*n*<=32 672 SQLUDF_VARCHAR	NULL 終了可変長ストリング

VARCHAR(*n*) FOR BIT DATA	struct {
	sqluint16 length;
	char[*n*]
	}
	
	1<=*n*<=32 672
	
	SQLUDF_VARCHAR_FBD
		NULL 終了可変長文字ストリングでない

LONG VARCHAR	struct {
	sqluint16 length;
	char[*n*]
	}
	
	1<=*n*<=32 700
	
	SQLUDF_LONG
		NULL 終了可変長文字ストリングでない

CLOB(*n*)	struct {
	sqluint32 length;
	char data[*n*];
	}
	
	1<=*n*<=2 147 483 647
	
	SQLUDF_CLOB
		4 バイト・ストリング長標識の NULL 終了可変長文字ストリングでない

BLOB(*n*)	struct {
	sqluint32 length;
	char data[n];
	}
	
	1<=*n*<=2 147 483 647
	
	SQLUDF_BLOB
		4 バイト・ストリング長標識の NULL 終了可変長バイナリー・ストリングでない

DATE	char[11] SQLUDF_DATE	以下のフォーマットの NULL 終了文字ストリング。 `yyyy-mm-dd`
TIME	char[9] SQLUDF_TIME	以下のフォーマットの NULL 終了文字ストリング。 `hh.mm.ss`
TIMESTAMP	char[27] SQLUDF_STAMP	以下のフォーマットの NULL 終了文字ストリング。 `yyyy-mm-dd-hh.mm.ss.nnnnnn`
LOB LOCATOR	sqluint32 SQLUDF_LOCATOR	32 ビットの符号付き整数

表 25. C/C++ 宣言にマップされた SQL データ型 (続き)

SQL 列名	C/C++ データ型	SQL 列タイプ記述

DATALINK	struct {
	sqluint32 version;
	char linktype[4];
	sqluint32 url_length;
	sqluint32 comment_length;
	char reserve2[8];
	char url_plus_comment[230];
	}
	
	SQLUDF_DATALINK	

注: 以下のデータ型は、WCHARTYPE NOCONVERT オプションを使用してプリコンパイルする場合の DBCS または EUC 環境でのみ使用できる。		
GRAPHIC(*n*)	sqldbchar[*n+1*] (n はデータを保持するだけの十分な大きさ) 1<=*n*<=127 SQLUDF_GRAPH	固定長、NULL 終了 2 バイト文字ストリング
VARGRAPHIC(*n*)	sqldbchar[*n+1*] (n はデータを保持するだけの十分な大きさ) 1<=*n*<=16 336 SQLUDF_GRAPH	NULL 終了、可変長 2 バイト文字ストリングでない

LONG VARGRAPHIC	struct {
	sqluint16 length;
	sqldbchar[*n*]
	}
	
	1<=*n*<=16 350
	
	SQLUDF_LONGVARG
		NULL 終了、可変長 2 バイト文字ストリングでない

DBCLOB(*n*)	struct {
	sqluint32 length;
	sqldbchar data[n];
	}
	
	1<=*n*<=1 073 741 823
	
	SQLUDF_DBCLOB
		4 バイト・ストリング長標識の NULL 終了可変長文字ストリングでない

関連概念:

- 174 ページの『C/C++ ルーチン用の組み込みファイル (sqludf.h)』
- 177 ページの『C/C++ ルーチンでの SQL データ型処理』
- 170 ページの『C/C++ ルーチン』

C/C++ ルーチンでの SQL データ型処理

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、それに対応する引き数を 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き数は、該当するデータ型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、さまざまなデータ型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。たとえば、BIGINT では、SQLUDF_BIGINT データ型を使用すれば、コンパイラーが異なっていても、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、引き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型です。適切なフォーマットで値を受け取るには、引き数のデータ型からのプロモーションが必要なことがあります。 DB2® は、引き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型を指定すると、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 文節で指定するデータ型です。 CAST FROM 文節がない場合は、RETURNS 文節で指定されるデータ型がフォーマットを定義します。

以下の例での CAST FROM 文節は、ルーチン本体が SMALLINT を戻し、 DB2 がその値を関数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 データ型は RETURNS データ型に対してキャスト*可能* でなければならないため、任意に他のデータ型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き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 の使用を検討してみてください。それは、DB2 は INTEGER 引き数を SMALLINT にはプロモートしないからです。たとえば、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 sqlsystm.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 sqlsystm.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 <= *n* <= 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 <= *n* <= 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 進数の値を渡したい場合は、パラメーターを DECIMAL からキャスト可能なデータ型 (CHAR や DOUBLE など) に定義して、引き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 進値引き数を明示的に 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 選択ステートメントで UDF WEEKLY_PAY を呼び出します。

```
SELECT WEEKLY_PAY (WAGE, HOURS, ...) ...;
```

DECIMAL 引き数は 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 引き数を持つ 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 引き数は 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 */
  ```

  CHAR(n) パラメーターの場合、 DB2 は *n* バイトのデータをバッファーに移動し、 *n+1* 桁のバイトを NULL 終了符 (X'00') に設定します。 FOR BIT DATA として指定されていない RETURNS CHAR(n) 値や、ストアード・プロシージャーの出力パラメーターの場合、 DB2 は CHAR 値の最初の *n* バイト内で NULL 終止符を探します。 NULL 終止符が見つかった場合、 DB2 は *n* バイトまでの残りのバイトを ASCII ブランクで埋め込みます。 FOR BIT DATA として指定されている RETURNS CHAR(n) 値や、ストアード・プロシージャーの出力パラメーターの場合、 DB2 は、*n* バイト内にストリング NULL 終止符があるかどうかに関係なく、最初の *n* バイトをコピーします。ストリング NULL 終止符は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は正常なことですが、この種の関数の多くは、ストリング引き数を区切る 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 を使用するようにしてください。これは、DB2 は VARCHAR 引き数を 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。たとえば、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 文節の場合、ルーチンに渡される長さはバッファーの長さです。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・プロシージャー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 (たとえば、 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 DATE          */
  char     result[9];     /* also acceptable */
  ```

- TIMESTAMP

  正しい例。 C では SQLUDF_STAMP または CHAR(26) と表されます。つまり char...[27] となります。タイム・スタンプの値は、常に以下の書式で渡されます。

  ```
  yyyy-mm-dd-hh.mm.ss.nnnnnn
  ```

  以下に例を示します。

  ```
  char     arg1[27];      /* example for TIMESTAMP  */
  char    *result;        /* also acceptable */
  ```

- GRAPHIC(n)

  正しい例。 C で SQLUDF_GRAPH または sqldbchar[n+1] として表します。 (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 を使用するようにしてください。これは、DB2 が VARGRAPHIC 引き数を GRAPHIC にプロモートしないためです。たとえば、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 SQLUDF_GRAPH または sqldbchar[n+1] として表します。 (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・プロシージャー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 */
};
```

wchar_t が 2 バイトの長さと定義されているオペレーティング・システムでは、上記の例の 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 文節またはストアード・プロシージャー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ーの長さです。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 文節またはストアード・プロシージャー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、バッファーの長さです。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 */
};
```

wchar_t が 2 バイトの長さと定義されているオペレーティング・システムでは、上記の例の 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 文節またはストアード・プロシージャー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ーの長さです。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) */
```

- 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;
```

- 構造化型

  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。構造化型パラメーターは、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。構造化型の結果は、TO SQL transform 関数のパラメーター・タイプで渡されます。

- DATALINK

  正しい例。 C では SQLUDF_DATALINK と表されます。そうでない場合、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されます。

```
     struct sqludf_datalink {
       sqluint32 version;
       char      linktype[4];
       sqluint32 url_length;
       sqluint32 comment_length;
       char      reserve2[8];
       char      url_plus_comment[230];
     }
```

関連概念:

- 312 ページの『Transform 関数と Transform グループ』
- 186 ページの『C/C++ ルーチンでの GRAPHIC ホスト変数』
- 174 ページの『C/C++ ルーチン用の組み込みファイル (sqludf.h)』
- 170 ページの『C/C++ ルーチン』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE ステートメント』
- 「*アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」の『C および C++ においてサポートされている SQL データ・タイプ』
- 175 ページの『C/C++ でサポートされている SQL データ型』

C/C++ ルーチンでの GRAPHIC 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変換は起こりません。 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・プロシージャーから送られてくる 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。

関連概念:

- 「*アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」の『C および C++ での WCHARTYPE プリコンパイラー・オプション』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『WCHARTYPE CONVERT プリコンパイル・オプション』

関連資料:

- 「*コマンド・リファレンス*」の『PRECOMPILE コマンド』

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き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、引き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。たとえば UNIX 上では、この行は次のようになります。

```
myfunc__FPlT1PsT3PcN35|     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 文節内に指定する必要があります。たとえば、次のようにします。

```
CREATE FUNCTION myfunco(...) RETURNS...
       ...
       EXTERNAL NAME '/whatever/path/myprog!myfunc__FPlT1PsT3PcN35'
       ...
```

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、コンパイラー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ので、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++ ストアード・プロシージャー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。

関連概念:

- 99 ページの『外部ルーチン用のパラメーター・スタイル』
- 170 ページの『C/C++ ルーチン』
- 59 ページの『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』

Java ルーチン

以下の項では、Java ルーチンを作成する方法を説明しています。

Java ルーチン

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 文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。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。

UDF およびメソッドの一部の機能は、PARAMETER STYLE JAVA ではインプリメントできません。それらは次のとおりです。

- 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。

上記の機能を UDF またはメソッドにインプリメントする必要がある場合、PARAMETER STYLE DB2GENERAL を使用して、 C でルーチンを作成するか、または Java で作成します。このような特定の例以外にも、本書で Java ルーチンと言う場合は、PARAMETER STYLE JAVA の使用を前提としています。

Java UDF およびメソッド:

PARAMETER STYLE JAVA の UDF と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型。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 文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 つの入力引き数の積を戻す Java の UDF の例を示します。

```
public static double product( double in1, double in2 ) throws SQLException
{
  return in1 * in2;
}
```

この UDF の対応する CREATE FUNCTION ステートメントは次のとおりです。

```
CREATE FUNCTION product( DOUBLE in1, DOUBLE in2 )
  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。

Java ストアード・プロシージャー:

PARAMETER STYLE JAVA ストアード・プロシージャー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 文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 文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、 ResultSet[] タイプのパラメーターを Java メソッド・シグニチャー内で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・プロシージャーの例を示します。

```
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 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 の機能を呼び出すことは可能です。それには、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す必要があります。

関連概念:

- 363 ページの『DB2GENERAL ルーチン』
- 68 ページの『Java の表関数実行モデル』

関連タスク:

- 198 ページの『Java ストアード・プロシージャーのデバッグ』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『JDBC ルーチンの作成』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『JDBC ルーチンの作成』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE TYPE (構造化) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE METHOD ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION (外部スカラー) ステートメント』
- 193 ページの『Java でサポートされている SQL データ型』
- 196 ページの『データベース・サーバーでの JAR ファイル管理』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『JDBC サンプル』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『SQLJ サンプル』
- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE (外部) ステートメント』

関連サンプル:

- 『SpServer.java -- Provide a variety of types of stored procedures to be called from (JDBC)』
- 『UDFjsrv.java -- Provide UDFs to be called by UDFjcli.java (JDBC)』
- 『UDFsqlsv.java -- Provide UDFs to be called by UDFsqlcl.java (JDBC)』
- 『UDFsrv.java -- Provide UDFs to be called by UDFcli.java (JDBC)』
- 『SpServer.sqlj -- Provide a variety of types of stored procedures to be called from (SQLj)』

- 『UDFjsrv.java -- Provide UDFs to be called by UDFjcli.sqlj (SQLj)』
- 『UDFsrv.java -- Provide UDFs to be called by UDFcli.sqlj (SQLj)』

Java でサポートされている SQL データ型

JDBC 仕様のデータ型マッピングに基づいた、各 SQL データ型に等しい Java の値を次の表に示します。 JDBC ドライバーは、アプリケーションとデータベースとの間で交換されるデータを、以下のマッピング・スキーマを使用して変換します。これらのマッピングは、 Java アプリケーションと PARAMETER STYLE JAVA プロシージャーと UDF で使用します。

注: DB2 がサポートするどのプログラム言語にも、 DATALINK データ型に対するホスト変数サポートはありません。

表 26. Java 宣言にマップされる SQL データ型

SQL 列名	Java データ型	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(*p*,*s*) (484 または 485)	java.math.BigDecimal	パック 10 進数
CHAR(*n*) (452 または 453)	java.lang.String	長さが *n* の固定長文字ストリング (*n* の範囲は 1 ～ 254 まで)
CHAR(*n*) FOR BIT DATA	バイト[]	長さが *n* の固定長文字ストリング (*n* の範囲は 1 ～ 254 まで)
VARCHAR(*n*) (448 または 449)	java.lang.String	可変長文字ストリング
VARCHAR(*n*) FOR BIT DATA	バイト[]	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	バイト[]	long 可変長文字ストリング
BLOB(*n*) (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(*n*) (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(*n*) (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 26. Java 宣言にマップされる SQL データ型 (続き)

SQL 列名	Java データ型	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	26 バイトの文字ストリング
GRAPHIC(*n*) (468 または 469)	java.lang.String	固定長 2 バイト文字ストリング
VARGRAPHIC(*n*) (464 または 465)	java.lang.String	2 バイトのストリング長標識を持つ、NULL 終了可変 2 バイト文字以外のストリング
LONGVARGRAPHIC (472 または 473)	java.lang.String	2 バイトのストリング長標識を持つ、NULL 終了可変 2 バイト文字以外のストリング

注:

1. DB2 UDB バージョン 8.1 クライアントから DB2 UDB 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。

Java クラスの配置場所

ストアード・プロシージャーや UDF のために個々の Java™ クラス・ファイルを使用することも、クラス・ファイルを JAR ファイルにまとめて、その JAR ファイルをデータベースにインストールすることもできます。 JAR ファイルを使用する場合の詳細については、 Java 関数とストアード・プロシージャーの登録に関する説明を参照してください。

注: Java ルーチン・クラス・ファイルを更新または置換した場合は、 CALL SQLJ.REFRESH_CLASSES() ステートメントを発行して、 DB2® で新しいクラスをロードできるようにする必要があります。 CALL SQLJ.REFRESH_CLASSES() の詳細については、ルーチンの Java クラスを更新する方法に関する説明を参照してください。

DB2 が Java 言語ルーチン (ストアード・プロシージャー、UDF、メソッド) を検出して使用するには、次のようにして対応するクラス・ファイルを保管する必要があります。

Unix および Windows® オペレーティング・システム

CLASSPATH 変数で指定した任意のディレクトリー・パス。 DB2 ルーチンに関連した Java クラス・ファイルは、 *function* ディレクトリー (`/u/$DB2INSTANCE/sqllib/function`) に保管することをお勧めします。 `/u/$DB2INSTANCE` は、現在アクティブになっているデータベース・マネージャーに関連付けられているディレクトリーです。

DB2 が呼び出す JVM は、 CLASSPATH 環境変数に基づいて Java ファイルを検出します。 DB2 は、 function ディレクトリーと sqllib/java/db2java.zip を自動的に CLASSPATH 設定の先頭に追加するので、この作業を手動で行う必要はありません。 DB2 ルーチンに関連した Java 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。 unfenced ルーチンに関連したクラスは、 /u/*$DB2INSTANCE*/sqllib/function/unfenced サブディレクトリーに保管してください。

DB2 が JVM を検出できるように環境を設定するには、 *jdk_path* 構成パラメーターを設定するか、デフォルトの値を使用します。アプリケーションのヒープ・サイズを増やすために、 *java_heap_sz* 構成パラメーターの設定が必要な場合もあります。

注: 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。たとえば、Linux 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 sqllib/function/ibm/tests に保管します。

関連タスク:

- 195 ページの『実行時の Java ルーチン (ストアード・プロシージャー、UDF、およびメソッド) の更新』

関連資料:

- 「*管理ガイド: パフォーマンス*」の『java_heap_sz - 「Java インタープリター最大ヒープ・サイズ」構成パラメーター』
- 「*管理ガイド: パフォーマンス*」の『jdk_path - 「Java Development Kit インストール・パス」構成パラメーター』

実行時の Java ルーチン (ストアード・プロシージャー、UDF、およびメソッド) の更新

手順:

Java ルーチン・クラスを更新する場合は、 CALL SQLJ.REFRESH_CLASSES() ステートメントも発行して、 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。 CALL SQLJ.REFRESH_CLASSES() ステートメントは、 FENCED ルーチンにのみ適用されます。 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。

注: NOT FENCED ルーチンの場合は、データベース・マネージャーをいったん停止してから再始動しないと更新ができません。

関連概念:

- 189 ページの『Java ルーチン』

関連資料:

- 202 ページの『Java デバッグ表 DB2DBG.ROUTINE_DEBUG』
- 368 ページの『DB2GENERAL ルーチン用の Java クラス』

データベース・サーバーでの JAR ファイル管理

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、データベースにインストールした JAR ファイルに置かれているか、またはオペレーティング・システムの正しい CLASSPATH パスに置かれていなければなりません。 DB2 クラス・ローダーは、CLASSPATH 内でクラスと JAR ファイルを探索して、指定名の付いた最初に見つかったクラスを取り込みます。

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・プロシージャーを使用します。

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
- SYSADM または 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・プロシージャー 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・プロシージャー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* でのユニーク値を指定する必要があります。

関連概念:

- 194 ページの『Java クラスの配置場所』
- 189 ページの『Java ルーチン』
- 30 ページの『ライブラリーおよびクラスの管理に関する考慮事項』

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。たとえば、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、並行スレッドによる予期しない干渉を避ける、独自でユニークの ConnectionContext (および基礎となる JDBC 接続) を作成します。

関連概念:

- 189 ページの『Java ルーチン』

- 「アプリケーション開発ガイド クライアント・アプリケーションのプログラミング」の『SQLJ アプリケーション作成の基本ステップ』
- 「アプリケーション開発ガイド クライアント・アプリケーションのプログラミング」の『SQLJ アプリケーションにおける SQL ステートメント』

Java のストアード・プロシージャーのデバッグ

以下の項では、Java ストアード・プロシージャーをデバッグする方法を説明しています。

Java ストアード・プロシージャーのデバッグ

DB2 には、 AIX、Linux、Solaris 操作環境、Windows NT、または Windows 2000 サーバー上でストアード・プロシージャーを実行する際に、 JDBC で作成されたストアード・プロシージャーを対話式にデバッグする機能があります。 Java ストアード・プロシージャーをデバッグするには、 DB2 デベロップメント・センターを介するのが最も簡単な方法です。

DB2 での作業中に Java ストアード・プロシージャーのデバッグを使用可能にするには、 DB2 分散デバッガー 9.2 を適切に構成しておく必要があります。分散デバッガーは、DB2 UDB バージョン 8 のすべてのパッケージ・オプションに組み込まれています。分散デバッガーは DB2 UDB バージョン 8 にインストールされている標準の SDK 1.3.1 レベルで機能するよう構成されています。異なるレベルの SDK を使用する場合は、DB2 コマンド・プロンプトから以下のコマンドを実行して、 DB2 データベース・マネージャーの構成を更新する必要があります。

```
db2 update dbm cfg using jdk_path <jdk131 path>
```

手順:

Java のストアード・プロシージャーをデバッグするには、次のようにします。

1. デバッグの準備
2. デバッグ表への移入
3. デバッガーの呼び出し

関連タスク:

- 198 ページの『Java ストアード・プロシージャーのデバッグの準備』
- 200 ページの『デバッグ表への移入』
- 200 ページの『デバッグ・プログラムの呼び出し』
- 43 ページの『ルーチンのデバッグ』

Java ストアード・プロシージャーのデバッグの準備

Java ストアード・プロシージャーの対話式デバッグを準備するには、ストアード・プロシージャー、クライアント、およびサーバーを処理します。

手順:

Java ストアード・プロシージャーのデバッグを準備するには、次のようにします。

1. SDK の資料に従って、デバッグ・モードでストアード・プロシージャーをコンパイルする。

2. サーバーを準備する。

   ソース・コードがサーバー上にある場合には、 Java ソース・コード・ディレクトリーを組み込むように CLASSPATH 環境変数を設定するか、または『データベース・サーバーでの JAR ファイルの管理』の項に説明されているように、function ディレクトリーにソース・コードを保管します。

3. クライアント環境変数を設定する。

   ソース・コードがクライアントに保管されている場合には、ストアード・プロシージャーのソース・コードが置かれたディレクトリーに DB2_DBG_PATH 環境変数を設定します。

4. デバッグ・テーブルを作成する。

   デバッグ・プログラムを呼び出すのにデベロップメント・センターを使用しない場合には、次のコマンドでデバッグ表を作成します。

   ```
   db2 -tf sqllib/misc/db2debug.ddl
   ```

   注: パーティション・データベース環境では、デフォルトのデータベース・パーティション・グループは USERSPACE1 表スペースの IBMDEFAULTGROUP であり、その有効範囲はすべてのデータベース・パーティションです。パーティション・データベース環境でのストアード・プロシージャーのデバッグのパフォーマンスを改善するには、デバッグが発生する 1 つのコーディネーター・パーティションがなければならず、そのデータベース・パーティションだけを収容するデータベース・パーティション・グループを定義しなければなりません。

5. 以下のようにして、分散デバッガーを構成する。

   a. DOS コマンド・プロンプトから、DB2SET コマンド `db2set DB2ROUTINE_DEBUG=on` を入力する。

   b. UNIX オペレーティング・システムで操作中の場合は、以下のステップを完了する (Windows ユーザーは、このステップをスキップしてステップ c に進んでください)

      1) `mkdir sqllib/function/src`

      2) `chmod 777 sqllib/function/src`

      3) `chmod 777 /home/youruserid/.DbgProf` (9.2.3 などのバージョンが新しい分散デバッガー)

   c. コマンド・プロンプトから `db2start` コマンドを入力して、 DB2 を始動する (DB2 がすでに始動している場合は再始動する)。

   d. クライアント・デーモンを開始するために、コマンド・プロンプトから以下のコマンドを入力する。

      ```
      idebug -qdaemon -quiport=portno
      ```

      ここで *quiport* は、未使用の TCP/IP ポート番号です。値が指定されないと、デバッグ・プログラムはデフォルト・ポート番号として 8000 を使用します。

以上で、デバッグ表へのデータの移入の準備ができました。

関連概念:

- 194 ページの『Java クラスの配置場所』

関連タスク:

- 200 ページの『デバッグ表への移入』
- 200 ページの『デバッグ・プログラムの呼び出し』
- 43 ページの『ルーチンのデバッグ』

関連資料:

- 202 ページの『Java デバッグ表 DB2DBG.ROUTINE_DEBUG』
- 196 ページの『データベース・サーバーでの JAR ファイル管理』

デバッグ・プログラムの呼び出し

デバッグ・プログラムでは、ソース・コードの一通りの確認、変数の表示、およびソース・コード内でのブレークポイントの設定を行うことができます。

手順:

デバッグの準備とデバッグ表へのデータの移入が済んだら、デバッグしたいストアード・プロシージャーを呼び出します。そのアクションによって、デバッグ表に指定した IP アドレスを使用してクライアントでデバッグ・プログラムが呼び出されます。

デベロップメント・センターで Java ストアード・プロシージャーのデバッグを実行するには、まずウィザードを使用して新規 Java ストアード・プロシージャーを作成します。オプション・パネルから、**「デバッグを使用可能にする」**を選択します。

「デバッグを使用可能にする」オプションで、まだビルドしていない既存の Java ストアード・プロシージャーをデバッグするには、次のようにします。

1. 「ストアード・プロシージャー」フォルダーを右クリックして、**「デバッグ用ビルド」**を選択します。
2. ツールバーから**「実行/デバッグ (Run/Debug)」**アイコンを選択して、デバッグ・モードでストアード・プロシージャーを実行します。

分散デバッガーを操作するための詳細については、分散デバッガー製品内のオンライン・ヘルプを参照してください。

関連タスク:

- 198 ページの『Java ストアード・プロシージャーのデバッグの準備』
- 200 ページの『デバッグ表への移入』
- 43 ページの『ルーチンのデバッグ』

関連資料:

- 202 ページの『Java デバッグ表 DB2DBG.ROUTINE_DEBUG』

デバッグ表への移入

デバッグ表には、デバッグされるストアード・プロシージャー、およびデバッグされるクライアント / サーバー環境についての情報が含まれます。 DBA または、表に対する INSERT、 UPDATE、DELETE 特権のあるユーザーだけが、基本表 DB2DBG.ROUTINE_DEBUG の値を直接操作することができます。しかし、DBA が

さらに制限事項を追加しない限り、ユーザー・ビュー DB2DBG.ROUTINE_DEBUG_USER を介して、だれでも行を追加、更新、または削除できます。この項の続きでは、ユーザー・ビューを使用してその表にデータを移入することを前提とします。

手順:

Development Center を使用してデバッグを呼び出す場合には、デバッグ・プログラムを使用してデバッグ表へのデータの書き込みとその管理を行います。そうでない場合、指定されたストアード・プロシージャーのデバッグ・サポートを使用可能にするには、 CLP から以下のコマンドを発行します。

```
DB2 INSERT INTO db2dbg.routine_debug_user (AUTHID, TYPE,
   ROUTINE_SCHEMA, SPECIFICNAME, DEBUG_ON, CLIENT_IPADDR)
VALUES ('authid', 'S', 'schema', 'proc_name', 'Y', 'IP_num')
```

詳細は次のとおりです。

authid ストアード・プロシージャーのデバッグに使用されるユーザー名。つまり、データベースに接続するのに使用されるユーザー名。

schema ストアード・プロシージャーのスキーマ名。

proc_name
個々のストアード・プロシージャー名。これは、CREATE PROCEDURE コマンドで指定した固有名か、固有名を指定しなかった場合には、システムが生成した ID になります。

IP_num
ストアード・プロシージャーのデバッグに使用されるクライアントの IP アドレス。形式は *nnn.nnn.nnn.nnn* です。

たとえば、デバッグするクライアントの IP アドレスが 192.168.111.222 の場合に、ユーザー *USER1* がストアード・プロシージャー *MySchema.myProc* をデバッグできるようにするには、以下のコマンドをタイプします。

```
DB2 INSERT INTO db2dbg.routine_debug_user (AUTHID, TYPE,
       ROUTINE_SCHEMA, SPECIFICNAME, DEBUG_ON, CLIENT_IPADDR)
    VALUES ('USER1', 'S', 'MySchema', 'myProc', 'Y', '192.168.111.222')
```

ストアード・プロシージャーをドロップする場合、このデバッグ情報はデバッグ表から自動的には削除されません。存在していないストアード・プロシージャーのデバッグ情報があっても、ご使用のデータベースまたはインスタンスに悪影響を与えることはありません。ただし、ストアード・プロシージャーの再作成の場合は、旧デバッグ情報が原因で混乱を生じる可能性があります。デバッグ表と DB2 カタログの同期を保ちたい場合には、デバッグ情報を手動で削除する必要があります。

以上で、デバッグ・プログラムの呼び出しの準備ができました。

関連タスク:

- 198 ページの『Java ストアード・プロシージャーのデバッグの準備』
- 200 ページの『デバッグ・プログラムの呼び出し』
- 43 ページの『ルーチンのデバッグ』

関連資料:

- 202 ページの『Java デバッグ表 DB2DBG.ROUTINE_DEBUG』

Java デバッグ表 DB2DBG.ROUTINE_DEBUG

手動または Development Center のどちらを使用してデバッグ表を作成した場合でも、そのデバッグ表には DB2DBG.ROUTINE_DEBUG という名前が付けられ、定義は以下のようになります。

表 27. DB2DBG.ROUTINE_DEBUG 表定義

列名	データ型	属性	説明
AUTHID	VARCHAR(128)	NOT NULL, DEFAULT USER	このストアード・プロシージャーのデバッグが実行されるアプリケーション authid。これはデータベースへの接続で提供されたユーザー ID です。
TYPE	CHAR(1)	NOT NULL	有効な値: 'S' (Procedure)
ROUTINE_SCHEMA	VARCHAR(128)	NOT NULL	デバッグされるストアード・プロシージャーのスキーマ名。
SPECIFICNAME	VARCHAR(18)	NOT NULL	デバッグされるストアード・プロシージャーの固有名。
DEBUG_ON	CHAR(1)	NOT NULL, DEFAULT 'N'	有効な値: • **Y** - ストアード・プロシージャーのデバッグを使用可能にします。 • **N** - ストアード・プロシージャーのデバッグを使用禁止にします。これはデフォルトです。
CLIENT_IPADDR	VARCHAR(15)	NOT NULL	デバッグを実行するクライアントの IP アドレス。形式は *nnn.nnn.nnn.nnn*。
CLIENT_PORT	INTEGER	NOT NULL, DEFAULT 8000	デバッグ通信のポート。デフォルトは 8000。
DEBUG_STARTN	INTEGER	NOT NULL	使用されません。
DEBUG_STOPN	INTEGER	NOT NULL	使用されません。
この表の主キーは、AUTHID、TYPE、ROUTINE_SCHEMA、SPECIFICNAME です。			

DB2DBG.ROUTINE_DEBUG_USER ビューではこの表へのアクセスは、データベースに接続しているユーザーに所属する行にのみ限定されます。

関連タスク:

- 198 ページの『Java ストアード・プロシージャーのデバッグ』
- 198 ページの『Java ストアード・プロシージャーのデバッグの準備』
- 200 ページの『デバッグ表への移入』
- 200 ページの『デバッグ・プログラムの呼び出し』
- 43 ページの『ルーチンのデバッグ』

OLE オートメーション・ルーチン

以下の項では、OLE オートメーション・ルーチンを作成する方法を説明しています。

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特性と方式を公開できます。 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特性と方式を利用して、これらのオブジェクトを統合することができます。

これらの特性と方式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 Corporation: 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。

関連概念:

- 205 ページの『オブジェクト・インスタンスとスクラッチパッドに関する考慮事項および OLE ルーチン』
- 207 ページの『BASIC および C++ での OLE オートメーション・ルーチン』

関連タスク:

- 203 ページの『OLE オートメーション・ルーチンの作成』

関連資料:

- 206 ページの『OLE オートメーションでサポートされている SQL データ型』

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。

手順:

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)
```

関連概念:

- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『Visual Basic でのオブジェクトのリンクと埋め込み (OLE) オートメーション』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『Visual C++ でのオブジェクトのリンクと埋め込み (OLE) オートメーション』
- 203 ページの『OLE オートメーション・ルーチンの設計』

- 205 ページの『オブジェクト・インスタンスとスクラッチパッドに関する考慮事項および OLE ルーチン』
- 207 ページの『BASIC および C++ での OLE オートメーション・ルーチン』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE TYPE (構造化) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION (外部スカラー) ステートメント』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『オブジェクトのリンクと埋め込み (OLE) のサンプル』
- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE (外部) ステートメント』
- 206 ページの『OLE オートメーションでサポートされている SQL データ型』

オブジェクト・インスタンスとスクラッチパッドに関する考慮事項および OLE ルーチン

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 文節の場合、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。

関連概念:

- 203 ページの『OLE オートメーション・ルーチンの設計』
- 207 ページの『BASIC および C++ での OLE オートメーション・ルーチン』

関連タスク:

- 203 ページの『OLE オートメーション・ルーチンの作成』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE TYPE (構造化) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION (外部スカラー) ステートメント』

- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE (外部) ステートメント』
- 206 ページの『OLE オートメーションでサポートされている SQL データ型』

OLE オートメーションでサポートされている SQL データ型

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型と、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 28. SQL と OLE オートメーション・データ型のマッピング

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(*n*)	BSTR	「*OLE Automation Programmer's Reference*」で説明されている、長さフィールド付ストリング
VARCHAR(*n*)	BSTR	
LONG VARCHAR	BSTR	
CLOB(*n*)	BSTR	
GRAPHIC(*n*)	BSTR	*OLE Automation Programmer's Reference* で説明されている、長さフィールド付ストリング
VARGRAPHIC(*n*)	BSTR	
LONG GRAPHIC	BSTR	
DBCLOB(*n*)	BSTR	
CHAR(*n*)	SAFEARRAY[unsigned char]	8 バイト無符号データ項目の 1 ディメンション Byte() 配列。(SAFEARRAY については、 *OLE Automation Programmer's Reference* で解説されています。)
VARCHAR(*n*)	SAFEARRAY[unsigned char]	
LONG VARCHAR	SAFEARRAY[unsigned char]	
CHAR(*n*) FOR BIT DATA	SAFEARRAY[unsigned char]	
VARCHAR(*n*) FOR BIT DATA	SAFEARRAY[unsigned char]	
LONG VARCHAR FOR BIT DATA	SAFEARRAY[unsigned char]	
BLOB(*n*)	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、DATALINK、LOCATORS などの SQL タイプ、ブール (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)。

関連概念:

- 203 ページの『OLE オートメーション・ルーチンの設計』
- 205 ページの『オブジェクト・インスタンスとスクラッチパッドに関する考慮事項および OLE ルーチン』
- 207 ページの『BASIC および C++ での OLE オートメーション・ルーチン』

関連タスク:

- 203 ページの『OLE オートメーション・ルーチンの作成』

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++ のデータ型へのマッピングを示しています。

表 29. SQL および OLE データ型から BASIC および C++ データ型へのマッピング

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(*n*)	BSTR	String	BSTR
CHAR(*n*) FOR BIT DATA	SAFEARRAY[unsigned char]	Byte()	SAFEARRAY
VARCHAR(*n*)	BSTR	String	BSTR
VARCHAR(*n*) FOR BIT DATA	SAFEARRAY[unsigned char]	Byte()	SAFEARRAY
LONG VARCHAR	BSTR	String	BSTR
LONG VARCHAR FOR BIT DATA	SAFEARRAY[unsigned char]	Byte()	SAFEARRAY
BLOB(*n*)	BSTR	String	BSTR
BLOB(*n*) FOR BIT DATA	SAFEARRAY[unsigned char]	Byte()	SAFEARRAY
GRAPHIC(*n*)、 VARGRAPHIC(*n*)、 LONG GRAPHIC、DBCLOB(*n*)	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 データ型に対応する BASIC データ型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・オブジェクトの特性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、特性、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。ルーチンの末尾引き数の場合、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、ストリングを処理する BSTR データ型を定義します。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。

関連概念:

- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『Visual Basic でのオブジェクトのリンクと埋め込み (OLE) オートメーション』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『Visual C++ でのオブジェクトのリンクと埋め込み (OLE) オートメーション』
- 203 ページの『OLE オートメーション・ルーチンの設計』

- 205 ページの『オブジェクト・インスタンスとスクラッチパッドに関する考慮事項および OLE ルーチン』

関連タスク:

- 203 ページの『OLE オートメーション・ルーチンの作成』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION ステートメント』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『オブジェクトのリンクと埋め込み (OLE) のサンプル』
- 206 ページの『OLE オートメーションでサポートされている SQL データ型』

OLE DB ユーザー定義表関数

以下の項では、OLE DB 表関数を作成する方法を説明しています。

OLE DB ユーザー定義表関数

Microsoft® OLE DB は、さまざまな情報ソースに保管されているデータへの同じ方法によるアクセスをアプリケーションに提供する、 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 Universal Database は、OLE DB ソースにアクセスする表関数を定義可能にすることによって、 OLE DB アプリケーションの作成を単純化します。 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。たとえば、OLE DB 表関数を定義して、 Microsoft Access データベースまたは Microsoft Exchange のアドレス帳からの表を戻し、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 とやりとりをします。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 Universal Database 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, 1998」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。

関連概念:

- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『オブジェクトのリンクと埋め込みデータベース (OLE DB) 表関数』
- 213 ページの『完全修飾行セット名』

関連タスク:

- 211 ページの『OLE DB 表 UDF の作成』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION (OLE DB 外部表) ステートメント』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『オブジェクトのリンクと埋め込みデータベース (OLE DB) 表関数のサンプル』
- 214 ページの『OLE DB でサポートされている SQL データ型』

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 文節に指定する。

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 文節には、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、たとえば、 "UPPERcase" のように、列名を引用符の間に置いてください。

EXTERNAL NAME 文節は、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 文節で*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。

たとえば、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¥nwind.mdb';

SELECT orderid, DATE(orderdate) AS orderdate,
                DATE(shippeddate) AS shippeddate
FROM TABLE(orders()) AS t
WHERE orderid = 10248;
```

EXTERNAL NAME 文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。たとえば、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・データ型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 文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型は、 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 文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;
```

関連概念:

- 213 ページの『完全修飾行セット名』
- 210 ページの『OLE DB ユーザー定義表関数』

関連タスク:

- 「*IBM DB2 Information Integrator インストール・ガイド*」の『DB2 Information Integrator をインストールして、OLE DB データ・ソースにアクセスするようサーバーをセットアップする』
- 「*IBM DB2 Information Integrator データ・ソース構成ガイド*」の『フェデレーテッド・サーバーへの OLE DB データ・ソースの追加』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE NICKNAME ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE SERVER ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE WRAPPER ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION (OLE DB 外部表) ステートメント』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『オブジェクトのリンクと埋め込みデータベース (OLE DB) 表関数のサンプル』
- 214 ページの『OLE DB でサポートされている SQL データ型』

完全修飾行セット名

EXTERNAL NAME 文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
  - カタログ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にカタログ名を置くかを判別するには、 OLE DB Provider の特性セット DBPROPSET_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。たとえば、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。

関連概念:

- 210 ページの『OLE DB ユーザー定義表関数』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION (OLE DB 外部表) ステートメント』

OLE DB でサポートされている SQL データ型

以下の表では、DB2 データ型を OLE DB データ型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。たとえば、データ型 INTEGER の列を使用して OLE DB 表関数を定義する場合、 DB2 は OLE DB プロバイダーからのデータを DBTYPE_I4 として要求します。

OLE DB Provider ソース・データ型から OLE DB データ型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、それぞれのデータ型を OLE DB データ型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 30. DB2 データ型の OLE DB へのマッピング

DB2 データ型	OLE DB データ型
SMALLINT	DBTYPE_I2
INTEGER	DBTYPE_I4
BIGINT	DBTYPE_I8
REAL	DBTYPE_R4
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 データ型変換規則は、「*Microsoft OLE DB 2.0 Programmer's Reference and Data Access SDK*, Microsoft Press, 1998」で定義されています。たとえば、次のようになります。

- OLE DB データ型 DBTYPE_CY を検索するには、データを OLE DB データ型 DBTYPE_NUMERIC(19,4) に変換することができ、これは DB2 データ型 DEC(19,4) にマップされます。
- OLE DB データ型 DBTYPE_I1 を検索するには、データを OLE DB データ型 DBTYPE_I2 に変換することができ、これは DB2 データ型 SMALLINT にマップされます。
- OLE DB データ型 DBTYPE_GUID を検索するには、データを OLE DB データ型 DBTYPE_BYTES に変換することができ、これは DB2 データ型 CHAR(12) FOR BIT DATA にマップされます。

関連概念:

- 210 ページの『OLE DB ユーザー定義表関数』

関連タスク:

- 211 ページの『OLE DB 表 UDF の作成』

第 5 章 ルーチンの呼び出し

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ら、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 文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。

プロシージャーの呼び出し:

プロシージャーを呼び出すには、そのプロシージャー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、プロシージャーを呼び出し、プロシージャーにパラメーターを渡し、プロシージャー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。プロシージャー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、プロシージャーが正常に戻された後に行えます。

プロシージャーは、 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。たとえば、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 (プロシージャー、UDF、またはメソッド)
- SQL ルーチン (プロシージャー、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ー (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ーを呼び出す場合、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、プロシージャーの言語以外の言語で作成してもかまいません。たとえば、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ー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、プロシージャー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、プロシージャー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。たとえば、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ーを呼び出すことができます。

プロシージャー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。たとえば、照会の選択リスト内や INSERT ステートメントの VALUES 文節内などの場所があります。表関数は FROM 文節内でしか参照できません。たとえば、照会やデータ変更ステートメントの FROM 文節内です。

メソッドの呼び出し:

メソッドは、構造化型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型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。

- 226 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ーの呼び出し』
- 227 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ーの呼び出し』

- CLI アプリケーションからのプロシージャーの呼び出し (Call a procedure from a CLI application)
- コマンド行プロセッサー (CLP) からのプロシージャーの呼び出し
- 238 ページの『スカラー関数またはメソッドの呼び出し』
- 239 ページの『ユーザー定義の表関数の呼び出し』

関連概念:

- 3 ページの『アプリケーション開発におけるルーチン』
- 222 ページの『ルーチンのコード・ページに関する考慮事項』
- 219 ページの『ルーチン名およびパス』
- 221 ページの『ネストされたルーチンの呼び出し』

関連タスク:

- 37 ページの『ルーチンの作成』
- 36 ページの『データベースでのルーチンの作成』
- 222 ページの『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』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CALL ステートメント』

ルーチン名およびパス

ストアード・プロシージャーまたは UDF の修飾名は schema-name.routine-name です。ストアード・プロシージャーまたは UDF を参照するどの場所でもこの修飾名を使用できます。たとえば、次のようにします。

```
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・プロシージャーまたは UDF の識別を試みます。たとえば、次のようにします。

```
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 DB2 により行われる*修飾されない* 参照を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・プロシージャー、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 FUNCPATH 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、それぞれのデータ型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 文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型に連結された修飾ルーチン名のことです。

メソッドは、関連した構造化型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、構造化型インスタンスに従って最も特定されたメソッドを実行します (たとえば、構造化型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、そこでは、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。

関連概念:

- 3 ページの『アプリケーション開発におけるルーチン』
- 271 ページの『ユーザー定義構造化型』
- 278 ページの『メソッドの動的ディスパッチング』
- 234 ページの『関数選択』
- 5 ページの『ルーチンのタイプ (プロシージャー、関数、メソッド)』
- 225 ページの『プロシージャーの選択』

関連タスク:

- 277 ページの『構造化型の振る舞いの定義』

関連資料:

- 「*SQL リファレンス 第 1 巻*」の『関数』
- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION ステートメント』

- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE TYPE (構造化) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『SET PATH ステートメント』
- 「コマンド・リファレンス」の『BIND コマンド』
- 「コマンド・リファレンス」の『PRECOMPILE コマンド』
- 「*SQL リファレンス 第 1 巻*」の『メソッド』

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。

16 個のレベルのネスト

ルーチンの呼び出しは、16 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 13 段階のネスト・レベルを指定することもできます。

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。たとえば、CONTAINS SQL 文節で作成された UDF は CONTAINS SQL 文節または NO SQL 文節のいずれかで作成されたプロシージャーを呼び出すことはできます。しかしこのルーチンは、 READS SQL DATA 文節または MODIFIES SQL DATA 文節を使用して作成されたストアード・プロシージャー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。

関連概念:

- 46 ページの『プロシージャー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』
- 27 ページの『ルーチンのセキュリティーに関する考慮事項』

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・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可能です。そのような環境で 32 ビット・ルーチンを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・プロシージャーの呼び出しでは、 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。

Java プロシージャーの場合、 32 ビット Java 仮想マシン (JVM) は 64 ビットのデータベース・サーバー上でも機能できます。このような JVM を使用する 32 ビット Java ルーチンの場合、パフォーマンスでのさらに別のオーバーヘッドはありません。これに匹敵し、しかも 64 ビット JVM を使用する 64 ビット・ルーチンのほうが高速で稼働することはありません。ただし、64 ビット・データベース・サーバーで稼働する 32 ビット Java ルーチンの場合、 FENCED NOT THREADSAFE モードで稼働する必要があるため、スケーラビリティーは良くありません。そのために、このようなルーチンを呼び出すには、そのつど独自の JVM を必要とします。

制約事項:

32 ビット・ルーチンは、64 ビット・インスタンスで稼働するには、 FENCED および NOT THREADSAFE で登録する必要があります。

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。

手順:

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: sqllib/function
   - Windows: sqllib¥function
2. ストアード・プロシージャーを 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ード・プロシージャーを呼び出す。

関連概念:

- 217 ページの『ルーチンの呼び出し』
- 189 ページの『Java ルーチン』

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 PARAMETER CCSID オプションによって示されるコード・ページで外部関数に渡されます。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。

たとえば、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 R を使用しているルーチンを呼び出す、コード・ページ S を持つセクションにアクセスすると、以下のことが起きます。

1. SQL ステートメントを呼び出すと、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、ルーチンが呼び出される前に、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。たとえば、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) に変換されます。 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。

関連概念:

- 「*SQL リファレンス 第 1 巻*」の『文字変換』
- 「*アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」の『コード・ページ値の導出』
- 「*アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」の『プリコンパイルおよびバインド用のアクティブ・コード・ページ』
- 「*アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」の『アプリケーション実行用のアクティブ・コード・ページ』
- 「*アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」の『異なるコード・ページ間での文字変換』

- 「アプリケーション開発ガイド クライアント・アプリケーションのプログラミング」の『コード・ページ変換はいつ行われるか』
- 「アプリケーション開発ガイド クライアント・アプリケーションのプログラミング」の『コード・ページ変換時の文字置換』
- 「アプリケーション開発ガイド クライアント・アプリケーションのプログラミング」の『サポートされるコード・ページ変換』
- 「アプリケーション開発ガイド クライアント・アプリケーションのプログラミング」の『コード・ページが異なる状況におけるアプリケーション開発』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE TYPE (構造化) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE METHOD ステートメント』
- 「*管理ガイド: プランニング*」の『サポートされているテリトリー・コードおよびコード・ページ』
- 「*管理ガイド: プランニング*」の『コード・ページ 923 および 924 の変換表』

プロシージャーの呼び出し

プロシージャーの参照

修飾名 (スキーマおよびストアード・プロシージャー名) と、その後に続く括弧で囲まれた引き数リストでストアード・プロシージャーを参照している CALL ステートメントによってストアード・プロシージャーは呼び出されます。また、スキーマ名を使用しないでストアード・プロシージャー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ード・プロシージャーを選ぶことになります。

ストアード・プロシージャー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・マーカー、式、または NULL で構成することができます。以下に、ストアード・プロシージャー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 データ型が Java クラス・タイプにマップされていない限り、 NULL を Java™ ストアード・プロシージャーに渡すことはできません。
- PARAMETER STYLE GENERAL ストアード・プロシージャーに NULL を渡すことはできません。

引き数の位置は重要で、それをセマンティクスするストアード・プロシージャーの定義に正確に従っていなければなりません。引き数の位置と、ストアード・プロシージャーの定義の両方がストアード・プロシージャー本体に従っていなければなりません。 DB2® は、引き数がストアード・プロシージャーの定義とうまく一致するように、引き数を入れ替えたりはしません。また、DB2 はストアード・プロシージャー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。

関連概念:

- 99 ページの『外部ルーチン用のパラメーター・スタイル』

関連タスク:

- 230 ページの『コマンド行プロセッサー (CLP) からのプロシージャーの呼び出し』
- 「*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・プロシージャーの呼び出し』
- 227 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ーの呼び出し』
- 226 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ーの呼び出し』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE ステートメント』
- 101 ページの『C/C++、OLE、COBOL で書かれたルーチンに引き数を渡すときの構文』

プロシージャーの選択

ストアード・プロシージャーが呼び出されると、データベース・マネージャーは同一名の付いたストアード・プロシージャーのうちのどれを呼び出せばよいかを決める必要があります。ストアード・プロシージャーの解決は、次のようなステップを経て行われます。

1. 以下がすべて真になるようなすべてのストアード・プロシージャーをカタログ (SYSCAT.ROUTINES) から探し出します。
   - スキーマ名を指定した呼び出し (すなわち修飾参照) の場合、スキーマ名とストアード・プロシージャー名は呼び出し名に一致する。
   - スキーマ名を指定していない呼び出し (すなわち非修飾参照) の場合、ストアード・プロシージャー名は、呼び出し名に一致し、 SQL パス内のスキーマのうちのいずれかに一致するスキーマ名をもっている。
   - 定義済みパラメーターの数は呼び出しに一致する。
   - 呼び出し側は、ストアード・プロシージャーでの EXECUTE 特権をもっている。
2. SQL パス内の先頭にあるスキーマをもつストアード・プロシージャーを選択している。

最初のステップの完了後も候補のストアード・プロシージャーがない場合、エラーが戻されます (SQLSTATE 42884)。

関連概念:

- 217 ページの『ルーチンの呼び出し』
- 224 ページの『プロシージャーの参照』

関連タスク:

- 227 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ーの呼び出し』
- 226 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ーの呼び出し』

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ーの呼び出し

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ー (ストアード・プロシージャーともいう) 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。

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ーを作成しておく必要があります。

外部プロシージャーの場合は、 CREATE PROCEDURE ステートメントの EXTERNAL 文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。

プロシージャーの呼び出し側には、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ーの呼び出し側は、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。

手順:

アプリケーションからプロシージャーを呼び出す場合は、そのアプリケーションに特定のエレメント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 ステートメントで必要なホスト変数またはパラメーター・マーカー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。
   - プロシージャー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・マーカーを割り当てます。
   - 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・マーカーを初期化します。
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. プロシージャーの呼び出しをコーディングします。プロシージャー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。プロシージャーが予期する 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・マーカー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、プロシージャー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ーでは、 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。

関連概念:

- 217 ページの『ルーチンの呼び出し』
- 225 ページの『プロシージャーの選択』
- 224 ページの『プロシージャーの参照』

関連タスク:

- 227 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ーの呼び出し』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『COMMIT ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『ROLLBACK ステートメント』

関連サンプル:

- 『spcall.c -- Call individual stored procedures』
- 『spclient.c -- Call various stored procedures』
- 『spclient.sqc -- Call various stored procedures (C)』
- 『spclient.sqC -- Call various stored procedures (C++)』
- 『SpClient.java -- Call a variety of types of stored procedures from SpServer.java (JDBC)』

トリガーまたは SQL ルーチンからのプロシージャーの呼び出し

プロシージャーの呼び出しは、 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ー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ーを作成しておく必要があります。
- 外部プロシージャーの場合は、 CREATE PROCEDURE ステートメントの EXTERNAL 文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、そのプロシージャーの 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。

制約事項:

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ー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 からプロシージャーを呼び出せません。
- 対称マルチプロセッサー (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 つのプロセッサーで実行されます。
- 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めません。 ROLLBACK TO SAVEPOINT ステートメントは使用できますが、指定のセーブポイントがプロシージャー内に存在している必要があります。
- トリガーから CALL ステートメントをロールバックしても、プロシージャー内で実行されるステートメント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。

- 73 ページの『SQL ルーチンの SQL アクセス・レベル』
- 116 ページの『外部ルーチンでの SQL』

手順:

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、割り当てステートメント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ーも読み取り/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、プロシージャー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 プロシージャーの呼び出し:

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ーを呼び出す方法と、 GET DIAGNOSTICS ステートメントを使用してプロシージャー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 言語プロシージャー、 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 DIANOSTICS 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 MODE DB2SQL
WHEN (n.c1 > 100);
BEGIN ATOMIC
   DECLARE rs INTEGER DEFAULT 0;
   CALL proc(n.c1, n.c2);
   GET DIANOSTICS 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、プロシージャーが呼び出されます。

```
UPDATE T1 SET c1 = c1+1 WHERE c2 = 'CA'@
```

関連概念:

- 73 ページの『SQL ルーチンの SQL アクセス・レベル』
- 116 ページの『外部ルーチンでの SQL』
- 217 ページの『ルーチンの呼び出し』
- 225 ページの『プロシージャーの選択』
- 224 ページの『プロシージャーの参照』

関連タスク:

- 「*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・プロシージャーの呼び出し』
- 226 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ーの呼び出し』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CALL ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE TRIGGER ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『GET DIAGNOSTICS ステートメント』

コマンド行プロセッサー (CLP) からのプロシージャーの呼び出し

DB2 コマンド行プロセッサー・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・プロシージャーを呼び出せます。呼び出すストアード・プロシージャーは、 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。

手順:

ストアード・プロシージャーを呼び出すには、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・プロシージャー名と IN または INOUT パラメーター値、さらに各 OUT パラメーター値のプレースホルダーとして '?' を入力します。

ストアード・プロシージャーのパラメーターは、プログラム・ソース・ファイル内のストアード・プロシージャーの CREATE PROCEDURE ステートメントで指定します。

SQL プロシージャーの例

SQL プロシージャーを作成するための詳細については、第 6 章『SQL プロシージャーの作成』を参照してください。

whiles.db2 ファイル内の DEPT_MEDIAN プロシージャー・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)
```

このプロシージャーを呼び出すには、プロシージャー名と適切なパラメーター引き数を指定した CALL ステートメントを使用します。これには、IN パラメーターの値と、OUT パラメーターの値の疑問符 (?) が指定されます。プロシージャーの 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。たとえば、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 ″¥″ を置くか、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ー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
```

C ストアード・プロシージャーの例

コマンド行プロセッサーから、サポートされているホスト言語で作成したストアード・プロシージャー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

シージャーを作成するためのファイルを用意しています。 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、いくつかのストアード・プロシージャーが入っています。ストアード・プロシージャーをカタログに登録するには、 spcreate.db2 ファイルを使用します。

spcreate.db2 ファイル内の MAIN_EXAMPLE プロシージャーの 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)
```

このストアード・プロシージャーを呼び出すには、 IN パラメーター job に CHAR 値を入力し、各 OUT パラメーターに疑問符 (?) を入力します。プロシージャーの 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、ストアード・プロシージャーを呼び出して、 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。

```
CALL stored procedure named MAIN_EXAMPLE
Stored procedure returned successfully
Average salary for job DESIGNER =  23731.25
```

関連タスク:

- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『SQL プロシージャーの作成』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『クライアント・アプリケーションによる SQL プロシージャーの呼び出し』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『Windows でのクライアント・アプリケーションによる SQL プロシージャーの呼び出し』
- 227 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ーの呼び出し』
- 226 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ーの呼び出し』

関連サンプル:

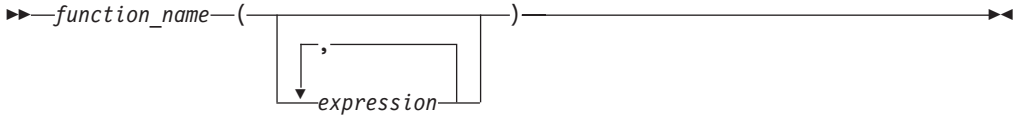
- 『spclient.sqc -- Call various stored procedures (C)』

- 『spcreate.db2 -- How to catalog the stored procedures contained in spserver.sqc (C)』
- 『spserver.sqc -- Definition of various types of stored procedures (C)』
- 『whiles.db2 -- To create the DEPT_MEDIAN SQL procedure 』
- 『whiles.sqc -- To call the DEPT_MEDIAN SQL procedure』

関数とメソッドの呼び出し

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。引き数は、0 ～ 90 の数字にすることができ、これらは式になります。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・マーカー

引き数の位置は重要で、それをセマンティクスする関数の定義に正確に従っていなければなりません。引き数の位置と、関数の定義の両方が関数本体に従っていなければなりません。 DB2® は、引き数が関数の定義とうまく一致するように、引き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 引き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き数を使用している場合、 FROM 文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・マーカーを使用するには、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(?)
```

引き数がどのデータ型になるかは、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・マーカーにタイプを指定することができます。たとえば次のように 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。たとえば 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
```

関連タスク:

- 238 ページの『スカラー関数またはメソッドの呼び出し』
- 239 ページの『ユーザー定義の表関数の呼び出し』

関連資料:

- 「*SQL リファレンス 第 1 巻*」の『関数』

関数選択

修飾および非修飾のどちらの関数参照の場合でも、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
- 関数参照の引き数と同じ数の定義済みパラメーター
- 対応する引き数のタイプに一致する、またはその引き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、修飾された参照の場合は名前の付いたスキーマ内の関数、修飾されない参照の場合は 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。修飾されない参照の場合のみ、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 パスが使用されます。

関数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。

たとえば、次のようにします。

```
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。たとえば、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とします。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。

関数選択について詳しく知りたければ、関連リンク先に一覧で示されている『関数』のトピックの『関数参照』の項を参照してください。

関連概念:

- 233 ページの『関数の参照』

関連タスク:

- 238 ページの『スカラー関数またはメソッドの呼び出し』
- 239 ページの『ユーザー定義の表関数の呼び出し』

関連資料:

- 「*SQL リファレンス 第 1 巻*」の『関数』
- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE TYPE (構造化) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE METHOD ステートメント』

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・データ型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 に対する引き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 の強いタイプ定義機能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。

関連概念:

- 「*アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」の『ユーザー定義タイプ (UDT) およびラージ・オブジェクト (LOB)』
- 234 ページの『関数選択』
- 225 ページの『プロシージャーの選択』

関連タスク:

- 321 ページの『構造化型パラメーターを外部ルーチンに渡す』
- 237 ページの『UDF パラメーターとしての LOB 値』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『SELECT ステートメント』

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。たとえば、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き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。たとえば、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。この関数の引き数には、定義されたタイプの LOB 値を取ることができます。引き数が 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、普通はホスト変数ロケーターを引き数として使用します。

関連概念:

- 「*アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」の『ユーザー定義タイプ (UDT) およびラージ・オブジェクト (LOB)』
- 234 ページの『関数選択』
- 225 ページの『プロシージャーの選択』

関連タスク:

- 246 ページの『LOB ロケーターによる LOB 値の検索』
- 236 ページの『UDF またはメソッドのパラメーターとしての特殊タイプ』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE METHOD ステートメント』

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 文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。

手順:

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。たとえば、照会の選択リストや VALUES 文節の中で、スカラー UDF やメソッドの参照を記述できます。

たとえば、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
```

関連概念:

- 233 ページの『関数の参照』
- 217 ページの『ルーチンの呼び出し』
- 219 ページの『ルーチン名およびパス』

- 15 ページの『ユーザー定義のスカラー関数』
- 18 ページの『メソッド』

関連タスク:

- 239 ページの『ユーザー定義の表関数の呼び出し』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『SELECT ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION (SQL スカラー、表、または行) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION (外部スカラー) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION (ソースまたはテンプレート) ステートメント』

関連サンプル:

- 『udfcli.sqc -- Call a variety of types of user-defined functions (C)』
- 『udfemcli.sqc -- Call a variety of types of embedded SQL user-defined functions. (C)』
- 『udfcli.sqC -- Call a variety of types of user-defined functions (C++)』
- 『udfemcli.sqC -- Call a variety of types of embedded SQL user-defined functions. (C++)』
- 『UDFcli.java -- Call the UDFs in UDFsrv.java (JDBC)』
- 『UDFjcli.java -- Call the UDFs in UDFjsrv.java (JDBC)』
- 『UDFcli.sqlj -- Call the UDFs in UDFsrv.java (SQLj)』
- 『UDFjcli.sqlj -- Call the UDFs in UDFjsrv.java (SQLj)』

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 文節を使用して呼び出すことができます。

前提条件:

- 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 文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。

制約事項:

ユーザー定義の表関数の呼び出しに関する制約事項の詳細は、関連リンクの CREATE FUNCTION の項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 文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 文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。

たとえば以下の 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, FIRSTNME 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
```

関連概念:

- 233 ページの『関数の参照』
- 219 ページの『ルーチン名およびパス』
- 17 ページの『ユーザー定義のスカラー関数』

関連タスク:

- 237 ページの『UDF パラメーターとしての LOB 値』
- 238 ページの『スカラー関数またはメソッドの呼び出し』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION (OLE DB 外部表) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION (SQL スカラー、表、または行) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION (外部表) ステートメント』

関連サンプル:

- 『udfcli.sqc -- Call a variety of types of user-defined functions (C)』
- 『udfemcli.sqc -- Call a variety of types of embedded SQL user-defined functions. (C)』
- 『udfcli.sqC -- Call a variety of types of user-defined functions (C++)』
- 『udfemcli.sqC -- Call a variety of types of embedded SQL user-defined functions. (C++)』
- 『UDFcli.java -- Call the UDFs in UDFsrv.java (JDBC)』
- 『UDFjcli.java -- Call the UDFs in UDFjsrv.java (JDBC)』
- 『UDFcli.sqlj -- Call the UDFs in UDFsrv.java (SQLj)』
- 『UDFjcli.sqlj -- Call the UDFs in UDFjsrv.java (SQLj)』

第 2 部 ラージ・オブジェクト、ユーザー定義特殊タイプ、トリガー

第 6 章 ラージ・オブジェクト

ラージ・オブジェクトの使用法

VARCHAR および VARGRAPHIC データ型のストレージ限界は、32K バイトです。これは普通サイズ以下のテキスト・データには十分ですが、アプリケーションでは大きなテキスト・ドキュメントを保管しなければならない場合がよくあります。また他にも音声、ビデオ、図面、テキストとグラフィックスの混合したもの、イメージなど、さまざまなデータ型を保管しなければならない場合もあります。DB2® は、このようなデータ・オブジェクトを 2 ギガバイト (GB) 以下のサイズのストリングとして保管する 3 つのデータ型を備えています。その 3 つのラージ・オブジェクト (LOB) データ型とは、バイナリー・ラージ・オブジェクト (BLOB)、文字ラージ・オブジェクト (CLOB)、および 2 バイト文字ラージ・オブジェクト (DBCLOB) です。

注: CLOB には単一バイト文字か 2 バイト文字のどちらかを入れることができます。 DBCLOB には 4 バイト文字か 2 バイト文字のどちらかを入れることができます。

DB2 の各表には、関連する LOB データを大量に含めることができます。この場合、1 つの LOB 値は 2 ギガバイトより大きくなることはありませんが、 LOB データは 1 行当たり 24 ギガバイト、1 表当たり 2 テラバイトまで可能です。

個別のデータベース・ロケーションは、すべてのラージ・オブジェクト値を表中のレコードの外側に保管します。表中の各行のラージ・オブジェクトには、それぞれラージ・オブジェクト記述子が付いています。ラージ・オブジェクト記述子には、ディスク上の別の場所に保管されているラージ・オブジェクト・データにアクセスするのに使用される制御情報が含まれています。レコードの外側にラージ・オブジェクト・データを保管できるので、 LOB のサイズは 2 GB まで可能です。ラージ・オブジェクト記述子にアクセスすることで、 LOB を操作する際にわずかなオーバーヘッドが生じます。 (保管およびパフォーマンス上の理由により、 LOB には小さいデータ項目を入れない方がよいでしょう。)

各ラージ・オブジェクト列の最大サイズは、 CREATE TABLE ステートメント中のラージ・オブジェクト・タイプの宣言の一部です。ラージ・オブジェクト列の最大サイズにより、その列中のすべての LOB 記述子の最大サイズが決まります。その結果、すべてのデータ型のうち、単一行当たりに適合できる列数も決まります。その行中で LOB 記述子により使用されるスペースは、対応するその列の最大サイズによって、およそ 60 ～ 300 バイトの範囲にわたります。

CREATE TABLE の lob-options-clause を使用して、 LOB 列への変更を記録する (またはしない) ことを選択できます。またこの文節を使用して、 LOB 記述子を簡

潔に表示する (またはしない) こともできます。これは、LOB を保管するのに十分なスペースだけを割り振ったり、将来の追加操作に向けて LOB に余分なスペースを割り振ることができるということを意味します。

CREATE TABLE の tablespace-options-clause を使用すると、倍精度フィールドの列値や LOB データ型を保管するための LARGE 表スペースを識別できます。

LOB が大きなサイズになる場合は、データベース内外への移動時にデータベース・システムのパフォーマンスが大幅に低下することがあります。たとえ DB2 が 1 GB より大きい LOB 値のロギングを実行しなくても、 LOB 値が 1 GB に近い値であれば (ログが行われるので)、すぐにデータベース・ログは限界に達してしまいます。 SQLCODE -355 (SQLSTATE 42993) というエラーは、 1 GB より大きいサイズの LOB のログを行おうとした場合に起こります。 CREATE TABLE および ALTER TABLE ステートメントに lob-options-clause を使用すると、特定の LOB 列のロギングをオフにすることができます。オプションを NOT LOGGED に設定するとパフォーマンスを向上できますが、最新のバックアップ後の LOB 値の変更がロールフォワード・リカバリー中に失われます。

LOB 値を選択する際には、次のオプションがあります。

- すべての LOB 値をホスト変数に指定する。すべての LOB 値がサーバーからクライアントへコピーされます。これは非効率ですし、実行できないこともあります。ホスト変数が使用するクライアント・メモリーのバッファーの容量が、大きい LOB 値を保留するのに十分でない場合もあります。
- LOB ロケーターのみをホスト変数に指定する。 LOB 値はサーバーに残され、LOB ロケーターはクライアントに移動されます。 LOB 値が非常に大きく、次の 1 つまたは複数 SQL ステートメントの入力値としてのみ必要な場合は、ロケーターの中に値を保持するのが最も良い方法です。ロケーターを使用すると、LOB 値をホスト変数へ転送しサーバーへ戻すために必要な、すべてのクライアント/サーバー通信の通信量を削除します。
- すべての LOB 値をファイル参照変数に指定する。 LOB 値 (またはその一部) はアプリケーションのメモリーを介さずにクライアントのファイルに移動されます。

関連概念:

- 244 ページの『ラージ・オブジェクト・ロケーター』
- 250 ページの『ラージ・オブジェクト・ファイル参照変数』

ラージ・オブジェクト・ロケーター

ラージ・オブジェクト・ロケーター (または LOB ロケーター) とは、データベース・サーバー内の単一の LOB 値を表す 4 バイトの値を持ったホスト変数です。LOB ロケーターは、アプリケーション・プログラムが実行するクライアント・マシン上に LOB 値全体を保管しなくても、そのアプリケーション・プログラム内のラージ・オブジェクトを簡単に操作できるようにするメカニズムをユーザーに提供します。そして、後続のステートメントを使用することにより、ロケーターは必ずしもラージ・オブジェクト全体を検索しなくても、データに対する操作を実行することができます。ロケーター変数は、 LOB にアクセスするアプリケーションのスト

レージの必要量を減らしたり、クライアントとサーバー間のデータの流れを少なくすることによってパフォーマンスを向上するために使用されます。

LOB ロケーターは、以下に示すような数々のプログラム・シーンに理想的に適しています。

1. かなり大きな LOB のごく一部のみをクライアント・プログラムに移動する場合。
2. LOB 全体がアプリケーションのメモリーに収まらない場合。
3. プログラムに LOB 式からの一時的な LOB 値が必要であるが、その結果を保管する必要はない場合。

また、LOB ロケーターは LOB 式に関連した値を表すこともできます。たとえば、LOB ロケーターは次の式に関連した値を表すことがあります。

```
SUBSTR( <lob 1> CONCAT <lob 2> CONCAT
        <lob 3>, <start>, <length> )
```

LOB ロケーターが、データベース内の行またはロケーションではなく、値を表すことを理解することは大切です。ロケーターに値を選択すると、そのロケーターが参照する値に影響を与える元の行または表に対してどんな操作も実行できません。ロケーターに関連した値は、トランザクションが終了するまでか、ロケーターが明示的に解放されるまでのうち、どちらかが最初に行われるまで有効です。ロケーターはこの機能を提供するためにデータの余分のコピーを強制することはなく、かわりにロケーターのメカニズムで LOB 基本値の記述を保管します。 LOB 値 (または上記で示した式) のマテリアライズは、その値が実際に特定のロケーション (ホスト変数の形式のユーザー・バッファーか、データベース内の別のレコードのフィールド値) に割り振られるまで据え置かれます。

LOB ロケーターは、トランザクション中に LOB 値を参照するために使用するメカニズムに過ぎません。 LOB ロケーターが作成されたトランザクションを超えて持続することはありません。 FREE LOCATOR ステートメントは、ロケーターをその値から解放します。同様に、コミットまたはロールバック操作は、トランザクションに関連したすべての LOB ロケーターを解放します。さらに、LOB ロケーターはデータベース・タイプではありません。それがデータベース内に保管されることはないため、ビューまたはチェック制約に関与することはできません。しかし、LOB ロケーターは LOB タイプをクライアントが表現したものであるため、 LOB ロケーターには SQLTYPE があります。したがって、FETCH、OPEN、および EXECUTE ステートメントによって使用される SQLDA 構造内で、LOB ロケーターについて記述することができます。また、DB2® と UDF の間で渡すこともできます。

アプリケーション・プログラムの通常のホスト変数については、ホスト変数に NULL 値を指定すると、標識変数が -1 に設定されます。これは、その値が NULL であることを示します。しかし LOB ロケーターの場合は、標識変数の意味が少し異なります。ロケーターのホスト変数自体は決して NULL にはならないので、負標識の変数値が、LOB ロケーターにより表される LOB 値は NULL であることを示します。

関連概念:

- 243 ページの『ラージ・オブジェクトの使用法』

関連タスク:

- 246 ページの『LOB ロケーターによる LOB 値の検索』
- 248 ページの『LOB 式の評価の据え置き』

関連サンプル:

- 『dtlob.out -- HOW TO USE THE LOB DATA TYPE (C)』
- 『dtlob.sqc -- How to use the LOB data type (C)』
- 『dtlob.out -- HOW TO USE THE LOB DATA TYPE (C++)』
- 『dtlob.sqC -- How to use the LOB data type (C++)』
- 『dtLob.bas -- Get/set Large Objects (LOBs)』
- 『DtLob.java -- How to use LOB data type (JDBC)』
- 『DtLob.out -- HOW TO USE LOB DATA TYPE. Connect to 'sample' database using JDBC type 2 driver (JDBC)』
- 『lobeval.sqb -- Demonstrates how to use a Large Object (LOB) (IBM COBOL)』
- 『lobloc.sqb -- Demonstrates the use of LOB locators (IBM COBOL)』

LOB ロケーターによる LOB 値の検索

データを LOB から抽出する必要がある場合は、LOB ロケーターを使用できます。この方法は、アクセスする LOB が大きい場合に便利です。ロケーターを使用すれば、LOB データのサブセットだけが必要な場合に LOB 全体をクライアントに転送しないですみます。

例では、C の組み込み SQL を使用します。

手順:

LOB ロケーターを使用して LOB 値を検索するには、次のようにします。

1. LOB ロケーター・ホスト変数を宣言する。

```
EXEC SQL BEGIN DECLARE SECTION;
   char number[7];
   sqlint32 deptInfoBeginLoc;
   sqlint32 deptInfoEndLoc;
   SQL TYPE IS CLOB_LOCATOR resume;
   SQL TYPE IS CLOB_LOCATOR deptBuffer;
   short lobind;
   char buffer[1000]="";
   char userid[9];
   char passwd[19];
EXEC SQL END DECLARE SECTION;
```

   ホスト変数宣言セクションでは、以下のようになります。
   - number には、カーソル c1 によって発行される、SELECT ステートメント内の empno が戻す値が含まれる。
   - deptInfoBeginLoc と deptInfoEnd は LOB ロケーターの値を一時的に保留する。
   - resume と deptBuffer は LOB ロケーターである。
   - lobind は、LOB 読み取りが NULL かどうかを示すために使用される。
   - buffer には、LOB から抽出されたデータが含まれる。

- userid と passwd はユーザー ID とパスワードの組み合わせであり、アプリケーションをデータベースに接続するために必要となる。

2. アプリケーションをデータベースに接続する。
3. カーソルを宣言およびオープンする。

```
EXEC SQL DECLARE c1 CURSOR FOR
   SELECT empno, resume FROM emp_resume WHERE resume_format='ascii'
   AND empno <> 'A00130';

EXEC SQL OPEN c1;
```

4. ホスト変数ロケーターに LOB 値を取り出す。

```
EXEC SQL FETCH c1 INTO :number, :resume :lobind;
```

5. LOB ロケーターを評価する。
   a. Department Information セクションの先頭を見付ける。

```
EXEC SQL VALUES (POSSTR(:resume, 'Department Information'))
   INTO :deptInfoBeginLoc;
```

   b. Education の先頭 (Department Information の末尾) を見付ける。

```
EXEC SQL VALUES (POSSTR(:resume, 'Education'))
   INTO :deptInfoEndLoc;
```

   c. SUBSTR を使用して、Department Information セクションだけを取得する。

```
EXEC SQL VALUES(SUBSTR(:resume, :deptInfoBeginLoc,
   :deptInfoEndLoc - :deptInfoBeginLoc)) INTO :deptBuffer;
```

   d. Department Information セクションを :buffer 変数に付加する。

```
EXEC SQL VALUES(:buffer || :deptBuffer) INTO :buffer;
```

6. LOB ロケーター resume および deptBuffer を解放する。

```
EXEC SQL FREE LOCATOR :resume, :deptBuffer;
```

7. カーソルをクローズする。

```
EXEC SQL CLOSE c1;
```

8. プログラムを終了する。

関連概念:

- 243 ページの『ラージ・オブジェクトの使用法』
- 244 ページの『ラージ・オブジェクト・ロケーター』

関連タスク:

- 「*アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」の『アプリケーションのデータベースへの接続』
- 「*アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」の『アプリケーション・プログラムの終了』
- 248 ページの『LOB 式の評価の据え置き』

関連サンプル:

- 『dtlob.out -- HOW TO USE THE LOB DATA TYPE (C)』
- 『dtlob.sqc -- How to use the LOB data type (C)』
- 『dtlob.out -- HOW TO USE THE LOB DATA TYPE (C++)』
- 『dtlob.sqC -- How to use the LOB data type (C++)』
- 『dtLob.bas -- Get/set Large Objects (LOBs)』

- 『DtLob.java -- How to use LOB data type (JDBC)』
- 『DtLob.out -- HOW TO USE LOB DATA TYPE. Connect to 'sample' database using JDBC type 2 driver (JDBC)』
- 『lobeval.sqb -- Demonstrates how to use a Large Object (LOB) (IBM COBOL)』
- 『lobloc.sqb -- Demonstrates the use of LOB locators (IBM COBOL)』

LOB 式の評価の据え置き

ターゲットの宛先に LOB 式の指定を行うまで、LOB 値のバイトの移動はありません。これは、ストリング関数および演算子と共に使用される LOB 値のロケーターが、割り当ての時点まで評価が延期される式を作成できるということを意味します。この技法は、LOB 式の評価の据え置きと呼ばれます。

評価を据え置くと、DB2 が LOB I/O のパフォーマンスを向上させる場合があります。これは、LOB 関数オプティマイザーが LOB 式を代替式に変形させようとするために起こります。これらの代替式は同じ結果を出しますが、通常はディスク I/O が少なくて済みます。

例では、C の組み込み SQL を使用します。

手順:

LOB 式の評価を据え置くには、次のようにします。

1. LOB ロケーター・ホスト変数を宣言する。

```
EXEC SQL BEGIN DECLARE SECTION;
   sqlint32 hv_start_deptinfo;
   sqlint32 hv_start_educ;
   sqlint32 hv_return_code;
   SQL TYPE IS CLOB(5K) hv_new_section_buffer;
   SQL TYPE IS CLOB_LOCATOR hv_doc_locator1;
   SQL TYPE IS CLOB_LOCATOR hv_doc_locator2;
   SQL TYPE IS CLOB_LOCATOR hv_doc_locator3;
   char userid[9];
   char passwd[19];
EXEC SQL END DECLARE SECTION;
```

   ホスト変数宣言セクションでは、以下のようになります。
   - hv_start_deptinfo、hv_return_code、および hv_start_educ は LOB ロケーターの値を一時的に保留する。
   - hv_new_section_buffer には、LOB から抽出されたデータが含まれる。
   - hv_doc_locator1、hv_doc_locator2、および hv_doc_locator3 は LOB ロケーターである。
   - userid と passwd はユーザー ID とパスワードの組み合わせであり、アプリケーションをデータベースに接続するために必要となる。
2. アプリケーションをデータベースに接続する。
3. ホスト変数ロケーターに LOB 値を取り出す。

```
EXEC SQL SELECT resume INTO :hv_doc_locator1 FROM emp_resume
   WHERE empno = '000130' AND resume_format = 'ascii';
```

4. ロケーターを使用して LOB データを操作する。以下の 5 つのステートメントは、 LOB フィールドに含まれている実データを移動せずに LOB データを操作します。
   a. POSSTR 関数を使用して、Department Information セクションの先頭を見付ける。

      ```
      EXEC SQL VALUES (POSSTR(:hv_doc_locator1, 'Department Information'))
         INTO :hv_start_deptinfo;
      ```

   b. POSSTR 関数を使用して、Education セクションの先頭を見付ける。

      ```
      EXEC SQL VALUES (POSSTR(:hv_doc_locator1, 'Education'))
         INTO :hv_start_educ;
      ```

   c. Department Information セクションを消去する。

      ```
      EXEC SQL VALUES (SUBSTR(:hv_doc_locator1, 1, :hv_start_deptinfo -1)
         || SUBSTR (:hv_doc_locator1, :hv_start_educ))
         INTO :hv_doc_locator2;
      ```

   d. Department Information セクションを hv_new_section_buffer に移動する。

      ```
      EXEC SQL VALUES (SUBSTR(:hv_doc_locator1, :hv_start_deptinfo,
         :hv_start_educ -:hv_start_deptinfo)) INTO :hv_new_section_buffer;
      ```

   e. 新しいセクションを末尾に付加する。実際には、これにより、 Department Information セクションが再開の最後に移動されます。

      ```
      EXEC SQL VALUES (:hv_doc_locator2 || :hv_new_section_buffer)
         INTO :hv_doc_locator3;
      ```

5. LOB データをターゲット宛先に移動する。

   ```
   EXEC SQL INSERT INTO emp_resume
                  VALUES ('A00130', 'ascii', :hv_doc_locator3);
   ```

   ターゲットの宛先に割り当てられた LOB の評価は、このステートメントまで延期されます。 LOB 値のバイトが最終的に移動するのは、この時点です。

6. LOB ロケーター hv_doc_locator1、hv_doc_locator2、および hv_doc_locator3 を解放する。

   ```
   EXEC SQL FREE LOCATOR :hv_doc_locator1, :hv_doc_locator2,
                         : hv_doc_locator3;
   ```

7. プログラムを終了する。

この例では、特定の再開 (empno = '000130') が、 EMP_RESUME という再開の表の中でシークされます。 Department Information という再開のセクションは、コピーおよび切り抜きされ、再開の最後に追加されます。そして、この新規の再開は EMP_RESUME という表に挿入されます。この表中の元の再開は、未変更のままです。

ロケーターは、元の再開から実際にバイトの移動またはコピーを行わずに、新規の再開をアセンブリーおよび検査することを許可します。最終的な割り当て、つまり INSERT ステートメントまで、バイトの移動は行われません。これはまた、サーバーにおいてのみ行われます。

関連概念:

- 243 ページの『ラージ・オブジェクトの使用法』

- 244 ページの『ラージ・オブジェクト・ロケーター』

関連タスク:

- 「*アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」の『アプリケーションのデータベースへの接続』
- 「*アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」の『アプリケーション・プログラムの終了』
- 246 ページの『LOB ロケーターによる LOB 値の検索』

関連サンプル:

- 『dtlob.out -- HOW TO USE THE LOB DATA TYPE (C)』
- 『dtlob.sqc -- How to use the LOB data type (C)』
- 『dtlob.out -- HOW TO USE THE LOB DATA TYPE (C++)』
- 『dtlob.sqC -- How to use the LOB data type (C++)』
- 『dtLob.bas -- Get/set Large Objects (LOBs)』
- 『DtLob.java -- How to use LOB data type (JDBC)』
- 『DtLob.out -- HOW TO USE LOB DATA TYPE. Connect to 'sample' database using JDBC type 2 driver (JDBC)』
- 『lobeval.sqb -- Demonstrates how to use a Large Object (LOB) (IBM COBOL)』

ラージ・オブジェクト・ファイル参照変数

LOB ファイル参照変数を使用すると、クライアント・アプリケーションのメモリーを通過しないので、データベース・サーバーからクライアントへの LOB 値の移動が容易になります。ファイル参照変数は、メモリー・バッファーでなくクライアント・ファイルの内外へデータを転送するために使用される場合以外は、ホスト変数と同様です。この方法を使用すると、クライアント・アプリケーションは、LOB データの移動を実行するために、ホスト変数 (これにはサイズ制限がある) を使用してファイルの読み込みおよび書き込みを行うユーティリティー・ルーチンを呼び出す必要はありません。

ファイル参照変数は、LOB ロケーターが LOB 値を (含むのではなく) 表すのと同様に、ファイルを (含むのではなく) 表します。データベースの照会、更新および挿入を行うと、単一の LOB 列値を保管または検索するためにファイル参照変数が使用される場合があります。

ファイル参照変数は LOB の直接ファイル入出力に使用され、すべてのホスト言語で定義することができます。これはネイティブのデータ型ではないので、SQL 拡張機能が使用され、プリコンパイラーが各変数を表すために必要なホスト言語の構造体を生成します。

ファイル参照変数には以下のプロパティーがあります。

1. データ型: BLOB、CLOB、または DBCLOB。このプロパティーは、変数が宣言されるときに指定されます。
2. ファイル名: アプリケーション・プログラムがランタイムにこれを指定します。それは以下のいずれかです。
   - ファイルの完全パス名 (推奨)。

- 相対ファイル名。相対ファイル名を指定する場合、クライアント・プロセスの現行パスに付加されます。アプリケーション内では、1 つのファイルは 1 つのファイル参照変数内でのみ参照されなければなりません。

3. ファイル名の長さ: アプリケーション・プログラムがランタイムにこれを指定します。これはファイル名の長さ (バイト数) です。
4. ファイル・オプション: アプリケーションは、ファイル参照変数を使用する前に、多数のオプションの中から 1 つをその変数に割り当てる必要があります。オプションは、ファイル参照変数構造内のフィールドにある INTEGER 値によって設定されます。ファイル参照変数ごとにファイル・オプションの 1 つを指定しなければなりません。

ファイル・オプション (言語別)	向き	説明
C: SQL_FILE_READ COBOL: SQL-FILE-READ FORTRAN: sql_file_read	入力	オープン、読み取り、クローズできるのは、正規のファイルです。
C: SQL_FILE_CREATE COBOL: SQL-FILE-CREATE FORTRAN: sql_file_create	出力	新規ファイルを作成します。ファイルがすでに存在している場合はエラーが戻されます。
C: SQL_FILE_OVERWRITE COBOL: SQL-FILE-OVERWRITE FORTRAN: sql_file_overwrite	出力	指定した名前を持つ既存のファイルが存在している場合は上書きされます。そうでない場合は、新規ファイルが作成されます。
C: SQL_FILE_APPEND COBOL: SQL-FILE-APPEND FORTRAN: sql_file_append	出力	指定した名前を持つ既存のファイルが存在している場合は、それに出力が付加されます。そうでない場合は、新規ファイルが作成されます。

5. データ長: これは入力時には未使用です。出力時には、データ長 (バイト数) はファイルに書き込まれる新規データの長さに設定されます。

アプリケーション・プログラムの通常のホスト変数については、ホスト変数に NULL 値を指定すると、標識変数が -1 に設定されます。これは、その値が NULL であることを示します。しかし、ファイル参照変数の場合は、標識変数の意味が少し異なります。ファイル参照変数自体は NULL にはならないので、負標識の変数値が、ファイル参照変数によって表される LOB 値が NULL であることを示します。

ファイル参照変数によって参照されるファイルは、プログラムを実行するシステムからアクセス可能 (しかし必ずしも常駐である必要はない) でなければなりません。ストアード・プロシージャーの場合、これはサーバーに当たります。

拡張 UNIX® コード (EUC) 環境では、 DBCLOB ファイル参照変数が指し示すファイルに、 GRAPHIC 列のストレージに適切な有効 EUC 文字だけが含まれ、 UCS-2 文字はまったく入っていないものと見なされます。

LOB ファイル参照変数を OPEN ステートメントで使用する場合、その LOB ファイル参照変数に関連付けられているファイルはカーソルが閉じられるまで絶対に削除しないでください。

関連概念:

- 243 ページの『ラージ・オブジェクトの使用法』

関連タスク:

- 252 ページの『CLOB 列からテキスト・ファイルへのデータの書き込み』
- 253 ページの『テキスト・ファイルから CLOB 列へのデータの挿入』

関連サンプル:

- 『dtlob.out -- HOW TO USE THE LOB DATA TYPE (C)』
- 『dtlob.sqc -- How to use the LOB data type (C)』
- 『dtlob.out -- HOW TO USE THE LOB DATA TYPE (C++)』
- 『dtlob.sqC -- How to use the LOB data type (C++)』
- 『dtLob.bas -- Get/set Large Objects (LOBs)』
- 『DtLob.java -- How to use LOB data type (JDBC)』
- 『DtLob.out -- HOW TO USE LOB DATA TYPE. Connect to 'sample' database using JDBC type 2 driver (JDBC)』
- 『lobfile.sqb -- Demonstrates the use of LOB file handles (IBM COBOL)』

CLOB 列からテキスト・ファイルへのデータの書き込み

データベースの外部の CLOB 列にあるデータにアクセスする必要がある場合、それをテキスト・ファイルに書き込みます。

以下の手順の例では、C の組み込み SQL を使用します。この例では、特定の再開 (empno = '000130') が CLOB 列から SELECT (選択) され、テキスト・ファイルに書き込まれます。

手順:

CLOB 列からテキスト・ファイルにデータを書き込むには、次のようにします。

1. CLOB FILE REFERENCE ホスト変数を宣言する。

```
EXEC SQL BEGIN DECLARE SECTION;
   SQL TYPE IS CLOB_FILE resume;
   char userid[9];
   char passwd[19];
   short lobind;
EXEC SQL END DECLARE SECTION;
```

   ホスト変数宣言セクションでは、以下のようになります。
   - `resume` は、CLOB 列から抽出したデータを入れるファイルを表す。
   - `userid` と `passwd` はユーザー ID とパスワードの組み合わせであり、アプリケーションをデータベースに接続するために必要となる。
2. アプリケーションをデータベースに接続する。
3. CLOB FILE REFERENCE ホスト変数をセットアップする。

```
strcpy (resume.name, "RESUME.TXT");
resume.name_length = strlen("RESUME.TXT");
resume.file_options = SQL_FILE_OVERWRITE;
```

strcpy 関数で提供されているパス記述では、次のようになります。

- RESUME.TXT は、表に挿入されるデータを持つファイルの名前である。

4. CLOB 列の再開フィールドからデータを SELECT (選択) し、指定したテキスト・ファイルに挿入する。

```
EXEC SQL SELECT resume INTO :resume :lobind FROM emp_resume
   WHERE resume_format='ascii' AND empno='000130';
```

5. プログラムを終了する。

関連概念:

- 244 ページの『ラージ・オブジェクト・ロケーター』
- 250 ページの『ラージ・オブジェクト・ファイル参照変数』

関連タスク:

- 「*アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」の『アプリケーションのデータベースへの接続』
- 「*アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」の『アプリケーション・プログラムの終了』
- 253 ページの『テキスト・ファイルから CLOB 列へのデータの挿入』

関連サンプル:

- 『dtlob.out -- HOW TO USE THE LOB DATA TYPE (C)』
- 『dtlob.sqc -- How to use the LOB data type (C)』
- 『dtlob.out -- HOW TO USE THE LOB DATA TYPE (C++)』
- 『dtlob.sqC -- How to use the LOB data type (C++)』
- 『dtLob.bas -- Get/set Large Objects (LOBs)』
- 『DtLob.java -- How to use LOB data type (JDBC)』
- 『DtLob.out -- HOW TO USE LOB DATA TYPE. Connect to 'sample' database using JDBC type 2 driver (JDBC)』
- 『lobfile.sqb -- Demonstrates the use of LOB file handles (IBM COBOL)』

テキスト・ファイルから CLOB 列へのデータの挿入

現在テキスト・ファイルにある CLOB データを処理するためのデータベースが必要な場合には、それを CLOB 列に挿入する必要があります。

UNIX ベースのファイル・システムの場合、例では C の組み込み SQL を使用します。

手順:

テキスト・ CLOB 列にデータを挿入するには、次のようにします。

1. CLOB FILE REFERENCE ホスト変数を宣言する。

```
EXEC SQL BEGIN DECLARE SECTION;
  SQL TYPE IS CLOB_FILE hv_text_file;
EXEC SQL END DECLARE SECTION;
```

hv_text_file はファイルを表します。

2. アプリケーションをデータベースに接続する。
3. CLOB FILE REFERENCE ホスト変数をセットアップする。

```
strcpy(hv_text_file.name, "/u/userid/dirname/filnam.1");
hv_text_file.name_length = strlen("/u/userid/dirname/filnam.1");
hv_text_file.file_options = SQL_FILE_READ;
```

strcpy 関数で提供されているパス記述では、次のようになります。

- userid は、1 ユーザー用のディレクトリーを表す。
- dirname は、″userid″ に属するサブディレクトリーを表す。
- filnam.1 は、表に挿入されるデータを持つファイルの名前である。
- clobtab は、CLOB データ型を使用した表の名前である。

4. データを hv_text_file から CLOB 表に挿入する。

```
EXEC SQL INSERT INTO CLOBTAB
  VALUES(:hv_text_file);
```

5. プログラムを終了する。

関連概念:

- 244 ページの『ラージ・オブジェクト・ロケーター』
- 250 ページの『ラージ・オブジェクト・ファイル参照変数』

関連タスク:

- 「*アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」の『アプリケーションのデータベースへの接続』
- 「*アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」の『アプリケーション・プログラムの終了』
- 252 ページの『CLOB 列からテキスト・ファイルへのデータの書き込み』

関連サンプル:

- 『dtlob.out -- HOW TO USE THE LOB DATA TYPE (C)』
- 『dtlob.sqc -- How to use the LOB data type (C)』
- 『dtlob.out -- HOW TO USE THE LOB DATA TYPE (C++)』
- 『dtlob.sqC -- How to use the LOB data type (C++)』
- 『dtLob.bas -- Get/set Large Objects (LOBs)』
- 『DtLob.java -- How to use LOB data type (JDBC)』
- 『DtLob.out -- HOW TO USE LOB DATA TYPE. Connect to 'sample' database using JDBC type 2 driver (JDBC)』
- 『lobfile.sqb -- Demonstrates the use of LOB file handles (IBM COBOL)』

第 7 章 ユーザー定義特殊タイプ

ユーザー定義タイプ

ユーザー定義タイプ (UDT) は既存のデータ型から派生するデータ型ですが、既存のデータ型とは別のもの、また互換性のないものと見なされます。 UDT を使用すると、DB2® ですでに使用可能な組み込みタイプを拡張して、独自のカスタマイズ・データ型を作成することができます。

ユーザー定義タイプには次の 2 つの種類があります。

- 特殊タイプ: 共通の表現を組み込みデータ型と共用します。
- 構造化型: それぞれがタイプを持っている一連の名前付き属性の表現を使用可能にします。タイプ階層を定義すれば、ある構造化型が別の構造化型 (スーパータイプと呼びます) のサブタイプになることができます。

関連概念:

- 255 ページの『ユーザー定義特殊タイプ』
- 271 ページの『ユーザー定義構造化型』

関連タスク:

- 258 ページの『特殊タイプの作成』

ユーザー定義特殊タイプ

特殊タイプとは、既存の DB2® 組み込みデータ型に基づいたユーザー定義タイプです。特殊タイプはその表現を既存のタイプ (ソース・タイプ) と内部的には共用しますが、それは別のもの、また互換性のないタイプと見なされます。

たとえば、特殊タイプは米ドルとカナダ・ドルなどの、様々な通貨を表すことができます。これらのタイプは、両方とも通貨を定義した組み込みタイプとして内部的に (ホスト言語プログラムでも) 表されています。たとえば、両方の通貨を DECIMAL として定義すると、それらはシステム内で 10 進データ型として表示されます。

さらに、DB2 にはラージ・オブジェクトを保管および操作するための組み込みタイプがあります。特殊タイプをそれらの中の 1 つのラージ・オブジェクト (LOB) データ型に基づいて定義すれば、これらを音声またはビデオ・ストリームなどを保管するために使用することが可能です。次の例は、AUDIO という名前の特殊タイプを作成する方法を示しています。

```
CREATE DISTINCT TYPE AUDIO AS BLOB (1M)
```

AUDIO は、組み込みデータ型 BLOB と同じ表現を持っていますが、 BLOB またはその他のタイプとは互換性のない別のタイプと見なされます。これにより、AUDIO 固有に作成された関数の作成が可能になり、それらの関数が他のタイプには適用されないことを保証します。

特殊タイプは、以下のような利点があります。

1. 拡張性: 新規のタイプを定義することにより、ユーザーのアプリケーションをサポートするために DB2 が提供するタイプのセットを増やすことができます。
2. 柔軟性: ユーザー定義関数 (UDF) を使用して新規のタイプにセマンティクスや振る舞いを指定し、システムで使用できるさまざまなタイプを増やすことができます。
3. 一貫性のある振る舞い: 強い型定義により、ご使用の特殊タイプが適切に振る舞うことが保証されます。また、ご使用の特殊タイプで定義された関数のみを特殊タイプのインスタンスに適用できることを保証します。
4. カプセル化: 特殊タイプに適用することができる一連の関数および演算子は、その特殊タイプの振る舞いを定義します。稼働中のアプリケーションはタイプに指定した内部表記に依存しないので、インプリメンテーションの点において柔軟性が得られます。
5. パフォーマンス: 特殊タイプがデータベース・マネージャーに高度に統合されています。特殊タイプは、内部的には、組み込みデータ型と同じ方法で表現されるので、組み込みデータ型の組み込み関数、比較演算子、索引などのコンポーネントをインプリメントする場合と同じ効率的なコードを使用します。

特殊タイプは修飾 ID によって識別されます。特殊タイプ名が CREATE DISTINCT TYPE、DROP DISTINCT TYPE、または COMMENT ON DISTINCT TYPE 以外のステートメントで使用されるときに、その特殊タイプ名を修飾するのにスキーマ名を使用しない場合、 SQL パスが順番に検索され、一致する特殊タイプを持つ最初のスキーマを探します。

LONG VARCHAR、LONG VARGRAPHIC、LOB タイプ、または DATALINK がソースになっている特殊タイプには、ソース・タイプと同じ制約が課されます。しかし、ユーザー定義関数を定義することにより、ソース・タイプの特定の関数または演算子を明示的に指定して、特殊タイプに適用することができます。 (これらの関数は、特殊タイプのソース・タイプに基づいて定義された関数をソースとしています。) ユーザー定義特殊タイプには比較演算子が自動的に生成されます。ただし、ソース・タイプとして LONG VARCHAR、 LONG VARGRAPHIC、 BLOB、 CLOB、 DBCLOB、または DATALINK を使用するものは除きます。さらに、ソース・タイプから特殊タイプへのキャスト、および特殊タイプからソース・タイプへのキャストをサポートする関数も生成されます。

関連概念:

- 257 ページの『ユーザー定義特殊タイプの強い型定義』
- 255 ページの『ユーザー定義タイプ』

関連タスク:

- 258 ページの『特殊タイプの作成』
- 259 ページの『特殊タイプに基づいた、列を持つ表の作成』
- 264 ページの『特殊タイプの操作』

関連サンプル:

- 『dtudt.out -- HOW TO CREATE/USE/DROP UDTs (C)』
- 『dtudt.sqc -- How to create, use, and drop user-defined distinct types (C)』
- 『dtudt.out -- HOW TO CREATE/USE/DROP UDTs (C++)』
- 『dtudt.sqC -- How to create, use, and drop user-defined distinct types (C++)』
- 『DtUdt.java -- How to create, use and drop user defined distinct types (JDBC)』
- 『DtUdt.out -- HOW TO CREATE, USE AND DROP USER DEFINED DISTINCT TYPES (JDBC)』
- 『DtUdt.out -- HOW TO CREATE, USE AND DROP USER DEFINED DISTINCT TYPES (SQLJ)』
- 『DtUdt.sqlj -- How to create, use and drop user defined distinct types (SQLj)』

ユーザー定義特殊タイプの強い型定義

特殊タイプに関連する最も重要な概念の 1 つに強い型定義があります。強い型定義を使うと、特殊タイプで明示的に定義された関数および演算子のみが確実にそのインスタンスに適用されるようにすることができます。

強い型定義は、特殊タイプのインスタンスの正確さを確実化するために重要です。たとえば、現行の為替相場に従って米ドルをカナダ・ドルに変換する関数を定義した場合、この同じ関数をユーロからカナダ・ドルへの変換には使用しないでしょう。なぜならこれは必ず誤った額を戻すからです。

強い型定義の結果、DB2® は、たとえば特殊タイプのインスタンスと特殊タイプのソース・タイプのインスタンスを比較するような照会を作成しません。 これと同じ理由で、DB2 は、別のタイプで定義された関数を特殊タイプに適用しません。特殊タイプのインスタンスを別のタイプのインスタンスと比較したい場合は、いずれか一方のタイプのインスタンスをキャストしなければなりません。同じ意味で、特殊タイプで定義されていない関数を特殊タイプのインスタンスに適用したい場合は、特殊タイプのインスタンスをこの関数のパラメーターのタイプにキャストしなければなりません。

関連概念:

- 255 ページの『ユーザー定義特殊タイプ』

関連タスク:

- 258 ページの『特殊タイプの作成』
- 259 ページの『特殊タイプに基づいた、列を持つ表の作成』

特殊タイプの作成

ユーザー定義特殊タイプは、既存のタイプ (整数、10 進数、または文字タイプ) から派生したデータ型です。特殊タイプを作成する際に、DB2 は cast 関数を生成します。これは特殊タイプからソース・タイプにキャストし、ソース・タイプから特殊タイプにキャストします。これらの関数は、照会中の特殊タイプを操作するために重要です。

WITH COMPARISONS 文節が CREATE DISTINCT TYPE ステートメントで指定されている場合 (手順のインスタンスにあるように)、同じ特殊タイプのインスタンスを相互に比較することができます。ソース・データ型がラージ・オブジェクト、DATALINK、LONG VARCHAR、または LONG VARGRAPHIC タイプの場合、WITH COMPARISONS 文節を指定することはできません。

前提条件:

特殊タイプを定義する必要のある特権のリストについては、 CREATE DISTINCT TYPE ステートメントを参照してください。

制約事項:

特殊タイプのソース・タイプは、 DB2 が特殊タイプを内部的に表現するために使用するデータ型です。したがって、組み込みデータ型でなければなりません。前に定義した特殊タイプを他の特殊タイプのソース・タイプとして使用することはできません。

手順:

特殊タイプを定義するには、タイプ名およびソース・タイプを指定して、 CREATE DISTINCT TYPE ステートメントを発行します。たとえば、次のステートメントは new_type という新規の特殊タイプを定義します。これには SMALLINT 値が含まれます。

```
CREATE DISTINCT TYPE new_type AS SMALLINT WITH COMPARISONS
```

上記のステートメントで定義された特殊タイプは SMALLINT に基づいているため、 WITH COMPARISONS パラメーターを指定する必要があります。

ユーザー定義特殊タイプのアプリケーションをさらに理解するために、サンプル・ビジネスに基づいた、以下の特殊タイプ定義の例を参照してください。

- 通貨に基づいた特殊タイプを定義する。
- ジョブ・アプリケーションの特殊タイプを定義する。

関連概念:

- 257 ページの『ユーザー定義特殊タイプの強い型定義』
- 255 ページの『ユーザー定義タイプ』
- 255 ページの『ユーザー定義特殊タイプ』

関連タスク:

- 261 ページの『通貨に基づいた特殊タイプの作成』

- 262 ページの『記入済みジョブ・アプリケーション・フォームの特殊タイプの作成』
- 264 ページの『特殊タイプの操作』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE DISTINCT TYPE ステートメント』

関連サンプル:

- 『dtudt.out -- HOW TO CREATE/USE/DROP UDTs (C)』
- 『dtudt.sqc -- How to create, use, and drop user-defined distinct types (C)』
- 『dtudt.out -- HOW TO CREATE/USE/DROP UDTs (C++)』
- 『dtudt.sqC -- How to create, use, and drop user-defined distinct types (C++)』
- 『DtUdt.java -- How to create, use and drop user defined distinct types (JDBC)』
- 『DtUdt.out -- HOW TO CREATE, USE AND DROP USER DEFINED DISTINCT TYPES (JDBC)』
- 『DtUdt.out -- HOW TO CREATE, USE AND DROP USER DEFINED DISTINCT TYPES (SQLJ)』
- 『DtUdt.sqlj -- How to create, use and drop user defined distinct types (SQLj)』

特殊タイプに基づいた、列を持つ表の作成

特殊タイプを定義した後、その特殊タイプに基づいた、列を持つ表を作成することができます。

前提条件:

特殊タイプを定義する必要のある特権のリストについては、 CREATE DISTINCT TYPE ステートメントを参照してください。

表を作成する必要のある特権のリストについては、 CREATE TABLE ステートメントを参照してください。

手順:

特殊タイプに基づいた、列を持つ表を作成するには、次のようにします。

1. 特殊タイプを定義する。

```
CREATE DISTINCT TYPE t_educ AS SMALLINT WITH COMPARISONS
```

2. 列タイプに特殊タイプ T_EDUC という名前を付けて表を作成する。

```
CREATE TABLE employee
  (empno CHAR(6) NOT NULL,
   firstnme VARCHAR(12) NOT NULL,
   lastname VARCHAR(15) NOT NULL,
   workdept CHAR(3),
   phoneno CHAR(4),
   photo BLOB(10M) NOT NULL,
   edlevel T_EDUC)
  IN RESOURCE
```

表のアプリケーションをさらに理解するために、サンプル・ビジネスに基づいた、以下の表作成の例を参照してください。

- 地域別売上を追跡するするための表を作成する。
- 記入済みジョブ・アプリケーション・フォームを保管するための表を作成する。

関連概念:

- 257 ページの『ユーザー定義特殊タイプの強い型定義』
- 255 ページの『ユーザー定義タイプ』
- 255 ページの『ユーザー定義特殊タイプ』

関連タスク:

- 263 ページの『地域別売上を追跡するための表の作成』
- 263 ページの『記入済みジョブ・アプリケーション・フォームを保管するための表の作成』
- 258 ページの『特殊タイプの作成』
- 264 ページの『特殊タイプの操作』
- 261 ページの『通貨に基づいた特殊タイプの作成』
- 262 ページの『記入済みジョブ・アプリケーション・フォームの特殊タイプの作成』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE DISTINCT TYPE ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE TABLE ステートメント』

ユーザー定義タイプのドロップ

DROP ステートメントを使用して、ユーザー定義タイプ (UDT) をドロップすることができます。次の場合には UDT をドロップすることはできません。

- 既存の表またはビューの列定義で使用されている場合。
- 既存の型付き表または型付きビューのタイプとして使用されている場合。
- 別の構造化型のスーパータイプとして使用されている場合。

データベース・マネージャーは、この UDT に従属するすべてのルーチンをドロップしようとします。ビュー、トリガー、表チェック制約、または別のルーチンが従属しているルーチンをドロップすることはできません。 DB2 が従属ルーチンをドロップできない場合は、DB2 は UDT をドロップしません。 UDT をドロップすると、それを使用していたパッケージまたはキャッシュされた動的 SQL すべてが無効になります。

ある UDT のための transform を作成してある場合で、その UDT をドロップしようとする場合は、関連した transform をドロップすることを考慮してください。transform をドロップするには、DROP TRANSFORM ステートメントを発行します。ドロップできるのはユーザー定義の変形体だけです。組み込み transform またはそれに関連したグループ定義をドロップすることはできません。

関連概念:

- 255 ページの『ユーザー定義タイプ』
- 255 ページの『ユーザー定義特殊タイプ』
- 271 ページの『ユーザー定義構造化型』
- 312 ページの『Transform 関数と Transform グループ』

関連タスク:

- 258 ページの『特殊タイプの作成』
- 272 ページの『構造化型の作成』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『DROP ステートメント』

関連サンプル:

- 『dtstruct.out -- Sample C++ program : dtstruct.sqC (C++)』
- 『dtstruct.sqC -- Create, use, drop a hierarchy of structured types and typed tables (C++)』
- 『dtudt.out -- HOW TO CREATE/USE/DROP UDTs (C++)』
- 『dtudt.sqC -- How to create, use, and drop user-defined distinct types (C++)』
- 『dtudt.out -- HOW TO CREATE/USE/DROP UDTs (C)』
- 『dtudt.sqc -- How to create, use, and drop user-defined distinct types (C)』
- 『DtUdt.java -- How to create, use and drop user defined distinct types (JDBC)』
- 『DtUdt.out -- HOW TO CREATE, USE AND DROP USER DEFINED DISTINCT TYPES (JDBC)』
- 『DtUdt.out -- HOW TO CREATE, USE AND DROP USER DEFINED DISTINCT TYPES (SQLJ)』
- 『DtUdt.sqlj -- How to create, use and drop user defined distinct types (SQLj)』

通貨に基づいた特殊タイプの作成

さまざまな通貨を扱う必要のあるアプリケーションを作成するとします。異なる通貨の値を比較する際に変換が必要な場合、これらの通貨を互いに直接比較したり操作するのを DB2 が許可しないようにします。特殊タイプは特殊タイプとのみ互換性があるため、表現する必要のある通貨ごとに 1 つずつ定義しなければなりません。

前提条件:

特殊タイプを定義する必要のある特権のリストについては、 CREATE DISTINCT TYPE ステートメントを参照してください。

手順:

ユーロ、米国通貨、およびカナダ通貨を表す特殊タイプを定義するには、以下のステートメントを発行します。

```
CREATE DISTINCT TYPE US_DOLLAR AS DECIMAL (9,3) WITH COMPARISONS
CREATE DISTINCT TYPE CANADIAN_DOLLAR AS DECIMAL (9,3) WITH COMPARISONS
CREATE DISTINCT TYPE EURO AS DECIMAL (9,3) WITH COMPARISONS
```

比較演算子は DECIMAL (9,3) でサポートされるため、 WITH COMPARISONS 文節を指定する必要があることに注意してください。

関連概念:

- 255 ページの『ユーザー定義特殊タイプ』

関連タスク:

- 259 ページの『特殊タイプに基づいた、列を持つ表の作成』
- 262 ページの『記入済みジョブ・アプリケーション・フォームの特殊タイプの作成』
- 263 ページの『地域別売上を追跡するための表の作成』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE DISTINCT TYPE ステートメント』

記入済みジョブ・アプリケーション・フォームの特殊タイプの作成

送られてきたジョブ・アプリケーション・フォームを DB2 表に保持し、それらのフォームから情報を抽出する関数を使用するとします。その場合は、そのフォームを表す特殊タイプを表に定義し、さらに関数のパラメーターとして定義することができます。

前提条件:

特殊タイプを定義する必要のある特権のリストについては、 CREATE DISTINCT TYPE ステートメントを参照してください。

手順:

記入済みジョブ・アプリケーション・フォームを表す特殊タイプを定義するには、次のステートメントを発行します。

```
CREATE DISTINCT TYPE PERSONNEL.APPLICATION_FORM AS CLOB(32K)
```

DB2 は CLOB での比較をサポートしないため、WITH COMPARISONS 文節を指定することはできません。上記のステートメントでは PERSONNEL スキーマが指定されます。なぜなら、このスキーマはアプリケーション・フォームを扱うすべての特殊タイプおよび UDF を含むように意図されているからです。

関連概念:

- 255 ページの『ユーザー定義特殊タイプ』

関連タスク:

- 259 ページの『特殊タイプに基づいた、列を持つ表の作成』
- 261 ページの『通貨に基づいた特殊タイプの作成』
- 263 ページの『記入済みジョブ・アプリケーション・フォームを保管するための表の作成』

地域別売上を追跡するための表の作成

さまざまな地域における会社の売上を追跡する表を定義するとします。該当する通貨特殊タイプを特定の地域の合計売上の列タイプとして使用して、表を作成することができます。

前提条件:

表を作成する必要のある特権のリストについては、 CREATE TABLE ステートメントを参照してください。

手順:

地域別売上を追跡するための表を作成するには、次のようにします。

1. 通貨に基づいた特殊タイプを作成する。
2. 以下の CREATE TABLE ステートメントを発行する。

```
CREATE TABLE US_SALES
  (PRODUCT_ITEM  INTEGER,
   MONTH         INTEGER CHECK (MONTH BETWEEN 1 AND 12),
   YEAR          INTEGER CHECK (YEAR > 1985),
   TOTAL         US_DOLLAR)

CREATE TABLE CANADIAN_SALES
  (PRODUCT_ITEM  INTEGER,
   MONTH         INTEGER CHECK (MONTH BETWEEN 1 AND 12),
   YEAR          INTEGER CHECK (YEAR > 1985),
   TOTAL         CANADIAN_DOLLAR)

CREATE TABLE GERMAN_SALES
  (PRODUCT_ITEM  INTEGER,
   MONTH         INTEGER CHECK (MONTH BETWEEN 1 AND 12),
   YEAR          INTEGER CHECK (YEAR > 1985),
   TOTAL         EURO)
```

関連概念:

- 255 ページの『ユーザー定義特殊タイプ』

関連タスク:

- 258 ページの『特殊タイプの作成』
- 261 ページの『通貨に基づいた特殊タイプの作成』
- 263 ページの『記入済みジョブ・アプリケーション・フォームを保管するための表の作成』

記入済みジョブ・アプリケーション・フォームを保管するための表の作成

応募者が記入したフォームを保持するための表を定義する必要があるとします。記入済みフォームを入れるための列タイプとして特殊タイプ PERSONNEL.APPLICATION_FORM を使用して、表を作成することができます。

前提条件:

表を作成する必要のある特権のリストについては、 CREATE TABLE ステートメントを参照してください。

手順:

記入済みジョブ・アプリケーション・フォームを入れる表を作成するには、次のようにします。

1. ジョブ・アプリケーション・フォームの特殊タイプを作成する。
2. 以下の CREATE TABLE ステートメントを発行する。

```
CREATE TABLE APPLICATIONS
  (ID               SYSIBM.INTEGER,
   NAME             VARCHAR (30),
   APPLICATION_DATE  SYSIBM.DATE,
   FORM             PERSONNEL.APPLICATION_FORM)
```

特殊タイプ名の修飾子が許可 ID と異なるため、特殊タイプ名は完全修飾名となっており、デフォルトの関数パスは変更されていません。なお、タイプおよび関数名が完全には修飾されていない場合、 DB2 は現行の関数パスにリストされているスキーマ全体を検索し、所定の修飾された名前に一致するタイプまたは関数名を探します。 SYSIBM は常に (それが省略されていれば) 現行関数パスにおいて考慮されるので、組み込みデータ型の修飾を省略できます。

関連概念:

- 255 ページの『ユーザー定義特殊タイプ』

関連タスク:

- 258 ページの『特殊タイプの作成』
- 262 ページの『記入済みジョブ・アプリケーション・フォームの特殊タイプの作成』
- 263 ページの『地域別売上を追跡するための表の作成』

特殊タイプの操作

特殊タイプの操作

特殊タイプを定義し、それらに基づく表を作成したら、実際の特殊型付き値を操作することができます。

手順:

様々な種類の特殊タイプ操作をインプリメントするには、次のようにします。

- 特殊タイプの間でキャストする。
- 特殊タイプの間で比較を実行する。
- 特殊タイプと定数の間で比較を実行する。
- ソース派生 UDF を特殊タイプに定義する。
- 特殊タイプを含む割り当てを実行する。
- 特殊タイプを含む割り当てを動的 SQL で実行する。
- 別の特殊タイプを含む割り当てを実行する。
- 特殊型付き列に対して UNION 操作を実行する。

関連概念:

- 255 ページの『ユーザー定義特殊タイプ』

関連タスク:

- 265 ページの『特殊タイプの間のキャスト』
- 266 ページの『特殊タイプがかかわる比較の実行』
- 267 ページの『特殊タイプと定数の比較の実行』
- 270 ページの『特殊タイプへのソース派生 UDF の定義』
- 267 ページの『組み込み SQL による、特殊タイプを含む割り当ての実行』
- 268 ページの『動的 SQL による、特殊タイプを含む割り当ての実行』
- 269 ページの『さまざまな特殊タイプがかかわる割り当ての実行』
- 270 ページの『特殊型付き列での UNION 操作の実行』
- 258 ページの『特殊タイプの作成』
- 259 ページの『特殊タイプに基づいた、列を持つ表の作成』

関連サンプル:

- 『dtudt.c -- How to create, use, and drop user-defined distinct types.』
- 『dtudt.out -- HOW TO CREATE/USE/DROP UDTs (C)』
- 『dtudt.sqc -- How to create, use, and drop user-defined distinct types (C)』
- 『dtudt.out -- HOW TO CREATE/USE/DROP UDTs (C++)』
- 『dtudt.sqC -- How to create, use, and drop user-defined distinct types (C++)』
- 『DtUdt.java -- How to create, use and drop user defined distinct types (JDBC)』
- 『DtUdt.out -- HOW TO CREATE, USE AND DROP USER DEFINED DISTINCT TYPES (JDBC)』
- 『DtUdt.out -- HOW TO CREATE, USE AND DROP USER DEFINED DISTINCT TYPES (SQLJ)』
- 『DtUdt.sqlj -- How to create, use and drop user defined distinct types (SQLj)』

特殊タイプの間のキャスト

ある通貨を米ドルに変換する UDF を定義するとします。この例の目的は、次のような表から現行の為替相場を求めることです。

```
CREATE TABLE
  exchange_rates(source CHAR(3), target CHAR(3), rate DECIMAL(9,3))
```

次の関数を使用して、exchange_rates 表に直接アクセスすることができます。

```
CREATE FUNCTION exchange_rate(src VARCHAR(3), trg VARCHAR(3))
  RETURNS DECIMAL(9,3)
  RETURN SELECT rate FROM exchange_rates
    WHERE source = src AND target = trg
```

上記の関数の通貨為替相場は、特殊タイプではなく DECIMAL タイプに基づいています。異なる通貨を表現するには、以下の特殊タイプ定義を使用してください。

```
CREATE DISTINCT TYPE CANADIAN_DOLLAR AS DECIMAL (9,3) WITH COMPARISONS
CREATE DISTINCT TYPE EURO AS DECIMAL(9,3) WITH COMPARISONS
CREATE DISTINCT TYPE US_DOLLAR AS DECIMAL (9,3) WITH COMPARISONS
```

CANADIAN_DOLLAR または EURO を US_DOLLAR に変換する UDF を作成するには、関係する値をキャストする必要があります。 exchange_rate 関数は為替相場として DECIMAL を戻すことに注意してください。たとえば、値 CANADIAN_DOLLAR を US_DOLLAR に変換する関数は、以下のステップを実行します。

- CANADIAN_DOLLAR 値を DECIMAL にキャストする。
- カナダ・ドルを米ドルに変換するための為替相場を exchange_rate 関数から取得する。この関数は為替相場を DECIMAL 値をとして戻します。
- カナダ・ドル DECIMAL の値に DECIMAL 為替相場を乗算する。
- この DECIMAL 値を US_DOLLAR にキャストする。
- US_DOLLAR 値を戻す。

以下は、US_DOLLAR 関数のインスタンスです (カナダ・ドルとユーロの両方)。これは上記のステップに従っています。

```
CREATE FUNCTION US_DOLLAR(amount CANADIAN_DOLLAR)
  RETURNS US_DOLLAR
  RETURN US_DOLLAR(DECIMAL(amount) * exchange_rate('CAN', 'USD'))

CREATE FUNCTION US_DOLLAR(amount EURO)
  RETURNS US_DOLLAR
  RETURN US_DOLLAR(DECIMAL(amount) * exchange_rate('EUR', 'USD'))
```

関連概念:

- 255 ページの『ユーザー定義特殊タイプ』

関連タスク:

- 258 ページの『特殊タイプの作成』
- 259 ページの『特殊タイプに基づいた、列を持つ表の作成』
- 270 ページの『特殊タイプへのソース派生 UDF の定義』

特殊タイプがかかわる比較の実行

たとえば、1999 年 7 月 (7/99) に米国において、カナダおよびドイツよりも売上の多かった製品を知りたいとします。

```
SELECT US.PRODUCT_ITEM, US.TOTAL
  FROM US_SALES AS US, CANADIAN_SALES AS CDN, GERMAN_SALES AS GERMAN
  WHERE US.PRODUCT_ITEM = CDN.PRODUCT_ITEM
  AND US.PRODUCT_ITEM = GERMAN.PRODUCT_ITEM
  AND US.TOTAL > US_DOLLAR (CDN.TOTAL)
  AND US.TOTAL > US_DOLLAR (GERMAN.TOTAL)
  AND US.MONTH = 7
  AND US.YEAR  = 1999
  AND CDN.MONTH = 7
  AND CDN.YEAR  = 1999
  AND GERMAN.MONTH = 7
  AND GERMAN.YEAR  = 1999
```

米ドルは、カナダ・ドルやユーロと直接比較できないので、カナダ・ドルの額を米ドルにキャストする UDF や、ユーロの額を米ドルにキャストする UDF を使用します。これらすべてを DECIMAL にキャストし、変換された DECIMAL の値を比較することはできません。すなわち、各総額が同じ通貨でないので貨幣上比較できないからです。

関連概念:

- 255 ページの『ユーザー定義特殊タイプ』

関連タスク:

- 258 ページの『特殊タイプの作成』
- 259 ページの『特殊タイプに基づいた、列を持つ表の作成』
- 265 ページの『特殊タイプの間のキャスト』

特殊タイプと定数の比較の実行

たとえば、1999 年 7 月 (7/99) に米国で 100,000.00 米ドルを超える売上があった商品を知りたいとします。

```
SELECT PRODUCT_ITEM
  FROM   US_SALES
  WHERE  TOTAL > US_DOLLAR (100000)
  AND    month = 7
  AND    year  = 1999
```

米ドルとそのソース・タイプのインスタンス (すなわち DECIMAL) を直接比較できないので、 DB2 により提供されるキャスト機能を使用して DECIMAL から米ドルにキャストしました。また、DB2 により提供されるもう 1 つのキャスト機能 (すなわち、米ドルから DECIMAL にキャストする機能) を使用して、列の合計を DECIMAL にキャストすることもできます。特殊タイプからのキャストの場合はキャストを実行するキャスト指定表記法を、特殊タイプへのキャストの場合は機能表記法をそれぞれ使用することができます。すなわち、上記の照会は以下のように作成してもかまいません。

```
SELECT PRODUCT_ITEM
  FROM   US_SALES
  WHERE  TOTAL > CAST (100000 AS us_dollar)
  AND    MONTH = 7
  AND    YEAR  = 1999
```

関連概念:

- 255 ページの『ユーザー定義特殊タイプ』

関連タスク:

- 258 ページの『特殊タイプの作成』
- 259 ページの『特殊タイプに基づいた、列を持つ表の作成』
- 265 ページの『特殊タイプの間のキャスト』

組み込み SQL による、特殊タイプを含む割り当ての実行

新規の応募者が記入したジョブ・アプリケーション・フォームをデータベースに保管するとします。記入されたフォームを表すために使用する文字ストリングの値を含むホスト変数を、次のように定義することができます。

```
EXEC SQL BEGIN DECLARE SECTION;
  SQL TYPE IS CLOB(32K) hv_form;
EXEC SQL END DECLARE SECTION;

/* Code to fill hv_form */
```

```
INSERT INTO APPLICATIONS
  VALUES (134523, 'Peter Holland', CURRENT DATE, :hv_form)
```

DB2 が、特殊タイプのソース・タイプのインスタンスをその特殊タイプを持つターゲットに割り当てるようにさせるため、 cast 関数を明示的に呼び出して、ホスト変数を特殊タイプ personal.application_form に変換することはありません。

関連概念:

- 255 ページの『ユーザー定義特殊タイプ』

関連タスク:

- 258 ページの『特殊タイプの作成』
- 259 ページの『特殊タイプに基づいた、列を持つ表の作成』
- 270 ページの『特殊タイプへのソース派生 UDF の定義』

動的 SQL による、特殊タイプを含む割り当ての実行

新規の応募者が記入したジョブ・アプリケーション・フォームをデータベースに保管するとします。記入されたフォームを表すために使用する文字ストリングの値を含むホスト変数を、次のように定義してあります。動的 SQL を使用するには、以下のようにパラメーター・マーカーを使用することができます。

```
EXEC SQL BEGIN DECLARE SECTION;
  long id;
  char name[30];
  SQL TYPE IS CLOB(32K) form;
  char command[80];
EXEC SQL END DECLARE SECTION;

/* Code to fill host variables */

strcpy(command,"INSERT INTO APPLICATIONS VALUES");
strcat(command,"(?, ?, CURRENT DATE, CAST (? AS CLOB(32K)))");

EXEC SQL PREPARE APP_INSERT FROM :command;
EXEC SQL EXECUTE APP_INSERT USING :id, :name, :form;
```

DB2 のキャスト指定を使用して、パラメーター・マーカーのタイプが CLOB(32K)、すなわち特殊タイプ列に割り当て可能なタイプであることを DB2 に通知します。なお、ホスト言語は特殊タイプをサポートしないので、特殊タイプのホスト変数を宣言することはできません。そのため、パラメーター・マーカーのタイプを特殊タイプと指定することはできません。

関連概念:

- 255 ページの『ユーザー定義特殊タイプ』

関連タスク:

- 258 ページの『特殊タイプの作成』
- 259 ページの『特殊タイプに基づいた、列を持つ表の作成』

さまざまな特殊タイプがかかわる割り当ての実行

米ドルおよびカナダ・ドルの SUM をサポートする SUM 組み込み関数に、 2 つのソース派生 UDF を定義したとします。

```
CREATE FUNCTION SUM (CANADIAN_DOLLAR)
  RETURNS CANADIAN_DOLLAR
  SOURCE SYSIBM.SUM (DECIMAL())

CREATE FUNCTION SUM (US_DOLLAR)
  RETURNS US_DOLLAR
  SOURCE SYSIBM.SUM (DECIMAL())
```

ここで、各地域における各製品についての 1 年間の総売上を、米ドルで別々の表に保存するように上司から要求されたとします。

```
CREATE TABLE US_SALES_94
  (PRODUCT_ITEM  INTEGER,
   TOTAL         US_DOLLAR)

CREATE TABLE GERMAN_SALES_94
  (PRODUCT_ITEM  INTEGER,
   TOTAL         US_DOLLAR)

CREATE TABLE CANADIAN_SALES_94
  (PRODUCT_ITEM  INTEGER,
   TOTAL         US_DOLLAR)

INSERT INTO US_SALES_94
  SELECT PRODUCT_ITEM, SUM (TOTAL)
  FROM US_SALES
  WHERE YEAR = 1994
  GROUP BY PRODUCT_ITEM

INSERT INTO GERMAN_SALES_94
  SELECT PRODUCT_ITEM, US_DOLLAR (SUM (TOTAL))
  FROM GERMAN_SALES
  WHERE YEAR = 1994
  GROUP BY PRODUCT_ITEM

INSERT INTO CANADIAN_SALES_94
  SELECT PRODUCT_ITEM, US_DOLLAR (SUM (TOTAL))
  FROM CANADIAN_SALES
  WHERE YEAR = 1994
  GROUP BY PRODUCT_ITEM
```

異なる特殊タイプを直接互いに割り当てることはできないため、カナダ・ドルおよびユーロの金額を米ドルに明示的に変換します。特殊タイプはそれ自体のソース・タイプにしかキャストできないので、キャスト指定構文を使用できません。

関連概念:

- 255 ページの『ユーザー定義特殊タイプ』

関連タスク:

- 258 ページの『特殊タイプの作成』
- 259 ページの『特殊タイプに基づいた、列を持つ表の作成』

特殊型付き列での UNION 操作の実行

たとえば、米国のユーザーに自分の会社の全製品の総売上を含むビューを提供したいとします。

```
CREATE VIEW ALL_SALES AS
  SELECT PRODUCT_ITEM, MONTH, YEAR, TOTAL
  FROM US_SALES
  UNION
  SELECT PRODUCT_ITEM, MONTH, YEAR, US_DOLLAR (TOTAL)
  FROM CANADIAN_SALES
  UNION
  SELECT PRODUCT_ITEM, MONTH, YEAR, US_DOLLAR (TOTAL)
  FROM GERMAN_SALES
```

特殊タイプは同じ特殊タイプとしか互換性を持たない union であるため、カナダ・ドルを米ドルに、そしてユーロを米ドルにキャストします。上記の例では、特殊タイプの間のキャスト で定義された UDF を使用して通貨間をキャストします。このため、キャスト指定ではなく機能的な表記が使用されます。

関連概念:

- 255 ページの『ユーザー定義特殊タイプ』

関連タスク:

- 258 ページの『特殊タイプの作成』
- 259 ページの『特殊タイプに基づいた、列を持つ表の作成』
- 265 ページの『特殊タイプの間のキャスト』

特殊タイプへのソース派生 UDF の定義

たとえば、ユーロの SUM をサポートする SUM 組み込み関数にソース派生 UDF を定義したとします。

```
CREATE FUNCTION SUM (EUROS)
  RETURNS EUROS
  SOURCE SYSIBM.SUM (DECIMAL())
```

1994 年のドイツにおける各製品の売上総額を知りたいとします。米ドルで総売上を求めます。

```
SELECT PRODUCT_ITEM, US_DOLLAR (SUM (TOTAL))
  FROM GERMAN_SALES
  WHERE YEAR = 1994
  GROUP BY PRODUCT_ITEM
```

上記と同じ方法で米ドルに SUM 関数を定義していないと、 `SUM (us_dollar (total))` を定義できません。

関連概念:

- 255 ページの『ユーザー定義特殊タイプ』

関連タスク:

- 258 ページの『特殊タイプの作成』
- 259 ページの『特殊タイプに基づいた、列を持つ表の作成』
- 267 ページの『組み込み SQL による、特殊タイプを含む割り当ての実行』

第 8 章 ユーザー定義構造化型

ユーザー定義構造化型

構造化型は、1 つ以上の名前付き属性を含み、それぞれがデータ型を持っている、ユーザー定義データ型です。属性は、タイプのインスタンスを記述するプロパティーです。たとえば、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。人物には、名前、住所などの属性があります。部門には、名前または他の種類の ID などの属性があります。

また、構造化型にはメソッド仕様の集合が含まれています。メソッドは、構造化型の動作を定義する手段になります。ユーザー定義関数 (UDF) と同様に、メソッドは SQL を拡張したルーチンです。しかし、メソッドの場合、動作は特定の構造化型にのみ統合されます。

構造化型は、表、ビュー、または列のタイプとして使用されます。表またはビューのタイプとして使用される場合、その表は型付き表、そのビューは型付きビューと呼ばれます。型付き表および型付きビューの場合、構造化型の属性の名前およびデータ型は、型付き表または型付きビューの列の名前およびデータ型になります。型付き表または型付きビューの行を、構造化型のインスタンスの表現と見なすことができます。

その他のオブジェクトがタイプを直接的または間接的に使用する場合、そのタイプをドロップすることはできません。たとえば、表またはビューの列が直接または間接的にタイプを使用する場合、タイプをドロップすることはできません。

関連概念:

- 255 ページの『ユーザー定義タイプ』
- 282 ページの『型付き表』
- 296 ページの『型付きビュー』

関連タスク:

- 272 ページの『構造化型の作成』
- 273 ページの『構造化型のインスタンスの保管』
- 277 ページの『構造化型の振る舞いの定義』
- 260 ページの『ユーザー定義タイプのドロップ』

関連サンプル:

- 『dtstruct.out -- Sample C++ program : dtstruct.sqC (C++)』
- 『dtstruct.sqC -- Create, use, drop a hierarchy of structured types and typed tables (C++)』

構造化型の作成

構造化型は、1 つ以上の名前付き属性を含み、それぞれが独自の名前とデータ型を持っている、ユーザー定義データ型です。構造化型は表またはビューのタイプを表すことができます。表の中のそれぞれの列の名前およびデータ型は構造化型の属性の 1 つから派生しています。また、構造化型は列のタイプまたはルーチンに対する引き数のタイプを表すこともできます。

前提条件:

構造化型を定義する必要のある特権のリストについては、 CREATE TYPE ステートメントを参照してください。

手順:

年齢および住所属性を持つ人物を表す構造化型を定義するには、以下のステートメントを発行します。

```
CREATE TYPE Person_t AS
   (Name VARCHAR(20),
   Age INT,
   Address Address_t)
   INSTANTIABLE
   REF USING VARCHAR(13) FOR BIT DATA
   MODE DB2SQL;
```

特殊タイプとは異なり、構造化型の属性は組み込み DB2 データ型以外のタイプにすることができます。上記の型宣言には Address という属性が含まれており、そのソース・タイプは別の構造化型である Address_t です。

関連概念:

- 255 ページの『ユーザー定義特殊タイプ』
- 271 ページの『ユーザー定義構造化型』
- 274 ページの『構造化型階層』

関連タスク:

- 273 ページの『構造化型のインスタンスの保管』
- 276 ページの『構造化型階層の作成』
- 260 ページの『ユーザー定義タイプのドロップ』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE TYPE (構造化) ステートメント』

関連サンプル:

- 『dtstruct.out -- Sample C++ program : dtstruct.sqC (C++)』
- 『dtstruct.sqC -- Create, use, drop a hierarchy of structured types and typed tables (C++)』

構造化型のインスタンスの保管

構造化型インスタンスは、次の 2 通りの方法でデータベースに保管することができます。

- 表のそれぞれの列が構造化型のインスタンスの属性になっている表の中に行として保管する。他の表からインスタンスを参照する必要がある場合は、型付き表を使用しなければなりません。オブジェクトを表の中に行として保管するには、表定義の中で個々の列を指定するのではなく、次のように構造化型を使用して表を定義します。

  ```
  CREATE TABLE Person OF Person_t
     ...
  ```

  表の中のそれぞれの列の名前とデータ型は、指示されている構造化型の属性のいずれか 1 つから派生したものです。このような表を、型付き表といいます。
- 列の値としての保管。オブジェクトを表の列に保管するには、列を構造化型として定義します。次のステートメントは、 Address_t 構造化型に属する構造化型 Address を持つ Properties 表を作成します。

```
CREATE TABLE Properties
   (ParcelNum INT,
   Photo BLOB(2K),
   Address Address_t)
   ...
```

関連概念:

- 271 ページの『ユーザー定義構造化型』
- 282 ページの『型付き表』

関連タスク:

- 287 ページの『型付き行へのオブジェクトの保管』
- 304 ページの『表列への構造化型オブジェクトの保管』

構造化型でのインスタンス生成可能性

タイプは、*INSTANTIABLE* または *NOT INSTANTIABLE* として定義することもできます。デフォルトでは、タイプはインスタンス生成可能となっています。これはオブジェクトのインスタンスが生成可能であるという意味です。一方、非インスタンス生成可能タイプは、タイプ階層をより細かく区分することを意図して作られたモデルです。たとえば、NOT INSTANTIABLE 文節を使用して Person_t を定義する場合は、人物のインスタンスをデータベースに保管することはできませんし、Person_t を使用して表またはビューを作成することもできません。保管できるのは Employee_t または開発者が定義する Person_t のその他のサブタイプのインスタンスだけです。

関連概念:

- 271 ページの『ユーザー定義構造化型』

関連タスク:

- 272 ページの『構造化型の作成』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE TYPE (構造化) ステートメント』

構造化型階層

従来のリレーショナル表とリレーショナル列を使用して、人物などのオブジェクトをモデル化することは確かに可能です。ただし、構造化型には*継承* という追加のプロパティーが用意されています。つまり、構造化型は、構造化型のすべての属性を再利用し、しかもサブタイプに固有の追加の属性を含むサブタイプ を持つことができるのです。元の構造化型は、*スーパータイプ* です。たとえば、構造化型 Person_t に、 Name、Age、および Address の属性があるとします。 Person_t はサブタイプとして、 Name、Age、および Address のすべての属性を含み、さらに SerialNum、Salary、および BusinessUnit の属性を含む Employee_t をもつことができます。

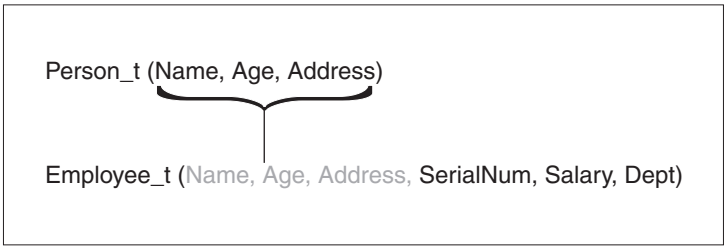


図 3. 構造化型 Employee_t はスーパータイプ Person_t から属性を継承する

同じスーパータイプに基づく (いくつかのレベルの) サブタイプのセットをタイプ階層と呼びます。たとえば、データ・モデルが、管理者という特殊な従業員を表す必要があるとします。管理者には、管理者ではない従業員よりも多くの属性があります。 `Manager_t` タイプは、従業員のために定義されている属性を継承していますが、管理者だけに適用される特別賞与属性など、いくつかの独自の追加の属性によっても定義されています。

次の図は、人物タイプおよび従業員タイプから派生したさまざまなサブタイプを表します。

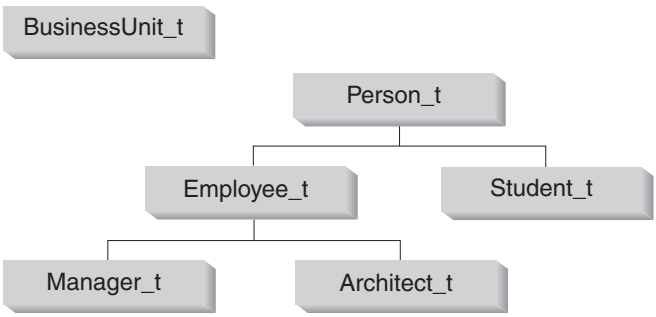


図 4. タイプ階層 (BusinessUnit_t と Person_t)

図 4 では、人物タイプ `Person_t` は、この階層の*ルート・タイプ* です。 `Person_t` は、下位のタイプ (ここでは、`Employee_t` という名前のタイプと `Student_t` という名前のタイプ) のスーパータイプでもあります。サブタイプとスーパータイプとの間には推移的な関係があります。つまり、サブタイプとスーパータイプとの間の関係は、タイプ階層の全体に存在するということです。したがって、`Person_t` は、`Manager_t` と `Architect_t` のスーパータイプでもあるのです。

部門タイプ `BusinessUnit_t` は小規模なタイプ階層です。これはサブタイプを持たない階層のルートです。

関連概念:

- 271 ページの『ユーザー定義構造化型』

関連タスク:

- 272 ページの『構造化型の作成』
- 276 ページの『構造化型階層の作成』

構造化型階層の作成

以下の図は構造化型階層を示しています。

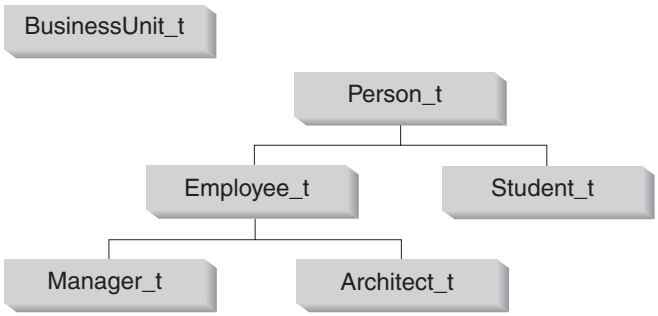


図 5. タイプ階層 (BusinessUnit_t と Person_t)

BusinessUnit_t タイプを作成するには、以下の CREATE TYPE SQL ステートメントを発行します。

```
CREATE TYPE BusinessUnit_t AS
   (Name VARCHAR(20),
   Headcount INT)
   MODE DB2SQL;
```

Person_t タイプ階層を作成するには、以下の SQL ステートメントを発行します。

```
CREATE TYPE Person_t AS
   (Name VARCHAR(20),
   Age INT,
   Address Address_t)
   REF USING VARCHAR(13) FOR BIT DATA
   MODE DB2SQL;

CREATE TYPE Employee_t UNDER Person_t AS
   (SerialNum INT,
   Salary DECIMAL (9,2),
   Dept REF(BusinessUnit_t))
   MODE DB2SQL;

CREATE TYPE Student_t UNDER Person_t AS
   (SerialNum CHAR(6),
   GPA DOUBLE)
   MODE DB2SQL;

CREATE TYPE Manager_t UNDER Employee_t AS
   (Bonus DECIMAL (7,2))
   MODE DB2SQL;

CREATE TYPE Architect_t UNDER Employee_t AS
   (StockOption INTEGER)
   MODE DB2SQL;
```

Person_t は、Name、 Age および Address という 3 つの属性を持っています。 2 つのサブタイプ Employee_t と Student_t は、両方とも Person_t の属性を継承していて、それぞれのタイプに固有のいくつかの追加の属性も持っています。たとえば、従業員と生徒には両方とも通し番号が振られていますが、生徒の通し番号に使用される形式は、従業員の通し番号に使用される形式とは異なります。

最後に、Manager_t と Architect_t は両方とも、 Employee_t のサブタイプです。つまり、これらは Employee_t のすべての属性を継承していて、その属性をそれぞれのタイプに適合するよう拡張しているのです。したがって、タイプ Manager_t のインスタンスは、 Name、Age、Address、 SerialNum、Salary、Dept、および Bonus という合計 7 つの属性を持つことになります。

関連概念:

- 271 ページの『ユーザー定義構造化型』
- 274 ページの『構造化型階層』

関連タスク:

- 272 ページの『構造化型の作成』
- 282 ページの『型付き表の作成』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE TYPE (構造化) ステートメント』

関連サンプル:

- 『dtstruct.out -- Sample C++ program : dtstruct.sqC (C++)』
- 『dtstruct.sqC -- Create, use, drop a hierarchy of structured types and typed tables (C++)』

構造化型の振る舞いの定義

構造化型の振る舞いを定義するために、ユーザー定義のメソッドを作成することができます。特殊タイプのためのメソッドを作成することはできません。メソッドは 1 つのタイプのために固有に作成されるので、タイプとその振る舞いは緊密に結び合わされているという点を除けば、メソッドを作成することは、関数を作成することと似ています。

CREATE METHOD ステートメントを発行する前に、メソッド仕様をタイプと関連付けておく必要があります。次のステートメントは、 calc_bonus というメソッドのメソッド仕様を Employee_t タイプに追加します。

```
ALTER TYPE Employee_t
   ADD METHOD calc_bonus (rate DOUBLE)
   RETURNS DECIMAL(7,2)
   LANGUAGE SQL
   CONTAINS SQL
   NO EXTERNAL ACTION
   DETERMINISTIC;
```

メソッド仕様をタイプと関連付けた後は、メソッドをメソッド仕様に従って外部メソッドまたは SQL 形式のメソッドとして作成することによって、タイプの振る舞いを定義することができます。たとえば、次のステートメントは、タイプ Employee_t と同じスキーマに常駐する calc_bonus という SQL メソッドを登録します。

```
CREATE METHOD calc_bonus (rate DOUBLE)
   RETURNS DECIMAL(7,2)
   FOR Employee_t
   RETURN SELF..salary * rate;
```

calc_bonus という名前のメソッドは、パラメーターの数または種類が異なっている限り、あるいは異なるタイプ階層の中のタイプについて定義されている限り、いくつでも作成することができます。つまり、Architect_t については、パラメーターの種類とパラメーターの数が同じである calc_bonus という別のメソッドを作成することはできないということです。

関連概念:

- 271 ページの『ユーザー定義構造化型』
- 278 ページの『メソッドの動的ディスパッチング』

関連タスク:

- 272 ページの『構造化型の作成』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『ALTER TYPE (構造化) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE TABLE ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE METHOD ステートメント』

メソッドの動的ディスパッチング

構造化型の振る舞いはそのメソッドによって表現されます。それらのメソッドは構造化型のインスタンスに対してのみ呼び出すことができます。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。

あるサブタイプについて使用するスーパータイプに対してメソッドを定義しない場合、そのメソッドをオーバーライドできます。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、構造化型インスタンスに従って最も特定されたメソッドを実行します (たとえば、構造化型タイプ階層にあるもの)。

オーバーライド・メソッドを定義するには、CREATE TYPE (または ALTER TYPE) ステートメントを使用し、 METHOD 文節の前に OVERRIDING 文節を指定します。 OVERRIDING が指定されない場合、元のメソッド (スーパータイプに属する) が使用されます。オーバーライド・メソッドを定義するには、以下の条件を満たしていなければなりません。

- 作成する (または変更する) タイプは、オーバーライドするメソッドを持つ構造化型のサブタイプでなければならない。
- 宣言するシグニチャー (メソッドの名前とパラメーター・リスト) が、スーパータイプに属するメソッドのシグニチャーと同一である。
- オーバーライド・メソッドは、元のメソッド 1 つだけを暗黙的にオーバーライドしなければならない。
- オーバーライドするルーチンは、ユーザー定義構造化型のインスタンス・メソッドである。

- 元のメソッドは、PARAMETER STYLE JAVA では宣言されない。

以下の例は、メソッドをオーバーライドするためのシナリオ例を示しています。

データ型:

```
CREATE TYPE a AS (z VARCHAR(20))
    METHOD foo(i INTEGER) RETURNS VARCHAR(80)
        LANGUAGE SQL;

CREATE TYPE b UNDER a AS (y VARCHAR(20))
    OVERRIDING METHOD foo(i INTEGER) RETURNS VARCHAR(80);

CREATE TYPE c UNDER a AS (x VARCHAR(20))
    OVERRIDING METHOD foo(i INTEGER) RETURNS VARCHAR(80);

CREATE TYPE d UNDER b AS (w VARCHAR(20))
    OVERRIDING METHOD foo(i INTEGER) RETURNS VARCHAR(80);
```

この状況では、a がスーパータイプです。タイプ b および c は a のサブタイプです。最終的に、d が b のサブタイプになります。

メソッド:

```
CREATE METHOD foo(i INTEGER) FOR a
  RETURN "In method foo_a. Input: " | char(i) | self..z | ".";

CREATE METHOD foo(i INTEGER) FOR b
  RETURN "In method foo_b. Input: " | char(i) | self..z |
         " y = " | self..y | ".";

CREATE METHOD foo(i INTEGER) FOR c
  RETURN "In method foo_c. Input: " | char(i) | self..z |
         " y = " | self..y | " x = " | self..x | ".";

CREATE METHOD foo(i INTEGER) FOR d
  RETURN "In method foo_d. Input: " | char(i) | self..z |
         " y = " | self..y | " w = " | self..w | ".";
```

ここで、元のメソッドは fooA です。 fooB、fooC、および fooD は fooA を明示的にオーバーライドします。 fooD は fooB および fooA を暗黙的にオーバーライドします。同様に、fooB は fooA を暗黙的にオーバーライドし、fooC は fooA を暗黙的にオーバーライドします。 (明示的なオーバーライドは暗黙的なオーバーライドを包含することに注意してください。)

関連概念:

- 271 ページの『ユーザー定義構造化型』
- 274 ページの『構造化型階層』

関連タスク:

- 272 ページの『構造化型の作成』
- 277 ページの『構造化型の振る舞いの定義』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『ALTER TYPE (構造化) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE TYPE (構造化) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE METHOD ステートメント』

構造化型のためのシステム生成ルーチン

構造化型の Comparison 関数と Cast 関数

DB2® は、参照タイプと表示タイプとの間で値を双方向にキャストする関数を自動的に作成します。 CREATE TYPE ステートメントには、オプションの CAST WITH 文節があります。この文節を指定すると、これらの 2 つの cast 関数の名前を選択することができます。デフォルトでは、2 つの cast 関数の名前は、構造化型の名前とその参照表示タイプの名前と同じです。たとえば、CREATE TYPE Person_t ステートメントは、次のフォーマットを持つ関数を自動的に作成します。

```
CREATE FUNCTION VARCHAR(REF(Person_t))
   RETURNS VARCHAR
```

DB2 は、次のような逆の操作を行う関数も作成します。

```
CREATE FUNCTION Person_t(VARCHAR(13))
   RETURNS REF(Person_t)
```

型付き表に新しい値を挿入する必要がある時、あるいは参照値を別の値と比較する時には、いつでもこれらの cast 関数を使用できます。

DB2 は、=、<>、<、<=、>、および >= という比較演算子を使用して参照タイプを比較することのできる関数も作成します。

関連概念:

- 271 ページの『ユーザー定義構造化型』
- 292 ページの『参照タイプ』

関連タスク:

- 272 ページの『構造化型の作成』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE TYPE (構造化) ステートメント』

構造化型の Constructor 関数

構造化型を作成する際に、 DB2® はタイプが作成されたときと同じ名前の関数を作成します。この関数は、パラメーターを持っておらず、タイプの属性すべてが NULL 設定になったタイプのインスタンスを戻します。たとえば、構造化型 Person_t について作成される関数には、次のフォーマットがあります。

```
CREATE FUNCTION Person_t ( ) RETURNS Person_t
```

サブタイプ Manager_t の場合、次のフォーマットを持つ constructor が作成されます。

```
CREATE FUNCTION Manager_t ( ) RETURNS Manager_t
```

列の中に挿入するタイプのインスタンスを構成するには、 constructor 関数を mutator メソッドと一緒に使用します。タイプを列ではなく表に保管する場合は、タイプのインスタンスを挿入するのに、 constructor 関数を mutator メソッドと一緒に使用する必要はありません。

関連概念:

- 271 ページの『ユーザー定義構造化型』

関連タスク:

- 272 ページの『構造化型の作成』

構造化型の Mutator メソッド

mutator メソッドは、オブジェクトのそれぞれの属性について存在します。メソッドが呼び出される対象となる構造化型のインスタンスのことを、メソッドの対象 インスタンスと言います。対象インスタンスに対して呼び出される mutator メソッドが属性の新しい値を受け取ると、属性が新しい値に更新された新しいインスタンスが戻されます。したがって、タイプ Person_t について、 DB2® は name、age、および address のそれぞれの属性のための mutator メソッドを作成します。

たとえば、属性 age のために DB2 が作成する mutator メソッドには、次のフォーマットがあります。

```
ALTER TYPE Person_t
   ADD METHOD AGE(int)
   RETURNS Person_t;
```

関連概念:

- 271 ページの『ユーザー定義構造化型』

関連タスク:

- 272 ページの『構造化型の作成』

構造化型の Observer メソッド

observer メソッドは、オブジェクトのそれぞれの属性について存在します。ある属性の observer メソッドが、所定のタイプまたは所定のサブタイプのオブジェクトを受け取ると、そのオブジェクトの属性値が戻されます。

タイプ Person_t の属性 age のために DB2® が作成する observer メソッドには、次のフォーマットがあります。

```
ALTER TYPE Person_t
   ADD METHOD AGE()
   RETURNS INTEGER;
```

構造化型に対してメソッドを呼び出すには、メソッド呼び出し演算子 '..' を使用します。

以下の例は、Person_t タイプのための observer メソッドの使用を示しています。

```
CREATE FUNCTION MailingAddress (p Person_t)
  RETURNS VARCHAR(40)
  RETURN p..name() || ' ' || p..address()
```

この関数では、Person_t インスタンスの name 列と address 列は observer メソッドによって検索され、 1 つのストリングに連結されて、メール・アドレスを形成します。

関連概念:

- 271 ページの『ユーザー定義構造化型』

関連タスク:

- 272 ページの『構造化型の作成』

型付き表

型付き表

型付き表とは、ユーザー定義構造化型で定義した表です。型付き表を使用すると、表階層と呼ばれる表の間で定義されたリレーションシップを持つ階層構造を設定することができます。表階層は 1 つのルート表、スーパー表、および副表で構成されます。

型付き表は、構造化型のインスタンスを行として保管します。その行ではタイプの各属性が別々の列に保管されます。

関連概念:

- 271 ページの『ユーザー定義構造化型』
- 292 ページの『参照タイプ』
- 286 ページの『型付き表での代理性』
- 296 ページの『型付きビュー』

関連タスク:

- 287 ページの『型付き行へのオブジェクトの保管』
- 285 ページの『型付き表のドロップ』
- 289 ページの『システム生成オブジェクト ID の定義』
- 291 ページの『オブジェクト ID 列に対する制約の定義』
- 282 ページの『型付き表の作成』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE TABLE ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『DROP ステートメント』

型付き表の作成

型付き表は、CREATE TYPE ステートメントを使用して特性が定義されているオブジェクトのインスタンスを実際に保管するのに使用されます。変形した CREATE TABLE ステートメントを使用して、型付き表を作成することができます。構造化型の階層を基にした型付き表の階層を作成することもできます。サブタイプのインスタンスを型付き表に保管するには、対応する表階層を作成する必要があります。

以下の図は、型付き表の階層を示しています。図の後に続く例は、この階層の作成を示しています。

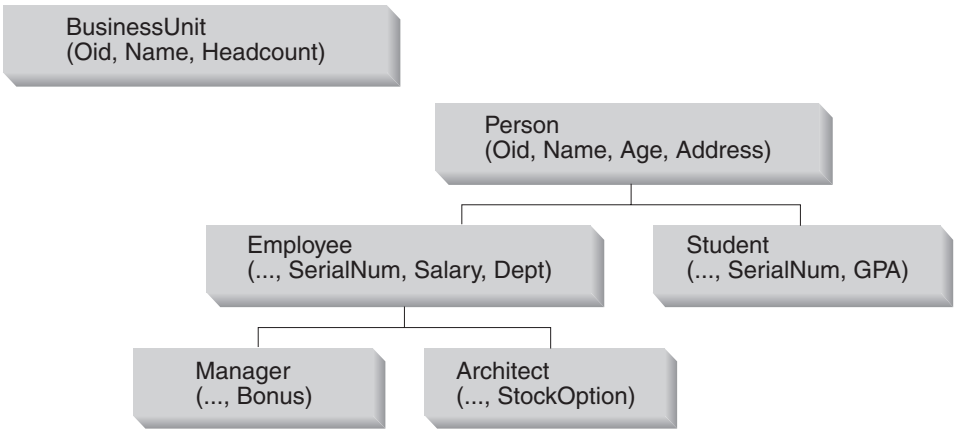


図 6. 型付き表の階層

BusinessUnit 型付き表を作成する SQL は、次のとおりです。

```
CREATE TABLE BusinessUnit OF BusinessUnit_t
   (REF IS Oid USER GENERATED);
```

Person 表階層に表を作成する SQL は、次のとおりです。

```
CREATE TABLE Person OF Person_t
   (REF IS Oid USER GENERATED);

CREATE TABLE Employee OF Employee_t UNDER Person
   INHERIT SELECT PRIVILEGES
   (SerialNum WITH OPTIONS NOT NULL,
   Dept WITH OPTIONS SCOPE BusinessUnit );

CREATE TABLE Student OF Student_t UNDER Person
   INHERIT SELECT PRIVILEGES;

CREATE TABLE Manager OF Manager_t UNDER Employee
   INHERIT SELECT PRIVILEGES;

CREATE TABLE Architect OF Architect_t UNDER Employee
   INHERIT SELECT PRIVILEGES;
```

表のタイプの定義

前述の例で最初に作成される型付き表は、BusinessUnit です。この表は、BusinessUnit_t のタイプとして定義されているので、このタイプのインスタンスを保留することになります。この表は、構造化型 BusinessUnit_t のそれぞれの属性に対応する列と、オブジェクト *ID 列* という 1 つの追加の列を持つようになるということです。

オブジェクト ID の命名

型付き表には他のオブジェクトが参照できるオブジェクトが入っているので、すべての型付き表の最初の列はオブジェクト *ID* 列になっています。この例では、オブジェクト ID 列のタイプは REF(BusinessUnit_t) です。 REF IS ... USER GENERATED 文節を使用して、オブジェクト ID 列に名前を付けることができます。この例では、Oid という名前が付けられています。 REF IS 文節の USER GENERATED 部分は、新たに挿入されるすべての行のオブジェクト ID 列の初期値を与える必要があることを示しています。オブジェクト指向の設計では、データと

オブジェクト ID を完全に分離することが一般的です。そのため、オブジェクト ID の挿入後は、オブジェクト ID の値を更新することはできません。 DB2 に OID 値を生成させる場合は、 SEQUENCE または GENERATE_UNIQUE() 関数を使用できます。

表階層での位置の指定

Person 型付き表のタイプは Person_t です。従業員と生徒のサブタイプのインスタンスを保管するには、 Person 表の副表である Employee と Student を作成する必要があります。 Employee_t の 2 つの追加のサブタイプにも表が必要です。これらの副表の名前は、Manager と Architect になります。サブタイプがスーパータイプの属性を継承するのと同様に、副表もオブジェクト ID 列を含めたスーパー表の列を継承します。

注: 副表はスーパー表と同じスキーマに常駐していなければなりません。

したがって、Employee 副表の中の行の列は、 Oid、Name、Age、Address、SerialNum、Salary、および Dept の合計 7 つになります。

スーパー表に対して作用する SELECT、UPDATE、または DELETE ステートメントは、デフォルトではすべての副表に対しても自動的に作用します。たとえば、Employee に対する UPDATE ステートメントは、 Employee、Manager、および Architect 表の中の行に影響を与えますが、 Manager 表に対する UPDATE ステートメントは、 Manager 行だけにしか影響を与えません。

SELECT、INSERT、または DELETE ステートメントのアクションを特定の表に制限する場合は、 ONLY オプションを使用します。

SELECT 特権は継承されたものであることの表示

CREATE TABLE ステートメントの必須 INHERIT SELECT PRIVILEGES 文節は、Employee などの結果副表が、 Person などのスーパー表 (結果表は UNDER 文節を使用してここから作成される) と同じユーザーおよびグループによって、最初はアクセス可能であることを指定しています。スーパー表に対して現在 SELECT 特権を保留しているユーザーまたはグループには、新しく作成された副表に対する SELECT 特権が付与されます。副表の作成者が、SELECT 特権の付与者になります。副表に対する DELETE や UPDATE などの特権を指定するには、正規表に対する特権を指定するのに使用するのと同じ明示的 GRANT または REVOKE ステートメントを発行する必要があります。

特権は、表階層のすべてのレベルで別々に付与したり取り消したりすることができます。副表を作成する場合は、その副表に対する継承された SELECT 特権を取り消すこともできます。継承された SELECT 特権を副表から取り消すと、スーパー表に対する SELECT 特権を持つユーザーは、副表だけに表示される列を見ることができなくなります。継承された SELECT 特権を副表から取り消すと、スーパー表に対する SELECT 特権しか持っていないユーザーは、副表の行のスーパー表の列だけを見ることができます。副表に対する必要な特権を保留しているユーザーが直接操作できるのは、副表だけです。したがって、ユーザーが副表の管理者の賞与を選択できないようにするには、その副表に対する SELECT 特権を取り消して、この情報が必要なユーザーだけに SELECT 特権を付与するようにします。

列オプションの定義

WITH OPTIONS 文節を使用すると、型付き表の中の個々の列に適用されるオプションを定義することができます。 WITH OPTIONS の形式は次のとおりです。

```
column-name WITH OPTIONS column-options
```

column-name は CREATE TABLE または ALTER TABLE ステートメントの中の列の名前を表し、 *column-options* は列に定義されているオプションを表しています。

たとえば、ユーザーが SerialNum 列に NULL を挿入できないようにするには、次のようにして NOT NULL 列オプションを指定します。

```
(SerialNum WITH OPTIONS NOT NULL)
```

参照列の有効範囲の定義

WITH OPTIONS の別の使用法は、列の有効範囲を指定するという方法です。たとえば、Employee 表とその副表の中では、

```
Dept WITH OPTIONS SCOPE BusinessUnit
```

という文節は、この表とその副表の Dept 列の*有効範囲* が、 BusinessUnit であると宣言しています。これは、Employee 表のこの列の中の参照値は、 BusinessUnit 表の中のオブジェクトを参照することになっているということです。

たとえば、Employee 表に対する次の照会では、逆参照演算子が使用されていて、Dept 列から BusinessUnit 表へパスをたどるよう DB2 に伝えています。逆参照演算子は、Name 列の値を戻します。

```
SELECT Name, Salary, Dept->Name
   FROM Employee;
```

関連概念:

- 271 ページの『ユーザー定義構造化型』
- 274 ページの『構造化型階層』
- 282 ページの『型付き表』

関連タスク:

- 272 ページの『構造化型の作成』
- 287 ページの『型付き行へのオブジェクトの保管』
- 285 ページの『型付き表のドロップ』
- 289 ページの『システム生成オブジェクト ID の定義』
- 291 ページの『オブジェクト ID 列に対する制約の定義』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE TABLE ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE TYPE (構造化) ステートメント』

型付き表のドロップ

型付き表をドロップすることは、非型付き表をドロップすることと似ています。重要な違いは、ドロップする表に副表がないことを確認する必要があるということで

す。ドロップしようとする表に副表がある場合、エラーが発生します。次の例は、Architect 表をドロップする方法を示しています。

```
DROP TABLE Architect;
```

副表を表階層からドロップすると、副表に関連した列にはアクセスできなくなります。代理性により、副表をドロップすると、スーパー表から副表のすべての行が削除されるというセマンティック上の効果があります。これにより、スーパー表で定義されているトリガーまたは参照保全制約が有効になります。

表や索引などの他のデータベース・オブジェクトは、パッケージやキャッシュされた動的ステートメントが無効としてマークされている場合でも、影響を受けることはありません。

表階層全体をドロップすることもできます。それには、HIERARCHY 文節を DROP TABLE ステートメントに追加して、階層のルート表の名前を付けます。以下に例を示します。

```
DROP TABLE HIERARCHY Person;
```

表階層をドロップすると、トリガーまたは参照保全制約が有効になりません。

関連概念:

- 274 ページの『構造化型階層』
- 282 ページの『型付き表』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『DROP ステートメント』

型付き表での代理性

型付き表に SELECT、UPDATE、または DELETE ステートメントが適用されると、指定された表とそのすべての副表に対して操作が行われます。たとえば、構造化型 Person_t から型付き表を作成して、その表のすべての行を選択する場合、アプリケーションは Person タイプのインスタンスだけでなく、 Employee サブタイプとその他のサブタイプのインスタンスについての Person 情報も受け取ることができます。

代理性の特性は、サブタイプから作成される副表にも適用されます。たとえば、Employee 副表の SELECT、UPDATE、および DELETE ステートメントは、Employee_t タイプとそのサブタイプの両方に適用されます。同様に、Address_t タイプを使用して定義されている列には、アメリカ式の住所またはブラジル式の住所のインスタンスを入れることができます。しかし、これは、たとえば Person_t 行が Employee_t データで更新される場合に、 UPDATE ステートメントが行のタイプを変更できることを意味するわけではありません。これを実行するには、 Person_t row を削除し、新規タイプとして Employee_t 行を挿入する必要があります。

SELECT、UPDATE、または DELETE ステートメントで代理性を制約するには、ONLY 文節を使用できます。たとえば、UPDATE ONLY(Person) SET は、行は Person 表の副表ではなく、Person 表でのみ更新されます。

これに対して INSERT 操作は、 INSERT ステートメントで指定されている表だけに適用されます。 Employee 表に挿入を行うと、 Person 表階層に Employee_t オブジェクトが作成されます。

構造化型をパラメーターとして関数に渡す場合、または構造化型を関数からの結果として渡す場合にも、サブタイプのインスタンスを代用することができます。関数が Address_t というタイプのパラメーターを持っている場合は、 Address_t のインスタンスではなく、サブタイプ内の 1 つ (US_addr_t など) のインスタンスを渡すことができます。外部表関数は、構造化型の列を戻すことはできません。

列または表が、1 つのタイプで定義されていても、そこにサブタイプのインスタンスが入っている場合があるので、定義に使用されるタイプと実行時に実際に戻されるインスタンスのタイプとを区別することが重要です。列、行、または関数パラメーターの中の構造化型の定義のことを*静的タイプ* といいます。構造化型インスタンスの実際のタイプを*動的タイプ*と言います。動的タイプについての情報を受け取るために、アプリケーションは TYPE_NAME、TYPE_SCHEMA、および TYPE_ID 組み込み関数を使用できます。

関連概念:

- 274 ページの『構造化型階層』
- 282 ページの『型付き表』

関連タスク:

- 276 ページの『構造化型階層の作成』
- 300 ページの『参照を逆参照する照会の発行』

型付き行へのオブジェクトの保管

オブジェクトを表の中に行として保管すると、表の中のそれぞれの列には、オブジェクトの 1 つの属性が入ります。非型付き表の場合と同様に、 NOT NULL として定義されているすべての列 (オブジェクト ID 列を含む) にデータを入れなければなりません。オブジェクト ID 列のタイプは REF (大文字で入力される) であるため、システムによって生成された cast 関数 (構造化型の作成時に作成された) を使用して、ユーザー提供のオブジェクト ID の値をキャストする必要があります。たとえば、人物のインスタンスを、名前の列と年齢の列を含む表に保管することができます。最初に、Person というインスタンスを保管するための CREATE TABLE ステートメントのインスタンスを示します。

```
CREATE TABLE Person OF Person_t
   (REF IS Oid USER GENERATED)
```

Person というインスタンスを表に挿入するには、次の構文を使用します。

```
INSERT INTO Person (Oid, Name, Age)
   VALUES(Person_t('a'), 'Andrew', 29);
```

表 31. Person 型付き表

Oid	Name	Age	Address
a	Andrew	29	

プログラムは、型付き表の列にアクセスすることによって、オブジェクトの属性にアクセスします。

```
UPDATE Person
   SET Age=30
   WHERE Name='Andrew';
```

上記の UPDATE ステートメントの実行後、表は次のようになります。

表 32. 更新後の Person 型付き表

Oid	Name	Age	Address
a	Andrew	30	

Employee_t という Person_t のサブタイプがあるため、 Employee_t のインスタンスを Person 表に保管することはできません。したがって、副表に保管する必要があります。次の CREATE TABLE ステートメントは、 Person 表の下に Employee 副表を作成します。

```
CREATE TABLE Employee OF Employee_t UNDER Person
   INHERIT SELECT PRIVILEGES
   (SerialNum WITH OPTIONS NOT NULL,
   Dept WITH OPTIONS SCOPE BusinessUnit);
```

Employee 表への挿入は、次のようになります。

```
INSERT INTO Employee (Oid, Name, Age, SerialNum, Salary)
   VALUES (Employee_t('s'), 'Susan', 39, 24001, 37000.48)
```

表 33. Employer 型付き副表

Oid	Name	Age	Address	SerialNum	Salary	Dept
s	Susan	39		24001	37000.48	

次の照会を実行すると、Susan の情報が戻されます。

```
SELECT *
   FROM Employee
   WHERE Name='Susan';
```

Person 表を対象として SQL ステートメントを実行するだけで、従業員と人物の両方のインスタンスにアクセスすることができます。この機能のことを*代理性* といいます。タイプ階層の上位のインスタンスを含む表を対象として照会を実行すると、その階層の下位のタイプのインスタンスを自動的に入手できます。つまり、SELECT、UPDATE、および DELETE ステートメントにとって、 Person 表は論理的には次のように写るということです。

表 34. Person インスタンスと Employee インスタンスを含む Person 表

Oid	Name	Age	Address
a	Andrew	30	(NULL)
s	Susan	39	(NULL)

次の照会を実行すると、 Andrew (人物) と Susan (従業員) の両方のオブジェクト ID と Person_t 情報を入手できます。

```
SELECT *
   FROM Person;
```

関連概念:

- 293 ページの『型付き表の中のオブジェクト間のリレーションシップ』
- 286 ページの『型付き表での代理性』
- 282 ページの『型付き表』

関連タスク:

- 273 ページの『構造化型のインスタンスの保管』
- 282 ページの『型付き表の作成』

システム生成オブジェクト ID の定義

ユニーク値を生成するための 2 つの共通の方法があります。どちらもオブジェクト ID に適用することができます。

- シーケンスを使用する
- GENERATE_UNIQUE 関数を使用する

オブジェクト ID として数値を使用する必要がある場合は、シーケンスを使用できます。最初に、REF USING 文節を使用して、オブジェクト参照の基本タイプが数値 (次の例では、INT) であることを指定します。

```
CREATE TYPE BusinessUnit_t AS
  (Name VARCHAR(20),
   Headcount INT)
   REF USING INT
   MODE DB2SQL
```

型付き表定義は、次のとおりです。

```
CREATE TABLE BusinessUnit OF BusinessUnit_t
  (REF IS oid USER GENERATED)
```

オブジェクト ID を生成するためのシーケンスは、次のように定義できます。

```
CREATE SEQUENCE BusinessUnitOid AS REF(BusinessUnit_t)
```

副表のデータを暗黙的に変更すると、すべてのスーパー表が変更されることに注意してください。そのため、オブジェクト ID を生成するためのシーケンスを呼び出すトリガーは、表階層のルートに追加するのが最善です。

```
CREATE TRIGGER Gen_Bunit_oid
  NO CASCADE
  BEFORE INSERT ON BusinessUnit
  REFERENCING NEW AS new
  FOR EACH ROW
  MODE DB2SQL
  SET new.oid = NEXTVAL FOR BusinessUnitOid
```

シーケンスは REF(BusinessUnitOid) として定義されるため、 oid 列に割り当てるためにキャストは必要ありません。

ここで、新しい業務単位を追加できます。

```
INSERT INTO BusinessUnit (Name, Headcount)
  VALUES('Software', 10)
```

シーケンスを使用すると、生成済みのオブジェクト ID を検索し、それを後続のステートメントで使用することもできます。たとえば、Dept 列のタイプが REF(BusinessUnit) であると仮定して、 Software BusinessUnit に従業員を追加することができます。

```
INSERT INTO Employee(Name, Age, SerialNum, Salary, Dept)
  VALUES('Tom', 28, 106, 60000, PREVVAL FOR BusinessUnitOid)
```

シーケンスを使用したオブジェクト ID の生成に代わる方法として、GENERATE_UNIQUE 関数を使用することができます。 GENERATE_UNIQUE は CHAR (13) FOR BIT DATA 値を戻すので、そのタイプの値を CREATE TYPE ステートメントの REF USING 文節に入れることができるようにします。 VARCHAR (16) FOR BIT DATA というデフォルトは、この目的にかなっています。たとえば、次のようにデフォルトの表示タイプを使用して (つまり、REF USING 文節を指定せずに)、BusinessUnit_t タイプが作成されたとします。

```
CREATE TYPE BusinessUnit_t AS
   (Name VARCHAR(20),
   Headcount INT)
   MODE DB2SQL;
```

型付き表定義は、次のとおりです。

```
CREATE TABLE BusinessUnit OF BusinessUnit_t
(REF IS Oid USER GENERATED);
```

USER GENERATED 文節は必ず指定する必要があります。

型付き表に行を挿入する INSERT ステートメントは、次のようになります。

```
INSERT INTO BusinessUnit (Oid, Name, Headcount)
   VALUES(BusinessUnit_t(GENERATE_UNIQUE( )), 'Toy' 15);
```

Toy 部門に所属する従業員を挿入するには、次のようなステートメントを使用します。これは、BusinessUnit 表からオブジェクト ID 列の値を検索するために副選択を発行し、その値を BusinessUnit_t タイプにキャストして、その値を Dept 列に挿入します。

```
INSERT INTO Employee (Oid, Name, Age, SerialNum, Salary, Dept)
   VALUES(Employee_t('d'), 'Dennis', 26, 105, 30000,
     BusinessUnit_t(SELECT Oid FROM BusinessUnit WHERE Name='Toy'));
```

生成されたオブジェクト ID を INSERT ステートメントで明示的に挿入するのではなく、オブジェクト ID の生成と挿入をトリガーの中でカプセル化することができます。階層のルート上にトリガーがあれば、GENERATE_UNIQUE 関数の呼び出しを自動化できます。次のトリガーは、Person、Employee、Architect、および Manager 表への挿入用の ID を生成します。

```
CREATE TRIGGER Gen_Person_oid
  NO CASCADE
  BEFORE INSERT ON Person
  REFERENCING NEW AS new
  FOR EACH ROW
  MODE DB2SQL
  SET new.oid = Person_t (generate_unique());
```

関連概念:

- 292 ページの『参照タイプ』
- 293 ページの『型付き表の中のオブジェクト間のリレーションシップ』

関連タスク:

- 276 ページの『構造化型階層の作成』
- 300 ページの『参照を逆参照する照会の発行』
- 282 ページの『型付き表の作成』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE TRIGGER ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE TYPE (構造化) ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE SEQUENCE ステートメント』

オブジェクト ID 列に対する制約の定義

オブジェクト ID を外部キーの中の親表のキー列として使用する場合は、最初に、オブジェクト ID 列に対する明示的でユニークな制約または主キー制約を追加するよう、型付き表を変更する必要があります。たとえば、図 7 に示されているような、従業員に対する自己参照リレーションシップを作成するとします。この自己参照リレーションシップでは、それぞれの従業員の管理者は必ず従業員表の中の従業員として存在している必要があります。

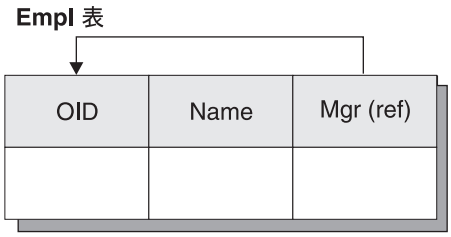


図 7. 自己参照タイプの例

オブジェクト ID 列に対する制約を定義して、オブジェクト上に自己参照リレーションシップを作成するには、次のようにします。

ステップ 1. タイプを作成する。たとえば、次のようにします。

```
CREATE TYPE Empl_t AS
   (Name VARCHAR(10), Mgr REF(Empl_t))
   MODE DB2SQL;
```

ステップ 2. 型付き表を作成する。たとえば、次のようにします。

```
CREATE TABLE Empl OF Empl_t
   (REF IS Oid USER GENERATED);
```

ステップ 3. Oid 列に対する基本制約またはユニーク制約を追加する。たとえば、次のようにします。

```
ALTER TABLE Empl ADD CONSTRAINT pk1 UNIQUE(Oid);
```

ステップ 4. 外部キー制約を追加する。たとえば、次のようにします。

```
ALTER TABLE Empl ADD CONSTRAINT fk1 FOREIGN KEY(Mgr)
   REFERENCES Empl (Oid);
```

関連概念:

- 292 ページの『参照タイプ』

関連タスク:

- 272 ページの『構造化型の作成』
- 287 ページの『型付き行へのオブジェクトの保管』
- 289 ページの『システム生成オブジェクト ID の定義』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE TABLE ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE TYPE (構造化) ステートメント』

参照タイプ

参照タイプ

DB2® は、開発者が作成するすべての構造化型について、自動的にコンパニオン・タイプを作成します。コンパニオン・タイプのことを*参照タイプ* といい、参照先の構造化型のことを*参照先タイプ* といいます。型付き表では参照タイプを特別な方法で使用できます。 SQL ステートメントでは、他のユーザー定義のタイプと同様に、参照タイプを使用することもできます。 SQL ステートメントで参照タイプを使用するには、 REF(type-name) を使用します。 type-name は参照先タイプのことです。

DB2 は、参照タイプを型付き表の中のオブジェクト ID 列のタイプとして使用します。オブジェクト ID によって、型付き表階層の中の行オブジェクトが一意的に識別されます。 DB2 は、参照タイプを使用して、行の参照を型付き表に保管することもします。参照タイプを使用して、表の中のそれぞれの行を参照することができます。

参照は強い型定義を使用します。したがって、式の中でタイプを使用する方法がなければなりません。タイプ階層のルート・タイプを作成する時に、 CREATE TYPE ステートメントの REF USING 文節を使用して、参照のための基本タイプを指定することができます。参照のための基本タイプのことを*表示タイプ* といいます。 REF USING 文節を使用して表示タイプを指定しない場合は、 DB2 は VARCHAR(16) FOR BIT DATA というデフォルト・データ型を使用します。ルート・タイプの表示タイプは、そのすべてのサブタイプに継承されます。 REF USING 文節を使用できるのは、階層のルート・タイプを定義するときだけです。このセクションで一貫して使用されている例では、 `BusinessUnit_t` タイプの表示タイプは INTEGER であり、 `Person_t` の表示タイプは VARCHAR(13) です。

関連概念:

- 296 ページの『参照保全と有効範囲が指定された参照』
- 293 ページの『型付き表の中のオブジェクト間のリレーションシップ』
- 282 ページの『型付き表』

関連タスク:

- 287 ページの『型付き行へのオブジェクトの保管』
- 300 ページの『参照を逆参照する照会の発行』
- 289 ページの『システム生成オブジェクト ID の定義』
- 291 ページの『オブジェクト ID 列に対する制約の定義』
- 282 ページの『型付き表の作成』

型付き表の中のオブジェクト間のリレーションシップ

ある型付き表の中にあるオブジェクトと別の表の中にあるオブジェクトとの間のリレーションシップを定義することができます。同じ型付き表の中にあるオブジェクト間のリレーションシップを定義することもできます。たとえば、部門というインスタンスが入っている型付き表を定義してあるとします。 Employee 表の中で部門番号を保守するのではなく、 Employee 表の Dept 列の中に、 BusinessUnit 表の中の 1 つの部門を指す論理ポインターを入れることができます。これらのポインターのことを*参照* といいます。これは 図 8 で図解されています。

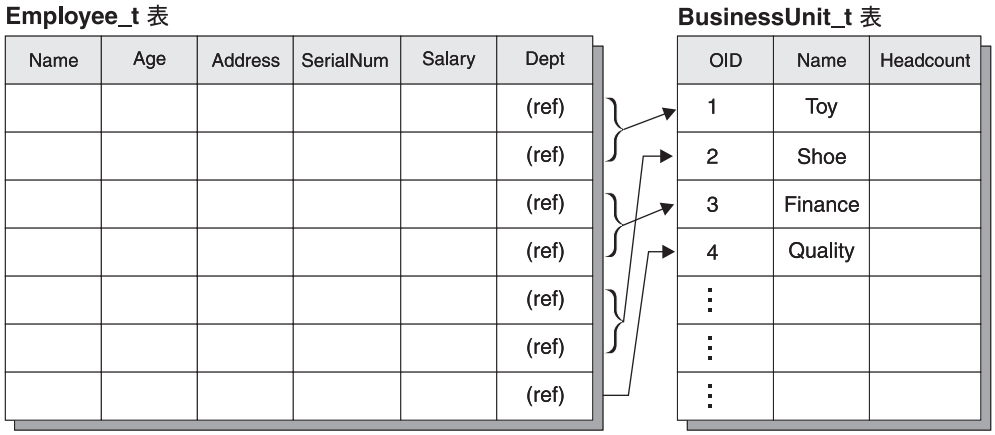


図 8. Employee_t から BusinessUnit_t に対する構造化型参照

通常の表 (型付き表ではない表) は、型付き表を参照する REF 列を持つことができます。しかし、型付き表は、通常の表を指し示す REF 列を持つことはできません。

重要: 参照の機能は、参照制約の機能とは異なります。存在しない部門を参照することができるのです。部門と従業員との間の保全性を維持するのが重要である場合は、これら 2 つの表の間に参照制約を定義してください。参照の実際の機能は、複数の表の間のリレーションシップをナビゲートする照会を作成できるようにすることです。照会が行うことは、リレーションシップを逆参照して、ポインターによって指し示されているオブジェクトをインスタンス化することです。このアクションを実行するのに使用する演算子のことを*逆参照演算子* と言い、-> で表します。

たとえば、Employee 表に対する次の照会では、逆参照演算子が使用されていて、Dept 列から BusinessUnit 表へパスをたどるよう DB2® に伝えています。逆参照演算子は、Name 列の値を戻します。

```
SELECT Name, Salary, Dept->Name
   FROM Employee;
```

関連概念:

- 292 ページの『参照タイプ』
- 296 ページの『参照保全と有効範囲が指定された参照』
- 282 ページの『型付き表』

関連タスク:

- 302 ページの『タイプ述部を使用して、戻されるタイプを制限する』

• 289 ページの『システム生成オブジェクト ID の定義』

参照を使ったセマンティック・リレーションシップの定義

CREATE TABLE の WITH OPTIONS 文節を使用すると、ある表の列と、同じ表または別の表のオブジェクトとの間に存在するリレーションシップを定義することができます。 CREATE TABLE の WITH OPTIONS 文節では、型付き表にある列の列プロパティーを定義します。これらの定義可能な表プロパティーには、ある表の中の列と同じ (または別の) 表の中のオブジェクトとの間のリレーションシップが含まれます。以下の例では、それぞれの従業員の部門は、実際には `BusinessUnit` 表のオブジェクトへの参照になっています。特定の参照列の宛先オブジェクトを定義するには、 WITH OPTIONS 文節で SCOPE キーワードを使用します。

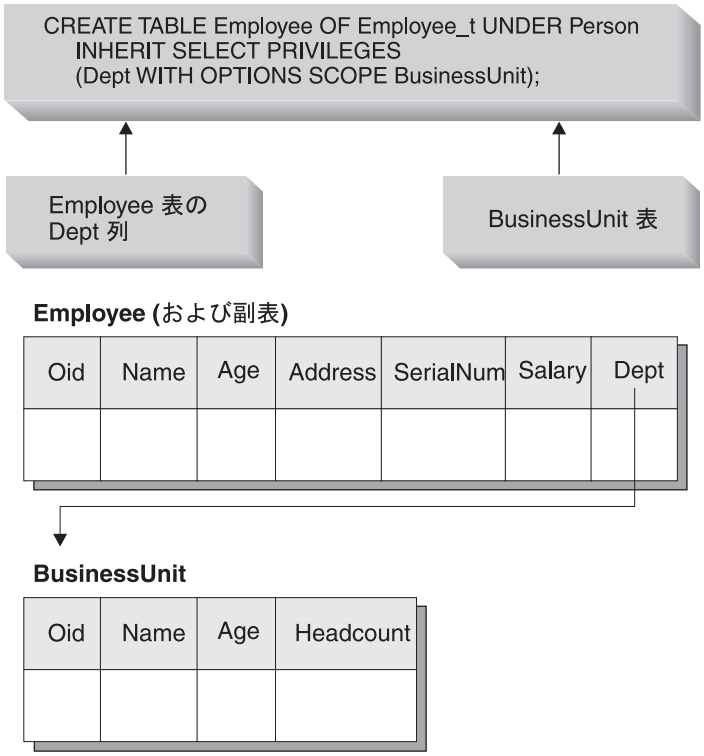


図 9. Dept 属性は BusinessUnit オブジェクトを参照している

自己参照リレーションシップ

同一の型付き表の中のオブジェクトへの有効範囲が指定された参照を定義することもできます。次の例のステートメントは、部品用に 1 つの型付き表と製造業者用に 1 つの型付き表を作成します。参照タイプ定義を示すために、このサンプルには参照タイプを作成するのに使用されるステートメントも含まれています。

```
CREATE TYPE Company_t AS
   (name VARCHAR(30),
   location VARCHAR(30))
   MODE DB2SQL

CREATE TYPE Part_t AS
   (Descript VARCHAR(20),
   Supplied_by REF(Company_t),
   Used_in REF(part_t))
```

```
   MODE DB2SQL

CREATE TABLE Suppliers OF Company_t
   (REF IS suppno USER GENERATED)

CREATE TABLE Parts OF Part_t
   (REF IS Partno USER GENERATED,
   Supplied_by WITH OPTIONS SCOPE Suppliers,
   Used_in WITH OPTIONS SCOPE Parts)
```

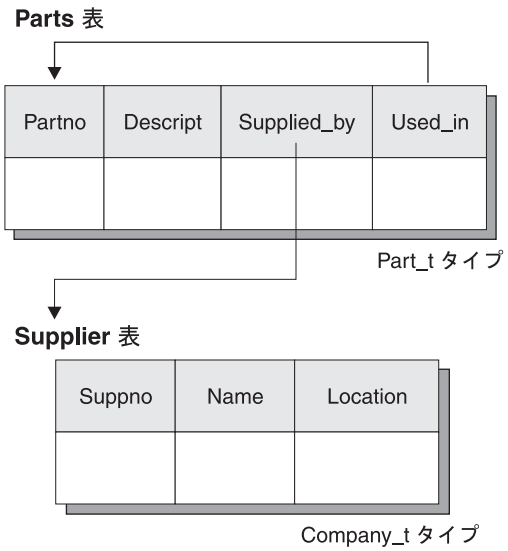


図 10. 自己参照の有効範囲の例

有効範囲が指定された参照を使用しない場合は、外部結合または相関副照会として作成する必要のある照会を、有効範囲が指定された参照を使用して作成することができます。たとえば、次の 2 つの照会では、部品 '1234' を使用している部品製造業者を検索します。

```
SELECT Used_in->Supplied_by->Name
  FROM Parts
  WHERE Partno = Part_t('1234')
```

有効範囲が指定された参照を使用しない場合、照会は次のようになります。

```
SELECT S.Name
  FROM (Parts AS P RIGHT OUTER JOIN Parts C ON P.Used_in = C.Partno)
    RIGHT OUTER JOIN Suppliers S ON C.Supplied_by = S.Suppno
  WHERE P.Partno = Part_t('1234')
```

関連概念:

- 292 ページの『参照タイプ』
- 296 ページの『参照保全と有効範囲が指定された参照』
- 293 ページの『型付き表の中のオブジェクト間のリレーションシップ』
- 282 ページの『型付き表』

関連タスク:

- 289 ページの『システム生成オブジェクト ID の定義』

参照保全と有効範囲が指定された参照

有効範囲が指定された参照は表の中のオブジェクト間のリレーションシップを定義しますが、これは参照保全リレーションシップとは異なります。有効範囲は、ターゲット表についての情報を提供するにすぎません。この情報は、そのターゲット表のオブジェクトを逆参照する時に使用されます。有効範囲が指定された参照では、他の表に値が存在していなければならないということはありません。これらのリレーションシップにおけるオブジェクトの存在を保証するには、表の間に参照制約を追加する必要があります。

関連概念:

- 292 ページの『参照タイプ』
- 282 ページの『型付き表』

関連タスク:

- 294 ページの『参照を使ったセマンティック・リレーションシップの定義』

型付きビュー

型付きビュー

型付きビューの場合、構造化型の属性の名前およびデータ型は、この型付きビューの列の名前およびデータ型になります。型付きビューの行を、構造化型のインスタンスの表現と見なすことができます。

型付き表と同様に、型付きビューもビュー階層の一部になることができます。副ビューはそのスーパービューから列を継承します。副ビューという語は、ビュー階層内で型付きビューの下にあるすべての型付きビューに適用されます。ビュー V の正しい副ビューは、型付きビュー階層内で V の下にあるビューです。

関連概念:

- 271 ページの『ユーザー定義構造化型』
- 282 ページの『型付き表』

関連タスク:

- 297 ページの『型付きビューの作成』
- 299 ページの『型付きビューの変更』
- 299 ページの『型付きビューのドロップ』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『ALTER VIEW ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE VIEW ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『DROP ステートメント』

型付きビューの作成

CREATE VIEW ステートメントを使用して、型付きビューを作成することができます。たとえば、型付き BusinessUnit 表のビューを作成するには、適切な属性を持つ構造化型を定義してから、その構造化型を使用して型付きビューを作成することができます。

```
CREATE TYPE VBusinessUnit_t AS (Name VARCHAR(20))
   MODE DB2SQL;

CREATE VIEW VBusinessUnit OF VBusinessUnit_t MODE DB2SQL
   (REF IS VObjectID USER GENERATED)
   AS SELECT VBusinessUnit_t(VARCHAR(Oid)), Name FROM BusinessUnit;
```

CREATE VIEW ステートメントの OF 文節は、指示された構造化型の属性をビューの列の基礎とするよう DB2 に伝えます。この例では、DB2 は VBusinessUnit_t 構造化型をビューの列の基礎とします。

ビューの VObjectID 列のタイプは、REF(VBusinessUnit_t) です。タイプ REF(BusinessUnit_t) から REF(VBusinessUnit_t) にキャストすることはできないので、最初に表 BusinessUnit の Oid 列の値をデータ型 VARCHAR にキャストしてから、データ型 VARCHAR からデータ型 REF(VBusinessUnit_t) にキャストする必要があります。

MODE DB2SQL 文節は、型付きビューのモードを指定します。現在サポートされているモードはこのモードだけです。

REF IS... 文節は、型付き CREATE TABLE ステートメントの REF IS... 文節と同じです。これは、ビューの最初の列であるビューのオブジェクト ID 列の名前 (この例では VObjectID) を指定します。ルート・ビューを作成する場合、そのビューにオブジェクト ID 列を指定する必要があります。副ビューを作成する場合、それはオブジェクト ID 列を継承します。

USER GENERATED 文節は、ユーザーが行の挿入時にオブジェクト ID 列の値を提供する必要があることを指定します。初期値を挿入した後は、オブジェクト ID 列を更新することはできません。

キーワード AS の後ろにあるビューの本体は、ビューの内容を決定する SELECT ステートメントです。この SELECT ステートメントが戻す列タイプは、オブジェクト ID 列を含む型付きビューの列タイプと互換性を保っている必要があります。

型付きビュー階層の作成を示すために、次の例では、いくつかの機密データが省略され、 Person 表階層からいくつかのタイプ区別が除去されているビュー階層を定義します。

```
CREATE TYPE VPerson_t AS (Name VARCHAR(20))
   MODE DB2SQL;

CREATE TYPE VEmployee_t UNDER VPerson_t
   AS (Salary INT, Dept REF(VBusinessUnit_t))
   MODE DB2SQL;

CREATE VIEW VPerson OF VPerson_t MODE DB2SQL
   (REF IS VObjectID USER GENERATED)
   AS SELECT VPerson_t (VARCHAR(Oid)), Name FROM ONLY(Person);
```

```
CREATE VIEW VEmployee OF VEmployee_t MODE DB2SQL
   UNDER VPerson INHERIT SELECT PRIVILEGES
   (Dept WITH OPTIONS SCOPE VBusinessUnit)
   AS SELECT VEmployee_t(VARCHAR(Oid)), Name, Salary,
      VBusinessUnit_t(VARCHAR(Dept))
   FROM Employee;
```

2 つの CREATE TYPE ステートメントは、この例のオブジェクト・ビュー階層を作成するのに必要な構造化型を作成します。

上記の最初の型付き CREATE VIEW ステートメントは、階層のルート・ビュー VPerson を作成し、 VBusinessUnit ビュー定義と非常によく似たものとなっています。 Person 表の中 (副表の中ではない) の Person 表階層の中の行だけが、VPerson ビューに組み込まれるようにするために、 ONLY(Person) が使用されている点が異なっています。これによって、VPerson の中の Oid 値は、 VEmployee の中の Oid 値と比較して一意的になります。 2 番目の CREATE VIEW ステートメントは、ビュー VPerson の下に副ビュー VEmployee を作成します。 CREATE TABLE...UNDER ステートメントの UNDER 文節の場合と同様に、 UNDER 文節はビュー階層を設定します。スーパービューとして、同じスキーマの中に副ビューを作成する必要があります。型付き表と同様に、副ビューはスーパービューから列を継承します。 VEmployee ビューの中の行は、列 VObjectID と Name を VPerson から継承していて、 VEmployee_t と関連した追加の列 Salary と Dept を持っています。

CREATE VIEW ステートメントを発行する時の INHERIT SELECT PRIVILEGES 文節の効果は、型付き CREATE TABLE ステートメントを発行する時と同じです。型付きビュー定義の中の WITH OPTIONS 文節の効果も、型付き表定義の中の WITH OPTIONS 文節の効果と同じです。 WITH OPTIONS 文節を指定すると、SCOPE などの列オプションを指定することができます。 READ ONLY 文節は、スーパービュー列を強制的に読み取り専用としてマークするので、これ以降の副ビュー定義は、読み取り専用となっている同じ列の式を指定できます。

ビューに VEmployee ビューの Dept 列のような参照列がある場合は、参照列を SQL 逆参照操作で使用するには、参照列に有効範囲を関連付ける必要があります。ビューの参照列に有効範囲を指定しないで、基本表またはビュー列に有効範囲を指定してある場合は、基礎列の有効範囲がビューの参照列に渡されます。 WITH OPTIONS 文節を使用して、ビューの参照列に明示的に有効範囲を割り当てることができます。前述の例では、VEmployee ビューの Dept 列は、 VBusinessUnit ビューを有効範囲として受け取ります。基本表またはビュー列に有効範囲が指定されていない場合で、ビュー定義で明示的に有効範囲が割り当てられていない場合、または ALTER VIEW ステートメントを使用して有効範囲が割り当てられている場合は、参照列には有効範囲が指定されません。

関連概念:

- 271 ページの『ユーザー定義構造化型』
- 282 ページの『型付き表』
- 296 ページの『型付きビュー』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE VIEW ステートメント』

型付きビューの変更

ALTER VIEW ステートメントは、有効範囲を追加するよう参照タイプ列を変更することによって、既存のビューを変更します。ビューに対してこれ以外の変更を行うには、ビューをドロップしてから再作成する必要があります。

ビューを変更する際には、有効範囲がまだ定義されていない既存の参照タイプ列に有効範囲を追加する必要があります。さらに、その列はスーパービューから継承されたものであってはなりません。

ALTER VIEW ステートメントの列名のデータ型は、 REF (型付き表名または型付きビュー名のタイプ) でなければなりません。

関連概念:

- 271 ページの『ユーザー定義構造化型』
- 282 ページの『型付き表』
- 296 ページの『型付きビュー』

関連タスク:

- 297 ページの『型付きビューの作成』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『ALTER VIEW ステートメント』

型付きビューのドロップ

EMP_VIEW のドロップ方法の例は次のとおりです。

```
DROP VIEW EMP_VIEW;
```

ドロップされたビューに従属するすべてのビューは、作動不能になります。

表や索引などの他のデータベース・オブジェクトは、パッケージやキャッシュされた動的ステートメントが無効としてマークされている場合でも、影響を受けることはありません。

表階層の場合と同様に、次の例のように階層のルート・ビューを指定して、 1 つのステートメントの中のビュー階層全体をドロップすることができます。

```
DROP VIEW HIERARCHY VPerson;
```

関連概念:

- 271 ページの『ユーザー定義構造化型』
- 282 ページの『型付き表』
- 296 ページの『型付きビュー』

関連タスク:

- 297 ページの『型付きビューの作成』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『DROP ステートメント』

型付き表と型付きビューの照会

参照を逆参照する照会の発行

有効範囲が指定された参照がある場合は、*逆参照操作* を使用して、逆参照操作を使用しなければ外部結合または相関副照会が必要となる照会を発行できます。BusinessUnit 表に有効範囲が指定されている、 Employee 表および Employee 表の副表の Dept 属性について考慮してみましょう。次の例は、データベース内のすべての従業員の名前、給与、および部門名、または NULL 値を使用できるところでは NULL 値を戻します。この照会は、Employee 表および Employee の副表の中のすべての行の値を戻すということです。相関副照会または外部結合を使用して、同様の照会を作成することができます。しかし、*逆参照操作* (->) を使用して、 Employee 表および副表の中の参照列から BusinessUnit 表へパスを渡って、 BusinessUnit 表の Name 列から結果を戻す方が簡単です。

単純な逆参照操作の形式は、次のとおりです。

```
scoped-reference-expression->column-in-target-typed-table
```

次の照会では、BusinessUnit 表から Name 列を獲得するために、逆参照操作を使用しています。

```
SELECT Name, Salary, Dept->Name
   FROM Employee
```

この照会の結果は、次のようになります。

```
NAME                 SALARY      NAME
-------------------- ----------- --------------------
Dennis               30000       Toy
Eva                  45000       Shoe
Franky               39000       Shoe
Iris                 55000       Toy
Christina            85000       Toy
Ken                  105000      Shoe
Leo                  92000       Shoe
Brian                112000      Toy
Susan                37000.48    ---
```

自己参照している参照を逆参照することもできます。部品表について考慮してみましょう。次の照会は、部品の製造業者の使用箇所を指定して、翼に直接使用されている部品をリストします。

```
SELECT P.Descript, P.Supplied_by->Location
   FROM Parts P
   WHERE P.Used_in->Descript='Wing';
```

DEREF 組み込み関数

DEREF 組み込み関数を使用して、構造化されたオブジェクト全体を 1 つの値として獲得するために、参照を逆参照することもできます。 DEREF の単純な形式は、次のとおりです。

```
DEREF (有効範囲が指定された参照式)
```

通常 DEREF は、TYPE_NAME などの他の組み込み関数のコンテキストの中で使用されるか、アプリケーションに結び付けるために構造化されたオブジェクト全体を獲得するために使用されます。

タイプに関連したその他の組み込み関数

DEREF 関数は、TYPE_NAME、TYPE_ID、または TYPE_SCHEMA 組み込み関数の一部として、呼び出されることがよくあります。これらの関数の目的は、式の動的タイプの名前、内部 ID、およびスキーマ名を戻すことです。たとえば、次の例は、 Responsible という属性を持つ Project 型付き表を作成します。

```
CREATE TYPE Project_t
   AS (Projid INT, Responsible REF(Employee_t))
   MODE DB2SQL;

CREATE TABLE Project
   OF Project_t (REF IS Oid USER GENERATED,
   Responsible WITH OPTIONS SCOPE Employee);
```

Responsible 属性は、 Employee 表への参照として定義されているので、管理者、設計者、および従業員のインスタンスを参照できます。アプリケーションがすべての行の動的タイプの名前を知る必要がある場合は、次のような照会を使用します。

```
SELECT Projid, Responsible->Name,
   TYPE_NAME(DEREF(Responsible))
   FROM PROJECT;
```

前述のインスタンスは、Employee 表の Name の値を戻すために逆参照操作を使用し、 Employee_t というインスタンスの動的タイプを戻すために DEREF 関数を呼び出しています。

許可要件: DEREF 関数を使用するには、表階層の参照先部分にあるすべての表および副表に対する SELECT 権限がなければなりません。たとえば上記の照会では、Employee、Manager、および Architect 型付き表に対する SELECT 特権が必要です。

関連概念:

- 271 ページの『ユーザー定義構造化型』
- 292 ページの『参照タイプ』
- 293 ページの『型付き表の中のオブジェクト間のリレーションシップ』
- 282 ページの『型付き表』
- 296 ページの『型付きビュー』

関連タスク:

- 287 ページの『型付き行へのオブジェクトの保管』
- 302 ページの『ONLY を使用して特定のタイプのオブジェクトを戻す』
- 302 ページの『タイプ述部を使用して、戻されるタイプを制限する』
- 303 ページの『OUTER を使用してすべての可能性のあるタイプを戻す』
- 289 ページの『システム生成オブジェクト ID の定義』

関連資料:

- 「*SQL リファレンス 第 1 巻*」の『DEREF スカラー関数』

ONLY を使用して特定のタイプのオブジェクトを戻す

特定のタイプのオブジェクトだけを戻し、サブタイプのオブジェクトは戻さない照会を行うには、 ONLY キーワードを使用します。たとえば、次の照会は、設計者でも管理者でもない従業員の名前だけを戻します。

```
SELECT Name
FROM ONLY(Employee);
```

前述の照会は、次の結果を戻します。

```
NAME
--------------------
Dennis
Eva
Franky
Susan
```

データのセキュリティーを確保するために、 ONLY を使用する時には、Employee のすべての副表に対する SELECT 特権が必要です。

UPDATE または DELETE ステートメントの操作を、指定された表だけで行えるようにするために、 ONLY 文節を使用することもできます。つまり、ONLY 文節は、指定された表の副表では操作が行われないようにするのです。

関連概念:

- 255 ページの『ユーザー定義特殊タイプ』
- 282 ページの『型付き表』

関連タスク:

- 287 ページの『型付き行へのオブジェクトの保管』
- 300 ページの『参照を逆参照する照会の発行』

タイプ述部を使用して、戻されるタイプを制限する

SQL ステートメントによって戻されるまたは影響を受ける行をより汎用的に制限する方法として、タイプ述部を使用する方法があります。タイプ述部を使用すれば、式の動的タイプを指定されたタイプ (1 つまたは複数) と比較することができます。タイプ述部の単純な構文は、次のとおりです。

```
<expression> IS OF (<type_name>[, ...])
```

ここで、*expression* は構造化型のインスタンスを戻す SQL 式を表し、 *type_name* はインスタンスが比較される構造化型 (1 つまたは複数) を表します。

たとえば、次の照会は、35 歳以上で管理者または設計者である人物を戻します。

```
SELECT Name
   FROM Employee E
   WHERE E.Age > 35 AND
   DEREF(E.Oid) IS OF (Manager_t, Architect_t);
```

前述の照会は、次の結果を戻します。

```
NAME
--------------------
Ken
```

関連概念:

- 271 ページの『ユーザー定義構造化型』
- 292 ページの『参照タイプ』
- 282 ページの『型付き表』
- 296 ページの『型付きビュー』

関連タスク:

- 287 ページの『型付き行へのオブジェクトの保管』
- 300 ページの『参照を逆参照する照会の発行』

OUTER を使用してすべての可能性のあるタイプを戻す

DB2 が構造化型の行の値を戻す時には、アプリケーションは、その特定のインスタンスに入っているまたは入っている可能性のある属性を、必ずしも知っているわけではありません。たとえば、人物を戻す時には、その人物は人物の属性だけを持っているかもしれず、従業員、管理者、または人物のサブタイプという属性を持っているかもしれません。アプリケーションが、1 つの SQL 照会で可能性のあるすべての属性を獲得する必要がある場合は、表参照でキーワード OUTER を使用します。

OUTER (*table-name*) および OUTER (*view-name*) は、表またはビューの列と、もしあればそれに続く表またはビューの副表によって導入される追加の列で構成される仮想表を戻します。追加の列は、副表階層を順番に降下しながら表の右側に追加されていきます。共通の親を持つ副表は、それぞれのタイプが作成された順序で処理されます。行には、*table-name* のすべての行と、 *table-name* の副表のすべての追加の行が入ります。 NULL 値は、行の副表の中にない列について戻されます。

たとえば、平均以上の給与を取得する傾向のある人物についての情報を見たい時に、 OUTER を使用できます。次の照会は、給与 `Salary` が高額であるか、平均成績点 `GPA` が高い `Person` 表階層の情報を戻します。

```
SELECT *
   FROM OUTER(Person) P
   WHERE P.Salary > 200000
   OR P.GPA > 3.95 ;
```

OUTER(Person) を使用すれば、 OUTER(Person) を使用しない場合は `Person` 照会で参照できないサブタイプ属性を参照することができます。

OUTER を使用するには、参照先の表のすべての副表またはビューに対する SELECT 特権が必要です。なぜなら、参照先の表のすべての副表またはビューの情報はすべて、 OUTER を使用することによって公開されるからです。

アプリケーションが、高給の人物の属性だけでなく、その個人の最も特異なタイプが何であるかも知る必要があるとします。これは、次のようにして、オブジェクトのオブジェクト ID を TYPE_NAME 組み込み関数に渡し、 OUTER 照会と結合することによって行うことができます。

```
SELECT TYPE_NAME(DEREF(P.Oid)), P.*
   FROM OUTER(Person) P
   WHERE P.Salary > 200000 OR
   P.GPA > 3.95 ;
```

Person 型付き表の Address 列には構造化型が入っているので、その列からデータを戻すためには追加の関数を定義し、追加の SQL を発行する必要があります。これらの追加のステップを実行すると、前述の照会は次の出力を戻します。 *Additional Attributes* には、GPA と Salary が入ります。

```
1                OID          NAME                Additional Attributes
---------------- ------------ ------------------- ...
PERSON_T         a            Andrew              ...
PERSON_T         b            Bob                 ...
PERSON_T         c            Cathy               ...
EMPLOYEE_T       d            Dennis              ...
EMPLOYEE_T       e            Eva                 ...
EMPLOYEE_T       f            Franky              ...
MANAGER_T        i            Iris                ...
ARCHITECT_T      l            Leo                 ...
EMPLOYEE_T       s            Susan               ...
```

関連概念:

- 271 ページの『ユーザー定義構造化型』
- 282 ページの『型付き表』
- 296 ページの『型付きビュー』

関連タスク:

- 304 ページの『表列への構造化型オブジェクトの保管』
- 300 ページの『参照を逆参照する照会の発行』

列タイプとしての構造化型

表列への構造化型オブジェクトの保管

オブジェクトを列に保管することは、 DB2 の組み込みデータ型では完全にモデル化することができない、ビジネス・オブジェクトについてのファクトをモデル化する必要がある場合に役立ちます。つまり、ビジネス・オブジェクト (従業員、部門など) を型付き表に保管する場合でも、これらのオブジェクトは、構造化型を使用すれば合理的にモデル化できる属性も持っている場合があるということです。

たとえば、アプリケーションが、住所の特定の部分にアクセスする必要があるとします。住所を構造化されていない文字ストリングとして保管するのではなく、 図 11 に示されているように構造化されたオブジェクトとして保管することができます。

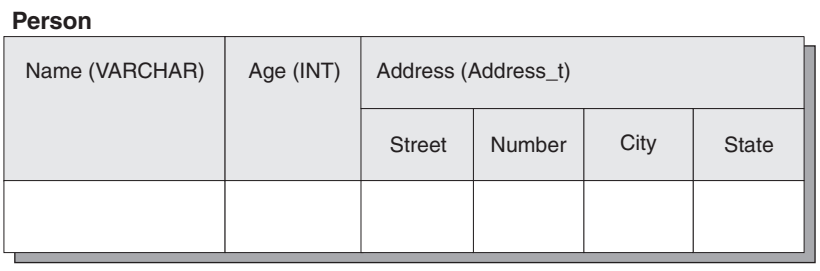


図 11. 構造化型としての Address 属性

さらに、住所のタイプ階層を定義して、さまざまな国で使用されている住所のさまざまな形式をモデル化することができます。たとえば、郵便番号が含まれているアメリカの住所タイプと、区域属性が必要なブラジルの住所タイプの両方を含めたいとします。

図 12 は、さまざまなタイプの住所の階層を示しています。ルート・タイプは Address_t です。これには、その地域での住所の形式のいくつかの局面を反映する追加の属性を持つ 3 つのサブタイプがあります。

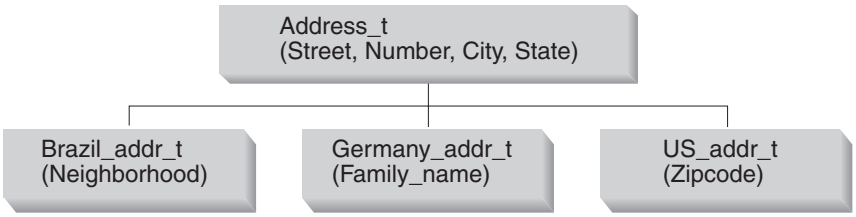


図 12. Address_t タイプの構造化型階層

```
CREATE TYPE Address_t AS
   (street VARCHAR(30),
   number CHAR(15),
   city VARCHAR(30),
   state VARCHAR(10))
   MODE DB2SQL;

CREATE TYPE Germany_addr_t UNDER Address_t AS
   (family_name VARCHAR(30))
   MODE DB2SQL;

CREATE TYPE Brazil_addr_t UNDER Address_t AS
   (neighborhood VARCHAR(30))
   MODE DB2SQL;

CREATE TYPE US_addr_t UNDER Address_t AS
   (zip CHAR(10))
   MODE DB2SQL;
```

オブジェクトが列の値として保管されている場合は、表の行の中に保管されているオブジェクトのように、属性を外部的に表現することはできません。この場合、属性を操作するためのメソッドを使用する必要があります。 DB2 は、属性を戻すための *observer* メソッドと、属性を変更するための *mutator* メソッドの両方を生成します。次の例では、住所を変更するために 1 つの observer メソッドと、 Number 属性用と Street 属性用の 2 つの mutator メソッドを使用しています。

```
UPDATE Employee
   SET Address=Address..Number('4869')..Street('Appletree')
   WHERE Name='Franky'
   AND Address..State='CA';
```

上記のインスタンスでは、UPDATE ステートメントの SET 文節は、タイプ Address_t のインスタンスの属性を更新するために、 Number および Street mutator メソッドを呼び出します。

さらに複雑な (特にネストされた) 構造化型のインスタンスの更新を実行するために、 DB2 は更新される属性を SET 文節の左側にドリルダウンできるようにします。

```
UPDATE Employee
   SET Address..Number = '4869',
       Address..Street = 'Appletree'
   WHERE Name='Franky' AND Address..State='CA'
```

WHERE 文節は、次の 2 つの述部によって UPDATE ステートメントの操作を制約します。それは、Name 列の同一性比較と、Address 列の State observer メソッドを呼び出す同一性比較です。

関連概念:

- 271 ページの『ユーザー定義構造化型』

関連タスク:

- 272 ページの『構造化型の作成』
- 273 ページの『構造化型のインスタンスの保管』
- 306 ページの『構造化型の属性を列に挿入する』
- 309 ページの『列の中の構造化型値の検索と変更』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『UPDATE ステートメント』

構造化型の属性を列に挿入する

ユーザー定義構造化型の属性を、組み込み静的 SQL を使用して属性と同じタイプの列に属性として挿入するには、インスタンスを示すホスト変数を括弧で囲み、右小括弧に 2 つのドット演算子と属性名を追加します。たとえば、以下のような状態を想定します。

```
- PERSON_T は、タイプ VARCHAR(30) の属性 NAME を持つ構造化型です。
- T1 は VARCHAR(30) の列 C1 を持つ表です。
- personhv は、プログラミング言語で、タイプ PERSON_T に宣言されている
ホスト変数です。
```

NAME 属性を列 C1 に挿入する正しい構文は次のようになります。

EXEC SQL INSERT INTO T1 (C1) VALUES ((:personhv)..NAME)

関連概念:

- 281 ページの『構造化型の Observer メソッド』

関連タスク:

- 272 ページの『構造化型の作成』
- 304 ページの『表列への構造化型オブジェクトの保管』
- 310 ページの『構造化型の属性の検索』

構造化型列を持つ表の定義および変更

構造化型列を持つ表を作成することは、大抵、DB2 SQL データ型だけを持つ表を作成することと違いがありません。定義する列ごとに、対応するデータ型を割り振ります。構造化型列の場合、対応するデータ型として構造化型名が提供されます。たとえば、次の ALTER TABLE ステートメントは、 Address_t タイプの列を Customer_List 非型付き表に追加します。

```
ALTER TABLE Customer_List
   ADD COLUMN Address Address_t;
```

これで、Address_t のインスタンスまたは Address_t のサブタイプをこの表に保管できるようになりました。

データ・レコード内での構造化型のレイアウトに関心がある場合は、 CREATE TYPE ステートメントで INLINE LENGTH 文節を使用することができます。この文節は、列の中の構造化型のインスタンスの最大サイズを示します。構造化型のインスタンスのサイズが定義された最大サイズより小さい場合、データは行の中のその他の値とともにインラインで保管されます。構造化型のインスタンスのサイズが定義された最大サイズを超える場合、構造化型データは表の外部に保管されます (LOB に類似)。

加えた変更を構造化型に適用するには、 ALTER TABLE ALTER COLUMN SET INLINE LENGTH ステートメントを発行することにより、影響を受ける構造化型列のサイズを変更することができます。列の長さを変更した後で、REORG ユーティリティーを呼び出してください。

関連概念:

- 271 ページの『ユーザー定義構造化型』

関連タスク:

- 272 ページの『構造化型の作成』
- 304 ページの『表列への構造化型オブジェクトの保管』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『ALTER TABLE ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE TABLE ステートメント』

構造化型属性を持つタイプの定義

タイプは、構造化型属性を持つものとして作成できます。あるいは、そのような属性を追加またはドロップするようタイプを (使用される前に) 変更できます。たとえば、次の CREATE TYPE ステートメントには、タイプ Address_t の属性が含まれています。

```
CREATE TYPE Person_t AS
   (Name VARCHAR(20),
   Age INT,
   Address Address_t)
   REF USING VARCHAR(13)
   MODE DB2SQL;
```

Person_t は、表のタイプ、正規表の中の列のタイプ、または別の構造化型の属性として使用できます。

関連タスク:

- 272 ページの『構造化型の作成』
- 304 ページの『表列への構造化型オブジェクトの保管』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE TYPE (構造化) ステートメント』

構造化型値が入っている行の挿入

構造化型を作成すると、DB2 はそのタイプのための constructor メソッドを自動的に生成して、そのタイプの属性のための mutator メソッドと observer メソッドを生成します。これらのメソッドを使用して、構造化型のインスタンスを作成し、これらのインスタンスを表の列に挿入することができます。

新しい行を Employee 型付き表に追加して、その行に住所を入れるとします。組み込みデータ型の場合と同様に、 VALUES 文節を指定した INSERT を使用してこの行を追加できます。しかし、住所に挿入する値を指定する時は、次のように、システムが提供する constructor 関数を呼び出してその値を作成する必要があります。

```
INSERT INTO Employee (Oid, Name, Age, SerialNum, Salary, Dept, Address)
   VALUES(Employee_t('m'), 'Marie', 35, 005, 55000, BusinessUnit_t(2),
   US_addr_t ( ) 1
      ..street('Bakely Avenue') 2
      ..number('555') 3
      ..city('San Jose') 4
      ..state('CA') 5
      ..zip('95141')); 6
```

前述のステートメントは、次のタスクを実行することによって、 US_addr_t タイプのインスタンスを作成します。

1. US_addr_t() の呼び出しは、すべての属性が NULL 値に設定されたタイプのインスタンスを作成するために、 US_addr_t タイプのための constructor 関数を呼び出す。
2. ..street('Bakely Avenue') の呼び出しは、値を 'Bakely Avenue' に設定するために、 street 属性のための mutator メソッドを呼び出す。
3. ..number('555') の呼び出しは、値を '555' に設定するために、 number 属性のための mutator メソッドを呼び出す。
4. ..city('San Jose') の呼び出しは、値を 'san Jose' に設定するために、 city 属性のための mutator メソッドを呼び出す。
5. ..state('CA') の呼び出しは、値を 'CA' に設定するために、 state 属性のための mutator メソッドを呼び出す。
6. ..zip('95141') の呼び出しは、値を '95141' に設定するために、 zip 属性のための mutator メソッドを呼び出す。

Employee 表の中の列 Address のタイプは Address_t として定義されていますが、代理性の特性を活用して、その列に US_addr_t のインスタンスを挿入することができます。 US_addr_t が Address_t のサブタイプだからです。

構造化型のインスタンスを作成する度に、構造化型のそれぞれの属性のための mutator メソッドを明示的に呼び出さなくてもよいようにするために、すべての属性を初期化する独自の SQL の constructor 関数を定義することを考慮してください。次の例は、US_addr_t タイプのための SQL の constructor 関数の宣言です。

```
CREATE FUNCTION US_addr_t
      (street VARCHAR(30),
      number CHAR(15),
      city VARCHAR(30),
      state VARCHAR(20),
      zip CHAR(10))
   RETURNS US_addr_t
```

```
    LANGUAGE SQL
    RETURN US_addr_t()..street(street)..number(number)
        ..city(city)..state(state)..zip(zipcode);
```

次のインスタンスは、前述のインスタンスの SQL の constructor 関数を呼び出して、 US_addr_t タイプのインスタンスを作成する方法を示しています。

```
INSERT INTO Employee(Oid, Name, Age, SerialNum, Salary, Dept, Address)
   VALUES(Employee_t('m'), 'Marie', 35, 005, 55000, BusinessUnit_t(2),
      US_addr_t('Bakely Avenue', '555', 'San Jose', 'CA', '95141'));
```

関連概念:

- 286 ページの『型付き表での代理性』
- 282 ページの『型付き表』

関連タスク:

- 272 ページの『構造化型の作成』
- 304 ページの『表列への構造化型オブジェクトの保管』
- 306 ページの『構造化型の属性を列に挿入する』
- 306 ページの『構造化型列を持つ表の定義および変更』
- 307 ページの『構造化型属性を持つタイプの定義』

列の構造化型値の変更

列の中の構造化型値の検索と変更

アプリケーションおよびユーザー定義の関数が、構造化型列の中のデータにアクセスする方法は 2 つあります。それは、オブジェクトの個々の属性にアクセスすることと、オブジェクトを単一の値と見なすことです。オブジェクトを単一の値として扱う場合は、最初に transform 関数を定義する必要があります。正しい transform 関数を定義した後は、他の任意の値も選択できますが、構造化オブジェクトを選択することができます。

```
SELECT Name, Dept, Address
   FROM Employee
   WHERE Salary > 20000;
```

以下のトピックでは、 DB2 の組み込み observer メソッドおよび mutator メソッドを呼び出して、オブジェクトの個々の属性に明示的にアクセスする方法を説明します。これらの組み込みメソッドを使用すれば、transform 関数を定義する必要はありません。

手順:

1. 構造化型の属性の検索
2. サブタイプの属性へのアクセス
3. 構造化型属性の変更
4. 構造化型についての情報を戻す

関連概念:

- 312 ページの『Transform 関数と Transform グループ』

関連タスク:

- 310 ページの『構造化型の属性の検索』
- 310 ページの『サブタイプの属性へのアクセス』
- 311 ページの『構造化型属性の変更』
- 311 ページの『構造化型についての情報を戻す』
- 304 ページの『表列への構造化型オブジェクトの保管』
- 306 ページの『構造化型の属性を列に挿入する』
- 308 ページの『構造化型値が入っている行の挿入』

構造化型の属性の検索

オブジェクトの個々の属性に明示的にアクセスするには、それらの属性に対して DB2 組み込み *observer* メソッドを呼び出します。 observer メソッドを使用すれば、オブジェクトを単一の値として扱うのではなく、属性を個別に検索することができます。

次の例は、 Address 列の定義済み静的タイプである Address_t に対して observer メソッドを呼び出すことによって、 Address 列の中のデータにアクセスします。

```
SELECT Name, Dept, Address..street, Address..number, Address..city,
   Address..state
   FROM Employee
   WHERE Salary > 20000;
```

注: DB2 では、*<type-name >..<method-name >()* か、*<type-name >..<method-name >* のどちらかを使用して、パラメーターのないメソッドを呼び出すことができます。ここで、*type-name* は構造化型の名前を表し、*attribute-name* はパラメーターがないメソッドの名前を表します。

observer メソッドを使用して、次のようにしてそれぞれの属性を選択してホスト変数に入れることもできます。

```
SELECT Name, Dept, Address..street, Address..number, Address..city,
   Address..state
   INTO :name, :dept, :street, :number, :city, :state
   FROM Employee
   WHERE Empno = '000250';
```

関連タスク:

- 306 ページの『構造化型の属性を列に挿入する』
- 310 ページの『サブタイプの属性へのアクセス』
- 311 ページの『構造化型属性の変更』
- 311 ページの『構造化型についての情報を戻す』

サブタイプの属性へのアクセス

Employee 表では、住所に指定できるタイプが 4 つあり、それらは Address_t、US_addr_t、Brazil_addr_t、および Germany_addr_t です。 Address_t のいずれかのサブタイプの値の属性にアクセスするには、特定のオブジェクトのタイプが US_addr_t、Germany_addr_t、または Brazil_addr_t のいずれかであることを DB2 に示すために、 TREAT 式を使用しなければなりません。 TREAT 式は、次の照会の中で示されているように、構造化型式をサブタイプのうちの 1 つにキャストします。

```
SELECT Name, Dept, Address..street, Address..number, Address..city,
   Address..state,
   CASE
      WHEN Address IS OF (US_addr_t)
      THEN TREAT(Address AS US_addr_t)..zip
      WHEN Address IS OF (Germany_addr_t)
      THEN TREAT (Address AS Germany_addr_t)..family_name
      WHEN Address IS OF (Brazil_addr_t)
      THEN TREAT (Address AS Brazil_addr_t)..neighborhood
   ELSE NULL END
   FROM Employee
   WHERE Salary > 20000;
```

関連タスク:

- 306 ページの『構造化型の属性を列に挿入する』
- 310 ページの『構造化型の属性の検索』
- 311 ページの『構造化型属性の変更』
- 311 ページの『構造化型についての情報を戻す』

構造化型属性の変更

構造化列値の属性を変更するには、変更する属性のための mutator メソッドを呼び出します。たとえば、住所の street 属性を変更するには、 street のための変更後の値を指定した mutator メソッドを呼び出します。戻り値は、street の新しい値が指定された住所になります。次の例は、Employee 表の中の住所タイプを更新するために、 street という属性のための mutator メソッドを呼び出します。

```
UPDATE Employee
   SET Address = Address..street('Bailey')
   WHERE Address..street = 'Bakely';
```

次の例は、前述の例と同じ更新を行いますが、更新する構造化列が指定されているわけではなく、 SET 文節が street という属性のための mutator メソッドに直接アクセスしています。

```
UPDATE Employee
   SET Address..street = 'Bailey'
   WHERE Address..street = 'Bakely';
```

関連タスク:

- 306 ページの『構造化型の属性を列に挿入する』
- 310 ページの『構造化型の属性の検索』
- 310 ページの『サブタイプの属性へのアクセス』
- 311 ページの『構造化型についての情報を戻す』

構造化型についての情報を戻す

組み込み関数を使用して、特定のタイプの名前、スキーマ、または内部タイプ ID を戻すことができます。次のステートメントは、 'Iris' という従業員に関連した住所値の正確なタイプを戻します。

```
SELECT TYPE_NAME(Address)
   FROM Employee
   WHERE Name='Iris';
```

関連タスク:

Transform 関数と Transform グループ

Transform 関数と Transform グループ

Transform 関数 は、構造化型の値をホスト言語プログラム、および外部関数やメソッドと交換するために使用されます。通常 transform 関数は、FROM SQL transform 関数と TO SQL transform 関数の対で使用します。 FROM SQL 関数は、構造化型オブジェクトを外部プログラムと交換可能なタイプに変換し、 TO SQL 関数はオブジェクトを構成します。

transform 関数を作成する時には、 transform 関数の論理対を 1 つのグループに入れます。 *transform グループ* 名によって、指定した構造化型のためのこれらの関数の対を一意的に識別できます。

transform 関数を使用する前に、 CREATE TRANSFORM ステートメントを使用して、 transform 関数をグループ名およびタイプと関連付ける必要があります。CREATE TRANSFORM ステートメントは、既存の関数 (1 つまたは複数) を識別して、その関数を transform 関数として使用できるようにします。次の例は、2 組みの関数が、タイプ `Address_t` のための transform 関数として使用されるように指定しています。このステートメントは、 `func_group` と `client_group` という 2 つの transform グループを作成します。これらのグループは、それぞれが FROM SQL transform と TO SQL transform で構成されています。

```
CREATE TRANSFORM FOR Address_t
   func_group ( FROM SQL WITH FUNCTION addresstofunc,
      TO SQL WITH FUNCTION functoaddress )
   client_group ( FROM SQL WITH FUNCTION stream_to_client,
      TO SQL WITH FUNCTION stream_from_client ) ;
```

CREATE TRANSFORM ステートメントにグループを追加することによって、`Address_t` タイプに追加の関数を関連付けることができます。 transform 定義を変更するには、追加の関数を指定した CREATE TRANSFORM ステートメントを再発行する必要があります。

transform 関数とタイプとの関連付けを解除するには、 SQL ステートメント DROP TRANSFORM を使用します。 DROP TRANSFORM ステートメントの実行後は、transform 関数がドロップされるわけではありませんが、このタイプのための transform 関数として使用されることはありません。次の例は、`Address_t` タイプのための transform 関数の特定のグループ `func_group` の関連付けを解除してから、`Address_t` タイプのためのすべての transform 関数の関連付けを解除しています。

```
DROP TRANSFORMS func_group FOR Address_t;

DROP TRANSFORMS ALL FOR Address_t;
```

transform 定義を変更するには、追加の関数を指定した CREATE TRANSFORM ステートメントを再発行する必要があります。たとえば、クライアント関数を異なる

ホスト言語プログラム用 (C 用、 Java™ 用など) にカスタマイズすることがあります。アプリケーションのパフォーマンスを最適化するために、 transform の対象をオブジェクト属性のサブセットだけに限定することもあります。あるいは、オブジェクト用のクライアントを表すものとして VARCHAR を使用する transform、BLOB を使用する transform を用意することもあります。

関連概念:

- 271 ページの『ユーザー定義構造化型』
- 327 ページの『transform 関数の要件』
- 314 ページの『Transform グループの指定』
- 316 ページの『Transform 関数を使用したホスト言語プログラムのマッピング』
- 317 ページの『function transform』
- 313 ページの『Transform グループの命名についての推奨事項』

関連タスク:

- 309 ページの『列の中の構造化型値の検索と変更』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『DROP ステートメント』
- 「*SQL リファレンス 第 2 巻*」の『CREATE TRANSFORM ステートメント』

Transform グループの命名についての推奨事項

transform グループ名は、*修飾されていない* ID です。つまり、特定のスキーマと関連付けられていないということです。サブタイプ・パラメーターを扱う transform を作成するのでない限り、すべての構造化型のための transform グループ名を割り当てるべきではありません。同一プログラムまたは同一 SQL ステートメント内で、関連付けがなされていないさまざまなタイプを使用する必要があるかもしれないので、 transform グループは、transform 関数が実行するタスクに従って命名する必要があります。

一般的に transform グループの名前は、タイプ名に依存したものではなく、実行される関数を反映したもの、あるいは、transform 関数の論理を何らかの方法で反映したものとなっているべきです。実行される関数や関数の論理は、タイプ間で非常に異なっているものです。たとえば、TO および FROM SQL 関数 transform が定義されているグループに、 `func_group` または `object_functions` という名前を使用することができます。 TO および FROM SQL クライアント transform が入っているグループに、 `client_group` または `program_group` という名前を使用することができます。

次の例では、`Address_t` および `Polygon` タイプは、非常に異なる transform を使用していますが、同じ関数グループ名を使用しています。

```
CREATE TRANSFORM FOR Address_t
   func_group (TO SQL WITH FUNCTION functoaddress,
   FROM SQL WITH FUNCTION addresstofunc );

CREATE TRANSFORM FOR Polygon
   func_group (TO SQL WITH FUNCTION functopolygon,
   FROM SQL WITH FUNCTION polygontofunc);
```

ふさわしい状況のもとで transform グループを func_group に設定した後は、address または polygon をバインドインするかバインドアウトする度に、 DB2® は正しい transform 関数を呼び出します。

制限: transform グループの名前は 'SYS' というストリングで始めることはできません。これは DB2 が使用する予約済みのグループを表します。

外部関数または外部メソッドを定義する場合で、transform グループ名を指定しない場合は、 DB2 は DB2_FUNCTION という名前を使用し、このグループ名は指定の構造化型のために指定されたものと想定します。指定の構造化型を参照するクライアント・プログラムをプリコンパイルする時にグループ名を指定しない場合は、DB2 は DB2_PROGRAM というグループ名を使用し、このグループ名はこの構造化型のために定義されたものと想定します。

このデフォルトの振る舞いは、便利なこともありますが、より複雑なデータベース・スキーマでは、 transform グループ名のためのもう少し詳細な規則が必要であると感じるかもしれません。たとえば、このデフォルトの振る舞いは、タイプをバインドアウトするさまざまな言語にさまざまなグループ名を使用する点で役立ちます。

関連概念:

- 312 ページの『Transform 関数と Transform グループ』
- 314 ページの『Transform グループの指定』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE TRANSFORM ステートメント』

Transform グループの指定

Transform グループの指定

ある構造化型に対して多数の transform グループを定義することができるため、プログラムまたは特定の SQL ステートメントでは、その構造化型のために使用する transform のグループを指定する必要があります。 transform グループを指定する必要がある状況には、次の 3 つがあります。

- 外部関数または外部メソッドが定義されている時は、参照先オブジェクトを*分解*して*構成* するグループを指定する必要があります。
- 静的 SQL をプリコンパイルまたはバインドする時は、参照先タイプのためにクライアント・バインドインとクライアント・バインドアウトを行う transform のグループを指定する必要があります。
- 動的 SQL を実行する時、またはコマンド行プロセッサーを使用する時は、参照先タイプのためにクライアント・バインドインとクライアント・バインドアウトを行う transform のグループを指定する必要があります。

関連概念:

- 312 ページの『Transform 関数と Transform グループ』
- 316 ページの『Transform 関数を使用したホスト言語プログラムのマッピング』

関連タスク:

- 315 ページの『外部ルーチン用の Transform グループの指定』
- 315 ページの『動的 SQL 用の Transform グループの指定』
- 316 ページの『静的 SQL 用の Transform グループの指定』

外部ルーチン用の Transform グループの指定

CREATE FUNCTION および CREATE METHOD ステートメントでは、TRANSFORM GROUP 文節を指定することができます。この文節は LANGUAGE 文節の値が SQL ではない場合にのみ有効になります。 SQL 言語関数では transform は必要ありませんが、外部関数では必要です。 TRANSFORM GROUP 文節を使用すれば、構造化型のパラメーターや結果に使用される TO SQL および FROM SQL transform が入っている transform グループを、指定の関数またはメソッドに指定することができます。次の例では、CREATE FUNCTION および CREATE METHOD ステートメントは、 transform グループ `func_group` を、 TO SQL および FROM SQL transform に指定しています。

```
CREATE FUNCTION stream_from_client (VARCHAR (150))
   RETURNS Address_t
   ...
   TRANSFORM GROUP func_group
   EXTERNAL NAME 'addressudf!address_stream_from_client'
   ...

CREATE METHOD distance ( point )
   FOR polygon
   RETURNS integer
   :
   TRANSFORM GROUP func_group ;
```

関連概念:

- 312 ページの『Transform 関数と Transform グループ』
- 314 ページの『Transform グループの指定』

関連タスク:

- 277 ページの『構造化型の振る舞いの定義』
- 315 ページの『動的 SQL 用の Transform グループの指定』
- 316 ページの『静的 SQL 用の Transform グループの指定』

動的 SQL 用の Transform グループの指定

動的 SQL を使用する場合は、 CURRENT DEFAULT TRANSFORM GROUP 特殊レジスターを設定できます。この特殊レジスターは、静的 SQL ステートメントには使用されません。また、外部関数またはメソッドとのパラメーターや結果の交換にも使用されません。 SET CURRENT DEFAULT TRANSFORM GROUP ステートメントは、動的 SQL ステートメントのためのデフォルト transform グループを設定するために使用します。

```
SET CURRENT DEFAULT TRANSFORM GROUP = client_group;
```

関連概念:

- 312 ページの『Transform 関数と Transform グループ』
- 314 ページの『Transform グループの指定』

関連タスク:

- 315 ページの『外部ルーチン用の Transform グループの指定』
- 316 ページの『静的 SQL 用の Transform グループの指定』

静的 SQL 用の Transform グループの指定

静的 SQL については、PRECOMPILE または BIND コマンドで TRANSFORM GROUP オプションを使用して、さまざまなタイプの値をホスト・プログラムと交換するために、静的 SQL ステートメントが使用する静的 transform グループを指定します。静的 transform グループは動的 SQL ステートメントには適用されません。また、外部関数またはメソッドとのパラメーターや結果の交換にも適用されません。 PRECOMPILE または BIND コマンドで静的 transform グループを指定するには、次のようにして TRANSFORM GROUP 文節を使用します。

```
PRECOMPILE ...
TRANSFORM GROUP client_group
... ;
```

関連概念:

- 312 ページの『Transform 関数と Transform グループ』
- 314 ページの『Transform グループの指定』

関連タスク:

- 315 ページの『外部ルーチン用の Transform グループの指定』
- 315 ページの『動的 SQL 用の Transform グループの指定』

関連資料:

- 「コマンド・リファレンス」の『BIND コマンド』
- 「コマンド・リファレンス」の『PRECOMPILE コマンド』

ホスト言語プログラムへのマッピングの作成

Transform 関数を使用したホスト言語プログラムのマッピング

アプリケーションは 1 つのオブジェクト全体を直接選択することはできませんが、オブジェクトの個々の属性を選択してアプリケーションに入れることはできます。アプリケーションは、通常はオブジェクト全体を直接挿入することはありませんが、 constructor 関数の呼び出しの結果を挿入することはできます。

```
INSERT INTO Employee(Address) VALUES (Address_t());
```

サーバー・アプリケーションとクライアント・アプリケーション、または外部関数の間で全オブジェクトを交換するには、通常は *transform 関数* を作成する必要があります。

ある transform 関数は、 DB2® がオブジェクトの内容にアクセスできるよう正しく定義された形式にオブジェクトを変換する方法、またはオブジェクトをバインドアウト する方法を定義します。別の transform 関数は、DB2 がデータベースに保管されるオブジェクトを戻す方法、または、DB2 がオブジェクトをバインドイン する方法を定義します。オブジェクトをバインドアウトする transform のことを FROM SQL transform 関数といい、列オブジェクトをバインドインする transform のことを TO SQL transform 関数といいます。

ルーチン、または外部 UDF および外部メソッドにオブジェクトを渡すための transform の種類は、オブジェクトをクライアント・アプリケーションに渡す transform の種類よりも多くなります。これは、オブジェクトを外部ルーチンに渡す時は、オブジェクトを*分解* して、パラメーターのリストとしてルーチンに渡すからです。クライアント・アプリケーションでは、オブジェクトを BLOB などの単一の組み込みタイプに変換する必要があります。このプロセスのことを、オブジェクトのエンコードといいます。これら 2 種類の transform は、多くの場合一緒に使用されます。

transform 関数を特定の構造化型と関連付けるには、SQL ステートメント CREATE TRANSFORM を使用します。 CREATE TRANSFORM ステートメント内で、transform 関数は対にされて *transform グループ* と呼ばれるものに入れられます。これによって、特定の変換目的に使用される関数を識別しやすくなります。 1 つの transform グループに入れることができるのは、特定の 1 つのタイプにつき、 1 つの FROM SQL transform と 1 つの TO SQL transform だけです。

関連概念:

- 327 ページの『transform 関数の要件』
- 312 ページの『Transform 関数と Transform グループ』
- 317 ページの『function transform』
- 322 ページの『client transform』

関連タスク:

- 319 ページの『SQL を本体として持つルーチンを使用した Transform 関数のインプリメント』
- 321 ページの『構造化型パラメーターを外部ルーチンに渡す』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE TRANSFORM ステートメント』

function transform

DB2® は、TO SQL および FROM SQL *function transform* を使用して、外部ルーチンとの間でオブジェクトのやり取りを行います。 SQL を本体として持つルーチンについては transform を使用する必要はありません。しかし、DB2 は、クライアント・プログラムとの間でオブジェクトをやり取りするプロセスの一部として、これらの関数を使用することがよくあります。

次の例は、MYUDF という外部 UDF を呼び出す SQL ステートメントを発行します。住所を入力パラメーターとして取り、(たとえば、通りの名前の変更を反映するために) 住所を変更して、変更された住所を戻します。

```
SELECT MYUDF(Address)
FROM PERSON;
```

318 ページの図 13 は、DB2 が住所を処理する方法を示しています。

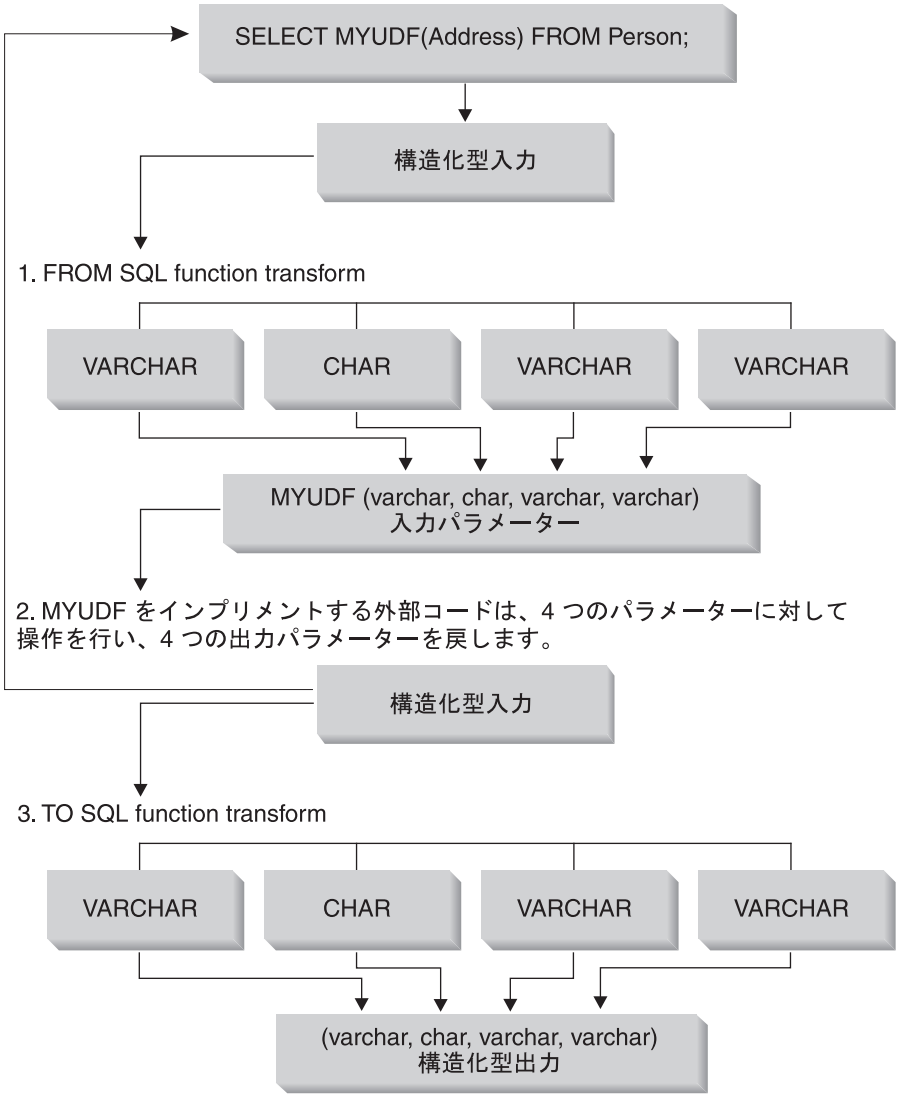


図 13. 構造化型パラメーターの外部ルーチンとの交換

1. FROM SQL transform 関数は、構造化型を基本属性の順序づけられたセットに分解します。これによって、ルーチンは、タイプが基本組み込みデータ型であるパラメーターの単純なリストとして、オブジェクトを受け取ることができるようになります。たとえば、住所オブジェクトを外部ルーチンに渡すとします。`Address_t` の属性は、順に VARCHAR、CHAR、 VARCHAR、VARCHAR となっています。このオブジェクトをルーチンに渡すための FROM SQL transform は、このオブジェクトを入力として受け入れ、 VARCHAR、CHAR、VARCHAR、VARCHAR を戻す必要があります。次にこれらの出力は、 4 つの対応する NULL 標識パラメーターと構造化型のための 1 つの NULL 標識と一緒に、 4 つの別々のパラメーターとして外部ルーチンに渡されます。`Address_t` タイプを戻すすべての関数でパラメーターの順序が同じである限り、FROM SQL 関数内のパラメーターの順序は重要ではありません。
2. 外部ルーチンは分解された住所を入力パラメーターとして受け入れ、これらの値に対して処理を行ってから、属性を出力パラメーターとして戻します。

3. TO SQL transform 関数は、MYUDF から戻される VARCHAR、CHAR、VARCHAR、 VARCHAR パラメーターを、タイプ Address_t のオブジェクトに変換する必要があります。言い換えると、TO SQL transform 関数は、 4 つのパラメーターとすべての対応する NULL 標識パラメーターを、ルーチンからの出力値として取る必要があるということです。 TO SQL 関数は、構造化オブジェクトを構成し、次に、指定された値を使用して属性を変化させます。

注: MYUDF も構造化型タイプを戻す場合、 SELECT 文節の中で UDF が使用されている時は、結果として生じる構造化型は別の transform 関数によって変換される必要があります。別の transform 関数が作成されないようにするには、次の例のように、observer メソッドを指定した SELECT ステートメントを使用します。

```
SELECT Name
   FROM Employee
   WHERE MYUDF(Address)..city LIKE 'Tor%';
```

関連概念:

- 312 ページの『Transform 関数と Transform グループ』
- 316 ページの『Transform 関数を使用したホスト言語プログラムのマッピング』
- 322 ページの『client transform』

関連タスク:

- 319 ページの『SQL を本体として持つルーチンを使用した Transform 関数のインプリメント』
- 321 ページの『構造化型パラメーターを外部ルーチンに渡す』

SQL を本体として持つルーチンを使用した Transform 関数のインプリメント

オブジェクトを外部ルーチンと交換する時にオブジェクトを分解して構成するには、 SQL で作成されたユーザー定義の関数を使用する必要があります。この関数のことを SQL を本体として持つルーチンと言います。 SQL を本体として持つ関数を作成するには、 LANGUAGE SQL 文節を指定して CREATE FUNCTION ステートメントを発行します。

SQL を本体として持つ関数では、変換を行うために constructor、observer、および mutator を使用できます。この SQL を本体として持つ transform は、SQL ステートメントと外部関数との間に介在します。 FROM SQL transform は、オブジェクトを SQL パラメーターとして取り、構造化型の属性を表す値の行を戻します。 SQL を本体として持つ関数を使用した、住所オブジェクトのための有効な FROM SQL transform 関数の例は、次のとおりです。

```
CREATE FUNCTION addresstofunc (A Address_t) 1
   RETURNS ROW  (Street VARCHAR(30), Number CHAR(15),
      City VARCHAR(30), State (VARCHAR(10)) 2

   LANGUAGE SQL 3
   RETURN VALUES (A..Street, A..Number, A..City, A..State) 4
```

次のリストは、前述の CREATE FUNCTION ステートメントの構文を説明しています。

1. この関数のシグニチャーは、タイプ Address_t のオブジェクトという 1 つのパラメーターを受け入れることを示しています。
2. RETURNS ROW 文節は、この関数が、 Street、Number、City、および State という 4 つの列を含む行を戻すことを示しています。
3. LANGUAGE SQL 文節は、これが外部関数ではなく、SQL を本体として持つ関数であることを示しています。
4. RETURN 文節は、関数本体の先頭をマークしています。本体は、Address_t オブジェクトのそれぞれの属性のための observer メソッドを呼び出す 1 つの VALUES 文節で構成されています。 observer メソッドは、オブジェクトを基本タイプのセットに分解します。この関数は、この基本タイプのセットを行として戻します。

DB2 には、開発者がこの関数を transform 関数として使用するつもりであることは分かりません。この関数を使用する transform グループを作成して、ふさわしい状況でその transform グループを指定するまでは、 DB2 はこの関数を transform 関数として使用することはできません。

TO SQL transform は、FROM SQL 関数と反対のことを行います。 TO SQL transform は、ルーチンからのパラメーターのリストを入力として取り、構造化型のインスタンスを戻します。次の FROM SQL 関数は、オブジェクトを構成するために、 Address_t タイプのための constructor 関数を呼び出します。

```
CREATE FUNCTION functoaddress (street VARCHAR(30), number CHAR(15),
                               city VARCHAR(30), state VARCHAR(10)) 1
   RETURNS Address_t 2
   LANGUAGE SQL
   CONTAINS SQL
   RETURN
      Address_t()..street(street)..number(number)
      ..city(city)..state(state) 3
```

次のリストは、前述のステートメントの構文を説明しています。

1. 関数は、基本タイプ属性を取ります。
2. 関数は、Address_t 構造化型を戻します。
3. 関数は、Address_t のための constructor と、それぞれの属性のための mutator を呼び出すことによって、入力タイプからオブジェクトを構成します。

この transform 関数を使用して住所を戻すすべての関数でパラメーターの順序が同じである限り、 FROM SQL 関数内のパラメーターの順序は重要ではありません。

関連概念:

- 317 ページの『function transform』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION (SQL スカラー、表、または行) ステートメント』

構造化型パラメーターを外部ルーチンに渡す

構造化型パラメーターを外部ルーチンに渡す時は、それぞれの属性のためのパラメーターを渡す必要があります。それぞれのパラメーターのための NULL 標識と、構造化型のための NULL 標識を渡す必要があります。次の例は、構造化型 Address_t を受け入れ、基本タイプを戻します。

```
CREATE FUNCTION stream_to_client (Address_t)
   RETURNS VARCHAR(150) ...
```

外部ルーチンは、Address_t タイプのインスタンスのための NULL 標識 (address_ind) と、 Address_t タイプのそれぞれの属性につき 1 つの NULL 標識を受け入れる必要があります。 VARCHAR 出力パラメーターのための NULL 標識もあります。次のコードは、UDF をインプリメントする関数のための C 言語関数のヘッダーを表しています。

```
void SQL_API_FN stream_to_client(
/* decomposed address */
   SQLUDF_VARCHAR *street,
   SQLUDF_CHAR *number,
   SQLUDF_VARCHAR *city,
   SQLUDF_VARCHAR *state,
/* VARCHAR output */
   SQLUDF_VARCHAR *output,
/* null indicators for type attributes */
   SQLUDF_NULLIND *street_ind,
   SQLUDF_NULLIND *number_ind,
   SQLUDF_NULLIND *city_ind,
   SQLUDF_NULLIND *state_ind,
/* null indicator for instance of the type */
   SQLUDF_NULLIND *address_ind,
/* null indicator for the VARCHAR output */
   SQLUDF_NULLIND *out_ind,
   SQLUDF_TRAIL_ARGS)
```

ルーチンが *st1* と *st2* という 2 つの異なる構造化型パラメーターを受け入れ、 *st3* という別の構造化型を戻すとします。

```
CREATE FUNCTION myudf (int, st1, st2)
   RETURNS st3
```

表 35. myudf パラメーターの属性

ST1	ST2	ST3
st1_att1 VARCHAR	st2_att1 VARCHAR	st3_att1 INTEGER
st2_att2 INTEGER	st2_att2 CHAR	st3_att2 CLOB
	st2_att3 INTEGER	

次のコードは、UDF をインプリメントするルーチンのための C 言語のヘッダーを表しています。引き数には、次のように、変数と、分解された構造化型の属性の NULL 標識および構造化型のそれぞれのインスタンスのための NULL 標識が入っています。

```
void SQL_API_FN myudf(
    SQLUDF_INTEGER *INT,
 /* Decomposed st1 input */
    SQLUDF_VARCHAR *st1_att1,
    SQLUDF_INTEGER  *st1_att2,
 /* Decomposed st2 input */
    SQLUDF_VARCHAR *st2_att1,
```

```
   SQLUDF_CHAR   *st2_att2,
   SQLUDF_INTEGER *st2_att3,
/* Decomposed st3 output */
   SQLUDF_VARCHAR *st3_att1out,
   SQLUDF_CLOB   *st3_att2out,
/* Null indicator of integer */
   SQLUDF_NULLIND *INT_ind,
/* Null indicators of st1 attributes and type */
   SQLUDF_NULLIND *st1_att1_ind,
   SQLUDF_NULLIND *st1_att2_ind,
   SQLUDF_NULLIND *st1_ind,
/* Null indicators of st2 attributes and type */
   SQLUDF_NULLIND *st2_att1_ind,
   SQLUDF_NULLIND *st2_att2_ind,
   SQLUDF_NULLIND *st2_att3_ind,
   SQLUDF_NULLIND *st2_ind,
/* Null indicators of st3_out attributes and type */
   SQLUDF_NULLIND *st3_att1_ind,
   SQLUDF_NULLIND *st3_att2_ind,
   SQLUDF_NULLIND *st3_ind,
/* trailing arguments */
   SQLUDF_TRAIL_ARGS
)
```

関連概念:

- 312 ページの『Transform 関数と Transform グループ』
- 316 ページの『Transform 関数を使用したホスト言語プログラムのマッピング』
- 317 ページの『function transform』
- 322 ページの『client transform』

関連タスク:

- 325 ページの『外部 UDF を使用した、クライアントからのバインドインのための client transform のインプリメント』

client transform

client transform は、構造化型をクライアント・アプリケーション・プログラムとの間で交換します。たとえば、次の SQL ステートメントを実行するとします。

```
...
SQL TYPE IS Address_t AS VARCHAR(150) addhv;
...

EXEC SQL SELECT Address
   FROM Person
   INTO :addhv
   WHERE AGE > 25
END EXEC;
```

323 ページの図 14 は、住所をクライアント・プログラムにバインドアウトするプロセスを示しています。

図 14. 構造化型のクライアント・アプリケーションへのバインドアウト

1. オブジェクトを基本タイプ属性に分解するために、最初にオブジェクトを FROM SQL 関数 transform に渡す必要がある。
2. FROM SQL client transform は、値をエンコードして、 VARCHAR または BLOB などの単一の組み込みタイプにする必要がある。これによって、クライアント・プログラムは、値全体を単一のホスト変数として受け取ることができます。

   このエンコードは、(必要な調整に備えて) 属性をストレージの連続区域にコピーすることと同じように簡単に行うことができます。通常、属性のエンコードとデコードは SQL を使用して行うことはできないので、 client transform は外部 UDF として作成されます。
3. クライアント・プログラムが値を処理する。

324 ページの図 15 は、住所をデータベースに戻す逆のプロセスを示しています。

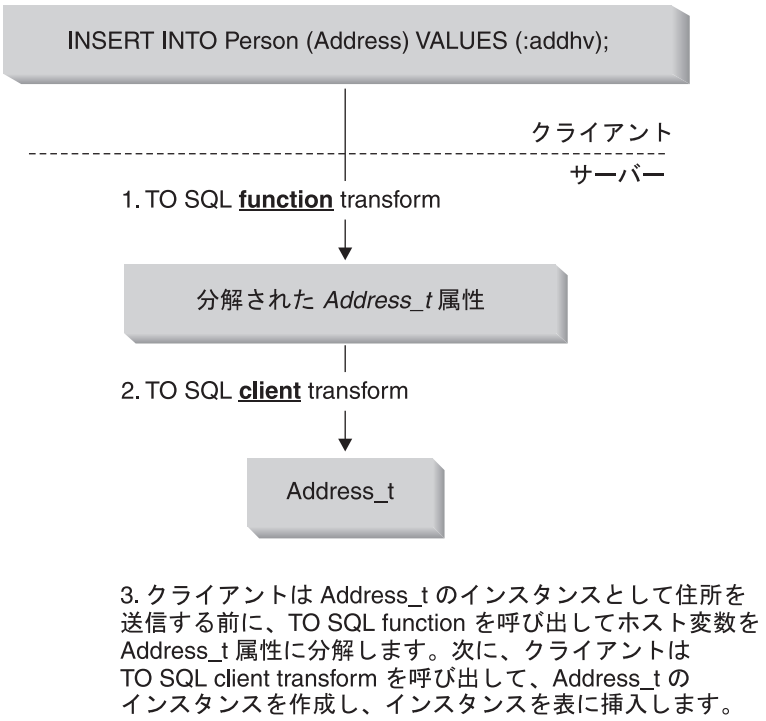


図 15. クライアントからの構造化型のバインドイン

1. クライアント・アプリケーションは、住所を TO SQL client transform が処理できる形式にエンコードする。
2. TO SQL client transform は、単一の組み込みタイプをその基本タイプ属性のセットに分解します。そしてこのセットは TO SQL transform 関数への入力として使用される。
3. TO SQL transform 関数は、住所を構成してそれをデータベースに戻す。

TRANSFORM GROUP 文節を含めます。これは、指定された関数の中の住所タイプを処理するために、どの transform のセットを使用するかを DB2® に伝えるものです。

関連概念:

- 316 ページの『Transform 関数を使用したホスト言語プログラムのマッピング』
- 317 ページの『function transform』

関連タスク:

- 324 ページの『外部 UDF を使用した client transform のインプリメント』
- 325 ページの『外部 UDF を使用した、クライアントからのバインドインのための client transform のインプリメント』

外部 UDF を使用した client transform のインプリメント

client transform をその他の外部 UDF と同じ方法で登録します。たとえば、住所のために適切なエンコードとデコードを行う外部 UDF を作成したと想定しましょう。 FROM SQL client transform には `from_sql_to_client`、 TO SQL client

transform には to_sql_from_client という名前を付けたとします。この両方の事例では、関数の出力は、ふさわしい FROM SQL および TO SQL transform 関数が入力として使用できる形式になります。

```
CREATE FUNCTION from_sql_to_client (Address_t)
   RETURNS VARCHAR (150)
   LANGUAGE C
   TRANSFORM GROUP func_group
   EXTERNAL NAME 'addressudf!address_from_sql_to_client'
   NOT VARIANT
   NO EXTERNAL ACTION
   NOT FENCED
   NO SQL
   PARAMETER STYLE SQL;
```

前述の例の DDL では、from_sql_to_client UDF が、タイプ Address_t のパラメーターを受け入れるかのように示されています。実際には、from_sql_to_client UDF が呼び出されるそれぞれの行のために、 Addresstofunc transform が Address をさまざまな属性に分解します。 from_sql_to_client UDF は、単純な文字ストリングを生成し、住所属性を表示できるように形式設定するので、次の単純な SQL 照会を使用して、 Person 表のそれぞれの行の Name および Address 属性を表示することができます。

```
SELECT Name, from_sql_to_client (Address)
FROM Person;
```

from_sql_to_client の DDL に TRANSFORM GROUP という文節が入っていることに注意してください。この文節は、これらの関数の中の住所タイプを処理するために、どの transform のセットを使用するかを DB2 に伝えています。

関連概念:

- 322 ページの『client transform』

関連タスク:

- 321 ページの『構造化型パラメーターを外部ルーチンに渡す』
- 325 ページの『外部 UDF を使用した、クライアントからのバインドインのための client transform のインプリメント』

外部 UDF を使用した、クライアントからのバインドインのための client transform のインプリメント

次の DDL は、クライアントからの VARCHAR でエンコードされたオブジェクトを受け取る関数を登録し、それをさまざまな基本タイプ属性に分解して、TO SQL transform 関数に渡します。

```
CREATE FUNCTION to_sql_from_client (VARCHAR (150))
   RETURNS Address_t
   LANGUAGE C
   TRANSFORM GROUP func_group
   EXTERNAL NAME 'addressudf!address_to_sql_from_client'
   NOT VARIANT
   NO EXTERNAL ACTION
   NOT FENCED
   NO SQL
   PARAMETER STYLE SQL;
```

to_sql_from_client が住所を直接戻すように見えますが、実際には、to_sql_from_client が VARCHAR (150) を基本タイプ属性のセットに変換します。その後で、DB2 は明示的に TO SQL transform functoaddress を呼び出して、データベースに戻される住所オブジェクトを構成します。

to_sql_from_client の DDL に、TRANSFORM GROUP という文節が入っていることに注意してください。この文節は、これらの関数の中の住所タイプを処理するために、どの transform のセットを使用するかを DB2 に伝えています。

関連概念:

- 322 ページの『client transform』

関連タスク:

- 324 ページの『外部 UDF を使用した client transform のインプリメント』

データ変換についての考慮事項

サーバーとクライアントとの間でデータ、特にバイナリー・データが交換される時は、考慮すべきデータ変換の問題がいくつかあります。たとえば、バイト順序づけ体系が異なるオペレーティング・システム間でデータが転送される時は、正しい数値をリストアするために、数値データに対してバイト反転処理を行う必要があります。異なるオペレーティング・システムでは、メモリー内の数値データを参照するための一定の調整要件も異なります。つまり、これらの要件が満たされていないと、プログラム例外が発生するオペレーティング・システムもあるということです。文字データが BLOB または VARCHAR FOR BIT DATA などのバイナリー・データの中に組み込まれていなければ、文字データ型は、データベースによって自動的に変換されます。

データ変換の問題を避けるには、次の 2 とおりの方法があります。

- 常にオブジェクトを数値データも含めた印刷可能文字タイプに変換する方法。

  この方法は、潜在的に多くの変換が必要になるので、パフォーマンスを低下させ、これらのオブジェクトにアクセスするクライアント上のコードまたは transform 関数自体のコードを複雑にしてしまいます。
- Java™ インプリメンテーションで取られる方法に似た方法で、バイナリー・データ型に変換されたオブジェクトについてオペレーティング・システムに依存しない形式を考案する方法。次のことを必ず実行するようにしてください。
  - これらのコンパクトにされたオブジェクトをパックまたはアンパックするときに、個々のデータ型を正しくエンコードまたはデコードし、データ破壊やプログラム障害を避けるように注意する。
  - エンコード・オブジェクトのヘッダー以降の部分が、クライアントまたはサーバー・オペレーティング・システムに依存せずに正しく解釈されるように、変換されたタイプに十分のヘッダー情報を含める。
  - CREATE FUNCTION の DBINFO オプションを使用して、データベース・サーバー環境に関連した transform 関数のさまざまな特性を渡す。これらの特性は、オペレーティング・システムに依存しない形式で、ヘッダーに入れることができます。

サーバーとクライアントとの間のデータの転送に関連した複雑な問題を transform 関数が正しく処理できるよう、transform 関数は可能な限り開発者が作成してください。アプリケーションを設計する時は、現在の環境の特定の要件を考慮し、完全な汎用性と単純性との間でのトレードオフを評価してください。たとえば、データベース・サーバーとそのすべてのクライアントは、両方とも AIX® 環境で稼働し、同じコード・ページを使用することが分かっている場合は、現在必要な変換は何もないので、前述の考慮事項は無視することができます。しかし、将来環境が変わる場合は、データ変換を正しく処理するよう元の設計を修正するのに、相当の努力を払わなければならなくなります。

関連概念:

- 312 ページの『Transform 関数と Transform グループ』
- 316 ページの『Transform 関数を使用したホスト言語プログラムのマッピング』
- 317 ページの『function transform』

transform 関数の要件

表 36 は、外部ルーチンまたはクライアント・アプリケーションにバインドアウトするかどうかに基づいて、必要な transform 関数が何であるかを決定するのに役立ちます。

表 36. transform 関数の特性

特性	外部ルーチンとの値の交換		クライアント・アプリケーションとの値の交換	
transform の方向	FROM SQL	TO SQL	FROM SQL	TO SQL
変換される対象	ルーチン・パラメーター	ルーチン結果	出力ホスト変数	入力ホスト変数
振る舞い	分解	構成	エンコード	デコード
transform 関数のパラメーター	構造化型	組み込みタイプの行	構造化型	1 つの組み込みタイプ
transform 関数の結果	組み込みタイプの行 (おそらくは属性)	構造化型	1 つの組み込みタイプ	構造化型
別の transform への依存性	なし	なし	FROM SQL UDF transform	TO SQL UDF transform
transform グループが指定される時期	UDF が登録される時		静的: プリコンパイル時 動的: 特殊レジスター	
データ変換についての考慮事項の有無	なし		あり	

注: 一般的な事例ではありませんが、次のどちらかが真である場合は、クライアント・タイプの transform は、実際に SQL で作成できます。

- 構造化型に 1 つの属性しかない場合。
- 組み込みタイプへの属性のエンコードおよびデコードが、 SQL 演算子または関数がいくつか組み合わされることによって行われる。

このような場合は、構造化型の値をクライアント・アプリケーションと交換するために、 transform 関数に依存する必要はありません。

関連概念:

- 312 ページの『Transform 関数と Transform グループ』

関連タスク:

- 328 ページの『サブタイプ・データの DB2 からの検索』

サブタイプ・データの DB2 からの検索

データ・モデルがサブタイプを利用する場合は、列の中の値は、さまざまなサブタイプのうちのいずれかになります。実際の入力タイプに基づいて、正しい transform 関数を動的に選択することができます。

次の SELECT ステートメントを発行するとします。

```
SELECT Address
   FROM Person
   INTO :hvaddr;
```

アプリケーションには、Address_t、 US_addr_t などのインスタンスが戻されるかどうかはまったく分かりません。この例を複雑なものにしないように、 Address_t か US_addr_t だけが戻されると想定しましょう。これらの構造は異なっているので、属性を分解する transform も異なっていなければなりません。正しい transform が呼び出されるようにするために、次のようにします。

ステップ 1. 住所のそれぞれのバリエーションのための FROM SQL transform 関数を作成する。

```
CREATE FUNCTION addresstofunc(A address_t)
   RETURNS ROW
   (Street VARCHAR(30), Number CHAR(15), City
   VARCHAR(30), STATE VARCHAR (10))
   LANGUAGE SQL
   RETURN VALUES
   (A..Street, A..Number, A..City, A..State)

 CREATE FUNCTION US_addresstofunc(A US_addr_t)
   RETURNS ROW
   (Street VARCHAR(30), Number CHAR(15), City
   VARCHAR(30), STATE VARCHAR (10), Zip
   CHAR(10))
   LANGUAGE SQL
   RETURN VALUES
   (A..Street, A..Number, A..City, A..State, A..Zip)
```

ステップ 2. それぞれのタイプ・バリエーションにつき 1 つの transform グループを作成する。

```
CREATE TRANSFORM FOR Address_t
   funcgroup1 (FROM SQL WITH FUNCTION addresstofunc)

CREATE TRANSFORM FOR US_addr_t
   funcgroup2 (FROM SQL WITH FUNCTION US_addresstofunc)
```

ステップ 3. それぞれのタイプ・バリエーションにつき 1 つの外部 UDF を作成する。

Address_t タイプのための外部 UDF を登録する。

```
CREATE FUNCTION address_to_client (A Address_t)
   RETURNS VARCHAR(150)
   LANGUAGE C
   EXTERNAL NAME 'addressudf!address_to_client'
   ...
   TRANSFORM GROUP funcgroup1
```

address_to_client UDF を作成する。

```
void SQL_API_FN address_to_client(
   SQLUDF_VARCHAR *street,
   SQLUDF_CHAR    *number,
   SQLUDF_VARCHAR *city,
   SQLUDF_VARCHAR *state,
   SQLUDF_VARCHAR *output,

   /* Null indicators for attributes */
   SQLUDF_NULLIND *street_ind,
   SQLUDF_NULLIND *number_ind,
   SQLUDF_NULLIND *city_ind,
   SQLUDF_NULLIND *state_ind,
   /* Null indicator for instance */
   SQLUDF_NULLIND *address_ind,
   /* Null indicator for output */
   SQLUDF_NULLIND *output_ind,
   SQLUDF_TRAIL_ARGS)

{
   sprintf (output, "[address_t] [Street:%s] [number:%s]
   [city:%s] [state:%s]",
   street, number, city, state);
   *output_ind = 0;
}
```

US_addr_t タイプのための外部 UDF を登録する。

```
CREATE FUNCTION address_to_client (A US_addr_t)
   RETURNS VARCHAR(150)
   LANGUAGE C
   EXTERNAL NAME 'addressudf!US_addr_to_client'
   ...
   TRANSFORM GROUP funcgroup2
```

US_addr_to_client UDF を作成する。

```
void SQL_API_FN US_address_to_client(
   SQLUDF_VARCHAR  *street,
   SQLUDF_CHAR     *number,
   SQLUDF_VARCHAR  *city,
   SQLUDF_VARCHAR  *state,
   SQLUDF_CHAR     *zip,
   SQLUDF_VARCHAR  *output,

   /* Null indicators */
   SQLUDF_NULLIND  *street_ind,
   SQLUDF_NULLIND  *number_ind,
   SQLUDF_NULLIND  *city_ind,
   SQLUDF_NULLIND  *state_ind,
   SQLUDF_NULLIND  *zip_ind,
   SQLUDF_NULLIND  *us_address_ind,
   SQLUDF_NULLIND  *output_ind,
   SQLUDF_TRAIL_ARGS)

{
   sprintf (output, "[US_addr_t] [Street:%s] [number:%s]
```

```
    [city:%s] [state:%s] [zip:%s]",
    street, number, city, state, zip);
    *output_ind = 0;
}
```

ステップ 4. インスタンスを処理するための正しい外部 UDF を選択する、SQL を本体として持つ UDF を作成する。次の UDF は、UNION ALL 文節で結合された SELECT ステートメントで TREAT を指定して、正しい FROM SQL クライアント transform を呼び出します。

```
CREATE FUNCTION addr_stream (ab Address_t)
   RETURNS VARCHAR(150)
   LANGUAGE SQL
   RETURN
   WITH temp(addr) AS
   (SELECT address_to_client(ta.a)
      FROM TABLE (VALUES (ab)) AS ta(a)
      WHERE ta.a IS OF (ONLY Address_t)
      UNION ALL
   SELECT address_to_client(TREAT (tb.a AS US_addr_t))
      FROM TABLE (VALUES (ab)) AS tb(a)
      WHERE tb.a IS OF (ONLY US_addr_t))
   SELECT addr FROM temp;
```

これで、アプリケーションは、 Addr_stream 関数を呼び出して、ふさわしい外部 UDF を呼び出すことができます。

```
SELECT Addr_stream(Address)
   FROM Employee;
```

ステップ 5. Addr_stream 外部 UDF を、 Address_t のための FROM SQL *client* transform として追加する。

```
CREATE TRANSFORM GROUP FOR Address_t
   client_group (FROM SQL
   WITH FUNCTION Addr_stream)
```

注: アプリケーションが、タイプ述部を使用して照会の中で特定の住所を指定する場合は、 Addr_stream を FROM SQL として US_addr_t のための client transform に追加します。これで、照会が US_addr_t のインスタンスを明確に要求する時に、Addr_stream を呼び出すことができるようになります。

ステップ 6. TRANSFORM GROUP オプションを client_group に設定して、アプリケーションをバインドする。

```
PREP myprogram TRANSFORM GROUP client_group
```

DB2 が SELECT Address FROM Person INTO :hvar ステートメントを含むアプリケーションをバインドする時は、 DB2 は FROM SQL client transform を探します。DB2 は、構造化型がバインドアウトされていることを認識し、transform グループ client_group の中を探します。なぜなら、これはステップ 6 でバインド時に指定された TRANSFORM GROUP であるからです。

transform グループには、ステップ 5 のルート・タイプ Address_t に関連した transform 関数 Addr_stream が入っています。 Addr_stream は、ステップ 4 で定義されている SQL を本体として持つ関数なので、これは他の transform 関数とは従属関係を持っていません。 Addr_stream 関数は、 :hvaddr ホスト変数が要求するデータ型 VARCHAR(150) を戻します。

Addr_stream 関数は、タイプ Address_t (この例では US_addr_t で代用できる) の入力値を取り、入力値の動的タイプを決定します。動的タイプを決定する時に Addr_stream は、動的タイプが Address_t である場合は、 address_to_client という値に対して、動的タイプが US_addr_t である場合は、 USaddr_to_client という値に対して、対応する外部 UDF を呼び出します。これらの 2 つの UDF はステップ 3 (328 ページ) で定義されます。これらの UDF は両方とも、それぞれの構造化型を、 Addr_stream transform 関数が要求するタイプである VARCHAR(150) に分解します。

構造化型を入力として受け入れるために、それぞれの UDF には、入力構造化型インスタンスを個々の属性パラメーターに分解するための FROM SQL transform 関数が必要です。ステップ 3 (328 ページ) の CREATE FUNCTION ステートメントは、これらの transform が入っている TRANSFORM GROUP に名前を付けています。

transform 関数のための CREATE FUNCTION ステートメントは、ステップ 1 (328 ページ) で発行されています。 transform 関数を transform グループと関連付ける CREATE TRANSFORM ステートメントは、ステップ 2 (328 ページ) で発行されています。

関連概念:

- 327 ページの『transform 関数の要件』
- 312 ページの『Transform 関数と Transform グループ』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION ステートメント』

サブタイプ・データを DB2 に戻す

次の構文を使用して、構造化型をアプリケーションから DB2 データベースに挿入するとします。

```
INSERT INTO person (Oid, Name, Address)
   VALUES ('n', 'Norm', :hvaddr);
```

構造化型のための INSERT ステートメントを実行するには、次のようにします。

ステップ 1. 住所のそれぞれのバリエーションのための TO SQL transform 関数を作成する。次の例は、Address_t および US_addr_t タイプを変換する、 SQL を本体としてもつ UDF を示しています。

```
CREATE FUNCTION functoaddress
 (str VARCHAR(30), num CHAR(15), cy VARCHAR(30), st VARCHAR (10))
RETURNS Address_t
LANGUAGE SQL
RETURN Address_t()..street(str)..number(num)..city(cy)..state(st);

CREATE FUNCTION functoaddress
 (str VARCHAR(30), num CHAR(15), cy VARCHAR(30), st VARCHAR (10),
  zp CHAR(10))
RETURNS US_addr_t
LANGUAGE SQL
RETURN US_addr_t()..street(str)..number(num)..city(cy)
    ..state(st)..zip(zp);
```

ステップ 2. それぞれのタイプ・バリエーションにつき 1 つの transform グループを作成する。

```
CREATE TRANSFORM FOR Address_t
   funcgroup1 (TO SQL
   WITH FUNCTION functoaddress);

CREATE TRANSFORM FOR US_addr_t
   funcgroup2 (TO SQL
   WITH FUNCTION functousaddr);
```

ステップ 3. それぞれのタイプ・バリエーションにつき 1 つのエンコードされた住所タイプを戻す外部 UDF を作成する。

Address_t タイプのための外部 UDF を登録する。

```
CREATE FUNCTION client_to_address (encoding VARCHAR(150))
   RETURNS Address_t
   LANGUAGE C
   TRANSFORM GROUP funcgroup1
   ...
   EXTERNAL NAME 'address!client_to_address';
```

client_to_address の Address_t バージョンのための外部 UDF を作成する。

```
void SQL_API_FN client_to_address (
   SQLUDF_VARCHAR *encoding,
   SQLUDF_VARCHAR *street,
   SQLUDF_CHAR    *number,
   SQLUDF_VARCHAR *city,
   SQLUDF_VARCHAR *state,

   /* Null indicators */
   SQLUDF_NULLIND *encoding_ind,
   SQLUDF_NULLIND *street_ind,
   SQLUDF_NULLIND *number_ind,
   SQLUDF_NULLIND *city_ind,
   SQLUDF_NULLIND *state_ind,
   SQLUDF_NULLIND *address_ind,
   SQLUDF_TRAIL_ARGS )
{
   char c[150];
   char *pc;

   strcpy(c, encoding);

   pc = strtok (c, ":]");
   pc = strtok (NULL, ":]");
   pc = strtok (NULL, ":]");
   strcpy (street, pc);
   pc = strtok (NULL, ":]");
   pc = strtok (NULL, ":]");
   strcpy (number, pc);
   pc = strtok (NULL, ":]");
   pc = strtok (NULL, ":]");
   strcpy (city, pc);
   pc = strtok (NULL, ":]");
   pc = strtok (NULL, ":]");
   strcpy (state, pc);

   *street_ind = *number_ind = *city_ind
   = *state_ind = *address_ind = 0;
}
```

US_addr_t タイプのための外部 UDF を登録する。

```
CREATE FUNCTION client_to_us_address (encoding VARCHAR(150))
   RETURNS US_addr_t
   LANGUAGE C
```

```
   TRANSFORM GROUP funcgroup1
   ...
   EXTERNAL NAME 'address!client_to_US_addr';
```

client_to_address の US_addr_t バージョンのための外部 UDF を作成する。

```
void SQL_API_FN client_to_US_addr(
   SQLUDF_VARCHAR *encoding,
   SQLUDF_VARCHAR *street,
   SQLUDF_CHAR    *number,
   SQLUDF_VARCHAR *city,
   SQLUDF_VARCHAR *state,
   SQLUDF_VARCHAR *zip,

   /* Null indicators */
   SQLUDF_NULLIND *encoding_ind,
   SQLUDF_NULLIND *street_ind,
   SQLUDF_NULLIND *number_ind,
   SQLUDF_NULLIND *city_ind,
   SQLUDF_NULLIND *state_ind,
   SQLUDF_NULLIND *zip_ind,
   SQLUDF_NULLIND *us_addr_ind,
   SQLUDF_TRAIL_ARGS)

{
   char c[150];
   char *pc;

   strcpy(c, encoding);

   pc = strtok (c, ":]");
   pc = strtok (NULL, ":]");
   pc = strtok (NULL, ":]");
   strcpy (street, pc);
   pc = strtok (NULL, ":]");
   pc = strtok (NULL, ":]");
   strncpy (number, pc,14);
   pc = strtok (NULL, ":]");
   pc = strtok (NULL, ":]");
   strcpy (city, pc);
   pc = strtok (NULL, ":]");
   pc = strtok (NULL, ":]");
   strcpy (state, pc);
   pc = strtok (NULL, ":]");
   pc = strtok (NULL, ":]");
   strncpy (zip, pc, 9);

   *street_ind = *number_ind = *city_ind
   = *state_ind = *zip_ind = *us_addr_ind = 0;
}
```

ステップ 4. インスタンスを処理するための正しい外部 UDF を選択する、SQL を本体として持つ UDF を作成する。次の UDF は、TYPE 述部を指定して、正しい client transform を呼び出します。結果は、一時表に置かれます。

```
CREATE FUNCTION stream_address (ENCODING VARCHAR(150))
   RETURNS Address_t
   LANGUAGE SQL
   RETURN
   (CASE(SUBSTR(ENCODING,2,POSSTR(ENCODING,']')-2))
   WHEN 'address_t'
      THEN client_to_address(ENCODING)
```

```
    WHEN 'us_addr_t'
       THEN client_to_us_addr(ENCODING)
    ELSE NULL
    END);
```

ステップ 5. stream_address UDF を、 Address_t のための TO SQL client transform として追加する。

```
CREATE TRANSFORM FOR Address_t
   client_group (TO SQL
   WITH FUNCTION stream_address);
```

ステップ 6. TRANSFORM GROUP オプションを client_group に設定して、アプリケーションをバインドする。

```
PREP myProgram2 TRANSFORM GROUP client_group
```

構造化型がバインドされた INSERT ステートメントが、アプリケーションに含まれている時は、 DB2 は TO SQL client transform を探します。 DB2 は、transform グループ client_group で transform を探します。なぜなら、これはステップ 6 でバインド時に指定された TRANSFORM GROUP だからです。 DB2 は、必要とする transform 関数 stream_address を見付けます。これは、ステップ 5 でルート・タイプ Address_t と関連付けられています。

stream_address は、ステップ 4 (333 ページ) で定義されている SQL を本体として持つ関数であるため、追加の transform 関数とは明示された従属関係を持っていません。入力パラメーターについては、 stream_address は VARCHAR(150) を受け入れます。これはアプリケーション・ホスト変数 :hvaddr に対応するものです。stream_address は、正しいルート・タイプ Address_t の値であり、正しい動的タイプの値でもある値を戻します。

stream_address は、 VARCHAR(150) 入力パラメーターを解析して、動的タイプ (この事例では、 'Address_t' か 'US_addr_t' のいずれか) を指定するサブストリングを探します。次に stream_address は、対応する外部 UDF を呼び出して、VARCHAR(150) を解析し、指定のタイプのオブジェクトを戻します。それぞれのタイプを戻す 2 つの client_to_address() UDF があります。これらの UDF はステップ 3 (332 ページ) で定義されています。それぞれの UDF は入力 VARCHAR(150) を取り、ふさわしい構造化型の属性を内部的に構成し、このようにして構造化型を戻します。

構造化型を戻すために、それぞれの UDF には、出力属性値を構造化型のインスタンスに構成するための TO SQL transform 関数が必要です。ステップ 3 (332 ページ) の CREATE FUNCTION ステートメントは、 transform が入っている TRANSFORM GROUP に名前を付けています。

ステップ 1 (331 ページ) の SQL を本体として持つ transform 関数と、ステップ 2 (331 ページ) の transform グループとの関連は、ステップ 3 (332 ページ) の CREATE FUNCTION ステートメントの中で指定されています。

関連概念:

- 327 ページの『transform 関数の要件』
- 312 ページの『Transform 関数と Transform グループ』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE FUNCTION ステートメント』

構造化型のホスト変数

構造化型ホスト変数の宣言

静的 SQL の中の構造化型ホスト変数を検索または送信するには、その構造化型を表すのに使用される組み込みタイプを指示する SQL 宣言を提供する必要があります。この宣言の形式は次のとおりです。

```
EXEC SQL BEGIN DECLARE SECTION ;

   SQL TYPE IS structured_type AS base_type host-variable-name ;

EXEC SQL END DECLARE SECTION;
```

たとえば、タイプ Address_t は、クライアント・アプリケーションに渡される時に、可変長文字タイプに変換されるとしましょう。 Address_t タイプのホスト変数には、次の宣言を使用します。

```
SQL TYPE IS Address_t AS VARCHAR(150) addrhv;
```

関連概念:

- 312 ページの『Transform 関数と Transform グループ』

関連タスク:

- 335 ページの『構造化型の記述』

構造化型の記述

構造化型変数を指定した DESCRIBE ステートメントを使用すると、 FROM SQL transform 関数の結果タイプの記述が、 DB2 によって SQLDA の基本 SQLVAR の SQLTYPE フィールドに入れられます。しかし、CURRENT DEFAULT TRANSFORM GROUP 特殊レジスターを使用して TRANSFORM GROUP が指定されていないか、指定したグループに FROM SQL transform 関数が定義されていないというどちらかの理由で、 FROM SQL transform 関数が定義されていない場合は、DESCRIBE はエラーを戻します。

構造化型の実際の名前は、SQLVAR2 の中に戻されます。

関連概念:

- 312 ページの『Transform 関数と Transform グループ』

関連タスク:

- 335 ページの『構造化型ホスト変数の宣言』

第 9 章 トリガー

アプリケーション開発でのトリガー

データベース・マネージャーを非アクティブなシステムからアクティブなシステムに変えるには、 SQL トリガーで表される機能を使用してください。

SQL *トリガー* は、単一の基本表と関連した名前付き規則です。トリガーでは、トリガー・イベントの発生時に条件付きで活動化されるアクションを指定します。この場合のトリガー・イベントは、特定の基本表をターゲットとした表の変更 (INSERT、UPDATE、または DELETE) です。また、トリガー・アクションをいつ実行するか (つまり、トリガー・イベントの前か後か) も指定します。トリガーの作成とドロップはそれぞれ、 CREATE TRIGGER ステートメント、DROP TRIGGER ステートメントで行います。以下の図に、トリガーの基本論理構造を示します。

トリガーは、それ自体 1 つのロジックと見なすことができます。つまり、イベントの発生時に、指定の条件が真の場合、アクションが実行される、というロジックです。イベントは、表に対するデータベース操作です。条件は、操作のイベント発生時のデータベースの状態か、または表の過渡的な状態です。アクションとしては、データベースをさらに変更する 1 つ以上の SQL ステートメントの実行、変更操作の実行を抑制するための例外のスロー、イベント操作で変更したデータの修正など、プロシージャーや関数の呼び出しに論理的に組み込めるあらゆるアクションを指定できます。プロシージャーや関数には複合ロジックも組み込めます。また、プロシージャーや関数はトリガーのサブルーチンとして使用できます。外部のユーザー定義関数やプロシージャーを使用して、トリガーから電子メールを送信することや、ファイル・システムのファイルにデータを書き込むことも可能です。

以下の図に、トリガーの論理構造を示します。

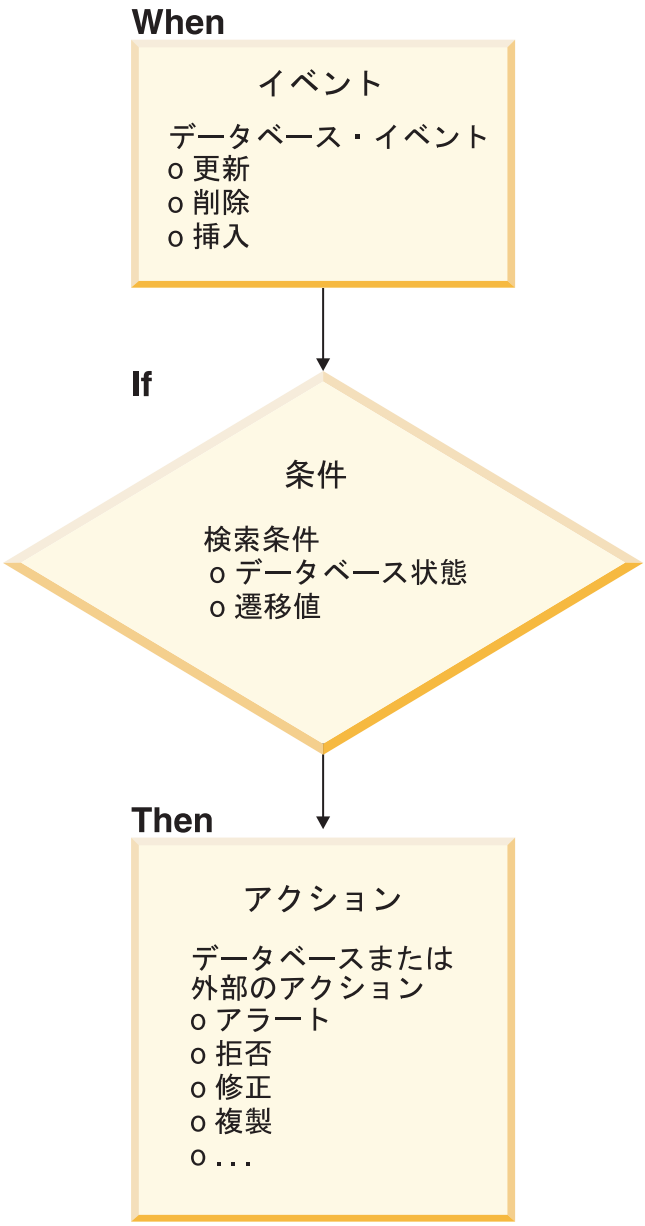


図 16. ルーチンの種別

トリガーを使用すると、業務規則などの一般的な保全形式をサポートすることができます。たとえば、カスタマーの要求以上の注文を断りたい業務があるとします。この制約は、トリガーを使用して実施することができます。トリガーは一般に、*過渡的な* 業務規則を収集する強力な機構です。過渡的な業務規則は、さまざまな状態のデータを含む規則です。

たとえば、給料は 10% 以上増やせないと仮定します。この規則をチェックするには、増える前と後の給料の額を比較しなければなりません。データの状態が 1 つしか関係していない規則では、チェックと参照保全に関する制約のほうが適しています。チェックおよび参照に関する制約は、その宣言のセマンティクスのため、過渡的でない制約として使用することをお勧めします。

また、サマリー・データを自動的に更新するタスクなどに対してトリガーを使うこともできます。トリガーは、そのようなアクションをデータベースの一部として保持したり、それらが自動的に行われることを確認することにより、データベースの保全性を高めます。たとえば、ある会社で雇用されている従業員数を自動的に追跡したいとします。

```
Tables: EMPLOYEE (from the Sample Tables)
  COMPANY_STATS (NBEMP, NBPRODUCT, REVENUE)
```

次の 2 つのトリガーを定義できます。

- 新しい人が雇われるたびに、すなわち新しい行が表 EMPLOYEE に挿入されるたびに従業員数を増分するトリガー。

```
CREATE TRIGGER NEW_HIRED
  AFTER INSERT ON EMPLOYEE
  FOR EACH ROW MODE DB2SQL
  UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1
```

- 従業員が会社を辞めるたびに、すなわち 1 つの行が表 EMPLOYEE から削除されるたびに従業員数を減少するトリガー。

```
CREATE TRIGGER FORMER_EMP
  AFTER DELETE ON EMPLOYEE
  FOR EACH ROW MODE DB2SQL
  UPDATE COMPANY_STATS SET NBEMP = NBEMP - 1
```

具体的には、以下の目的でトリガーを使用できます。

- SIGNAL SQLSTATE SQL ステートメントおよび組み込み RAISE_ERROR 関数を使用して入力データの妥当性を検査する。または、無効なデータが発見された場合にエラーの発生を示す SQLSTATE を戻すストアード・プロシージャー (シリアルのみ) または UDF を呼び出す。過渡的でないデータの妥当性チェックは、通常はチェックおよび参照に関する制約により正しく処理されます。対照的に、トリガーは過渡的なデータの妥当性検査、すなわち更新操作の前と後の値を比較する必要がある妥当性検査に適しています。
- 新しく挿入された行に対して値を自動的に生成する (これは*代理関数* として知られている)。つまり、その行の別の値や他の表の値に基づく可能性のあるユーザー定義のデフォルト値をインプリメントすることです。従属列を機能的にインプリメントするために、DB2® は GENERATED 列もサポートします。これらの列が持つ値は、常に同じ行の他の値から決定的な形で派生します。
- 相互参照の目的で他表からの読み取りを行う。
- 監査証跡の目的で他表への書き込みを行う。
- *警告* をサポートする (たとえば電子メールのメッセージを介して)。

データベース・マネージャーでトリガーを使用すると、次のような結果が得られます。

- **迅速なアプリケーション開発**

  トリガーはリレーショナル・データベースに保管されるので、トリガーにより実行されるアクションは各アプリケーションごとにコーディングする必要はありません。

- **業務規則のグローバルな実施**

トリガーは一度定義するだけで、表を変更する任意のアプリケーションに対して使用することができます。

- **簡単な保守**

  営業方針の変更時には、各アプリケーション・プログラムを変更する代わりにそれに対応するトリガーだけを変更する必要があります。

トリガー SQL ステートメントを実行すると、別の、または同じイベントのトリガーが発生し、順々に他のトリガー (または同じトリガーの 2 番目のインスタンス) を活動させることがあります。したがって、あるトリガーを活動化すると他の複数のトリガーの活動化をカスケードすることができます。

サポートされているトリガー・カスケードの実行時の深度レベルは 16 です。レベル 17 のトリガーが活動化されると SQLCODE -724 (SQLSTATE 54038) が戻され、トリガー・ステートメントはロールバックされます。

関連概念:

- 341 ページの『INSERT、UPDATE、および DELETE トリガー』
- 345 ページの『トリガーの細分性』
- 346 ページの『トリガー活動化時間』
- 342 ページの『参照制約とのトリガーの対話』
- 343 ページの『トリガー作成のガイドライン』
- 342 ページの『INSTEAD OF トリガー』

関連タスク:

- 344 ページの『トリガーの作成』
- 358 ページの『トリガーを使用した業務規則の定義』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE TRIGGER ステートメント』

関連サンプル:

- 『tbtrig.out -- HOW TO USE TRIGGERS (C)』
- 『tbtrig.sqc -- How to use a trigger on a table (C)』
- 『tbtrig.out -- HOW TO USE TRIGGERS (C++)』
- 『tbtrig.sqC -- How to use a trigger on a table (C++)』
- 『trigsql.sqb -- How to use a trigger on a table (IBM COBOL)』
- 『TbTrig.java -- How to use triggers (JDBC)』
- 『TbTrig.out -- HOW TO USE TRIGGERS. Connect to 'sample' database using JDBC type 2 driver (JDBC)』
- 『TbTrig.out -- HOW TO USE TRIGGERS. Connect to 'sample' database using JDBC type 2 driver (SQLJ)』
- 『TbTrig.sqlj -- How to use triggers (SQLj)』

INSERT、UPDATE、および DELETE トリガー

トリガーはどれもあるイベントと関連しています。トリガーは、それに対応するイベントがデータベースで発生すると活動化されます。このトリガー・イベントは、特定のイベント、すなわち UPDATE、INSERT、または DELETE (参照を制約するアクションにより生じる操作を含む) が対象表で実行される際に発生します。以下に例を示します。

```
CREATE TRIGGER NEW_HIRE
  AFTER INSERT ON EMPLOYEE
  FOR EACH ROW
  UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1
```

上のステートメントは、挿入操作が表 employee で行われる際に活動化されるトリガー new_hire を定義します。

どのトリガー・イベントも (したがってどのトリガーも)、 1 つだけの対象表と 1 つだけの変更操作に関連付けることができます。以下のような変更操作があります。

挿入操作

挿入操作は INSERT ステートメントによってのみ行われます。したがって、LOAD コマンドなどの INSERT を使用しないユーティリティーを用いてデータをロードすると、トリガーは活動化されません。

更新操作

更新操作は、UPDATE ステートメントか、 ON DELETE SET NULL の参照を制約する規則の結果として行われます。

削除操作

削除操作は、DELETE ステートメントか、 ON DELETE CASCADE の参照を制約する規則の結果として行われます。

トリガー・イベントが更新操作である場合、そのイベントは対象表の特定の列と関連させることができます。この場合のトリガーは、更新操作が特定の列のどれかを更新しようとする場合に限り活動化されます。これにより、トリガーを活動化するイベントをさらに細分化することができます。

たとえば、次のトリガー REORDER は、更新操作を表 PARTS の列 ON_HAND か MAX_STOCKED で実行する場合に限り活動化します。

```
CREATE TRIGGER REORDER
  AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
  REFERENCING NEW AS N_ROW
  FOR EACH ROW
  WHEN (N_ROW.ON_HAND < 0.10 * N_ROW.MAX_STOCKED)
  BEGIN ATOMIC
  VALUES(ISSUE_SHIP_REQUEST(N_ROW.MAX_STOCKED -
                            N_ROW.ON_HAND,
                            N_ROW.PARTNO));
  END
```

関連概念:

- 345 ページの『トリガーの細分性』
- 346 ページの『トリガー活動化時間』
- 342 ページの『参照制約とのトリガーの対話』

- 337 ページの『アプリケーション開発でのトリガー』
- 342 ページの『INSTEAD OF トリガー』

関連タスク:

- 344 ページの『トリガーの作成』

参照制約とのトリガーの対話

トリガー・イベントは参照制約が原因で変更されることがあります。たとえば、DEPT と EMP という 2 つの表があるとすれば、 DEPT を削除および更新すると参照保全制約により EMP も伝搬して削除および更新され、 EMP で定義された削除および更新トリガーは、DEPT で定義された参照制約の結果として活動化されます。 EMP でのトリガーはその活動化時間にしたがって、 EMP 内の行の削除 (ON DELETE CASCADE の場合) または更新 (ON DELETE SET NULL の場合) の前 (BEFORE) か後 (AFTER) に実行されます。

関連概念:

- 341 ページの『INSERT、UPDATE、および DELETE トリガー』
- 345 ページの『トリガーの細分性』
- 337 ページの『アプリケーション開発でのトリガー』

INSTEAD OF トリガー

INSTEAD OF トリガーは、ビューが複雑であるため、挿入、更新、および削除操作を固有にサポートできない場合に、それらのビューに対して操作を実行する方法を記述します。 INSTEAD OF トリガーにより、アプリケーションはすべての SQL 操作 (挿入、削除、更新、および選択) のための単独インターフェースとしてビューを使用できます。通常、INSTEAD OF トリガーはビュー本体で適用されている論理の逆のものを含んでいます。たとえば、列のソース表から列を復号するビューについて考えます。このビューの INSTEAD OF トリガーはデータを暗号化し、それをソース表に挿入します。こうして、対称操作を実行します。

INSTEAD OF トリガーを使用すると、指定に対して要求された変更操作がトリガー論理で置き換えられ、ビューに代わって操作を実行します。アプリケーション側から見ると、これは透過的に行われるため、すべての操作がビューに対して実行されているように見えます。対象となる特定のビューに対して実行される操作ごとに、1 つの INSTEAD OF トリガーだけが許可されます。

ビュー自体は非型付きビューまたは非型付きビューに対する別名でなければなりません。さらに、WITH CHECK OPTION (対称ビュー) を使用して定義されたビューや、対称ビューが直接的または間接的に定義されているビューであってはなりません。

以下の例は、定義されたビュー (EMPV) に対して INSERT、UPDATE、および DELETE のための論理を提供する 3 つの INSTEAD OF トリガーを示します。ビュー EMPV にはその from 文節からの結合が含まれているため、変更操作を固有にサポートすることはできません。

```
CREATE VIEW EMPV(EMPNO, FIRSTNME, MIDINIT, LASTNAME, PHONENO,
                 HIREDATE, DEPTNAME)
AS SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME, PHONENO,
          HIREDATE, DEPTNAME
          FROM EMPLOYEE, DEPARTMENT WHERE
               EMPLOYEE.WORKDEPT = DEPARTMENT.DEPTNO

CREATE TRIGGER EMPV_INSERT INSTEAD OF INSERT ON EMPV
REFERENCING NEW AS NEWEMP FOR EACH ROW
INSERT INTO EMPLOYEE (EMPNO, FIRSTNME, MIDINIT, LASTNAME,
                      WORKDEPT, PHONENO, HIREDATE)
        VALUES(EMPNO, FIRSTNME, MIDINIT, LASTNAME,
               COALESCE((SELECT DEPTNO FROM DEPARTMENT AS D
                         WHERE D.DEPTNAME = NEWEMP.DEPTNAME),
                        RAISE_ERROR('70001', 'Unknown dept name')),
               PHONENO, HIREDATE)

CREATE TRIGGER EMPV_UPDATE INSTEAD OF UPDATE ON EMPV
REFERENCING NEW AS NEWEMP OLD AS OLDEMP
  FOR EACH ROW
BEGIN ATOMIC
 VALUES(CASE WHEN NEWEMP.EMPNO = OLDEMP.EMPNO THEN 0
             ELSE RAISE_ERROR('70002', 'Must not change EMPNO') END);
 UPDATE EMPLOYEE AS E
   SET (FIRSTNME, MIDINIT, LASTNAME, WORKDEPT, PHONENO, HIREDATE)
     = (NEWEMP.FIRSTNME, NEWEMP.MIDINIT, NEWEMP.LASTNAME,
               COALESCE((SELECT DEPTNO FROM DEPARTMENT AS D
                         WHERE D.DEPTNAME = NEWEMP.DEPTNAME),
                        RAISE_ERROR ('70001', 'Unknown dept name')),
               NEWEMP.PHONENO, NEWEMP.HIREDATE)
 WHERE NEWEMP.EMPNO = E.EMPNO;
END

CREATE TRIGGER EMPV_DELETE INSTEAD OF DELETE ON EMPV
REFERENCING OLD AS OLDEMP FOR EACH ROW
DELETE FROM EMPLOYEE AS E WHERE E.EMPNO = OLDEMP.EMPNO
```

関連概念:

- 341 ページの『INSERT、UPDATE、および DELETE トリガー』
- 337 ページの『アプリケーション開発でのトリガー』

関連タスク:

- 344 ページの『トリガーの作成』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE TRIGGER ステートメント』

トリガー作成のガイドライン

トリガーを作成したら、それを表と関連付ける必要があります。この表は、トリガーの*対象表* と呼ばれます。*変更操作* という用語は、対象表の状態に対する何らかの変更を意味します。変更操作は、以下のいずれかによって開始されます。

- INSERT ステートメント
- UPDATE ステートメント、または UPDATE を実行する参照制約
- DELETE ステートメント、または DELETE を実行する参照制約

各トリガーをこれら 3 つのタイプの変更操作の 1 つと関連付ける必要があります。この関連付けは、その特定のトリガーの*トリガー・イベント* と呼ばれます。

さらに、トリガー・イベントが発生する際にトリガーによって実行されるトリガー・アクション というアクションを定義する必要もあります。トリガー・アクションは 1 つまたは複数の SQL ステートメントから構成され、データベース・マネージャーがトリガー・イベントを実行する前または後に実行されます。トリガー・イベントが発生すると、データベース・マネージャーは対象表の中から変更操作の影響を受ける一連の行を判別して、トリガーを実行します。

トリガーを作成する際には、以下のような属性および振る舞いを宣言する必要があります。

- トリガーの名前
- 対象表の名前
- トリガー活動化時間 (変更操作実行の BEFORE または AFTER)
- トリガー・イベント (INSERT、DELETE、または UPDATE)
- 以前の値の遷移変数 (存在する場合)
- 新しい値の遷移変数 (存在する場合)
- 以前の値の遷移表 (存在する場合)
- 新しい値の遷移表 (存在する場合)
- 細分性 (FOR EACH STATEMENT または FOR EACH ROW)
- トリガーのトリガー・アクション (トリガー・アクション条件とトリガー SQL ステートメントを含む)
- トリガー・イベントが UPDATE の場合、トリガーのトリガー・イベントに対するトリガー列リスト。またその他に、トリガー列リストが明示か暗黙かの指示。

関連概念:

- 341 ページの『INSERT、UPDATE、および DELETE トリガー』
- 345 ページの『トリガーの細分性』
- 346 ページの『トリガー活動化時間』
- 337 ページの『アプリケーション開発でのトリガー』

関連タスク:

- 344 ページの『トリガーの作成』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE TRIGGER ステートメント』

トリガーの作成

コントロール・センターからトリガーを作成するには、「トリガーの作成」ダイアログを使用します。「トリガーの作成」ダイアログを見付けるには、オブジェクト・ツリーを展開して、「トリガー」フォルダーを右クリックします。

コマンド行を使用してトリガーを作成するには、次の CREATE TRIGGER ステートメントのテンプレートを使用します。

```
CREATE TRIGGER <name>
 <action> ON <table_name>
 <operation>
 <triggered_action>
```

以下の SQL ステートメントは、新しい人が雇われるたびに従業員数を増分するトリガーを作成します。それには、EMPLOYEE 表に行が追加されるたびに COMPANY_STATS 表の従業員数 (NBEMP) 列に 1 を加えます。

```
CREATE TRIGGER NEW_HIRED
 AFTER INSERT ON EMPLOYEE
 FOR EACH ROW
 UPDATE COMPANY_STATS SET NBEMP = NBEMP+1;
```

関連概念:

- 341 ページの『INSERT、UPDATE、および DELETE トリガー』
- 345 ページの『トリガーの細分性』
- 346 ページの『トリガー活動化時間』
- 337 ページの『アプリケーション開発でのトリガー』
- 343 ページの『トリガー作成のガイドライン』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE TRIGGER ステートメント』

関連サンプル:

- 『tbtrig.out -- HOW TO USE TRIGGERS (C)』
- 『tbtrig.sqc -- How to use a trigger on a table (C)』
- 『tbtrig.out -- HOW TO USE TRIGGERS (C++)』
- 『tbtrig.sqC -- How to use a trigger on a table (C++)』
- 『trigsql.sqb -- How to use a trigger on a table (IBM COBOL)』
- 『TbTrig.java -- How to use triggers (JDBC)』
- 『TbTrig.out -- HOW TO USE TRIGGERS. Connect to 'sample' database using JDBC type 2 driver (JDBC)』
- 『TbTrig.out -- HOW TO USE TRIGGERS. Connect to 'sample' database using JDBC type 2 driver (SQLJ)』
- 『TbTrig.sqlj -- How to use triggers (SQLj)』

トリガーの細分性

トリガーは、活動化されると次のような細分性に従って実行されます。

FOR EACH ROW
: 影響される行の数と同じ回数だけ実行されます。トリガー・アクションに影響される特定の行を参照する必要がある場合、 FOR EACH ROW 細分性を使用します。この例として、AFTER UPDATE トリガーで更新行の新しい値と古い値を比較することが挙げられます。

FOR EACH STATEMENT
: トリガー・イベントに対して一度だけ実行されます。

影響される行が空の場合 (すなわち、 WHERE 文節が行を限定しなかった探索済み UPDATE または DELETE の場合)、 FOR EACH ROW トリガーは実行されません。ただし、FOR EACH STATEMENT トリガーはやはり一度実行されます。

たとえば、FOR EACH ROW を使用して従業員数の計算を保持することができます。

```
CREATE TRIGGER NEW_HIRED
  AFTER INSERT ON EMPLOYEE
  FOR EACH ROW
  UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1
```

FOR EACH STATEMENT の細分性を使用して更新を行っても同じ結果が得られます。

```
CREATE TRIGGER NEW_HIRED
  AFTER INSERT ON EMPLOYEE
  REFERENCING NEW_TABLE AS NEWEMPS
  FOR EACH STATEMENT
  UPDATE COMPANY_STATS
  SET NBEMP = NBEMP + (SELECT COUNT(*) FROM NEWEMPS)
```

注: FOR EACH STATEMENT の細分性は、 BEFORE トリガーにはサポートされません。

関連概念:

- 341 ページの『INSERT、UPDATE、および DELETE トリガー』
- 346 ページの『トリガー活動化時間』
- 337 ページの『アプリケーション開発でのトリガー』
- 343 ページの『トリガー作成のガイドライン』

関連タスク:

- 344 ページの『トリガーの作成』

トリガー活動化時間

トリガー活動化時間 は、いつトリガーを活動化するかを指定します。すなわち、そのトリガー・イベントが実行される BEFORE、AFTER、 INSTEAD OF のどれかを指定します。たとえば、次のトリガーの活動化時間は employee での INSERT 操作の後 (AFTER) です。

```
CREATE TRIGGER NEW_HIRE
  AFTER INSERT ON EMPLOYEE
  FOR EACH ROW
  UPDATE COMPANY_STATS SET NBEMP = NBEMP + 1
```

活動化時間が BEFORE の場合、トリガー・アクションは、トリガー・イベントが実行される前に影響される行のそれぞれに対して活動化されます。そのため、BEFORE トリガーがそれぞれの行に対する実行を完了した後にのみ、サブジェクト表は変更されます。 BEFORE トリガーは、FOR EACH ROW の細分性を持たなければならないことに注意してください。

活動化時間が AFTER の場合、トリガー・アクションは、影響される行のそれぞれに対して、またはステートメントに対して、トリガーの細分性に従って活動化されます。これはトリガー・イベントが実行された後、またトリガー・イベントによって影響される可能性がある制約 (参照制約のアクションも含む) すべてをデータベース・マネージャーがチェックした後で起きます。 AFTER トリガーは、FOR EACH ROW か FOR EACH STATEMENT のどちらかの細分性を持つことができます。

活動化時間が INSTEAD OF の場合、トリガー・アクションは、トリガー・イベントを実行する代わりに、影響される行のセットにある行のそれぞれに対して活動化されます。 INSTEAD OF トリガーは、FOR EACH ROW の細分性を持たなければなりません。また、サブジェクト表はビューでなければなりません。その他のトリガーはサブジェクト表としてビューを使用することはできません。

以下の図に、BEFORE トリガーと AFTER トリガーの実行モデルを示します。

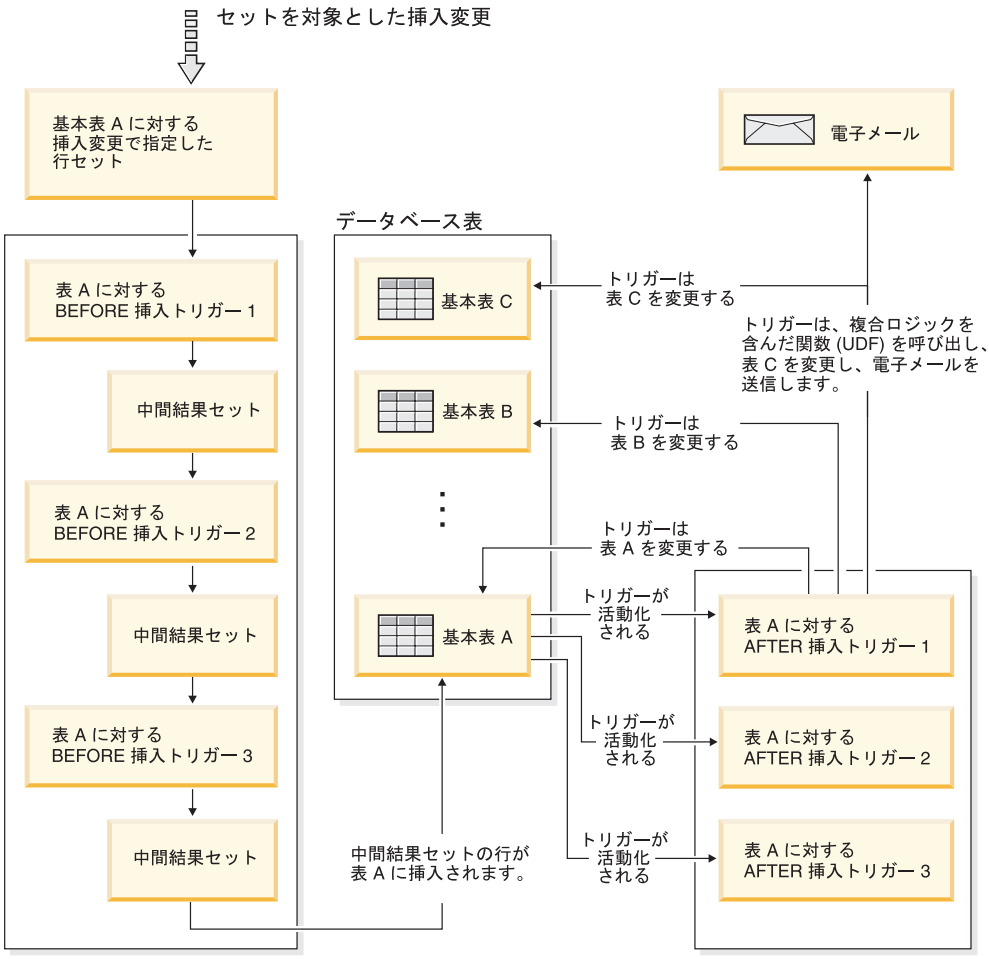


図 17. ルーチンの種別

BEFORE トリガーと AFTER トリガーの両方を持つ特定の表と、これらのトリガーに関連した変更イベントについては、まずすべての BEFORE トリガーが活動化されます。そのイベントについて最初に活動化された BEFORE トリガーは、操作のターゲットになっている行セットに対して、ロジックで指定されているすべての更新操作を実行します。その BEFORE トリガーの出力を次の BEFORE トリガーが入力として受け取ります。そのイベントによって活動化されたすべての BEFORE トリガーが起動されると、中間結果セット、つまり、トリガー・イベント操作のターゲットになっている行に対する BEFORE トリガーの変更の結果が基本表に適用されます。その後、イベントに関連した各 AFTER トリガーが起動されます。AFTER トリガーでは、同じ表を変更することも、別の表を変更することも、データベースの外部のアクションを実行することもできます。

トリガーの活動化時間が異なると、トリガーの目的も異なります。根本的に、BEFORE トリガーはデータベース管理システムの制約付きサブシステムに対する拡張です。したがって、通常は次のような目的で使用します。

- 入力データの検査を行う
- 新しく挿入された行に対して値を自動的に生成する
- 相互参照の目的で他表からの読み取りを行う

BEFORE トリガーはトリガー・イベントがデータベースに適用される前に活動化されるので、これを使用してデータベースをさらに修正することはできません。そのため、BEFORE トリガーの活動化は、保全性に関する制約をチェックした後に行われます。

逆に、AFTER トリガーは、特殊なイベントが起こるたびにデータベースで実行されるアプリケーション・ロジックのモジュールと見なすことができます。 AFTER トリガーは、アプリケーションの一部として常に一定の状態でデータベースを参照します。また、保全性に関する制約が検証された後に実行されます。したがって、このトリガーは主にアプリケーションでも実行できる操作を行うために使用できます。以下に例を示します。

- 結果としてデータベースでの変更操作を生じる操作を行う。
- データベースの外側で、警告のサポートなどのアクションを行う。トリガーがロールバックされても、データベースの外側で行われるアクションはロールバックされないことに注意してください。

対照的に、INSTEAD OF トリガーは、それが定義されているビューの逆の操作の記述として表示することができます。たとえば、ビュー内の選択リストに基本表に関する式が含まれる場合、 INSTEAD OF INSERT トリガーの本体の INSERT ステートメントには逆の式が含まれます。

BEFORE、AFTER、および INSTEAD OF トリガーにはさまざまな特質のものがあるため、それらのトリガー・アクションを定義するためにさまざまな SQL 操作を使用することができます。たとえば、更新操作は BEFORE トリガーでは実行できません。これは、トリガー・アクションにおいて保全性に関する制約が違反されないという保証がないためです。同様に、BEFORE、AFTER、および INSTEAD OF トリガーでは、さまざまなトリガーの細分性がサポートされています。たとえば、FOR EACH STATEMENT は BEFORE トリガーでは実行できません。これは、トリガー・アクションによる制約の違反がなく、それゆえにその操作が順々に失敗するということがないことを保証できないためです。

すべてのトリガーのトリガー SQL ステートメントが、動的コンパウンド・ステートメントであることがあります。しかし、BEFORE トリガーにはいくつかの制約事項があります。このトリガーには以下の SQL ステートメントを含めることはできません。

- UPDATE
- DELETE
- INSERT

関連概念:

- 341 ページの『INSERT、UPDATE、および DELETE トリガー』
- 345 ページの『トリガーの細分性』

- 353 ページの『SQL ステートメントで構成されるトリガー・アクション』
- 337 ページの『アプリケーション開発でのトリガー』
- 343 ページの『トリガー作成のガイドライン』

関連タスク:

- 344 ページの『トリガーの作成』

遷移変数

FOR EACH ROW トリガーをインプリメントする際に、影響される一連の行内の行の列の値で、トリガーで現在実行されている値を参照する必要があります。データベース内の表 (主題表を含む) の列を参照するには、正規の SELECT ステートメントを使用できることに注意してください。 FOR EACH ROW トリガーは現在実行中の行の列を、 CREATE TRIGGER ステートメントの REFERENCING 文節で指定できる 2 つの遷移変数を使用して参照します。遷移変数には、相関名とともに *OLD* および *NEW* として指定される 2 種類があります。この分類には次のようなセマンティクスがあります。

OLD AS 相関名
: 行の元の状態 (つまりトリガー・アクションがデータベースに適用される前の状態) を収集する相関名を指定します。

NEW AS 相関名
: トリガー・アクションがデータベースに適用される際に、データベースの行を更新するために使用される (または使用された) 値を収集する相関名を指定します。

次の例を考えてください。

```
CREATE TRIGGER REORDER
  AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
  REFERENCING NEW AS N_ROW
  FOR EACH ROW
  WHEN (N_ROW.ON_HAND < 0.10 * N_ROW.MAX_STOCKED
  AND N_ROW.ORDER_PENDING = 'N')
  BEGIN ATOMIC
    VALUES(ISSUE_SHIP_REQUEST(N_ROW.MAX_STOCKED -
                              N_ROW.ON_HAND,
                              N_ROW.PARTNO));
    UPDATE PARTS SET PARTS.ORDER_PENDING = 'Y'
    WHERE PARTS.PARTNO = N_ROW.PARTNO;
  END
```

上で説明した遷移変数の OLD および NEW の定義により、すべての遷移変数がすべてのトリガーに対して定義できるわけではないことが分かります。遷移変数は、次のようなトリガー・イベントの種類に基づいて定義することができます。

UPDATE
: UPDATE トリガーは、OLD と NEW の両方の遷移変数を参照できます。

INSERT
: INSERT トリガーは、NEW 遷移変数のみを参照できます。これは INSERT 操作の活動化の前に、影響される行がデータベースに存在しないためです。すなわち、トリガー・アクションがデータベースに適用される前の古い値を定義する行の元の状態がありません。

DELETE

DELETE トリガーは OLD 遷移変数のみを参照できます。これは、削除操作で指定された新しい値がないためです。

注: 遷移変数は FOR EACH ROW トリガーに対してのみ指定できます。 FOR EACH STATEMENT トリガーでは遷移変数を参照しても、影響される行のうち遷移変数が参照中の行を指定することはできません。

関連概念:

- 341 ページの『INSERT、UPDATE、および DELETE トリガー』
- 345 ページの『トリガーの細分性』
- 350 ページの『遷移表』

関連タスク:

- 344 ページの『トリガーの作成』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE TRIGGER ステートメント』

遷移表

FOR EACH ROW と FOR EACH STATEMENT の両方のトリガーでは、影響される行の全セットを参照しなければならないことがあります。これはたとえば、トリガー本体が影響される行のセットを超えた集約を適用する必要がある場合に当てはまります (たとえば列の値の MAX、MIN、または AVG)。トリガーは、影響される行のセットを CREATE TRIGGER ステートメントの REFERENCING 文節で指定できる 2 つの遷移表を使用して参照します。遷移表には、遷移変数のように OLD_TABLE および NEW_TABLE として*表名* とともに指定される 2 種類があります。この 2 種類の遷移表のセマンティクスは次のとおりです。

OLD_TABLE AS 表名

影響される行のセットの元の状態 (トリガー SQL 操作がデータベースに適用される前の状態) を収集する表の名前を指定します。

NEW_TABLE AS 表名

トリガー・アクションがデータベースに適用される際に、データベースの行を更新するために使用される値を収集する表の名前を指定します。

以下に例を示します。

```
CREATE TRIGGER REORDER
  AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
  REFERENCING NEW_TABLE AS N_TABLE
  NEW AS N_ROW
  FOR EACH ROW
  WHEN ((SELECT AVG (ON_HAND) FROM N_TABLE) > 35)
  BEGIN ATOMIC
    VALUES(INFORM_SUPERVISOR(N_ROW.PARTNO,
                             N_ROW.MAX_STOCKED,
                             N_ROW.ON_HAND));
  END
```

NEW_TABLE は、FOR EACH ROW トリガーにおいても、更新された行の全セットを常に持つことに注意してください。トリガーはそれが定義される表上で実行さ

れると、 NEW_TABLE にはそのトリガーを活動化したステートメントから変更された行が入ります。ただし、トリガー内のステートメントによって変更された行は入りません。これはトリガーの活動化を分離させるためです。

遷移表は読み取り専用です。遷移表は、トリガー・イベントに対して定義できる遷移変数の種類を定義するのと同じ、次のような規則で定義できます。

UPDATE

UPDATE トリガーは、 OLD_TABLE と NEW_TABLE の両方の遷移表を参照できます。

INSERT

INSERT トリガーは、NEW_TABLE 遷移表のみを参照できます。これは、INSERT 操作の活動化の前に、影響される行がデータベースに存在しないためです。すなわち、トリガー・アクションがデータベースに適用される前の古い値を定義している*行の元の状態* がありません。

DELETE

DELETE トリガーは、OLD 遷移表のみを参照できます。これは、削除操作で指定された新しい値がないためです。

注: 遷移表が AFTER トリガーの FOR EACH ROW と FOR EACH STATEMENT の両方の細分性に対して指定できることは知っておく必要があります。

OLD_TABLE と NEW_TABLE の*表名* の有効範囲はトリガー本体です。この有効範囲では、表名はスキーマ内にある同一の非修飾*表名* を持つ他のすべての表の名前より優先されます。したがって、たとえば OLD_TABLE や NEW_TABLE の*表名* が X の場合、 SELECT ステートメントの FROM 文節で X (すなわち非修飾の X) を参照することで、トリガー作成者のスキーマ内に X という名の表があったとしても遷移表を常に参照します。この場合、ユーザーはスキーマ内の表 X を参照するために完全修飾名を使わなければなりません。

関連概念:

- 341 ページの『INSERT、UPDATE、および DELETE トリガー』
- 345 ページの『トリガーの細分性』
- 349 ページの『遷移変数』

関連タスク:

- 344 ページの『トリガーの作成』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE TRIGGER ステートメント』

トリガー・アクション

トリガー・アクション

トリガーを活動化すると、それに関連するトリガー・アクションが実行されます。すべてのトリガーには、次のような 2 つのコンポーネントを順々に持つトリガー・アクションが 1 つだけあります。

- オプションの*トリガー・アクション条件* または WHEN 文節

- 一連のトリガー *SQL* ステートメント

トリガー・アクション条件は、トリガー・アクションが実行中の行やステートメントに対してトリガー・ステートメントのセットが実行されるかどうかを定義します。トリガー・ステートメントのセットは、トリガー・イベントが発生した結果としてトリガーによりデータベースで実行される一連のアクションを定義します。

たとえば以下のトリガー・アクションは、 on_hand 列の値が max_stocked 列の値の 10% より小さい行に対してのみ、トリガー SQL ステートメントのセットが活動化されることを指定します。この場合、トリガー SQL ステートメントのセットが issue_ship_request 関数を呼び出します。

```
CREATE TRIGGER REORDER
  AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
  REFERENCING NEW AS N_ROW
  FOR EACH ROW

  WHEN (N_ROW.ON_HAND < 0.10 * N_ROW.MAX_STOCKED)
  BEGIN ATOMIC
    VALUES(ISSUE_SHIP_REQUEST(N_ROW.MAX_STOCKED -
                              N_ROW.ON_HAND,
                              N_ROW.PARTNO));
  END
```

関連概念:

- 352 ページの『条件によって限定されるトリガー・アクション』
- 353 ページの『SQL ステートメントで構成されるトリガー・アクション』
- 353 ページの『プロシージャーまたは関数の参照を含むトリガー・アクション』

条件によって限定されるトリガー・アクション

トリガー・アクション条件 は、探索条件を指定するトリガー・アクションの任意指定の文節です。トリガー・アクション内で SQL ステートメントを実行するためには、探索条件は*真* と評価されなければなりません。 WHEN 文節を省略すると、トリガー・アクション内の SQL ステートメントは常に実行されます。

トリガー・アクション条件は、FOR EACH ROW トリガーの場合にはそれぞれの行に対して一度評価され、 FOR EACH STATEMENT トリガーの場合にはステートメントに対して一度評価されます。

さらに WHEN 文節は、トリガーに代わって活動化されるアクションを正しく調整するために使用できるように制御されます。たとえばこの文節は、入ってくる値がある一定の範囲の内側か外側になる場合だけトリガー・アクションが活動化されるという、データ従属の規則を強調するのに役立ちます。

関連概念:

- 351 ページの『トリガー・アクション』
- 353 ページの『SQL ステートメントで構成されるトリガー・アクション』
- 353 ページの『プロシージャーまたは関数の参照を含むトリガー・アクション』

SQL ステートメントで構成されるトリガー・アクション

トリガー SQL ステートメントのセットは、トリガーの活動化により行われる実際のアクションを実行します。すべての SQL 操作がすべてのトリガーに有効となるわけではありません。トリガー活動化時間が BEFORE か AFTER かにより、異なる種類の操作がトリガー SQL ステートメントに適用されます。

ほとんどの場合、トリガー SQL ステートメントが負の戻りコードを戻すと、トリガー SQL ステートメントはトリガーおよび参照を制約するすべてのアクションとともにロールバックされ、その結果エラー SQLCODE -723 (SQLSTATE 09000) が戻されます。その失敗したトリガー SQL ステートメントからは、トリガー名、SQLCODE、 SQLSTATE、およびトークンの大部分が戻されます。トリガーの実行中に発生し、作業単位全体を否定またはロールバックするエラー条件は、SQLCODE -723 (SQLSTATE 09000) では戻されません。

すべてのトリガーのトリガー SQL ステートメントが、動的コンパウンド・ステートメントであることがあります。すなわち、以下の 1 つ以上が含まれている場合があります。

- DECLARE 変数ステートメント
- SET 変数ステートメント
- WHILE ループ
- FOR ループ
- IF ステートメント
- SIGNAL ステートメント
- ITERATE ステートメント
- LEAVE ステートメント
- GET DIGNOSTIC ステートメント
- 全選択

しかし、トリガーの AFTER と INSTEAD に限り、以下の 1 つ以上を含めることができます。

- UPDATE SQL ステートメント
- DELETE SQL ステートメント
- INSERT SQL ステートメント

関連概念:

- 351 ページの『トリガー・アクション』
- 352 ページの『条件によって限定されるトリガー・アクション』
- 353 ページの『プロシージャーまたは関数の参照を含むトリガー・アクション』

プロシージャーまたは関数の参照を含むトリガー・アクション

トリガーのトリガー・アクション内から、プロシージャーや関数 (ユーザー定義関数 (UDF) を含む) を呼び出せます。プロシージャーを呼び出すには、CALL ステートメントを使用します。関数はどのトリガー SQL ステートメント内からでも呼び出せます。トリガーからルーチンを呼び出すことにより、トリガーに複合ロジック

を含めることができます。 TOTAL_SALES という名前の SQL プロシージャーの呼び出しを含むトリガーの定義を示す以下の例を検討してみましょう。

```
CREATE TRIGGER trig1 AFTER UPDATE ON t1
REFERENCING NEW AS n
FOR EACH ROW MODE DB2SQL
WHEN (n.c1 > 100);
BEGIN ATOMIC
   DECLARE rs INTEGER DEFAULT 0;
   CALL TOTAL_SALES(n.c1, n.c2);
   GET DIANOSTICS rs = RETURN_STATUS;
   VALUES(CASE WHEN rc < 0
               THEN RAISE_ERROR('70001',
                      'PROC CALL failed'));
END;
```

プロシージャーは、トリガーのサブルーチンと見なせます。 SQL プロシージャーが呼び出された後、 GET DIAGNOSTICS ステートメントの実行によりプロシージャーの戻り状況がチェックされます。戻り状況でプロシージャーのエラーが検出されると、エラーになります。

以下に、トリガーの本体の中の関数参照の例を示します。関数参照は VALUES 文節の中にあります。

```
CREATE TRIGGER REORDER
  AFTER UPDATE OF ON_HAND, MAX_STOCKED ON PARTS
  REFERENCING NEW AS N_ROW
  FOR EACH ROW
  WHEN (N_ROW.ON_HAND < 0.10 * N_ROW.MAX_STOCKED)
  BEGIN ATOMIC
    VALUES (ISSUE_SHIP_REQUEST
            (N_ROW.MAX_STOCKED - N_ROW.ON_HAND, N_ROW.PARTNO));
  END
```

関数 ISSUE_SHIP_REQUEST は、配送部門に電子メールを送信して部品のオーダーが必要であることを通知する外部関数として機能します。この関数は、パラメーターとして遷移変数を含む式を取ります。

トリガー・アクションに、非修飾プロシージャー名を持つプロシージャー呼び出しや、非修飾関数名を持つ関数参照を含むトリガー・アクション SQL ステートメントが組み込まれている場合、そのプロシージャーまたは関数は以下に基づいて変換されます。

- トリガー作成時の SQL パス
- トリガーの作成者が持っているルーチンに関する EXECUTE 特権

ルーチンは、SQL、Java™、C、C++、.NET のいずれかの言語で作成できます。 UDF は論理フローの制御、エラー処理とリカバリー、システムおよびライブラリー関数へのアクセスを可能にします。この機能を使うと、トリガー・アクションはトリガーが活動化された際に非 SQL タイプの操作を実行することができます。たとえばトリガーから呼び出される UDF は、電子メールのメッセージを送信し、それにより警告機構として作用することができます。メッセージなどの外部アクションはコミット制御下にはなく、他のトリガー・アクションの成否に関係なく実行されます。

また関数にエラーが発生した結果、トリガー SQL ステートメントが失敗したことを示す SQLSTATE を戻すことがあります。これは、ユーザー定義の制約を実行す

る 1 つの方法です。 (SIGNAL SQLSTATE ステートメントを使用する方法もあります。) 複雑なユーザー定義の制約をチェックする手段としてトリガーを使用するために、 RAISE_ERROR 組み込み関数をトリガー SQL ステートメントで使用することができます。この関数は、ユーザー定義の SQLSTATE (SQLCODE -438) をアプリケーションに戻すことができます。

たとえば、EMPLOYEE 表の HIREDATE 列に関連する次のような規則を考えてください。この場合の HIREDATE は、従業員が作業を始める日付です。

- HIREDATE は、挿入日かそれ以降の日付でなければならない。
- HIREDATE は、挿入日から 1 年以上経過した日付となることはあり得ない。
- HIREDATE が挿入日から 6 ～ 12 か月経過している場合は、 send_note を呼び出す UDF を使用して管理者に知らせる。

以下のトリガーは、このような規則をすべて INSERT で処理します。

```
CREATE TRIGGER CHECK_HIREDATE
  NO CASCADE BEFORE INSERT ON EMPLOYEE
  REFERENCING NEW AS NEW_EMP
  FOR EACH ROW
  BEGIN ATOMIC
  VALUES CASE
    WHEN NEW_EMP.HIREDATE - CURRENT DATE > 600.
      AND NEW_EMP.HIREDATE - CURRENT DATE <eq; 10000.
      THEN SEND_NOTE('persmgr', NEW_EMP.EMPNO, 'late.txt')
    WHEN NEW_EMP.HIREDATE < CURRENT DATE
      THEN RAISE_ERROR('85001', 'HIREDATE has passed')
    WHEN NEW_EMP.HIREDATE - CURRENT DATE > 10000.
      THEN RAISE_ERROR('85002', 'HIREDATE too far out')
    END;
  END
```

関連概念:

- 351 ページの『トリガー・アクション』
- 352 ページの『条件によって限定されるトリガー・アクション』
- 353 ページの『SQL ステートメントで構成されるトリガー・アクション』
- 217 ページの『ルーチンの呼び出し』

関連タスク:

- 239 ページの『ユーザー定義の表関数の呼び出し』
- 227 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ーの呼び出し』

複数のトリガー

CREATE TRIGGER ステートメントを使用してトリガーを定義すると、この作成時間はデータベース内にタイム・スタンプの形式で登録されます。このタイム・スタンプの値は、同時に実行すべきトリガーが複数存在した際に、トリガーの活動化を順序付けするために引き続き使用されます。たとえばタイム・スタンプは、同一対象表に同じイベントと同じ活動化時間を持つトリガーが複数存在する場合に使用されます。また、トリガー・アクションによって直接的または間接的に (これは、他の参照制約によって、反復的に行われることを意味します) 発生したトリガー・イベントおよび参照制約アクションによって活動化された、 1 つ以上の AFTER または INSTEAD OF トリガーが存在する場合にも使用されます。

次の 2 つのトリガーを考えてください。

```
CREATE TRIGGER NEW_HIRED
  AFTER INSERT ON EMPLOYEE
  FOR EACH ROW
  BEGIN ATOMIC
    UPDATE COMPANY_STATS
    SET NBEMP = NBEMP + 1;
  END

CREATE TRIGGER NEW_HIRED_DEPT
   AFTER INSERT ON EMPLOYEE
   REFERENCING NEW AS EMP
   FOR EACH ROW
     BEGIN ATOMIC
       UPDATE DEPTS
       SET NBEMP = NBEMP + 1
       WHERE DEPT_ID = EMP.DEPT_ID;
     END
```

上記のトリガーは、employee 表で INSERT 操作を実行すると活動化されます。この場合、トリガー作成のタイム・スタンプは、上の 2 つのトリガーのうちどちらが最初に活動化されるかを定義します。

トリガーの活動化は、タイム・スタンプ値の昇順で処理されます。したがって、データベースに新しく追加されたトリガーは、事前に定義されている他のすべてのトリガーの後で実行されます。

旧トリガーは新規トリガーの前に活動化され、新規トリガーがデータベースに影響を及ぼす変更に対して*増分の* 加算として使用できるようにします。たとえば、トリガー T1 のトリガー SQL ステートメントが新しい行を表 T に挿入すると、 T1 の後に実行されるトリガー T2 のトリガー SQL ステートメントを使用して、特定の値を持つ T の中の行を更新することができます。作成時の昇順でトリガーを活動化することにより、新規トリガーのアクションが旧トリガーすべての活動化の結果を反映するデータベースで実行されることが保証できます。

関連概念:

- 337 ページの『アプリケーション開発でのトリガー』

関連タスク:

- 357 ページの『トリガーを使用した UDT、UDF、および LOB からの情報の抽出』
- 358 ページの『トリガーを使用した表操作の抑制』
- 358 ページの『トリガーを使用した業務規則の定義』
- 359 ページの『トリガーを使用したアクションの定義』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE TRIGGER ステートメント』

トリガー、制約、ルーチンの協調

トリガーを使用した UDT、UDF、および LOB からの情報の抽出

ELECTRONIC_MAIL 表の MESSAGE 列内に、 LOB 値として完全な電子メール・メッセージを保管するアプリケーションを作成できます。電子メールを操作するには、 SQL ステートメント内でその情報が必要とされるたびにストアード・プロシージャーまたは UDF を使用してメッセージ列から情報を抽出します。

照会は、情報を 1 回抽出してそれを表の列として明確に保管することはしないことに注意してください。これにより、ストアード・プロシージャーまたは UDF が繰り返し呼び出されることがないだけでなく、抽出した情報に索引を定義できるため、照会のパフォーマンスは向上します。

トリガーを使うと、新しい電子メールがデータベースに保管されるたびにこの情報を抽出することができます。これを行うには、BEFORE トリガーを定義し、該当する情報を次のように抽出してください。

```
CREATE TRIGGER EXTRACT_INFO
  NO CASCADE BEFORE INSERT ON ELECTRONIC_MAIL
  REFERENCING NEW AS N
  FOR EACH ROW
  BEGIN ATOMIC
    SET (N.SENDER, N.RECEIVER, N.SENT_ON, N.SUBJECT)
      = (SELECT SENDER, RECEIVER, SENT_ON, SUBJECT FROM
         TABLE(EMAIL_HEADER(N.MESSAGE)) AS H)
  END
```

また、これは生成された列を ELECTRONIC_MAIL 表に追加することによって、行うこともできます。

```
ALTER TABLE ELECTRONIC_MAIL
  ADD COLUMN SENDER VARCHAR(200) GENERATED ALWAYS
    AS (SENDER(N.MESSAGE))
  ADD COLUMN RECEIVER VARCHAR(200) GENERATED ALWAYS
    AS (RECEIVER(N.MESSAGE))
  ADD COLUMN SENT_ON DATE GENERATED ALWAYS
    AS (SENDING_DATE(N.MESSAGE))
  ADD COLUMN SUBJECT VARCHAR(200) GENERATED ALWAYS
    AS (SUBJECT(N.MESSAGE))
```

このようにして、新しい電子メールがメッセージ列に挿入されると常に、その差出人、宛先、送信日、および主題がメッセージから抽出され、別々の列に保管されます。

関連概念:

- 351 ページの『トリガー・アクション』
- 352 ページの『条件によって限定されるトリガー・アクション』
- 353 ページの『SQL ステートメントで構成されるトリガー・アクション』
- 353 ページの『プロシージャーまたは関数の参照を含むトリガー・アクション』
- 355 ページの『複数のトリガー』

関連タスク:

- 358 ページの『トリガーを使用した表操作の抑制』

- 358 ページの『トリガーを使用した業務規則の定義』
- 359 ページの『トリガーを使用したアクションの定義』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE TRIGGER ステートメント』

トリガーを使用した表操作の抑制

送信した結果未送信だったメールと戻ってきたメール (おそらく電子メールのアドレスが誤っていたため) が、電子メールの表に保管されないようにしたいと仮定します。

そのようにするには、特定の SQL INSERT ステートメントを実行しないようにする必要があります。それには次の 2 とおりの方法があります。

- 電子メールの対象が*未送信だったメール* のときは必ずエラーとなる BEFORE トリガーを定義する。

```
CREATE TRIGGER BLOCK_INSERT
  NO CASCADE BEFORE INSERT ON ELECTRONIC_MAIL
  REFERENCING NEW AS N
  FOR EACH ROW
  WHEN (SUBJECT(N.MESSAGE) = 'undelivered mail')
  BEGIN ATOMIC
    SIGNAL SQLSTATE '85101'
      SET MESSAGE_TEXT = ('Attempt to insert undelivered mail');
    END
```

- 新しい列対象の値を*未送信だったメール* と異なるものにするチェックの制約を定義する。

```
ALTER TABLE ELECTRONIC_MAIL
  ADD CONSTRAINT NO_UNDELIVERED
  CHECK (SUBJECT <> 'undelivered mail')
```

制約の宣言上の特質の利点のため、制約は通常トリガーの代わりに定義されます。

関連概念:

- 355 ページの『複数のトリガー』
- 337 ページの『アプリケーション開発でのトリガー』

関連タスク:

- 357 ページの『トリガーを使用した UDT、UDF、および LOB からの情報の抽出』
- 358 ページの『トリガーを使用した業務規則の定義』
- 359 ページの『トリガーを使用したアクションの定義』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE TRIGGER ステートメント』

トリガーを使用した業務規則の定義

お客様の苦情を扱う電子メールは、マーケティング管理者の Mr. Nelson にカーボン・コピー (CC) のリストで提出しなければならないという方針が会社にあるとし

ます。これは規則であるため、制約として明記するほうがよいかもしれません (この場合は、これをチェックする CC_LIST UDF の存在が前提になる)。以下のような方法があります。

```
ALTER TABLE ELECTRONIC_MAIL ADD
  CHECK (SUBJECT <> 'Customer complaint' OR
         CONTAINS (CC_LIST(MESSAGE), 'nelson@vnet.ibm.com') = 1)
```

ただしこの制約により、マーケティング管理者に CC リストで提出しないお客様の苦情を扱う電子メールは挿入できなくなります。このことは、この会社の業務規則の目的ではないことは明らかです。その目的とは、マーケティング管理者にはコピーされていないお客様の苦情を扱う電子メールをマーケティング管理者に転送することです。このような業務規則は、宣言上の制約により表すことのできないアクションを行うことを要求するので、トリガーを使用してのみ表すことができます。トリガーは、E_MAIL タイプのパラメーターと文字ストリングを持つ SEND_NOTE 関数があると仮定します。

```
CREATE TRIGGER INFORM_MANAGER
  AFTER INSERT ON ELECTRONIC_MAIL
  REFERENCING NEW AS N
  FOR EACH ROW
  WHEN (N.SUBJECT = 'Customer complaint' AND
    CONTAINS (CC_LIST(MESSAGE), 'nelson@vnet.ibm.com') = 0)
  BEGIN ATOMIC
    VALUES(SEND_NOTE(N.MESSAGE, 'nelson@vnet.ibm.com'));
  END
```

関連概念:

- 355 ページの『複数のトリガー』
- 337 ページの『アプリケーション開発でのトリガー』

関連タスク:

- 357 ページの『トリガーを使用した UDT、UDF、および LOB からの情報の抽出』
- 358 ページの『トリガーを使用した表操作の抑制』
- 359 ページの『トリガーを使用したアクションの定義』

トリガーを使用したアクションの定義

総管理者が、72 時間以内に別々の表に 3 つ以上の苦情を送ってきたカスタマーの名前を保持したいとします。また、顧客名がこの表に複数回挿入されたら必ず総管理者に知らせるようにしたいと仮定します。

このようなアクションを定義するには、次のように定義します。

- UNHAPPY_CUSTOMERS table:

  ```
  CREATE TABLE UNHAPPY_CUSTOMERS (
    NAME           VARCHAR (30),
    EMAIL_ADDRESS  VARCHAR (200),
    INSERTION_DATE DATE)
  ```

- 3 日以内に 3 つ以上のメッセージを受信した場合に、 UNHAPPY_CUSTOMERS 内に行を自動的に挿入するトリガー (NAME 列と E_MAIL_ADDRESS 列を含む CUSTOMERS 表があることを前提とする)。

```
CREATE TRIGGER STORE_UNHAPPY_CUST
  AFTER INSERT ON ELECTRONIC_MAIL
  REFERENCING NEW AS N
  FOR EACH ROW MODE DB2SQL
  WHEN (3 <= (SELECT COUNT(*)
              FROM ELECTRONIC_MAIL
              WHERE SENDER = N.SENDER
                AND SENDING_DATE(MESSAGE) > CURRENT DATE - 3 DAYS)
       )
  BEGIN ATOMIC
    INSERT INTO UNHAPPY_CUSTOMERS
    VALUES ((SELECT NAME
    FROM CUSTOMERS
    WHERE E_MAIL_ADDRESS = N.SENDER), N.SENDER, CURRENT DATE);
  END
```

- 同じカスタマーが複数回 UNHAPPY_CUSTOMERS に挿入された場合に総管理者に通知を送るトリガー (2 文字のストリングを入力とする SEND_NOTE 関数があることを前提とする)。

```
CREATE TRIGGER INFORM_GEN_MGR
  AFTER INSERT ON UNHAPPY_CUSTOMERS
  REFERENCING NEW AS N
  FOR EACH ROW
  WHEN (1 <(SELECT COUNT(*)
            FROM UNHAPPY_CUSTOMERS
            WHERE EMAIL_ADDRESS = N.EMAIL_ADDRESS)
       )
  BEGIN ATOMIC
    VALUES(SEND_NOTE('Check customer:' CONCAT N.NAME,
                     'bigboss@vnet.ibm.com'));
  END
```

関連概念:

- 355 ページの『複数のトリガー』
- 337 ページの『アプリケーション開発でのトリガー』

関連タスク:

- 357 ページの『トリガーを使用した UDT、UDF、および LOB からの情報の抽出』
- 358 ページの『トリガーを使用した表操作の抑制』
- 358 ページの『トリガーを使用した業務規則の定義』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『CREATE TRIGGER ステートメント』

第 3 部 付録

付録 A. DB2GENERAL ルーチン

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・プロシージャーから JDBC API を呼び出すことはできますが、 UDF からそれらを呼び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 文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機能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。

上記の機能のどれも使用しない PARAMETER STYLE DB2GENERAL ルーチンの場合、移植できるようにそれを PARAMETER STYLE JAVA に移行することをお勧めします。

関連概念:

- 364 ページの『DB2GENERAL UDF』
- 189 ページの『Java ルーチン』
- 68 ページの『Java の表関数実行モデル』

関連資料:

- 202 ページの『Java デバッグ表 DB2DBG.ROUTINE_DEBUG』
- 196 ページの『データベース・サーバーでの JAR ファイル管理』
- 366 ページの『DB2GENERAL ルーチンでサポートされている SQL データ型』
- 368 ページの『DB2GENERAL ルーチン用の Java クラス』
- 369 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.StoredProc』
- 370 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.UDF』

- 373 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.Lob』
- 373 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.Blob』
- 374 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.Clob』

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 タイプの引き数 *t1*、*t2*、および *t3*、戻りタイプの引き数 *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* ～ *T4* は、 SQL タイプの *t1* ～ *t4* に対応する Java タイプ
- *a*、*b*、および *c* は、入力引き数のための変数名
- *d* は、出力引き数を表す変数名

たとえば、INTEGER を戻し、CHAR(5)、BLOB(10K)、および DATE タイプの引き数を取る sample!test3 という UDF があるとすると、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、さらに多くの引き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。たとえば、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、どんな入力引き数でも関数 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、引き数番号は 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' は出力引き数の索引で、value は互換タイプのリテラルまたは変数です。引き数の番号は、選択された出力の引き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 変数は 4 の索引を持っています。 2 番目の例の result1 ～ result3 は 2 ～ 4 の索引をもっています。

UDF とストアード・プロシージャーで使用される 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・マネージャーの 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 文節を指定する場合、 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、別の区画に別個の 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、遅くし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。

関連概念:

- 363 ページの『DB2GENERAL ルーチン』
- 189 ページの『Java ルーチン』
- 68 ページの『Java の表関数実行モデル』

関連資料:

- 202 ページの『Java デバッグ表 DB2DBG.ROUTINE_DEBUG』
- 366 ページの『DB2GENERAL ルーチンでサポートされている SQL データ型』
- 368 ページの『DB2GENERAL ルーチン用の Java クラス』
- 369 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.StoredProc』
- 370 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.UDF』
- 373 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.Lob』
- 373 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.Blob』
- 374 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.Clob』

関連サンプル:

- 『UDFsqlsv.java -- Provide UDFs to be called by UDFsqlcl.java (JDBC)』
- 『UDFsrv.java -- Provide UDFs to be called by UDFcli.java (JDBC)』
- 『UDFsrv.java -- Provide UDFs to be called by UDFcli.sqlj (SQLj)』

DB2GENERAL ルーチンでサポートされている SQL データ型

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 37. DB2 SQL タイプおよび Java オブジェクト

SQL 列名	Java データ型
SMALLINT	short

表 37. DB2 SQL タイプおよび Java オブジェクト (続き)

SQL 列名	Java データ型
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 データ型 (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。

関連概念:

- 363 ページの『DB2GENERAL ルーチン』
- 364 ページの『DB2GENERAL UDF』
- 189 ページの『Java ルーチン』
- 68 ページの『Java の表関数実行モデル』

関連資料:

- 193 ページの『Java でサポートされている SQL データ型』
- 368 ページの『DB2GENERAL ルーチン用の Java クラス』
- 369 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.StoredProc』
- 370 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.UDF』
- 373 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.Lob』
- 373 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.Blob』
- 374 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.Clob』

DB2GENERAL ルーチン用の Java クラス

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 ストアード・プロシージャーまたは 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

関連概念:

- 363 ページの『DB2GENERAL ルーチン』
- 364 ページの『DB2GENERAL UDF』
- 189 ページの『Java ルーチン』

関連資料:

- 366 ページの『DB2GENERAL ルーチンでサポートされている SQL データ型』
- 369 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.StoredProc』
- 370 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.UDF』
- 373 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.Lob』

- 373 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.Blob』
- 374 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.Clob』

DB2GENERAL Java クラス: COM.IBM.db2.app.StoredProc

PARAMETER STYLE DB2GENERAL ストアード・プロシージャー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ード・プロシージャー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す。たとえば、ストアード・プロシージャーが戻った後には、 LOB 引き数に対する操作 (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。

引き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、最初の引き数の 1 から開始します。 PARAMETER STYLE DB2GENERAL ストアード・プロシージャーのすべての引き数は INOUT、つまり入出力であると見なされます。

ストアード・プロシージャー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き数が 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き数を所定の値に設定します。この索引は有効な出力引き数を参照し、データ型は一致し、値は有効な長さと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。

関連概念:

- 363 ページの『DB2GENERAL ルーチン』
- 364 ページの『DB2GENERAL UDF』
- 189 ページの『Java ルーチン』

関連資料:

- 366 ページの『DB2GENERAL ルーチンでサポートされている SQL データ型』
- 368 ページの『DB2GENERAL ルーチン用の Java クラス』
- 370 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.UDF』
- 373 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.Lob』
- 373 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.Blob』
- 374 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.Clob』

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。たとえば、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 引き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。

引き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、最初の引き数の 1 から開始します。出力引き数は、入力引き数よりも大きな番号が付けられます。たとえば、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または無視します。

```
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き数が 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き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き数を所定の値に設定します。この索引は有効な出力引き数を参照し、データ型は一致し、値は有効な長さと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 (たとえば、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 の名前を戻します。

```
public String getSpecificName() throws Exception
```

この関数は、実行中の UDF の特定名を戻します。

```
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戻します。まず、DBINFO 構造をバイト配列で戻すことを 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戻します。

```
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ーのルーチン ID を戻します。そのルーチン ID は、呼び出し元プロシージャー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 を戻します。

```
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・ページ番号を戻します。戻された整数の配列には、最初の 3 つのエレメント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戻します。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()` が戻すサイズと同じでなければなりません。

関連概念:

- 363 ページの『DB2GENERAL ルーチン』
- 364 ページの『DB2GENERAL UDF』
- 189 ページの『Java ルーチン』

関連資料:

- 366 ページの『DB2GENERAL ルーチンでサポートされている SQL データ型』
- 368 ページの『DB2GENERAL ルーチン用の Java クラス』
- 369 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.StoredProc』

- 373 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.Lob』
- 373 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.Blob』
- 374 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.Clob』

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。

関連概念:

- 363 ページの『DB2GENERAL ルーチン』
- 364 ページの『DB2GENERAL UDF』
- 189 ページの『Java ルーチン』

関連資料:

- 366 ページの『DB2GENERAL ルーチンでサポートされている SQL データ型』
- 368 ページの『DB2GENERAL ルーチン用の Java クラス』
- 369 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.StoredProc』
- 370 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.UDF』
- 373 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.Blob』
- 374 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.Clob』

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すが、実行中のルーチンの目的に追随してのみ作成できます。追随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。

関連概念:

- 363 ページの『DB2GENERAL ルーチン』
- 364 ページの『DB2GENERAL UDF』
- 189 ページの『Java ルーチン』

関連資料:

- 366 ページの『DB2GENERAL ルーチンでサポートされている SQL データ型』
- 368 ページの『DB2GENERAL ルーチン用の Java クラス』
- 369 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.StoredProc』
- 370 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.UDF』
- 373 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.Lob』
- 374 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.Clob』

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すが、実行中のルーチンの目的に追随してのみ作成できます。追随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。

関連概念:

- 363 ページの『DB2GENERAL ルーチン』

- 364 ページの『DB2GENERAL UDF』
- 189 ページの『Java ルーチン』

関連資料:

- 366 ページの『DB2GENERAL ルーチンでサポートされている SQL データ型』
- 368 ページの『DB2GENERAL ルーチン用の Java クラス』
- 369 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.StoredProc』
- 370 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.UDF』
- 373 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.Lob』
- 373 ページの『DB2GENERAL Java クラス: COM.IBM.db2.app.Blob』

付録 B. COBOL プロシージャー

COBOL プロシージャー

COBOL プロシージャーは、 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 プロシージャー内でのパラメーターの処理

プロシージャー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。たとえば、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ーから来ています。

```
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 データ型が正しく SQL データ型にマップされるようにします。 SQL と COBOL の間のデータ型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 データ型』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 プロシージャーの終了

プロシージャー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、プロシージャーはクライアント・アプリケーションに正しく戻ります。プロシージャーがローカルの 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 プロシージャーを構築する場合、使用するオペレーティング・システムおよびコンパイラー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 ディレクトリーにあります。 IBM® COBOL 用のビルド・スクリプトは、 sqllib/samples/cobol ディレクトリーにあります。

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 のプロシージャーの例を示します。

```
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 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あることが前提になっています。

注: COBOL プロシージャーを 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 文節でストアード・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。プロシージャー本体を識別するのに絶対パス ID を使用する場合、 .dll 拡張子を付加する必要があります。たとえば、次のようになります。

```
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.dll'
```

関連概念:

- 13 ページの『プロシージャー』
- 「*アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」の『COBOL における組み込み SQL』
- 「*アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」の『COBOL のホスト変数』

関連タスク:

- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『Building IBM COBOL routines on AIX』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『UNIX Micro Focus COBOL ルーチンの構築』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『Windows での IBM COBOL ルーチンの構築』
- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『Windows での Micro Focus COBOL ルーチンの構築』

関連資料:

- 380 ページの『COBOL でサポートされている SQL データ型』
- 「*SQL リファレンス 第 2 巻*」の『CREATE PROCEDURE (外部) ステートメント』
- 101 ページの『C/C++、OLE、COBOL で書かれたルーチンに引き数を渡すときの構文』

COBOL でサポートされている SQL データ型

一部の事前定義 COBOL データ型は、列タイプと一致します。ホスト変数として宣言できるのは、その種の COBOL データ型だけです。

以下の表に、各列タイプに対応する COBOL データ型を示します。プリコンパイラーはホスト変数宣言を検出すると、対応する SQL タイプの値を判別します。データベース・マネージャーはその値に基づいて、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。

注: どの DB2 ホスト言語にも、 DATALINK データ型に対するホスト変数サポートはありません。

表 38. COBOL 宣言にマップされる SQL データ型

SQL 列タイプ[1]	COBOL データ型	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
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(*p*,*s*) (484 または 485)	01 name PIC S9(*m*)V9(*n*) 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(*n*) (452 または 453)	01 name PIC X(*n*)	固定長文字ストリング
VARCHAR(*n*) (448 または 449)	01 name 49 length PIC S9(4) COMP-5 49 name PIC X(*n*) 1<=*n*<=32 672	可変長文字ストリング
LONG VARCHAR (456 または 457)	01 name 49 length PIC S9(4) COMP-5 49 data PIC X(*n*) 32 673<=*n*<=32 700	long 可変長文字ストリング
CLOB(*n*) (408 または 409)	01 MY-CLOB USAGE IS SQL TYPE IS CLOB(n) 1<=*n*<=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	サーバーにある CLOB エンティティーを識別する
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

表 38. COBOL 宣言にマップされる SQL データ型 (続き)

SQL 列タイプ[1]	COBOL データ型	SQL 列タイプ記述
BLOB(*n*) (404 または 405)	01 MY-BLOB USAGE IS SQL TYPE IS BLOB(n) 1<=*n*<=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	サーバーにある BLOB エンティティーを識別する
BLOB ファイル参照変数[4] (916 または 917)	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
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 (392 または 393)	01 identifier PIC X(26)	26 バイトの文字ストリング
注: 以下のデータ型は、DBCS 環境のみで使用できる。		
GRAPHIC(*n*) (468 または 469)	01 name PIC G(*n*) DISPLAY-1	固定長 2 バイト文字ストリング
VARGRAPHIC(*n*) (464 または 465)	01 name 49 length PIC S9(4) COMP-5 49 name PIC G(*n*) DISPLAY-1 1<=*n*<=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 (472 または 473)	01 name 49 length PIC S9(4) COMP-5 49 name PIC G(*n*) DISPLAY-1 16 337<=*n*<=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(*n*) (412 または 413)	01 MY-DBCLOB USAGE IS SQL TYPE IS DBCLOB(n) 1<=*n*<=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 ロケーター変数[4] (968 または 969)	01 MY-DBCLOB-LOCATOR USAGE IS SQL TYPE IS DBCLOB-LOCATOR	サーバーにある DBCLOB エンティティーを識別する
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、それぞれのデータ型の SQLDA の SQLTYPE フィールドに現れます。
2. FLOAT(*n*) (0 < *n* < 25) は、REAL と同義です。 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)
   - DOUBLE PRECISION
4. これは列タイプではなく、ホスト変数タイプです。

サポートされる COBOL データ型については、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。

関連概念:

- 「*アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」の『COBOL 用のホスト変数がある SQL 宣言セクション』

付録 C. DB2 Universal Database の技術情報

DB2 資料およびヘルプ

DB2 技術情報を入手するには、以下のツールや方法を使用します。

- DB2 インフォメーション・センター
  - 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
- ダウンロード可能な PDF ファイル、CD 上の PDF ファイル、印刷資料
  - ガイド
  - リファレンス・マニュアル
- コマンド行ヘルプ
  - コマンド・ヘルプ
  - メッセージ・ヘルプ
  - SQL 状態ヘルプ
- インストール済みのソース・コード
  - サンプル・プログラム

技術ノート、白書、レッドブックなど、 DB2 Universal Database のその他の技術情報についても、 ibm.com からオンラインでアクセスできます。 DB2 情報管理ライブラリーのサイト (www.ibm.com/software/data/db2/udb/support.html) をご覧ください。

DB2 資料の更新情報

IBM では、 DB2 インフォメーション・センターの資料フィックスパックや更新情報を周期的に提供しています。 http://publib.boulder.ibm.com/infocenter/db2help/ で DB2 インフォメーション・センターにアクセスすれば、常に最新の情報を確認できます。 DB2 インフォメーション・センターをローカル・インストールしている場合に更新記事を表示するには、まず手動で更新をインストールしてください。新しい情報が提供された時点で資料の更新情報を利用すれば、 *DB2 インフォメーション・センター CD* からインストールした情報を更新していけます。

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ています。 DB2 の最新の技術情報を入手するには、資料の更新情報が提供された時点で更新をインストールするか、 www.ibm.com サイトの DB2 インフォメーション・センターをご覧ください。

関連概念:

- 「*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 サンプル・プログラム』

- 「*アプリケーション開発ガイド アプリケーションの構築および実行*」の『Java サンプル・プログラム』
- 384 ページの『DB2 インフォメーション・センター』

関連タスク:

- 405 ページの『DB2 ツールからコンテキスト・ヘルプを呼び出す』
- 395 ページの『コンピューターまたはイントラネット・サーバーへの DB2 インフォメーション・センターの更新インストール』
- 406 ページの『コマンド行プロセッサーからメッセージ・ヘルプを呼び出す』
- 406 ページの『コマンド行プロセッサーからコマンド・ヘルプを呼び出す』
- 407 ページの『コマンド行プロセッサーから SQL 状態ヘルプを呼び出す』

関連資料:

- 397 ページの『DB2 PDF 資料および印刷された資料』

DB2 インフォメーション・センター

DB2® インフォメーション・センターを使用すると、 DB2 Universal Database™、 DB2 Connect™、 DB2 Information Integrator および DB2 Query Patroller™ などの DB2 ファミリー製品を最大限に活用するのに必要なすべての情報にアクセスできます。また、DB2 インフォメーション・センターは、 DB2 の主な機能とコンポーネントに関する情報を提供します (レプリケーション、データウェアハウジング、および DB2 の種々の Extender など)。

Mozilla 1.0 以上または Microsoft® Internet Explorer 5.5 以上で表示する場合、 DB2 インフォメーション・センターには以下の機能があります。以下のいくつかの機能では、 JavaScript™ のサポートを使用可能にする必要があります:

柔軟なインストール・オプション
以下の中から、 ご使用の環境に最も適したオプションを使って DB2 資料を表示できます。

- 最新の資料を常に自動的に利用できるようにするには、 IBM® の Web サイト (http://publib.boulder.ibm.com/infocenter/db2help/) にある DB2 インフォメーション・センターからすべての資料に直接アクセスします。
- 更新処理を最小化し、イントラネット内のネットワーク・トラフィックだけに制限するには、イントラネット上の 1 つのサーバーに DB2 資料をインストールします。
- 柔軟性を改善し、ネットワーク接続への依存を軽減するには、個々のコンピューターに DB2 資料をインストールします。

検索 「**検索**」テキスト・フィールドに検索語を入力することにより、 DB2 インフォメーション・センターのすべてのトピックを検索できます。複数の語句を引用符で囲めば、完全一致を検索できます。また、ワイルドカード演算子 (*、?) とブール演算子 (AND、 NOT、OR) を使用して検索を絞り込むことができます。

タスク指向の目次
単一の目次の中から、DB2 資料のトピックを見付けることができます。目

次は、主に実行するタスクの種類に従って編成されていますが、そのほかに製品概要、特定のゴール (目的) の情報、参照情報、索引、および用語集も含まれます。

- 製品概要では、DB2 ファミリーで使用可能な製品間の関係、そうした各製品で提供される機能、および各製品の最新リリース情報について説明されています。
- インストール、管理および開発などのゴール・カテゴリーには、タスクを迅速に完了し、そのための背景情報をよく理解できるようにするトピックが含まれています。
- 「参照」トピックでは、その対象に関する詳細な情報 (ステートメントとコマンドの構文、メッセージ・ヘルプ、構成パラメーターなど) が説明されています。

現在のトピックを目次に表示する

現在のトピックが目次のどの部分に該当するかを表示するには、目次フレーム内の**「リフレッシュ/現在のトピックの表示 (Refresh/Show Current Topic)」**ボタンをクリックするか、コンテンツ・フレーム内の**「目次に表示 (Show in Table of Contents)」**ボタンをクリックします。幾つかのファイルで関連トピックへの複数のリンクをたどった場合、または検索結果からトピックにアクセスした場合には、この機能が役立ちます。

索引 索引から、すべての資料にアクセスすることができます。索引では、用語が 50 音順に編成されています。

用語集 用語集を見れば、DB2 資料で使われているさまざまな用語の定義を調べることができます。用語集では、用語が 50 音順に編成されています。

組み込まれているローカライズ情報

DB2 インフォメーション・センターは、ブラウザーで設定された言語でトピックを表示します。設定された言語のトピックが利用できない場合、DB2 インフォメーション・センターにはそのトピックの英語版が表示されます。

iSeries™ 技術情報については、 IBM eServer™ iSeries Information Center (www.ibm.com/eserver/iseries/infocenter/) を参照してください。

関連概念:

- 386 ページの『DB2 インフォメーション・センターのインストール・シナリオ』

関連タスク:

- 395 ページの『コンピューターまたはイントラネット・サーバーへの DB2 インフォメーション・センターの更新インストール』
- 396 ページの『目的の言語による DB2 インフォメーション・センター・トピックの表示』
- 394 ページの『DB2 インフォメーション・センターの呼び出し』
- 389 ページの『DB2 セットアップ・ウィザードを使用した DB2 インフォメーション・センターのインストール (UNIX)』
- 391 ページの『DB2 セットアップ・ウィザードを使用した DB2 インフォメーション・センターのインストール (Windows)』

DB2 インフォメーション・センターのインストール・シナリオ

作業環境が違えば、 DB2 製品資料にアクセスする方法も違ってきます。 DB2 製品資料には、IBM Web サイト、イントラネット・サーバー、自分のコンピューターにインストールされているバージョン、という 3 つの方法でアクセスできます。この 3 つのケースのいずれでも、資料は DB2 インフォメーション・センター内に置かれています。このセンターは、ブラウザーを使用して表示できるようにアーキテクチャー化されたトピック・ベースの情報の Web サイトです。デフォルトでは、DB2 製品は、IBM Web サイト上の DB2 インフォメーション・センターにアクセスします。これに対して、イントラネット・サーバー上または自分のコンピューター上の DB2 インフォメーション・センターにアクセスする場合は、製品メディア・パック内にある DB2 インフォメーション・センター CD を使用して DB2 インフォメーション・センターをインストールする必要があります。以下の 3 つのインストール・シナリオを参考にしながら、それぞれの作業環境で DB2 インフォメーション・センターにアクセスするための最適な方法を見定め、どのようなインストール上の問題に配慮する必要があるかを検討してください。

シナリオ: IBM Web サイト上の DB2 インフォメーション・センターへのアクセス:

Colin は、セミナー企業に所属する情報技術コンサルタントです。データベース・テクノロジーと SQL が専門で、 DB2 Universal Database Express Edition を使用して北米一帯の企業を対象にこれらの分野のセミナーを開催しています。セミナーでは、教材として DB2 資料も使用します。たとえば、SQL の講習コースでは、データベース照会の基本構文と拡張構文を教えるために SQL に関する DB2 資料を使用します。

受講する企業のほとんどは、インターネットにアクセスできます。このような状況から判断して、Colin は、自分のモバイル・コンピューターに最新バージョンの DB2 Universal Database Express Edition をインストールしたときに、 IBM Web サイト上の DB2 インフォメーション・センターにアクセスするための設定を行いました。この設定によって、Colin はセミナーの開催時に、最新の DB2 資料にオンラインでアクセスできるようになります。

ところが、旅行の先々でインターネットにアクセスできない状況が発生します。特にセミナーの準備のために DB2 資料にアクセスしなければならないときは、このような状況が問題になります。このような事態を回避するために、 Colin は自分のモバイル・コンピューターに DB2 インフォメーション・センターのコピーをインストールしました。

Colin は、DB2 資料のコピーをいつでも自由に活用できます。 **db2set** コマンドを使用すれば、自分のモバイル・コンピューター上で簡単にレジストリー変数を設定できるので、どこにいるかに応じて、 IBM Web サイト上の DB2 インフォメーション・センターにアクセスするか、自分のモバイル・コンピューター上の DB2 インフォメーション・センターにアクセスするかを変更できます。

シナリオ: イントラネット・サーバー上の DB2 インフォメーション・センターへのアクセス:

Eva は、生命保険会社の上級データベース管理者として働いています。会社の UNIX データベース・サーバー上で最新バージョンの DB2 Universal Database Enterprise Server Edition のインストールや構成などの管理業務を担当しています。この会社は最近、セキュリティー上の理由から、就業中のインターネット・アクセスを禁止することを社員に通知しました。この会社にはネットワーク環境があるので、 Eva は DB2 インフォメーション・センターのコピーをイントラネット・サーバー上にインストールして、社内のデータウェアハウスを定期的に利用するすべての社員 (営業担当者、営業部長、業務分析担当者) が DB2 資料にアクセスできるようにしました。

イントラネット・サーバー上に DB2 インフォメーション・センターをインストールするときに、 DB2 セットアップ・ウィザードから、 DB2 インフォメーション・センターがネットワーク上の他のコンピューターからの着信通信を受け取るために使用するポートを指定するための画面が表示されます。この画面で、 DB2 インフォメーション・センターをインストールするイントラネット・サーバーのサービス名とポート番号を指定します。

Eva はさらに、データベース・チームに対して、応答ファイルを使用して全社員のコンピューター上に最新バージョンの DB2 をインストールし、イントラネット・サーバーのホスト名とポート番号を使用して DB2 インフォメーション・センターにアクセスするための設定を各コンピューターで行うように指示します。

ところが、Eva のチームの下級データベース管理者である Migual はその指示を誤解し、数人の社員のコンピューター上で、イントラネット・サーバー上の DB2 インフォメーション・センターにアクセスするように DB2 Universal Database を設定する代わりに、 DB2 インフォメーション・センターのコピーをインストールしてしまいました。この状況に対応するために、Eva は Migual に対して、 **db2set** コマンドを使用してそれらのコンピューター上の DB2 インフォメーション・センターのレジストリー変数 (ホスト名の DB2_DOCHOST、ポート番号の DB2_DOCPORT) を変更するように指示しました。これで、ネットワーク上のすべてのコンピューターが DB2 インフォメーション・センターにアクセスでき、社員が DB2 資料の中から DB2 に関する疑問点を解決できるようになりました。

シナリオ: 自分のコンピューター上の DB2 インフォメーション・センターへのアクセス:

Tsu-Chen は小さな町で工場を経営しています。その町には、インターネット・アクセスを提供するローカル ISP がありません。この人は、在庫、製品オーダー、銀行口座情報、営業経費を管理するために DB2 Universal Database Personal Edition を購入しました。以前に DB2 製品を使用したことがないので、 DB2 製品資料で使用方法を調べる必要があります。

標準インストール・オプションを選択して DB2 Universal Database Personal Edition を自分のコンピューターにインストールした後、 DB2 資料にアクセスしようとします。ところが、開こうとしたページが見つからないという趣旨のエラー・メッセージがブラウザーから表示されます。「*DB2 Universal Database Personal Edition 概説およびインストール*」を調べたところ、自分のコンピューター上で DB2 資料にアクセスするには、 DB2 インフォメーション・センターをインストールしなけれ

ばならないと書いてあります。そこで、メディア・パックの中にあった *DB2* インフォメーション・センター *CD* を見つけて、インフォメーション・センターをインストールします。

オペレーティング・システムのアプリケーション・ランチャーから DB2 インフォメーション・センターにアクセスできるので、 DB2 製品の使用方法を調べて、事業の成功に役立てることができるようになりました。

DB2 資料にアクセスするための方法のサマリー:

それぞれの環境で DB2 インフォメーション・センター内の DB2 製品資料にアクセスするための最適なオプションについての推奨事項を以下の表にまとめます。

インターネット・アクセス	イントラネット・アクセス	推奨
あり	あり	IBM Web サイト上の DB2 インフォメーション・センターへのアクセス、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・センターへのアクセス
あり	なし	IBM Web サイト上の DB2 インフォメーション・センターへのアクセス
なし	あり	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・センターへのアクセス
なし	なし	ローカル・コンピューター上の DB2 インフォメーション・センターへのアクセス

関連概念:

- 384 ページの『DB2 インフォメーション・センター』

関連タスク:

- 395 ページの『コンピューターまたはイントラネット・サーバーへの DB2 インフォメーション・センターの更新インストール』
- 389 ページの『DB2 セットアップ・ウィザードを使用した DB2 インフォメーション・センターのインストール (UNIX)』
- 391 ページの『DB2 セットアップ・ウィザードを使用した DB2 インフォメーション・センターのインストール (Windows)』

関連資料:

- 「コマンド・リファレンス」の『db2set - DB2 プロファイル・レジストリー・コマンド』

DB2 セットアップ・ウィザードを使用した DB2 インフォメーション・センターのインストール (UNIX)

DB2 製品資料には、IBM Web サイト、イントラネット・サーバー、自分のコンピューターにインストールされているバージョン、という 3 つの方法でアクセスできます。デフォルトでは、DB2 製品は、IBM Web サイト上の DB2 資料にアクセスします。イントラネット・サーバー上または自分のコンピューター上の DB2 資料にアクセスする場合は、 *DB2 インフォメーション・センター CD* から資料をインストールする必要があります。インストール設定を定義し、 UNIX オペレーティング・システムを使用するコンピューターに DB2 インフォメーション・センターをインストールするために、 DB2 セットアップ・ウィザードを使用できます。

前提条件:

ここでは、UNIX コンピューターに DB2 インフォメーション・センターをインストールするためのハードウェア、オペレーティング・システム、ソフトウェア、通信に関する要件を示します。

- **ハードウェア要件**

  以下のいずれかのプロセッサーが必要です。

  – PowerPC (AIX)
  – HP 9000 (HP-UX)
  – Intel 32 ビット (Linux)
  – Solaris UltraSPARC コンピューター (Solaris オペレーティング環境)

- **オペレーティング・システム要件**

  次のいずれかのオペレーティング・システムが必要です。

  – IBM AIX 5.1 (PowerPC)
  – HP-UX 11i (HP 9000)
  – Redhat Linux 8.0 (Intel 32 ビット)
  – SuSE Linux 8.1 (Intel 32 ビット)
  – Sun Solaris バージョン 8 (Solaris オペレーティング環境の UltraSPARC コンピューター上)

- **ソフトウェア要件**

  – 以下のブラウザーがサポートされています。

    - Mozilla バージョン 1.0 以上

- DB2 セットアップ・ウィザードは、グラフィック・インストーラーです。ご使用のマシンで DB2 セットアップ・ウィザードのグラフィカル・ユーザー・インターフェースを表示するための X Window システム・ソフトウェアをインプリメントする必要があります。 DB2 セットアップ・ウィザードを実行するには、ディスプレイを正しくエクスポートしたことを確認する必要があります。この例では、次のコマンドをコマンド・プロンプトで入力します。

  ```
  export DISPLAY=9.26.163.144:0.
  ```

- **通信要件**

  – TCP/IP

手順:

次のようにして、DB2 セットアップ・ウィザードを使用して DB2 インフォメーション・センターをインストールします。

1. システムにログオンします。
2. DB2 インフォメーション・センター製品 CD をシステムに挿入してマウントします。
3. 次のコマンドを入力することによって、 CD がマウントされているディレクトリーに移動します。

   ```
   cd /cd
   ```

   /cd は、CD のマウント・ポイントを表しています。
4. **./db2setup** コマンドを入力して、 DB2 セットアップ・ウィザードを開始します。
5. **IBM DB2 セットアップ・ランチパッド**が開きます。 DB2 インフォメーション・センターのインストールに直接進むには、**「製品のインストール」**をクリックします。残りのステップについて説明しているオンライン・ヘルプを利用できます。オンライン・ヘルプを呼び出すには、**「ヘルプ (Help)」**をクリックします。**「キャンセル」**をクリックすれば、いつでもインストールを終了できます。
6. **「インストールしたい製品を選択します」**ウィンドウで、**「次へ」**をクリックします。
7. **「DB2 インフォメーション・センターの DB2 セットアップ・ウィザードにようこそ (Welcome to the DB2 Setup wizard for DB2 Information Center)」**ウィンドウで、**「次へ」**をクリックします。 DB2 セットアップ・ウィザードがプログラムのセットアップ操作を案内します。
8. インストールを進めるには、ご使用条件を受け入れる必要があります。**「ご使用条件 (License Agreement)」**ウィンドウで、**「ご使用条件に同意します (I accept the terms in the license agreement)」**を選択して、**「次へ」**をクリックします。
9. **「インストール・アクションの選択 (Select the installation action)」**ウィンドウで、 DB2 インフォメーション・センターをインストールするロケーションを選択します。応答ファイルを使用して、このコンピューターまたは他のコンピューターに後から DB2 インフォメーション・センターをインストールする場合は、 **「設定を応答ファイルに保管する」**を選択します。**「次へ」**をクリックします。
10. **「インストールする言語の選択 (Select the languages to install)」**ウィンドウで、 DB2 インフォメーション・センターをインストールする言語を選択します。**「次へ」**をクリックします。
11. **「DB2 インフォメーション・センター・ポートの指定」**で、着信通信用に DB2 インフォメーション・センターを構成します。**「次へ」**をクリックしてインストールを続けます。
12. **「ファイルのコピーの開始 (Start copying files)」**ウィンドウで、インストールに関するこれまでの選択内容を確認します。設定の確認や変更を行うには、

「戻る」をクリックします。 DB2 インフォメーション・センターのファイルをコンピューターにコピーする処理を開始するには、**「インストール」**をクリックします。

応答ファイルを使用して、DB2 インフォメーション・センターもインストールできます。

インストール・ログの db2setup.his、 db2setup.log、db2setup.err は、デフォルトで /tmp ディレクトリーに配置されます。ログ・ファイルのロケーションは自分で指定することもできます。

db2setup.log ファイルには、エラーも含めて、DB2 製品のインストールに関するあらゆる情報が書き込まれます。 db2setup.his ファイルには、コンピューター上の DB2 製品のインストール内容がすべて記録されます。 DB2 は、db2setup.his ファイルに db2setup.log ファイルを付加します。 db2setup.err ファイルには、例外やトラップ情報など、Java から戻されるエラー出力がすべてキャプチャーされます。

インストールが完了したら、どの UNIX オペレーティング・システムかに応じて、DB2 インフォメーション・センターは次のディレクトリーのいずれかにインストールされています。

- AIX: /usr/opt/db2_08_01
- HP-UX: /opt/IBM/db2/V8.1
- Linux: /opt/IBM/db2/V8.1
- Solaris オペレーティング環境: /opt/IBM/db2/V8.1

関連概念:

- 384 ページの『DB2 インフォメーション・センター』
- 386 ページの『DB2 インフォメーション・センターのインストール・シナリオ』

関連タスク:

- 「*インストールおよび構成 補足*」の『応答ファイルによる DB2 のインストール (UNIX)』
- 395 ページの『コンピューターまたはイントラネット・サーバーへの DB2 インフォメーション・センターの更新インストール』
- 396 ページの『目的の言語による DB2 インフォメーション・センター・トピックの表示』
- 394 ページの『DB2 インフォメーション・センターの呼び出し』
- 391 ページの『DB2 セットアップ・ウィザードを使用した DB2 インフォメーション・センターのインストール (Windows)』

DB2 セットアップ・ウィザードを使用した DB2 インフォメーション・センターのインストール (Windows)

DB2 製品資料には、IBM Web サイト、イントラネット・サーバー、自分のコンピューターにインストールされているバージョン、という 3 つの方法でアクセスできます。デフォルトでは、DB2 製品は、IBM Web サイト上の DB2 資料にアクセスします。イントラネット・サーバー上または自分のコンピューター上の DB2 資料

にアクセスする場合は、 *DB2* インフォメーション・センター *CD* から DB2 資料をインストールする必要があります。インストール設定を定義し、 Windows オペレーティング・システムを使用するコンピューターに DB2 インフォメーション・センターをインストールするために、 DB2 セットアップ・ウィザードを使用できます。

前提条件:

ここでは、Windows に DB2 インフォメーション・センターをインストールするためのハードウェア、オペレーティング・システム、ソフトウェア、通信に関する要件を示します。

- **ハードウェア要件**

  以下のプロセッサーが必要です。

  – 32 ビット・コンピューター: Pentium または Pentium 互換の CPU。

- **オペレーティング・システム要件**

  次のいずれかのオペレーティング・システムが必要です。

  – Windows 2000

  – Windows XP

- **ソフトウェア要件**

  – 次のブラウザーがサポートされています。

    - Mozilla 1.0 以上

    - Internet Explorer バージョン 5.5 または 6.0 (Windows XP の場合はバージョン 6.0)

- **通信要件**

  – TCP/IP

手順:

DB2 セットアップ・ウィザードを使用して DB2 インフォメーション・センターをインストールするには、以下のようにします。

1. DB2 インフォメーション・センターのインストール用に定義したアカウントでシステムにログオンします。
2. CD をドライブに挿入します。自動実行機能が使用可能になっていれば、 IBM DB2 セットアップ・ランチパッドが起動します。
3. DB2 セットアップ・ウィザードは、システム言語を判別してから、その言語用のセットアップ・プログラムを立ち上げます。セットアップ・プログラムを別の言語で実行したい場合や、セットアップ・プログラムが自動開始しない場合には、 DB2 セットアップ・ウィザードを手動で開始できます。

   DB2 セットアップ・ウィザードを手動で開始するには、次のようにします。

   a. **「スタート」**をクリックし、**「ファイル名を指定して実行」**を選択します。

   b. **「名前」**フィールドで、次のコマンドを入力します。

      ```
      x:¥setup language
      ```

      x: は CD ドライブ、 *language* はセットアップ・プログラムを実行する言語を表します。

c. **「OK」**をクリックします。

4. **IBM DB2 セットアップ・ランチパッド**が開きます。 DB2 インフォメーション・センターのインストールに直接進むには、**「製品のインストール」**をクリックします。残りのステップについて説明しているオンライン・ヘルプを利用できます。オンライン・ヘルプを呼び出すには、**「ヘルプ (Help)」**をクリックします。**「キャンセル」**をクリックすれば、いつでもインストールを終了できます。
5. **「インストールしたい製品を選択します」**ウィンドウで、**「次へ」**をクリックします。
6. **「DB2 インフォメーション・センターの DB2 セットアップ・ウィザードにようこそ (Welcome to the DB2 Setup wizard for DB2 Information Center)」**ウィンドウで、**「次へ」**をクリックします。 DB2 セットアップ・ウィザードがプログラムのセットアップ操作を案内します。
7. インストールを進めるには、ご使用条件を受け入れる必要があります。**「ご使用条件 (License Agreement)」**ウィンドウで、**「ご使用条件に同意します (I accept the terms in the license agreement)」**を選択して、**「次へ」**をクリックします。
8. **「インストール・アクションの選択 (Select the installation action)」**ウィンドウで、 DB2 インフォメーション・センターをインストールするロケーションを選択します。応答ファイルを使用して、このコンピューターまたは他のコンピューターに後から DB2 インフォメーション・センターをインストールする場合は、 **「設定を応答ファイルに保管する」**を選択します。**「次へ」**をクリックします。
9. **「インストールする言語の選択 (Select the languages to install)」**ウィンドウで、 DB2 インフォメーション・センターをインストールする言語を選択します。**「次へ」**をクリックします。
10. **「DB2 インフォメーション・センター・ポートの指定」**で、着信通信用に DB2 インフォメーション・センターを構成します。**「次へ」**をクリックしてインストールを続けます。
11. **「ファイルのコピーの開始 (Start copying files)」**ウィンドウで、インストールに関するこれまでの選択内容を確認します。設定の確認や変更を行うには、**「戻る」**をクリックします。 DB2 インフォメーション・センターのファイルをコンピューターにコピーする処理を開始するには、**「インストール」**をクリックします。

応答ファイルを使用して、DB2 インフォメーション・センターをインストールできます。 **db2rspgn** コマンドを使用すれば、既存のインストール・システムに基づいて応答ファイルを生成することも可能です。

インストール時に発生したエラーの詳細については、 'My Documents'¥DB2LOG¥ ディレクトリーにある db2.log ファイルと db2wi.log ファイルを参照してください。 My Documents ディレクトリーのロケーションは、ご使用のコンピューターの設定によって異なります。

db2wi.log ファイルには、最新の DB2 インストールの情報がキャプチャーされます。 db2.log には、 DB2 製品のインストールの履歴が記録されます。

関連概念:

- 384 ページの『DB2 インフォメーション・センター』
- 386 ページの『DB2 インフォメーション・センターのインストール・シナリオ』

関連タスク:

- 「*インストールおよび構成 補足*」の『応答ファイルによる DB2 製品のインストール (Windows)』
- 395 ページの『コンピューターまたはイントラネット・サーバーへの DB2 インフォメーション・センターの更新インストール』
- 396 ページの『目的の言語による DB2 インフォメーション・センター・トピックの表示』
- 394 ページの『DB2 インフォメーション・センターの呼び出し』
- 389 ページの『DB2 セットアップ・ウィザードを使用した DB2 インフォメーション・センターのインストール (UNIX)』

関連資料:

- 「コマンド・リファレンス」の『db2rspgn - 応答ファイル生成プログラム・コマンド』

DB2 インフォメーション・センターの呼び出し

DB2 インフォメーション・センターは、Linux、UNIX、および Windows オペレーティング・システム用の DB2 製品 (DB2 Universal Database、 DB2 Connect、DB2 Information Integrator、 DB2 Query Patroller など) を使用するために必要なすべての情報を提供します。

DB2 インフォメーション・センターは、以下の場所から呼び出すことができます。

- DB2 UDB クライアントまたはサーバーがインストールされているコンピューター
- 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イントラネット・サーバーまたはローカル・コンピューター
- IBM の Web サイト

前提条件:

DB2 インフォメーション・センターを呼び出すための要件は、以下のとおりです。

- *オプショナル*: 希望する言語でトピックを表示するようブラウザーを構成する
- *オプショナル*: 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・センターを使用するよう DB2 クライアントを構成する

手順:

DB2 UDB クライアントまたはサーバーがインストールされているコンピューターから DB2 インフォメーション・センターを呼び出すには、以下のようにします。

- (Windows オペレーティング・システムの)「スタート」メニューから: **「スタート」→「プログラム」→「IBM DB2」→「情報」→「インフォメーション・センター」**をクリックします。
- コマンド行プロンプトから:

– Linux および UNIX オペレーティング・システムの場合、 **db2icdocs** コマンドを発行します。
– Windows オペレーティング・システムの場合、 **db2icdocs.exe** コマンドを発行します。

イントラネット・サーバーまたはローカル・コンピュータ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 Web ブラウザーで開くには、以下のようにします。

- Web ページ http://<host-name>:<port-number>/ を開きます (<host-name> はホスト名、 <port-number> は DB2 インフォメーション・センターを利用可能なポート番号)。

IBM Web サイトにある DB2 インフォメーション・センターを Web ブラウザーで開くには、以下のようにします。

- Web ページ publib.boulder.ibm.com/infocenter/db2help/ を開きます。

関連概念:

- 384 ページの『DB2 インフォメーション・センター』

関連タスク:

- 396 ページの『目的の言語による DB2 インフォメーション・センター・トピックの表示』
- 405 ページの『DB2 ツールからコンテキスト・ヘルプを呼び出す』
- 395 ページの『コンピューターまたはイントラネット・サーバーへの DB2 インフォメーション・センターの更新インストール』
- 406 ページの『コマンド行プロセッサーからメッセージ・ヘルプを呼び出す』
- 406 ページの『コマンド行プロセッサーからコマンド・ヘルプを呼び出す』
- 407 ページの『コマンド行プロセッサーから SQL 状態ヘルプを呼び出す』

コンピューターまたはイントラネット・サーバーへの DB2 インフォメーション・センターの更新インストール

http://publib.boulder.ibm.com/infocenter/db2help/ から利用できる DB2 インフォメーション・センターは、資料の新規追加または変更によって定期的に更新されます。さらに、更新された DB2 インフォメーション・センターをコンピューターまたはイントラネット・サーバーにダウンロードしてインストールできる場合もあります。DB2 インフォメーション・センターを更新しても、 DB2 クライアント製品またはサーバー製品は更新されません。

前提条件:

インターネットに接続されたコンピューターへのアクセスが必要です。

手順:

DB2 インフォメーション・センターの更新を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するには、以下のようにします。

1. IBM の Web サイト (http://publib.boulder.ibm.com/infocenter/db2help/) にある DB2 インフォメーション・センターを開きます。
2. 「DB2 インフォメーション・センターにようこそ」ページの見出し「サービスおよびサポート」の「ダウンロード」セクションで、**「DB2 資料」**リンクをクリックします。
3. 最新のドキュメンテーション・イメージのレベルと、インストール済みのドキュメンテーション・レベルを比較して、 DB2 インフォメーション・センターを更新する必要があるかどうかを確認します。「DB2 インフォメーション・センターにようこそ」ページに、インストール済みのドキュメンテーションのレベルがリストされます。
4. より新しいバージョンの DB2 インフォメーション・センターが存在する場合、ご使用のオペレーティング・システムに対応する最新の *DB2 インフォメーション・センター・イメージをダウンロードします。
5. 最新の *DB2 インフォメーション・センター・イメージをインストールするには、 Web ページの指示に従ってください。

関連概念:

- 386 ページの『DB2 インフォメーション・センターのインストール・シナリオ』

関連タスク:

- 394 ページの『DB2 インフォメーション・センターの呼び出し』
- 389 ページの『DB2 セットアップ・ウィザードを使用した DB2 インフォメーション・センターのインストール (UNIX)』
- 391 ページの『DB2 セットアップ・ウィザードを使用した DB2 インフォメーション・センターのインストール (Windows)』

目的の言語による DB2 インフォメーション・センター・トピックの表示

DB2 インフォメーション・センターは、ブラウザー設定に指定されている言語でトピックを表示しようとします。 DB2 インフォメーション・センターのトピックがその言語に翻訳されていない場合は、英語で表示されます。

手順:

Internet Explorer ブラウザーの場合、トピックを目的の言語で表示するには、次のようにします。

1. Internet Explorer で、**「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. 目的の言語が言語リストの最初の項目として指定されているかどうかを確認します。
   - 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、その言語でトピックを表示するために必要なフォントがコンピューターに存在しない場合もあります。
   - 言語をリストの先頭に移動するには、その言語を選択して、その言語が言語リストの先頭に来るまで**「上へ」**をクリックします。

3. DB2 インフォメーション・センターを目的の言語で表示するために、ページを最新の表示に更新します。

Mozilla ブラウザーの場合、トピックを目的の言語で表示するには、次のようにします。

1. Mozilla で、**「編集」** —> **「設定」** —> **「言語」**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. 目的の言語が言語リストの最初の項目として指定されているかどうかを確認します。
   - 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、「言語を追加」ウィンドウから言語を選択します。
   - 言語をリストの先頭に移動するには、その言語を選択して、その言語が言語リストの先頭に来るまで**「上へ」**をクリックします。
3. DB2 インフォメーション・センターを目的の言語で表示するために、ページを最新の表示に更新します。

関連概念:

- 384 ページの『DB2 インフォメーション・センター』

DB2 PDF 資料および印刷された資料

以下の表は、正式な資料名、資料番号、および PDF ファイル名を示しています。ハードコピー版の資料を注文するには、正式な資料名を知っておく必要があります。 PDF ファイルを印刷するには、PDF ファイル名を知っておく必要があります。

DB2 資料は、以下のカテゴリーに分類されています。

- DB2 中核情報
- 管理情報
- アプリケーション開発情報
- ビジネス・インテリジェンス情報
- DB2 Connect 情報
- 入門情報
- チュートリアル情報
- オプショナル・コンポーネント情報
- リリース・ノート

以下の表は、DB2 ライブラリー内の各資料について、その資料のハードコピー版を注文したり、PDF 版を印刷または表示したりするのに必要な情報を示しています。DB2 ライブラリー内の各資料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。

DB2 の基本情報

こうした資料の情報は、すべての DB2 ユーザーに基本的なもので、プログラマーおよびデータベース管理者にとって役立つ情報であるとともに、DB2 Connect、

DB2 Warehouse Manager、または他の DB2 製品を使用するユーザーにとっても役立つ内容です。

表 39. DB2 の基本情報

資料名	資料番号	PDF ファイル名
「*IBM DB2 Universal Database コマンド・リファレンス*」	SC88-9140	db2n0j81
「*IBM DB2 Universal Database 用語集*」	資料番号なし	db2t0j81
「*IBM DB2 Universal Database メッセージ・リファレンス 第 1 巻*」	GC88-9152 (ハードコピーなし)	db2m1j81
「*IBM DB2 Universal Database メッセージ・リファレンス 第 2 巻*」	GC88-9153 (ハードコピーなし)	db2m2j81
「*IBM DB2 Universal Database 新機能*」	SC88-9158	db2q0j81

管理情報

これらの資料の情報は、DB2 データベース、データウェアハウス、およびフェデレーテッド・システムを効果的に設計し、インプリメントし、保守するために必要なトピックを扱っています。

表 40. 管理情報

資料名	資料番号	PDF ファイル名
「*IBM DB2 Universal Database 管理ガイド: プランニング*」	SC88-9135	db2d1j81
「*IBM DB2 Universal Database 管理ガイド: インプリメンテーション*」	SC88-9133	db2d2j81
「*IBM DB2 Universal Database 管理ガイド: パフォーマンス*」	SC88-9134	db2d3j81
「*IBM DB2 Universal Database 管理 API リファレンス*」	SC88-9136	db2b0j81
「*IBM DB2 Universal Database 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*」	SC88-9142	db2dmj81
「*IBM DB2 Universal Database 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*」	SC88-9143	db2haj81
「*IBM DB2 Universal Database データウェアハウス・センター 管理ガイド*」	SC88-9165	db2ddj81
「*IBM DB2 Universal Database SQL リファレンス 第 1 巻*」	SC88-9155	db2s1j81
「*IBM DB2 Universal Database SQL リファレンス 第 2 巻*」	SC88-9156	db2s2j81

表 40. 管理情報 (続き)

資料名	資料番号	PDF ファイル名
「*IBM DB2 Universal Database システム・モニター ガイドおよびリファレンス*」	SC88-9157	db2f0j81

アプリケーション開発情報

これらの資料の情報は、 DB2 Universal Database (DB2 UDB) のアプリケーション開発者またはプログラマーが特に関心を持つ内容です。サポートされるさまざまなプログラミング・インターフェース (組み込み SQL、ODBC、JDBC、SQLJ、CLI など) を使用して DB2 UDB にアクセスするのに必要な資料とともに、サポートされる言語およびコンパイラーについても紹介されています。また、DB2 インフォメーション・センターをご使用の場合には、サンプル・プログラムのソース・コードの HTML バージョンにアクセスすることもできます。

表 41. アプリケーション開発情報

資料名	資料番号	PDF ファイル名
「*IBM DB2 Universal Database アプリケーション開発ガイド アプリケーションの構築および実行*」	SC88-9137	db2axj81
「*IBM DB2 Universal Database アプリケーション開発ガイド クライアント・アプリケーションのプログラミング*」	SC88-9138	db2a1j81
「*IBM DB2 Universal Database アプリケーション開発ガイド サーバー・アプリケーションのプログラミング*」	SC88-9139	db2a2j81
「*IBM DB2 Universal Database 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*」	SC88-9159	db2l1j81
「*IBM DB2 Universal Database 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*」	SC88-9160	db2l2j81
「*IBM DB2 Universal Database データウェアハウス・センター アプリケーション統合ガイド*」	SC88-9166	db2adj81
「*IBM DB2 Universal Database XML Extender 管理およびプログラミングのガイド*」	SC88-9172	db2sxj81

ビジネス・インテリジェンス情報

これらの資料の情報は、さまざまなコンポーネントを使用して、 DB2 Universal Database のデータウェアハウジング機能および分析機能を拡張する方法を説明しています。

表 42. ビジネス・インテリジェンス情報

資料名	資料番号	PDF ファイル名
「*IBM DB2 Warehouse Manager Standard Edition インフォメーション・カタログ・センター 管理ガイド*」	SC88-9167	db2dij81
「*IBM DB2 Warehouse Manager Standard Edition インストール・ガイド*」	GC88-9164	db2idj81
「*IBM DB2 Warehouse Manager Standard Edition DB2 Warehouse Manager を使用時の ETI ソリューション・コンバージョン・プログラムの管理*」	SC88-9894	iwhe1mstx80

DB2 Connect 情報

このカテゴリーの情報は、 DB2 Connect Enterprise Edition または DB2 Connect Personal Edition を使用して、メインフレーム・サーバーおよびミッドレンジ・サーバー上のデータにアクセスする方法を説明しています。

表 43. DB2 Connect 情報

資料名	資料番号	PDF ファイル名
「*IBM コネクティビティー 補足*」	資料番号なし	db2h1j81
「*IBM DB2 Connect Enterprise Edition 概説およびインストール*」	GC88-9145	db2c6j81
「*IBM DB2 Connect Personal Edition 概説およびインストール*」	GC88-9146	db2c1j81
「*IBM DB2 Connect ユーザーズ・ガイド*」	SC88-9147	db2c0j81

入門情報

このカテゴリーの情報は、サーバー、クライアント、および他の DB2 製品をインストールして構成する場合に役立ちます。

表 44. 入門情報

資料名	資料番号	PDF ファイル名
「*IBM DB2 Universal Database DB2 クライアント機能 概説およびインストール*」	GC88-9144 (ハードコピーなし)	db2itj81
「*IBM DB2 Universal Database DB2 サーバー機能 概説およびインストール*」	GC88-9148	db2isj81
「*IBM DB2 Universal Database DB2 Personal Edition 概説およびインストール*」	GC88-9150	db2i1j81
「*IBM DB2 Universal Database インストールおよび構成 補足*」	GC88-9149 (ハードコピーなし)	db2iyj81
「*IBM DB2 Universal Database DB2 Data Links Manager 概説およびインストール*」	GC88-9141	db2z6j81

チュートリアル情報

チュートリアル情報は、DB2 機能を紹介し、さまざまなタスクを実行する方法を示します。

表 45. チュートリアル情報

資料名	資料番号	PDF ファイル名
「*ビジネス・インテリジェンス・チュートリアル: データウェアハウス・センターの紹介*」	資料番号なし	db2tuj81
「*ビジネス・インテリジェンス・チュートリアル: データウェアハウジングの上級者向けガイド*」	資料番号なし	db2taj81
「*インフォメーション・カタログ・センター チュートリアル*」	資料番号なし	db2aij81
「*Video Central for e-business チュートリアル*」	資料番号なし	db2twj81
「*Visual Explain チュートリアル*」	資料番号なし	db2tvj81

オプショナル・コンポーネント情報

このカテゴリーの情報は、 DB2 のオプショナル・コンポーネントを使用する方法について説明しています。

表 46. オプショナル・コンポーネント情報

資料名	資料番号	PDF ファイル名
「*IBM DB2 Cube Views Guide and Reference*」	SC18–7298	db2aax81
「*IBM DB2 Query Patroller インストール、管理、使用法のガイド*」	GC88–9154	db2dwj81
「*IBM DB2 Spatial Extender and Geodetic Extender ユーザーズ・ガイドおよびリファレンス*」	SC88-9171	db2sbj81
「*IBM DB2 Universal Database Data Links Manager 管理ガイドおよびリファレンス*」	SC88-9169	db2z0x82
「*DB2 Net Search Extender 管理およびユーザーズ・ガイド*」 **注:** この資料の HTML 版は、HTML ドキュメンテーション CD からインストールされません。	SH88-8546	N/A

リリース・ノート

リリース・ノートは、ご使用の製品のリリースおよびフィックスパック・レベルに特有の追加情報を紹介します。また、リリース・ノートには、各リリース、アップデート、およびフィックスパックで組み込まれた資料上の更新の要約も含まれています。

表 47. リリース・ノート

資料名	資料番号	PDF ファイル名
「*DB2 リリース・ノート*」	「注」を参照。	「注」を参照。
「*DB2 インストール情報*」	製品 CD-ROM でのみ参照可能。	使用できません。

注: リリース・ノートは以下の形式で入手できます。

- XHTML およびテキスト形式 (製品 CD 内)
- PDF 形式 (PDF ドキュメンテーション CD 内)

さらに、リリース・ノートの中で、『*既知の問題と予備手段*』および『*リリース間の非互換性*』に関する部分は DB2 インフォメーション・センターにも表示されます。

UNIX ベースのプラットフォームでテキスト形式でリリース・ノートを確認するには、 Release.Notes ファイルを参照してください。このファイルは、DB2DIR/Readme/*%L* ディレクトリーに収録されています。 *%L* はロケール名を表しています。 DB2DIR は以下になります。

- AIX オペレーティング・システムの場合: /usr/opt/db2_08_01
- その他のすべての UNIX ベースのオペレーティング・システムの場合: /opt/IBM/db2/V8.1

関連概念:

- 383 ページの『DB2 資料およびヘルプ』

関連タスク:

- 403 ページの『PDF ファイルからの DB2 資料の印刷方法』
- 404 ページの『DB2 の印刷資料の注文方法』
- 405 ページの『DB2 ツールからコンテキスト・ヘルプを呼び出す』

PDF ファイルからの DB2 資料の印刷方法

DB2 PDF ドキュメンテーション CD に収録されている DB2 資料を印刷することができます。 Adobe Acrobat Reader を使用すれば、資料全体または特定のページを印刷できます。

前提条件:

Adobe Acrobat Reader がインストールされていることを確認してください。 Adobe Acrobat Reader をインストールする必要がある場合、 Adobe Web サイト (www.adobe.com) から入手できます。

手順:

PDF ファイルから DB2 資料を印刷するには以下のようにします。

1. *DB2 PDF* ドキュメンテーション CD をドライブに挿入します。 UNIX オペレーティング・システムの場合、 DB2 PDF ドキュメンテーション CD をマウントします。 UNIX オペレーティング・システムで CD をマウントする方法については、「*概説およびインストール*」を参照してください。
2. index.htm を開きます。ブラウザー・ウィンドウにファイルが開きます。
3. 参照したい PDF のタイトルをクリックします。 Acrobat Reader で PDF が開きます。
4. **「ファイル」 → 「印刷」**を選択して、所要の資料の任意の部分を印刷します。

関連概念:

- 384 ページの『DB2 インフォメーション・センター』

関連タスク:

- 「*DB2 Universal Database サーバー機能 概説およびインストール*」の『CD-ROM のマウント (AIX)』
- 「*DB2 Universal Database サーバー機能 概説およびインストール*」の『HP-UX 上での CD-ROM のマウント』
- 「*DB2 Universal Database サーバー機能 概説およびインストール*」の『CD-ROM のマウント (Linux)』
- 404 ページの『DB2 の印刷資料の注文方法』

- 「*DB2 Universal Database サーバー機能 概説およびインストール*」の『CD-ROM のマウント (Solaris)』

関連資料:

- 397 ページの『DB2 PDF 資料および印刷された資料』

DB2 の印刷資料の注文方法

ハードコピー版の資料を望む場合には、以下のいずれかの方法で注文できます。

印刷資料の注文方法:

一部の国または地域では、印刷された資料を注文することもできます。お客様がお住まいの国または地域でこのサービスが利用可能かどうかを確認するには、お住まいの国または地域の IBM Publications Web サイトをご覧ください。資料のご注文が可能な場合、以下のようにすることができます。

- 正規の IBM 製品販売業者または営業担当員に連絡してください。お客様がお住まいの地域の IBM 担当員の情報については、お手数ですが IBM の Web サイト (www.ibm.com/planetwide) の IBM Worldwide Directory of Contacts で確認してください。
- IBM Publications Center (http://www.ibm.com/shop/publications/order) にアクセスしてください。なお、IBM Publications Center から資料を注文できない国もあります。

DB2 製品がご利用可能になった時点で、印刷された資料は *DB2 PDF* ドキュメンテーション *CD* にある PDF 形式の資料と同じものです。さらに、 *DB2* インフォメーション・センター *CD* に収録されている印刷された資料の内容もまた、これらと同じです。ただし、DB2 インフォメーション・センター CD には、 PDF 資料にない追加情報も含まれます (たとえば、SQL 管理作業や HTML サンプル)。 DB2 PDF ドキュメンテーション CD に収録されている資料の中には、ハードコピーとしてご注文できない資料もあります。

注: DB2 インフォメーション・センターは、 PDF またはハードコピー の資料よりも頻繁に更新されます。ドキュメンテーションの更新が入手可能になった時点でインストールするか、 DB2 インフォメーション・センター (http://publib.boulder.ibm.com/infocenter/db2help/) を参照して最新の情報を入手してください。

関連タスク:

- 403 ページの『PDF ファイルからの DB2 資料の印刷方法』

関連資料:

- 397 ページの『DB2 PDF 資料および印刷された資料』

DB2 ツールからコンテキスト・ヘルプを呼び出す

コンテキスト・ヘルプは、特定のウィンドウ、ノートブック、ウィザード、またはアドバイザーに関連したタスクまたはコントロールの情報を提供します。コンテキスト・ヘルプは、グラフィカル・ユーザー・インターフェースのある DB2 管理ツールおよび開発ツールから利用できます。コンテキスト・ヘルプには、以下の 2 種類があります。

- それぞれのウィンドウまたはノートブックにある**「ヘルプ」**ボタンからアクセス可能なヘルプ
- infopop (ポップアップ情報ウィンドウ)。これは、マウス・カーソルを特定のフィールドまたはコントロール上に置いたとき、またはウィンドウ、ノートブック、ウィザード、アドバイザー内でフィールドまたはコントロールを選択して F1 を押すと表示されます。

「ヘルプ」ボタンを押すと、概説、前提条件、およびタスク情報が表示されます。infopop は、それぞれのフィールドおよびコントロールについて説明します。

手順:

コンテキスト・ヘルプを呼び出すには、以下のようにします。

- ウィンドウおよびノートブックのヘルプを表示するには、いずれかの DB2 ツールを開始して、任意のウィンドウまたはノートブックを開きます。ウィンドウまたはノートブックの右下隅にある**「ヘルプ」**ボタンをクリックして、コンテキスト・ヘルプを呼び出します。

  また、それぞれの DB2 ツール・センターの上部にある**「ヘルプ」**メニュー項目からコンテキスト・ヘルプにアクセスすることもできます。

  ウィザードおよびアドバイザーでは、最初のページの「タスクの概要」リンクをクリックすると、コンテキスト・ヘルプを表示できます。
- ウィンドウまたはノートブック上の各コントロールの infopop ヘルプを表示するには、コントロールをクリックしてから、**F1** を押します。コントロールの詳細情報を示すポップアップ情報が、黄色いウィンドウに表示されます。

  注: フィールドまたはコントロールにマウス・カーソルを置いておくだけで infopops が表示されるようにするには、「ツール設定」ノートブックの**「文書 (Documentation)」**ページの**「infopops の自動表示」**チェック・ボックスを選択します。

  infopop に似た別のコンテキスト・ヘルプに、診断ポップアップ情報があります。これにはデータ入力規則が示されます。診断ポップアップ情報は、無効または不十分なデータが入力されたとき、紫色のウィンドウに表示されます。診断ポップアップ情報は、以下に関して表示されます。
  - 必須フィールド。
  - 日付フィールドのように、正確なフォーマットを必要とするデータのフィールド。

関連タスク:

- 394 ページの『DB2 インフォメーション・センターの呼び出し』
- 406 ページの『コマンド行プロセッサーからメッセージ・ヘルプを呼び出す』

- 406 ページの『コマンド行プロセッサーからコマンド・ヘルプを呼び出す』
- 407 ページの『コマンド行プロセッサーから SQL 状態ヘルプを呼び出す』
- 『DB2 UDB ヘルプの使用法: Common GUI help』
- 『DB2 コンテキスト・ヘルプと資料へのアクセスを設定する: Common GUI help』

コマンド行プロセッサーからメッセージ・ヘルプを呼び出す

メッセージ・ヘルプは、メッセージが出された原因と、エラーへの応答として実行すべきアクションを説明します。

手順:

メッセージ・ヘルプを呼び出すには、コマンド行プロセッサーを開いて以下のように入力します。

```
? XXXnnnnn
```

ここで、*XXXnnnnn* は有効なメッセージ ID を表します。

たとえば、`? SQL30081` と入力すると、メッセージ SQL30081 に関するヘルプを表示します。

関連概念:

- 「メッセージ・リファレンス *第 1 巻*」の『メッセージの概要』

関連資料:

- 「コマンド・リファレンス」の『db2 - コマンド行プロセッサーの呼び出しコマンド』

コマンド行プロセッサーからコマンド・ヘルプを呼び出す

コマンド・ヘルプは、コマンド行プロセッサーでのコマンドの構文を説明します。

手順:

コマンド・ヘルプを呼び出すには、コマンド行プロセッサーを開いて以下のように入力します。

```
? command
```

ここで *command* はキーワードまたはコマンド全体を表します。

たとえば、`? catalog` と入力すると、すべての CATALOG コマンドに関するヘルプが表示され、 `? catalog database` と入力すると、 CATALOG DATABASE コマンドのヘルプだけが表示されます。

関連タスク:

- 405 ページの『DB2 ツールからコンテキスト・ヘルプを呼び出す』
- 394 ページの『DB2 インフォメーション・センターの呼び出し』
- 406 ページの『コマンド行プロセッサーからメッセージ・ヘルプを呼び出す』

- 407 ページの『コマンド行プロセッサーから SQL 状態ヘルプを呼び出す』

関連資料:

- 「コマンド・リファレンス」の『db2 - コマンド行プロセッサーの呼び出しコマンド』

コマンド行プロセッサーから SQL 状態ヘルプを呼び出す

DB2 Univerrsal Database は、 SQL ステートメントの結果の原因と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。

手順:

SQL 状態ヘルプを呼び出すには、コマンド行プロセッサーを開いて以下のように入力します。

```
? sqlstate または ? 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。

たとえば、`? 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。

関連タスク:

- 394 ページの『DB2 インフォメーション・センターの呼び出し』
- 406 ページの『コマンド行プロセッサーからメッセージ・ヘルプを呼び出す』
- 406 ページの『コマンド行プロセッサーからコマンド・ヘルプを呼び出す』

DB2 チュートリアル

DB2® チュートリアルは、 DB2 Universal Database のさまざまな機能について学習するのを支援します。このチュートリアルでは、アプリケーションの開発、SQL 照会のパフォーマンス調整、データウェアハウスの処理、メタデータの管理、および DB2 を使用した Web サービスの開発の各分野で、段階的なレッスンが用意されています。

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。

チュートリアルの中で、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、それぞれのチュートリアルを参照してください。

DB2 Universal Database チュートリアル:

以下に示すチュートリアルのタイトルをクリックすると、そのチュートリアルを表示できます。

ビジネス・インテリジェンス・チュートリアル: データウェアハウス・センターの紹介
データウェアハウス・センターを使用して簡単なデータウェアハウジング・タスクを実行します。

ビジネス・インテリジェンス・チュートリアル: データウェアハウジングの上級者向けガイド
データウェアハウス・センターを使用して高度なデータウェアハウジング・タスクを実行します。

インフォメーション・カタログ・センター・チュートリアル
インフォメーション・カタログを作成および管理して、インフォメーション・カタログ・センターを使用してメタデータを配置し使用します。

Visual Explain チュートリアル
Visual Explain を使用して、パフォーマンスを向上させるために SQL ステートメントを分析し、最適化し、調整します。

DB2 トラブルシューティング情報

DB2® 製品を使用する際に役立つ、トラブルシューティングおよび問題判別に関する広範囲な情報を利用できます。

DB2 ドキュメンテーション
トラブルシューティング情報は、DB2 インフォメーション・センター、および DB2 ライブラリーに含まれる PDF 資料の中でご利用いただけます。DB2 インフォメーション・センターで、 (ブラウザー・ウィンドウの左側の) ナビゲーション・ツリーの「サポートおよびトラブルシューティング (Support and troubleshooting)」ブランチを参照すると、 DB2 トラブルシューティング・ドキュメンテーションの詳細なリストが見つかります。

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 出版物、TechNotes、プログラム診断依頼書 (APAR)、フィックスパック、DB2 内部エラー・コードの最新リスト、その他のリソースが用意されています。この知識ベースを活用して、問題に対する有効なソリューションを探し出すことができます。

DB2 Technical Support の Web サイト
(http://www.ibm.com/software/data/db2/udb/winos2unix/support) にアクセスしてください。

DB2 Problem Determination Tutorial Series
DB2 製品で作業中に直面するかもしれない問題を素早く識別し、解決する方法に関する情報を見つけるには、 DB2 Problem Determination Tutorial Series の Web サイトを参照してください。あるチュートリアルでは、使用可能な DB2 問題判別機能およびツールを紹介し、それらをいつ使用すべきかを判断する助けを与えます。別のチュートリアルは、『データベース・エンジン問題判別 (Database Engine Problem Determination)』、『パフォーマンス問題判別 (Performance Problem Determination)』、『アプリケーション問題判別 (Application Problem Determination)』などの関連トピックを扱っています。

DB2 Technical Support
(http://www.ibm.com/software/data/support/pdm/db2tutorials.html) には、 DB2 問題判別チュートリアルがすべて揃っています。

関連概念:

- 384 ページの『DB2 インフォメーション・センター』
- 「*問題判別の手引き*」の『Introduction to Problem Determination - DB2 テクニカル・サポートのチュートリアル』

アクセス支援

アクセス支援機能は、身体に障害のある (身体動作が制限されている、視力が弱いなど) ユーザーがソフトウェア製品を十分活用できるように支援します。 DB2® バージョン 8 製品に備わっている主なアクセス支援機能は、以下のとおりです。

- すべての DB2 機能は、マウスの代わりにキーボードを使ってナビゲーションできます。詳細については、『キーボードによる入力およびナビゲーション』を参照してください。
- DB2 ユーザー・インターフェースのフォント・サイズおよび色をカスタマイズすることができます。詳細については、 410 ページの『アクセスしやすい表示』を参照してください。
- DB2 製品は、Java™ Accessibility API を使用するアクセス支援アプリケーションをサポートします。詳細については、 410 ページの『支援テクノロジーとの互換性』を参照してください。
- DB2 資料は、アクセスしやすい形式で提供されています。詳細については、 410 ページの『アクセスしやすい資料』を参照してください。

キーボードによる入力およびナビゲーション

キーボード入力

キーボードだけを使用して DB2 ツールを操作できます。マウスを使って実行できる操作は、キーまたはキーの組み合わせによっても実行できます。標準のオペレーティング・システム・キー・ストロークを使用して、標準のオペレーティング・システム操作を実行できます。

キーまたはキーの組み合わせによって操作を実行する方法について、詳しくは キーボード・ショートカットおよびアクセラレーター: Common GUI help を参照してください。

キーボード・ナビゲーション

キーまたはキーの組み合わせを使用して、 DB2 ツールのユーザー・インターフェースをナビゲートできます。

キーまたはキーの組み合わせによって DB2 ツールをナビゲートする方法の詳細については、キーボード・ショートカットおよびアクセラレーター: Common GUI help を参照してください。

キーボード・フォーカス

UNIX® オペレーティング・システムでは、アクティブ・ウィンドウの中で、キー・ストロークによって操作できる領域が強調表示されます。

アクセスしやすい表示

DB2 ツールには、視力の弱いユーザー、その他の視力障害をもつユーザーのためにアクセシビリティーを向上させる機能が備わっています。これらのアクセシビリティー拡張機能には、フォント・プロパティーのカスタマイズを可能にする機能も含まれています。

フォントの設定

「ツール設定」ノートブックを使用して、メニューおよびダイアログ・ウィンドウに使用されるテキストの色、サイズ、およびフォントを選択できます。

フォント設定に関する詳細情報は、メニューおよびテキストのフォントを変更する: Common GUI help を参照してください。

色に依存しない

本製品のすべての機能を使用するために、ユーザーは必ずしも色を識別する必要はありません。

支援テクノロジーとの互換性

DB2 ツールのインターフェースは、Java Accessibility API をサポートします。これによって、スクリーン・リーダーその他の支援テクノロジーを DB2 製品で利用できるようになります。

アクセスしやすい資料

DB2 形式は、ほとんどの Web ブラウザーで表示可能な XHTML 1.0 形式で提供されています。 XHTML により、ご使用のブラウザーに設定されている表示設定に従って資料を表示できます。さらに、スクリーン・リーダーや他の支援テクノロジーを使用することもできます。

シンタックス・ダイアグラムはドット 10 進形式で提供されます。この形式は、スクリーン・リーダーを使用してオンライン・ドキュメンテーションにアクセスする場合にのみ使用できます。

関連概念:

• 410 ページの『ドット 10 進シンタックス・ダイアグラム』

ドット 10 進シンタックス・ダイアグラム

スクリーン・リーダーを使用してインフォメーション・センターを利用するユーザーのために、シンタックス・ダイアグラムがドット 10 進形式で提供されます。

ドット 10 進形式では、各シンタックス・エレメントは別々の行に書き込まれます。複数のシンタックス・エレメントが常に同時に存在する (または常に同時に不在の) 場合、単一のコンパウンド・シンタックス・エレメントとみなせるので同一行に表示できます。

各行は、ドット 10 進数で開始します。たとえば、3 または 3.1 ないしは 3.1.1 です。こうした数を適切に聞き取るには、スクリーン・リーダーが句読点を読み取るように設定されていることを確認してください。同じドット 10 進数を持つすべてのシンタックス・エレメント (たとえば、 3.1 という数値を持つすべてのシンタックス・エレメント) は、相互に排他的な代替エレメントです。 3.1 USERID および 3.1 SYSTEMID という行を聞き取る場合、シンタックスには両方ではなく USERID または SYSTEMID のどちらかが含まれることが分かります。

ドット 10 進レベルは、ネストのレベルを表示します。たとえば、ドット 10 進数 3 のシンタックス・エレメントの後に、一連のドット 10 進数 3.1 のシンタックス・エレメントが続きます。 3.1 の番号が付されたシンタックス・エレメントすべては、番号 3 の付されたシンタックス・エレメントに従属します。

シンタックス・エレメントに関する情報を追加するため、ドット 10 進数の次に特定のワードおよびシンボルが使用されます。 時折、こうしたワードおよびシンボルはエレメントの最初に表示される場合もあります。 簡単に識別するため、ワードやシンボルがシンタックス・エレメントの一部である場合には、円記号 (¥) 文字が先頭に付きます。 * シンボルはドット 10 進数の次に使用でき、シンタックス・エレメントが反復することを示します。 たとえば、ドット 10 進数 3 のシンタックス・エレメント *FILE は、3 ¥* FILE という形式になります。 3* FILE という形式は、シンタックス・エレメント FILE が反復されることを示します。 3* ¥* FILE という形式は、シンタックス・エレメント * FILE が反復されることを示します。

シンタックス・エレメントのストリングを分離するのに使用されるコンマなどの文字は、シンタックス内の分離する項目の直前に表示されます。 こうした文字は、それぞれの項目と同一行に表示するか、同じドット 10 進数を持つ関連する項目のある別の行に表示できます。 またその行には、シンタックス・エレメントに関する情報を提供する別のシンボルを表示することも可能です。 たとえば、複数の LASTRUN および DELETE シンタックス・エレメントを使用している場合には、5.1*、5.1 LASTRUN、および 5.1 DELETE という行は、エレメントをコンマで区切る必要があります。区切り文字が指定されないと、各シンタックス・エレメントを区切るのにブランクが使用されると想定されます。

シンタックス・エレメントの前に % シンボルが付く場合、他の箇所で定義されている参照であることを示します。 % シンボルの後のストリングは、リテラルではなくシンタックス・フラグメントの名前です。 たとえば、2.1 %OP1 という行は別のシンタックス・フラグメント OP1 を参照すべきことを意味します。

以下のワードおよびシンボルが、ドット 10 進数の次に使用されます。

- ? は、オプションのシンタックス・エレメントであることを表します。? シンボルが後に続くドット 10 進数は、対応するドット 10 進数のシンタックス・エレメント、および任意の従属のシンタックス・エレメントがオプションであることを示します。ドット 10 進数の付いたシンタックス・エレメントが 1 つしかない場合、 ? シンボルはそのシンタックス・エレメントと同じ行に表示されます (たとえば、5? NOTIFY)。ドット 10 進数の付いたシンタックス・エレメントが複数

ある場合、 ? シンボルだけで行に表示され、その後にオプションのシンタックス・エレメントが続きます。たとえば、「5 ?、5 NOTIFY、および 5 UPDATE」という行を聞き取る場合、シンタックス・エレメント NOTIFY および UPDATE がオプションである、つまりそのいずれかを選択でき、どちらも選択しないこともできることが分かります。 ? シンボルは、線路型ダイアグラムのバイパス線に相当します。

- ! は、デフォルトのシンタックス・エレメントであることを表します。! シンボルおよびシンタックス・エレメントが後に続くドット 10 進数は、そのシンタックス・エレメントが、同じドット 10 進数を共用するシンタックス・エレメントすべてのデフォルト・オプションであることを示します。同じドット 10 進数を共用するシンタックス・エレメントのうち 1 つだけに、 ! シンボルを指定できます。たとえば、「2? FILE、2.1! (KEEP)、および 2.1 (DELETE)」という行を聞き取る場合、 FILE キーワードのデフォルト・オプションは (KEEP) になります。この例では、FILE キーワードを含めてもオプションを指定しない場合には、デフォルト・オプション KEEP が適用されます。デフォルト・オプションは、次に高位のドット 10 進数にも適用されます。 この例の場合、FILE キーワードが省略されると、デフォルトの FILE(KEEP) が使用されます。しかし、「2? FILE、2.1、2.1.1! (KEEP)、および 2.1.1 (DELETE)」という行を聞き取る場合、デフォルト・オプション KEEP は次に高位のドット 10 進数 2.1 (関連キーワードを持っていない) にのみ適用され、 2? FILE には適用されません。キーワード FILE が省略されると、どれも使用されません。
- * は、0 回以上反復できるシンタックス・エレメントを示します。 * シンボルが後に続くドット 10 進数は、このシンタックス・エレメントが 0 回以上使用できること、つまりオプションであり、なおかつ反復できることを表します。たとえば、5.1* データ域という行を聞き取る場合、 1 つまたは複数のデータ域を含めるか、またはデータ域を全く含めないことが可能です。「3*、3 HOST、および 3 STATE」という行を聞き取る場合、 HOST、STATE をどちらか一方または両方同時に含めるか、どちらも含めないことができます。

  注:

  1. ドット 10 進数の後にアスタリスク (*) が付き、ドット 10 進数の付いた項目が 1 つしかない場合には、同じ項目を複数回反復できます。
  2. ドット 10 進数の後にアスタリスクが付き、ドット 10 進数の付いた項目が複数ある場合、リストから複数の項目を使用できますが、各項目を複数回使用することはできません。 前述の例では、HOST STATE と書くことはできますが、HOST HOST とは書けません。
  3. * シンボルは、線路型シンタックス・ダイアグラムのループバック線に相当します。
- + は、1 回以上含める必要のあるシンタックス・エレメントであることを示します。 + シンボルが後に続くドット 10 進数は、このシンタックス・エレメントを 1 回以上含める必要があること、つまり少なくとも 1 回は含める必要があり、反復できることを表します。たとえば、「6.1+ データ域」という行を聞き取る場合、データ域を少なくとも 1 回は含めなければなりません。「2+、2 HOST、および 2 STATE」という行を聞き取る場合には、 HOST、STATE、またはその両方を含める必要があります。 * シンボルと同様に、+ シンボルは、ドット 10 進

数の付いた項目が 1 つしかない場合に限り、その特定の項目のみを反復できます。 * シンボルと同様、 + シンボルは線路型シンタックス・ダイアグラムのループバック線に相当します。

関連概念:

- 409 ページの『アクセス支援』

関連タスク:

- 『目次』

関連資料:

- 「*SQL リファレンス 第 2 巻*」の『構文図の見方』

DB2 Universal Database 製品の共通基準認証

DB2 Universal Database は、 Common Criteria の評価検定レベル 4 (EAL4) で認証の評価を受けています。 Common Criteria の詳細については、以下の Common Criteria の Web サイトを参照してください。 http://niap.nist.gov/cc-scheme/

付録 D. 特記事項

本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒106-0032
東京都港区六本木 3-2-31
IBM World Trade Asia Corporation
Licensing

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で提供し、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すが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、 IBM より提供されます。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性がありますが、その測定値が、一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。 IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確証できません。 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にお願いします。

IBM の将来の方向または意向に関する記述については、予告なしに変更または撤回される場合があり、単に目標を示しているものです。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

著作権使用許諾:

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、利便性もしくは機能性があることをほのめかしたり、保証することはできません。

それぞれの複製物、サンプル・プログラムのいかなる部分、またはすべての派生した創作物には、次のように、著作権表示を入れていただく必要があります。

商標

以下は、IBM Corporation の商標です。

ACF/VTAM
AISPO
AIX
AIXwindows
AnyNet
APPN
AS/400
BookManager
C Set++
C/370
CICS
Database 2
DataHub
DataJoiner
DataPropagator
DataRefresher
DB2
DB2 Connect
DB2 Extenders
DB2 OLAP Server
DB2 Information Integrator
DB2 Query Patroller
DB2 Universal Database
Distributed Relational Database Architecture
DRDA
eServer
Extended Services
FFST
First Failure Support Technology
IBM
IMS
IMS/ESA
iSeries
LAN Distance
MVS
MVS/ESA
MVS/XA
Net.Data
NetView
OS/390
OS/400
PowerPC
pSeries
QBIC
QMF
RACF
RISC System/6000
RS/6000
S/370
SP
SQL/400
SQL/DS
System/370
System/390
SystemView
Tivoli
VisualAge
VM/ESA
VSE/ESA
VTAM
WebExplorer
WebSphere
WIN-OS/2
z/OS
zSeries

以下は、それぞれ各社の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。

Pentium は、Intel Corporation の米国およびその他の国における商標です。

Java およびすべての Java 関連の商標およびロゴは、Sun Microsystems, Inc. の米国およびその他の国における商標または登録商標です。

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]

[カ行]

[サ行]

[タ行]

[ハ行]

[マ行]

[ヤ行]

[ラ行]

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

[特殊文字]

IBM と連絡をとる

技術上の問題がある場合は、お客様サポートにご連絡ください。

製品情報

DB2 Universal Database 製品に関する情報は、
http://www.ibm.com/software/data/db2/udb から入手できます。

このサイトには、技術ライブラリー、資料の注文方法、製品のダウンロード、ニュースグループ、フィックスパック、ニュース、および Web リソースへのリンクに関する最新情報が掲載されています。

米国以外の国で IBM に連絡する方法については、 IBM Worldwide ページ (www.ibm.com/planetwide) にアクセスしてください。

Printed in Japan

SC88-9139-01

日本アイ・ビー・エム株式会社
〒106-8711　東京都港区六本木3-2-12